UNIT IV

MULTITHREADING AND GENERIC PROGRAMMING

Syllabus

Difference between Multi-threading and Multitasking, thread life cycle, creating threads, synchronizing threads, Inter-thread communication, daemon threads, thread groups, Generic Programming - Generic classes - generic methods – Bounded Types – Restrictions and Limitations

4.1 Multitasking



Executing several tasks simultaneously is called multi-tasking. Multitasking, in an operating system (OS), is the ability of performing more than one computer task at a time. The operating system is able to keep track of status of these tasks and switch from one to the other without losing information.

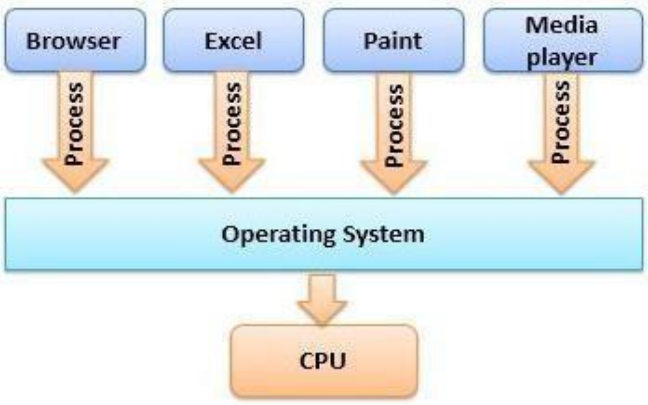
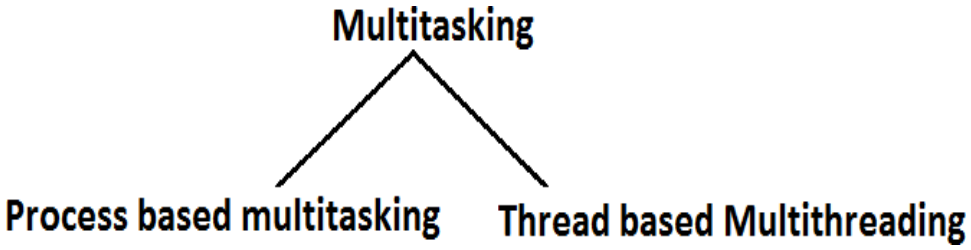
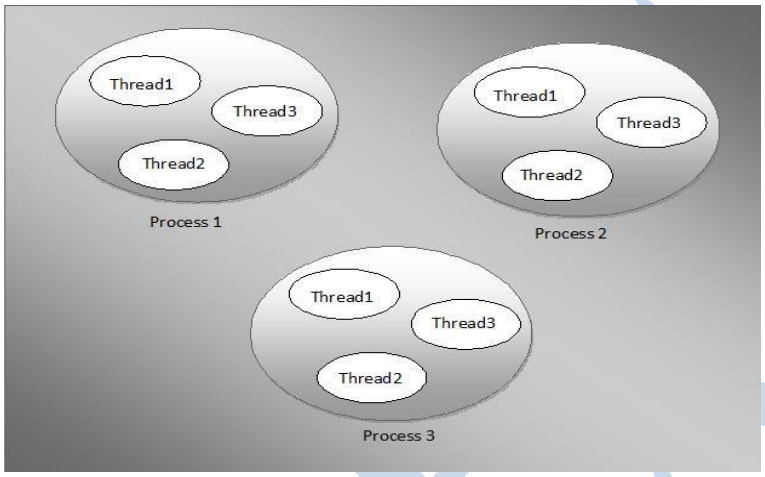


Figure 4.1 Multitasking

Multitasking can be achieved in two ways: Process-based and Thread-based.





Process-based Multitasking

Process-based multitasking is achieved by executing (or) running several programs (or) applications concurrently on a computer.

Example:

- Listen to music and browse internet at the same time.
- The processes in this example are the music player and browser.

Here every program needs different set of resources and memory space.

Thread-based Multitasking

Thread-based multitasking is achieved by executing different parts of the same program concurrently.

Example:

- Printing and formatting text in a word processor at the same time.
- MSWord does an auto correction using the red and green waves

Note:

In Windows the multi-threading can be observed by opening Windows Task manager. Select the Performance view, and click on the CPU box. Right click on the moving graph and select "change graph to logical processors", in order to see one moving graph per CPU core.

4.2 Multithreading

- A computer system has to perform several tasks simultaneously. A task can be known as a process or a program in execution.
- Creating processes for each task is not efficient due the resources used. To avoid that, a process can be divided into multiple sub-processes. One sub-process is a unit of the process. That unit is known as a thread. In multithreading, a process is divided into multiple threads and those threads are executing parallel at the same time.

Define Thread in Java: A thread, in the context of Java, is the path followed when executing a program. All Java programs have at least one thread, known as the main thread, which is created by the Java Virtual Machine (JVM) at the program's start.

Types of threads:

- Single thread
    - When there is one thread in a process, it's known as single threaded process
- Multi-thread
    - When multiple threads are running in the process, it's multithreaded process.
    - Multithreading is useful to run several tasks at the same time.

The below figure explains how a single process is split into different threads and executed at the same time as single and multi-threads.

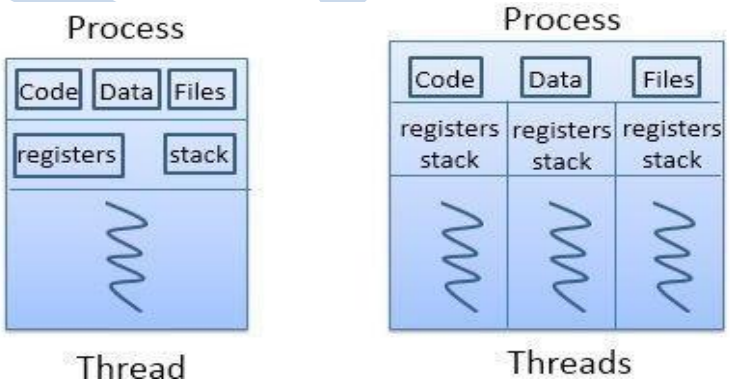
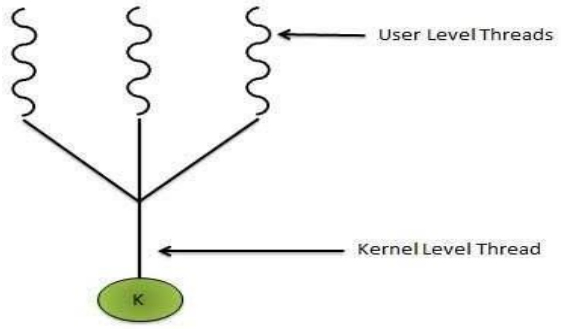


Figure 4.2 Multithreading

Threads are categorized into two levels: **User-level threads and Kernel-level threads**.

All thread operations are implemented in the kernel and the OS schedules all threads in the system.

- OS managed threads are called kernel-level threads
- User-Level threads are managed entirely by the user-level library



Difference between Multitasking and Multithreading:

Feature	MULTITASKING	MULTITHREADING
Concept	Allows the CPU to execute multiple tasks at the same time.	Allows the CPU to execute multiple threads of a process simultaneously.
Switching	The CPU switches between programs frequently.	The CPU switches between the threads frequently.
Memory & Resource	Separate memory and resources are allocated to each program.	Multiple threads shares the same memory and resources.
Generally	Referred to as a feature of OS	Referred to as the feature of process
Complexity	It is heavy-weight and hard to create.	It is light-weight and easy to create.

4.3 Threads in Java

A thread, in the context of Java, is the path followed when executing a program. All Java programs have at least one thread, known as the main thread, which is created by the Java Virtual Machine (JVM) at the program's start. When a Java program is executed; the execution starts from main() method and statements inside main() are executed sequentially one after the other. In a single threaded environment, at a time only one statement is executed and no parallel execution takes place. When a Java program starts, the JVM creates a thread -- the main thread and calls the program's main() method within that thread to execute statements contained inside body of main() method.

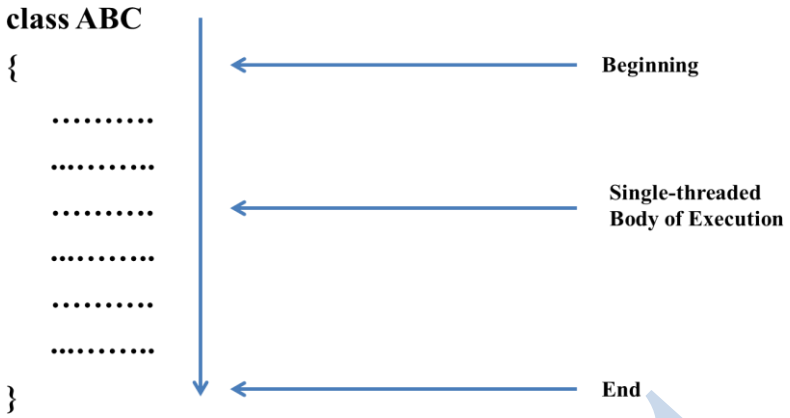


Figure 4.3 Structure of a single threaded program

Java is a multi-threaded programming language. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same. The below figure is the structure of a multithreaded program. The main() method is a thread and it starts the other threads.

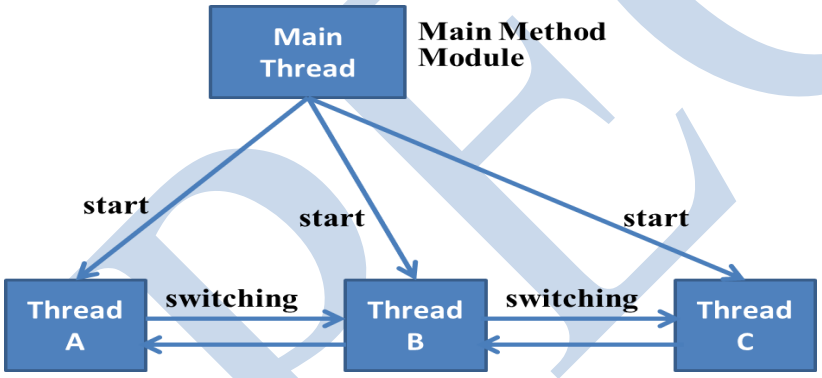


Figure 4.4 Structure of a Muthithreaded program

4.3.1 Thread Life Cycle

A thread goes through various stages in its life cycle. A thread can be in one of the five states. According to Sun, there are only 4 states in thread life cycle in java new, runnable, non-runnable and terminated. There is no running state. The life cycle of the thread in java is controlled by JVM.

In general the java thread states are as follows:

- New
- Runnable
- Running
- Non-Runnable (Blocked)
- Dead

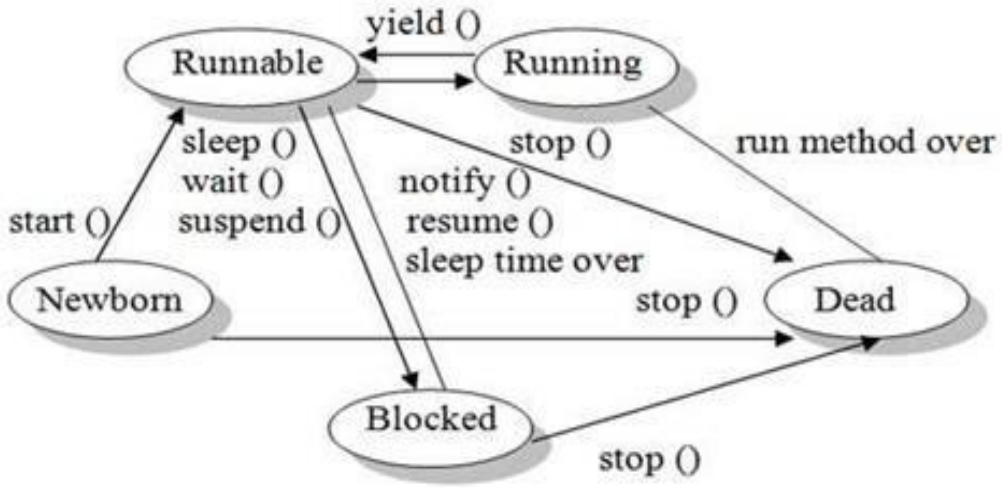


Figure 4.5 Life Cycle of a Thread

New – A thread begins its life cycle in the new state. When we create a thread object, the thread is born and is said to be in new-born state. The thread is not yet scheduled for running. It remains in this state until the program starts the thread. At this state, we can do only one of the following with it:

- Schedule it for running using start() method.
- Kill it using stop() method.

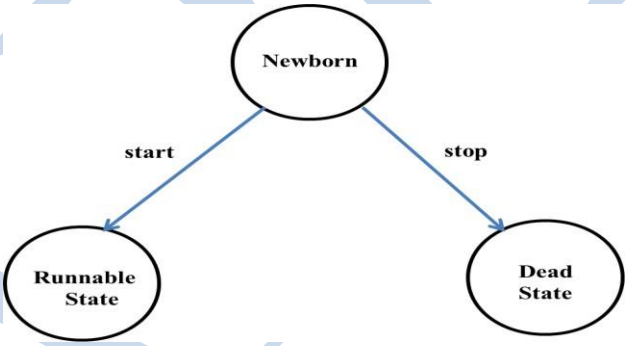


Figure 4.6 Scheduling a Newborn Thread

Runnable – After a newly born thread is started, the thread becomes runnable. In this state, thread is considered to be executing its task and is waiting for the availability of the processor ie. Thread has joined the queue of threads that are waiting for execution. If all threads have equal priority, then they are given time slots for execution in first-come, first-serve manner. This process of assigning time to threads is known as time-slicing. If we want a thread to give up control to another thread of equal priority before its turn comes, we can do so by using the yield() method.

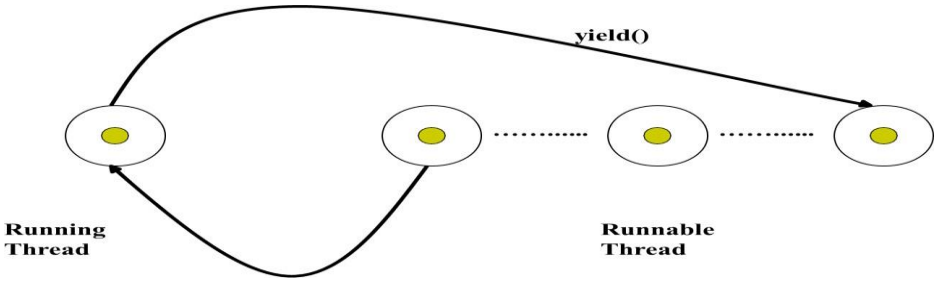
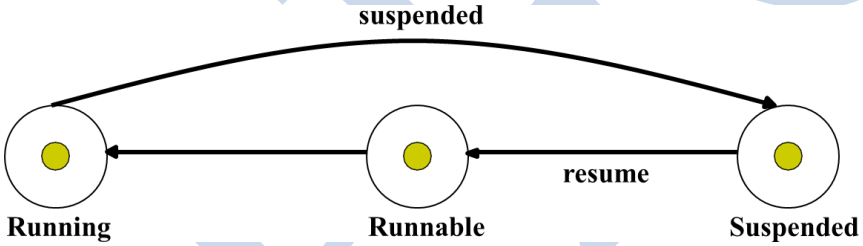


Figure 4.7 Releasing Control Using yield()

Running means that the processor has given its time to the thread for its execution. The thread runs until it give up control on its own or it is pre-empted by a higher priority thread. A running thread may give up its control in one of the following situations:

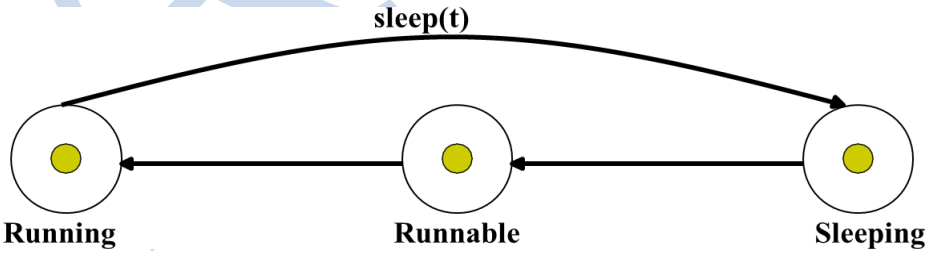
(i) Suspend() and resume() Methods:-

This approach is useful when we want to suspend a thread for some time due to certain reason, but do not want to kill it.

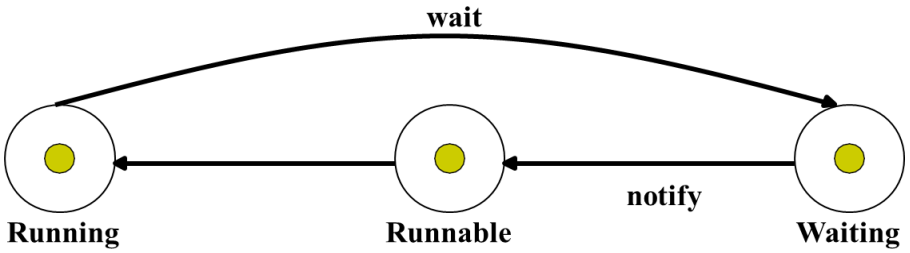


(ii) Sleep() Method :-

This means that the thread is out of the queue during this time period. The thread re-enter the runnable state as soon as this time period is elapsed.



(iii) Wait() and notify() methods :- blocked until certain condition occurs



Non-Runnable (Blocked) - A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements. A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.

Dead – A runnable thread enters the dead state when it completes its task or otherwise. A running thread ends its life when it has completed executing its run() method. It is a natural death. However, we can kill it by sending the stop message to it at any state thus causing a premature death to it. A thread can be killed as soon it is born, or while it is running, or even when it is in "not runnable" (blocked) condition.

4.3.2 Thread Creation

Thread class in the **java.lang package** allows us to create and manage threads. A new thread can be created in two ways:

- By extending a thread class
- By implementing an interface

By extending a thread class

The **java.lang.Thread** class can be extended to create our own java thread class. Thread class provide constructors and methods to create and perform operations on a thread. *Thread class* ***extends*** *Object class and* ***implements*** *Runnable interface.*

Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

Methods of Thread class	
Method	**Description**
void **run()**	Used to perform action for a thread.
void **start()**	Starts the execution of the thread. JVM calls the run() method on the thread.
void **sleep(long milisec)**	Causes the currently executing thread to sleep for the specified number of milliseconds.
void **join()**	Waits for a thread to die.

Methods of Thread class	
Method	**Description**
void **join(long miliseconds)**	Waits for a thread to die for the specified miliseconds.
int **getPriority()**	Returns the priority of the thread.
int **setPriority(int priority)**	Changes the priority of the thread.
String **getName()**	Returns the name of the thread.
void **setName(String name)**	Changes the name of the thread.
Thread **currentThread()**	Returns the reference of currently executing thread.
int **getId()**	Returns the id of the thread.
Thread.State **getState()**	Returns the state of the thread.
void **yield()**	Causes the currently executing thread object to temporarily pause and allow other threads to execute.
void **suspend()**	Used to suspend the thread.
void **resume()**	Used to resume the suspended thread.
void **stop()**	Used to stop the thread.

Program 4.1 Creation of Thread by extending Thread class

```
class MyThread extends Thread
{
   //Override run method
   public void run()
   {
      for (int i = 1; i < 3; i++)
            System.out.println("My thread is executing : " + i);
      }
}
```

→ **Defining a Thread**

→ **Job of the Thread**

```
public class Test
{
        public static void main(String[] args)
        {
                MyThread t = new MyThread();
                t.start();
                forr(int i = 1; i < 3; i++)
                        System.out.println("Main thread");

        }
}
```

→ **Instantiation of a Thread**

→ **starting of a Thread**

Output:

```
Main thread
Main thread
My thread is executing : 1
My thread is executing : 2
```

Creation of Multiple Threads by extending Thread class

```
class MyThread extends Thread
{
   //Override run method
   public void run()
   {
      for (int i = 1; i < 3; i++)
            System.out.println("My thread is executing : " + i);
       }
}

public class Test
{
        public static void main(String[] args)
        {
                MyThread t1 = new MyThread();
                t1.start();
                MyThread t2 = new MyThread();
                t2.start();
                MyThread t3 = new MyThread();
                t3.start();

                for(int i = 1; i < 3; i++)
                        System.out.println("Main thread");
        }
}
```

Note: Multiple threads t1,t2,t3 are created

Output:

```
Main thread
Main thread
My thread is executing : 1
My thread is executing : 2
My thread is executing : 1
My thread is executing : 2
My thread is executing : 1
My thread is executing : 2
```

Instead of start() method, run() method is used

```
class MyThread extends Thread
{
   //Override run method
   public void run()
   {
      for (int i = 1; i < 3; i++)
            System.out.println("My thread is executing : " + i);
       }
}


public class Test
{
        public static void main(String[] args)
        {
                MyThread t = new MyThread();
                t.run();

                for(int i = 1; i < 3; i++)
                        System.out.println("Main thread");
        }
}
```

Note: run() method, works like normal member function.

Output:

```
My thread is executing : 1
My thread is executing : 2
Main thread
Main thread
```

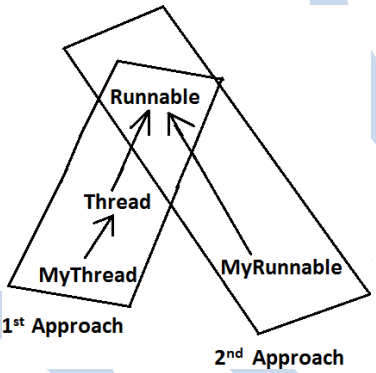
Difference between start() and run() method:

- t.start() → a new Thread is created which is responsible for the execution of run() method.
- t.run() → no new Thread will be created and run() method will be executed just like a normal method.

By implementing an interface

- Define a Thread even by implementing Runnable interface. Runnable interface present in **java.lang.pkg** and contains only one method **run().**

public void run(): is used to perform action for a thread.



Steps:

Step 1: Place the code into the run() method of a class that implements the Runnable interface.

```
class Sample implements Runnable
{
   //Override run method
   public void run()
   {
         //Task
    }
}
```

Step 2: Construct an object the class.

```
Runnable r = new MyRunnable();
```

Step 3: Construct a Thread object from the Runnable

```
Thread t = new Thread();
```

Step 4: Start the thread

```
t.start();
```

Program 4.2 Creation of Thread by implementing Runnable Interface

```
class MyRunnable implements Runnable
{
   //Override run method
   public void run()
   {
      for (int i = 1; i < 3; i++)
            System.out.println("My thread is executing : " + i);
      }
}
```

Defining a Runnable interface

Job of the Thread

```
public class Test
{
        public static void main(String[] args)
        {
                MyRunnable r = new MyRunnable();
                Thread t = new Thread(r);   → r is the target runnable
                t.start();
                forr(int i = 1; i < 3; i++)
                        System.out.println("Main thread");

        }
}
```

Output:

```
Main thread
Main thread
My thread is executing : 1
My thread is executing : 2
```

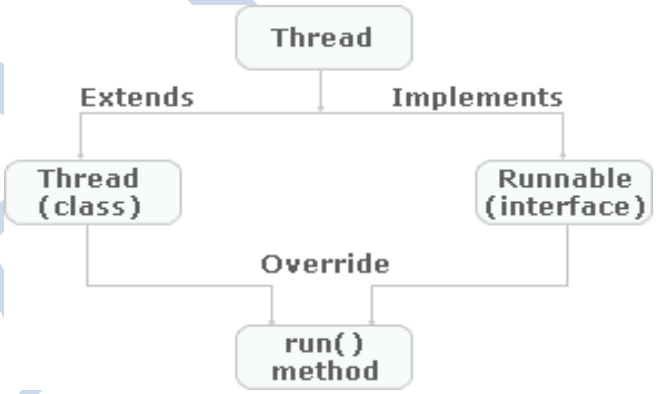


Figure 4.8 Comparison of Thread class and Runnable interface

However for Multi-threading we need two things:

- Something that can run inside a Thread (Runnable).
- Something that can start a new Thread (Thread).

BASIS FOR COMPARISON	THREAD	RUNNABLE
Basic	Each thread creates a unique object and gets associated with it.	Multiple threads share the same objects.
Memory	As each thread creates a unique object, more memory required.	As multiple threads share the same object, less memory is used.
Extending	As multiple inheritance is not allowed after a class extends Thread class, it cannot extend any further.	If a class define thread implementing the Runnable interface it has a chance of extending one class.
Use	A user must extend thread class only if it wants to override the other methods in Thread class.	If you only want to specialize run method then implementing Runnable is a better option.
Coupling	Introduces tight coupling	Introduces loose coupling

Threads can also be created by using an anonymous subclass of Thread. The below code snippet creates a thread an_thread without extending the Thread class or implementing the Runnable interface. This thread is called **anonymous thread.**

```
Thread an_thread = new Thread()
{
   public void run()
  {
     System.out.println("Thread Running");
   }
}
an_thread.start();
```

Thread Priorities

- 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.

Java program to explain priorities of threads

```
public class Test extends Thread
{
 public void run()
 {
   System.out.println("Running Thread name :"+Thread.currentThread().getName());
   System.out.println("Running Thread priority
:"+Thread.currentThread().getPriority());

  }

 public static void main(String args[])
 {
  Test m1 = new Test();
  Test m2 = new Test();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();
 }
}
```

OUTPUT:

```
Running thread name is:Thread-1
Running thread priority is:10
Running thread name is:Thread-0
Running thread priority is:1
```

4.3.3 Thread Synchronisation

When two or more threads access 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* .

Definition: The process of making only one thread access the object when two or more threads are accessing the same object is called Synchronization.

Java program to demonstrate NON-SYNCHRONIZATION

```
class Display
{
        public void wish(String name)
        {
                System.out.print("Good morning : ");
                try
                   {
                        Thread.sleep(1000);
                   }
                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);
        }
}
public class Test
{
        public static void main(String[] args)
        {
                Display d1=new Display();
                MyThread t1=new MyThread(d1,"MS Dhoni");
                MyThread t2=new MyThread(d1,"Yuvaraj Singh");
                t1.start();
                t2.start();
        }
}
```

OUTPUT:

```
Good morning : Good morning :
Yuvaraj Singh
MS Dhoni
```

But the expected output is:

Good morning : MS Dhoni

Good morning : Yuvaraj Singh

To prevent such problems, synchronization was introduced.

Two ways to achieve synchronization:

Method 1: **By providing locks**

- Code Synchronization helps in preventing threads working on the same code at the same time. The synchronization works based on the locking concept.
- Threads that work on the synchronized code should acquire the lock.
- Synchronization can only be applied for method or block of code. But not for classes or variables.

Syntax:

```
public synchronize void method(){ }
public static synchronize void method{…}
```

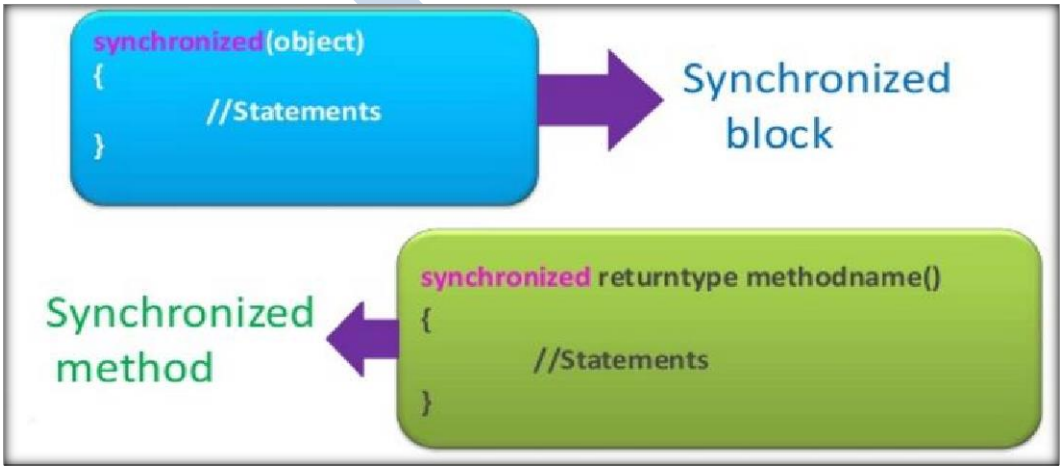
Method 2 : **By synchronized statement block**

- Synchronize access to common data is via a synchronized statement block.

Syntax:

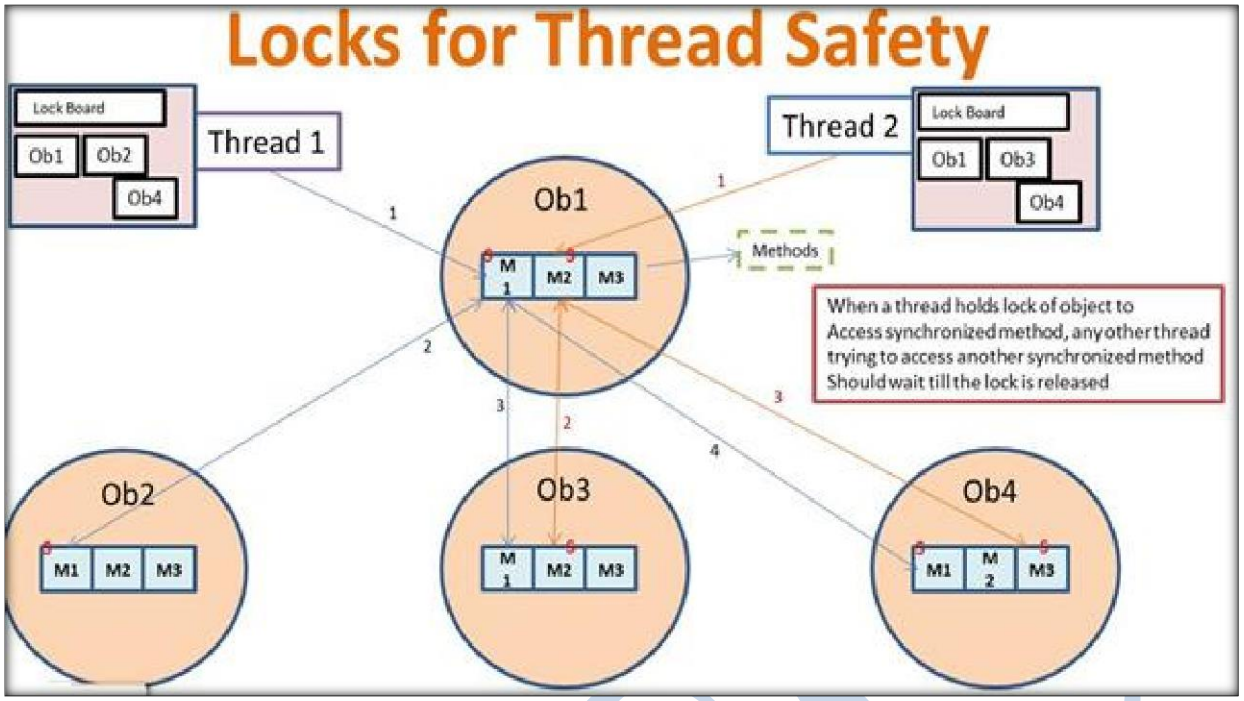
```
synchronized (obj)
{
    //statement block
  }
```

When two or more threads access a shared resource at the same time, it will lead to distorted results.



Synchronized Method

The synchronized keyword is used before the method that uses resources. When a thread invokes a synchronized method, an lock is acquired for that method's object. It is released when the method completes execution.



Assume there are four objects Obj1, Obj2, Obj 3, Obj4. Each has different methods and the methods marked with **s** are synchronized. Now consider we have two threads, Thread 1 and Thread 2 Both trying to access different methods on ObJ1 which in turn connect to different method of other objects. Consider that both threads are started together.

Thread 1 execution.When T1 come to M1 in Object 1, it sees that this method is synchronized and it acquires the lock of Obj1. From Obj1 M1 method, the calls are going to other methods and in this process; the thread acquires locks of different threads. The locks are given in the lock board of thread 1.

Thread 2 execution. Thread 2 trying to access M2 on Obj1 is waiting for the lock of Obj1 to be released by thread 1. Even though both are executing different methods, you should remember that because there is only one lock per object, there no other go for Thread 2 but to wait till lock is released by thread 1.

Java program to demonstrate Synchronisation using Synchronisation Method

```
class Display
{
        public synchronized void wish(String name)
        {
                System.out.print("Good morning : ");
                try
                   {
                        Thread.sleep(1000);
                   }
                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);
        }
}
public class Test
{
        public static void main(String[] args)
        {
                Display d1=new Display();
                MyThread t1=new MyThread(d1,"MS Dhoni");
                MyThread t2=new MyThread(d1,"Yuvaraj Singh");
                t1.start();
                t2.start();
        }
}
```

OUTPUT:

```
Good morning : MS Dhoni
Good morning : Yuvaraj Singh
```

Synchronized Statement or Synchronized blocks:

A block of code can be synchronized. Instead of locking the entire method, only the statements that are critical are locked. Synchronized statements enable the avoidance of unnecessary blocking. For example, suppose the method contains 100 lines of code but there are only 10 lines contain critical section of code. We need to synchronize only these 10 lines of code to avoid any modification in state of the Object.

Java program to demonstrate Synchronisation using Synchronized Block

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

  }
 }


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);    }
}

public class Test
{
   public static void main(String args[])
   {
      Table obj = new Table();
      MyThread1 t1 = new MyThread1(obj);
      MyThread2 t2 = new MyThread2(obj);
      t1.start();
      t2.start();
   }
}
```

→ **Synchronize Block**

→ **only one object**

OUTPUT:

```
5
10
15
20
25
100
200
300
400
500
```

APEC

4.3.4 Thread Communication

The process of communication between multiple threads is called Inter-thread communication. When one thread is executing the other thread will wait for the thread to be executed.

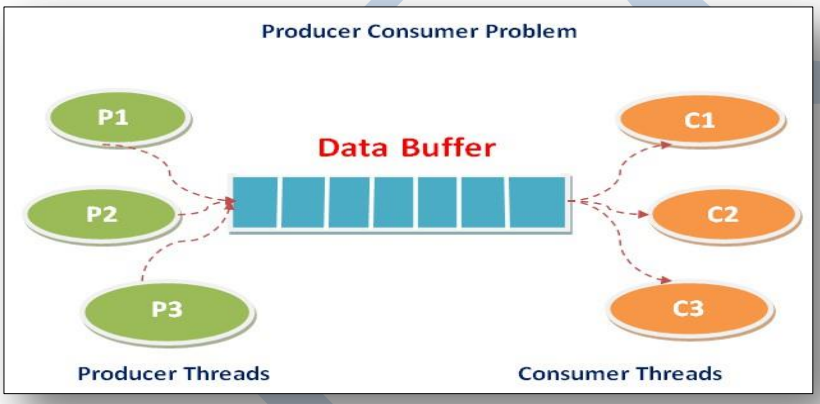
What is Polling and what are problems with it?

The process of testing a condition repeatedly till it becomes true is known as polling. Polling is usually implemented with the help of loops to check whether a particular condition is true or not. If it is true, certain action is taken. This may waste many CPU cycles and makes the implementation inefficient.

How Java multi-threading tackles this problem?

To avoid polling, Java uses three methods, namely, wait(), notify() and notifyAll().

Example: Producer – Consumer problem.



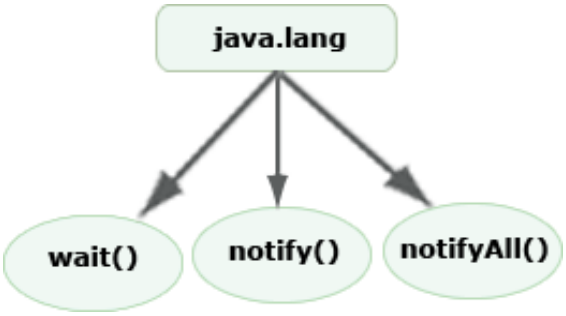
Producer side

- Check if Buffer is full or not. If full, then wait() for buffer items to get consumed.
- Generate data and put it into Buffer.
- Notify Consumer that data has been placed into Buffer.
- Repeat the above steps

Consumer side

- Check if Buffer has items. If empty, then wait() for buffer to get filled.
- Consume data from Buffer.
- Notify Producer that data has been consumed from Buffer
- Repeat the above steps

Methods for Thread Synchronization



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) specified time elapsed.

```
Syntax: public final void wait() throws InterruptedException
```

notify()

- Wakes up a single thread that is waiting on this object monitor. If many threads are waiting, then one of them is chosen to be awakened.

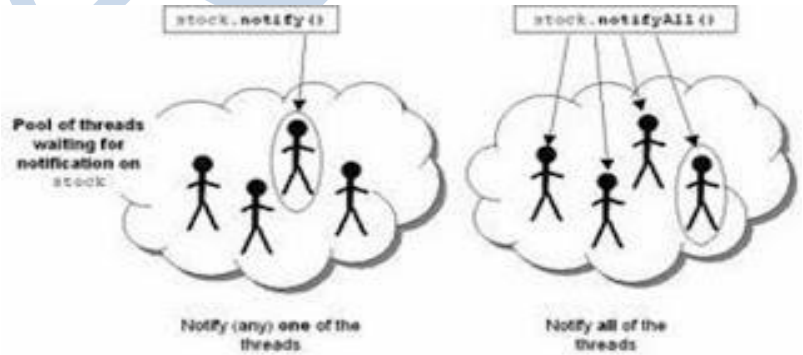
```
Syntax: public final void notify()
```

notifyAll()

- Wakes up all threads that are waiting on this object's monitor.

```
Syntax: public final void notifyAll()
```

Difference between notify() and notifyAll()



Note:

wait(), notify() and notifyAll() methods are available in Object class but not in Thread class because Thread can call these methods on any common object. To call wait(), notify() and notifyAll() methods compulsorily the current Thread should be owner of that object
i.e., current Thread should has lock of that object
i.e., current Thread should be in synchronized area.

Producer-Consumer solution using threads in Java

In computing, the producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

Problem

To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

Solution

Producer (producer Thread) will produce the items to the queue and consumer (consumer thread) will consume the items from the queue. If the queue is empty then consumer has to call wait() method on the queue object then it will entered into waiting state.

After producing the items producer Thread call notify() method on the queue to give notification so that consumer Thread will get that notification and consume items.

```
class Producer
{
    Producer()
    {
        synchronized(q)
        {
            produce items to the queue
                q.notify();
        }
    }
}
```

```
{
    Consumer()
    {
        synchronized(q)
        {
            if(q is empty)
            {
                q.wait();
            }
            else
                continue items;
        }
    }
}
```

Producer Consumer Program in Java using Threads – wait(), notify() & notifyAll()

```
import java.util.*;
import java.util.logging.Level;
import java.util.logging.Logger;


public class ProducerConsumerSolution
{

    public static void main(String args[])
    {
        Vector sharedQueue = new Vector();
        int size = 4;
        Thread prodThread = new Thread(new Producer(sharedQueue, size), "Producer");
        Thread consThread = new Thread(new Consumer(sharedQueue, size), "Consumer");
        prodThread.start();
        consThread.start();
    }
}

class Producer implements Runnable
{

    private final Vector sharedQueue;
    private final int SIZE;

    public Producer(Vector sharedQueue, int size)
    {
        this.sharedQueue = sharedQueue;
        this.SIZE = size;
    }

    @Override
    public void run()
    {
        for (int i = 0; i < 7; i++)
        {
            System.out.println("Produced: " + i);
            try
            {
                produce(i);
            }
           catch (InterruptedException ex)
            {
                Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, ex);
            }

        }
    }

private void produce(int i) throws InterruptedException
{
        //wait if queue is full
    while (sharedQueue.size() == SIZE)
    {
        synchronized (sharedQueue)
        {
```

```
            System.out.println("Queue is full " + Thread.currentThread().getName()
                    + " is waiting , size: " + sharedQueue.size());

            sharedQueue.wait();
        }
    }

    //producing element and notify consumers
    synchronized (sharedQueue)
    {
        sharedQueue.add(i);
        sharedQueue.notifyAll();
      }
    }
}

class Consumer implements Runnable
{

    private final Vector sharedQueue;
    private final int SIZE;

    public Consumer(Vector sharedQueue, int size)
    {
        this.sharedQueue = sharedQueue;
        this.SIZE = size;
    }

    @Override
    public void run()
    {
        while (true)
        {
            try
            {
                System.out.println("Consumed: " + consume());
                Thread.sleep(50);
            }

            catch (InterruptedException ex)
            {
                Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null, ex);
            }

        }
    }


private int consume() throws InterruptedException
{
        //wait if queue is empty
    while (sharedQueue.isEmpty())
    {
        synchronized (sharedQueue)
        {
            System.out.println("Queue is empty " + Thread.currentThread().getName()
                        + " is waiting , size: " + sharedQueue.size());

            sharedQueue.wait();
```

```
    }
}

//Otherwise consume element and notify waiting producer
synchronized (sharedQueue)
{
   sharedQueue.notifyAll();
   return (Integer) sharedQueue.remove(0);
   }
 }
}
```

Output:

```
Produced: 0
Produced: 1
Produced: 2
Produced: 3
Produced: 4
Queue is full Producer is waiting , size: 4
Consumed: 0
Produced: 5
Queue is full Producer is waiting , size: 4
Consumed: 1
Produced: 6
Queue is full Producer is waiting , size: 4
Consumed: 2
Consumed: 3
Consumed: 4
Consumed: 5
Consumed: 6
Queue is empty Consumer is waiting , size: 0
```

Daemon Threads:

When Java programs execute, there is always one thread running and that is the main thread. The two types of threads in Java

- User threads → created by the user
- Daemon threads →threads that work in the background providing service to other threads.
    - e.g. Garbage Collector Thread
    - Daemon threads are very low priority threads.
- When user thread exits, JVM checks to find out if any other thread is running. If the only executing threads are Daemon threads, it exits.

Methods of Daemon Threads	
Method Name	**Description**
void setDaemon(boolean)	Used to mark the current thread as daemon thread or user thread. Example: if I have a user thread T1 then T1.setDaemon(true) would make it Daemon thread. If I have a Daemon thread T2 then by calling T2.setDaemon(false) would make it user thread.
boolean isDaemon()	This method is used to check that current thread is daemon. Returns true if the thread is Daemon else returns false.

```
class Demo extends Thread
{
 public void run()
 {
   if (Thread.currentThread().isDaemon()) → Checking for Daemon Thread
       System.out.println("This is Daemon Thread");
   else
       System.out.println("This is User Thread");
   }
}

public class Test
{
 public static void main(String[] args)
 {
   Demo t1 = new Demo(); → Creating a Thread
   Demo t2 = new Demo();
   Demo t3 = new Demo();

   t1.setDaemon(true); → Now set, Thread t1 as Daemon Thread
   t1.start();//starting threads
   t2.start();
   t3.start();
  }
}
```

Output:

```
This is Daemon Thread
This is User Thread
This is User Thread
```

Java program to explain Daemon Threads

Note: If you make a user thread as Daemon Thread, it must not be started. If we start, it will throw IllegalThreadStateException.

```
class Demo extends Thread
{
 public void run()
 {
  System.out.println("Name: "+Thread.currentThread().getName());
  System.out.println("Daemon: "+Thread.currentThread().isDaemon());
 }
}

public class Test
{
 public static void main(String[] args)
 {
  Demo t1 = new Demo();
  Demo t2 = new Demo();
  t1.start();
  t1.setDaemon(true);   → Will throw exception here
  t2.start();
 }
}
```

Output:

```
Name: Thread-0
Daemon: false
Exception in thread "main" java.lang.IllegalThreadStateException
        at java.lang.Thread.setDaemon(Thread.java:1359)
        at Test.main(Test.java:17)
```

4.3.5 Thread Groups

In the complex environment, multiple threads exist in the system. Managing multiple threads is very complicated task in the thread management.

Grouping the Threads:

- To simplify thread management, threads in program are organized in group called **thread groups**
- $\boldsymbol{java.lang.ThreadGroup}$ class is used to manage threads
- Each thread is created under the system group. User created threads is under any of the thread group.
- System group organizes all the threads related to JVM which are useful for system related tasks.

The top-most thread group in a Java application is the thread group named main. The threads and thread groups can be create in the main group.

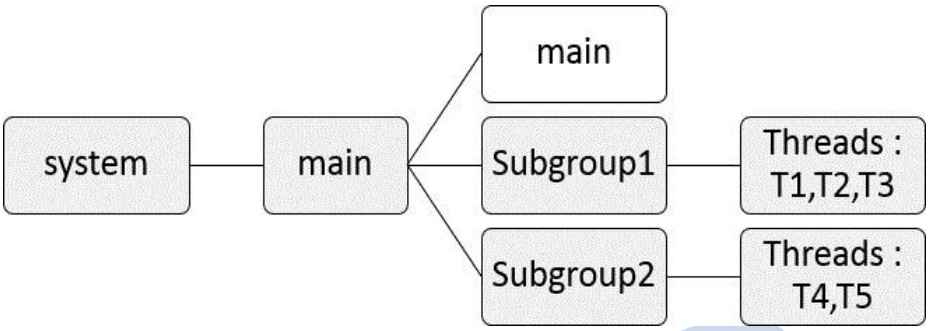
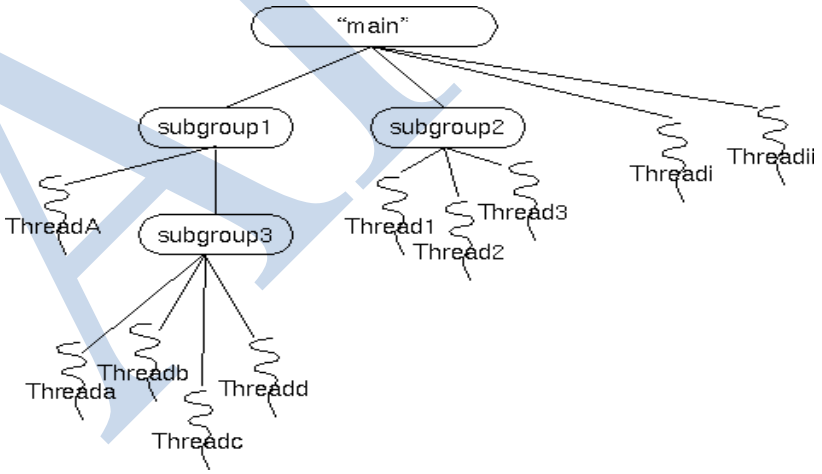


Figure 4.9: Thread groups

ThreadGroup is a convenient class that groups some related threads as a single unit and allows you to perform some operations on a group as a whole, rather than with each separate thread. For example, you can start or suspend all the threads within a group with a single method call. The main use of thread groups is to handle multiple threads simultaneously.

A thread group represents a set of threads. A thread group can also include other thread groups. The thread groups form a tree in which every thread group except the initial thread group has a parent. A thread is allowed to access information about its own thread group, but not to access information about its thread group's parent thread group or any other thread groups.



Constructors of Thread groups	
Constructor	**Description**
ThreadGroup(String name)	Creates a thread group with given name.
ThreadGroup(ThreadGroup parent, String name)	Creates a thread group with given parent group and name.

Methods in Thread Groups	
Method	**Description**
int activeCount()	Returns no. of threads running in current group.
int activeGroupCount()	Returns a no. of active group in this thread group.
void destroy()	Destroys this thread group and all its sub groups.
String getName()	Returns the name of this group.
ThreadGroup getParent()	Returns the parent of this group.
void interrupt()	Interrupts all threads of this group.
void list()	Prints information of this group to standard console.
int getMaxPriority()	Returns the maximum priority of the thread group
int getParent()	Returns the parent of the thread group
boolean isDaemon()	Tests if the thread group is a daemon thread group
boolean isDestroyed()	Tests if the thread group has been destroyed
void setDaemon(boolean)	Changes the daemon status of the thread group
void setMaxPriority(int)	Sets the maximum priority of the thread group
String toString()	Returns a string representation of the Thread group

Let's see a code to group multiple threads.

```
ThreadGroup tg = new ThreadGroup("Group A");
Thread t1 = new Thread(tg,new MyRunnable(),"one");
Thread t2 = new Thread(tg,new MyRunnable(),"two");
Thread t3 = new Thread(tg,new MyRunnable(),"three");
```

Now all 3 threads belong to one group. Here, tg is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names. Now we can interrupt all threads by a single line of code only.

```
Thread.currentThread().getThreadGroup().interrupt();
```

Java Program to implement Thread Groups

```
class Demo implements Runnable
{
   public void run()
   {
      System.out.println(Thread.currentThread().getName());
   }
}

public class Test
{
   public static void main(String[] args)
   {
      Demo r = new Demo();
      ThreadGroup tg = new ThreadGroup("Parent ThreadGroup");

      Thread t1 = new Thread(tg, r,"one");
      t1.start();
      Thread t2 = new Thread(tg, r,"two");
      t2.start();
      Thread t3 = new Thread(tg, r,"three");
      t3.start();

      System.out.println("Thread Group Name: "+tg.getName());
      tg.list();
   }
}
```

Output:

```
one
two
three
Thread Group Name: Parent ThreadGroup
java.lang.ThreadGroup[name=Parent ThreadGroup,maxpri=10]
    Thread[one,5,Parent ThreadGroup]
    Thread[two,5,Parent ThreadGroup]
    Thread[three,5,Parent ThreadGroup]
```

4.4 Generic Programming

Generic programming is a style of computer programming in which algorithms are written first and then specifying the data types.

Imagine a simple example to find maximum number in arrays of different types; int, float, double and long. The logic behind is the same. Instead of writing four methods, a single method can achieve the purpose. Java Generic methods and

generic classes enable programmers to specify, with a single method declaration or with a single class declaration.

Advantages of generics.

- **Type-safety** *:* Provide compile-time type safety that allows programmers to catch invalid types at compile time
- **Code Reuse***:* Single method/class/interface written once can be used for any type
- **Type casting is not required***:* There is no need to typecast the object.
- **Compile-Time Checking***:* It is checked at compile time so problem will not occur at runtime.

Java program to explain Generics concept

```
import java.util.*;
public class Test
{
   public static void main(String args[])
   {
      ArrayList<String> list = new ArrayList<String>();   → Array list is created
      list.add("II CSE");
      list.add("III CSE");                                 → Add elements in the List
      //list.add(32);  → Compile time error,  Through generics we get type safety.

      String s = list.get(1);   → Type casting is not required
      System.out.println("Elements : "+s);

      Iterator<String> itr = list.iterator();
      System.out.println("Given Elements : ");
      while(itr.hasNext())
         System.out.println(itr.next());
   }
}
```

Output:

```
Elements : III CSE
Given Elements :
II CSE
III CSE
```

4.4.1 Generic classes

A class that can refer to any type is known as generic class. Here, we are using **T type parameter** to create the generic class of specific type.

Syntax: $\boldsymbol{Classname} < \boldsymbol{T} > \boldsymbol{obj} = \boldsymbol{new\ Classname} < \boldsymbol{T} > ()$

Java program to explain Generics concept – Passing one parameter

```
class Gen<T>
{
 T obj;

  Gen(T obj)
  {
        this.obj = obj;
  }

  public void show()
  {
  System.out.println("Type of object :"+obj.getClass().getName());
  }

  public T get()
  {
      return obj;
  }
}

public class Test
{
   public static void main(String[] args)
   {
   Gen <Integer> g1 = new Gen <Integer> (10);
   g1.show();
   System.out.println(g1.get());

   Gen <String> g2 = new Gen <String> ("APEC");
   g2.show();
   System.out.println(g2.get());

   Gen <Double> g3 = new Gen <Double> (10.5);
   g3.show();
   System.out.println(g3.get());
   }
}
```

Constructor

Member Function 1

Member Function 2

Create an instance of generic class

Output:

```
Type of object :java.lang.Integer
10
Type of object :java.lang.String
APEC
Type of object :java.lang.Double
10.5
```

APEC

Explain Generics concept – Passing multiple type parameters

```
class Gen<T,U>
{
 T obj1;  ⟶ An object of type T
 U obj2;  ⟶ An object of type U
   Gen(T obj1, U obj2 )
   {
         this.obj1 = obj1;
         this.obj2 = obj2;
   }                                  ⟶ Constructor

   public void show()
   {
   System.out.println("Parameter 1 :"+ obj1.getClass().getName() + " "   + obj1);
   System.out.println("Parameter 2 :"+ obj2.getClass().getName() + "  " + obj2);
   }
}

public class Test
{
   public static void main(String[] args)
   {
   Gen <Integer,String> g1 = new Gen <Integer, String> (10, "APEC");
   g1.show();
   }
}
```

Output:

```
Type of object :java.lang.Integer
10
Type of object :java.lang.String
APEC
```

Formal Type Parameter Naming Convention

Use an uppercase single-character for formal type parameter. For example,

- **<E>** for an element of a collection;
- **<T>** for type;
- **<K, V>** for key and value.
- **<N>** for number
- **S, U, V,** etc. for 2nd , 3rd , 4th type parameters

Note: In Parameter type we cannot use primitives like - 'int','char' or 'double'.

4.4.2 Generic methods

Single generic method declaration can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

Rules to define Generic Methods:

- All generic method declarations have a type parameter section delimited by angle brackets < and >
- Each type parameter section contains one or more type parameters separated by commas.
- A generic method's body is declared like that of any other method.

Java program to explain Generics Method

```
public class Test
{
   public static < E > void printArray(E[] elements)
   {
      for ( E element : elements)
         System.out.println(element);
      System.out.println();
   }

   public static void main( String args[] )
   {
      Integer[] intArray = { 10, 20, 30, 40, 50 };
      Character[] charArray = { 'J', 'A', 'V', 'A'};

      System.out.println( "Printing Integer Array" );
      printArray( intArray  );

      System.out.println( "Printing Character Array" );
      printArray( charArray );
   }
}
```

Output:

```
Printing Integer Array
10
20
30
40
50

Printing Character Array
J
A
V
A
```

4.4.3 Bounded types with generics in Java

There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the ***extends*** keyword, followed by its upper bound.

Declare a bounded type parameter

- List the type parameter's name
- Along by the extends keyword
- And by its upper bound

Syntax: $< T \ extends \ superClassName >$

Java program to explain Bounded types

```
public class Test
{

public static <T extends Comparable<T>> T maximum(T x, T y, T z)
{
   T max = x; → Assume x is initially the largest

   if(y.compareTo(max) > 0)
   {
      max = y;                                  → Compare Y with X
   }

   if(z.compareTo(max) > 0)
   {
      max = z;   // z is the largest now        → Compare Z with Y
   }

   return max;   →Returns the largest object
}

public static void main(String args[])
{
   System.out.println("Max of 3,4,5 :" + maximum( 3,4,5 ));
   System.out.println("Max of 6.6,8.8,7.7 : " + maximum( 6.6,8.8,7.7 ));
   System.out.println("Max of pear,apple,orange : "
                                          +maximum("pear","apple","orange"));
  }
}
```

Note: **public static <T extends Comparable<T>>**
Determines the largest of three Comparable objects

Output:

```
Max of 3,4,5 :5
Max of 6.6,8.8,7.7 : 8.8
Max of pear,apple,orange : pear
```

Java program to no of times a particular element is present in the array using Bounded types

```
public class Test
{

   public static <T extends Comparable<T>> int find(T[] elements, T elem)
   {
      int count = 0;
      for (T e: elements)
         if (e.compareTo(elem) == 0)
            ++count;
      return count;
   }

   public static void main(String args[])
   {
      Double[] d = {1.2, 3.4, 0.0, 4.5, 4.5, 0.0};
      Integer[] i = {0, 0, 0, 0, 1, 3, 5};
      String[] s = {"hi", "bye", "oop", "java", "jav", "java"};

      System.out.println("No of 4.5 in Double Array = " + find(d, 4.5));
      System.out.println("No of zeros in Integer Array = " + find(i, 0));
      System.out.println("No of java in String Array = " + find(s, "java"));
   }
}
```

Output:

```
No of 4.5 in Double Array = 2
No of zeros in Integer Array = 4
No of java in String Array = 2
```

4.4.4 Restrictions and Limitations

(i) No Instance:

A type parameter cannot be used to instantiate its object inside a method.

Example:

```
class Test <T>
{
   T item = new T();  → Cannot instantiate the type T
}
```

(ii) No Primitive Types

The primitive types cannot be passed as type parameters. Instead of the primitive type int , the Integer object must be passed.

(iii) No Static field

The type parameters are not allowed to be static.

(iv) No Cast

Casting to a parameterized type is not allowed.

(v) No instanceOf

Runtime difference between Box<Integer> and Box<String> cannot be verified using instanceOf operator.

(vi) No Exceptions

A generic class is not allowed to extend the Throwable class directly or indirectly.

//The generic class Box<T> may not be the subclass of java.lang.Throwable

$$\boldsymbol{class\ Box} < \boldsymbol{T} > \boldsymbol{extends\ Exception}\ \{\}$$

//The generic class Box<T> may not be the subclass of java.lang.Throwable

$$\boldsymbol{class\ Box1} < \boldsymbol{T} > \boldsymbol{extends\ Throwable}\ \{\}$$

(vii) No Overload

A class is not allowed to have two overloaded methods.

Example:

```
class Box
{
   public void print(List<String> stringList) { }
   public void print(List<Integer> integerList) { }
}
```

(viii) No Array

Arrays of parameterized types are not allowed.

//Cannot create a generic array of Box<Integer>

Box<Integer>[] arrayOfLists = new Box<Integer>[2];

Because compiler uses type erasure, the type parameter is replaced with Object and user can add any type of object to the array. And at runtime, code will not able to throw ArrayStoreException.

2 Mark Questions:

1. Define Thread (P. No. 161)
2. List the states of Thread. (P. No. 163)
3. List the uses of MultiThreading. (P. No. 163)
4. How do you implement Thread in Java? ((P. No. 167)
5. What is the difference between start() and run() method of Thread class? (P. No. 168)
6. Draw the Life Cycle diagram of Thread (P. No. 163 – P. No. 166)
7. Differentiate between MultiThreading and MultiTasking. (P. No. 162)
8. List the methods in Thread class (P. No. 166 – P. No. 167)
9. Differentiate between Runnable interface and Thread Class. (P. No. 170 - 171)
10. What is a Daemon Thread? (P. No. 183)
11. List the uses of ThreadGroup. P. No. 185 - 186)
12. Why do we need synchronization of threads? (P. No. 171)
13. Differentiate notify and notifyall methods. (P. No. 179)
14. List the different ways to achieve synchronisation. (P. No. 174)
15. Differentiate Sleep and Wait methods. (P. No. 196)
16. What is meant by Generic Programming? (P. No. 188)
17. Define a Generic Class. (P. No. 190)
18. Define a Generic Method. (P. No. 192)
19. List the limitations of Generic Class. (P. No. 194)
20. How can the parameters in Generic Methods be bounded? (P. No. 193)

13 Mark Question:

21. Illustrate the concept and challenges of Multithreading. Write a java program to develop producer – Consumer problem.
22. Explain the concept and thread and its types. How a thread is created in Java. Mention the properties and states of Thread.
23. Discuss briefly the motivation of Generic programming
24. Explain the concept of thread synchronization in detail.

Difference between wait() and sleep()

wait()	**sleep()**
Used in the synchronised block	Can be used anywhere
Monitor is released	monitor is not released
The thread is awaken when notify() or notifyAll() method is called.	The thread does not get awake when notify() or notifyAll() method is called
It is not a static method	It is a static method
Generally used on condition	Used to put thread on sleep.