UNIT-4 (Multithreading and Generic Programming)

Multitasking :

The process of executing one or more tasks concurrently or at the same time is called **multitasking**. It is the ability of an operating system to execute multiple tasks at once. The main purpose of multitasking is to use the idle time of CPU.

Multitasking can be implemented in two ways:

1. Process-based multitasking (Multiprocessing)
2. Thread-based multitasking (Multithreading)

Process-based Multitasking (Multiprocessing)

The process of executing multiple programs or processes at the same time (concurrently) is called process-based multitasking or program-based multitasking. In process-based multitasking, several programs are executed at the same time by the microprocessor.

Therefore, it is also called multiprocessing in Java. It is a heavyweight. A process-based multitasking feature allows to execute two or more programs concurrently on the computer.

A good example is, running spreadsheet program while also working with word-processor.

Each program (process) has its own address space in the memory. In other words, each program is allocated in a separate memory area.

The operating system requires some CPU time to switch from one program to another program. The switching of CPU among programs is called context switching.

The switching from one program to another program is so fast that it appears to the user that multiple tasks are being done at the same time.

Thread-based Multitasking (Multithreading)

A thread is a separate path of execution of code for each task within a program. In thread-based multitasking, a program uses multiple threads to perform one or more tasks at the same time by a processor.

That is, thread-based multitasking feature allows you to execute several parts of the same program at once. Each thread has a different path of execution.

Realtime Example of Multithreading in Java

Let's take different examples of thread-based multithreading in Java.

1. A very good example of thread-based multithreading is a word processing program that checks the spelling of words in a document while writing the document. This is possible only if each action is performed by a separate thread.

2. Another familiar example is a browser that starts rendering a web page while it is still downloading the rest of page.

Consider a program as shown in the below figure. The program is divided into two parts. These parts may represent two separate blocks of code or two separate methods that can perform two different tasks of the same program.

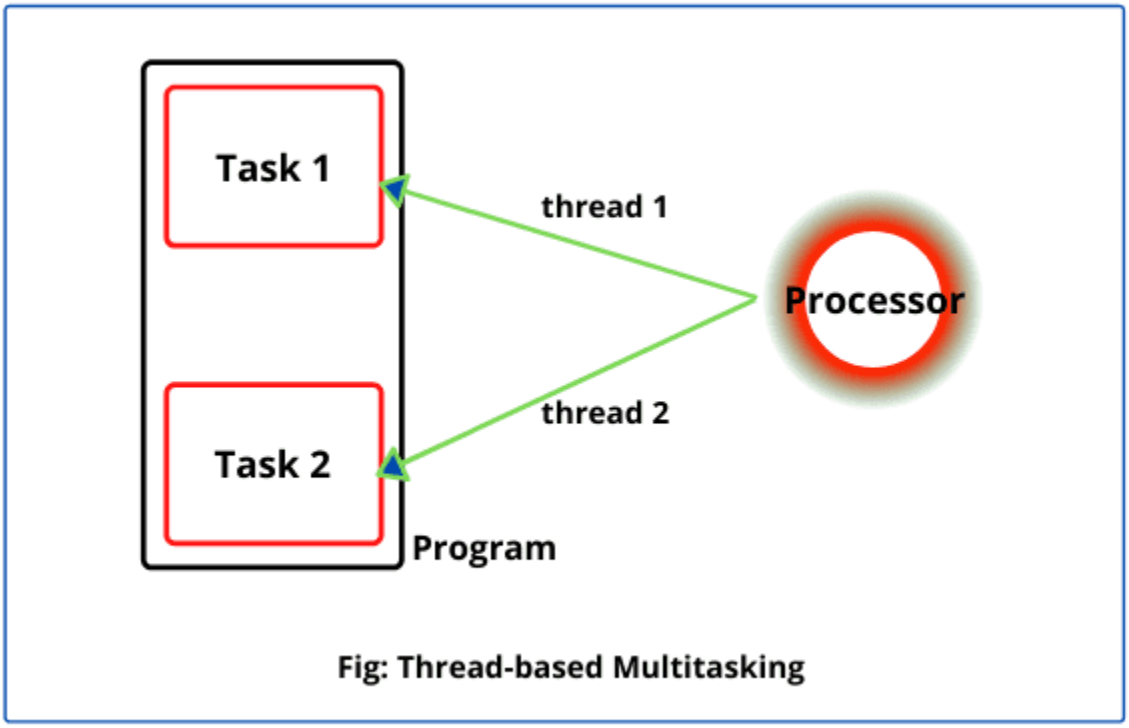


Fig: Thread-based Multitasking

Hence, a processor will create two separate threads to execute these two parts simultaneously. Each thread acts as an individual process that will execute a separate block of code.

Thus, a processor has two threads that will perform two different tasks at a time. This multitasking is called thread-based multiple tasking.

Advantage of Thread-based Multitasking over Process-Thread Multitasking

The main advantages of thread-based multitasking as compared to process-based tasking are

1. Threads share the same memory address space.

2. Context switching from one thread to another thread is less expensive than between processes.

3. The cost of communication between threads is relatively low.

4. Threads are lightweight as compared to processes (heavyweight). They utilize the minimum resources of the system. They take less memory and less processor time.

Java supports thread-based multitasking and provides a high quality of facilities for multithreading programming.

A great way to remember the difference between process-based multitasking and thread-based multitasking is process-based multitasking works with multiple programs or processes whereas thread-based multitasking works with parts of one program.

In process-based multitasking, a program is the smallest unit of executable code whereas, in thread-based multitasking, thread is the smallest unit of executable code.

Multithreading :

Multithreading means multiple threads of execution concurrently. The process of executing multiple

threads simultaneously (concurrently) is called **multithreading in Java**.

In other words, multithreading is a technique or programming concept in which a program (process) is divided into two or more subprograms (subprocesses), each of which can perform different tasks simultaneously (at the same time and in parallel manner). Each subprogram of a program is called thread in Java.

Look at the below figure where a Java program has three threads, one main and two others. The other two threads ThreadA and ThreadB are created and started from the main thread. When a program contains multiple flows of control, it is called multithreaded program. Once initiated by the main thread, thread ThreadA and ThreadB run simultaneously and share the resources together. It is the same as people living in a joint family and sharing certain resources among all of them. Once initiated by the main thread, thread ThreadA and ThreadB run simultaneously and share the resources together. It is the same as people living in a joint family and sharing certain resources among all of them.

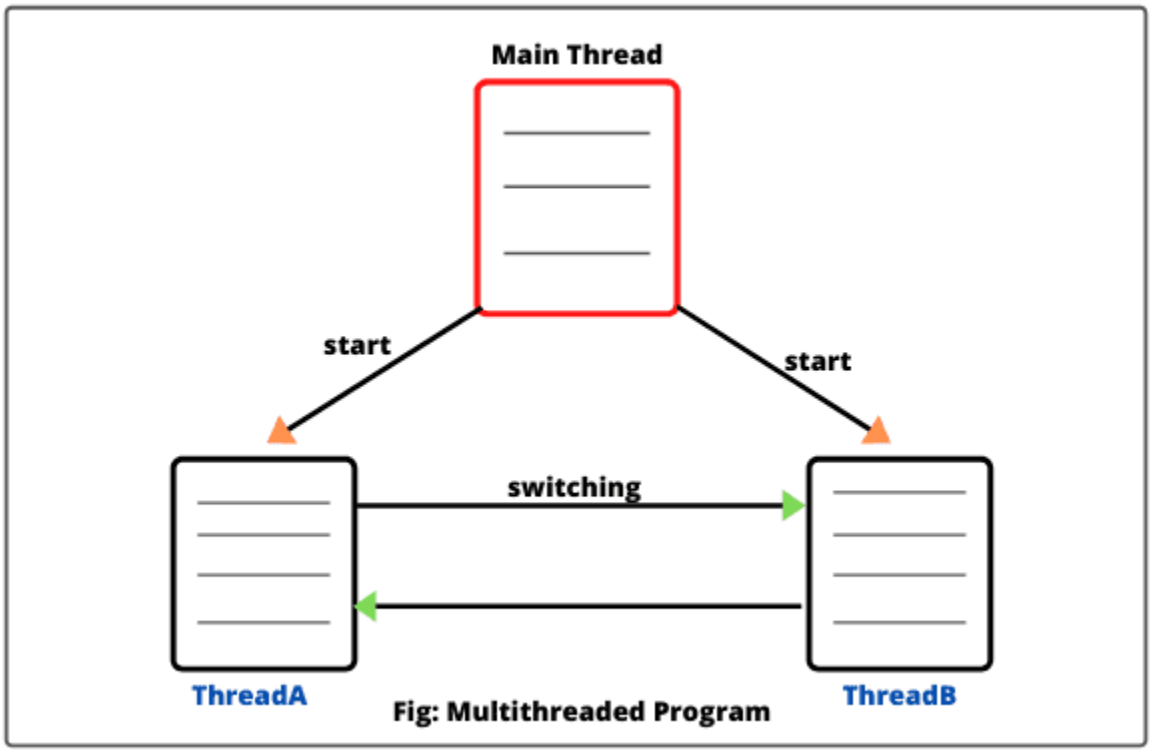


Fig: Multithreaded Program

Advantage of Multithreading :

The advantages of using multithreading programming concept are as follows:

1. In a multithreaded application program, different parts of the application are executed by different threads. The entire application does not stop even if an exception occurs in any of the threads. It does not affect other threads during the execution of the application.

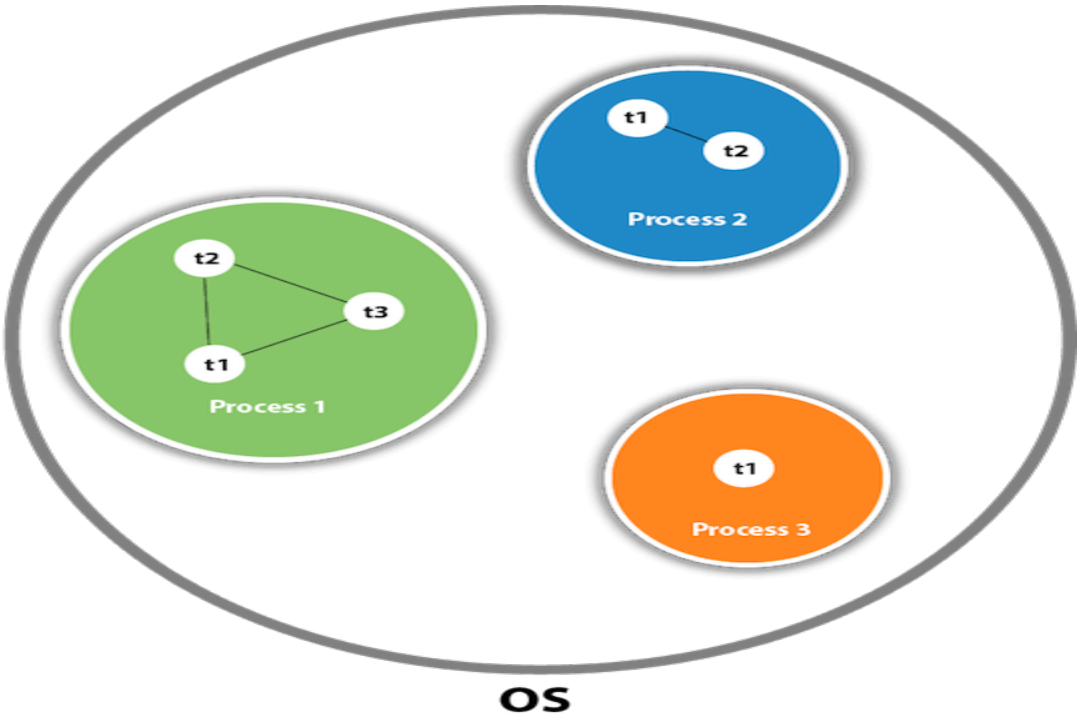
2. Different threads are allotted to different processors and each thread is executed in different processors in parallel.

3. Multithreading helps to reduce computation time.
4. Multithreading technique improves the performance of the application.
5. Threads share the same memory address space. Hence, it saves memory.
6. Multithreaded program makes maximum utilization of CPU and keeping the idle time of CPU to minimum.
7. Context switching from one thread to another thread is less expensive than between processes.

Thread :

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

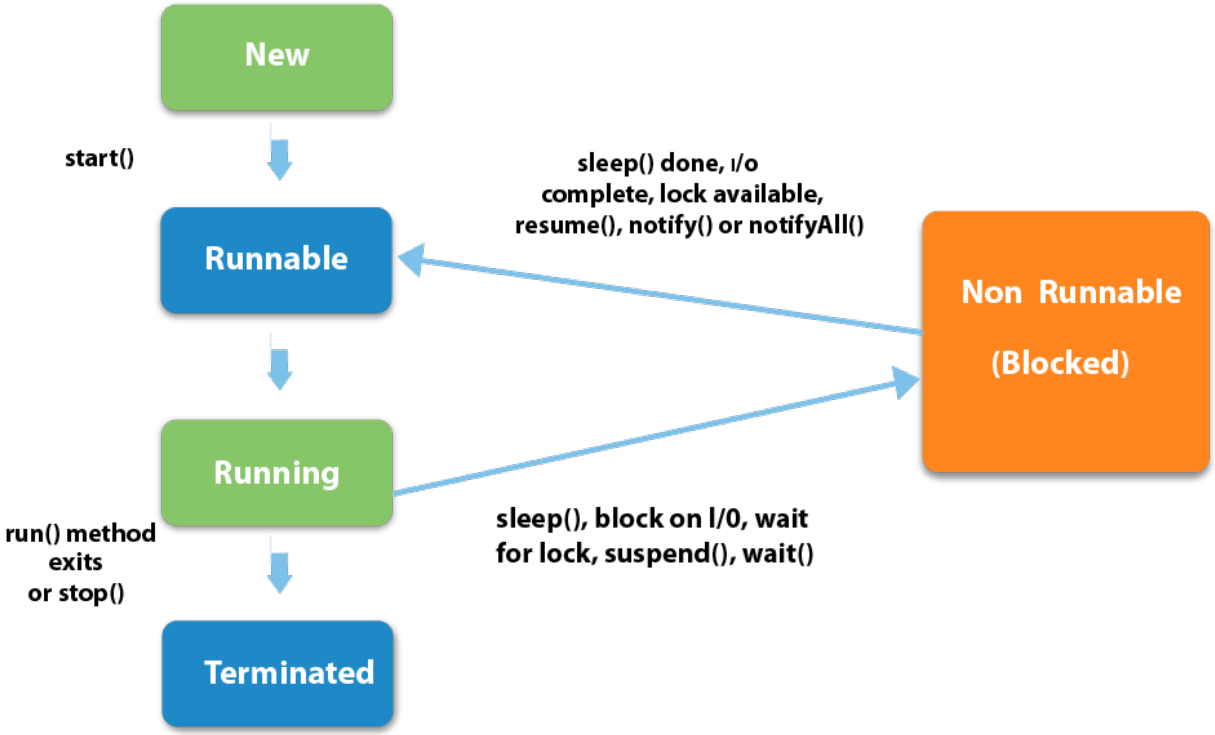
Note: At a time one thread is executed only.

Life cycle of a Thread:

A thread can be in one of the five states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

How to create thread

There are two ways to create a thread:

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

1.By extending Thread class:

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.

Commonly used Constructors of Thread class:

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

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

Thread Example by extending Thread class

```
class MultiEx extends Thread{
public void run(){
System.out.println("thread is running by using Thread class...");
}
public static void main(String args[]){
Multi Ex t1=new MultiEx();
t1.start();
 }
}
```

Output:thread is running by using Thread class...

2.By implementing Runnable interface:

Runnable Interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

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

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

Thread Example by implementing Runnable interface

```
class MultiRun implements Runnable{
public void run(){
System.out.println("thread is running using Runnable interface...");
}
  public static void main(String args[]){
MultRun m1=new MultiRun();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Output:thread is running using Runnable interface...

If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitly create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.

Thread Scheduler

Thread scheduler in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Difference between preemptive scheduling and time slicing

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

Sleep method

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Syntax of sleep() method in java

The Thread class provides two methods for sleeping a thread:

- public static void sleep(long miliseconds)throws InterruptedException
- public static void sleep(long miliseconds, int nanos)throws InterruptedException

Example of sleep method in java

```
class TestSleepMethod1 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
    System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();

  t1.start();
  t2.start();
 }
}
```

The join() method

This method is used to wait the current thread until the thread that has called the join() method completes its execution.Syntax:

public void join()throws InterruptedException

public void join(long milliseconds)throws
InterruptedException

Example of join() method

```
class TestJoin  extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }
 }
public static void main(String args[]){
 TestJoin    t1=new TestJoin();
 TestJoin   t2=new TestJoin();
 TestJoin  t3=new TestJoin();
 t1.start();
 try{
  t1.join();
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```

in the above example,when t1 completes its task then t2 and t3 starts executing.

Example of join(long miliseconds) method

```
class TestJoinEx extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }
 }
public static void main(String args[]){
 TestJoinEx  t1=new TestJoinEx();
 TestJoinEx  t2=new TestJoinEx();
 TestJoinEx  t3=new TestJoinEx();
 t1.start();
 try{
  t1.join(1500);
 }catch(Exception e){System.out.println(e);}
 t2.start();
 t3.start();
 }
}
```

In the above example,when t1 is completes its task for 1500 miliseconds(3 times) then t2 and t3 starts executing.

getName(), setName(String) and getId() method of Thread class::

public String getName(): returns the name of the thread.
public void setName(String name): changes the name of the thread.
public int getId(): returns the id of the thread.

Example:

```
class GetName extends Thread{
  public void run(){
   System.out.println("running...");
  }
 public static void main(String args[]){
  GetName  t1=new GetName();
  GetName  t2=new GetName();
  System.out.println("Name of t1:"+t1.getName());
  System.out.println("Name of t2:"+t2.getName());
  System.out.println("id of t1:"+t1.getId());
  t1.start();
  t2.start();
  t1.setName("Hello");
  System.out.println("After changing name of t1:"+t1.getName());
 }
}
```

The currentThread() method:

The currentThread() method returns a reference to the currently executing thread object.

Syntax:

```
public static Thread
currentThread()
```

Example of currentThread() method

```
class TestCurrent extends Thread{
 public void run(){
  System.out.println(Thread.currentThread().getName());
 }
 }
 public static void main(String args[]){
  TestCurrent  t1=new TestCurrent();
  TestCurrent  t2=new TestCurrent();
  t1.start();
  t2.start();
 }
}
```

Priority of a Thread (Thread Priority):

Each thread have 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:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. 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.

How to get Priority of Current Thread in Java?

Thread class provides a method named getPriority() that is used to determine the priority of a thread. It returns the priority of a thread through which it is called.

Example of priority of a Thread:

```
class TestMultiPriority  extends Thread{
 public void run(){
   System.out.println("running thread name is:"+Thread.currentThread().getName());
   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());

  }
 public static void main(String args[]){
  TestMultiPriority  m1=new TestMultiPriority1();
  TestMultiPriority  m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();

 }
}
Output:running thread name is:Thread-0
      running thread priority is:10
      running thread name is:Thread-1
      running thread priority is:1
```

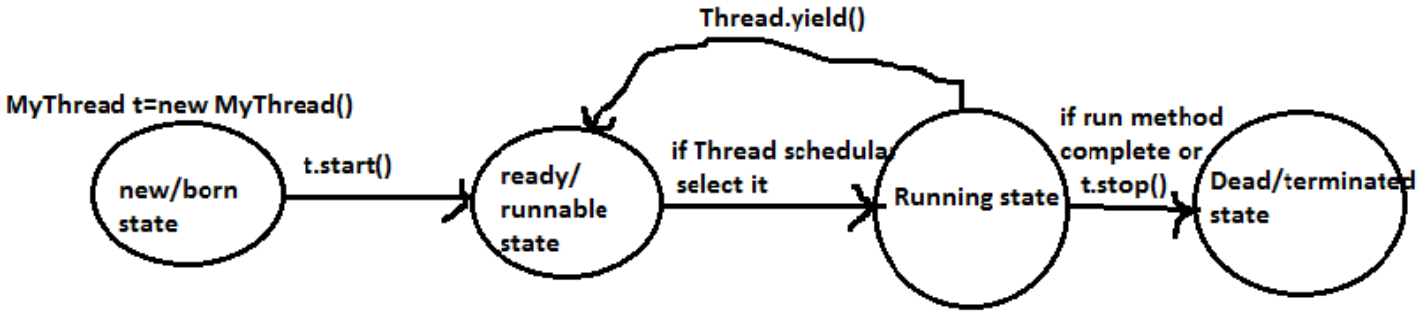
public int setPriority(int priority): changes the priority of the thread.

```
example:
  public class A implements Runnable
{
public void run()
{
  System.out.println(Thread.currentThread()); // This method is static.
}
public static void main(String[] args)
{
 A a = new A();
 Thread t1 = new Thread(a, "First Thread");
 Thread t2 = new Thread(a, "Second Thread");
 Thread t3 = new Thread(a, "Third Thread");

 t1.setPriority(4); // Setting the priority of first thread.
 t2.setPriority(2); // Setting the priority of second thread.
 t3.setPriority(8); // Setting the priority of third thread.

 t1.start();
 t2.start();
 t3.start();
  }
}
Output:
      Thread[Third Thread,8,main]
      Thread[First Thread,4,main]
      Thread[Second Thread,2,main]
```

yield(): yield method causes to pause current executing thread to give the chance for waiting thread of same priority.

=>if there are no waiting thread or all waiting thread has low priority then same thread can continue its execution.
=>If multiple thread are waiting with the same priority then which thread get the chance we cannot expect it depends on Thread Schedular.
=>The thread which is yield() when it will get the chance once again it depends on thread Scheduler.

ssy



syntax:
Public static void yield(){}

Example:

```
class MyThread extends Thread{
public void run(){
for(int i=0;i<100;i++){
System.out.println ("child thread");
Thread.yield();

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

Synchronization :

=>Normally, multiple threads in a single program run asynchronously if threads do not share a common resource.

=>Under some circumstances, it is possible that more than one thread access the common resource (shared resource). That is, more than one thread access the same object of a class.

=>In this case, care must be taken by the programmer that only one thread use shared resource at a time. The technique through which the solution to this problem is achieved is called synchronization

=>Synchronization is the capability *to control the access of multiple threads to any shared resource*.

=>If multiple Threads are trying to operate simultaneously on same java object then there may be a chance of data inconsistency problem.
=>To overcome this problem we should go for synchronized keyword. If a method or block is declared as synchronized then at a time only one thread is allowed to execute that method or block on the given object. So the data inconsistency problem will be resolved.

=>synchronized is the modifier applicable only for method or block but not for classes and variables.

Advantages:

The main advantage of synchronized keyword is we can resolve data inconsistency problems.

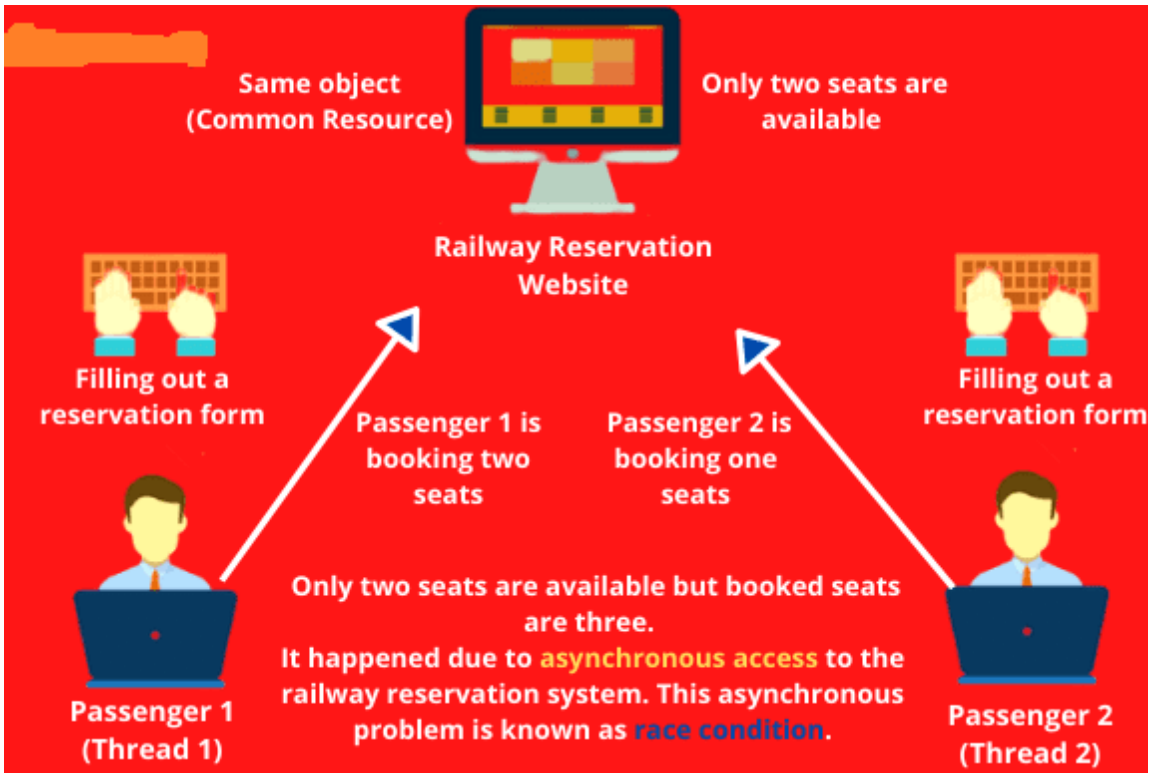
Disadvantages:

The main disadvantage is it increases the waiting time of threads and creates performance problems. Hence if there is no specific requirements then it is not recommended to use synchronized keyword.

Example of Synchronization :

1. Suppose a thread in a program is reading a record from a file while another thread is still writing the same file. In this situation, the program may produce undesirable output.

2. Let's take a scenario of railway reservation system where two passengers are trying to book seats from Dhanbad to Delhi in Rajdhani Express. Both passengers are trying to book tickets from different locations.



Now suppose that both passengers start their reservation process at 11 am and observe that only two seats are available. First passenger books two seats and simultaneously the second passenger books one seat.

Since the available number of seats is only two but booked seats are three. This problem happened due to asynchronous access to the railway reservation system.

In this realtime scenario, both passengers can be considered as threads, and the reservation system can be considered as a single object, which is modified by these two threads asynchronously.

This asynchronous problem is known as **race condition** in which multiple threads access the same object and modify the state of object inconsistently.

The solution to this problem can be solved by a synchronization mechanism in which when one thread is accessing the state of object, another thread will wait to access the same object at a time until their come turn.

Object Lock in Java:

The code in Java program can be synchronized with the help of a lock. A lock has two operations: acquire and release. The process of acquiring and releasing object lock (monitor) of an object is handled by Java runtime system (JVM) internally and programmer is not responsible for this activity.

In Java programming language, every object has a default object lock that can be used to lock on a thread. This object lock is also known as **monitor** that allows only one thread to use the shared resources (objects) at a time.

To acquire an object lock on a thread, we call a method or a block with the synchronized keyword. Before entering a synchronized method or a block, a thread acquires an object lock of the object.i.e once thread got the lock then it is allowed to execute any synchronized method on that object.

On exiting synchronized method or block, thread releases the object' monitor lock. i.e once method execution completes automatically thread releases the lock. A thread can again acquire the object' monitor lock as many times as it wants.

Object lock is like a room with only one door. A person enters the room and locks the door from inside. The second person who wants to enter the room will wait until the first person come out.

Similarly, a thread also locks the object after entering it, the next thread cannot enter it until the first thread comes out. Since the object is locked mutually on threads, therefore, this object is called **mutex** (mutually exclusive lock).

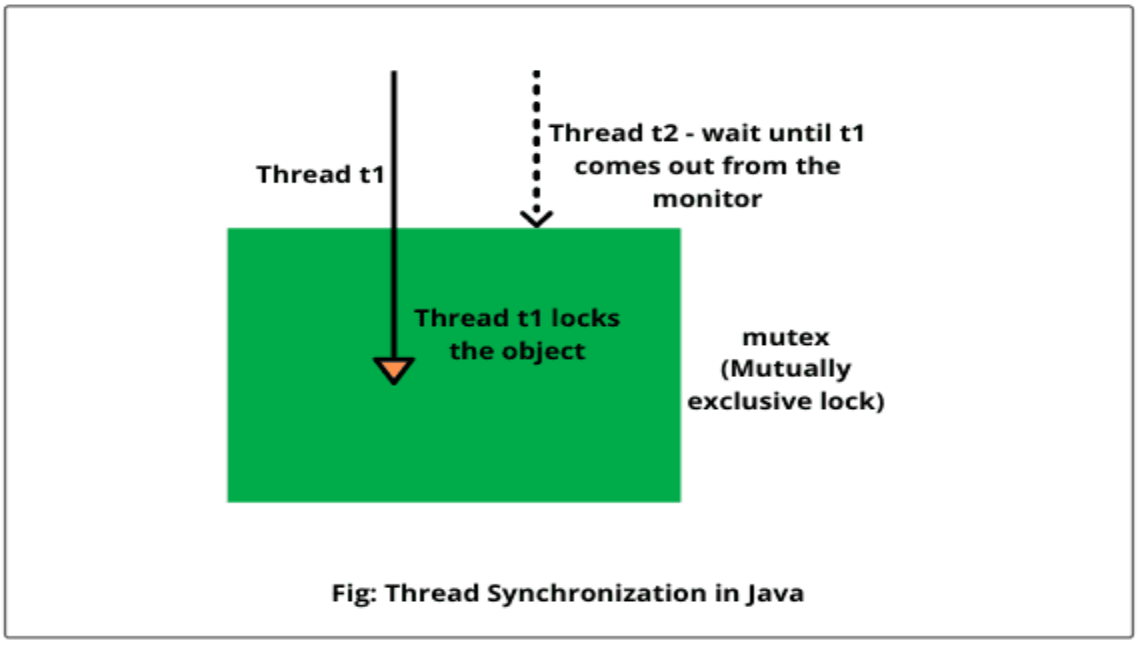
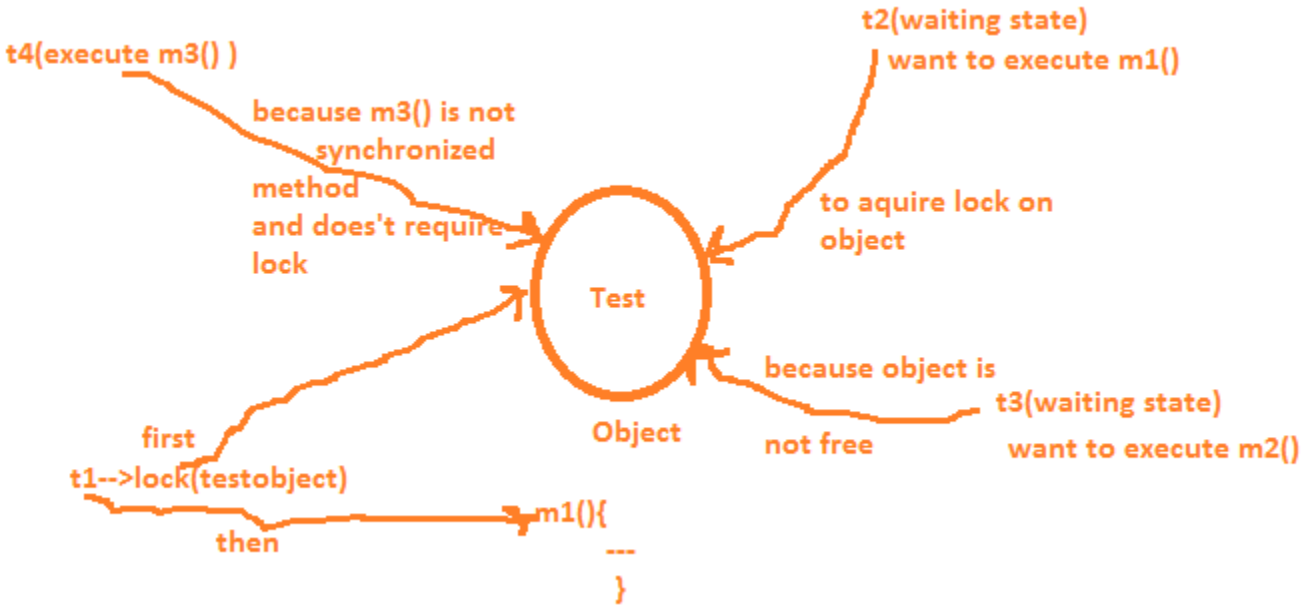
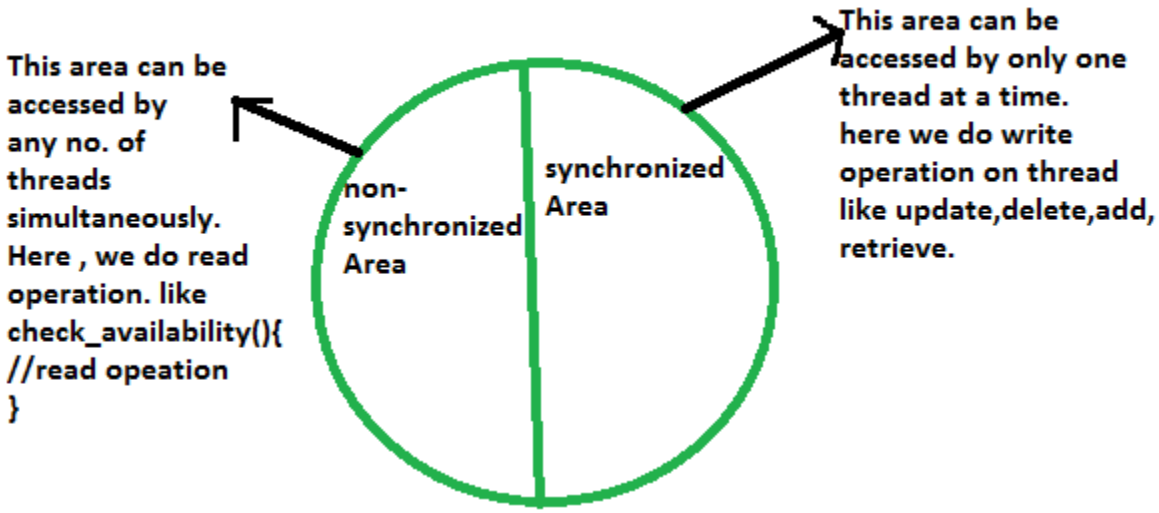


Fig: Thread Synchronization in Java

```
Ex:   class  Test{
              synchronized  void m1(){ }
              synchronized  void m2(){ }
                            void m3(){ }
                             }
```



=>While a thread executing synchronized method on the given object the remaining thread are not allowed to execute and synchronized method simultaneously on the same object, but the remaining thread are allowed to execute non- synchronized method simultaneously.



Ex:

```
Class Test{
        Synchronized(){
            Where ever we are performing update operation[add/remove/delete/retrieve]
            I.e. where state of object changing
                Ex: reserveSeat(){
                        //Write operation
                        }
                }
        Non-synchronized(){
                Where ever object state will not be changed ex:  read operation
                    Ex: check_availability(){
                            //read operation
                                }
                    }
        }
Ex:     class Reservation_System{
          Non-synchronized check_availability(){
                //just read operation
            }
```

```
    synchronized bookTicket(){
        //write operation
    }

}
```

There are two ways by which we can achieve or implement synchronization in Java. That is, we can synchronize the object. They are:

1. synchronized Method
2. synchronized Block
3. synchronized static Method

1. Synchronized Method:

When we declare a synchronized keyword in the header of a method, it is called **synchronized method**. Once a method is made synchronized, and thread calls the synchronized method, it gets locked on the method.

Syntax:

```
synchronized data_type method_name()
{
  // Code to be synchronized.
}
```

For example, if you want to synchronize the code of show() method, add the synchronize before the method name.

```
synchronized void show()
{
  // statements to be synchronized
}
```

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

```
class Table{
void printTable(int a,String s){
System.out.println("Table of:"+s);
for(int i=1;i<=5;i++){
        System.out.println(a*i);
}
}
}
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
        t.printTable(5,"5");
}

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

```
t.printTable(10,"10");
}
}
class SynchronizationEx{
public static void main(String args[]){
Table t=new Table();
MyThread1 t1=new MyThread1(t);
MyThread2 t2=new MyThread2(t);
t1.start();
t2.start();
}
}
```

Output:

```
C:\javaprograms\Multithreading>javac SynchronizationEx.java

C:\javaprograms\Multithreading>java SynchronizationEx
Table of:5
Table of:10
10
20
30
40
50
5
10
15
20
25
```

Solving problem using synchronized keyword:

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.

Example:

```
class Table{
synchronized void printTable(int a,String s){
System.out.println("Table of:"+s);
for(int i=1;i<=5;i++){
        System.out.println(a*i);
    }
}
}
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
        t.printTable(5,"5");
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
```

```
this.t=t;
}
public void run(){
t.printTable(10,"10");
}
}
class SynchronizationEx{
public static void main(String args[]){
Table t=new Table();
MyThread1 t1=new MyThread1(t);
MyThread2 t2=new MyThread2(t);
t1.start();
t2.start();
}
}
```

Output:

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.

Points to remember for Synchronized block

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

Syntax to use synchronized block

```
synchronized (object reference expression) {
  //code block
}
```

Example of synchronized block

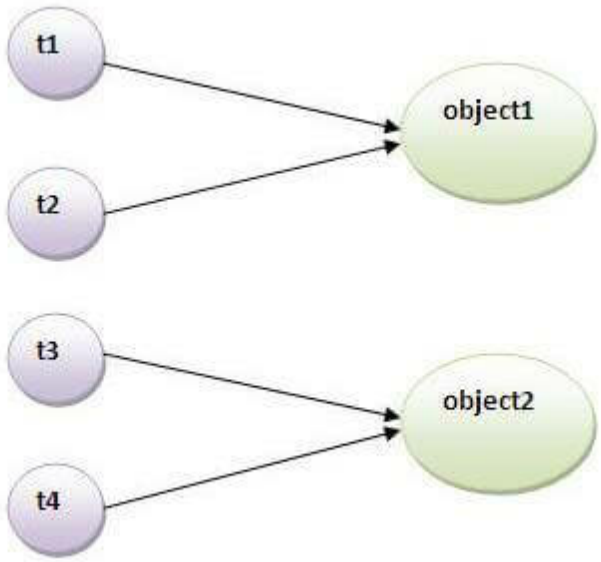
```
class Table{
void printTable(int a,String s){
        synchronized(this){
System.out.println("Table of:"+s);
for(int i=1;i<=5;i++){
        System.out.println(a*i);
}
```

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

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(10,"10");
}
}
class SynchronizationBlockEx{
public static void main(String args[]){
Table t=new Table();
MyThread1 t1=new MyThread1(t);
MyThread2 t2=new MyThread2(t);
t1.start();
t2.start();
}
}
```

```
C:\javaprograms\Multithreading>javac SynchronizationBlockEx.java

C:\javaprograms\Multithreading>java SynchronizationBlockEx
Table of:5
5
10
15
20
25
Table of:10
10
20
30
40
50
```

Static Synchronization

If you make any static method as synchronized, the lock will be on the class not on object.



Problem without static synchronization

Suppose there are two objects of a shared class(e.g. Table) named object1 and object2.In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock.But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock.I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.

Example of static synchronization

In this example we are applying synchronized keyword on the static method to perform static synchronization.

```
class Table{
  synchronized static void printTable(int n ,String s){
  System.out.println("Table of :"+s);
  for(int i=1;i<=5;i++){
    System.out.println(n*i);
    try{
      Thread.sleep(400);
    }catch(Exception e){}
  }
 }
}
  class MyThread1 extends Thread{
public void run(){
Table.printTable(2,"2");
}
}
  class MyThread2 extends Thread{
public void run(){
Table.printTable(3,"3");
}
}
  class MyThread3 extends Thread{
public void run(){
Table.printTable(4,"4");
}
}
  class MyThread4 extends Thread{
public void run(){
```

```
Table.printTable(5,"5");
}
}
  public class StaticSynchronization{
public static void main(String t[]){
MyThread1 t1=new MyThread1();
MyThread2 t2=new MyThread2();
MyThread3 t3=new MyThread3();
MyThread4 t4=new MyThread4();
t1.start();
t2.start();
t3.start();
t4.start();
}
}
```

```
C:\javaprograms\Multithreading>java StaticSynchronization
Table of :2
2
4
6
8
10
Table of :5
5
10
15
20
25
Table of :4
4
8
12
16
20
Table of :3
3
6
9
12
15
```

Inter-thread communication in Java:

Inter-thread communication in Java is a technique through which multiple threads communicate with each other.

It provides an efficient way through which more than one thread communicate with each other by reducing CPU idle time. CPU idle time is a process in which CPU cycles are not wasted.

When more than one threads are executing simultaneously, sometimes they need to communicate with each other by exchanging information with each other. A thread exchanges information before or after it changes its state.

There are several situations where communication between threads is important.

For example, suppose that there are two threads A and B. Thread B uses data produced by Thread A and performs its task.

If Thread B waits for Thread A to produce data, it will waste many CPU cycles. But if threads A and B communicate with each other when they have completed their tasks, they do not have to wait and check each other's status every time.

Thus, CPU cycles will not waste. This type of information exchanging between threads is called **inter-thread communication in Java**.

How to achieve Inter thread communication in Java

Two Threads can communicate with each other by using following methods

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

=>The Thread which is expecting updation is responsible to call wait() method then that thread enter into waiting state.

=>The thread which is responsible to perform updation and after performing updation it is responsible to call notify() method, then waiting thread get those notification and continue its execution with those updated items.

=>To call wait(),notify() ,notifyAll() methods on any object ,thread should be owner of that object. i.e The thread should has lock of that object. i.e The thread should be inside synchronized area.

=>Hence, These methods can be called only from within a synchronized method or synchronized block of code otherwise, an exception named **IllegalMonitorStateException** is thrown.

=>If a Thread calls wait() method on any object it immediately releases the lock of that particular object and entered into waiting state.

=> If a Thread calls notify() method on any object it releases the lock of that object but may not immediately as wait method. Except Wait, notify, notifyAll methods there is no other method where thread releases the lock.

=>All these methods are declared as final. Since it throws a checked exception, therefore, you must be used these methods within Java try-catch block.

wait() Method in Java

---

wait() method in Java notifies the current thread to give up the monitor (lock) and to go into waiting state until another thread wakes it up by calling notify() method. This method throws InterruptedException.

Various forms of wait() method allow us to specify the amount of time a thread can wait. They are as follows:

Syntax:

```
public final void wait()
public final void wait(long millisecond) throws InterruptedException
public final void wait(long millisecond, long nanosecond) throws InterruptedException
```

All overloaded forms of wait() method throw InterruptedException. If time is specified in the wait() method, a thread can wait for maximum time.

Note:
1. A monitor is an object which acts as a lock. It is applied to a thread only when it is inside a synchronized method.
2. Only one thread can use monitor at a time. When a thread acquires a lock, it enters the monitor.
3. When a thread enters into the monitor, other threads will wait until first thread exits monitor.
4. A lock can have any number of associated conditions.

notify() Method in Java

The notify() method wakes up a single thread that called wait() method on the same object. If more than one thread is waiting, this method will awake one of them. The general syntax to call notify() method is as follows:

Syntax:

public final void notify()

notifyAll() Method in Java

The notifyAll() method is used to wake up all threads that called wait() method on the same object. The thread having the highest priority will run first.

The general syntax to call notifyAll() method is as follows:

Syntax:
public final void notifyAll()

Let us take an example program where a threa

process of inter-thread communication:--
=>Threads enter to acquire lock.
=>Lock is acquired by on thread.
=>Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
=>If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
=>Now thread is available to acquire lock.
=>After completion of the task, thread releases the lock and exits the monitor state of the object.

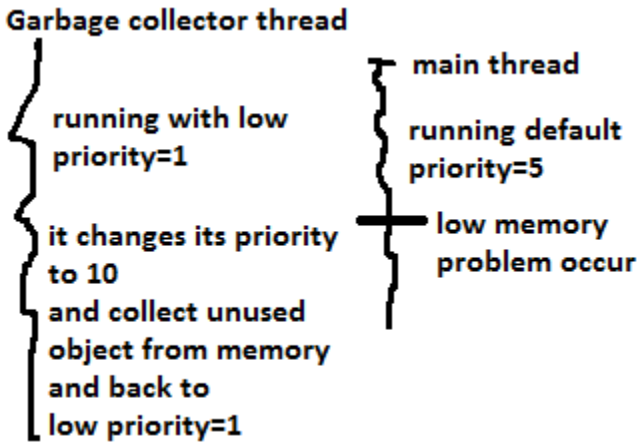
Example of inter thread communication in java:--

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

```
    void deposit(int amount){
      synchronized(this) {
     System.out.println("going  to deposit...");
    this.amount+=amount;
    System.out.println(this.amount+"is deposit and deposit completed... ");
    notify();
                }
          }
  }
class WithdrawThread extends Thread{
       Customer ob;
       WithdrawThread(Customer ob){
       this.ob=ob;
          }
    public void run(){
    ob.withdraw(20000);
           }
    }
 class DepositThread extends Thread{
Customer ob;
          DepositThread(Customer ob){
          this.ob=ob;
          }
public void run(){
      ob.deposit(15000);
    }
    }
    class InterThreadCommunication{
    public static void main(String args[]){
       Customer c=new Customer();
       WithdrawThread   t1=new WithdrawThread(c);
       DepositThread      t2=new DepositThread(c);
          t1.start();
          t2.start();
}}
```

Daemon Thread:

The Thread which are executed at background are called Daemon thread.
Ex: Garbage collector
=>The main purpose of daemon thread is to provide support for non-daemon Thread main() thread.
=> For ex: if main() thread runs with low memory then jvm runs garbage collector to destroy useless objects. So number of bytes of free memory will improved with this free memory main thread can continue its execution.



=>usually daemon thread having low priority but based on our requirement daemon thread can run with high priority also.
=>We can check daemon nature of of a thread by using isDaemon() method of thread class.

Syntax: public Boolean isDaemon()
=> We can change daemon nature of a thread by using setDaemon() method.
Syntax: public void setDaemon(Boolean b)
=>but changing daemon nature is possible before starting of a thread only. After starting a thread if we are trying to change daemon nature, we get runtime exception illegalThreadStateException.
Default Nature of a Thread:
=>By default main thread is always Non-Daemon and for all remaining Thread Daemon nature will be inherited from parent to child.
=>i.e if parent thread is Daemon then automatically child thread is also daemon and if parent thread is non-daemon then automatically child thread is also non-daemon.
Note: it is impossible to change daemon nature of main thread because it is already started by JVM at beginning.

Example:

```
class MyThread extends Thread{

}
class TestDaemonThread{
public static void main(String args[]){
System.out.println(Thread.currentThread().isDaemon());//false
//Thread.currentThread().setDaemon(true);
MyThread t=new MyThread();
System.out.println(t.isDaemon());//false
t.setDaemon(true);
System.out.println(t.isDaemon());//true
}
}
```

=>Whenever last non-daemon thread terminates then automatically all daemon thread will be terminated irrespective of their position.

```
class MyThread extends Thread{
public void run(){
for(int i=1;i<=10;i++){
System.out.println("child thread");
}
try{
Thread.sleep(2000);
}
catch(InterruptedException e){}
}

}
class DaemonThreadEx{
public static void main(String args[]){
MyThread t=new MyThread();
t.setDaemon(true);
t.start();
System.out.println("End of main thread");
}
}
```

Note:
=>if we are commenting line t.setDaemon(true) then both main and child thread are non-daemon thread ,hence both thread will be executed until their execution.
=>If we are not commenting line t.setDaemon(true) then main thread is non-daemon and child thread is daemon, so whenever main thread terminated ,automatically child thread is terminated .

ThreadGroup

Java provides a convenient way to group multiple threads in a single object. In such way, we can

suspend, resume or interrupt group of threads by a single method call.

Java thread group is implemented by *java.lang.ThreadGroup* class.

A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates 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 it cannot access the information about its thread group's parent thread group or any other thread groups.

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

No.	Constructor	Description
1)	ThreadGroup(String name)	creates a thread group with given name.
2)	ThreadGroup(ThreadGroup parent, String name)	creates a thread group with given parent group and name.

ThreadGroup Example

```
public class ThreadGroupEx extends Thread{
   public void run() {
        System.out.println(Thread.currentThread().getName());
                          }
  public static void main(String[] args) {
     ThreadGroupEx  tgd = new ThreadGroupEx();
      ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");
      Thread t1 = new Thread(tg1, tgd,"one");
        t1.start();
        Thread t2 = new Thread(tg1, tgd,"two");
        t2.start();
        Thread t3 = new Thread(tg1, tgd,"three");
        t3.start();
        System.out.println("Thread Group Name: "+tg1.getName());
         tg1.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]
```

Generic Programming:

Generics:

=>The main objective of generics are to provide type-safety and to resolve type -casting problems.
=>It makes the code stable by detecting the bugs at compile time.
=>Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

1) Type-safety: We can hold only a single type of objects in generics. It doesn't allow to store other objects.

=>Without Generics, we can store any type of objects.

```
List list = new ArrayList();
list.add(10);
list.add("10");
With Generics, it is required to specify the type of object we need to store.
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

2) Type casting is not required: There is no need to typecast the object.

Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
After Generics, we don't need to typecast the object.
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

3) Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

Syntax to use generic collection

```
Class Or Interface<Type>
```

Example to use Generics in java

```
ArrayList<String>
List<String>
```

Full Example of Generics:

Here, we are using the ArrayList class, but we can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc.

```
import java.util.*;
class TestGenerics1{
public static void main(String args[]){
```

```
ArrayList<String> list=new ArrayList<String>();
list.add("Rohit");
list.add("jayant");
//list.add(32);//compile time error
String s=list.get(1);//type casting is not required
System.out.println("element is: "+s);
Iterator<String> itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
```

Output:

```
element is: jayant
Rohit
jayant
```

Generic class

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

example to create and use the generic class.

Creating a generic class:

```
class MyGen<T>{
T obj;
void add(T obj){
this.obj=obj;
}
T get(){
return obj;
}
}
```

The T type indicates that it can refer to any type (like String, Integer, and Employee). The type you specify for the class will be used to store and retrieve the data.

Using generic class:

the code to use the above generic class is as following.

```
class TestGenerics3{
public static void main(String args[]){
MyGen<Integer> m=new MyGen<Integer>();
m.add(2);
//m.add("vivek");//Compile time error
System.out.println(m.get());
}}
```

Output

Type Parameters

The type parameters naming conventions are important for generics.The common type parameters are as follows:

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

Generic Method:

Like the generic class, we can create a generic method that can accept any type of arguments. Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.

example of java generic method to print array elements. We are using here **E** to denote the element.

```
public class GenericMethod{
    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 = { 'G', 'E', 'N', 'E', 'R','I','C','S'};
         System.out.println( "Printing Integer Array" );
       printArray( intArray  );
        System.out.println( "Printing Character Array" );
       printArray( charArray );
   }
}
```

Bounded Types: Restrictions and Limitations.

1. Upper Bounded:

The purpose of upper bounded wildcards is to decrease the restrictions on a variable. It restricts the unknown type to be a specific type or a subtype of that type. It is used by declaring wildcard character ("?") followed by the extends (in case of, class) or implements (in case of, interface) keyword, followed by its upper bound.

Syntax: m List<? extends Number>

Here, **?** is a wildcard character, **extends**, is a keyword , **Number**, is a class present in java.lang package

Suppose, we want to write the method for the list of Number and its subtypes (like Integer, Double). Using **List<? extends Number>** is suitable for a list of type Number or any of its subclasses whereas **List<Number>** works with the list of type Number only. So, **List<? extends Number>** is less restrictive than **List<Number>**.

Example of Upper Bound

In this example, we are using the upper bound wildcards to write the method for List<Integer> and List<Double>.

```
import java.util.ArrayList;
 public class UpperBoundEx{
    private static Double add(ArrayList<? extends Number> num) {
  double sum=0.0;
       for(Number n:num)
     {
        sum = sum+n.doubleValue();
     }
         return sum;
  }
    public static void main(String[] args) {
     ArrayList<Integer>  l1=new ArrayList<Integer>();
      l1.add(10);
      l1.add(20);
      System.out.println("displaying the sum= "+add(l1));
     ArrayList<Double> l2=new ArrayList<Double>();
      l2.add(30.0);
      l2.add(40.0);
      System.out.println("displaying the sum= "+add(l2));
} }
```

Output
displaying the sum= 30.0
displaying the sum= 70.0

2.Lower Bounded:

The purpose of lower bounded wildcards is to restrict the unknown type to be a specific type or a supertype of that type. It is used by declaring wildcard character ("?") followed by the super keyword, followed by its lower bound.

Syntax: List<? super Integer>

Here, **?** is a wildcard character, **super**, is a keyword, **Integer**, is a wrapper class.

Suppose, we want to write the method for the list of Integer and its supertype (like Number, Object). Using **List<? super Integer>** is suitable for a list of type Integer or any of its superclasses whereas **List<Integer>** works with the list of type Integer only. So, **List<? super Integer>** is less restrictive than **List<Integer>**.

Example of Lower Bound Wildcard

In this example, we are using the lower bound wildcards to write the method for List<Integer> and List<Number>.

```
import java.util.*;
class LowerBoundEx{
static void addNumber(ArrayList<? super Integer> list){
for(Object n:list)
{
        System.out.println(n);
```

```
}
}
public static void main(String args[]){
ArrayList<Integer> al=new ArrayList<Integer>();
al.add(10);
al.add(20);
System.out.println ("Dispalay value of Integer type");
addNumber(al);
ArrayList<Number> al2=new ArrayList<Number>();
al2.add(10.0);
al2.add(20.0);
System.out.println ("Dispalay value of Number type");
addNumber(al2);
}
}
```

Output

```
displaying the Integer values
1
2
3
displaying the Number values
1.0
2.0
3.0
```