Unit - 3

Multithreading: -

1. **Differences between multi-threading and multitasking**
2. **Thread life cycle**
3. **Creating threads**
4. **Thread priorities**
5. **Synchronizing threads**
6. **Inter thread communication**

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

Process:-

- A Program does nothing unless its instructions are executed by a CPU. A program in execution is called a process.

1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

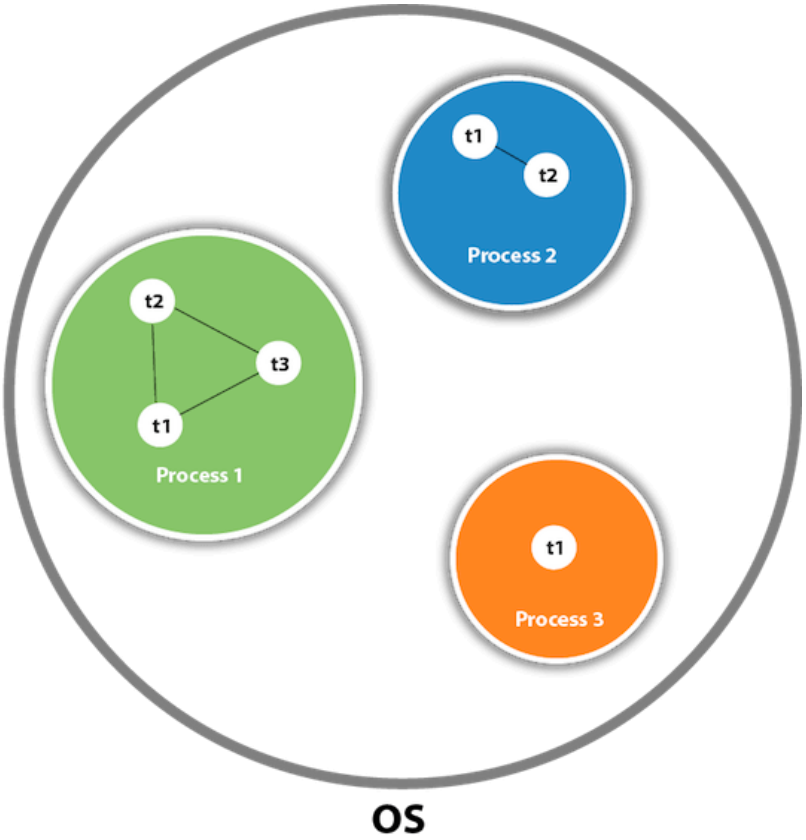
- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

Note: At least one process is required for each thread.

Thread:-

A thread is a lightweight sub process, 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.

Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

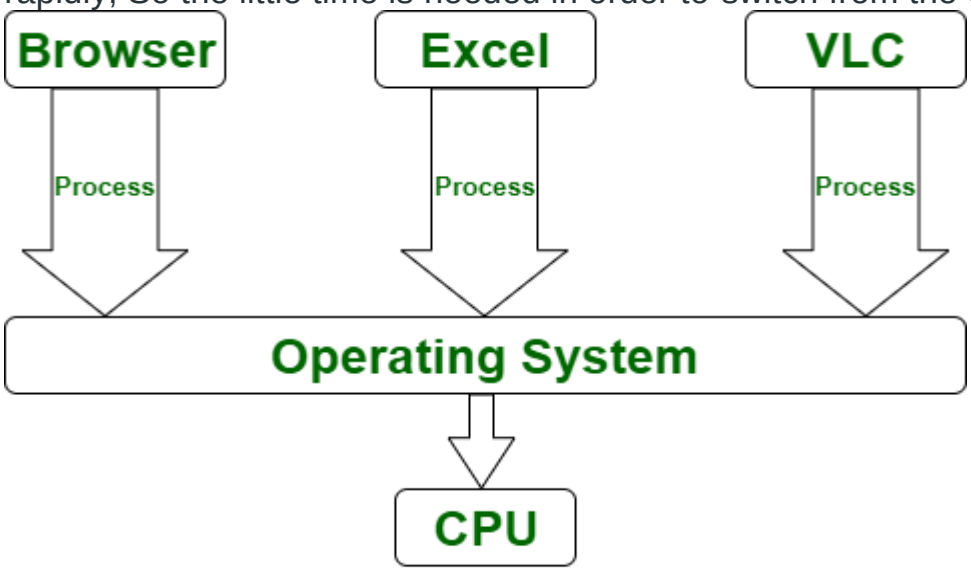
Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

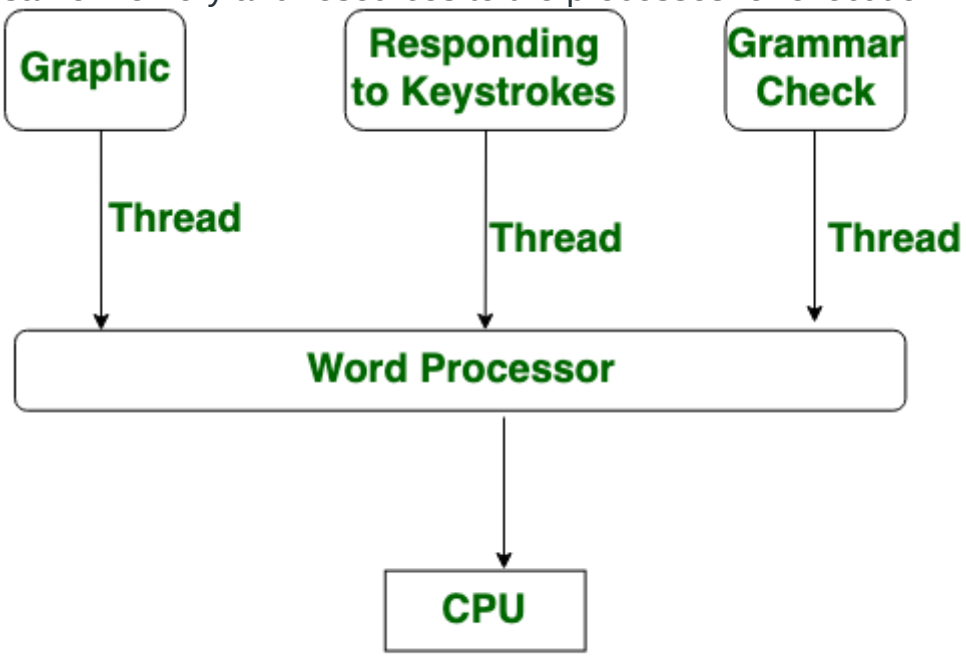
3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking: Multitasking is when a CPU is provided to execute multiple tasks at a time. Multitasking involves often CPU switching between the tasks, so that users can collaborate with each program together. Unlike multithreading, In multitasking, the processes share separate memory and resources. As multitasking involves CPU switching between the tasks rapidly, So the little time is needed in order to switch from the one user to next.



Multitasking

Multithreading: Multithreading is a system in which many threads are created from a process through which the computer power is increased. In multithreading, CPU is provided in order to execute many threads from a process at a time, and in multithreading, process creation is performed according to cost. Unlike multitasking, multithreading provides the same memory and resources to the processes for execution.



Multithreading

Example Program:-

- Write a program to find the thread used by JVM to execute the statements.

```
class ThreadClass2
{
    public static void main(String args[])
    {
        System.out.println(Thread.currentThread());
        System.out.println(Thread.currentThread().getName());
    }
}
```

O/P:-

```
Thread[main,5,main]
Main
```

1.difference between multitasking and multithreading:

S.NO	Multitasking	Multithreading
1.	In multitasking, users are allowed to perform many tasks by CPU.(or) Multitasking is a process of executing multiple tasks simultaneously.	In multithreading, multiple threads are created from a process.(or) Multithreading is a process of executing multiple threads simultaneously.
2.	Multitasking involves often CPU switching between the tasks.	While in multithreading also, CPU switching is often involved between the threads.
3.	In multitasking, the processes share separate memory.	While in multithreading, processes are allocated the same memory.
4.	In multitasking, the CPU is provided in order to execute many tasks at a time.	While in multithreading also, a CPU is provided in order to execute many threads from a process at a time.
5.	In multitasking, processes don’t share the same resources, each process is allocated separate resources.	While in multithreading, each process shares the same resources.
6.	Multitasking is slow compared to multithreading.	While multithreading is faster.
7.	In multitasking, termination of a process takes more time.	While in multithreading, termination of thread takes less time.
8.	Involves running multiple independent processes or tasks	Involves dividing a single process into multiple threads that can execute concurrently
9.	Each process or task has its own memory space and resources	Threads share the same memory space and resources of the parent process
10.	Each process or task has its own memory space and resources	Threads share the same memory space and resources of the parent process
11.	Examples: running multiple applications on a computer, running multiple servers on a network	Examples: splitting a video encoding task into multiple threads, implementing a responsive user interface in an application

2.Thread life cycle

Starting from the birth of a thread, till its death, a thread exists in different states which are collectively called 'Thread Life Cycle'.

Thread States in Java

A thread is a path of execution in a program that enters any one of the following five states during its life cycle. The five states are as follows:

1. New
2. Runnable
3. Running
4. Blocked (Non-runnable state)
5. Dead

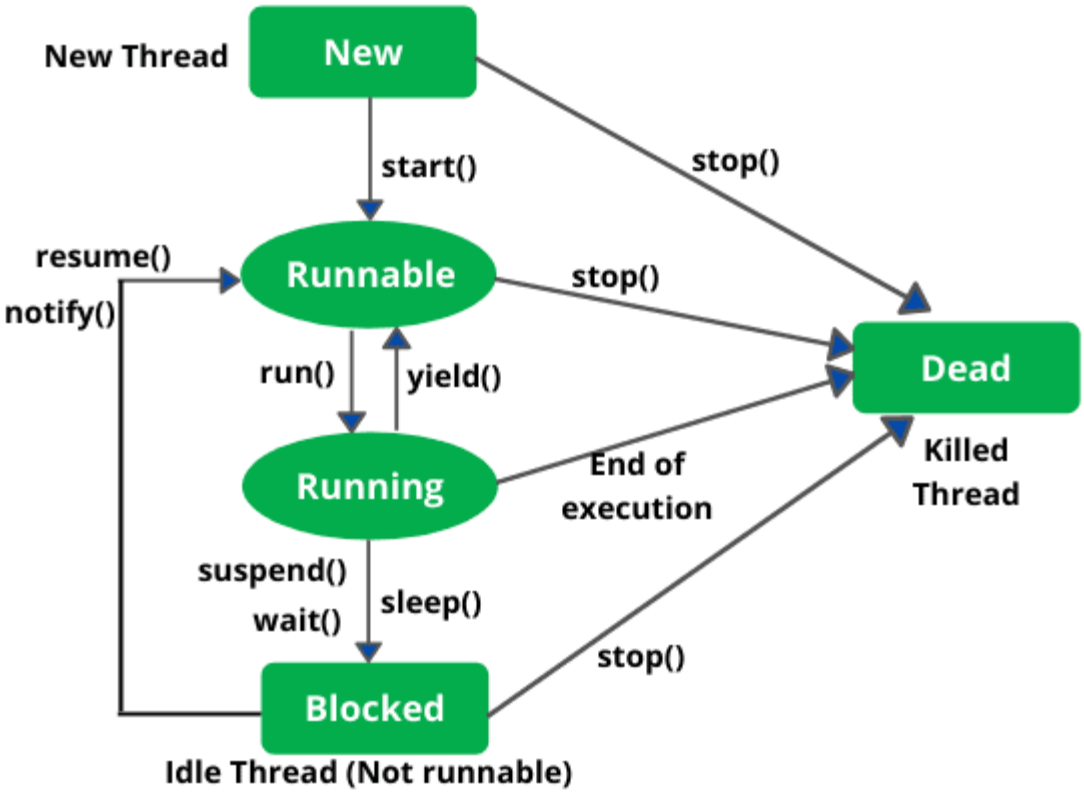


Fig. Thread Life Cycle

1. **New (Newborn State):** When we create a thread object using Thread class, thread is born and is known to be in Newborn state. That is, when a thread is born, it enters into new state but the start() method has not been called yet on the instance.
   In other words, Thread object exists but it cannot execute any statement because it is not an execution of thread. Only start() method can be called on a new thread; otherwise, an **IllegalThreadStateException** will be thrown.

2. **Runnable state:** Runnable state means a thread is ready for execution. When the start() method is called on a new thread, thread enters into a runnable state.
    - In runnable state, thread is ready for execution and is waiting for availability of the processor (CPU time). That is, thread has joined queue (line) of threads that are waiting for execution.
    - If all threads have equal priority, CPU allocates time slots for thread execution on the basis of first-come, first-serve manner. The process of allocating time to threads is known as **time slicing**. A thread can come into runnable state from running, waiting, or new states.
3. **Running state:** Running means Processor (CPU) has allocated time slot to thread for its execution. When thread scheduler selects a thread from the runnable state for execution, it goes into running state. Look at the above figure.
    - In running state, processor gives its time to the thread for execution and executes its run method. This is the state where thread performs its actual functions. A thread can come into running state only from runnable state.
    - A running thread may give up its control in any one of the following situations and can enter into the blocked state.

      a) When sleep() method is invoked on a thread to sleep for specified time period, the thread is out of queue during this time period. The thread again reenters into the runnable state as soon as this time period is elapsed.

      b) When a thread is suspended using suspend() method for some time in order to satisfy some conditions. A suspended thread can be revived by using resume() method.

      c) When wait() method is called on a thread to wait for some time. The thread in wait state can be run again using notify() or notifyAll() method.
4. **Blocked state:** A thread is considered to be in the blocked state when it is suspended, sleeping, or waiting for some time in order to satisfy some condition.
5. **Dead state:** A thread dies or moves into dead state automatically when its run() method completes the execution of statements. That is, a thread is terminated or dead when a thread comes out of run() method. A thread can also be dead when the stop() method is called.

Thread Class

Thread class contains several constructors for creating threads for tasks and methods for controlling threads. It is a predefined class declared in java.lang default package.

Each thread in Java is created and controlled by a unique object of the Thread class. An object of thread controls a thread running under JVM.
Thread class contains various methods that can be used to start, control, interrupt the execution of a thread, and for many other thread related activities in a program.

Thread class extends Object class and it implements Runnable interface. The declaration of thread class is as follows:

```
public class Thread

  extends Object

    implements Runnable
```

Look at the below figure that shows class diagram of Java Thread class.

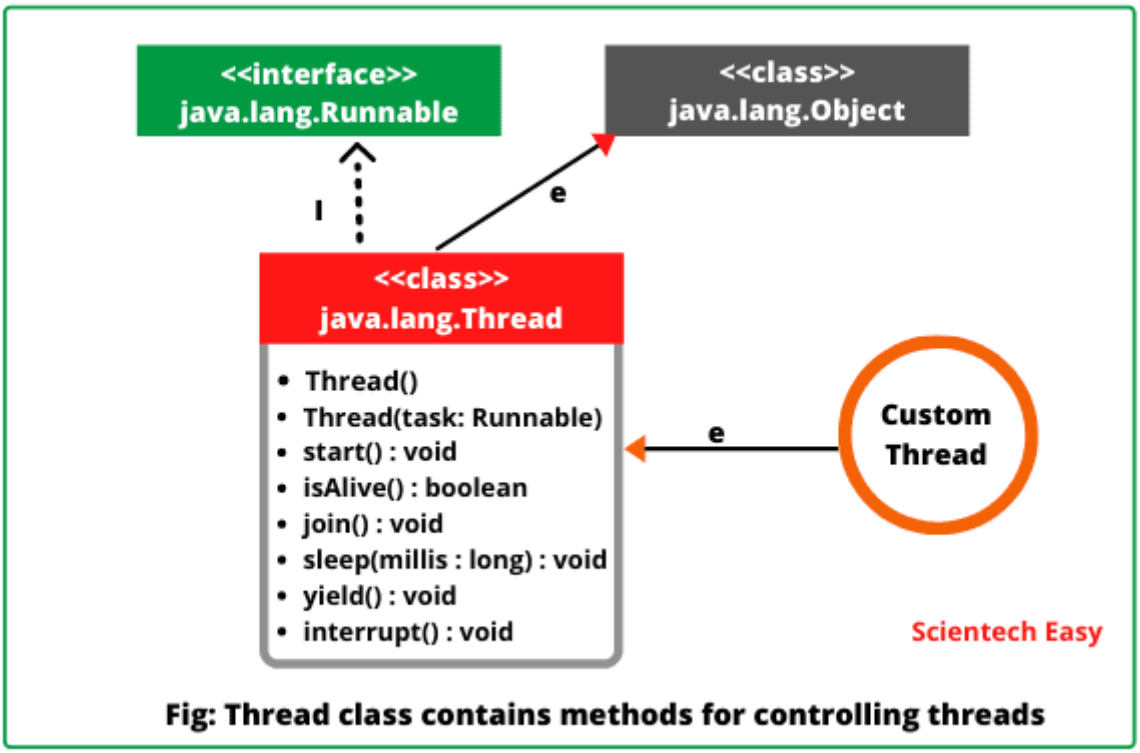


Fig: Thread class contains methods for controlling threads

Thread class Constructor: -

The various constructors of thread class are defined in java.lang package that can be used to create an object of thread are as follows:

1. **Thread():** This is a basic and default constructor without parameters. It simply creates an object of Thread class.
2. **Thread(String name):** It creates a new thread object with specified name to a thread.
3. **Thread(Runnable r):** It creates a thread object by passing a parameter r as a reference to an object of the class that implements Runnable interface.
4. **Thread(Runnable r, String name):** This constructor creates a thread object by passing two arguments r and name. Here, variable r is a reference of an object of class that implements Runnable interface.

Methods of Thread class:-

Thread class provides various static methods that are as follows:

1. **currentThread():** The currentThread() returns the reference of currently executing thread. Since this is a static method, so we can call it directly using the class name. The general syntax for currentThread() is as follows:

Syntax:

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

2. **sleep():** The sleep() method puts currently executing thread to sleep for specified number of milliseconds. This method is used to pause the current thread for specified amount of time in milliseconds.
Since this method is static, so we can access it through Thread class name. The general syntax of this method is as follows:

Syntax:

```
public static void sleep(long milliseconds) throws InterruptedException
```

The general syntax for overloaded version of sleep method is as follows:

Syntax:

```
public static void sleep(long milliseconds, int nanoseconds ) throw InterruptedException
```

The overloaded version of sleep() method is used to pause specified period of time in milliseconds and nanoseconds. Both methods throw InterruptedException and must be used within Java try-catch block.

3. **yield():** The yield() method pauses the execution of current thread and allows another thread of equal or higher priority that are waiting to execute. Currently executing thread give up the control of the CPU. The general form of yield() method is as follows:

Syntax:

```
public static void yield()
```

4. **activeCount():** This method returns the number of active threads.

Syntax:

```
public static int activeCount()
```

Since this method is static, so it can be accessed through Thread class name. It does not accept anything.

The instance methods of Thread class are as follows:

1. **start():** The start() method is used to start the execution of a thread by calling its run() method. JVM calls the run() method on the thread. The general syntax for start() method is as follows:

Syntax:

```
public void start()
```

2. **run():** The run() method moves the thread into running state. It is executed only after the start() method has been executed. The general syntax is as follows:

Syntax:

```
public void run()
```

3. **getName():** This method returns the name of the thread. The return type of this method is String. The general form of this method is:

Syntax:

```
public final String getName()
```

4. **setName():** This method is used to set the name of a thread. It takes an argument of type String. It returns nothing.

Syntax:

```
public final void setName(String name)
```

5. **getPriority():** This method returns the priority of a thread. It returns priority in the form of an integer value ranging from 1 to 10. The maximum priority is 10, the minimum priority is 1, and normal priority is 5.

Syntax:

```
public final int getPriority() // Return type is an integer.
```

6. **setPriority():** This method is used to set the priority of a thread. It accepts an integer value as an argument. The general syntax is given below:

Syntax:

```
public final void setPriority(int newPriority)
```

7. **isAlive():** This method is used to check the thread is alive or not. It returns a boolean value (true or false) that indicates thread is running or not. The isAlive() method is final and native. The general syntax for isAlive() method is as follows:

Syntax:

```
public final native boolean isAlive()
```

8. **join():** The join() method is used to make a thread wait for another thread to terminate its process. The general syntax is

Syntax:

```
public final void join() throw InterruptedException
```

This method throws InterruptedException and must be used within a try-catch block.

9. **stop():** This method is used to stop the thread. The general form for this method is as follows:

Syntax:

```
public final void stop()
```

This method neither accepts anything nor returns anything.

10. **suspend():** The suspend() method is used to suspend or pause a thread.

Syntax:

```
public final void suspend()
```

11. **resume():** This method is used to resume the suspended thread. It neither accepts anything nor returns anything.

Syntax:

```
public final void resume()
```

12. **isDaemon():** This method is used to check the thread is daemon thread or not.

Syntax:

```
public final boolean isDaemon()
```

Runnable Interface Method:-

public void run():-

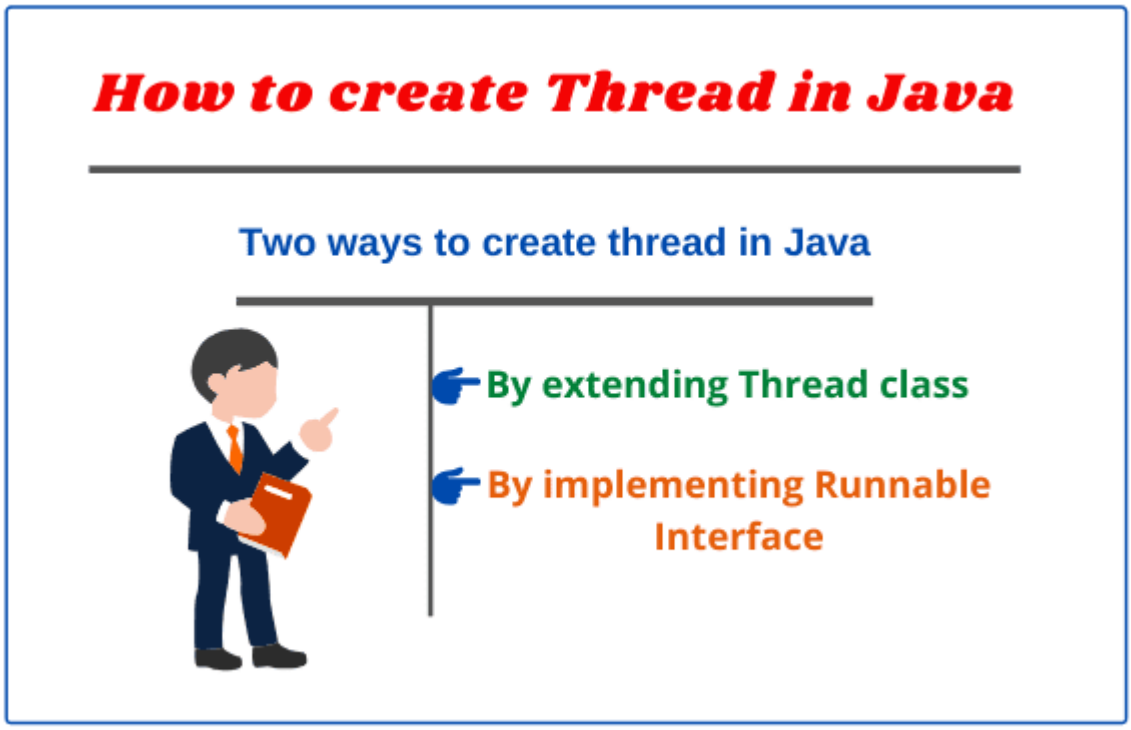
This method takes in no arguments. When the object of a class implementing Runnable class is used to create a thread, then the run method is invoked in the thread which executes separately.

3.Creating threads

Creating threads in Java | We know that every Java program has at least one thread called main thread. When a program starts, main thread starts running immediately. Apart from this main thread, we can also create our own threads in a program that is called child thread. Every child threads create from its main thread known as parent thread.

There are two ways to create a new thread in Java. They are as follows:

1. One is by extending java.lang.Thread class
2. Another is by implementing java.lang.Runnable interface



1.By Extending Thread class

Extending Thread class is the easiest way to create a new thread in Java. The following steps can be followed to create your own thread in Java.

1. Create a class that extends the Thread class. In order to extend a thread, we will use a keyword extends. The Thread class is found in java.lang package.

The syntax for creating a new class that extends Thread class is as follows:

```
Class Myclass extends Thread
```

2. Now in this newly created class, define a method run(). Here, run() method acts as entry point of the new thread. The run() method contains the actual task that thread will perform. Thus, we override run() method of Thread class.

```
public void run()

{

 // statements to be executed.

}
```

3. Create an object of newly created class so that run() method is available for execution. The syntax to create an object of Myclass is as follows:

```
Myclass obj = new Myclass();
```

4. Now create an object of Thread class and pass the object reference variable created to the constructor of Thread class.

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

or,

 Thread t = new Thread(obj, "threadname");
```

5. Run the thread. For this, we need to call to start() method of Thread class because we cannot call run() method directly. The syntax to call start() method is as follows:

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

Now, the thread will start execution on the object of Myclass, and statements inside run() method will be executed while calling it. By following all the above steps, you can create a Thread in Java.

Example Program 1:

```
class MyThread extends Thread{
   @Override
   public void run()
   {
      for(int i=1;i<=2;i++)
      {
         System.out.println("Child Thread");
      }
   }
   public static void main(String args[])
   {
   MyThread t=new MyThread();
   t.start();
   }
}
```

Out Put:-

```
Child Thread
```

Child Thread

Example Program 2:-

```
class MyThread extends Thread{
    @Override
    public void run()
    {
        for(int i=1;i<=2;i++)
        {
            System.out.println(Thread.currentThread());
        }
    }
    public static void main(String args[])
    {
    MyThread t=new MyThread();
    MyThread t1=new MyThread();
     t.start();
     t1.start();
    for(int i=1;i<=2;i++)
        {
            System.out.println(Thread.currentThread());
        }
    }
}
```

Out Put:-

```
Thread[Thread-0,5,main]
Thread[Thread-1,5,main]
Thread[main,5,main]
Thread[Thread-1,5,main]
Thread[Thread-0,5,main]
Thread[main,5,main]
```

2. By Implementing Runnable Interface

1. Threads can also be created by implementing Runnable interface of java.land package. Creating a thread by implementing Runnable interface is very similar to creating a thread by extending Thread class.
2. All steps are similar for creating a thread by implementing Runnable interface except the first step. The first step is as follows:
3. 1. To create a new thread using this method, create a class that implements Runnable interface of java.lang package.
4. The syntax for creating a new class that implements Runnable interface is as follows:
5. class Myclass implements Runnable

Example Program:-

```
class MyThread implements Runnable
{
  public void run()
  {
   for(int i=1;i<=5;i++)
   {
      System.out.println("My Child Thread");
   }

  }
   public static void main(String args[])
   {
      // Create an object of MyThread class.
      MyThread r=new MyThread();
      // Create an object of Thread class and pass reference variable r to Thread class constructor.
      Thread t=new Thread(r);
      t.start();// This thread will execute statements inside run() method of MyThread object.
      for(int i=1;i<=5;i++)
   {
```

```
            System.out.println("Parent Thread");
        }

    }
}
```

Out Put:-

```
Parent Thread
My Child Thread
My Child Thread
My Child Thread
My Child Thread
My Child Thread
Parent Thread
Parent Thread
Parent Thread
Parent Thread
```

Thread Schedular:-

Thread scheduler in Java is the component of JVM that determines the execution order of multiple threads on a single processor (CPU).
It decides the order in which threads should run. This process is called **thread scheduling in Java**.
When a system with a single processor executes a program having multiple threads, CPU executes only a single thread at a particular time.

Other threads in the program wait in Runnable state for getting the chance of execution on CPU because at a time only one thread can get the chance to access the CPU.

The thread scheduler selects a thread for execution from runnable state. But there is no guarantee that which thread from runnable pool will be selected next to run by the thread scheduler.

Java runtime system mainly uses one of the following two strategies:

1. **Preemptive scheduling**
2. **Time-sliced scheduling**

Preemptive Scheduling

This scheduling is based on priority. Therefore, this scheduling is known as priority-based scheduling. In the priority-based scheduling algorithm, Thread scheduler uses the priority to decide which thread should be run.

If a thread with a higher priority exists in Runnable state (ready state), it will be scheduled to run immediately.

In case more than two threads have the same priority then CPU allocates time slots for thread execution on the basis of first-come, first-serve manner.

Time-Sliced Scheduling

The process of allocating time to threads is known as **time slicing in Java**. Time-slicing is based on non-priority scheduling. Under this scheduling, every running thread is executed for a fixed time period.
A currently running thread goes to the Runnable state when its time slice is elapsed and another thread gets time slots by CPU for execution.

With time-slicing, threads having lower priorities or higher priorities gets the same time slots for execution.

4. Thread priorities

Thread priority in Java is a number assigned to a thread that is used by *Thread scheduler* to decide which thread should be allowed to execute.
In Java, each thread is assigned a different priority that will decide the order (preference) in which it is scheduled for running.

Thread priorities are represented by a number from 1 to 10 that specifies the relative priority of one thread to another. The thread with the highest priority is selected by the scheduler to be executed first.

The default priority of a thread is 5. *Thread class in Java* also provides several priority constants to define the priority of a thread. These are:

3 constants defined in Thread class:

1. MIN_PRIORITY = 1
2. NORM_PRIORITY = 5
3. MAX_PRIORTY = 10

These constants are public, final, and static members(or) variables of the Thread class.

Thread scheduler selects the thread for execution on the first-come, first-serve basis. That is, the threads having equal priorities share the processor time on the first-come, first-serve basis.

When multiple threads are ready for execution, the highest priority thread is selected and executed by JVM. In case when a high priority thread stops, yields, or enters into the blocked state, a low priority thread starts executing.

If any high priority thread enters into the runnable state, it will preempt the currently running thread forcing it to move to the runnable state. Note that the highest priority thread always preempts any lower priority thread.

Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the thread priority.

public final int getPriority(): The java.lang.Thread.getPriority() method returns the priority of the given thread.

public final void setPriority(int newPriority): The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

Syntax:

```
ThreadName.setPriority(n);

 where, n is an integer value which ranges from 1 to 10.
```

Example program:-

```
class PriorityEx implements Runnable
{
   @Override
   public void run()
   {
      System.out.println(Thread.currentThread().getPriority());
   public static void main(String args[])
   {
PriorityEx a=new PriorityEx ();
      Thread t=new Thread(a,"c1");
```

```
    Thread t1=new Thread(a,"c2");
    t.setPriority(4);//If you assign priority value above 10 number its generate IllegalArgumentException
    t1.setPriority(3);
    System.out.println(t1.MIN_PRIORITY);
    Thread.currentThread().setPriority(7);//main thread priority assigning
    System.out.println(Thread.currentThread().getPriority());//calling main thread priority number
    t.start();
    t1.start();
  }

}
```

O/P:-

7

4

3

1

5.Synchronizing threads:-

Thread synchronization in java is a way of programming several threads to carry out independent tasks easily. It is capable of controlling access to multiple threads to a particular shared resource. The main reason for using thread synchronization are as follows:

- To prevent interference between threads.
- To prevent the problem of consistency.

Types of Thread Synchronization

In Java, there are two types of thread synchronization:

1. Process synchronization
2. Thread synchronization

Process synchronization- The process means a program in execution and runs independently isolated from other processes. CPU time, memory, etc resources are allocated by the operating system.

Thread synchronization- It refers to the concept where only one thread is executed at a time while other threads are in the waiting state. This process is called thread synchronization. **It is used because it avoids interference of thread and the problem of inconsistency.**

In java, thread synchronization is further divided into two types:

- Mutual exclusive- it will keep the threads from interfering with each other while sharing any resources.
- Inter-thread communication- It is a mechanism in java in which a thread running in the critical section is paused and another thread is allowed to enter or lock the same critical section that is executed.

Mutual Exclusive

It is used for preventing threads from interfering with each other and to keep distance between the threads. Mutual exclusive is achieved using the following:

- Synchronized Method
- Synchronized Block
- Static Synchronization

Lock Concept in Java

- Synchronization Mechanism developed by using the synchronized keyword in java language. It is built on top of the locking mechanism, this locking mechanism is taken care of by Java Virtual Machine (JVM). The synchronized keyword is only applicable for methods and blocks, it can't apply to classes and variables. Synchronized keyword in java creates a block of code is known as a critical section. To enter into the critical section thread needs to obtain the corresponding object's lock.

The problem without Synchronization:

Example Program:-

```
class E implements Runnable
{
   int available=5000,amount;
   E(int amount)
   {
      this.amount=amount;
   }
   @Override
   public void run()
   {
      String name=Thread.currentThread().getName();
      if(available>=amount)
      {
         System.out.println(name+" Withdraw Amount");
         available=available-amount;
      }
      else
      {
```

```
            System.out.println("Sorry amount not available");
        }
        try{
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {
        }
    }
}
public class ThreadSynchronization {
   public static void main(String args[])
   {
       E e=new E(5000);
       Thread t=new Thread(e);
       Thread t1=new Thread(e);
       Thread t2=new Thread(e);
         t.setName("Rahul");
         t1.setName("Sujan");
         t2.setName("Vishnu");
       t.start();
       t1.start();
       t2.start();
   }
}
```

OutPut:-

Vishnu withdraw Amount

Sujan withdraw Amount

Rahul withdraw Amount

Here we didn't use the synchronized keyword so multiple threads are executing at a time so in the output, vishnu is interfering with sujan and Rahul, and hence, we are getting inconsistent results

Java Synchronized Method

If we use the Synchronized keywords in any method then that method is Synchronized Method.

- It is used to lock an object for any shared resources.
- The object gets the lock when the synchronized method is called.
- The lock won't be released until the thread completes its function.

Syntax:

Acess_modifiers synchronized return_type method_name (Method_Parameters) {

// Code of the Method.

}

Example program:-

```
class E implements Runnable
{
   int available=5000,amount;
   E(int amount)
   {
      this.amount=amount;
   }
   @Override
   public synchronized void run()
   {
      String name=Thread.currentThread().getName();
      if(available>=amount)
      {
         System.out.println(name+" Withdraw Amount");
         available=available-amount;
```

```
        }
        else
        {
            System.out.println("Sorry amount not available");
        }
        try{
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {
        }
    }
}
public class ThreadSynchronization {
   public static void main(String args[])
   {
       E e=new E(5000);
       Thread t=new Thread(e);
       Thread t1=new Thread(e);
       Thread t2=new Thread(e);
         t.setName("Rahul");
         t1.setName("Sujan");
         t2.setName("Vishnu");
       t.start();
       t1.start();
       t2.start();
   }
}
```

OutPut:-

Rahul withdraw Amount

Sorry amount not available

Sorry amount not available

Here we used synchronized keywords. It helps to execute a single thread at a time. It is not allowing another thread to execute until the first one is completed, after completion of the first thread it allowed the second thread. Now we can see the output correctly only Rahul withdraw amount next other persons get no amount.

(Or)

Example Program for synchronized method:-

```
public class ThreadEx implements Runnable{
   public synchronized void run()
   {
      for(int i=1;i<=5;i++)
      {
         System.out.println(Thread.currentThread().getName());
      }
   }
   public static void main(String args[])
   {
      ThreadEx a=new ThreadEx();
      Thread t1=new Thread(a,"First");
      Thread t2=new Thread(a,"Second");
      Thread t3=new Thread(a,"Third");
      t1.start();
      t2.start();
      t3.start();
   }
}
```

Synchronized Block

- Suppose you don't want to synchronize the entire method, you want to synchronize few lines of code in the method, then a synchronized block helps to synchronize those few lines of code. It will take the object as a parameter. It will work the same as Synchronized Method. In the case of synchronized method lock accessed is on the method but in the case of synchronized block lock accessed is on the object.

Syntax:

```
synchronized (object) {

//code of the block.

}
```

Program to understand the Synchronized Block:

Example:-

```
class E implements Runnable
{
   int available=5000,amount;
   E(int amount)
   {
      this.amount=amount;
   }

   @Override
   public void run()
   {
      System.out.println(Thread.currentThread().getName());
      String name=Thread.currentThread().getName();
      synchronized(this){
      if(available>=amount)
      {
         System.out.println(name+" withdraw Amount");
         available=available-amount;
      }
```

```
        else
        {
            System.out.println("Sorry amount not available");
        }
        try{
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {
        }
        }
    }
}
public class ThreadSynchronization {
   public static void main(String args[])
   {
       E e=new E(1);
       Thread t=new Thread(e);
       Thread t1=new Thread(e);
       Thread t2=new Thread(e);
         t.setName("Rahul");
         t1.setName("Sujan");
         t2.setName("Vishnu");
       t.start();
       t1.start();
       t2.start();
   }
}
```

OutPut:-
Rahul
Vishnu
Sujan
Rahul withdraw Amount
Sorry amount not available
Sorry amount not available

In this example, we didn't synchronize the entire method but we synchronized few lines of code in the method. We got the results exactly as the synchronized method.

(or)

Example Program for Synchronized block:-

```
public class ThreadEx implements Runnable{

   public void run()

   {

Synchronized(this){

      for(int i=1;i<=5;i++)

      {

         System.out.println(Thread.currentThread().getName());

      }

}

   }

   public static void main(String args[])

   {

      ThreadEx a=new ThreadEx();

      Thread t1=new Thread(a,"First");

      Thread t2=new Thread(a,"Second");

      Thread t3=new Thread(a,"Third");

      t1.start();

      t2.start();

      t3.start();

   }

}
```

Static Synchronization

- In java, every object has a single lock (monitor) associated with it. The thread which is entering into synchronized method or synchronized block will get that lock, all other threads which are remaining to use the shared resources have to wait for the completion of the first thread and release of the lock.
- Suppose in the case of where we have more than one object, in this case, two separate threads will acquire the locks and enter into a synchronized block or synchronized method with a separate lock for each object at the same time. To avoid this, we will use static synchronization.
- In this, we will place synchronized keywords before the static method. In static synchronization, lock access is on the class not on object and Method.

Syntax:

```
synchronized static return_type method_name (Parameters) {

//code

}
```

Program without Static Synchronization:
Example:-

```
class E implements Runnable
{
   int available=5000,amount;
   E(int amount)
   {
      this.amount=amount;
   }

   @Override
   public synchronized void run()
   {
     // System.out.println(Thread.currentThread().getName());
      String name=Thread.currentThread().getName();
      //synchronized(this){
      if(available>=amount)
      {
         System.out.println(name+" withdraw Amount");
         available=available-amount;
      }
      else
      {
         System.out.println("Sorry amount not available");
      }
      try{
         Thread.sleep(1000);
      }
      catch(InterruptedException e2)
      {

      }
      //}
   }
```

```
}
public class ThreadSynchronization {
  public static void main(String args[])
  {
   E e=new E(5000);
     Thread t1=new Thread(e);
     Thread t2=new Thread(e);
     E e1=new E(5000);
     Thread t3=new Thread(e1);
     Thread t4=new Thread(e1);
      t1.setName("Rahul");
      t2.setName("Sujan");
      t3.setName("Vishnu");
      t4.setName("Rushi");
     t1.start();
     t2.start();
     t3.start();
     t4.start();
  }
}
```

OutPut:-

```
Rahul withdraw Amount
Vishnu withdraw Amount
Sorry amount not available
Sorry amount not available
```

Program With static synchronized

Example:-

```
class E extends Thread
{
   static int available=5000,amount;
   E(int amount)
   {
      this.amount=amount;
   }

   public static synchronized void withdraw()
   {
      //System.out.println(Thread.currentThread().getName());
      String name=Thread.currentThread().getName();
      //synchronized(this){
      if(available>=amount)
      {
         System.out.println(name+" Withdraw Amount");
         available=available-amount;
      }
      else
      {
         System.out.println("Sorry amount not available");
      }
      //}
   }
```

```
    @Override
    public void run()
    {
        withdraw();
    }
}
public class ThreadSynchronization {
  public static void main(String args[])
  {
     E e=new E(5000);
     Thread t1=new Thread(e);
     Thread t2=new Thread(e);
     E e1=new E(5000);
     Thread t3=new Thread(e1);
     Thread t4=new Thread(e1);
      t1.setName("Rahul");
      t2.setName("Sujan");
      t3.setName("Vishnu");
      t4.setName("Rushi");
     t1.start();
     t2.start();
     t3.start();
     t4.start();
  }
}
```

OutPut:-

```
Rahul Withdraw Amount
Sorry amount not available
Sorry amount not available
Sorry amount not available
```

(or)

Example Program for static synchronized:-

```
public class ThreadEx implements Runnable{

   public static synchronized void display()

   {

      for(int i=1;i<=5;i++)

      {

         System.out.println(Thread.currentThread().getName());

      }

   }

Public void run()

{
```

```
This.display();
}
    public static void main(String args[])
    {
        ThreadEx a=new ThreadEx();
        Thread t1=new Thread(a,"First");
        Thread t2=new Thread(a,"Second");
        ThreadEx a1=new ThreadEx();
        Thread t3=new Thread(a1,"Third");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

6.Inter-thread Communication in Java

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

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**:

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

1) wait() method

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

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

Method	Description
public final void wait()throws InterruptedException	It waits until object is notified.
public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.

2) notify() method

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

```
1. public final void notify()
```

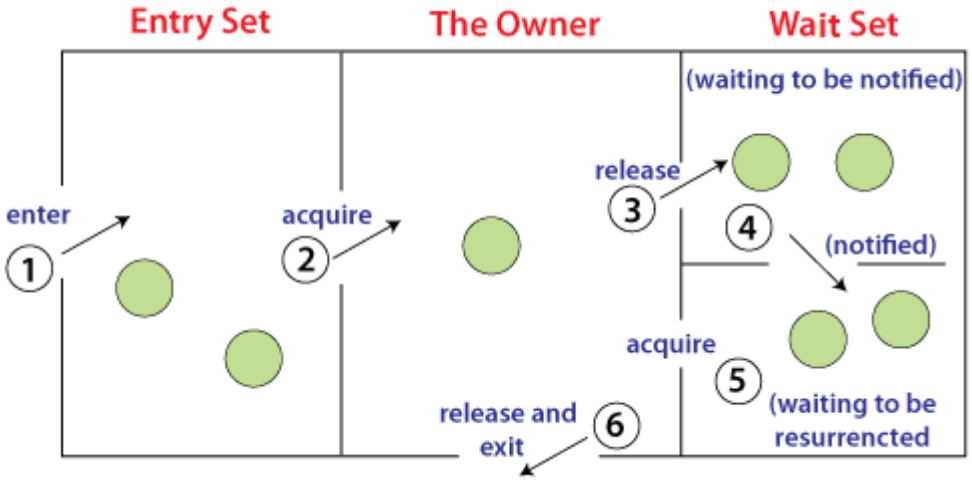
3) notifyAll() method

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

Syntax:

```
1. public final void notifyAll()
```

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.

6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	sleep()
The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method
It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is completed.

Example of Inter Thread Communication in Java

Let's see the simple example of inter thread communication.

Test.java

```
1. class Customer{
2. int account=5000;
3.
4. synchronized void withdraw(int amount){
5. System.out.println("going to withdraw...");
6.
7. if(account<amount){
8. System.out.println("Less balance; waiting for deposit...");
9. try{wait();}catch(Exception e){}
10. }
11. account-=amount;
12. System.out.println("withdraw completed...");
13. }
14.
15. synchronized void deposit(int amount){
16. System.out.println("going to deposit...");
17. account+=amount;
```

```
18. System.out.println("deposit completed... ");
19. notify();
20. }
21. }
22.
23. class Test{
24. public static void main(String args[]){
25.  Customer c=new Customer();
26. new Thread(){
27. public void run(){c.withdraw(15000);}
28. }.start();
29. new Thread(){
30. public void run(){c.deposit(15000);}
31. }.start();
32.
33. }}
```

Output:

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