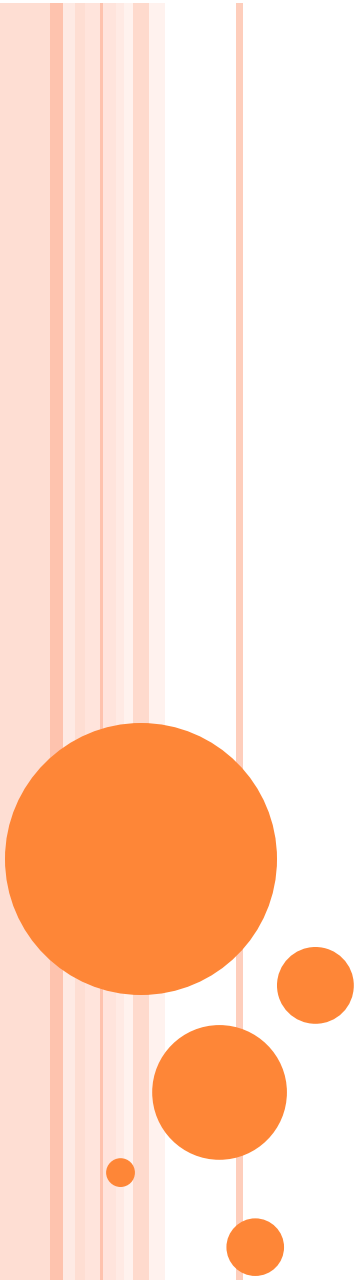


Multithreading

MULTITHREADING IN JAVA

- **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.

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

Advantages of Java Multithreading

- It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
- You **can perform many operations together, so it saves time**.
- Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

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-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 register, memory maps, updating lists, etc.

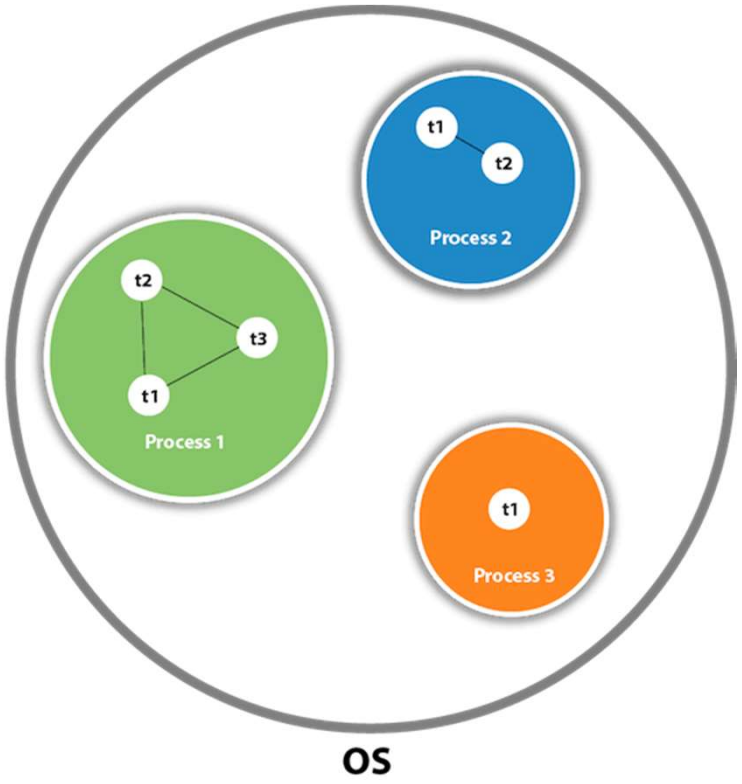
Thread-based Multitasking (Multithreading)

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

What is Thread in java

- 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 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.



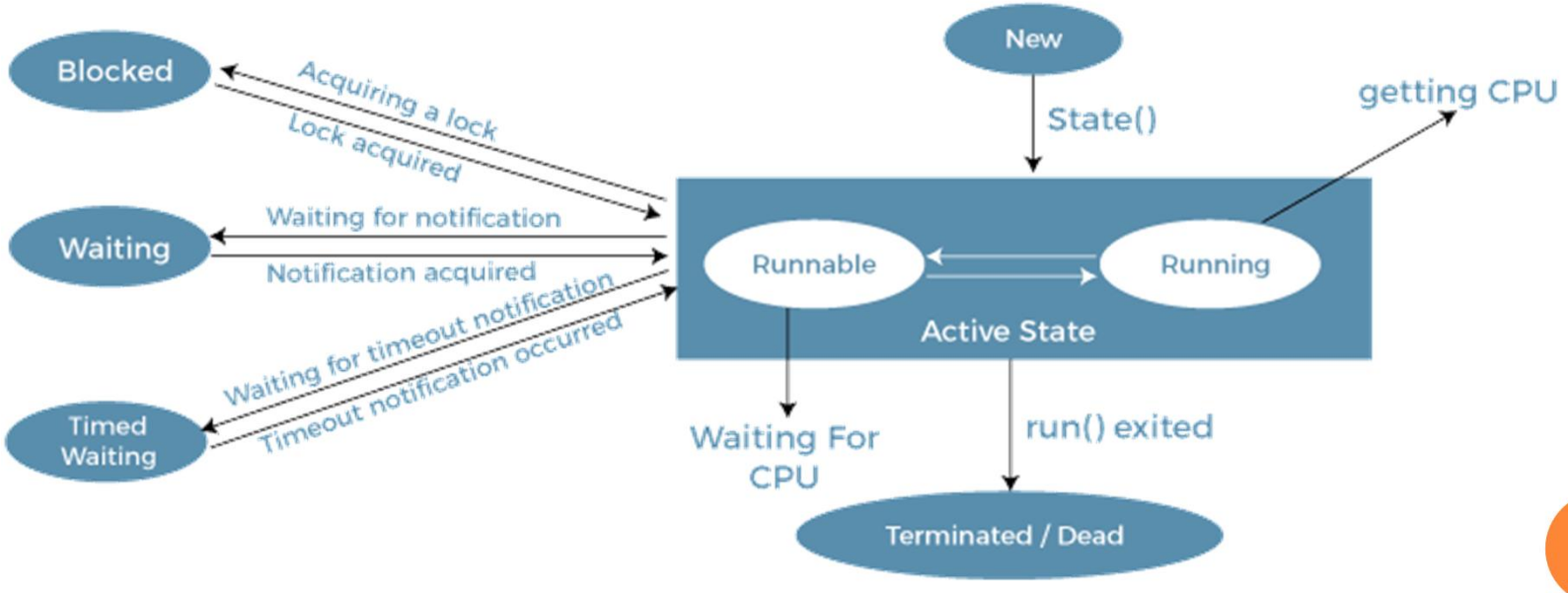
JAVA THREAD CLASS

- Java provides **Thread class** to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends object class and implements Runnable interface.

LIFE CYCLE OF A THREAD (THREAD STATES)

In Java, a thread always exists in any one of the following states. These states are:

- **New**
- **Active**
- **Blocked / Waiting**
- **Timed Waiting**
- **Terminated**



Life Cycle of a Thread

STATES OF THREADS

- **New:** Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.
- **Active:** When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **runnable**, and the other is **running**.

CONTINUE…

- **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time.
- **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

CONTINUE…

- **Blocked or Waiting:** Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

- **Terminated:** A thread reaches the termination state because of the following reasons:
  - When a thread has finished its job, then it exists or terminates normally.
  - **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.

How to create a thread in Java

- There are two ways to create a thread:
  - By extending Thread class
  - By implementing Runnable interface.

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).

CONTINUE…

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.

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().

STARTING A THREAD:

- The **start() method** of Thread class is used to start a newly created thread. It performs the 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.

JAVA THREAD EXAMPLE BY EXTENDING THREAD CLASS

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

Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable
{
   public void run()
   {
        System.out.println("thread is running...");
   }
   public static void main(String args[])
   {
        Multi3 m1=new Multi3();
        Thread t1 =new Thread(m1);
        // Using the constructor Thread(Runnable r)
        t1.start();
    }
}
```

USING THE THREAD CLASS

```
public class MyThread1
{
   public static void main(String argvs[])
   {
         Thread t= new Thread("My first thread");
         t.start();
         String str = t.getName();
         System.out.println(str);
   }
}
```

USING THE THREAD CLASS: THREAD(RUNNABLE R, STRING NAME)

```
public class MyThread2 implements Runnable
{
    public void run()
    {
            System.out.println("Now the thread is running ...");
    }
    public static void main(String argvs[])
    {
            Runnable r1 = new MyThread2();

            Thread th1 = new Thread(r1, "My new thread");
            th1.start();
            String str = th1.getName();
            System.out.println(str);
    }
}
```

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()

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.

  public final void wait()

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.

public final void notify()

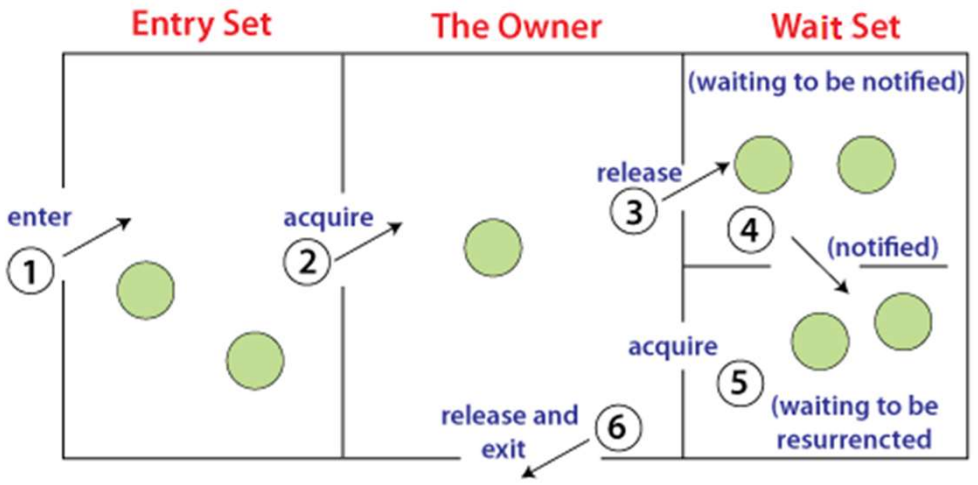
NOTIFYALL() METHOD

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

public final void notifyAll()

UNDERSTANDING THE PROCESS OF INTER-THREAD COMMUNICATION

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.



THREAD.SLEEP()

- The Java Thread class provides the two variant of the sleep() method. First one accepts only an arguments, whereas the other variant accepts two arguments. The method sleep() is being used to halt the working of a thread for a given amount of time.

  - **public static void** sleep(**long** mls) **throws** InterruptedException
  - **public static void** sleep(**long** mls, **int** n) **throws** InterruptedException

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

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

Output:

1 1 2 2 3 3 4 4

Priority of a Thread (Thread Priority)

- Each thread has a priority. Priorities are represented by a number between 1 and 10
  - 3 constants defined in Thread class
    - public static int MIN_PRIORITY
    - public static int NORM_PRIORITY
    - public static int MAX_PRIORITY

EXAMPLE OF PRIORITY OF A THREAD:

Output:
Priority of Thread: 5
Name of Thread: NewThread
Thread[NewThread,5,main]

```
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 t = new Thread(a, "NewThread");

 System.out.println("Priority of Thread: " +t.getPriority());
 System.out.println("Name of Thread: " +t.getName());
 t.start();
  }
}
```

Output:
Priority of Thread: 2
Name of Thread: NewThread
Thread[NewThread,2,main]

```
public class A implements Runnable
{
public void run()
{
  System.out.println(Thread.currentThread());
}
public static void main(String[] args)
{
 A a = new A();
 Thread t = new Thread(a, "NewThread");
 t.setPriority(2); // Setting the priority of thread.

 System.out.println("Priority of Thread: " +t.getPriority());
 System.out.println("Name of Thread: " +t.getName());
 t.start();
  }
}
```

```
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]

Synchronization in Java

- Synchronization in Java is the capability to control the access of multiple threads to any shared resource.
- Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization?

- The synchronization is mainly used to
  - To prevent thread interference.
  - To prevent consistency problem.

Types of Synchronization

- There are two types of synchronization
  - Process Synchronization
  - Thread Synchronization

Thread Synchronization

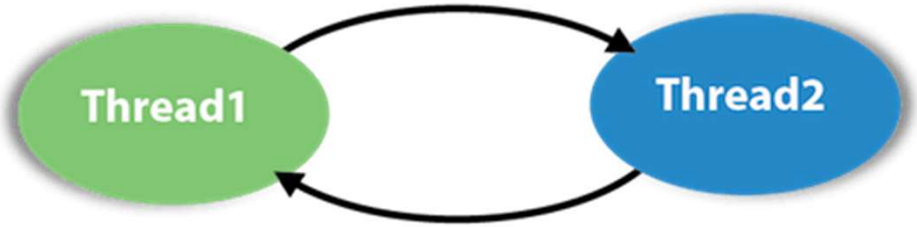
- There are two types of thread synchronization mutual exclusive and inter-thread communication.
- Mutual Exclusive
  - Synchronized method.
  - Synchronized block.
  - Static synchronization.
- Cooperation (Inter-thread communication in java)

Mutual Exclusive

- Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:
  - By Using Synchronized Method
  - By Using Synchronized Block
  - By Using Static Synchronization

Deadlock in Java

- Deadlock in Java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



JAVA THREAD STOP() METHOD

- The **stop()** method of thread class terminates the thread execution. Once a thread is stopped, it cannot be restarted by start() method.

Syntax

- **public final void** stop()
- **public final void** stop(Throwable obj)

```
public class JavaStopExp extends Thread
{
   public void run()
   {
      for(int i=1; i<5; i++)
      {
         try
         {
            // thread to sleep for 500 milliseconds
            sleep(500);
            System.out.println(Thread.currentThread().getName());
         }catch(InterruptedException e){System.out.println(e);}
         System.out.println(i);
      }
   }
   public static void main(String args[])
   {
      // creating three threads
      JavaStopExp t1=new JavaStopExp ();
      JavaStopExp t2=new JavaStopExp ();
      JavaStopExp t3=new JavaStopExp ();
      // call run() method
      t1.start();
      t2.start();
      // stop t3 thread
      t3.stop();
      System.out.println("Thread t3 is stopped");
   }
}
```

JAVA PROGRAM TO USE EXCEPTIONS WITH THREAD

- Exceptions are the events that occur due to the programmer error or machine error which causes a disturbance in the normal flow of execution of the program. When a method encounters an abnormal condition that it can not handle, an exception is thrown as an exception statement. Exceptions are caught by handlers(here catch block).

EXAMPLE

```
import java.io.*;
class GFG extends Thread
{
        public void run()
        {
                System.out.println("Thread is running");
                for (int i = 0; i < 10; ++i) {
                        System.out.println("Thread loop running " + i);
                }
        }
        public static void main(String[] args)
        {
                try {
                        GFG ob = new GFG();
                        throw new RuntimeException();
                }
                catch (Exception e)
                {
                System.out.println("Another thread is not supported");
                }
        }
}
```

Thanks