UNIT-III

3.1 Introduction to Exception

Exception:

- Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions.
- Exceptions can be caught and handled by the program.
- When an exception occurs within a method, it creates an object. This object is called the exception object.
- It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

Major reasons why an exception Occurs:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out-of-disk memory)
- Code errors
- Out of bound
- Null reference
- Type mismatch
- Opening an unavailable file
- Database errors
- Arithmetic errors

Errors: Error represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

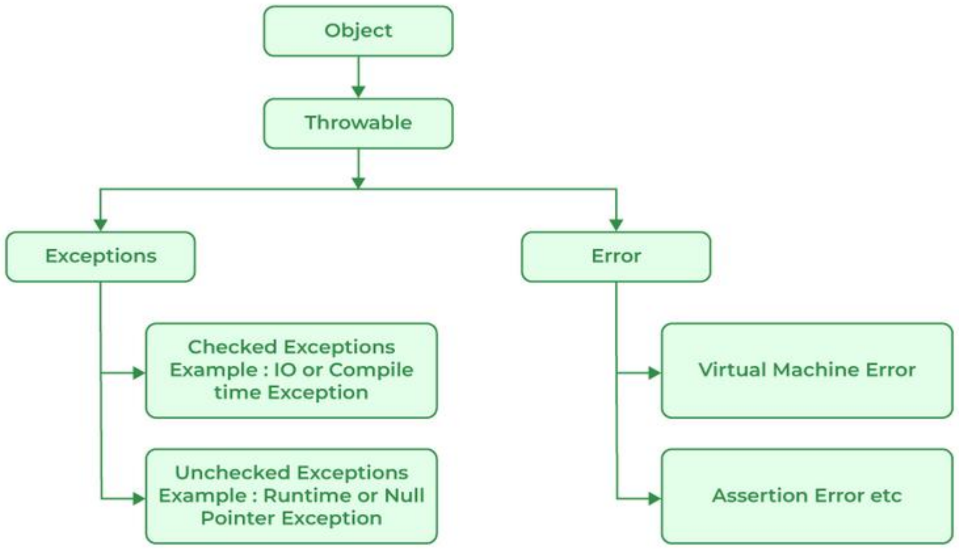
Difference between Error and Exception:

Error: An Error indicates a serious problem that a reasonable application should not try to catch.

Exception: Exception indicates conditions that a reasonable application might try to catch.

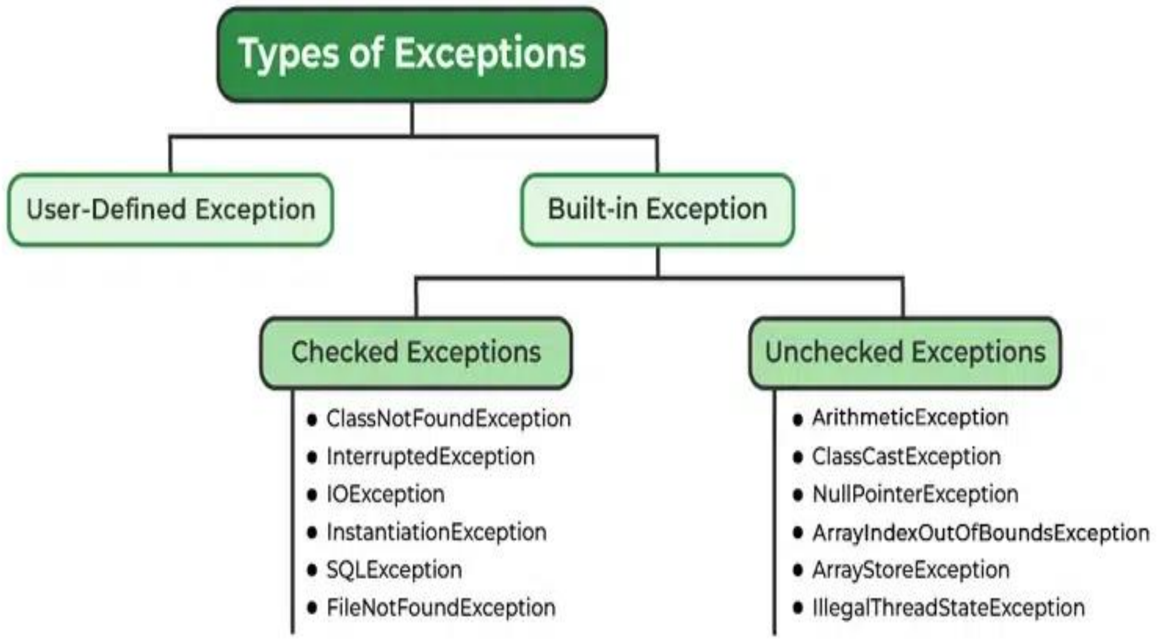
3.2 Exception Types:

- All exception and error types are subclasses of the class **Throwable**, which is the base class of the hierarchy.
- One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception.
- Another branch, **Error** is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.


Object
Throwable
Exceptions
Error
Checked Exceptions Example : IO or Compile time Exception
Unchecked Exceptions Example : Runtime or Null Pointer Exception
Virtual Machine Error
Assertion Error etc

Types of Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



1. Built-in Exceptions

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

Checked Exceptions: Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

Unchecked Exceptions: The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time.

Some of the built-in exceptions are:

1**. ArithmeticException:** It is thrown when an exceptional condition has occurred in an arithmetic operation.

2. ArrayIndexOutOfBoundsException: It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

3. ClassNotFoundException: This Exception is raised when we try to access a class whose definition is not found

4. FileNotFoundException: This Exception is raised when a file is not accessible or does not open.

5. IOException: It is thrown when an input-output operation failed or interrupted

6. InterruptedException: It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.

7. NoSuchFieldException: It is thrown when a class does not contain the field (or variable) specified

8. NoSuchMethodException: It is thrown when accessing a method that is not found.

9. NullPointerException: This exception is raised when referring to the members of a null object. Null represents nothing

10. NumberFormatException: This exception is raised when a method could not convert a string into a numeric format.

11. RuntimeException: This represents an exception that occurs during runtime.

12. StringIndexOutOfBoundsException: It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string

2. User-Defined Exceptions:

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called 'user-defined Exceptions'.

Methods to print the Exception information:

1. printStackTrace(): This method prints exception information in the format of the Name of the exception: description of the exception, stack trace.

2. toString(): The toString() method prints exception information in the format of the Name of the exception: description of the exception.

3. getMessage() : The getMessage() method prints only the description of the exception.

3.3 Exception Handling Techniques

- When an exception occurs within a method, it creates an object. This object is called the exception object.
- Exception object contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.
- By using “try”, “catch”, ”finally”, “throw”, ”throws” keywords exceptions can be handled.
- Exceptions are represented as classes in java. When there is an exception the programmer should do the following tasks:
- Let's try to understand the problem if we don't use try-catch block.

```
class Exception1
{
    public static void main(String args[])
    {
        int a,b,c,d;
        a=10;
        b=5;
        c=a/(b-5);
        d= a/(b+4);
        System.out.println(c);
        System.out.println(d);
    }
}
```

As displayed in the above example, while calculating ‘c’ value there is an exception is raised that is ‘division by zero’. So the next line that is calculating ‘d’ value is not executed. There may be 100 lines of code after exception. So all the code after exception will not be executed.

<u>Java Exception Handling techniques:</u>

try block:

- Java try block is used to enclose the code that might throw an exception. It must be used within the method.
- Java try block must be followed by either catch or finally block.
- If the programmer suspects any exception in program statements, he should write them inside try block and is called Risky code.

catch block:

- When there is an exception in try block JVM will not terminate the program abnormally JVM stores exception details in an exception stack and then JVM jumps into catch block.
- The programmer should display exception details and any message to the user in catch block is called handling code.
- Catch block does not exist without a try block, but a try block exist without a catch block.

finally block:

- Programmer should close all the files and databases by writing them inside finally block.
- Finally block is executed whether there is an exception or not or Exception handled or Not. This code is called clean-up code

Syntax:

```
try
{
    code that may throw exception (Risky code)
}
catch(ExceptionclassName object)
{
    ststements; (Handling code)
}
finally
{
    statements;(clean -up code)
}
```

Example progfram for Exception handling (try, catch, finally blocks):

```
class Exception2
{
    public static void main(String args[])
    {
        try
        {
            int a,b,c;
            a=10;
            b=5;
            c=a/(b-5);
            System.out.println(c);
        }
        catch(ArithmeticException e)
        {
            System.out.println("Please provide valid input");
            System.out.println(e.getMessage());
            System.out.println(e.toString());
            e.printStackTrace();
        }
        finally
        {
            System.out.println("Thank you very much for providing exception handling");
        }
    }
}
```

3.4 Java Multi catch block:

- A try block can be followed by one or more catch blocks.
- Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Note:

1. At a time only one exception occurs and at a time only one catch block is executed.
2. All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Example program1 for Multiple catch blocks:

```
public class Exception5
 {
      public static void main(String[] args)
      {
            try
            {
                int a[]=new int[5];
                System.out.println(a[10]);
            }
            catch(ArithmeticException e)
            {
               System.out.println("Arithmetic Exception occurs");
            }
            catch(ArrayIndexOutOfBoundsException e)
            {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
             }
            catch(Exception e)
            {
                  System.out.println("Parent Exception occurs");
            }
            System.out.println("rest of the code");
   }
}
```

Example program2 for Multiple catch blocks:

```
public class Exception6
 {
     public static void main(String[] args)
     {
          try
          {
             int a[]=new int[5];
             a[5]=30/0;
             System.out.println(a[10]);
          }
          catch(ArithmeticException e)
          {
              System.out.println("Arithmetic Exception occurs");
           }
           catch(ArrayIndexOutOfBoundsException e)
           {
               System.out.println("ArrayIndexOutOfBounds Exception occurs");
```

```
            }

catch(Exception e)
            {
               System.out.println("Parent Exception occurs");
            }
            System.out.println("rest of the code");
   }
}
```

3.5 Nested try block:

- A try block inside another try block is called as nested try block.
- For example, the inner try block can be used to handle ArrayIndexOutOfBoundsException while the outer try block can handle the ArithemeticException (division by zero).

Why use nested try block:

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
//main try block
try
{
   statement 1;
   statement 2;
   //try catch block within another try block
   try
   {
      statement 3;
      statement 4;
      //try catch block within nested try block
    }
   catch(Exception e1)
   {
        //exception message
   }
}
//catch block of parent (outer) try block
catch(Exception e2)
{
     //exception message
}
```

Example program for nested try:

```
public class NestedTry
{
     public static void main(String args[])
    {
        //outer try block
        try
        {
            //inner try block 1
            try
            {
               System.out.println("going to divide by 0");
               int b =39/0;
            }
            //catch block of inner try block 1
            catch(ArithmeticException e)
            {
                System.out.println(e);
            }
            finally
            {
                System.out.println("This is inner finally block");
            }
            int a[]=new int[5];
            System.out.println("going to assign a value out of array bounds");
            a[-1]=40;
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println(e);
        }
        finally
        {
            System.out.println("This is outer finally block");
        }
    }
}
```

3.6 throw keyword:

- The throw keyword in Java is used to explicitly throw an exception from a method or any block of code.
- We specify the exception object which is to be thrown.
- We can throw either checked or unchecked exception.
- The throw keyword is mainly used to throw custom exceptions.
- throw clause can be used to throw out user defined exceptions It is useful to create an exception object and throw it out of try block.

Syntax:

```
throw Instance
```

Example:

```
throw new ArithmeticException("/ by zero");
```

Example program for throw keyword:

```
public class Throw
{
   public static void validate(int age)
   {
      if(age<18)
      {
         //throw Arithmetic exception if not eligible to vote
         throw new ArithmeticException("Person is not eligible to vote");
      }
      else
     {
         System.out.println("Person is eligible to vote!!");
      }
   }
   public static void main(String args[])
  {
      validate(13);
      System.out.println("rest of the code...");
  }
}
```

3.7 throws keyword:

- The Java throws keyword is used to declare an exception.
- The throws keyword indicates what exception type may be thrown by a method.
- It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
- If a method throws a checked exception (e.g., IOException), it must either handle it using a try-catch block or declare it using the throws keyword.

Syntax:

```
returnType methodName(parameters) throws ExceptionType1, ExceptionType2
{
   // method body
}
```

Example program:

```
public class Example
{
     public void performTask() throws ArithmeticException
     {
        int result = 10 / 0; // This will throw ArithmeticException
      }
     public static void main(String[] args)
     {
        Example obj = new Example();
        try
        {
           obj.performTask();
        }
        catch (ArithmeticException e)
        {
           System.out.println("Caught an exception: " + e);
        }
    }
}
```

Differences between throw and throws:

Feature	throw	throws
Purpose	Used to explicitly throw an exception.	Declares exceptions a method might throw.
Placement	Inside the method body.	In the method signature.
Usage	Followed by an exception instance.	Followed by exception class names.
Example	throw new IOException();	void method() throws IOException {}

3.8 Creating your own Exception subclasses

- A custom exception is a user-defined exception created by extending the Exception or RuntimeException class.
- In Java, we can create our own exceptions that are derived classes of the Exception class.
- Creating our own Exception is known as custom exception or user-defined exception.
- Basically, Java custom exceptions are used to customize the exception according to user need.

Why Use Custom Exceptions?

1. **Specific Error Handling**: Helps identify and handle specific problems in an application.
2. **Better Debugging**: Provides detailed error messages that make debugging easier.
3. **Improved Readability**: Makes the code self-explanatory.
4. **Encapsulation of Error Logic**: Keeps error-related logic separated from other application logic.

Steps to Create a Custom Exception:

1. **Define the Class**:
   - Extend Exception for checked exceptions.
   - Extend RuntimeException for unchecked exceptions.
2. **Implement Constructors**:
   - Default constructor.
   - Constructor with a message (String).
   - Constructor with a cause (Throwable).
   - Constructor with both a message and a cause.
3. **Throw and Handle**:
   - Use the throw keyword to raise the exception.
   - Use try-catch blocks to handle it or propagate it with the throws keyword.

Example program for Custom exceptions:

```
class YoungException extends RuntimeException
{
    YoungException(String msg)
    {
        super(msg);
    }
}
class OldException extends RuntimeException
{
     OldException(String m)
    {
       super(m);
    }
}
class MyException
{
   public static void main(String[] args)
   {
      int age=Integer.parseInt(args[0]);
      if(age<18)
```

```
        {
           throw new YoungException("wait for some years to get marry");
        }
        else if(age>60)
        {
            throw new OldException("do not try for marriage");
        }
        else
        {
           System.out.println("this is right time to marry ");
         }
      }
 }
```

3.9 Multithreading Java: Thread Model

Multithreading in java is a process of executing multiple threads simultaneously. Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common 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.

Advantage of Java Multithreading

- It doesn't block the user because threads are independent and you can perform multiple operations at same time.
- You can perform many operations together so it saves time.
- Threads are independent so it doesn't affect other threads if exception occur 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 by two ways:

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

1) Process-based Multitasking (Multiprocessing)

- Each process has its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

Example:

- Writing java program.
- Listening mp3 songs
- Download any software from Internet

2)Thread-based Multitasking (Multithreading)

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

3.10 Life cycle of a Thread

- A thread is a lightweight sub process, a 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 share as a common memory area.
- Generally, all the programs have at least one thread , known as the main thread, that is provided by JVM.

Life cycle of a Thread (Thread States)

- Life Cycle of Thread in Java is basically changes the state of a thread that starts from its birth and ends on its death.
- When an instance of a thread is created and is executed by calling start() method of Thread class, the thread goes into runnable state.
- When sleep() or wait() method is called by Thread class, the thread enters into non-runnable state.
- From non-runnable state, thread comes back to runnable state and continues execution of statements.
- When the thread comes out of run() method, it dies.

The java thread states are as follows:

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

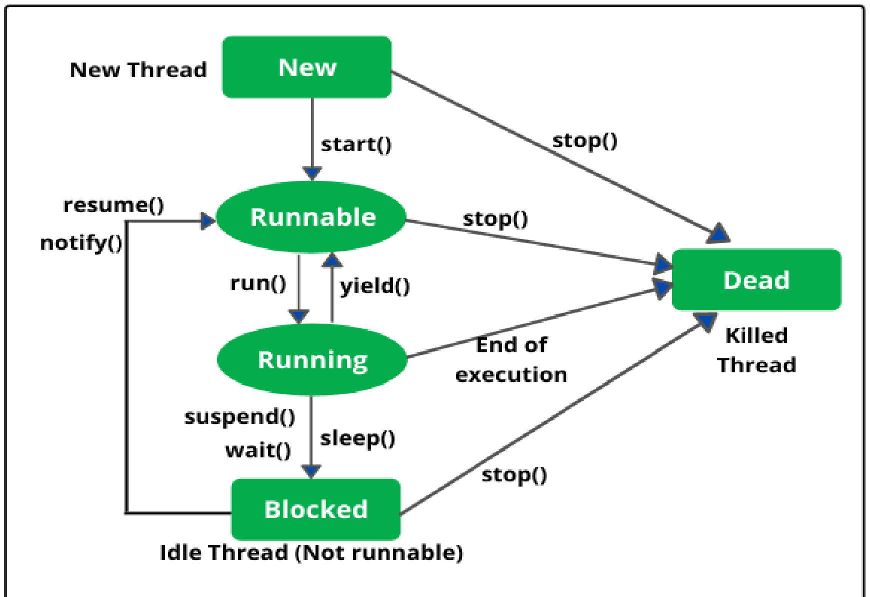
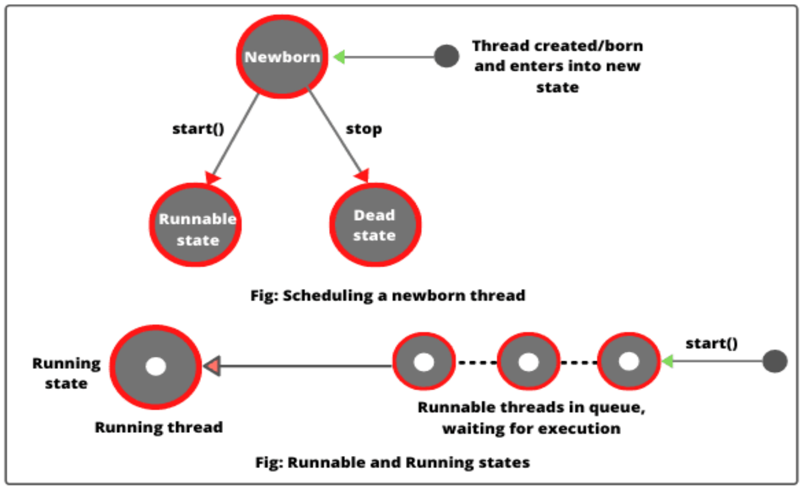
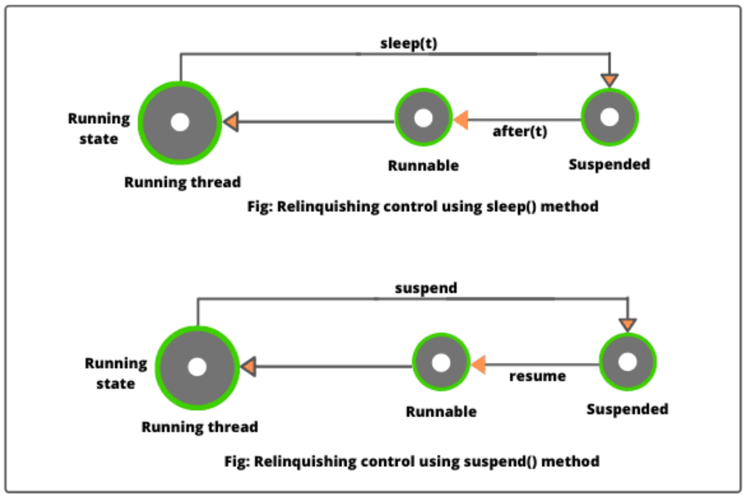


Fig: State Transition Diagram of a Thread

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). If all threads have equal priority, CPU allocates time slots for thread execution on the basis of first-come, first-serve manner. A thread can come into runnable state from running, waiting, or new states.


Newborn
Thread created/born and enters into new state
start()
stop
Runnable state
Dead state
Fig: Scheduling a newborn thread
Running state
start()
Running thread
Runnable threads in queue, waiting for execution
Fig: Runnable and Running states

3. **Running state:** Running means Processor (CPU) has allocated time slot to thread for its execution. Whenthread scheduler selects a thread from the runnable state for execution, it goes into running state. 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.



   1. When sleep() method is invoked on a thread to sleep for specified time period. The thread again reenters into the runnable state as soon as this time period is elapsed.
   2. 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.
   3. 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.

3.11 Java Thread Priorities

- Each thread has a priority.
- Priorities are represented by a number between 1 and 10.
- In most cases, the thread scheduler 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.
- Not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority:

1. **public final int getPriority():** The java.lang.Thread.getPriority() method returns the priority of the given thread.
2. **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).

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.

Example program for Thread priority:

```
import java.lang.*;
public class ThreadPriority extends Thread
{
   public void run()
   {
       System.out.println("Inside the run() method");
   }
  public static void main(String argvs[])
  {
    ThreadPriority th1 = new ThreadPriority();
    ThreadPriority th2 = new ThreadPriority();
    ThreadPriority th3 = new ThreadPriority();
    System.out.println("Priority of the thread th1 is : " + th1.getPriority());
    System.out.println("Priority of the thread th2 is : " + th2.getPriority());
    System.out.println("Priority of the thread th2 is : " + th2.getPriority());
    th1.setPriority(6);
    th2.setPriority(3);
    th3.setPriority(9);
    System.out.println("Priority of the thread th1 is : " + th1.getPriority());
    System.out.println("Priority of the thread th2 is : " + th2.getPriority());
    System.out.println("Priority of the thread th3 is : " + th3.getPriority());
    System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName());
    System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
      }
}
```

3.12 Thread creation (Runnable interface and Thread Class)

There are two ways to create a thread:

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

1.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 (Runnable r)
- Thread (String name)
- Thread (Runnable r, String name)
- Thread (ThreadGroup g, 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 mili seconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milli seconds.

4.public void join(): waits for a thread to die.

5.public void join(long mili seconds): waits for a thread to die for the specified milli seconds.

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.

Example program

```
class AThread extends Thread
{
   public void run()
   {
     for(int i=1;i<=10;i++)
     {
       System.out.println("i=" +i);
     }
   }
}
class BThread extends Thread
{
   public void run()
  {
     for(int j=11;j<=20;j++)
     {
        System.out.println("j=" +j);
     }
  }
}
public class ThreadClass
{
    public static void main(String args[])
    {
        AThread t1 = new AThread();
        t1.start();
        BThread t2 = new BThread();
        t2.start();
     }
}
```

2. Runnable interface:

Runnable is an interface that is to be implemented by a class whose instances are intended to be executed by a thread. Runnable is available in java.lang package.

Steps to create a new thread using Runnable Interface:

1. Create a Runnable implementer and implement the run() method.
2. Instantiate the Thread class and pass the implementer to the Thread, Thread has a constructor which accepts Runnable instances.
3. Invoke start() of Thread instance, start internally calls run() of the implementer. Invoking start() creates a new Thread that executes the code written in run(). Calling run() directly doesn't create and start a new Thread, it will run in the same thread.

Example program:

```
class AThread implements Runnable
{
   public void run()
   {
      for(int i=1;i<=10;i++)
```

```
        {
           System.out.println("i=" +i);
        }
    }
}
class BThread implements Runnable
{
    public void run()
    {
       for(int j=11;j<=20;j++)
       {
           System.out.println("j=" +j);
       }
   }
}
public class ThreadRunnable
{
    public static void main(String args[])
    {
         AThread ta= new AThread();
         Thread t1 = new Thread(ta);
         t1.start();
         BThread tb= new BThread();
         Thread t2 = new Thread(tb);
         t2.start();
     }
}
```

3.13 Thread Synchronization

- When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues.
- For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.
- So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time.

Why use Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusion
   - Synchronized method.
   - Synchronized block.
   - static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusion:

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method 2. by synchronized block 3. by static synchronization

1. Java synchronized method

If you declare any method as synchronized, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource. When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Example program:

```
class Display
{
  public synchronized void wish(String n)
 {
    for(int i=0;i<5;i++)
    {
       System.out.print("Good Morning:");
       try
       {
          Thread.sleep(1000);
       }
       catch(InterruptedException e) { }
       System.out.println(n);
    }
 }
}
class MyThread extends Thread
{
   Display d;
   String name;
   MyThread(Display a, String b)
   {
      d=a;
      name=b;
   }
   public void run()
   {
     d.wish(name);
   }
}
class SynchronizedMethod
{
   public static void main(String[] args)
  {
     Display o=new Display();
     MyThread t1=new MyThread(o,"AUS");
     MyThread t2=new MyThread(o,"MCA A Brain Boosters");
     MyThread t3=new MyThread(o,"Adityans");
     t1.start();
     t2.start();
     t3.start();
  }
}
```

2. Synchronized block in java

Synchronized block can be used to perform synchronization on any specific resource of the method. Suppose you have 10,000 lines of code in your method, but you want to synchronize only 10 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.

Example program:

```
class Display
{
    public void wish(String name)
    {
         System.out.println("Good Morning:1");
            System.out.println("Good Morning:2");
         System.out.println("Good Morning:3");
         System.out.println("Good Morning:4");
            ;;;;;;;;//10k lines of code
         synchronized(this)
        {
           for(int i=0;i<5;i++)
```

```
            {
                System.out.print("Good Morning:");
                try
                {
                    Thread.sleep(2000);
                 }
                 catch(InterruptedException e) {}
                 System.out.println(name);
             }
          }
       }
 }
class MyThread extends Thread
{
     Display d;
     String name;
     MyThread(Display d,String name)
     {
        this.d=d;
        this.name=name;
     }
     public void run()
     {
        d.wish(name);
      }
}
class Synchronizedblock
{
    public static void main(String[] args)
    {
       Display d=new Display();
       MyThread t1=new MyThread(d,"AUS");
          MyThread t2=new MyThread(d,"Aditya");
          MyThread t3=new MyThread(d,"MCA-A");
       t1.start();
       t2.start();
          t3.start();
    }
}
```

3. Static synchronization:

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

Example program:

```
class Display
{
    public static synchronized void wish(String name)
    {
       for(int i=0;i<5;i++)
       {
         System.out.print("Good Morning:");
         try
         {
                Thread.sleep(500);
            }
           catch(InterruptedException e) { }
           System.out.println(name);
         }
    }
}
class MyThread extends Thread
{
    Display d;
    String name;
    MyThread(Display d,String name)
    {
      this.d=d;
      this.name=name;
     }
    public void run()
    {
       d.wish(name);
     }
}
class Synchronizedstatic
{
  public static void main(String[] args)
  {
     Display d1=new Display();
     Display d2=new Display();
     MyThread t1=new MyThread(d1,"dhoni");
     MyThread t2=new MyThread(d2,"yuvaraj");
     MyThread t3=new MyThread(d1,"kohili");
     MyThread t4=new MyThread(d2,"sachin");
     t1.start();
     t2.start();
     t3.start();
     t4.start();
 }
}
```

3.14 Inter Thread Communication

- 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:
  1. wait()
  2. notify()
  3. 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.

Syntax:

public final void wait() throws InterruptedException
public final void wait(long ms) throws InterruptedException
public final void wait(long ms,int ns) throws InterruptedException

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: public final void notify()

3. notifyAll() method: Wakes up all threads that are waiting on this object's monitor.

Syntax: public final void notifyAll()

Example program:

```
class ThreadA extends Thread
{
    int total=0;
    public void run()
    {
       synchronized(this)
       {
           System.out.println("child Thread perform task");
           for(int i=1;i<=10;i++)
           {
                total=total+i;
           }
           System.out.println(" After completion of Task child Thread send notification to main Thread");
           this.notify();
       }
   }
}
class ITCDemo
{
     public static void main(String[] args)throws InterruptedException
```

```
      {
         ThreadA b=new ThreadA();
         b.start();
         synchronized(b)
         {
              System.out.println("main Thread wait() untill get notification");
              b.wait();
         }
         System.out.println(b.total);
     }
}
```