Module 3	Exception Handling and Multithreading
Exception Handling: Concepts of exception handling, Types of exceptions, Usage of try, catch,throw, throws and finally k eywords, Built-in exceptions, Creating user defined exception; **Multithreading:** Concepts of multithreading, Differences between process and thread, Threadlife cycle, Creating multiple threads using Thread class and Runnable interface, Synchronization, Thread priorities, Inter thread communication.	

Exception Handling

Exception:

- **Exception is an abnormal condition that arises in a code sequence at a run time and disrupts thenormal flow of the program.**
- It can be caught and handled by program or code.
- An object is created when an exception occurs. This object is called **exception object** and containsinformation like, name and description of exception along with the state of the program.
- There is always a reason behind an exception. Following are the reason why exception can occur inthe code:
    - Invalid input of the users
    - Failure of device
    - Weak network connection
    - Limitation of resources
    - Errors in code (syntax or logical error)
    - Opening a file that is unavailable

Types of Exceptions:

- There are two types of exceptions in java:

1. Checked exceptions (95% of exceptions in java are checked):
2. Unchecked exceptions (5% of exceptions in java are unchecked)

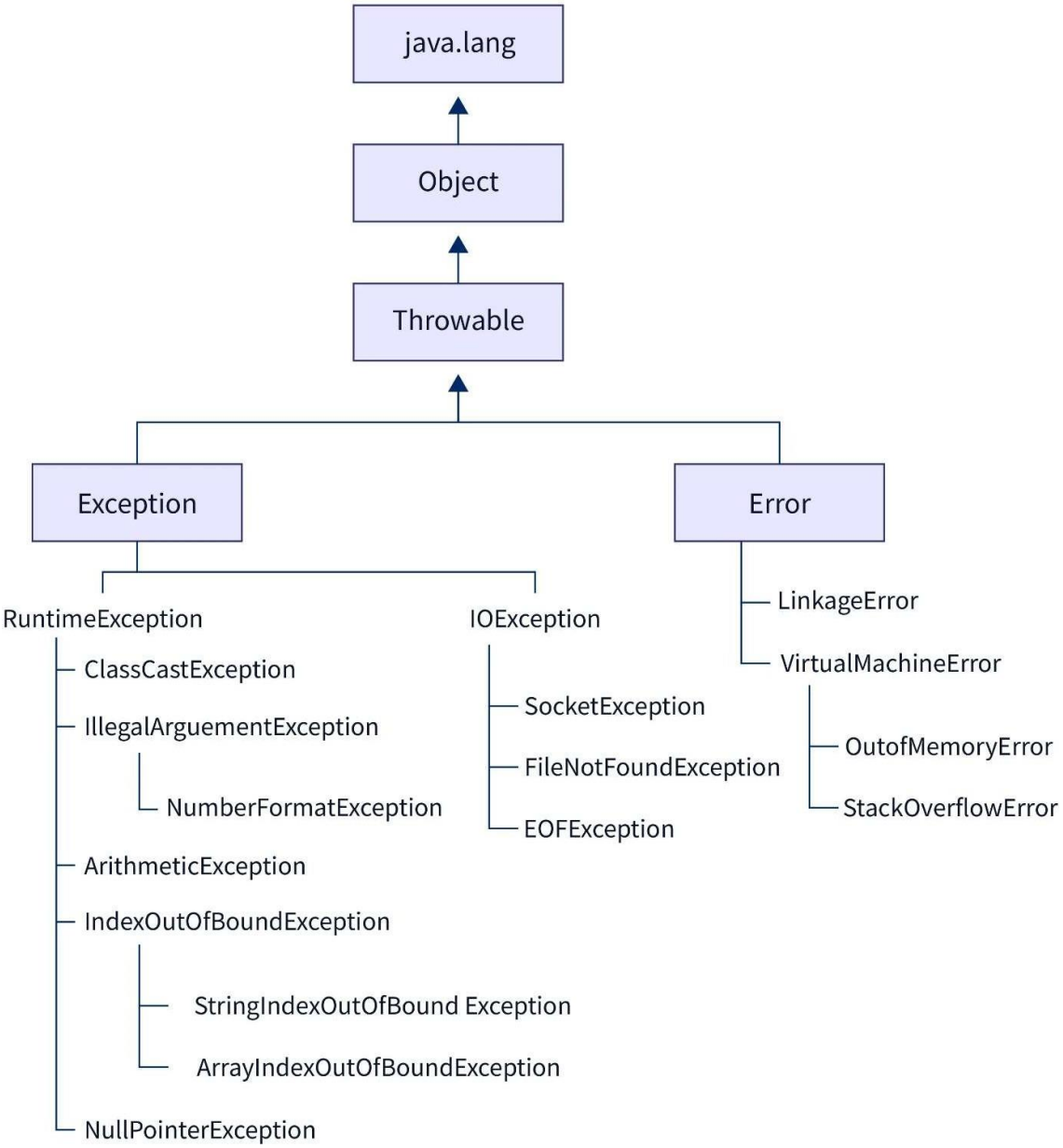
1. **Checked Exceptions**
   - The exceptions that are checked at compilation time by java compiler are called checkedexceptions.
   - In case of checked exceptions, the programmer should either handle them or throw withouthandling them.
   - Programmer cannot ignore checked exceptions as java compiler will remind of exception untiltaken care off.
   - **Example:**

Excepti on	Descripti on
IOExcepion	IO activity could not be performed.
ClassNotFoundException	Class not found exception
IIlegalAccessException	Access to a class is denied

2. **Unchecked Exceptions**
3.
   - The exceptions that are checked by the JVM at runtime are called unchecked exceptions.
   - Programmers can write a java program with unchecked exceptions and can compile theprogram.
   - Programmer can see their effect only when he/she runs the program.
   - **Example:**

Excepti on	Descripti on
ArithmeticException	Arithemetic eror, such as divide by zero
ArrayIndexOutOfBoundsExc eption	Array index is out of bound
NegativeArraySizeException	Array created with a negative size
NullPointerException	Invalid use of null reference
NumberFormatException	Invalid conversion of a string to a numericformat

Exception Hierarchy:



Object Class:

- **Object** class is present in **java.lang** package.
- Every class in Java is directly or indirectly derived from the **Object** class.
- If a class does not extend any other class, then it is a direct child class of **Object** and if extends anotherclass then it is indirectly derived.
- Therefore, the Object class methods are available to all Java classes.
- Hence Object class acts as a root of the inheritance hierarchy in any Java Program.

Throwable Class:

- The **Throwable** class is the superclass of every error and exception in the Java language.
- Only objects that are one of the subclasses this class are thrown by any "Java Virtual Machine" or may be thrown by the Java throw statement.
- For the motives of checking of exceptions during compile-time, **Throwable** and any subclass of Throwable which is not also a subclass of either Error or RuntimeException are considered as checked exceptions.
- Throwable class is the root class of Java Exception Hierarchy and is inherited by two subclasses:
    1. Exception
    2. Error

Exception Handling:

- Exception handling helps program to prevent abnormal crashing.
- Whenever an exception occurs inside a method, an exception object is created by a method and ishanded over to the JVM, this is called **default exception handling**.
- There are 5 keywords used in java exception handling:
    1. try
    2. catch
    3. finally
    4. throw
    5. throws

Keyword	Description
try	• The "try" keyword is used to specify a block where we should place anexception code. • It means we can't use try block alone. • The try block must be followed by either catch or finally.
catch	• The "catch" block is used to handle the exception. • It must be preceded by try block which means we can't use catch blockalone. • It can be followed by finally block later.
finally	• The "finally" block is used to execute the necessary code of theprogram.

throw	• It is executed whether an exception is handled or not. • The "throw" keyword is used to throw an exception.
throws	• The "throws" keyword is used to declare exceptions. • It specifies that there may occur an exception in the method. It doesn'tthrow an exception. It is always used with method signature.

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
1. int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
1. String s=null;
2. System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

```
1. String s="abc";
2. int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

```
1. int a[]=new int[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException
```

Java try-catch block

Java try block

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
try{
//code that may throw an exception
}finally{}
```

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Problem without exception handling

Let's try to understand the problem if we don't use a try-catch block.

Example 1

TryCatchExample1.java

```
public class TryCatchExample1 {

   public static void main(String[] args) {
```

```
4.
5.         int data=50/0; //may throw exception
6.
7.         System.out.println("rest of the code");
8.
9.     }
10.
11.}
```

Test it Now

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Solution by exception handling

Let's see the solution of the above problem by a java try-catch block.

Example 2

TryCatchExample2.java

```
1.  public class TryCatchExample2 {
2.      public static void main(String[] args) {
3.        try
4.        {
5.        int data=50/0; //may throw exception
6.        }
7.           //handling the exception
8.        catch(ArithmeticException e)
9.        {
10.          System.out.println(e);
11.       }
12.       System.out.println("rest of the code");
13.   }
14.
15.}
```

Test it Now

Output:

```
java.lang.ArithmeticException: / by zero
```

rest of the code

java Catch Multiple Exceptions

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.

Points to remember

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

Example 1

Let's see a simple example of java multi-catch block.

MultipleCatchBlock1.java

```
1. public class MultipleCatchBlock1 {
2.
3.    public static void main(String[] args) {
4.
5.          try{
6.              int a[]=new int[5];
7.              a[5]=30/0;
8.             }
9.             catch(ArithmeticException e)
10.              {
11.               System.out.println("Arithmetic Exception occurs");
12.              }
13.            catch(ArrayIndexOutOfBoundsException e)
14.              {
15.               System.out.println("ArrayIndexOutOfBounds Exception occurs");
16.              }
17.            catch(Exception e)
18.              {
```

```
19.              System.out.println("Parent Exception occurs");
20.             }
21.          System.out.println("rest of the code");
22.   }
23.}
```

Test it Now

Output:

```
Arithmetic Exception occurs
rest of the code
```

Example 2

MultipleCatchBlock2.java

```
1.  public class MultipleCatchBlock2 {
2.
3.     public static void main(String[] args) {
4.
5.          try{
6.              int a[]=new int[5];
7.
8.              System.out.println(a[10]);
9.             }
10.            catch(ArithmeticException e)
11.              {
12.               System.out.println("Arithmetic Exception occurs");
13.              }
14.            catch(ArrayIndexOutOfBoundsException e)
15.              {
16.               System.out.println("ArrayIndexOutOfBounds Exception occurs");
17.              }
18.            catch(Exception e)
19.              {
20.               System.out.println("Parent Exception occurs");
21.              }
22.            System.out.println("rest of the code");
23.   }
```

```
24.}
```

Test it Now

Output:

```
ArrayIndexOutOfBounds Exception occurs
rest of the code
```

.

Java Nested try block

In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

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.

NestedTryBlock.java

```
1. public class NestedTryBlock{
2.  public static void main(String args[]){
3.  //outer try block
4.   try{
5.   //inner try block 1
6.    try{
7.     System.out.println("going to divide by 0");
8.     int b =39/0;
9.    }
10.   //catch block of inner try block 1
11.   catch(ArithmeticException e)
12.   {
13.    System.out.println(e);
14.   }
15.   //inner try block 2
16.   try{
```

```
17.   int a[]=new int[5];
18.    //assigning the value out of array bounds
19.    a[5]=4;
20.    }
21.     //catch block of inner try block 2
22.   catch(ArrayIndexOutOfBoundsException e)
23.   {
24.      System.out.println(e);
25.   }
26.          System.out.println("other statement");
27.  }
28.  //catch block of outer try block
29.  catch(Exception e)
30.  {
31.   System.out.println("handled the exception (outer catch)");
32.  }
33.     System.out.println("normal flow..");
34. }
35.}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac NestedTryBlock.java

C:\Users\Anurati\Desktop\abcDemo>java NestedTryBlock
going to divide by 0
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5
other statement
normal flow..
```

Java finally block

Java finally block is a block used to execute important code such as closing the connection, etc.Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.
-

```
public class TestFinallyBlock1{
    public static void main(String args[]){

    try {

     System.out.println("Inside the try block");

     //below code throws divide by zero exception
     int data=25/0;
     System.out.println(data);
    }
    //cannot handle Arithmetic type exception
    //can only accept Null Pointer type exception
    catch(NullPointerException e){
      System.out.println(e);
    }

    //executes regardless of exception occured or not
    finally {
      System.out.println("finally block is always executed");
    }

    System.out.println("rest of the code...");

    }
  }
```

Outp

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock1.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock1
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

TestFinallyBlock2.java

```
1. public class TestFinallyBlock2{
2.     public static void main(String args[]){
3.
4.     try {
5.
6.       System.out.println("Inside try block");
7.
8.       //below code throws divide by zero exception
9.      int data=25/0;
10.     System.out.println(data);
11.    }
12.
13.    //handles the Arithmetic Exception / Divide by zero exception
14.    catch(ArithmeticException e){
15.      System.out.println("Exception handled");
16.      System.out.println(e);
17.    }
18.
19.    //executes regardless of exception occured or not
20.    finally {
21.      System.out.println("finally block is always executed");
22.    }
23.
24.    System.out.println("rest of the code...");
25.    }
26.  }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestFinallyBlock2.java

C:\Users\Anurati\Desktop\abcDemo>java TestFinallyBlock2
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...
```

java throw keyword

The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

throw Instance i.e.,

```
1. throw new exception_class("error message");
```

Let's see the example of throw IOException.

```
1. throw new IOException("sorry device error");
```

Where the Instance must be of type Throwable or subclass of Throwable. For example, Exception is the sub class of Throwable and the user-defined exceptions usually extend the Exception class.

Java throw keyword Example

Example 1: Throwing Unchecked Exception

In this example, we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

TestThrow1.java

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
1. public class TestThrow1 {
2.
3.    //function to check if person is eligible to vote or not
```

```
4.    public static void validate(int age) {
5.       if(age<18) {
6.          //throw Arithmetic exception if not eligible to vote
7.          throw new ArithmeticException("Person is not eligible to vote");
8.       }
9.       else {
10.         System.out.println("Person is eligible to vote!!");
11.      }
12.   }
13.   //main method
14.   public static void main(String args[]){
15.      //calling the function
16.      validate(13);
17.      System.out.println("rest of the code...");
18.  }
19.}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow1.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow1
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to
 vote
        at TestThrow1.validate(TestThrow1.java:8)
        at TestThrow1.main(TestThrow1.java:18)
```

```
1.  public class TestThrow {
2.     //defining a method
3.     public static void checkNum(int num) {
4.        if (num < 1) {
5.           throw new ArithmeticException("\nNumber is negative, cannot calculate square"
    );
6.        }
7.        else {
8.           System.out.println("Square of " + num + " is " + (num*num));
9.        }
10.   }
11.   //main method
```

```
12.   public static void main(String[] args) {
13.         TestThrow obj = new TestThrow();
14.         obj.checkNum(-3);
15.         System.out.println("Rest of the code..");
16.   }
17.}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow
Exception in thread "main" java.lang.ArithmeticException:
Number is negative, cannot calculate square
        at TestThrow.checkNum(TestThrow.java:6)
        at TestThrow.main(TestThrow.java:16)
```

Java throws keyword

The **Java throws keyword** is used to declare an exception. 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 the normal flow of the program can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

Syntax of Java throws

```
1. return_type method_name() throws exception_class_name{
2. //method code
3. }
```

Let's see the example of Java throws clause which describes that checked exceptions can be propagated by throws keyword.

Testthrows1.java

```
1. import java.io.IOException;
2. class Testthrows1{
3.   void m()throws IOException{
4.     throw new IOException("device error");//checked exception
5.   }
6.   void n()throws IOException{
```

```
7.      m();
8.    }
9.    void p(){
10.   try{
11.    n();
12.   }catch(Exception e){System.out.println("exception handled");}
13.  }
14.  public static void main(String args[]){
15.   Testthrows1 obj=new Testthrows1();
16.   obj.p();
17.   System.out.println("normal flow...");
18.  }
19.}
```

Test it Now

Output:

```
exception handled
normal flow...
```

Testthrows2.java

```
1.  import java.io.*;
2.  class M{
3.   void method()throws IOException{
4.    throw new IOException("device error");
5.   }
6.  }
7.  public class Testthrows2{
8.     public static void main(String args[]){
9.      try{
10.     M m=new M();
11.     m.method();
12.    }catch(Exception e){System.out.println("exception handled");
13.}
14.    System.out.println("normal flow...");
15.  }
16.}
```

Test it Now

Output:

```
exception handled
    normal flow...
```

Differences Between throw vs throws:

Sr. no.	throw	throws
1.	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
2.	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.
3.	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	throw is used within the method.	throws is used with the method signature.
5.	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Difference between final, finally and finalize

Sr. no.	Key	final	finally	finalize
1.	Definition	final is the keyword and access modifier which is used to apply restrictions on a class, method or variable.	finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not.	finalize is the method in Java which is used to perform clean up processing just before object is garbage collected.
2.	Applicable to	Final keyword is used with the classes, methods and	finally block is always related to the try and catch block in	finalize() method is used with the objects.

		variables.	exception handling.	
3.	Functionality	(1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited.	(1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block	finalize method performs the cleaning activities with respect to the object before its destruction.
4.	Execution	Final method is executed only when we call it.	finally block is executed as soon as the try-catch block is executed.	finalize method is executed just before the object is destroyed.

Built-in Exceptions

Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

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

   If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.
5. **NoSuchMethodException:**

   It is thrown when accessing a method which is not found.

User defined exception:

- Creating our own Exception is known as custom exception or user-defined exception.
- In Java, we can create our own exceptions that are derived classes of the

Exception class.

- **Create User-Defined Exceptions:**
  - To create a custom exception in java, we have to create a class by extending it with an Exceptionclass from the java.lang package.
  - Below is the template to be followed for creating a user-defined exception in java.

```
2. class InvalidAgeException  extends Exception
3. {
4.    public InvalidAgeException (String str)
5.    {
6.       // calling the constructor of parent Exception
7.       super(str);
8.    }
9. }
10.    // class that uses custom exception InvalidAgeException
11.    public class TestCustomException1
12.    {
13.
14.       // method to check the age
15.       static void validate (int age) throws InvalidAgeException{
16.          if(age < 18){
17.
18.           // throw an object of user defined exception
19.           throw new InvalidAgeException("age is not valid to vote");

20.       }
21.          else {
22.           System.out.println("welcome to vote");
23.           }
24.        }
25.
26.       // main method
27.       public static void main(String args[])
28.       {
29.          try
30.          {
31.             // calling the method
32.             validate(13);
```

```
33.         }
34.         catch (InvalidAgeException ex)
35.         {
36.            System.out.println("Caught the exception");
37.
38.            // printing the message from InvalidAgeException object

39.            System.out.println("Exception occured: " + ex);
40.         }
41.
42.         System.out.println("rest of the code...");
43.      }
44.   }
```

Output:

```
public class SimpleCustomException extends Exception{

}
```

Multithreading

Process -- Definition:

A process is a program in execution.

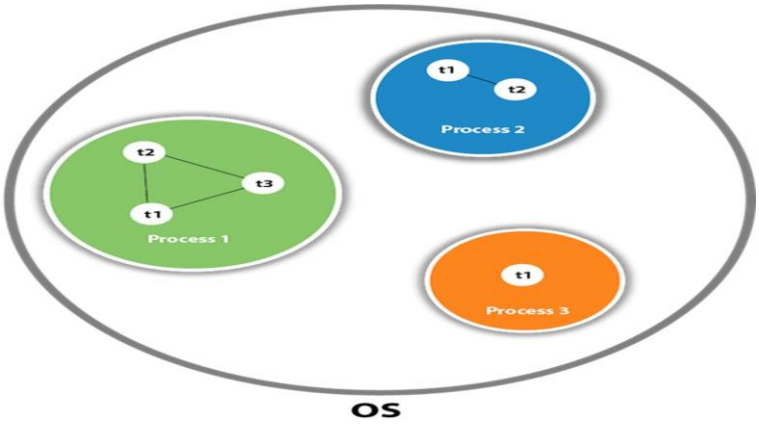
Program VS Process:

- A program is a **passive entity**, such as a file containing a list of instructions stored on disk (also called anexecutable file).
- A process is an **active entity**, with a program counter specifying the next instruction to execute and a setof associated resources.

Thread - Definition:

A thread is an **extremely lightweight process**, or the smallest component of the

process, that enablessoftware to work more effectively by doing numerous tasks concurrently.



Difference Table Between Process and Thread

Process	Thread
A process is an instance of a program that is being executed or processed.	Thread is a segment of a process or a lightweight process that is managed by the scheduler independently.
Processes are independent of each other and hence don't share a memory or other resources.	Threads are interdependent and share memory.
Each process is treated as a new process by the operating system.	The operating system takes all the user-level threads as a single process.
If one process gets blocked by the operating system, then the other process can continue the execution.	If any user-level thread gets blocked, all of its peer threads also get blocked because OS takes all of them as a single process.
Context switching between two processes takes much time as they are heavy compared to thread.	Context switching between the threads is fast because they are very lightweight.
The data segment and code segment of each process are independent of the other.	Threads share data segment and code segment with their peer threads; hence are the same for other threads also.

The operating system takes more time to terminate a process.	Threads can be terminated in very little time.

Multithreading:

Multithreading is a process of executing multiple threads simultaneously.

Advantages of Java Multithreading

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

Thread Life Cycle:

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

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

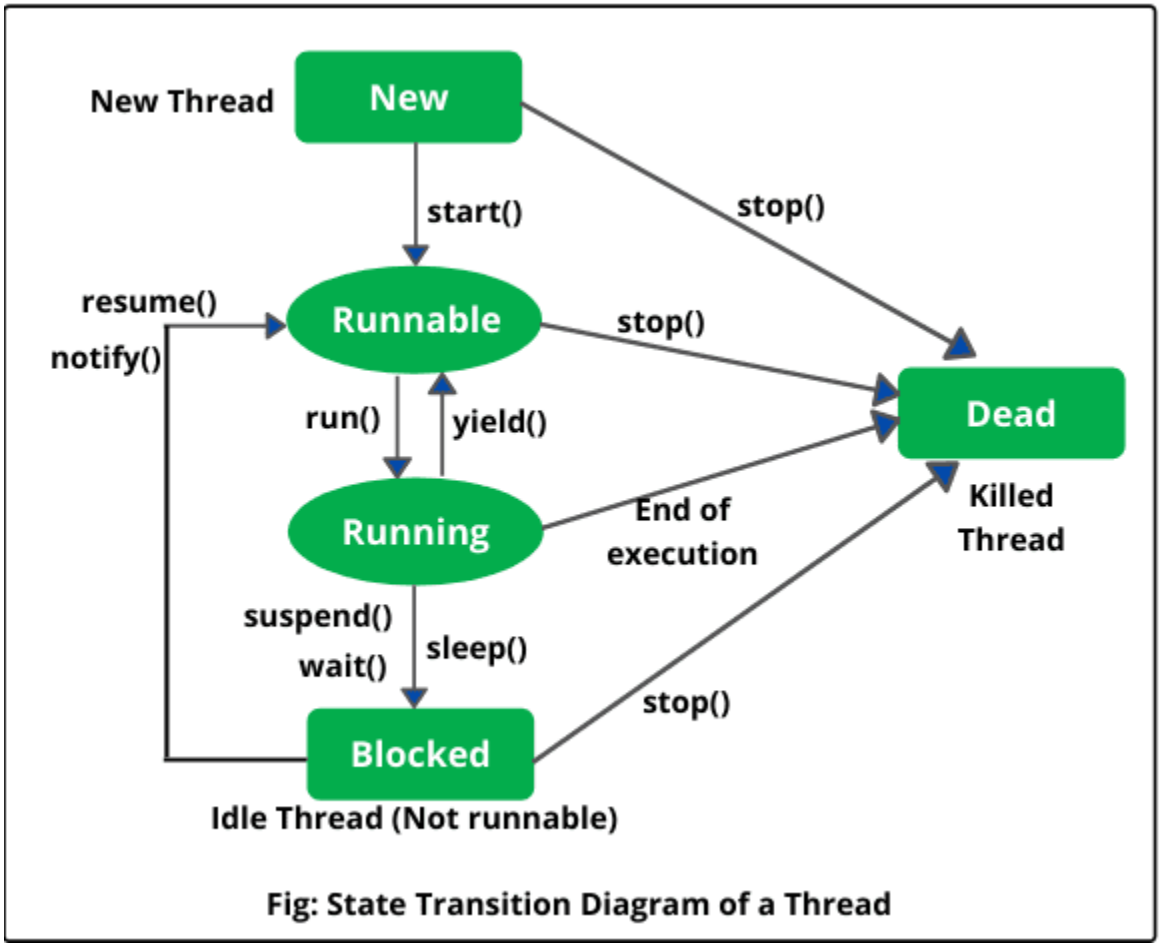


Fig: State Transition Diagram of a Thread

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.
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.

   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.

During the life cycle of thread in Java, a thread moves from one state to another state in a variety of ways. This is because in multithreading environment, when multiple threads are executing, only one thread can use CPU at a time.

All other threads live in some other states, either waiting for their turn on CPU or waiting for satisfying some conditions. Therefore, a thread is always in any of the five states.

Main Thread:

When a Java program starts up, one thread begins running immediately. This is usually called the **main thread** of your program, because it is the one that is executed when your program begins.

The main thread is important for two reasons:

1. It is the thread from which other "child" threads will be spawned.
2. Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a Thread object. To do so, you must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread. Its general form is shown here:

static Thread currentThread( )

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Let's begin by reviewing the following **example:**

```
public class MainThreadDemo {
  public static void main(String[]args)
    {
  Thread t = Thread.currentThread();
    System.out.println("CurrenThred"+t);
  }
}
```

Output:

Current Thread: Thread[main,5,main]

This displays, in order: the name of the thread, its priority, and the name of its

group. By default, the name of the main thread is main. Its priority is 5, which is the default value, and main is also the name of the group of threads to which this thread belongs.

The 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 milis**
4. **econds):** Causes the currently executing thread to sleep (temporarilycease execution) for the specified number of milliseconds.
5. **public void join():** waits for a thread to die.
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 yie**
15. **ld():** causes the currently executing thread object to temporarily pause and allowother threads to execute.
16. **public void suspend():** is used to suspend the thread(depricated).
17. **public void resume():** i8s used to resume the suspended thread(depricated).
18. **public void stop():** is used to stop the thread(depricated).

The Runnable Interface

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

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

Creating Threads:

There are two ways to create a thread:

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

Creating Thread by extending Thread Class:

We can make our class runnable as thread by extending the class Thread. This gives us access to all thethread methods directly.

Example:

Write a Java Program that creates a thread by extending Thread class.

```
1. class Multi extends Thread{
2. public void run(){
3. System.out.println("thread is running...");
4. }
5. }
6. class Multi2 extends Thread{
7. public void run(){
8. System.out.println("thread is running successfully...");
9. }
10.public static void main(String args[]){
11.Multi t1=new Multi();
12.Multi2 t2=new Multi();
13.t1.start();
14.t2.start();
15.
16. }
17.}
```

Output:

```
thread is running...
```

Creating Thread by implementing Runnable Interface:

Example:

Write a Java Program that creates a thread by extending Runnable class.

2) Java Thread Example by implementing Runnable interface

FileName: Multi3.java

```
1. class Multi3 implements Runnable{
2. public void run(){
3. System.out.println("thread is running...");
4. }
5.
6. public static void main(String args[]){
7. Multi3 m1=new Multi3();
8. Thread t1 =new Thread(m1);   // Using the constructor Thread(Runnable r)
9. t1.start();
10. }
11.}
```

Output:

```
thread is running...
```

Programs

Write a Java Program that creates a thread with a specified name, and print the name and id of thethread.

Program:

```
public class ThreadExample {
   public static void main(String[] args) {

// Create a new thread with a specified name
      Thread myThread = new Thread(new MyRunnable(), "MyThread");

      // Start the thread
```

```
        myThread.start();

        // Get and print the name and id of the thread
        long threadId = myThread.getId();
        String threadName = myThread.getName();
        System.out.println("Thread Name: " + threadName);
        System.out.println("Thread ID: " + threadId);
    }

    static class MyRunnable implements Runnable {
        @Override
        public void run() {
            // Task to be performed by the thread
            System.out.println("Thread is running...");
        }
    }
}
```

Output:

Thread Name: MyThread
Thread ID: 10
Thread is running...

Sleep()

thread Class is a class that is basically a thread of execution of the programs. It is present in Java.lang package. Thread class contains the **Sleep()** method. There are two overloaded methods of Sleep() method present in Thread Class, one is with one argument and another one is with two arguments. The sleep() method is used to stop the execution of the current thread(whichever might be executing in the system) for a specific duration of the time and after that time duration gets over, the thread which is executing earlier starts to execute again.

Program:

Write a Java Program that creates 2 threads, each thread prints numbers from 1 to 5 and also eachthread sleeps 500 milliseconds.

Description:

Use sleep( ) method to sleep a thread for the specified milliseconds of time.

Program:

```
public class ThreadExample {
    public static void main(String[] args) {
        // Create and start two threads
        Thread thread1 = new NumberPrinter("Thread 1");
```

```
        Thread thread2 = new NumberPrinter("Thread 2");
        thread1.start();
        thread2.start();
    }

    static class NumberPrinter extends Thread {
        public NumberPrinter(String name) {
            super(name);
        }

        @Override
        public void run() {
            for (int i = 1; i <= 5; i++) {
                System.out.println(getName() + ": " + i);
                try {
                    Thread.sleep(500); // Sleep for 500 milliseconds
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Output:

Thread 1: 1
Thread 2: 1
Thread 1: 2
Thread 2: 2
Thread 1: 3
Thread 2: 3
Thread 1: 4
Thread 2: 4
Thread 1: 5
Thread 2: 5

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. Note that 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

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).

- Each thread has a priority. Priorities are represented by a number between 1 and 10.
- In most cases, thread schedular schedules the threads according to their priority (known as preemptivescheduling).
- But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.
- 3 priority 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 of priority of a Thread:

FileName: ThreadPriorityExample.java

```
1. // Importing the required classes
2. import java.lang.*;
3.
4. public class ThreadPriorityExample extends Thread
5. {
6.
7. // Method 1
8. // Whenever the start() method is called by a thread
9. // the run() method is invoked
10.public void run()
```

```
11.{
12.// the print statement
13.System.out.println("Inside the run() method");
14.}
15.
16.// the main method
17.public static void main(String argvs[])
18.{
19.// Creating threads with the help of ThreadPriorityExample class
20.ThreadPriorityExample th1 = new ThreadPriorityExample();
21.ThreadPriorityExample th2 = new ThreadPriorityExample();
22.ThreadPriorityExample th3 = new ThreadPriorityExample();
23.System.out.println("Priority of the thread th1 is : " + th1.getPriority());
24.System.out.println("Priority of the thread th2 is : " + th2.getPriority());
25.  System.out.println("Priority of the thread th2 is : " + th2.getPriority());
26.  th1.setPriority(6);
27.th2.setPriority(3);
28.th3.setPriority(9);
29.  // 6
30.System.out.println("Priority of the thread th1 is : " + th1.getPriority());
31.  // 3
32.System.out.println("Priority of the thread th2 is : " + th2.getPriority());
33.  // 9
34.System.out.println("Priority of the thread th3 is : " + th3.getPriority());
35.  // Main thread
36.  // Displaying name of the currently executing thread
37.System.out.println("Currently Executing The Thread : " +
38.Thread.currentThread().getName());

39.System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriorit
    y());
40.  // Priority of the main thread is 10 now
41.Thread.currentThread().setPriority(10);
42.System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriorit
    y());
43.}
44.}
```

Output:

```
Priority of the thread th1 is : 5
Priority of the thread th2 is : 5
```

```
Priority of the thread th2 is : 5
Priority of the thread th1 is : 6
Priority of the thread th2 is : 3
Priority of the thread th3 is : 9
Currently Executing The Thread : main
Priority of the main thread is : 5
Priorityofthemainthreadis:10
```

Synchronization in Java

- Synchronization in java is the capability of control the access of multiple threads to any sharedresource.
- Java Synchronization is better option where we want to allow only one thread to access the sharedresource.
- Synchronization in Java is the process that allows only one thread at a particular time to complete a given task entirely.
- By default, the JVM gives control to all the threads present in the system to access the shared resource, due to which the system approaches race condition.

Why use Synchronization

The synchronization is mainly used to

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

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Understanding the concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire theobject's lock before accessing them, and then release the lock when it's done with them.
From Java 5 the package java.util.concurrent.locks contains several lock implementations.

Understanding the problem without Synchronization

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

Pack.java

```java
package test;

class School{
    public void Class(String teacherName) {
        for(int i=0; i<3; i++) {
            System.out.println(i+". " + teacherName );
        }
    }
}
class MyThread extends Thread{
    School sch;
    String teacherName;
    @Override
    public void run() {
        sch.Class(teacherName);
    }
    MyThread(School sch, String teacherName){
        this.sch=sch;
        this.teacherName=teacherName;
    }
}
public class Pack extends Thread {
    public static void main(String[] args) {
        School sch = new School();
        MyThread t1 = new MyThread(sch, "Raj");
        MyThread t2 = new MyThread(sch, "Rohan");
        t1.start();
        t2.start();


    }

}
```

Console

<terminated> Pack [Jav

```
0. Raj
0. Rohan
1. Raj
1. Rohan
2. Raj
2. Rohan
```

In the above code, as you can see, both the statements are running simultaneously, which creates an inconsistency in the program. Just using the synchronized keyword

before the method can easily solve this problem, as shown below.

```java
package test;

class School{
    synchronized public void Class(String teacherName) {
        for(int i=0; i<3; i++) {
            System.out.println(i+". " + teacherName );
        }
    }
}
class MyThread extends Thread{
    School sch;
    String teacherName;
    @Override
    public void run() {
        sch.Class(teacherName);
    }
    MyThread(School sch, String teacherName){
        this.sch=sch;
        this.teacherName=teacherName;
    }
}
public class Pack extends Thread {
    public static void main(String[] args) {
        School sch = new School();
        MyThread t1 = new MyThread(sch, "Raj");
        MyThread t2 = new MyThread(sch, "Rohan");
        t1.start();
        t2.start();


    }

}

```

```
<terminated> Pack [Java Application] C:\Progra
0. Rohan
1. Rohan
2. Rohan
0. Raj
1. Raj
2. Raj
```

Thread Synchronization:

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

1. Mutual Exclusive
2. Cooperation (Inter-thread communication)

1. Mutual Exclusive:

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

1. Synchronized Method
2. Synchronized Block
3. Static Synchronization

1. Synchronized method:

It is a method that can be declared synchronized using the keyword "synchronized" before the method name. By writing this, it will make the code in a method thread-safe so that no resources are shared when the method is executed.

Syntax:

```
synchronized public void methodName( ) { }
```

Synchronized Block:

If a block is declared as synchronized then the code which is written inside a method is only executedinstead of the whole code. It is used when sequential access to code is required.

Syntax:

```
synchronized (object reference)
{
  // Insert code here
}
```

Static Synchronization:

The method is declared static in this case. It means that lock is applied to the class instead of an objectand only one thread will access that class at a time.

Syntax:

```
synchronized static return_type method_name{ }
```

Inter Thread Communication

Inter thread communication in Java is a method that allows several threads to communicate with one another. It provides an effective way for multiple threads to communicate with each other by minimizing CPU idle time.

CPU idle time is a procedure that prevents CPU cycles from being wasted. When many threads are running at the same time, they may need to communicate with one another by sharing information. A thread exchanges data before or after changing its state. Communication between threads is vital in a variety of cases.

For example; Assume there are two threads: A and B. Thread B executes its task using the data developed by Thread A. Thread B would waste several CPU cycles while it waits for Thread A to create data. On the other hand, threads A and B do not have to wait and check each other's status every time they complete their tasks if they engage with each other after they have finished their tasks, As a result, you would not waste CPU cycles either.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 invokesthe 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 methodonly 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:

public final void notify( )

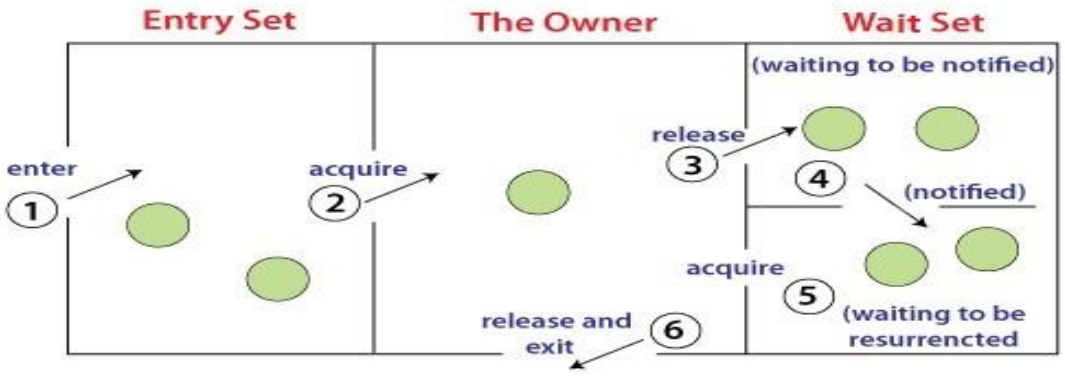
3. notifyAll() method

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

Syntax:

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 lockand 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?

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