Exception handling

Exception Handling

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.
Exception is an abnormal condition.
In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.
Exception Handling is a mechanism to handle runtime errors.

Advantage of Exception Handling

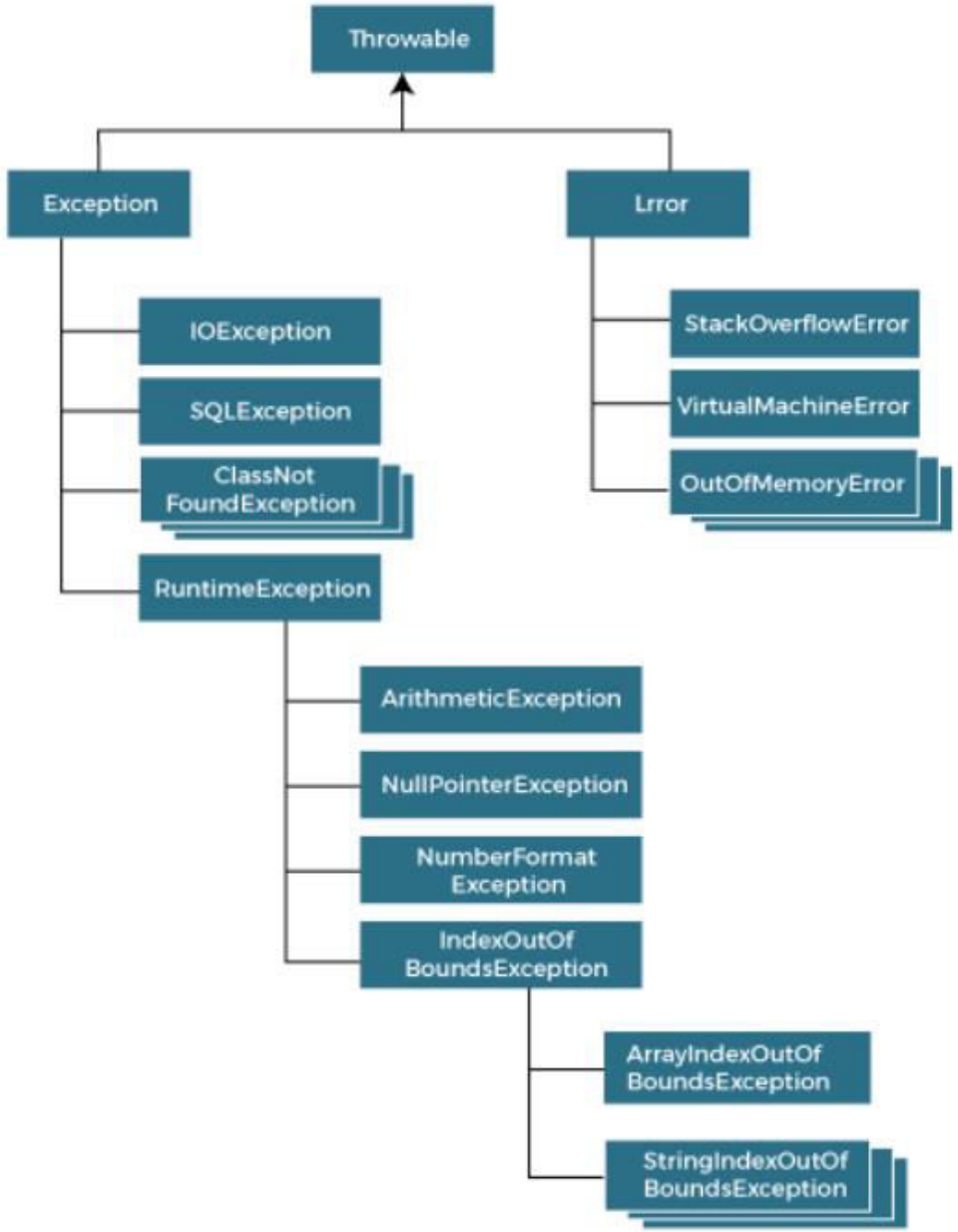
The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions.

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

Hierarchy of Java Exception classes:

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error



Types of Java Exceptions

- There are three types of exceptions namely:

1.Checked Exception

2.Unchecked Exception

3.Error

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords:

Java provides five keywords that are used to handle the exception.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception 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 block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	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't throw an exception. It is always used with method signature.

1) A scenario where ArithmeticException occurs

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

```
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.

```
String s=null;
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.

```
String s="abc";
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.

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

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

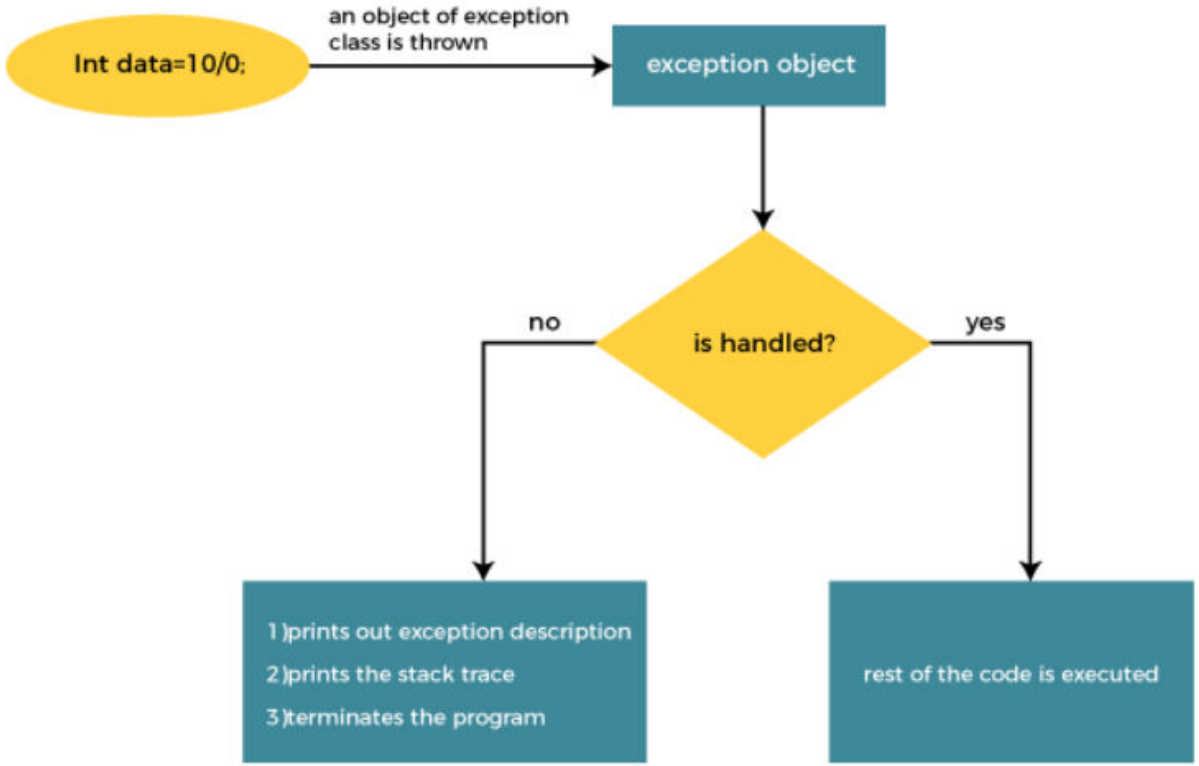
```
try{
//code that may throw an exception
}finally{}
```

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.

Internal Working of Java try-catch block



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

```java
public class TryCatchExample1 {
    public static void main(String[] args) {
        int data=50/0; //may throw exception
        System.out.println("rest of the code");
    }
}
```

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

```java
public class TryCatchExample4 {

  public static void main(String[] args) {
    try
    {
    int data=50/0; //may throw exception
    }
      // handling the exception by using Exception class
    catch(Exception e)
    {
      System.out.println(e);
    }
    System.out.println("rest of the code");
  }
}
```

```
java.lang.ArithmeticException: / by zero
rest of the code
```

```java
public class TryCatchExample2 {

  public static void main(String[] args) {
    try
    {
    int data=50/0; //may throw exception
    }
      //handling the exception
    catch(ArithmeticException e)
    {
      System.out.println(e);
    }
    System.out.println("rest of the code");
  }

}
```

```
java.lang.ArithmeticException: / by zero
rest of the code
```

```java
public class TryCatchExample8 {

   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception

      }
         // try to handle the ArithmeticException using ArrayIndexOutOfBoundsExceptio
      catch(ArrayIndexOutOfBoundsException e)
      {
         System.out.println(e);
      }
      System.out.println("rest of the code");
   }

}
```

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

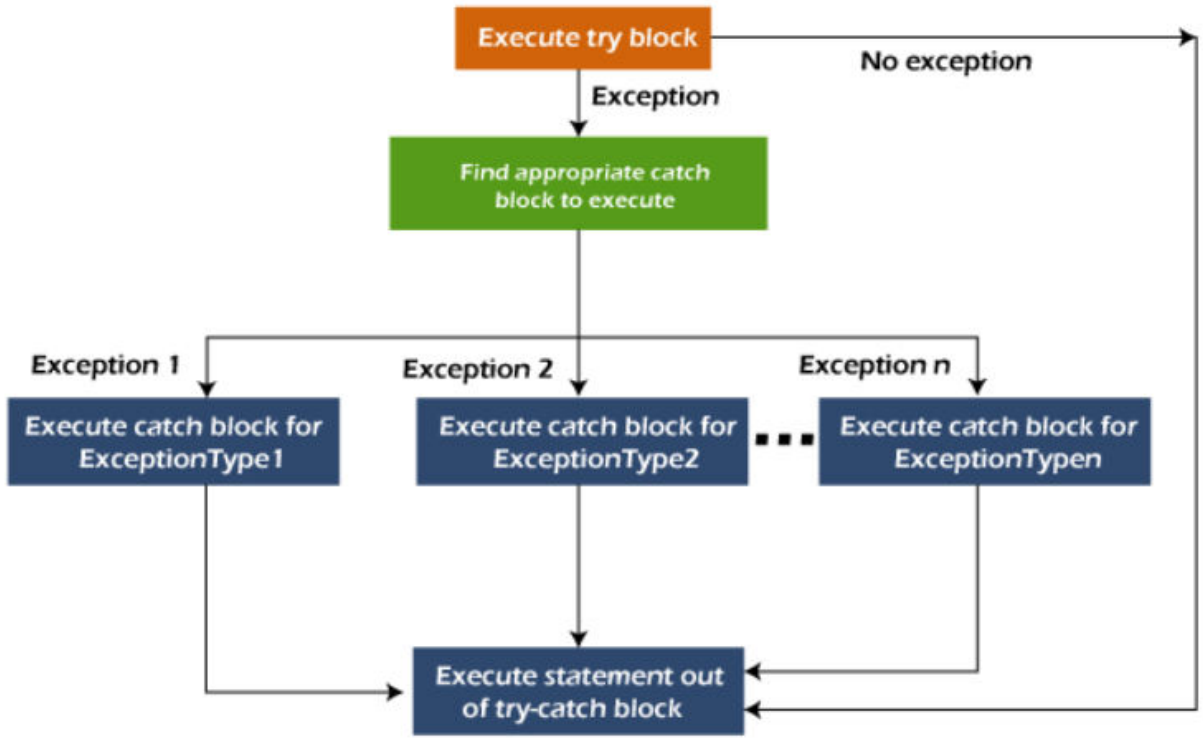
```java
public class TryCatchExample9 {

   public static void main(String[] args) {
      try
      {
      int arr[]= {1,3,5,7};
      System.out.println(arr[10]); //may throw exception
      }
         // handling the array exception
      catch(ArrayIndexOutOfBoundsException e)
      {
         System.out.println(e);
      }
      System.out.println("rest of the code");
   }
```

```
java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code
```

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.

- 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.



```java
public class MultipleCatchBlock1 {

   public static void main(String[] args) {

        try{
            int a[]=new int[5];
            a[5]=30/0;
           }
          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");
   }
}
```

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

```java
public class MultipleCatchBlock2 {

   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");
   }
}
```

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

```java
public class MultipleCatchBlock4 {

   public static void main(String[] args) {


        try{
            String s=null;
            System.out.println(s.length());
           }
           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");
   }
}
```

```
Parent Exception occurs
rest of the code
```

```
class MultipleCatchBlock5{
 public static void main(String args[]){
  try{
   int a[]=new int[5];
   a[5]=30/0;
  }
  catch(Exception e){System.out.println("common task completed");}
  catch(ArithmeticException e){System.out.println("task1 is completed");}
  catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
  System.out.println("rest of the code...");
 }
}
```

```
Compile-time error
```

```java
public class TryCatchExample3 {

   public static void main(String[] args) {
      try
      {
      int data=50/0; //may throw exception
                     // if exception occurs, the remaining statement will not exceute
      System.out.println("rest of the code");
      }
          // handling the exception
      catch(ArithmeticException e)
      {
         System.out.println(e);
      }

   }

}
```

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

if an exception occurs in the try block, the rest of the block code will not execute.

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.

```
....
//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
      try
      {
         statement 5;
         statement 6;
    }
      catch(Exception e2)
      {
//exception message
      }
   }
   catch(Exception e1)
   {
//exception message
   }
}
//catch block of parent (outer) try block
catch(Exception e3)
{
//exception message
}
....
```

```
public class NestedTryBlock{
 public static void main(String args[]){
  try{
    try{
          System.out.println("going to divide by 0");
          int b =39/0;
       }
    catch(ArithmeticException e)
       {
            System.out.println(e);
       }
      try{
          int a[]=new int[5];
          a[5]=4;
        }
      catch(ArrayIndexOutOfBoundsException e)
        {       System.out.println(e);     }
   System.out.println("other statement");
 }
   catch(Exception e)
 {     System.out.println("handled the exception (outer catch)");       }
   System.out.println("normal flow..");
 }
}
```

```
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.

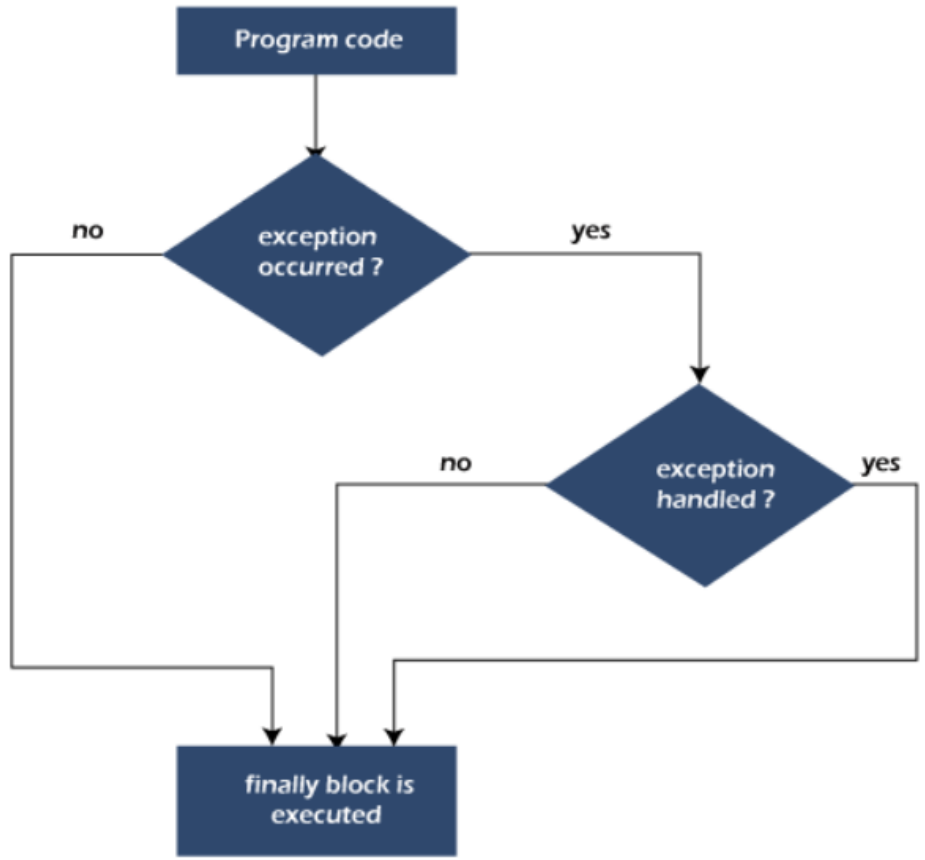
The finally block follows the try-catch block.

Why use Java finally 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.

For each try block there can be zero or more catch blocks, but only one finally block.

Flowchart of finally block



```java
class TestFinallyBlock {
  public static void main(String args[]){
  try{
//below code do not throw any exception
   int data=25/5;
   System.out.println(data);
  }
//catch won't be executed
  catch(NullPointerException e){
System.out.println(e);
}
//executed regardless of exception occurred or not
 finally {
System.out.println("finally block is always executed");
}

System.out.println("rest of phe code...");
  }
}
```

```
5
finally block is always executed
rest of the code...
```

```java
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...");
    }
```

```
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at TestFinallyBlock1.main(TestFinallyBlock1.java:9)
```

throw:
The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exceptions.

Syntax:

```
throw Instance
Example:
throw new ArithmeticException("/ by zero");
```

Instance must be of type **Throwable** or a subclass of **Throwable**. For example Exception is a sub-class of Throwable and user defined exceptions typically extend Exception class.

The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing **try** block is checked to see if it has a **catch** statement that matches the type of exception. If it finds a match, controlled is transferred to that statement otherwise next enclosing **try** block is checked and so on. If no matching **catch** is found then the default exception handler will halt the program.

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println(1/0);
    }
}
```

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

```java
class ThrowExcep
{
    static void fun()
    {
        try
        {
            throw new NullPointerException("demo");
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught inside fun().");
            throw e; // rethrowing the exception
        }
    }

    public static void main(String args[])
    {
        try
        {
            fun();
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught in main.");
        }
    }
}
```

```
Caught inside fun().
Caught in main.
```

throws

throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

- throws keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless.
- throws keyword is required only to convince compiler and usage of throws keyword does not prevent abnormal termination of program.
- By the help of throws keyword we can provide information to the caller of the method about the exception.
- **unchecked exception:** under our control so we can correct our code.
- **error:** beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

```
return_type method_name() throws exception_class_name{
//method code
}
```

```java
import java.io.IOException;
class Testthrows1{
  void m()throws IOException{
    throw new IOException("device error");//checked exception
  }
  void n()throws IOException{
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   Testthrows1 obj=new Testthrows1();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

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

Case 1: Handle Exception Using try-catch block

```java
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
public class Testthrows2{
   public static void main(String args[]){
    try{
     M m=new M();
     m.method();
    }catch(Exception e){System.out.println("exception handled");}

    System.out.println("normal flow...");
  }
}
```

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

Case 2: Declare Exception

- In case we declare the exception, if exception does not occur, the code will be executed fine.
- In case we declare the exception and the exception occurs, it will be thrown at runtime because **throws** does not handle the exception.

A) If exception does not occur

```
import java.io.*;
class M{
 void method()throws IOException{
  System.out.println("device operation performed");
 }
}
class Testthrows3{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

```
device operation performed
        normal flow...
```

B) If exception occurs

```
import java.io.*;
class M{
 void method()throws IOException{
  throw new IOException("device error");
 }
}
class Testthrows4{
   public static void main(String args[])throws IOException{//declare exception
     M m=new M();
     m.method();

    System.out.println("normal flow...");
  }
}
```

```
Exception in thread "main" java.io.IOException: device error
    at M.method(Testthrows4.java:4)
    at Testthrows4.main(Testthrows4.java:10)
```

User defined exceptions:

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.

Using the custom exception, we can have your own exception and message. Here, we have passed a string to the constructor of superclass i.e. Exception class that can be obtained using getMessage() method on the object we have created.

Java exceptions cover almost all the general type of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.

In order to create custom exception, we need to extend Exception class that belongs to java.lang package.

```
public class WrongFileNameException extends Exception {
    public WrongFileNameException(String errorMessage) {
    super(errorMessage);
    }
}
```

```
class InvalidAgeException  extends Exception
{
    public InvalidAgeException (String str)
    {
        // calling the constructor of parent Exception
        super(str);
    }
}
  // class that uses custom exception InvalidAgeException
public class TestCustomException1
{
    // method to check the age
     static void validate (int age) throws InvalidAgeExcepti
on{
  if(age < 18){
        // throw an object of user defined exception
         throw new InvalidAgeException("age is not valid to v
ote");
    }
       else {
        System.out.println("welcome to vote");
        }
     }
```

```
 // main method
    public static void main(String args[])
    {
        try
        {
            // calling the method
            validate(13);
        }
        catch (InvalidAgeException ex)
        {
            System.out.println("Caught the exception");
           // printing the message from InvalidAgeException o
bject               System.out.println("Exception occured: " + e
x);
        }
          System.out.println("rest of the code...");
    }
}
```

```
Caught the exception
Exception occured: InvalidAgeException: age is not valid to vote
rest of the code...
```

```java
// class representing custom exception
class MyCustomException extends Exception
{


}
```

```java
// class that uses custom exception MyCustomException
public class TestCustomException2
{
    // main method
    public static void main(String args[])
    {
        try
        {
            // throw an object of user defined exception
            throw new MyCustomException();
        }
        catch (MyCustomException ex)
        {
            System.out.println("Caught the exception");
            System.out.println(ex.getMessage());
        }


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

```
Caught the exception
null
rest of the code...
```

Multithreading in Java

- **Multithreading in Java** is a process of executing multiple threads simultaneously for maximum utilization of the CPU.
- A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.
- However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- In Multi-threading, multiple activities can proceed concurrently in the same program.
- Advantages of Java Multithreading

1) Threads are independent and you can perform multiple operations at the same time.

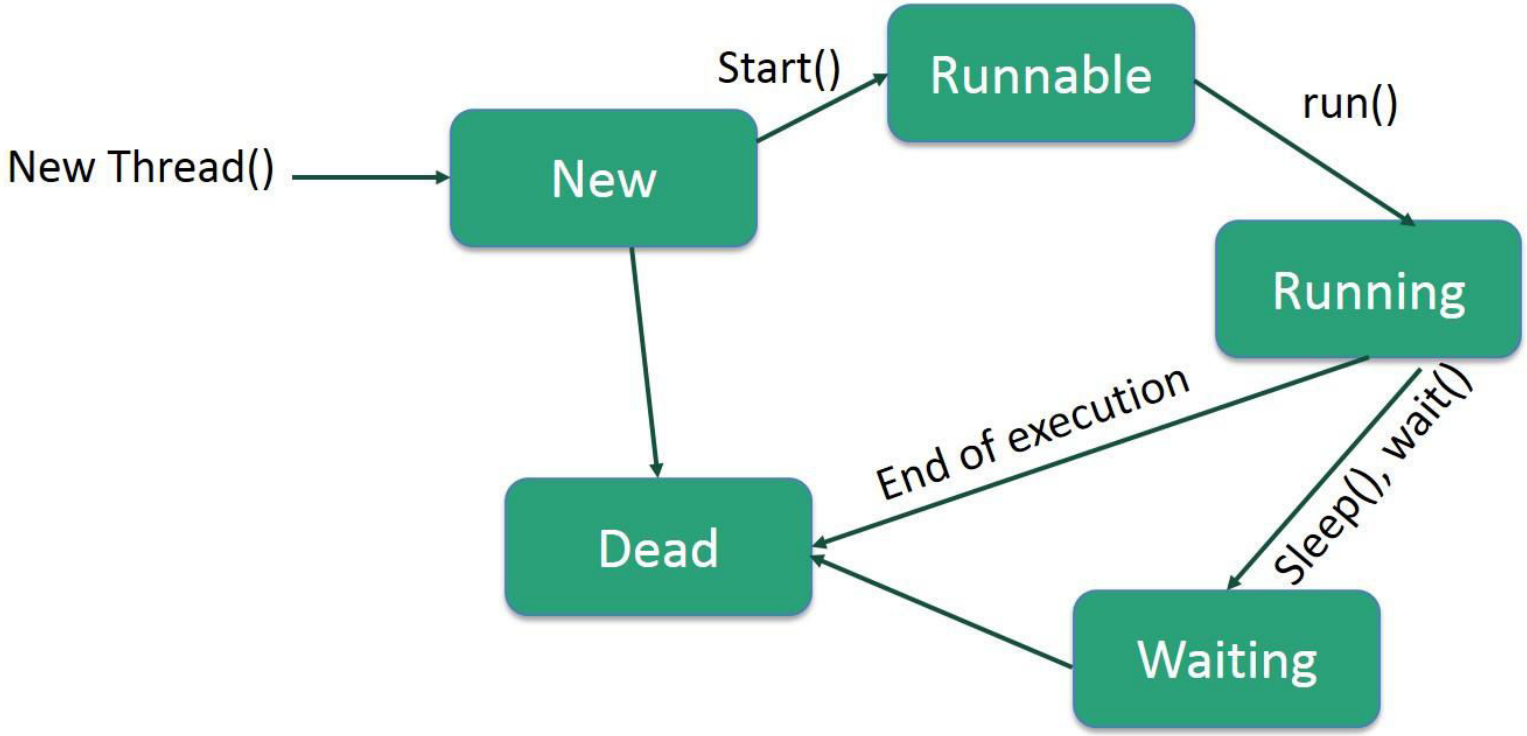
2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception

Life Cycle of a Thread

A thread goes through various stages in its life cycle. This life cycle is controlled by JVM (Java Virtual Machine). These states are:

1.New
2.Runnable
3.Running
4.Waiting
5.Terminated(Dead)



1.New: In this phase, the thread is created using class "Thread class".It remains in this state till the program **starts** the thread. It is also known as born thread.
2.Runnable: In this page, the instance of the thread is invoked with a start method. The thread control is given to scheduler to finish the execution. It depends on the scheduler, whether to run the thread.
3.Running: When the thread starts executing, then the state is changed to "running" state. The scheduler selects one thread from the thread pool, and it starts executing in the application.
4.Waiting: This is the state when a thread has to wait. As there multiple threads are running in the application, there is a need for synchronization between threads. Hence, one thread has to wait, till the other thread gets executed. Therefore, this state is referred as waiting state.
5.Dead: This is the state when the thread is terminated. The thread is in running state and as soon as it completed processing it is in "dead state".

Thread Priorities

- Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.
- Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).
- Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads.

Threads can be created by using two mechanisms :
1.Extending the Thread class
2.Implementing the Runnable Interface

Thread creation by extending the Thread class

We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.
Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

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

Runnable interface:

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

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

Starting a thread:

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

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

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

```
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();
 }
}
```

```
Output:thread is running...
```

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

```
Output:thread is running...
```

Priority of a Thread (Thread Priority):

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

3 constants defined in Thread class:

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

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

```java
class TestMultiPriority1 extends Thread{
 public void run(){
  System.out.println("running thread name is:"+Thread.currentThread().getName());
  System.out.println("running thread priority is:"+Thread.currentThread().getPriority());

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

 }
}
```

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

Synchronization in java

Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous results.

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.

Java Synchronized Method

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

•It is used to lock an object for any shared resources.

•The object gets the lock when the synchronized method is called.

•The lock won't be released until the thread completes its function.

Syntax:

```
Acess_modifiers synchronized return_type method_name (Method_Parameters) {
// Code of the Method.
}
```

```
class Table{
void printTable(int n){//method not synchr
onized
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println
(e);}     }
   }
}

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

```
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

class TestSynchronization1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
Output: 5
        100
        10
        200
        15
        300
        20
        400
        25
        500
```

```
//example of java synchronized method
class Table{
 synchronized void printTable(int n){//synchronized meth
od
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }

 }
}

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

}
```

```
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
Output: 5
        10
        15
        20
        25
        100
        200
        300
        400
        500
```

Synchronized Block in Java

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to remember for Synchronized block

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

Syntax to use synchronized block

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

```
class Table{

 void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }//end of the method
}

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

}
```

```
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

public class TestSynchronizedBlock1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
Output: 5
        10
        15
        20
        25
        100
        200
        300
        400
        500
```

Static Synchronization

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

```
class Table{
  synchronized static void printTable(int n){
   for(int i=1;i<=10;i++){
     System.out.println(n*i);
     try{
       Thread.sleep(400);
     }catch(Exception e){}
   }
 }
}
 class MyThread1 extends Thread{
public void run(){
Table.printTable(1);
}
}
class MyThread2 extends Thread{
public void run(){
Table.printTable(10);
}
}
 class MyThread3 extends Thread{
public void run(){
Table.printTable(100);
}
}
 class MyThread4 extends Thread{
public void run(){
Table.printTable(1000);
}
}
 public class TestSynchronization4{
public static void main(String t[]){
MyThread1 t1=new MyThread1();
MyThread2 t2=new MyThread2();
MyThread3 t3=new MyThread3();
MyThread4 t4=new MyThread4();
t1.start();
t2.start();
t3.start();
t4.start();
}
}
```

```
Output: 1
        2
        3
        4
        5
        6
        7
        8
        9
        10
        10
        20
        30
        40
        50
        60
        70
        80
        90
        100
        100
        200
        300
        400
        500
```

Output: 1 2 3 4 5 6 7 8 9 10 10 20 30 40 50 60 70 80 90 100 100 200 300 400 500 600 700 800 900 1000 1000 2000 3000 4000 5000 6000 7000 800

- **Inter-thread communication in Java** is a technique through which multiple threads communicate with each other.
- It provides an efficient way through which more than one thread communicate with each other by reducing CPU idle time. CPU idle time is a process in which CPU cycles are not wasted.
- When more than one threads are executing simultaneously, sometimes they need to communicate with each other by exchanging information with each other. A thread exchanges information before or after it changes its state.
- There are several situations where communication between threads is important.
- For example, suppose that there are two threads A and B. Thread B uses data produced by Thread A and performs its task.
- If Thread B waits for Thread A to produce data, it will waste many CPU cycles. But if threads A and B communicate with each other when they have completed their tasks, they do not have to wait and check each other's status every time.
- Thus, CPU cycles will not waste. This type of information exchanging between threads is called **inter-thread communication in Java**.

Inter thread communication in Java can be achieved by using three methods provided by Object class of java.lang package. They are:

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

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

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

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

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

Syntax:

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

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

Note:

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

notify() Method in Java

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

```
public final void notify()
```

notifyAll() Method in Java

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

```
public final void notifyAll()
```

```
public class A
{
int i;
 synchronized void deliver(int i)
 {
  this.i = i;
  System.out.println("Data Delivered: " +i);
 }
 synchronized int receive()
 {
  System.out.println("Data Received: " + i);
  return i;
 }
}
```

```
public class Thread1 extends Thread
{
 A obj;
 Thread1(A obj)
 {
  this.obj = obj;
 }
public void run()
{
 for(int j = 1; j <= 5; j++){
   obj.deliver(j);
  }
 }
}
```

```
public class Thread2 extends Thread
{
A obj;
Thread2(A obj)
{
 this.obj = obj;
}
public void run()
{
for(int k = 0; k <= 5; k++){
 obj.receive();
}
 }
}
```

```
public class NoCommunication
{
public static void main(String[] args)
{
A obj = new A();
Thread1 t1 = new Thread1(obj);
Thread2 t2 = new Thread2(obj);
 t1.start();
 t2.start();
 }
}
```

```
Output:
    Data Delivered: 1
    Data Delivered: 2
    Data Delivered: 3
    Data Delivered: 4
    Data Delivered: 5
    Data Received: 5
    Data Received: 5
    Data Received: 5
    Data Received: 5
    Data Received: 5
    Data Received: 5
```

without using wait() and notify() method

```
public class A
{
 int i;
 boolean flag = false; // flag will be true when data production is over.
synchronized void deliver(int i)
{
 if(flag)
 try
 {
  wait(); // Wait till a notification is received from Thread2. There will be no wastage of time.
 }
 catch(InterruptedException ie)
 {
  System.out.println(ie);
 }
   this.i = i;
   flag = true; // When data production is over, it will store true into flag.
   System.out.println("Data Delivered: " +i);
   notify(); // When data production is over, it will notify Thread2 to use it.
 }
```

```
synchronized int receive()
{
if(!flag)
try {
 wait(); // Wait till a notification is received from Thread1.
}
catch(InterruptedException ie){
 System.out.println(ie);
}
 System.out.println("Data Received: " + I);
  flag = false; // It will store false into flag when data is received.
  notify(); // When data received is over, it will notify Thread1 to produce next data.
  return i;
 }
}
```

```
public class Thread1 extends Thread
{
 A obj;
 Thread1(A obj)
 {
  this.obj = obj;
 }
public void run()
{
for(int j = 1; j <= 5; j++){
 obj.deliver(j);
  }
}}
public class Thread2 extends Thread
{
A obj;
Thread2(A obj)
{
 this.obj = obj;
}
public void run()
{
for(int k = 0; k <= 5; k++){
 obj.receive();
}
 }}
```

```
public class Communication
{
public static void main(String[] args)
{
 A obj = new A(); // Creating an object of class A.

// Creating two thread objects and pass reference variable obj as parameter to Thread1 and Thread2.
Thread1 t1 = new Thread1(obj);
Thread2 t2 = new Thread2(obj);
// Run both threads.
  t1.start();
  t2.start();
 }
}
```

Output:

```
Data Delivered: 1
Data Received: 1
Data Delivered: 2
Data Received: 2
Data Delivered: 3
Data Received: 3
Data Delivered: 4
Data Received: 4
Data Delivered: 5
Data Received: 5
```