Unit :- 2

Exception Handling

Exception Handling: The Idea behind Exception, Exceptions & Errors, Types of Exception, Control Flow in Exceptions, JVM Reaction to Exceptions, Use of try, catch, finally, throw, throws in Exception Handling, In-built and User Defined Exceptions, Checked and Un-Checked Exceptions.

An exception is **something that is left out or not done on purpose**. An exception to a rule does not follow that rule.

In Java "**an event that occurs during the execution of a program that disrupts the normal flow of instructions**" is called an exception. This is generally an unexpected or unwanted event which can occur either at compile-time or run-time in application code.

The Idea Behind Exceptions

An exception usually signals an error and is so called because errors in your Java programs are bound to be the exception rather than the rule—by definition! An exception doesn't always indicate an error though—it can also signal some particularly unusual event in your program that deserves special attention.

If you try to deal with the myriad and often highly unusual error conditions that might arise in the midst of the code that deals with the normal operation of the program, your program structure will soon become very complicated and difficult to understand. One major benefit of having an error signaled by an exception is that it separates the code that deals with errors from the code that is executed when things are moving along smoothly. Another positive aspect of exceptions is that they provide a way of enforcing a response to particular errors. With many kinds of exceptions, you must include code in your program to deal with them; otherwise, your code will not compile.

Errors V/s Exceptions

In Short errors and exceptions **represent different types of problems that can occur during program execution**. Errors are usually caused by serious problems that cannot be recovered from, while exceptions are used to handle recoverable errors within a program.

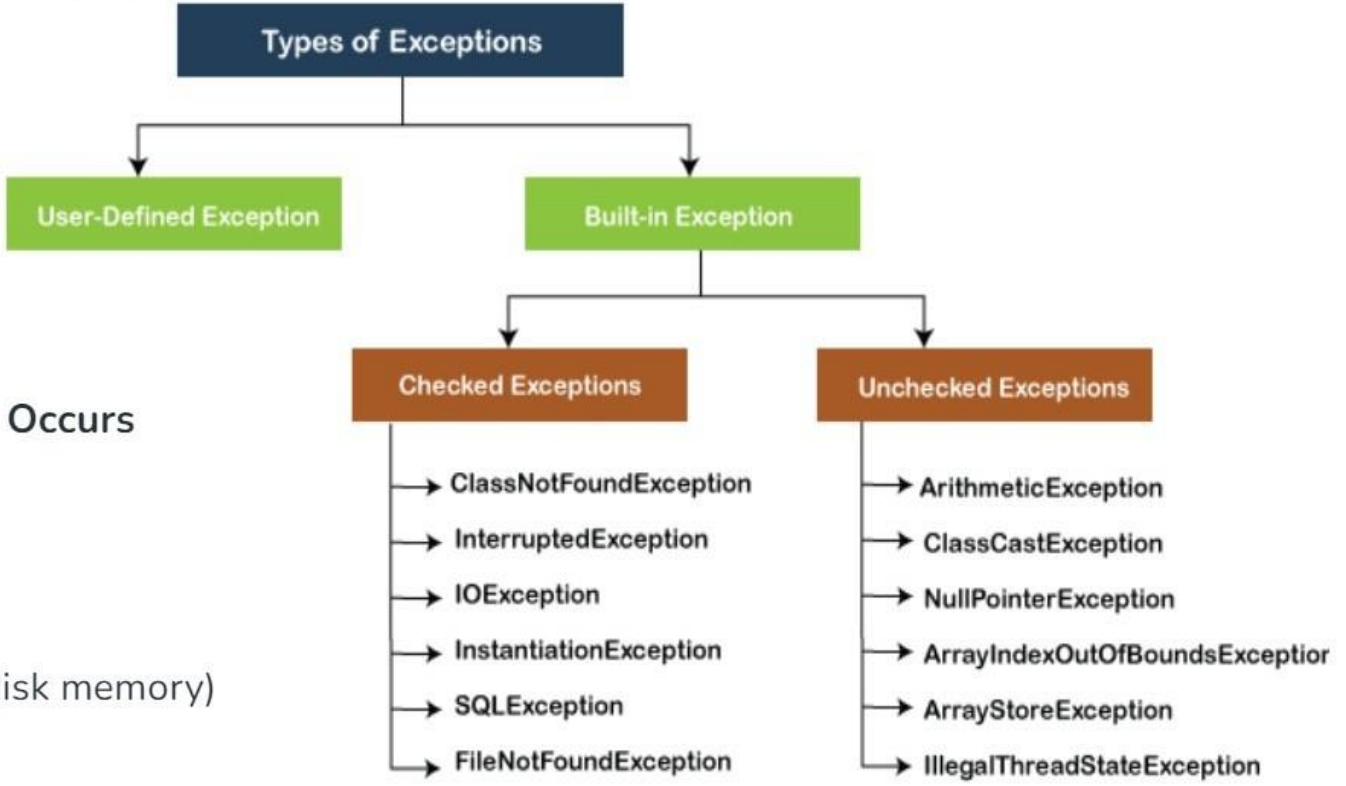
Types of Exception

Java defines several types of exceptions that relate to its various class libraries.

Java also allows users to define their own exceptions.

Exceptions can be categorized into two ways:

1. Built-in Exceptions
    - Checked Exception
    - Unchecked Exception
2. User-Defined Exceptions

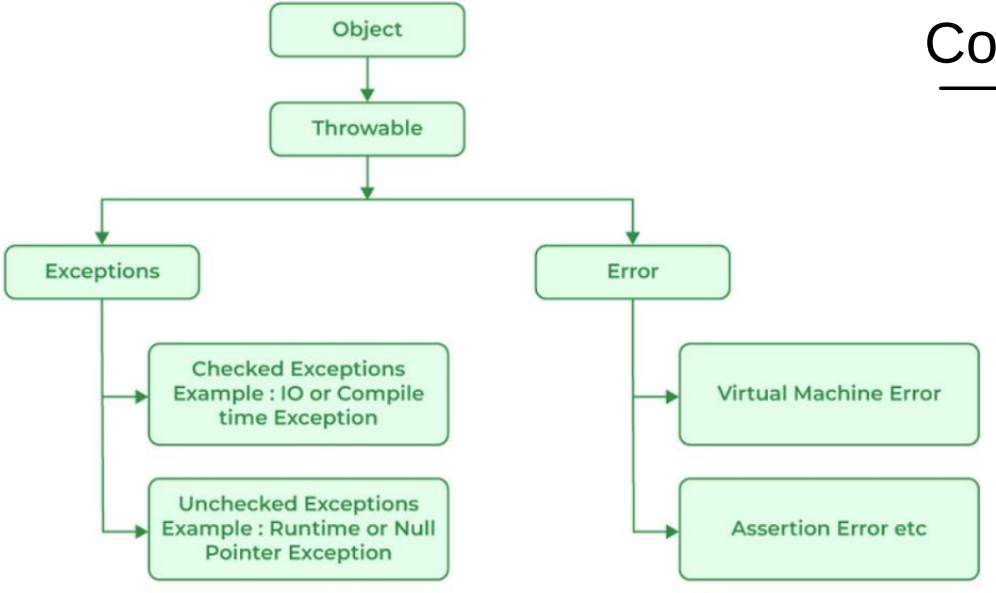


Major reasons why an exception Occurs

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out-of-disk memory)
- Code errors
- Opening an unavailable file

Exception Hierarchy

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



Control Flow in Exceptions

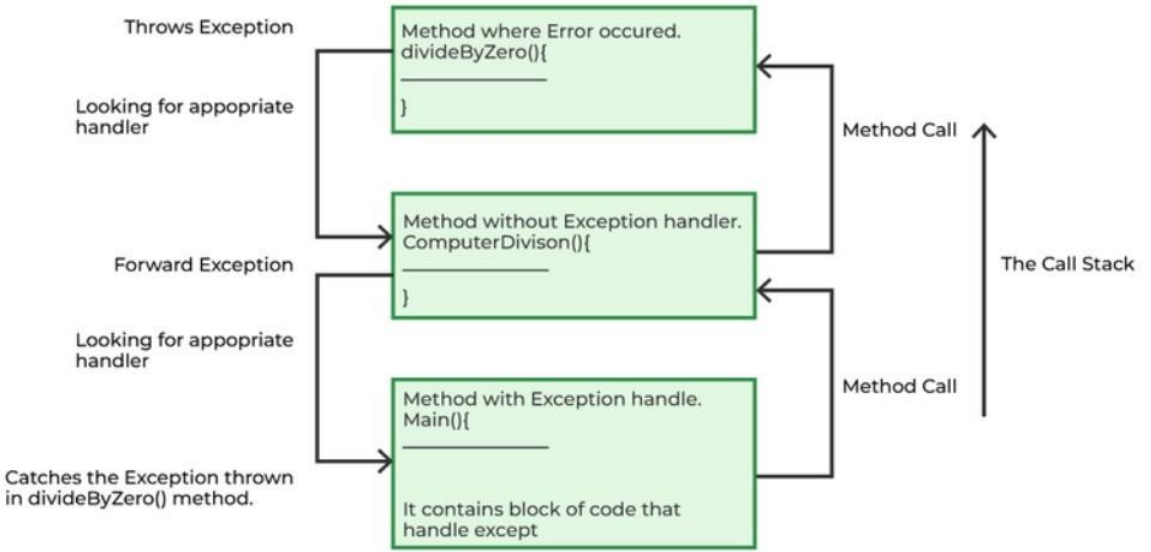
```
public class TryCatchExample1 {

  public static void main(String[] args) {

    int data=50/0; //may throw exception

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

  }

}
```

Output:

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

JVM Reaction to Exceptions

The JVM starts the search from the method where the exception occurred and then goes up the call stack, checking each method in turn for a catch block that can handle the exception. If a catch block is found, the JVM transfers control to that block and the exception is considered to be handled.



The Call Stack and searching the call stack for exception handler.

Exception Handling

In Exception handling , we should have an alternate source through which we can handle the Exception.

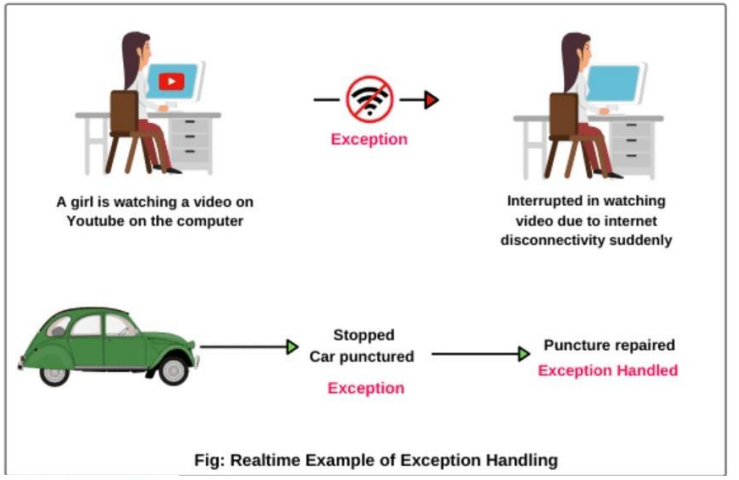


Fig: Realtime Example of Exception Handling

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. Let's consider a scenario:

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

Suppose 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.

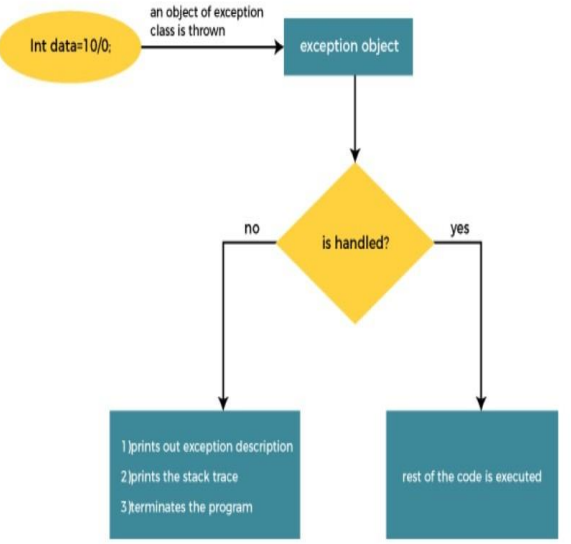
Java Exception Keywords

Try :

a. Java uses keyword "try" to preface a block of code that is likely to cause an error condition and "throw" an exception.

b. The try block can have one or more statements that could generate an exception.

```
try
{
        statement ;  // generates an exception
}
catch (Exception_type e)
        statement ;  // processes the exception
}
```




```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");
  }

}
```

c. If any one statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch block that is placed next to the try block.

d. Every try statement should be followed by at least one catch statement; otherwise compilation error will occur.

Output:

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

Catch :

a. A catch block defined by the keyword "catch" catches the exception thrown by the try block and handles it appropriately. The catch block is added immediately after the try block.

b. The catch block can have one or more statements that are necessary to process the exception.

c. The catch statement is passed as a single parameter, which is reference to the type of exception object thrown by the try block.

d. If the catch parameter matches with the type of exception object, then the exception is caught and statements in the catch block will be executed.

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.

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).

```
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");
   }
}
```

Output:

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

Finally :

a. Java supports another statement known as finally statement that can be used to handle an exception that is not caught by any of the previous catch statements.

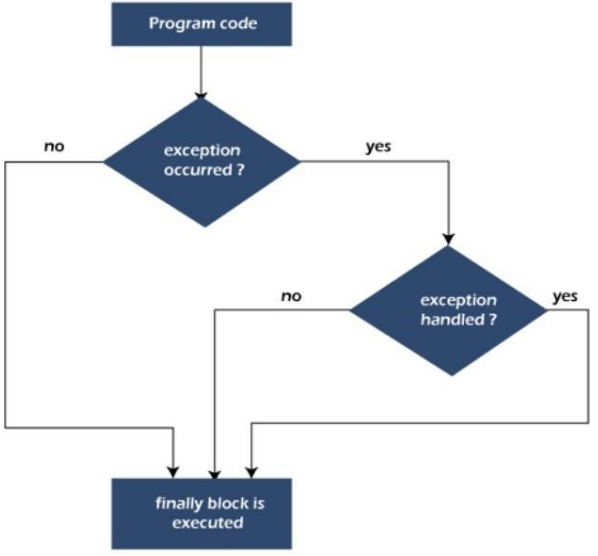
b. Finally block can be used to handle any exception generated within a try block.

c. It may be added immediately after the try block or after the last catch block as follows :

```
try
{
      .....

}
finally
{
      .....
}
```

```
try
{
      .....

}
catch (....)
{
      .....
}
catch (....)
{
      .....
}
:
finally
{
      .....
      .....
}
```

Flowchart of finally block




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

Output:

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

Throw :

a. Java supports "throw keyword" which is used if we want to throw our own exceptions.

b. We can do this by using the keyword throw as follows :

throw new Throwable_subclass;

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.

Difference between throw and throws :

S. No.	Throw	Throws
1.	Throw keyword is used to throw an exception explicitly.	Throws clause is used to declare an exception.

2.	General form of throw is :
	Throw throwable instance;
		General form of throws is :
		type method_name(parameter-list) throws exception-list
		{
		// body of method
		}

For example :

```
public class TestThrow1 {
  //function to check if person is eligible to vote or not
  public static void validate(int age) {
    if(age<18) {
      //throw Arithmetic exception if not eligible to vote
      throw new ArithmeticException("Person is not eligible to vote");
    }
    else {
      System.out.println("Person is eligible to vote!!");
    }
  }
  //main method
  public static void main(String args[]){
    //calling the function
    validate(13);
    System.out.println("rest of the code...");
  }
}
```

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)
```

The below keywords are used in Java for Exception handling.

Keyword	Description
Try	We specify the block of code that might give rise to the exception in a special block with a “Try” keyword.
Catch	When the exception is raised it needs to be caught by the program. This is done using a “catch” keyword. So a catch block follows the try block that raises an exception. The keyword catch should always be used with a try.
Finally	Sometimes we have an important code in our program that needs to be executed irrespective of whether or not the exception is thrown. This code is placed in a special block starting with the “Finally” keyword. The Finally block follows the Try-catch block.
Throw	The keyword “throw” is used to throw the exception explicitly.
Throws	The keyword “Throws” does not throw an exception but is used to declare exceptions. This keyword is used to indicate that an exception might occur in the program or method.

Built-in Exception

Exceptions that are already available in **Java libraries** are referred to as **built-in exception**. These exceptions are able to define the error situation so that we can understand the reason of getting this error. It can be categorized into two broad categories, i.e., **checked exceptions** and **unchecked exception**.

Checked Exception

Checked exceptions are called **compile-time** exceptions because these exceptions are checked at compile-time by the compiler. The compiler ensures whether the programmer handles the exception or not. The programmer should have to handle the exception; otherwise, the system has shown a compilation error.

Unchecked Exceptions

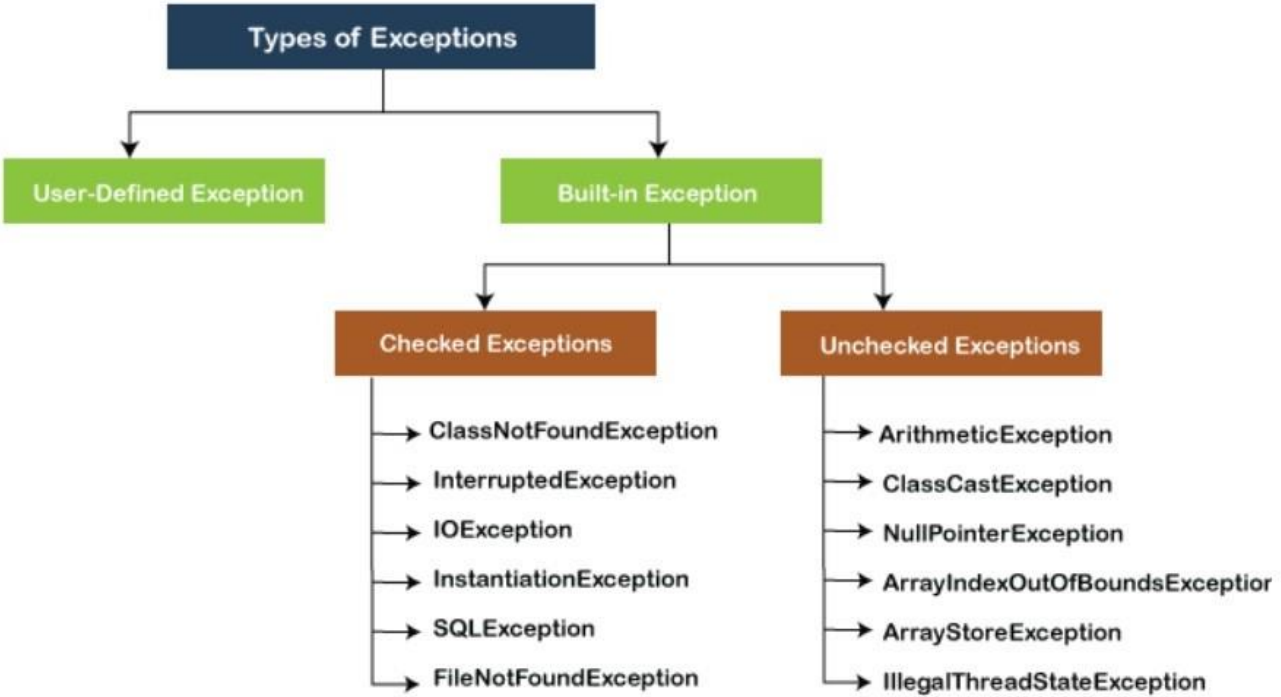
The **unchecked** exceptions are just opposite to the **checked** exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error. Usually, it occurs when the user provides bad data during the interaction with the program.

User-defined Exception

In Java, we already have some built-in exception classes like **ArrayIndexOutOfBoundsException**, **NullPointerException**, and **ArithmeticException**. These exceptions are restricted to trigger on some predefined conditions. In Java, we can write our own exception class by extends the Exception class. We can throw our own exception on a particular condition using the throw keyword. For creating a user-defined exception, we should have basic knowledge of **the try-catch** block and **throw** keyword.

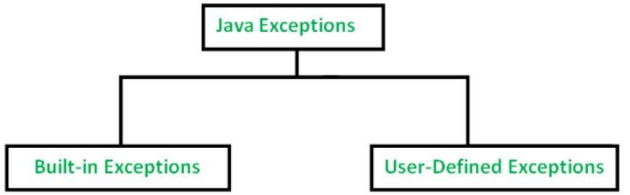
Exceptions can be categorized into two ways:

1. Built-in Exceptions
    - Checked Exception
    - Unchecked Exception
2. User-Defined Exceptions



Built-in Exceptions:

Built-in exceptions are the exceptions that 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. **ArrayIndexOutOfBoundsException:** It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **ClassNotFoundException:** This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException:** This Exception is raised when a file is not accessible or does not open.
5. **IOException:** It is thrown when an input-output operation failed or interrupted
6. **InterruptedException:** It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.

7. **NoSuchFieldException:** It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException:** It is thrown when accessing a method that is not found.
9. **NullPointerException:** This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException:** This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException:** This represents an exception that occurs during runtime.
12. **StringIndexOutOfBoundsException:** It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string
13. **IllegalArgumentException :** This exception will throw the error or error statement when the method receives an argument which is not accurately fit to the given relation or condition. It comes under the unchecked exception.
14. **IllegalStateException :** This exception will throw an error or error message when the method is not accessed for the particular operation in the application. It comes under the unchecked exception.

User-Defined Exceptions

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

The following steps are followed for the creation of a user-defined Exception.

- The user should create an exception class as a subclass of the Exception class. Since all the exceptions are subclasses of the Exception class, the user should also make his class a subclass of it. This is done as:

```
class MyException extends Exception
```

- We can write a default constructor in his own exception class.

```
MyException(){}
```

- We can also create a parameterized constructor with a string as a parameter.
  We can use this to store exception details. We can call the superclass(Exception) constructor from this and send the string there.

```
MyException(String str)
{
   super(str);
}
```

- To raise an exception of a user-defined type, we need to create an object to his exception class and throw it using the throw clause, as:

```
MyException me = new MyException("Exception details");
throw me;
```

- The following program illustrates how to create your own exception class MyException.

```
// This program throws an exception whenever balance
// amount is below Rs 1000
class MyException extends Exception
{
    //store account information
    private static int accno[] = {1001, 1002, 1003, 1004};

    private static String name[] =
                {"Nish", "Shubh", "Sush", "Abhi", "Akash"};

    private static double bal[] =
         {10000.00, 12000.00, 5600.0, 999.00, 1100.55};

    // default constructor
    MyException() {    }

    // parameterized constructor
    MyException(String str) { super(str); }

    // write main()
    public static void main(String[] args)
    {
        try  {

   // display the heading for the table
   System.out.println("ACCNO" + "\t" + "CUSTOMER" +
                                   "\t" + "BALANCE");

        // display the actual account information
        for (int i = 0; i < 5 ; i++)
        {
            System.out.println(accno[i] + "\t" + name[i] +
                                          "\t" + bal[i]);

            // display own exception if balance < 1000
            if (bal[i] < 1000)
            {
                MyException me =
                   new MyException("Balance is less than 1000");
                throw me;
            }
        }
    } //end of try

    catch (MyException e) {
        e.printStackTrace();
    }
}
```

Runtime Error

```
MyException: Balance is less than 1000
   at MyException.main(fileProperty.java:36)
```

Details of account numbers, customer names, and balance amounts are taken in the form of three arrays.
In main() method, the details are displayed using a for-loop. At this time, a check is done if in any account the balance amount is less than the minimum balance amount to be apt in the account.
If it is so, then MyException is raised and a message is displayed “Balance amount is less”.

Input / Output Basics

Input /Output Basics: Byte Streams and Character Streams, Reading and Writing File in Java.

File I/O

In Java, we can read data from files and also write data in files.

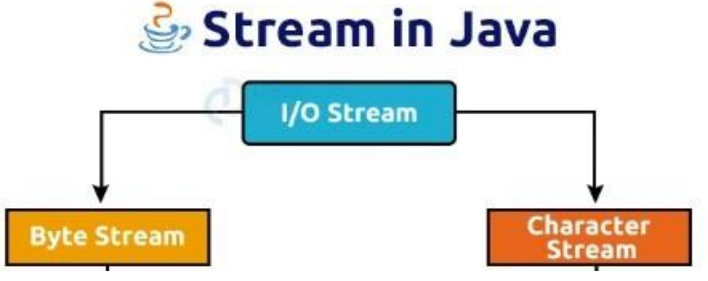
We do this using **streams**. Java has many input and output streams that are used to read and write data. Same as a continuous flow of water is called water stream, in the same way input and output flow of data is called stream.

Stream

Java provides many input and output stream classes which are used to read and write.

Streams are of two types.

- **Byte Stream**
- **Character Stream**



These streams can be different in how they are handling data and the type of data they are handling.

Byte Streams:

Byte streams are designed to deal with raw binary data, which includes all kinds of data, including characters, pictues, audio, and video. These streams are represented through cclasses that cease with the word "InputStream" or "OutputStream" of their names,along with FileInputStream,BufferedInputStream, FileOutputStream and BufferedOutputStream.

Byte streams offer a low-stage interface for studying and writing character bytes or blocks of bytes. They are normally used for coping with non-textual statistics, studying and writing files of their binary form, and running with network sockets. Byte streams don't perform any individual encoding or deciphering. They treat the data as a sequence of bytes and don't interpret it as characters.

Character Streams:

Character streams are designed to address character based records, which includes textual records inclusive of letters, digits, symbols, and other characters. These streams are represented by way of training that quit with the phrase "Reader" or "Writer" of their names, inclusive of FileReader, BufferedReader, FileWriter, and BufferedWriter.

Character streams offer a convenient manner to read and write textual content-primarily based information due to the fact they mechanically manage character encoding and decoding. They convert the individual statistics to and from the underlying byte circulation the usage of a particular individual encoding, such as UTF-eight or ASCII.It makes person streams suitable for operating with textual content files, analyzing and writing strings, and processing human-readable statistics.

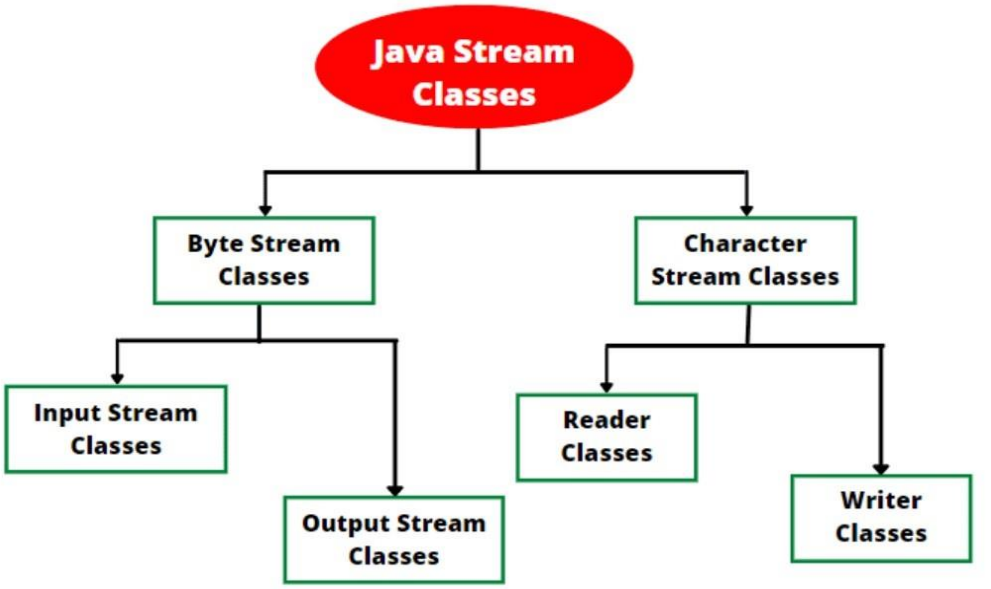


Fig: Classification of Java Stream Classes

Byte Stream

8 bits carrier

InputStream

- **BufferedInputStream**
  Used for Buffered Input Stream
- **ByteArrayInputStream**
  Used for reading from a byte array
- **DataInputStream**
  Used for reading java standard data type
- **ObjectInputStream** - Input stream for objects
- **FileInputStream** - Used for reading from a File
- **PipedInputStream** - Input pipe
- **InputStream** - Describe stream input
- **FilterInputStream** - Implements **InputStream**

OutputStream

- **BufferedOutputStream**
  Used for Buffered Output Stream
- **ByteArrayOutputStream**
  Used for writing into a byte array
- **DataOutputStream**
  Used for writing java standard data type
- **ObjectOutputStream** - Output stream for objects
- **FileOutputStream** - Used for writing into a File
- **PipedOutputStream** - Output pipe
- **OutputStream** - Describe stream output
- **FilterOutputStream** - Implements **OutputStream**
- **PrintStream** - Contains **print( )** and **println( )**

read( ) and **write( )** both are key methods of byte stream

Example code for Byte Stream:

```
import java.io.ByteArrayInputStream;
public class ByteStreamExample
{
   public static void main(String[] args)
    {
      // Creates the array of bytes
      byte[] array = {10,20,30,40};
      try {
         ByteArrayInputStream input=new ByteArrayInputStream(array);
         System.out.println("The bytes read from the input stream:");
         for (int i=0;i<array.length;i++)
         {
            // Reads the bytes
            int data=input.read();
            System.out.print(data+",");
         }
         input.close();
      } catch (Exception ex)
{
         ex.getStackTrace();
      }
   }
}
```

Output:

```
The bytes read from the input stream:10,20,30,40.
```

Example code for Character Stream:

```
import java.io.CharArrayReader;
import java.io.IOException;
public class CharacterStreamExample
{
   public static void main(String[] args) {
      // Creates an array of characters
      char[] array = {'H','e','l','l','o'};
      try {
         CharArrayReader reader=new CharArrayReader(array);
         System.out.print("The characters read from the reader:");
         int charRead;
         while ((charRead=reader.read())!=-1) {
            System.out.print((char)charRead+",");
         }
         reader.close();
      } catch (IOException ex)
{
         ex.printStackTrace();
      }
   }
```

Output:

```
The characters read from the reader:H,e,l,l,o,
```

Reading and Writing File in Java

Reading from a file:

1. In Java, the FileInputSream class is used for reading binary data from a file.
2. It is an input stream that reads bytes from a file in a file system.

3. To read data from a file using FileInputStream, these steps you need to follow :-

Step 1: Create an instance of the FileInputStream class and pass the path of the file that you want to read as an argument to its constructor.

Step 2: Create a byte array of a fixed size to read a chunk of data from the file.

Step 3: Use the read() method of the FileInputStream class to read data from the file into the byte array. This method returns the number of bytes read , or -1 if the end of the file has been reached.

Step 4: Continue reading data from the file until the read() method returns -1.

Step 5: Close the FileInputStream object using to release the close() method to release any system resources associated with it.

Following example demonstrates how to use Fileinputstream to read data from a file :

```
import java.io.*
public class ReadFileExample {
public static void main(String[] args) {
try {
FileInputStream fileinput = new
FileInputStream ("file.txt");
byte[] buffer = new byte[1024];
int bytesRead = 0;
while ((bytesRead = fileInput.read(buffer))!=-1){
System.out.println/new String buffer, 0, bytesread));
}
fileInput.close();
} catch (IOException e) {
   e.printStackTrace();
    }
  }
}
```

In the example above, we create a FileInputStream object to read data from a file called "file.txt".

We then create a byte array of size 1024 to read a chunk of data from the file.

We use the read() method to read data into the buffer untill the end of the file is reached, and then we close the FileInputStream object.

Finally, we print the data that we read from the file to the console

Writing in a file:

1. In Java, FileOutputStream class in used for writing binary data to a file.

2. It is an output stream that writes bytes to a file in a file system.

3. To write data to a file using FileOutputStream, you need to follow these steps:

Step 1: Create an instance of the FileOutputStream class and pass the path of the file that you want to write as an argument to its constructor. If the file dosen't exist, it will be created automatically.

Step 2: Create a byte array that contains the date that you want to write to the file.

Step 3: Use the write() method of the FileOutputStream class to write the data to the File This method writes the entire array to the the file.

Step 4: Close the FileOutputStream object using the close() method to release any system resources associated with it.

Following example demonstrates how to use FileOutputstream to write data to a file :

```
import java.io.*;
public class Write File Example {
public static void main(String[] args) {
try {
FileOutputStream fileOutput = new
FileOutputStream("file.txt");
String data ="This is some data that will be written to a file";
byte[] bytes = data.getBytes();
fileOutput.write(bytes);
fileOutput.close();
} catch (IOException e) {
e.printStackTrace();
    }
  }
}
```

In the example above, we create a FileOutputStream object write data to a file called "file.txt".

We then create a string that contains some data that we wants write to the file.

We convert this string to a byte array using the getbytes() method and then we use the write() method to write the data to the file.

Finally, we close the FileOutputStream object.

Multithreading

Multithreading: Thread, Thread Life Cycle, Creating Threads, Thread Priorities, Synchronizing Threads, Inter-thread Communication.

MULTI THREADING IN JAVA

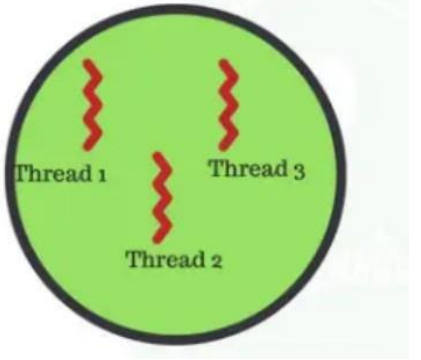
Multithreading in java is a process of executing multiple threads simultaneously. Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

Advantages of Java Multithreading

1) It doesn't block the user because threads are independent and you can perform multiple operations at 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 exception occur in a single thread.



What is Thread in java

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.

Life cycle of a Thread (Thread States)

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

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM.

Life cycle of a Thread

1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

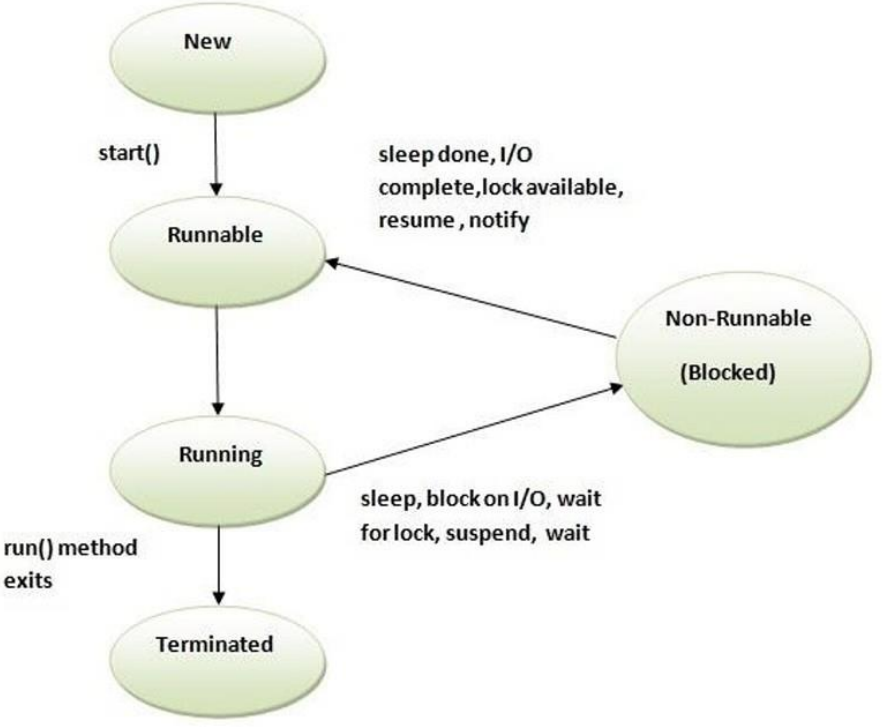
The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.



How to create thread

There are two ways to create a thread:

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

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.

6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
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:

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

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

1) Java Thread Example by extending Thread class

FileName: Multi.java

```
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...
```

2) Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);   // Using the constructor Thread(Runnable r)
t1.start();
 }
}
```

Output:

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

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

3) Using the Thread Class: Thread(String Name)

We can directly use the Thread class to spawn new threads using the constructors defined above

```
public class MyThread1
{
// Main method
```

```
public static void main(String argvs[])
{
// creating an object of the Thread class using the constructor Thread(String name)
Thread t= new Thread("My first thread");

// the start() method moves the thread to the active state
t.start();
// getting the thread name by invoking the getName() method
String str = t.getName();
System.out.println(str);
}
}
```

Output:

```
My first thread
```

4) Using the Thread Class: Thread(Runnable r, String name)

```
public class MyThread2 implements Runnable
{
public void run()
{
System.out.println("Now the thread is running ...");
}

// main method
public static void main(String argvs[])
{
// creating an object of the class MyThread2
Runnable r1 = new MyThread2();

// creating an object of the class Thread using Thread(Runnable r, String name)
Thread th1 = new Thread(r1, "My new thread");

// the start() method moves the thread to the active state
th1.start();

// getting the thread name by invoking the getName() method
String str = th1.getName();
System.out.println(str);
}
}
```

Output:

```
My new thread
Now the thread is running ...
```

Priority of a Thread (Thread Priority)

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

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).

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.

Example of priority of a Thread:

```
// Java Program to Illustrate Priorities in Multithreading
// via help of getPriority() and setPriority() method
// Importing required classes
import java.lang.*;
// Main class
class ThreadDemo extends Thread {
// Method 1
// run() method for the thread that is called
// as soon as start() is invoked for thread in main()
public void run()
{
    // Print statement
    System.out.println("Inside run method");
}

// Main driver method
public static void main(String[] args)
{
    // Creating random threads
    // with the help of above class
    ThreadDemo t1 = new ThreadDemo();
    ThreadDemo t2 = new ThreadDemo();
    ThreadDemo t3 = new ThreadDemo();
  // Thread 1
  // Display the priority of above thread
  // using getPriority() method
  System.out.println("t1 thread priority : "
                     + t1.getPriority());
System.out.println("t2 thread priority : "
                     + t2.getPriority());

// Thread 3
System.out.println("t3 thread priority : "
                     + t3.getPriority());

// Setting priorities of above threads by
// passing integer arguments
t1.setPriority(2);
t2.setPriority(5);
t3.setPriority(8);

// t3.setPriority(21); will throw
// IllegalArgumentException

// 2
System.out.println("t1 thread priority : "
                     + t1.getPriority());

// 5
System.out.println("t2 thread priority : "
                     + t2.getPriority());
// 8
System.out.println("t3 thread priority : "
                    + t3.getPriority());

// Main thread

// Displays the name of
// currently executing Thread
System.out.println(
    "Currently Executing Thread : "
    + Thread.currentThread().getName());

System.out.println(
    "Main thread priority : "
    + Thread.currentThread().getPriority());

// Main thread priority is set to 10
Thread.currentThread().setPriority(10);

System.out.println(
    "Main thread priority : "
    + Thread.currentThread().getPriority());

    }
}
```

Output

```
t1 thread priority : 5
t2 thread priority : 5
t3 thread priority : 5
t1 thread priority : 2
t2 thread priority : 5
t3 thread priority : 8
Currently Executing Thread : main
Main thread priority : 5
Main thread priority : 10
```

Synchronization in Java

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

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization

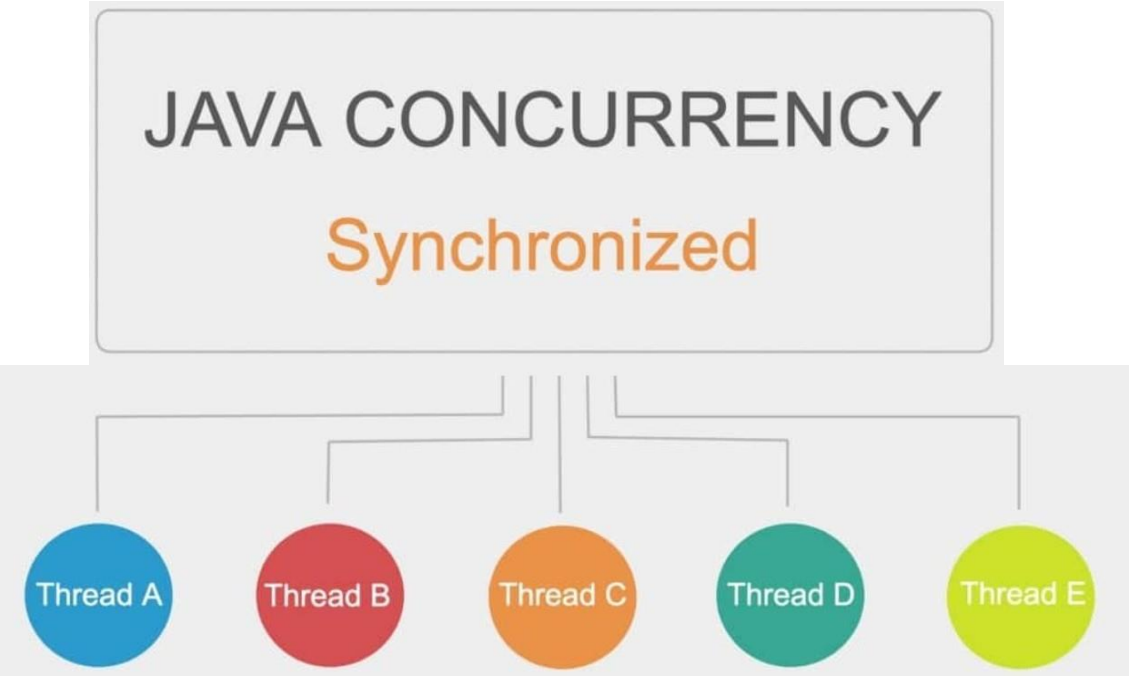
The synchronization is mainly used to

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



Thread Synchronization

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

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

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

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 the object'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.

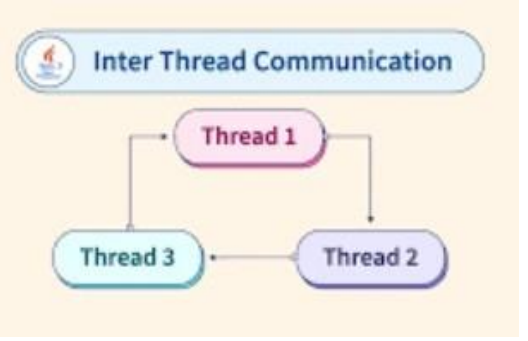
```
1.  //example of java synchronized method
2.  class Table{
3.   synchronized void printTable(int n){//synchronized method
4.     for(int i=1;i<=5;i++){
5.       System.out.println(n*i);
6.       try{
7.        Thread.sleep(400);
8.       }catch(Exception e){System.out.println(e);}
9.     }
10.
11.          }
12.         }
13.
14.         class MyThread1 extends Thread{
15.         Table t;
16.         MyThread1(Table t){
17.         this.t=t;
18.         }
19.         public void run(){
20.         t.printTable(5);
21.         }
22.
23.         }
24.         class MyThread2 extends Thread{
25.         Table t;
26.         MyThread2(Table t){
27.         this.t=t;
28.         }
29.         public void run(){
30.         t.printTable(100);
31.         }
32.         }
33.
34.         public class TestSynchronization2{
35.         public static void main(String args[]){
36.         Table obj = new Table();//only one object
37.         MyThread1 t1=new MyThread1(obj);
38.         MyThread2 t2=new MyThread2(obj);
39.         t1.start();
40.         t2.start();
41.         }
42.         }
```

Inter-thread communication in Java

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

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

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



1) wait() method

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

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()
```

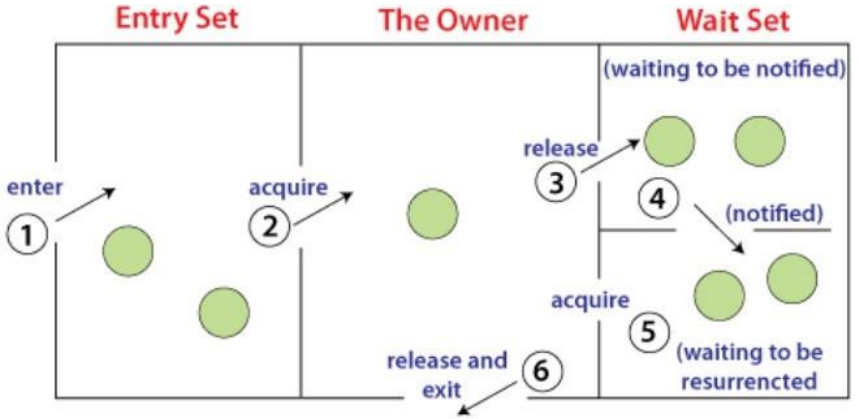
Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

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.

Understanding the process of inter-thread communication



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

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

Example of Inter Thread Communication in Java

```
class Customer{
int amount=10000;

synchronized void withdraw(int amount){
System.out.println("going to withdraw...");

if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}
```

```
synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}
}

class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
public void run(){c.deposit(10000);}
}.start();

}}
```

Output:

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