Unit – 3: EXCEPTION HANDLING AND MULTITHREADING

UNIT- 3 EXCEPTION HANDLING AND MULTITHREADING

Exception Handling basics - Multiple catch Clauses- Nested try Statements - Java's built-in exceptions, User defined exception, Multithreaded Programming: Java Thread Model, Creating a thread and multiple threads- Priorities- Synchronization- Inter-Thread communication-Suspending-Resuming and Stopping Threads- Multithreading. Wrappers- Auto boxing.

3.1: EXCEPTION HANDLING BASICS

Definition:

An Exception is an event that occurs during program execution which disrupts the normal flow of a program. It is an object which is thrown at runtime.

Occurrence of any kind of exception in java applications may result in an abrupt termination of the JVM or simply the JVM crashes.

In Java, an exception is an **object** that contains:

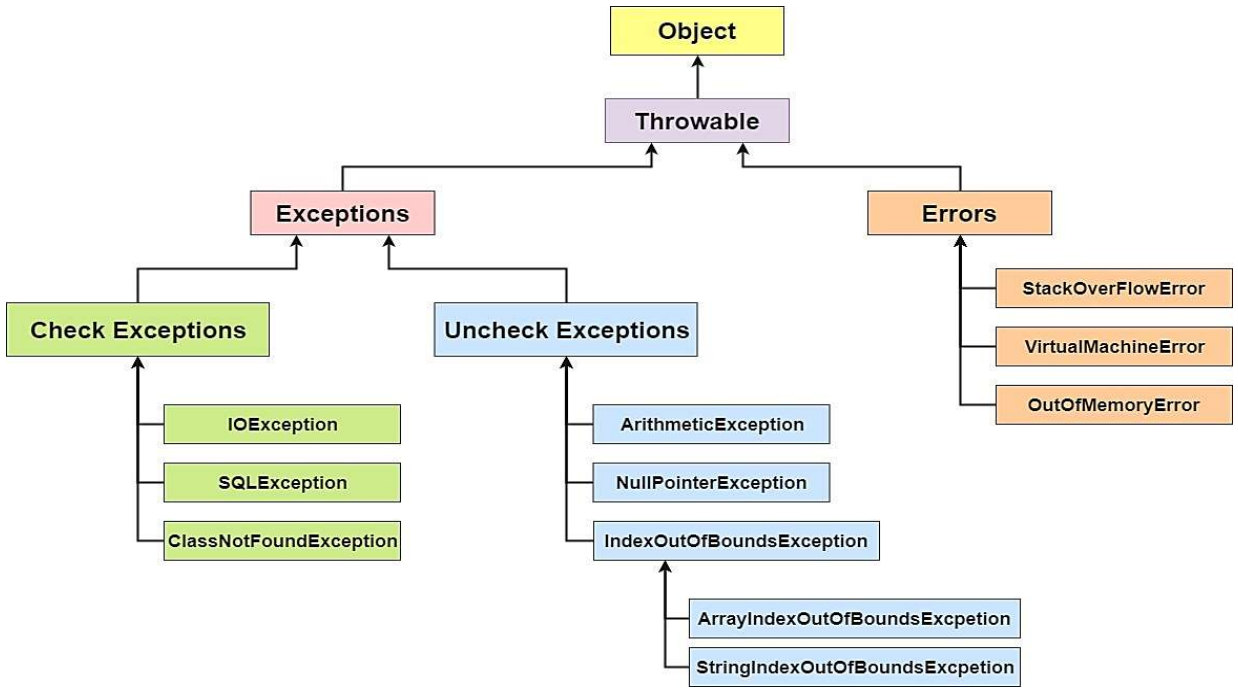
- Information about the error including its type
- The state of the program when the error occurred
- Optionally, other custom information

3.1.1: Exception Hierarchy

All exceptions and errors extend from a common java.lang.Throwable parent class.

The **Throwable** class is further divided into two classes:

1. **Exceptions** and
2. **Errors.**



Exceptions: Exceptions represents errors in the Java application program, written by the user. Because the error is in the program, exceptions are expected to be handled, either

- Try to recover it if possible
- Minimally, enact a safe and informative shutdown.

Sometimes it also happens that the exception could not be caught and the program may get terminated. Eg. **ArithmeticException**

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Errors: Errors represent internal errors of the Java run-time system which could not be handled easily. Eg. **OutOfMemoryError.**

DIFFERENCE BETWEEN EXCEPTION AND ERROR:

S.No.	Exception	Error
1.	Exceptions can be recovered	Errors cannot be recovered
2.	Exceptions are of type **java.lang.Exception**	Errors are of type **java.lang.Error**
3.	Exceptions can be classified into two types: a) Checked Exceptions b) Unchecked Exceptions	There is no such classification for errors. Errors are always unchecked.
4.	In case of Checked Exceptions, compiler will have knowledge of checked exceptions and force to keep try…catch block. Unchecked Exceptions are not known to compiler because they occur at run time.	In case of Errors, compiler won't have knowledge of errors. Because they happen at run time.
5.	Exceptions are mainly caused by the application itself.	Errors are mostly caused by the environment in which application is running.

6.	Examples: **Checked Exceptions:** SQLException, IOException **Unchecked Exceptions:** ArrayIndexOutOfBoundsException, NullPointerException	Examples: Java.lang.StackOverFlowError, java.lang.OutOfMemoryError

3.1.2: Exception Handling

What is exception handling?

Exception Handling is a mechanism to handle runtime errors, such as ClassNotFoundException, IOException, SQLException, RemoteException etc. by taking the necessary actions, so that normal flow of the application can be maintained.

Advantage of using Exceptions:

- Maintains the normal flow of execution of the application.
- Exceptions separate error handling code from regular code.
  - Benefit: Cleaner algorithms, less clutter
- Meaningful Error reporting.
- Exceptions standardize error handling.

JAVA EXCEPTION HANDLING KEYWORDS

Exception handling in java is managed using the following five keywords:

S.No.	Keyword	Description
1	**try**	A block of code that is to be monitored for exception.
2	**catch**	The catch block handles the specific type of exception along with the try block. For each corresponding try block there exists the catch block.
3	**finally**	It specifies the code that must be executed even though exception may or may not occur.
4	**throw**	This keyword is used to explicitly throw specific exception from the program code.
5	**throws**	It specifies the exceptions that can be thrown by a particular method.

- **try Block:**
  - ✓ The java code that might throw an exception is enclosed in try block. It must be used within the method and must be followed by either catch or finally block.

- ✓ If an exception is generated within the try block, the remaining statements in the try block are not executed.

- **catch Block:**
  - ✓ Exceptions thrown during execution of the try block can be caught and handled in a catch block.
  - ✓ On exit from a catch block, normal execution continues and the finally block is executed.

- **finally Block:**

  A finally block is always executed, regardless of the cause of exit from the try block, or whether any catch block was executed.
  - ✓ Generally finally block is used for freeing resources, cleaning up, closing connections etc.
  - ✓ Even though there is any exception in the try block, the statements assured by **finally** block are sure to execute.
  - ✓ **Rule:**
    - **For each try block there can be zero or more catch blocks, but only one finally block.**
    - **The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).**

The try-catch-finally structure(Syntax):

```
try {
   // Code block
}
catch (ExceptionType1 e1) {
   // Handle ExceptionType1 exceptions
}
catch (ExceptionType2 e2)  {
   // Handle ExceptionType2 exceptions
}
// ...
finally  {
   // Code always executed after the
   // try and any catch block
}
```

Rules for try, catch and finally Blocks:

1) Statements that might generate an exception are placed in a try block.
2) Not all statements in the try block will execute; the execution is interrupted if an exception occurs
3) For each try block there can be zero or more catch blocks, but only one finally block.
4) The try block is followed by
   i. one or more catch blocks
   ii. or, if a try block has no catch block, then it must have the finally block
5) A try block must be followed by either at least one catch block or one finally block.
6) A catch block specifies the type of exception it can catch. It contains the code known as exception handler
7) The catch blocks and finally block must always appear in conjunction with a try block.
8) The order of exception handlers in the catch block must be from the most specific exception

Program without Exception handling: (Default exception handler):

```
class Simple
{
  public static void main(String args[])
{
    int data=50/0;

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

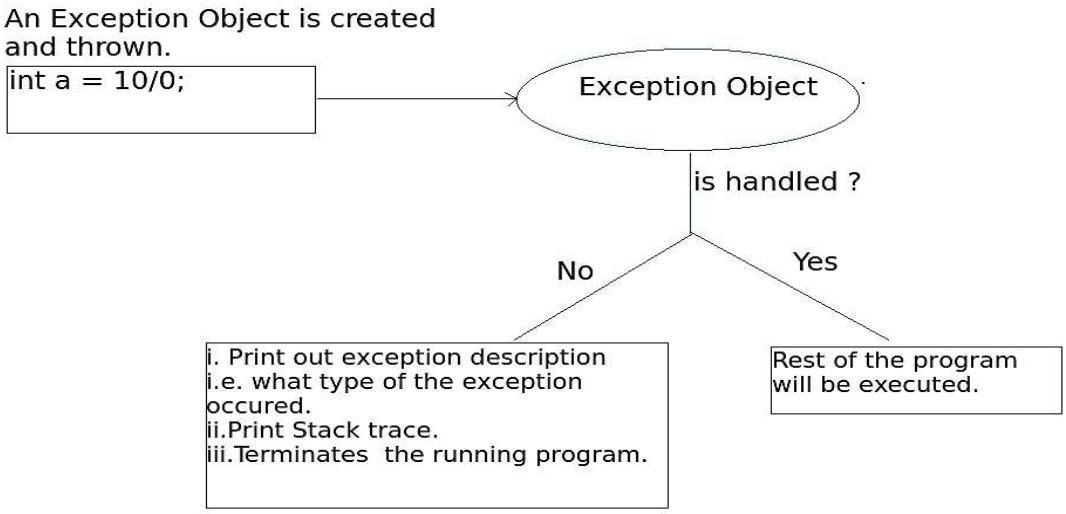
Output:

Exception in thread main java.lang.ArithmeticException:/ by zero
As displayed in the above example, rest of the code is not executed i.e. rest of the code... statement is not printed.

Program Explanation:

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.

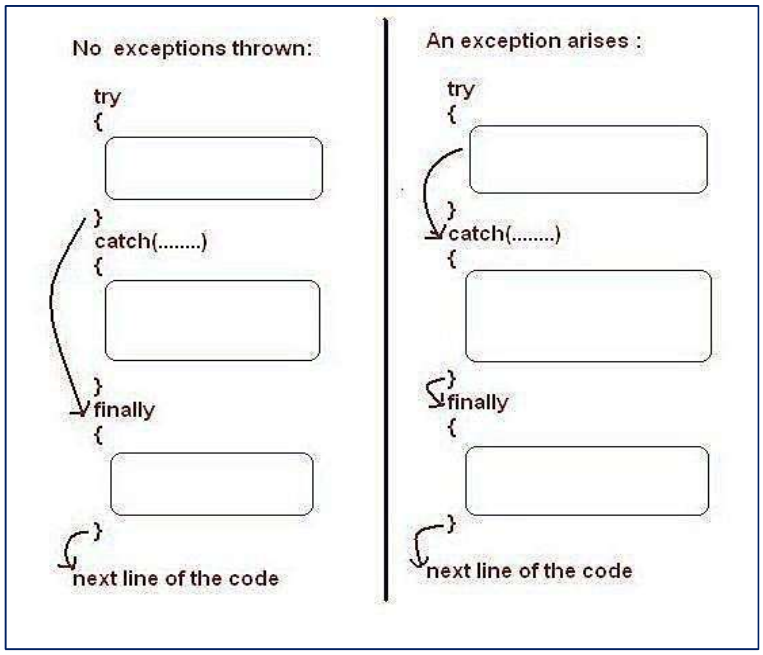


Example:

```
public class Demo
{
 public static void main(String args[])
{
 try  {
        int data=25/0;
        System.out.println(data);
  }
 catch(ArithmeticException e)
  {
        System.out.println(e);
  }
 finally    {
      System.out.println("finally block is always executed");
  }
      System.out.println("rest of the code...");
 }
}
```

Output:

java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...



3.2: Multiple catch blocks

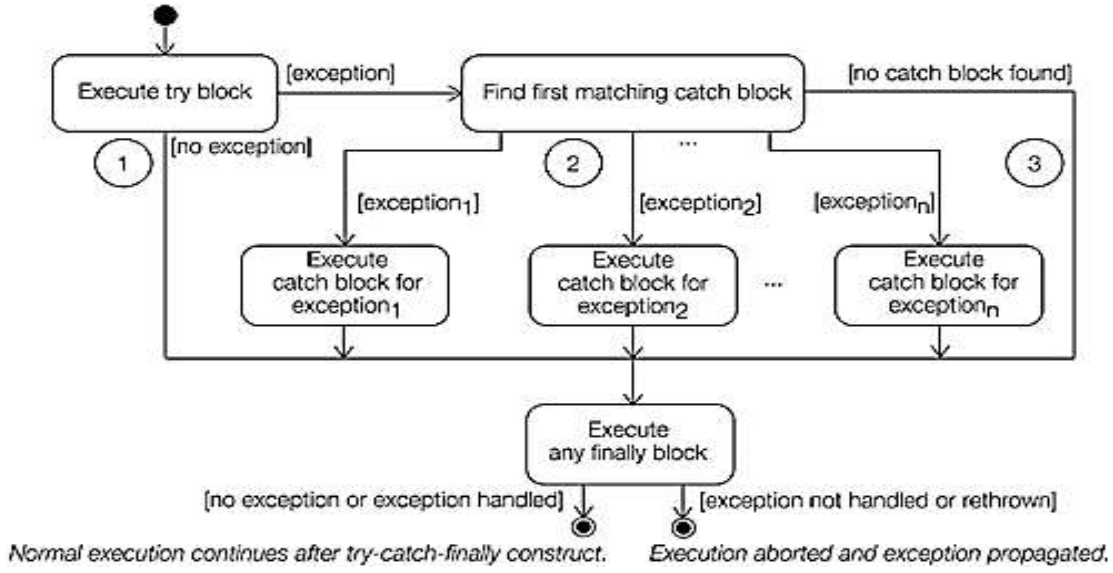
Multiple catch is used to handle many different kind of exceptions that may be generated while running the program. i.e more than one catch clause in a single try block can be used.

Rules:

- *At a time only one Exception can occur 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.*

Syntax:

```
try {
   // Code block
}
catch (ExceptionType1 e1) {
   // Handle ExceptionType1 exceptions
}
catch (ExceptionType2 e2)  {
   // Handle ExceptionType2 exceptions
}
```



Example:

```
public class MultipleCatchBlock2 {

public static void main(String[] args) {

  try
{
    int a[]= {1,5,10,15,16};
    System.out.println("a[1] = "+a[1]);
    System.out.println("a[2]/a[3] = "+a[2]/a[3]);
    System.out.println("a[5] = "+a[5]);
  }
  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:

a[1] = 5
a[2]/a[3] = 0
ArrayIndexOutOfBounds Exception occurs
rest of the code

3.3: Nested Try Block

Definition: try block within a try block is known as nested try block.

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.
- ✓ If an inner try statement does not have a matching catch statement for a particular exception, the control is transferred to the next try statement's catch handlers that for a matching catch statement.
- ✓ If none of the catch statement match, then the Java run-time system will handle the exception.

Syntax:

```
....
try
{
   statement 1;
   statement 2;
   try
   {
      statement 1;
      statement 2;
   }
   catch(Exception e)
   {
   }
}
catch(Exception e)
{
}
....
```

Example:

```
class NestedExcep
{
 public static void main(String[] args)
 {
  try
  {
   int arr[]={1,5,4,10};
   try
   {
    int x=arr[3]/arr[1];
```

```
      System.out.println("Quotient = "+x);
     }
     catch(ArithmeticException ae)
     {
      System.out.println("divide by zero");
     }
     arr[4]=3;
    }
    catch(ArrayIndexOutOfBoundsException e)
    {
     System.out.println("array index out of bound exception");
    }
    System.out.println("...End of Program...");
   }
  }
```

Output:

Quotient = 2
array index out of bound exception
...End of Program...

3.4: THROWING AND CATCHING EXCEPTIONS

Before catching an exception, it is must to throw an exception first. This means that there should be a code somewhere in the program that could catch exception thrown in the try block.

An exception can be thrown explicitly

1. Using the **throw** statement
2. Using the **throws** statement

1: Using the throw statement

- ✓ A program can explicitly throw an exception using the throw statement besides the implicit exception thrown.
- ✓ We can throw either checked, uncheked exceptions or custom(user defined) exceptions
- ✓ When **throw** statement is called:

1) It causes the termination of the normal flow of control of the program code and stops the execution of the subsequent statements.
2) It transfers the control to the nearest catch block handling the type of exception object thrown
3) If no such catch block exists, then the program terminates.

The general format of the **throw** statement is as follows:

throw <exception reference>;

The Exception reference must be of type Throwable class or one of its subclasses. A detail message can be passed to the constructor when the exception object is created.

Example:

```
1)  public class ThrowDemo
2)  {
3)  static void validate(int age)
4)  {
5)  if(age<18)
6)  throw new ArithmeticException("not valid");
7)  else
8)  System.out.println("welcome to vote");
9)  }
10) public static void main(String args[])
11) {
12) validate(13);
13) System.out.println("rest of the code...");
14) }
15) }
```

Output:

Exception in thread "main" java.lang.ArithmeticException: not valid
at ThrowDemo.validate(ThrowDemo.java:6)
at ThrowDemo.main(ThrowDemo.java:12)

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.

2: Using throws keyword:

- ✓ The **throws** statement is used by a method to specify the types of exceptions the method throws.
- ✓ If a method is capable of raising an exception that it does not handle, the method must specify that the exception have to be handled by the calling method.
- ✓ This is done using the throws clause. The throws clause lists the types of exceptions that a method might throw.

Syntax:

Return-type method_name(arg_list) **throws** *exception_list*
{
// method body
}

Example:

```
1.  import java.util.Scanner;
2.  public class ThrowsDemo
3.  {
4.  static void divide(int num, int din) throws ArithmeticException
5.  {
6.  int result=num/din;
7.  System.out.println("Result : "+result);
8.  }
9.  public static void main(String args[])
10. {
11. int n,d;
12. Scanner in=new Scanner(System.in);
13. System.out.println("Enter the Numerator : ");
14. n=in.nextInt();
15. System.out.println("Enter the Denominator : ");
16. d=in.nextInt();
17. try
18. {
19. divide(n,d);
20. }
21. catch(Exception e)
22. {
23. System.out.println(" Can't Handle : divide by zero ERROR");
24. }
25. System.out.println(" ** Continue with rest of the code ** ");
26. }
27. }
```

Output:

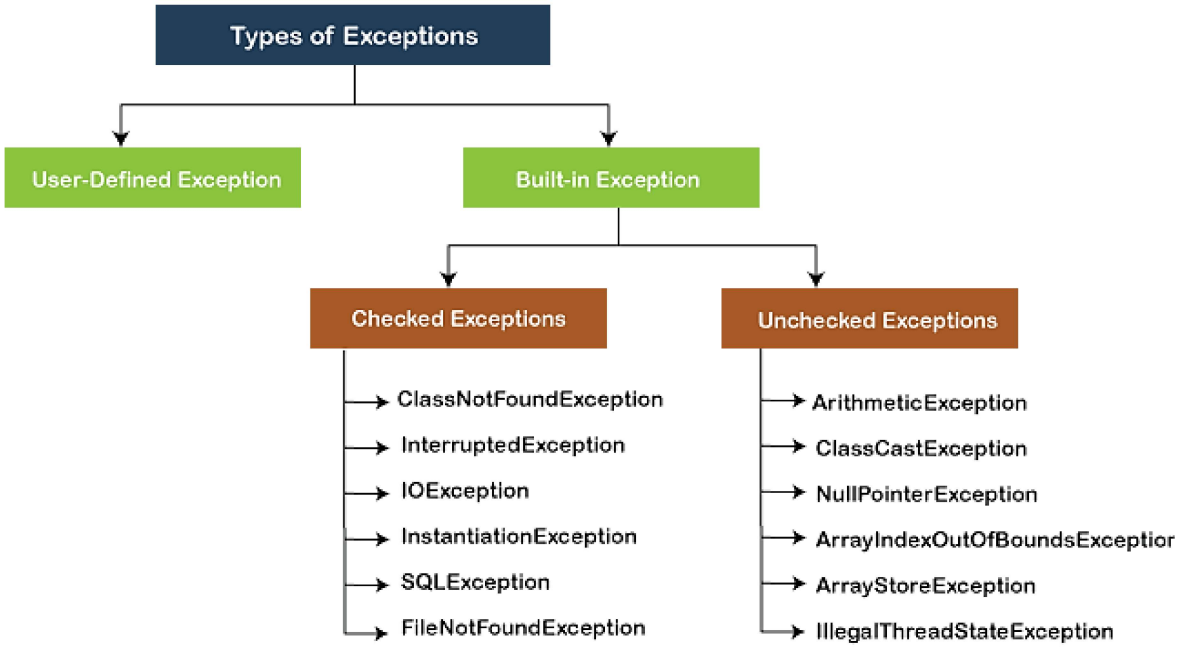
Enter the Numerator :
4

Enter the Denominator :
0
Can't Handle : divide by zero ERROR
**** Continue with rest of the code ****
Enter the Numerator :
6
Enter the Denominator :
2
Result : 3
**** Continue with rest of the code ****

Difference between throw and throws:

throw keyword	throws keyword
1) throw is used to explicitly throw an exception.	throws is used to declare an exception.
2) checked exception cannot be propagated without throws.	checked exception can be propagated with throws.
3) throw is followed by an instance.	throws is followed by class.
4) throw is used within the method.	throws is used with the method signature.
5)You cannot throw multiple exception	You can declare multiple exception e.g. **public void method()throws IOException,SQLException.**

3.5: Types of Exceptions



3.5.1: 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.

S. No.	Exception	Description
1.	**ArithmeticException**	Thrown when a problem in arithmetic operation is noticed by the JVM.
2.	**ArrayIndexOutOfBoundsException**	Thrown when you access an array with an illegal index.
3.	**ClassNotFoundException**	Thrown when you try to access a class which is not defined
4.	**FileNotFoundException**	Thrown when you try to access a non-existing file.
5.	**IOException**	Thrown when the input-output operation has failed or interrupted.
6.	**InterruptedException**	Thrown when a thread is interrupted when it is processing, waiting or sleeping
7.	**IllegalAccessException**	Thrown when access to a class is denied
8.	**NoSuchFieldException**	Thrown when you try to access any field or variable in a class that does not exist
9.	**NoSuchMethodException**	Thrown when you try to access a non-existing method.
10.	**NullPointerException**	Thrown when you refer the members of a null object
11.	**NumberFormatException**	Thrown when a method is unable to convert a string into a numeric format
12.	**StringIndexOutOfBoundsException**	Thrown when you access a String array with an illegal index.

A. Checked Exceptions:

- ✓ Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
- ✓ Checked Exceptions forces programmers to deal with the exception that may be thrown.
- ✓ The compiler ensures whether the programmer handles the exception using try.. catch () block or not. The programmer should have to handle the exception; otherwise, compilation will fail and error will be thrown.

Example:

1. ClassNotFoundException
2. CloneNotSupportedException
3. IllegalAccessException,
4. MalformedURLException.
5. NoSuchFileException
6. NoSuchMethodException
7. IOException

Example Program: (Checked Exception)

`FileNotFoundException` is a checked exception in Java. Anytime, we want to read a file from filesystem, Java forces us to handle error situation where file may not be present in place.

Without try-catch

```
import java.io.*;

public class CheckedExceptionExample {
public static void main(String[] args)
{
    FileReader file = new FileReader("src/somefile.txt");
}
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
Unhandled exception type FileNotFoundException
```

To make program able to compile, you must handle this error situation in try-catch block. **Below given code will compile absolutely fine.**

With try-catch

```
import java.io.*;
public class CheckedExceptionExample {
public static void main(String[] args) {
try {
@SuppressWarnings("resource")
FileReader file = new FileReader("src/somefile.java");
System.out.println(file.toString());
}
catch(FileNotFoundException e){
System.out.println("Sorry...Requested resource not availabe...");
} }
}
```

Output:

```
Sorry...Requested resource not availabe...
```

B. Unchecked Exceptions(RunTimeException):

- ✓ The **unchecked** exceptions are just opposite to the **checked** exceptions.
- ✓ Unchecked exceptions are not checked at compile-time rather they are checked at runtime.
- ✓ The compiler doesn't force the programmers to either catch the exception or declare it in a throws clause.
- ✓ In fact, the programmers may not even know that the exception could be thrown.

Example:

1. ArrayIndexOutOfBoundsException
2. ArithmeticException
3. NullPointerException.

Example: Unchecked Exception

Consider the following Java program. It compiles fine, but it throws *ArithmeticException* when run. The compiler allows it to compile, because *ArithmeticException* is an unchecked exception.

```
class Main {
    public static void main(String args[]) {
        int x = 0;
        int y = 10;
        int z = y/x;
    }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Main.main(Main.java:5)
```

Example 1: NullPointer Exception

```
//Java program to demonstrate NullPointerException
class NullPointer_Demo
{
   public static void main(String args[])
   {
     try {
         String a = null; //null value
         System.out.println(a.charAt(0));
      } catch(NullPointerException e) {
         System.out.println("NullPointerException..");
      }
   }
}
```

Output:

NullPointerException..

Example 2: NumberFormat Exception

```
// Java program to demonstrate NumberFormatException
class NumberFormat_Demo
{
   public static void main(String args[])
   {
     try {
       // "akki" is not a number
       int num = Integer.parseInt ("akki") ;
       System.out.println(num);
     } catch(NumberFormatException e) {
         System.out.println("Number format exception");
       }
    }
}
```

Output:

Number format exception

3.5.2: USER-DEFINED EXCEPTIONS (CUSTOM EXCEPTIONS)

Exception types created by the user to describe the exceptions related to their applications are known as **User-defined Exceptions** or **Custom Exceptions.**

To create User-defined Exceptions:

1. Pick a self-describing ***Exception** class name.
2. Decide if the exception should be checked or unchecked.
   - ✓ Checked : **extends Exception**
   - ✓ Unchecked: **extends RuntimeException**
3. Define constructor(s) that call into super class constructor(s), taking message that can be displayed when the exception is raised.
4. Write the code that might generate the defined exception inside the try-catch block.
5. If the exception of user-defined type is generated, handle it using **throw clause** as follows:

   throw ExceptionClassObject;

Example:

The following program illustrates how user-defined exceptions can be created and thrown.

```
public class EvenNoException extends Exception
{
  EvenNoException(String str)
  {
    super(str);  // used to refer the superclass constructor
  }

  public static void main(String[] args)
  {
      int arr[]={2,3,4,5};
      int rem;
      int i;
      for(i=0;i<arr.length;i++)
      {
       rem=arr[i]%2;
       try
       {
```

```
            if(rem==0)
            {
               System.out.println(arr[i]+" is an Even Number");
            }
            else
            {
               EvenNoException exp=new EvenNoException(arr[i]+" is
               not an Even Number");
               throw exp;
            }
          }
           catch(EvenNoException exp)
           {
            System.out.println("Exception thrown is "+exp);
           }
      } // for loop
   } // main()
} // class
```

Output:

2 is an Even Number
Exception thrown is EvenNoException: 3 is not an Even Number
4 is an Even Number
Exception thrown is EvenNoException: 5 is not an Even Number

Program Explanation:

In the above program, the **EvenNumberException** class is created which inherits the **Exception** super class. Then the constructor is defined with the call to the super class constructor. Next, an array **arr** is created with four integer values. In the **main(),** the array elements are checked one by one for even number. If the number is odd, then the object of EvenNumberException class is created and thrown using **throw** clause. The EvenNumberException is handled in the catch block.

Comparison Chart - final Vs. finally Vs. finalize

Basis for comparison	final	finally	finalize
Basic	Final is a "Keyword" and "access modifier" in Java.	Finally is a "block" in Java.	Finalize is a "method" in Java.
Applicable	Final is a keyword applicable to classes, variables and methods.	Finally is a block that is always associated with try and catch block.	finalize() is a method applicable to objects.
Working	(1) Final variable becomes constant, and it can't be reassigned. (2) A final method can't be overridden by the child class. (3) Final Class can not be extended.	A "finally" block, clean up the resources used in "try" block.	Finalize method performs cleans up activities related to the object before its destruction.
Execution	Final method is executed upon its call.	"Finally" block executes just after the execution of"try-catch" block.	finalize() method executes just before the destruction of the object.

Example	class FinalExample{
	public static void main(String[] args){
	final int x=100;
	x=200;//Compile Time Error
	}}
		class FinallyExample{
		public static void main(String[] args){
		try{
		int x=300;
		}catch(Exception e){System.out.println(e);}
		finally{
		System.out.println("finally block is executed");
		}
		}}
			class FinalizeExample{
			public void finalize(){System.out.println("finalize called");}
			public static void main(String[] args){
			FinalizeExample f1=new FinalizeExample();
			FinalizeExample f2=new FinalizeExample();
			f1=null;
			f2=null;
			System.gc();
			}}

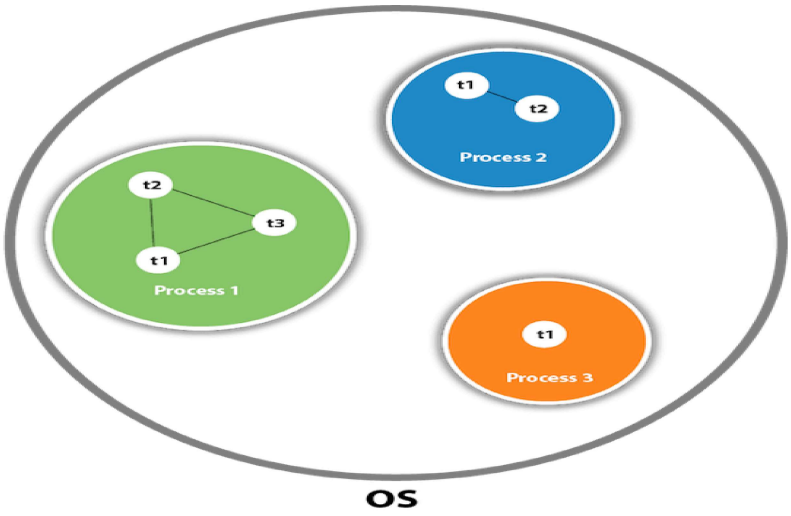
3.6: MULTITHREADED PROGRAMMING

3.6.1: Introduction to Thread

Definition: Thread

A thread is a lightweight sub-process that defines a separate path of execution. It is the smallest unit of processing that can run concurrently with the other parts (other threads) of the same process.

- ✓ Threads are independent.
- ✓ If there occurs exception in one thread, it doesn't affect other threads.
- ✓ It uses a shared memory area.



- ✓ As shown in the above figure, a thread is executed inside the process.
- ✓ There is context-switching between the threads.
- ✓ There can be multiple processes inside the OS, and one process can have multiple threads.

DIFFERENCE BETWEEN THREAD AND PROCESS:

S.NO	PROCESS	THREAD
1)	Process is a heavy weight program	Thread is a light weight process
2)	Each process has a complete set of its own variables	Threads share the same data
3)	Processes must use IPC (Inter-Process Communication) to communicate with sibling processes	Threads can directly communicate with each other with the help of shared variables
4)	Cost of communication between	Cost of communication between

	processes is high.	threads is low.
5)	Process switching uses interface in operating system.	Thread switching does not require calling an operating system.
6)	Processes are independent of one another	Threads are dependent of one another
7)	Each process has its own memory and resources	All threads of a particular process shares the common memory and resources
8)	Creating & destroying processes takes more overhead	Takes less overhead to create and destroy individual threads

3.6.2: MULTITHREADING

A program can be divided into a number of small processes. Each small process can be addressed as a single thread.

Definition: Multithreading

Multithreading is a technique of executing more than one thread, performing different tasks, simultaneously.

Multithreading enables programs to have more than one execution paths which executes concurrently. Each such execution path is a thread. For example, one thread is writing content on a file at the same time another thread is performing spelling check.

Advantages of Threads / Multithreading:

1. Threads are light weight compared to processes.
2. Threads share the same address space and therefore can share both data and code.
3. Context switching between threads is usually less expensive that between processes.
4. Cost of thread communication is low than inter-process communication.
5. Threads allow different tasks to be performed concurrently.
6. Reduces the computation time.
7. Through multithreading, efficient utilization of system resources can be achieved.

MULTITASKING

Definition: Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to maximize the utilization of CPU.

Multitasking can be achieved in two ways:

1) **Process-based Multitasking (Multiprocessing):-**
   - ❖ **It is a feature of executing two or more programs concurrently.**
   - ❖ For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site.

2) **Thread-based Multitasking (Multithreading):-**
   - ❖ **It is a feature that a single program can perform two or more tasks simultaneously.**
   - ❖ For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

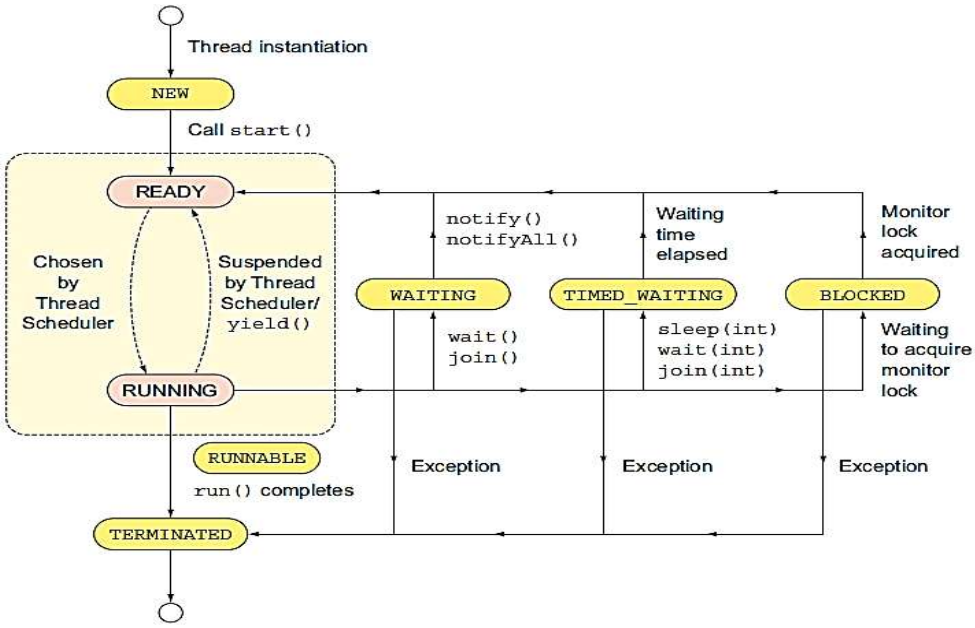
Differences between multi-threading and multitasking

Characteristics	Multithreading	Multitasking
Meaning	A process is divided into several different sub-processes called as threads, which has its own path of execution. This concept is called as multithreading.	The execution of more than one task simultaneously is called as multitasking.
Number of CPU	Can be one or more than one	One
Number of process being executed	Various components of the same process are being executed at a time.	One by one job is being executed at a time.
Number of users	Usually one.	More than one.
Memory Space	Threads are lighter weight. They share the same address space	Processes are heavyweight tasks that require their own separate address spaces.
Communication between units	Interthread communication is inexpensive	Interprocess communication is expensive and limited.
Context Switching	Context switching from one thread to the next is lower in cost.	Context switching from one process to another is also costly.

3.7: Thread Model / Thread Life Cycle (Different states of a Thread)

Different states, a thread (or applet/servlet) travels from its object creation to object removal (garbage collection) is known as life cycle of thread. A thread goes through various stages in its life cycle. At any time, a thread always exists in any one of the following state:

1. New State
2. Runnable State
3. Running State
4. Waiting/Timed Waiting/Blocked state
5. Terminated State/ dead state



1. New State:

A new thread begins its life cycle in the new state. It remains in this state until the program starts

the thread by calling **start()** method, which places the thread in the **runnable** state.

- ✓ A new thread is also referred to as a born thread.
- ✓ When the thread is in this state, only **start()** and **stop()** methods can be called. Calling any other methods causes an **IllegalThreadStateException.**
- ✓ Sample Code: **Thread myThread=new Thread();**

2. Runnable State:

After a newly born thread is started, the thread becomes runnable or running by calling the **run()** method.

- ✓ A thread in this state is considered to be executing its task.
- ✓ Sample code: **myThread.start();**
- ✓ The **start()** method creates the system resources necessary to run the thread, schedules the thread to run and calls the thread's run() method.

3. Running state:

- ✓ **Thread scheduler** selects thread to go from runnable to running state. In running state Thread starts executing by entering **run()** method.

- ✓ Thread scheduler selects thread from the runnable pool on basis of priority, if priority of two threads is same, threads are scheduled in unpredictable manner. Thread scheduler behaviour is completely unpredictable.
- ✓ When threads are in running state, **yield()** method can make thread to go in Runnable state.

4. Waiting/Timed Waiting/Blocked State :

❖ **Waiting State:**

Sometimes one thread has to undergo in waiting state because another thread starts executing. A runnable thread can be moved to a waiting state by calling the **wait**() method.

- ✓ A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- ✓ A call to **notify()** and **notifyAll()** may bring the thread from waiting state to runnable state.

❖ **Timed Waiting:**

A runnable thread can enter the timed waiting state for a specified interval of time by calling the **sleep()** method.

- ✓ After the interval gets over, the thread in waiting state enters into the runnable state.
- ✓ Sample Code:

```
try {
Thread.sleep(3*60*1000);// thread sleeps for 3 minutes
}
catch(InterruptedException ex) { }
```

❖ **Blocked State:**

When a particular thread issues an I/O request, then operating system moves the thread to

blocked state until the I/O operations gets completed.

- ✓ This can be achieved by calling **suspend()** method.
- ✓ After the I/O completion, the thread is sent back to the runnable state.

5. Terminated State:

A runnable thread enters the terminated state when,

(i) It completes its task (when the run() method has finished)

```
public void run() { }
```

(ii) Terminates (when the stop() is invoked) – **myThread.stop();**

A terminated thread cannot run again.

New : A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.
Runnable : After invocation of start() method on new thread, the thread becomes runnable.
Running : A thread is in running state if the thread scheduler has selected it.
Waiting : A thread is in waiting state if it waits for another thread to perform a task. In this stage the thread is still alive.
Terminated : A thread enter the terminated state when it complete its task.

THE "main" THREAD

The "main" thread is a thread that begins running immediately when a java program starts up. The "main" thread is important for two reasons:
1. It is the thread form which other child threads will be spawned.
2. It must be the last thread to finish execution because it performs various shutdown actions.

- ✓Although the main thread is created automatically when our program is started, it can be controlled through a **Thread** object.
- ✓To do so, we must obtain a reference to it by calling the method **currentThread()**.

Example:

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

// change the name of the main thread
t.setName("My Thread");
System.out.println("After name change : "+t);

try {
for(int n=5;n>0;n--) {
System.out.println(n);
Thread.sleep(1000);// delay for 1 second
}
```

```
        } catch(InterruptedException e) {
              System.out.println("Main Thread Interrrupted");
        }
    }
    }
```

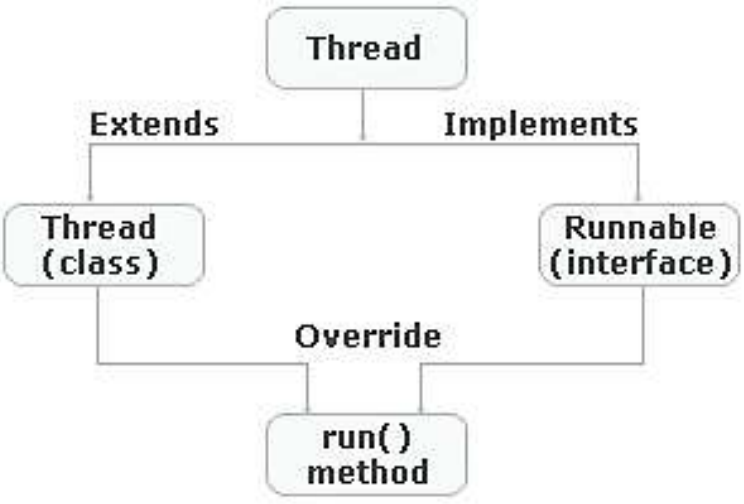
Output:

```
Current Thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

3.8: Creating Threads

We can create threads by instantiating an object of type **Thread.** Java defines two ways to create threads:

1. By implementing **Runnable** interface (**java.lang.Runnable)**
2. By extending the **Thread** class (**java.lang.Thread)**



1. Creating threads by implementing Runnable interface:

- The Runnable interface should be implemented by any class whose instances are intended to be executed as a thread.

- Implementing thread program using Runnable is preferable than implementing it by extending Thread class because of the following two reasons:
  1. If a class extends a Thread class, then it cannot extend any other class.
  2. If a class Thread is extended, then all its functionalities get inherited. This is an expensive operation.
- The Runnable interface has only one method that must be overridden by the class which implements this interface:

  public void run()// run() contains the logic of the thread
  {
  // implementation code
  }
- **Steps for thread creation:**

  1. Create a class that implements **Runnable** interface. An object of this class is **Runnable** object.

  public class MyThread implements Runnable
  {

  }

  2. Override the **run()** method to define the code executed by the thread.
  3. Create an object of type Thread by passing a Runnable object as argument.

  Thread t=new Thread(Runnable threadobj, String threadName);

  4. Invoke the **start()** method on the instance of the Thread class.

  t.start();

- **Example:**

```
class MyThread implements Runnable
{

public void run()
{
for(int i=0;i<3;i++)
{
System.out.println(Thread.currentThread().getName()+" # Printing "+i);
try
{
Thread.sleep(1000);
}catch(InterruptedException e)
{
System.out.println(e);
```

```
    }
      }
}
}
public class RunnableDemo {
public static void main(String[] args)
{
MyThread obj=new MyThread();
MyThread obj1=new MyThread();
Thread t=new Thread(obj,"Thread-1");
t.start();
Thread t1=new Thread(obj1,"Thread-2");
t1.start();
}
}
```

Output:

```
Thread-0 # Printing 0
Thread-1 # Printing 0
Thread-1 # Printing 1
Thread-0 # Printing 1
Thread-1 # Printing 2
Thread-0 # Printing 2
```

2. Creating threads by extending Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.

Commonly used Constructors of Thread class:

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

All the above constructors creates a new thread.

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 boolean isAlive():** tests if the thread is alive.
12. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
13. **public void suspend():** is used to suspend the thread(depricated).
14. **public void resume():** is used to resume the suspended thread(depricated).
15. **public void stop():** is used to stop the thread(depricated).
16. **public boolean isDaemon():** tests if the thread is a daemon thread.
17. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
18. **public void interrupt():** interrupts the thread.
19. **public boolean isInterrupted():** tests if the thread has been interrupted.
20. **public static boolean interrupted():** tests if the current thread has been interrupted.

- **<u>Steps for thread creation:</u>**

  1. Create a class that extends **java.lang.Thread** class.

  ```
  public class MyThread extends Thread
  {
  ---
  }
  ```

  2. Override the **run()** method in the sub class to define the code executed by the thread.

  3. Create an object of this sub class.

  ```
  MyThread t=new MyThread(String threadName);
  ```

  4. Invoke the **start()** method on the instance of the subclass to make the thread for running.

  ```
  start();
  ```

- **Example:**

```
class SampleThread extends Thread
{
public void run()
{
for(int i=0;i<3;i++)
{
System.out.println(Thread.currentThread().getName()+" # Printing "+i);
try
{
Thread.sleep(1000);
}catch(InterruptedException e)
{
System.out.println(e);
}
}
}
}

public class ThreadDemo {
public static void main(String[] args) {
SampleThread obj=new SampleThread();
obj.start();
SampleThread obj1=new SampleThread();
obj1.start();
}
}
```

Output:

```
Thread-0 # Printing 0
Thread-1 # Printing 0
Thread-1 # Printing 1
Thread-0 # Printing 1
Thread-0 # Printing 2
Thread-1 # Printing 2
```

3.9: THREAD PRIORITY

- ✓ Thread priority determines how a thread should be treated with respect to others.

- ✓ Every thread in java has some priority, it may be default priority generated by JVM or customized priority provided by programmer.
- ✓ Priorities are represented by a number between 1 and 10.
  1 – Minimum Priority5 – Normal Priority 10 – Maximum Priority
- ✓ Thread scheduler will use priorities while allocating processor. The thread which is having highest priority will get the chance first.
- ✓ **Thread scheduler** is a part of Java Virtual Machine (JVM). It decides which thread should execute first among two or more threads that are waiting for execution.
- ✓ It is decided based on the priorities that are assigned to threads. The thread having highest priority gets a chance first to execute.
- ✓ If two or more threads have same priorities, we can't predict the execution of waiting threads. It is completely decided by thread scheduler. It depends on the type of algorithm used by thread scheduler.
- ✓ Higher priority threads get more CPU time than lower priority threads.
- ✓ A higher priority thread can also preempt a lower priority thread. For instance, when a lower priority thread is running and a higher priority thread resumes (for sleeping or waiting on I/O), it will preempt the lower priority thread.
- ✓ If two or more threads have same priorities, we can't predict the execution of waiting threads. It is completely decided by thread scheduler. It depends on the type of algorithm used by thread scheduler.
- ✓ **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.
- ✓ To set a thread's priority, use the **setPriority()** method.
- ✓ To obtain the current priority of a thread, use **getPriority()** method.

- ✓ **Example:**

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

3.10: Thread Synchronization

Definition: Thread Synchronization

Thread synchronization is the concurrent execution of two or more threads that share critical resources.

When two or more threads need to use a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process of ensuring single thread access to a shared resource at a time is called **synchronization.**

Threads should be synchronized to avoid critical resource use conflicts. Otherwise, conflicts may arise when parallel-running threads attempt to modify a common variable at the same time.

✓ **Why use Synchronization**

The synchronization is mainly used to

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

✓ **Thread Synchronization**

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

1. Mutual 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 two ways in java:

1. by synchronized method
2. by synchronized block

✓ **Concept of Lock in Java**

Synchronization is built around an internal entity known as the lock or monitor. Every object has a 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.

1. Java synchronized method

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

Syntax to use synchronized method:

```
Access_modifier synchronized return_type method_name(parameters)
{ …….. }
```

Example of java synchronized method:

```
class Table{
 synchronized void printTable(int n)//synchronized method
{
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
```

2. 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

1. **synchronized (object reference expression) {**
2. **//code block**
3. **}**

Example of synchronized 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
```

Difference between synchronized method and synchronized block:

Synchronized method	Synchronized block
1. Lock is acquired on whole method. 2. Less preferred. 3. Performance will be less as compared to synchronized block.	1. Lock is acquired on critical block of code only. 2. Preferred. 3. Performance will be better as compared to synchronized method.

3.11: Inter-Thread Communication

Inter-Thread Communication or Co-operation is all about allowing synchronized threads to communicate with each other.

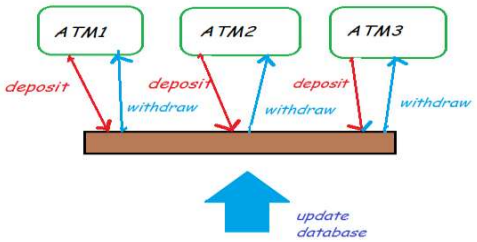
Definition: Inter-Thread Communication
Inter-thread communication is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

It is implemented by following methods of Object class and all these methods can be called only from within a synchronized context.

S.No.	Method & Description
1	**public final void wait() throws InterruptedException** Causes the current thread to wait until another thread invokes the notify().
2	**public final void wait(long timeout) throws InterruptedException** Causes current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed. **Parameters:** **timeout** – *the maximum time to wait in milliseconds.*
3	**public final void notify()** Wakes up a single thread that is waiting on this object's monitor.
4	**Public final void notifyAll()** Wakes up all the threads that called wait( ) on the same object.

Difference between wait() and sleep()		
Parameter	**wait()**	**sleep()**
Synchonized	wait should be called from synchronized context i.e. from block or method, If you do not call it using synchronized context, it will throw IllegalMonitorStateException	It need not be called from synchronized block or methods
Calls on	wait method operates on Object and defined in Object class	Sleep method operates on current thread and is in java.lang.Thread
Release of lock	wait release lock of object on which it is called and also other locks if it holds any	Sleep method does not release lock at all
Wake up condition	until call notify() or notifyAll() from Object class	Until time expires or calls interrupt()
static	wait is non-static method	sleep is static method

Example: The following program illustrates simple bank transaction operations with inter-thread communication:




```
class Customer{
int Balance=10000;

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

  if(Balance<amount)
  {
          System.out.println("Less balance;  Balance = Rs. "+Balance+"\nWaiting for
  deposit...\n");
          try
 {
  wait();
                  }
 catch(Exception e){}
  }
```

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

class ThreadCommn
{
  public static void main(String args[]) {
            Customer c=new Customer();
            new Thread()
            {
                        public void run(){c.withdraw(20000);}
            }.start();
            new Thread(){
                        public void run(){c.deposit(15000);}
            }.start();
  }
}
```

Output:

```
going to withdraw...20000
Less balance;  Balance = Rs. 10000
Waiting for deposit...

going to deposit... Rs. 15000
deposit completed... Balance = 25000
withdraw completed...
```

3.12: Suspending, Resuming and Stopping threads

The functions of Suspend, Resume and Stop a thread is performed using Boolean-type flags in a multithreading program. These flags are used to store the current status of the thread.

1. If the suspend flag is set to true, then run() will suspend the execution of the currently running thread.
2. If the resume flag is set to true, then run() will resume the execution of the suspended thread.
3. If the stop flag is set to true, then a thread will get terminated.

Example

```
class NewThread implements Runnable
 {
    String name;      //name of thread
    Thread thr;
    boolean suspendFlag;
    boolean stopFlag;

    NewThread(String threadname)
    {
       name = threadname;
       thr = new Thread(this, name);
       System.out.println("New thread : " + thr);
       suspendFlag = false;
       stopFlag = false;
       thr.start();     // start the thread
    }

    /* this is the entry point for thread */
    public void run()
    {
       try
       {
          for(int i=1; i<10; i++)
          {
             System.out.println(name + " : " + i);
             Thread.sleep(1000);
```

```
            synchronized(this)
            {
               while(suspendFlag)
               {
                  wait();
               }
               if(stopFlag)
                  break;
            }
         }
      }
      catch(InterruptedException e)
      {
         System.out.println(name + " interrupted");
      }

      System.out.println(name + " exiting...");
   }

   synchronized void mysuspend()
   {
      suspendFlag = true;
   }

   synchronized void myresume()
   {
      suspendFlag = false;
      notify();
   }

    synchronized void mystop()
  {
   suspendFlag=false;
   stopFlag=true;
   notify();
   System.out.println("Thread "+name+" Stopped!!!");
  }

}
```

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

      NewThread obj1 = new NewThread("One");
      NewThread obj2 = new NewThread("two");

      try
      {
         Thread.sleep(1000);
         obj1.mysuspend();
         System.out.println("Suspending thread One...");
         Thread.sleep(1000);
         obj1.myresume();
         System.out.println("Resuming thread One...");

         obj2.mysuspend();
         System.out.println("Suspending thread Two...");
         Thread.sleep(1000);
         obj2.myresume();
         System.out.println("Resuming thread Two...");
         obj2.mystop();

      }
      catch(InterruptedException e)
      {
         System.out.println("Main thread Interrupted..!!");
      }

      System.out.println("Main thread exiting...");

   }
}
```

Output:

```
New thread : Thread[One,5,main]
New thread : Thread[two,5,main]
```

One : 1
two : 1
One : 2
Suspending thread One...
two : 2
two : 3
Resuming thread One...
One : 3
Suspending thread Two...
One : 4
Resuming thread Two...
two : 4
Thread two Stopped!!!
Main thread exiting...
two exiting...
One : 5
One : 6
One : 7
One : 8
One : 9
One exiting...

3.13: Wrappers

Wrappers

Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.

The table below shows the primitive type and the equivalent wrapper class:

Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
Boolean	Boolean
char	Character

3.13.1. Use of Wrapper classes

- ✓ **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- ✓ **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- ✓ **Synchronization:** Java synchronization works with objects in Multithreading.
- ✓ **java.util package:** The java.util package provides the utility classes to deal with objects.
- ✓ **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

Example:

```
//Java Program to convert all primitives into its corresponding
//wrapper objects and vice-versa
public class WrapperExample3{
public static void main(String args[]){
byte b=10;
short s=20;
int i=30;
long l=40;
float f=50.0F;
double d=60.0D;
char c='a';
boolean b2=true;

//Autoboxing: Converting primitives into objects
Byte byteobj=b;
Short shortobj=s;
Integer intobj=i;
Long longobj=l;
Float floatobj=f;
Double doubleobj=d;
Character charobj=c;
Boolean boolobj=b2;

//Printing objects
```

```
System.out.println("---Printing object values---");
System.out.println("Byte object: "+byteobj);
System.out.println("Short object: "+shortobj);
System.out.println("Integer object: "+intobj);
System.out.println("Long object: "+longobj);
System.out.println("Float object: "+floatobj);
System.out.println("Double object: "+doubleobj);
System.out.println("Character object: "+charobj);
System.out.println("Boolean object: "+boolobj);

//Unboxing: Converting Objects to Primitives
byte bytevalue=byteobj;
short shortvalue=shortobj;
int intvalue=intobj;
long longvalue=longobj;
float floatvalue=floatobj;
double doublevalue=doubleobj;
char charvalue=charobj;
boolean boolvalue=boolobj;

//Printing primitives
System.out.println("---Printing primitive values---");
System.out.println("byte value: "+bytevalue);
System.out.println("short value: "+shortvalue);
System.out.println("int value: "+intvalue);
System.out.println("long value: "+longvalue);
System.out.println("float value: "+floatvalue);
System.out.println("double value: "+doublevalue);
System.out.println("char value: "+charvalue);
System.out.println("boolean value: "+boolvalue);
}
}
```

Output

```
---Printing object values---
Byte object: 10
Short object: 20
Integer object: 30
Long object: 40
```

```
Float object: 50.0
Double object: 60.0
Character object: a
Boolean object: true
---Printing primitive values---
byte value: 10
short value: 20
int value: 30
long value: 40
float value: 50.0
double value: 60.0
char value: a
boolean value: true
```

3.14: Autoboxing

3.14. Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Example:

```
public class WrapperExample1{
public static void main(String args[]){
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer explicitly
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
System.out.println(a+" "+i+" "+j);
}
}
```

Output

```
20  20  20
```

3.14.1. Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing.

Example:

```
//Unboxing example of Integer to int
public class WrapperExample2
{
public static void main(String args[])
{
//Converting Integer to int
Integer a=new Integer(3);
int i=a.intValue();          //converting Integer to int explicitly
int j=a;                     //unboxing, now compiler will write a.intValue() internally
System.out.println(a+" "+i+" "+j);
}
}
```

Output

3 3 3

A1: STACK TRACE ELEMENTS

A Stack Trace is a list of method calls from the point when the application was started to the current location of execution within the program. A Stack Trace is produced automatically by the Java Virtual Machine when an exception is thrown to indicate the location and progression of the program up to the point of the exception. They are displayed whenever a Java program terminates with an uncaught exception.

- ✓ We can access the text description of a stack trace by calling the **printStackTrace()** method of the **Throwable** class.
- ✓ The **java.lang.StackTraceElement is a** class where each element represents a single stack frame.
- ✓ We can call the **getStackTrace()** method to get an array of **StackTraceElement** objects that we want analyse in our program.

Class Declaration

Following is the declaration for **java.lang.StackTraceElement** class

public final class StackTraceElement extends **Object** implements **Serializable**

Class constructors

Constructor & Description
StackTraceElement(String declaringClass, String methodName, String fileName, int lineNumber) This creates a stack trace element representing the specified execution point.

Parameters:

- **declaringClass –** the fully qualified name of the class containing the execution point represented by the stack trace element.
- **methodName –** the name of the method containing the execution point represented by the stack trace element.
- **fileName –** the name of the file containing the execution point represented by the stack trace element, or null if this information is unavailable
- **lineNumber –** the line number of the source line containing the execution point represented by this stack trace element, or a negative number if this information is unavailable. A value of -2 indicates that the method containing the execution point is a native method.

Throws: NullPointerException – if declaringClass or methodName is null.

Methods in StackTraceElement class:

Method Name	Description
String getFileName()	Gets the name of the source file containing the execution point represented by the **StackTraceElement.**
int getLineNumber()	Gets the line number of the source file containing the execution point represented by the **StackTraceElement.**
String getClassName()	Gets the fully qualified name of the class containing the execution point represented by the **StackTraceElement.**
String getMethodName()	Gets the name of the method containing the execution point represented by the **StackTraceElement.**
boolean isNativeMethod()	Returns true if the execution point of the **StackTraceElement** is inside a native method.
String toString()	Returns a formatted string containing the class name, method name, file name and the line number, if available.

Example:

The following program for finding factorial(using recursion) prints the stack trace of a recursive factorial function.

```
import java.util.Scanner;

public class StackTraceTest
{
   public static int factorial(int n)
   {
      System.out.println(" Factorial ("+n+"):");
      Throwable t=new Throwable();
      StackTraceElement[] frames=t.getStackTrace();
      for(StackTraceElement f:frames)
      {
         System.out.println(f);
      }
      int r;
      if(n<=1)
         r=1;
      else
         r=n*factorial(n-1);
      System.out.println("return "+r);
      return r;
   }

   public static void main(String[] args)
   {
      Scanner in=new Scanner(System.in);
      System.out.println("Enter n: ");
      int n=in.nextInt();
      factorial(n);
   }
}
```

Output:

Enter n: 3

Factorial (3):
StackTraceTest.factorial(StackTraceTest.java:10)
StackTraceTest.main(StackTraceTest.java:30)
Factorial (2):
StackTraceTest.factorial(StackTraceTest.java:10)
StackTraceTest.factorial(StackTraceTest.java:20)
StackTraceTest.main(StackTraceTest.java:30)
Factorial (1):
StackTraceTest.factorial(StackTraceTest.java:10)
StackTraceTest.factorial(StackTraceTest.java:20)
StackTraceTest.factorial(StackTraceTest.java:20)
StackTraceTest.main(StackTraceTest.java:30)
return 1
return 2
return 6

A2: "assert" Keyword

Java assert **keyword is used to create assertions in Java, which enables us to test the assumptions about our program. For example, an assertion may be to make sure that an employee's age is positive number.**

Assertions are Boolean expressions that are used to test/validate the code. It is a statement in java that can be used to test your assumptions about the program.

- Assertion is achieved using "assert" keyword in java.
- While executing assertion, it is believed to be true. If it fails, JVM will throw an error named AssertionError. It is mainly used for testing purpose.

- **Following are the situations in which we can use the assertions:**

  - For making the program more readable and user friendly, the assert statements are used.
  - For validating the internal control flow and class invariant, the assertions are used.

- **Syntax of using Assertion:**

  There are two ways to use assertion.

  First way:

  1. **assert expression;**

Here the **Expression** is evaluated by the JVM and if any error occurs then **AssertionError** occurs.

Second way:

2. **assert expression1 : expression2;**

In this, Expression1 is evaluated and if it is false then the error message is displayed with the help of Expression2.

- **Assertion Enabling and Disabling:**

By default, assertions are disabled. They have to be enabled explicitly

For Enabling:

We can enable the assertions by running the java program with the **-enableassertions** (or) **-ea** option:

```
java –enableassertions AssertionDemo
        (or)
java –ea AssertionDemo
```

For Disabling: **-disableassertions** (or) **-da**

```
java –disableassertions AssertionDemo
        (or)
java –da AssertionDemo
```

When assertions are disabled, the class loader strips out the assertion code so that it won't slow excetuion.

Example:

```
// Java program to demonstrate syntax of assertion
import java.util.Scanner;

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

```
{
int value = 15;
assert value >= 20 : " Underweight";
System.out.println("value is "+value);
}
}
```

Output:

value is 15

After enabling assertions

Output:

Exception in thread "main" java.lang.AssertionError:
Underweight

- **Advantage of Assertions:**
  - It provides an effective way to detect and correct programming errors.

- **Where not to use Assertions**
  - Assertions should not be used to replace error messages
  - Do not use assertions for argument checking in public methods. Because if arguments are erroneous then that situation result in appropriate runtime exception such as **IllegalArgumentException, IndexOutOfBoundsException or NullPointerException.**