UNIT-3

Exception handling and Multithreading - Concepts of exception handling, benefits of exception handling, Termination or resumptive models, exception hierarchy, usage of try, catch, throw, throws and finally, built in exceptions, creating own exception subclasses. String handling, Exploring java.util. Differences between multithreading and multitasking, thread life cycle, creating threads, thread priorities, synchronizing threads, inter thread communication, thread groups, daemon threads. Enumerations, autoboxing , annotations, generics.

CONCEPTS OF EXCEPTION HANDLING

EXCEPTION

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

Major reasons why an exception Occurs

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

Errors

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

EXCEPTION HANDLING IN JAVA

- The Exception Handling in Java is one of the powerful feature to handle the runtime errors so that normal flow of the application can be maintained.
- Exception Handling is used to convert system error message into user friendly error message.

Let's take 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 your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed.
- If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

UNCAUGHT EXCEPTIONS(WITH OUT USING TRY & CATCH)

Example without Exception Handling

```
class ExceptionDemo
{
 public static void main(String[] args)
 {
int a=30, b=0;
int c=a/b;
System.out.println("Denominator should not be zero");
 }
 }
```

Output: Exception in thread "main" java.lang.ArithmeticException: / by zero at ExceptionDemo.main(ExceptionDemo.java:7)

Explanation: Abnormally terminate program and give a message like below,

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

at ExceptionDemo.main(ExceptionDemo.java:7)

 This error message is not understandable by user so we convert this error message into user friendly error message, like "denominator should not be zero".

BENEFITS OF EXCEPTION HANDLING

1. Provision to Complete Program Execution
2. Easy Identification of Program Code and Error-Handling Code
3. Propagation of Errors
4. Meaningful Error Reporting
5. Identifying Error Types

TERMINATION OR RESUMPTIVE MODELS

In java, there are two exception models. Java programming language has two models of exception handling. The exception models that java supports are as follows.

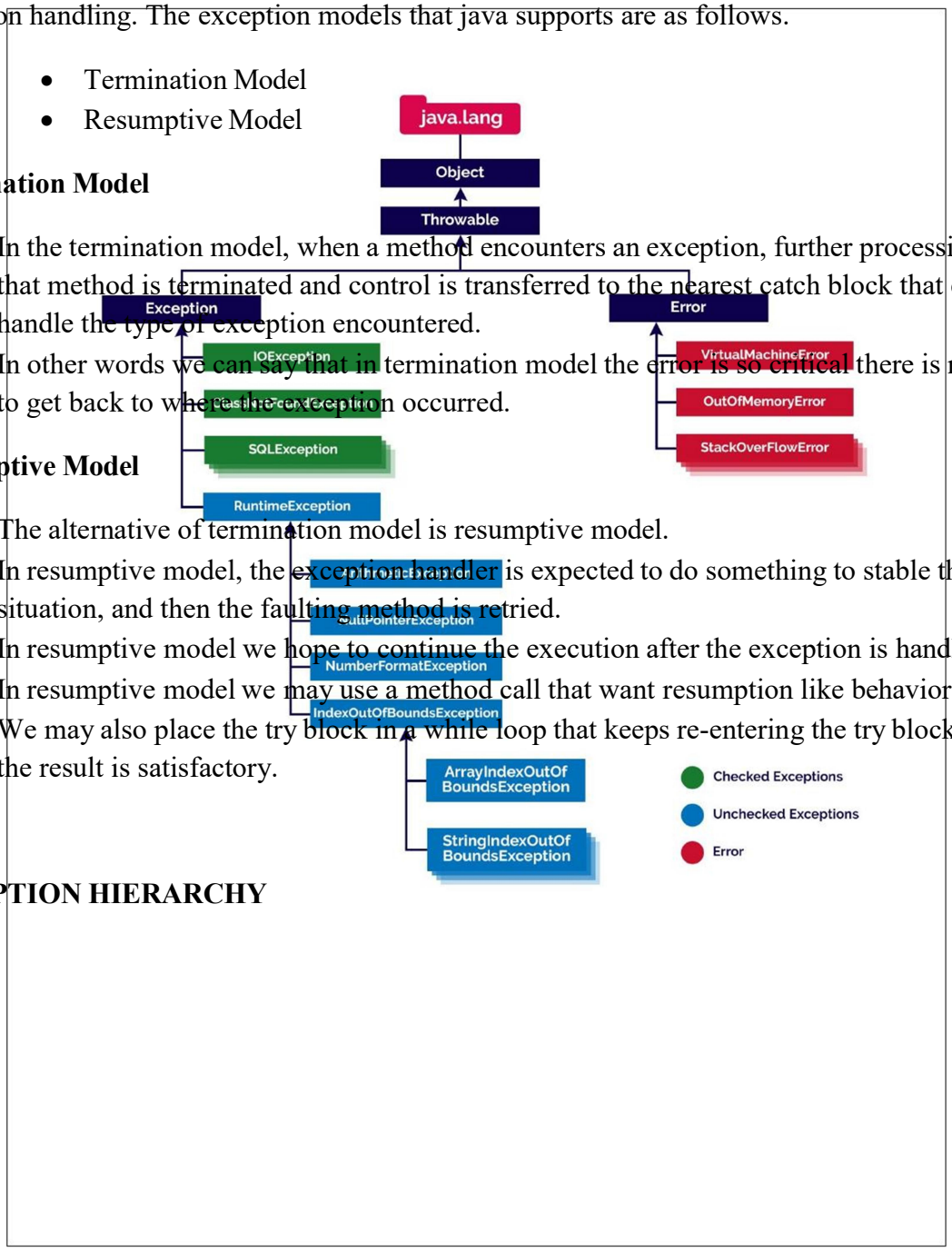
- Termination Model
- Resumptive Model

Termination Model

- In the termination model, when a method encounters an exception, further processing in that method is terminated and control is transferred to the nearest catch block that can handle the type of exception encountered.
- In other words we can say that in termination model the error is so critical there is no way to get back to where the exception occurred.

Resumptive Model

- The alternative of termination model is resumptive model.
- In resumptive model, the exception handler is expected to do something to stable the situation, and then the faulting method is retried.
- In resumptive model we hope to continue the execution after the exception is handled.
- In resumptive model we may use a method call that want resumption like behavior.
- We may also place the try block in a while loop that keeps re-entering the try block until the result is satisfactory.



EXCEPTION HIERARCHY

HOW TO HANDLE THE EXCEPTION

Use Five keywords for Handling the Exception

1. try
2. catch
3. throw
4. throws
5. finally

1. try block

- The try block contains set of statements where an exception can occur.
- In other words try block always contains problematic statements.
- A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or finally block or both.

Syntax :

```
try
{
 //statements that may cause an exception
}
```

2. catch block

- A catch block is where we handle the exceptions, this block must follow the try block.
- A single try block can have multiple catch blocks associated with it. We can catch different exceptions in different catch blocks.
- When an exception occurs in try block, the corresponding catch block that handles that particular exception executes.
- For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

Syntax of try-catch in java

```
try
{
 // statements causes problem at run time
}
catch(type of exception-1 object-1)
{
 // statements provides user friendly error message
}
catch(type of exception-2 object-2)
{
 // statements provides user friendly error message
}
```

Example1: ArithmeticException

```
class ExceptionDemo
{
public static void main(String[] args)
{
int a=30, b=0;
try
{
int c=a/b;
}
catch (ArithmeticException e)
{
System.out.println("Denominator should not be zero");
}
}}
```

Output: Denominator should not be zero

Example2: 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..

Example3: FileNotFoundException

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
class File_notFound_Demo
{
 public static void main(String args[])
 {
 try
 { // Following file does not exist
 File file = new File("E://file.txt");
 FileReader fr = new FileReader(file);
 }
catch(FileNotFoundException e)
{
 System.out.println("File does not exist");
```

```
 }
 }
}
```

Output: File does not exist

MULTIPLE CATCH BLOCKS

We can write multiple catch blocks for generating multiple user friendly error messages to make our application strong.

Example

```
class ExceptionDemo
{
public static void main(String[] args)
{
 int a=30, b=0;
try
{
int c=a/b;
System.out.println("Result: "+c);
}
catch(NullPointerException e)
{
 System.out.println("Enter valid number");
}
catch(ArithmeticException  e)
{
 System.out.println("Denominator not be zero");
```

}

}

}

NESTED TRY STATEMENTS

The try block within a try block is known as nested try block in java. Why use nested try block.Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax :

```
try {
  statement 1;
  statement 2;
  try
  {
  statement 1;
  statement 2;
  }
  catch(Exception e)
  {
  ……
  } } catch(Exception e) {
  ….....} Example
  class NestedTry
  { public static void main(String args[])
  {
  try
```

```
{
try
{
int a[]=new int[5];
a[5]=4;
}
catch(ArrayIndexOutOfBoundsException  e)
{
System.out.println(e);
}
}
catch(Exception e)
{
System.out.println("handeled");
} System.out.println("normal flow..");
}
}
```

3.throw

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

Syntax:

```
throw Instance
```

Example

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

Example

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

```
System.out.println("Caught in main.");
}
}
}
```

Output: Caught inside fun().

Caught in main.

4. throws

throws is a keyword in java language which is used to throw the exception which is raised in the called method to it's calling method throws keyword always followed by method signature.

Syntax

```
returnType methodName(parameter) throws Exception_class....
{
.....
}
```

Example

```
class ThrowsExecp
{
 static void fun() throws IllegalAccessException
 {
 System.out.println("Inside fun(). ");
 throw new IllegalAccessException("demo");
 }
 public static void main(String args[])
 {
```

```
try
{
fun();
}
catch(IllegalAccessException e)
{
System.out.println("caught in main.");
}
}
}
```

Output:

```
Inside fun().
caught in main.
```

5. finally Block

- Java finally block is a block that is used to execute important code such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.

Example

```
class TestFinallyBlock
{
public static void main(String args[])
{
try
{
int data=25/0;
```

```
System.out.println(data);
}
catch(NullPointerException e)
{
System.out.println(e);
}
finally
{
System.out.println("finally block is always executed");
} System.out.println("rest of the code...");
}
}
```

Output:

```
finally block is always executed
Exception in thread main java.lang.ArithmeticException:/ by zero
```

RE-THROWING EXCEPTIONS

- Sometimes we may need to rethrow an exception in Java.
- If a catch block cannot handle the particular exception it has caught, we can rethrow the exception.
- The rethrow expression causes the originally thrown object to be rethrown.
- Because the exception has already been caught at the scope in which the rethrow expression occurs, it is rethrown out to the next enclosing try block. Therefore, it cannot be handled by catch blocks at the scope in which the rethrow expression occurred.
- Any catch blocks for the enclosing try block have an opportunity to catch the exception.

Example

```
class RethrowExcep
{
```

```
static void fun()
{
try
{
throw new NullPointerException("demo");
}
catch(NullPointerException e)
{
System.out.println("Caught inside fun().");
throw e; // rethrowing the exception
}
}
public static void main(String args[])
{
try
{
fun();
}
catch(NullPointerException e)
{
System.out.println("Caught in main.");
}
}
}
```

Output:

Caught inside fun().

Caught in main.

CREATING OWN EXCEPTION(CUSTOM EXCEPTION IN JAVA)

If any exception is design by the user known as user defined or Custom Exception. Custom Exception is created by user. Rules to design user defined Exception

1. Create a package with valid user defined name.

2. Create any user defined class.

3. Make that user defined class as derived class of Exception or RuntimeException class.

4. Declare parametrized constructor with string variable.

5. call super class constructor by passing string variable within the derived class constructor.

6. Save the program with public class name.java

Example

```
package nage;
public class InvalidAgeException extends Exception
{
public InvalidAgeException (String s)
{
super(s);
}
}
```


A Class that uses above InvalidAgeException:

```
class CustomException
{
```

```
static void validate(int age) throws InvalidAgeException
{
if(age<18)
throw new InvalidAgeException("not valid");
else
System.out.println("welcome to vote");
}
public static void main(String args[])
{
try
{
validate(13);
}
catch(Exception m)
{
System.out.println("Exception occured: "+m);
}
System.out.println("rest of the code...");
}
}
```

Output:

```
Exception occured: InvalidAgeException:not valid
rest of the code...
```

BUILT IN EXCEPTIONS

The Java programming language has several built-in exception class that support exception handling.

Every exception class is suitable to explain certain error situations at run time.

All the built-in exception classes in Java were defined a package java.lang.

EXCEPTION TYPES IN JAVA

In java, exceptions are mainly categorized into two types, and they are as follows.

- Checked Exceptions
- Unchecked Exceptions

Checked Exceptions

- The checked exception is an exception that is checked by the compiler during the compilation process to confirm whether the exception is handled by the programmer or not. If it is not handled, the compiler displays a compilation error using built-in classes.
- The checked exceptions are generally caused by faults outside of the code itself like missingresources, networking errors, and problems with threads come to mind.
- The following are a few built-in classes used to handle checked exceptions in java.
- In the exception class hierarchy, the checked exception classes are the direct children of the Exception class.

List of checked exceptions in Java

1 ClassNotFoundException

It is thrown when the Java Virtual Machine (JVM) tries to load a particular class and the specified class cannot be found in the classpath.

2 CloneNotSupportedException

Used to indicate that the clone method in class Object has been called to clone an object, but that the object's class does not implement the Cloneable interface.

3 IllegalAccessException

It is thrown when one attempts to access a method or member that visibility qualifiers donot allow.

4 InstantiationException

It is thrown when an application tries to create an instance of a class using the newInstance method in class, but the specified class object cannot be instantiated because it is an interface or is an abstract class.

5 InterruptedException

It is thrown when a thread that is sleeping, waiting, or is occupied is interrupted.

6 NoSuchFieldException

It indicates that the class doesn't have a field of a specified name.

7 NoSuchMethodException

It is thrown when some JAR file has a different version at runtime that it had at compile time, a No Such Method Exception occurs during reflection when we try to access a method that does not exist.

Unchecked Exceptions

- The unchecked exception is an exception that occurs at the time of program execution. The unchecked exceptions are not caught by the compiler at the time of compilation.
- The unchecked exceptions are generally caused due to bugs such as logic errors, improper use of resources, etc.
- In the exception class hierarchy, the unchecked exception classes are the children of RuntimeException class, which is a child class of Exception class.

List of unchecked exceptions in Java

1 ArithmeticException

It handles the arithmetic exceptions like division by zero

2 ArrayIndexOutOfBoundsException

It handles the situations like 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 ArrayStoreException

It handles the situations like when an attempt has been made to store the wrong type of

object into an array of objects

5 ClassCastException

It handles the situation when we try to improperly cast a class from one type to another. 6

6 IllegalArgumentException

This exception is thrown in order to indicate that a method has been passed an illegal orinappropriate argument.

7 IllegalMonitorStateException

This indicates that the calling thread has attempted to wait on an object's monitor, or has attempted to notify other threads that wait on an object's monitor, without owning the specified monitor.

8 IllegalStateException

It signals that a method has been invoked at an illegal or inappropriate time.

9 IllegalThreadStateException

It is thrown by the Java runtime environment, when the programmer is trying to modify the state of the thread when it is illegal.

10 IndexOutOfBoundsException

It is thrown when attempting to access an invalid index within a collection, such as an array, vector , string , and so forth.

11 NegativeArraySizeException

It is thrown if an applet tries to create an array with negative size.

12 NullPointerException

it is thrown when program attempts to use an object reference that has the null value.

13 NumberFormatException

It is thrown when we try to convert a string into a numeric value such as float or integer, but the format of the input string is not appropriate or illegal.

14 SecurityException

It is thrown by the Java Card Virtual Machine to indicate a security violation.

15 StringIndexOutOfBounds

It is thrown by the methods of the String class, in order to indicate that an index is eithernegative, or greater than the size of the string itself.

16 UnsupportedOperationException

It is thrown to indicate that the requested operation is not supported.

STRING HANDLING

- A string is a sequence of characters surrounded by double quotations. In a java programming language, a string is the object of a built-in class String.
- The string created using the String class can be extended. It allows us to add more characters after its definition, and also it can be modified.

Example

```
String siteName = "javaprogramming";

siteName = "javaprogramminglanguage";
```

String handling methods

In java programming language, the String class contains various methods that can be used to handle string data values.

The following table depicts all built-in methods of String class in java.

1 charAt(int) Finds the character at given index

2 length() Finds the length of given string

3 compareTo(String) Compares two strings

4 compareToIgnoreCase(String) Compares two strings, ignoring case

5 concat(String) Concatenates the object string with argument string.

6 contains(String) Checks whether a string contains sub-string

7 contentEquals(String) Checks whether two strings are same

8 equals(String) Checks whether two strings are same

9 equalsIgnoreCase(String) Checks whether two strings are same, ignoring case

10 startsWith(String) Checks whether a string starts with the specified string

11 isEmpty() Checks whether a string is empty or not

12 replace(String, String) Replaces the first string with second string

13replaceAll(String, String)Replaces the first string with second string at all occurrences.

14substring(int, int)Extracts a sub-string from specified start and end index values

15 toLowerCase() Converts a string to lower case letters

16 toUpperCase() Converts a string to upper case letters

17 trim() Removes whitespace from both ends

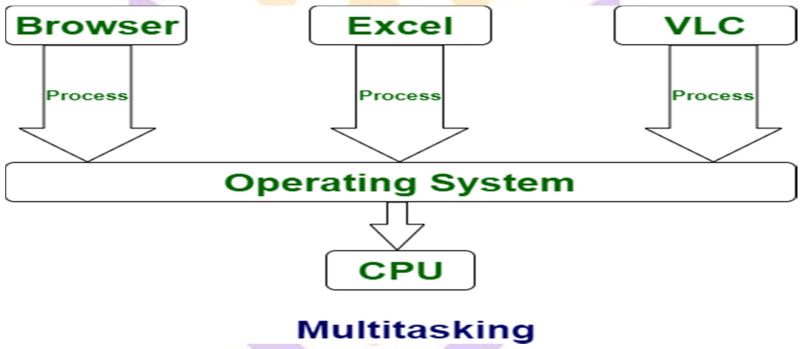
18 toString(int) Converts the value to a String object

Example

```
public class JavaStringExample
{
public static void main(String[] args)
 {
String title = "Java Programming";
String siteName = "String Handling Methods";
System.out.println("Length of title: " + title.length());
System.out.println("Char at index 3: " + title.charAt(3));
System.out.println("Index of 'T': " + title.indexOf('T'));
System.out.println("Empty: " + title.isEmpty());
System.out.println("Equals: " + siteName.equals(title));
System.out.println("Sub-string: " + siteName.substring(9, 14));
System.out.println("Upper case: " + siteName.toUpperCase());
 }
}
```

MULTI-TASKING AND MULTI-THREADING

- Multi-tasking and multi-threading are two techniques used in operating systems to manage multiple processes and tasks.
- Multi-tasking is the ability of an operating system to run multiple processes or tasks concurrently, sharing the same processor and other resources.
- In multi-tasking, the operating system divides the CPU time between multiple tasks, allowing them to execute simultaneously.

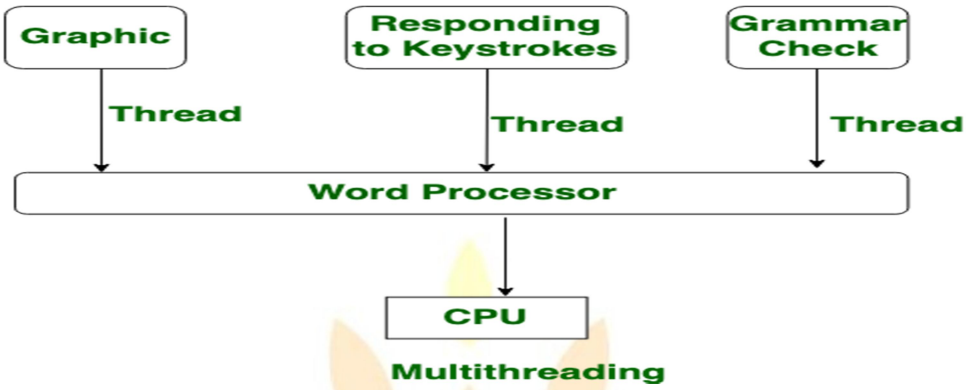


Multitasking

- Each task is assigned a time slice, or a portion of CPU time, during which it can execute its code.
- Multi-tasking is essential for increasing system efficiency, improving user productivity, and achieving optimal resource utilization.

Multi-threading is a technique in which an operating system divides a single process into multiple threads, each of which can execute concurrently.

- Threads share the same memory space and resources of the parent process, allowing them to communicate and synchronize data easily.
- Multi-threading is useful for improving application performance by allowing different parts of the application to execute simultaneously.



DIFFERENCE BETWEEN MULTI-TASKING AND MULTI-THREADING

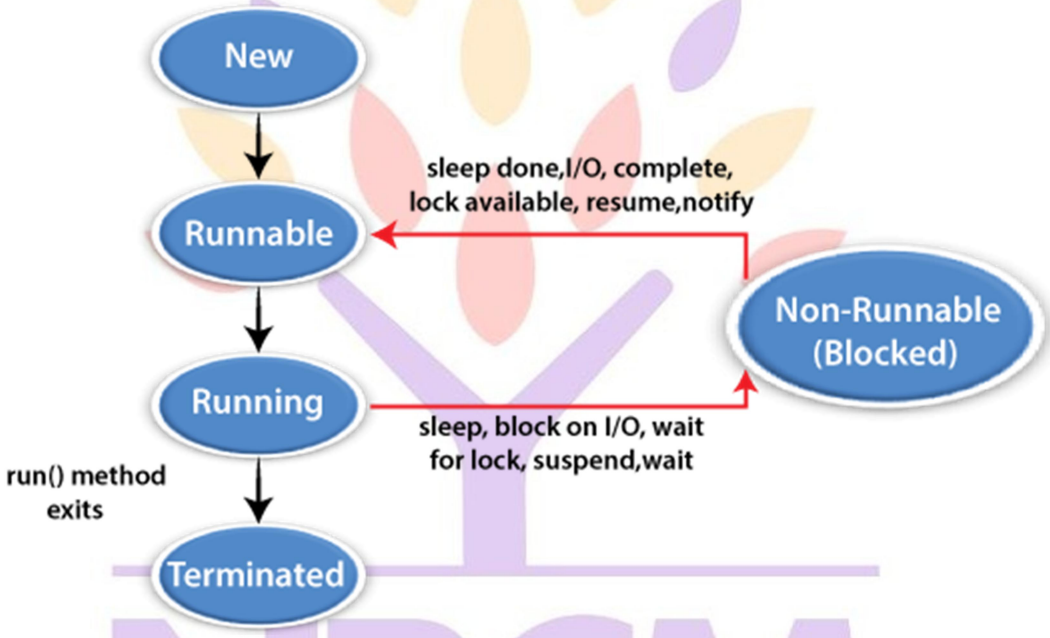
Process-based Multitasking	Thread-based Multitasking
Two or more processes/programs can be run concurrently.	Two or more threads can be run concurrently.
The smallest unit of execution is a process.	The smallest unit of execution is a thread.
Inter-process communication is costly and inefficient.	Inter-thread communication is inexpensive and efficient.
Processes take more time for context switching.	Threads take less time for context switching.
It is unable to gain access over CPU idle time.	It can gain access over CPU idle time.
Every program has its own address space.	Threads share the same address space.
Overhead is more.	Overhead is less.

JAVA THREAD MODEL (LIFE CYCLE OF A THREAD)

- In java, a thread goes through different states throughout its execution.
- These stages are called thread life cycle states or phases.
- A thread can be in one of the five states in the thread.
- The life cycle of the thread is controlled by JVM.

The thread states are as follows:

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



1. New

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

Example

```
Thread t1 = new Thread();
```

2. Runnable

- When a thread calls start( ) method, then the thread is said to be in the Runnable state.
- This state is also known as a Ready state.

Example

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

3. Running

When a thread calls run( ) method, then the thread is said to be Running. The run( ) method of a thread called automatically by the start( ) method.

4. Non-Runnable (Blocked)

- This is the state when the thread is still alive, but is currently not eligible to run.
- A thread in the Running state may move into the blocked state due to various reasons like sleep( )method called, wait( ) method called, suspend( ) method called, and join( ) method called, etc.
- When a thread is in the blocked or waiting state, it may move to Runnable state due to reasons like sleep time completed, waiting time completed, notify( ) or notifyAll( ) method called, resume( ) method called, etc.

Example

```
Thread.sleep(1000);

wait(1000);

wait();

suspend();

notify();

notifyAll();

resume();
```

5. Terminated

- A thread in the Running state may move into the dead state due to either its execution completed or the stop( ) method called.
- The dead state is also known as the terminated state.

CREATING THREADS

There are two ways to create a thread:

1. By extending Thread class

2. By implementing Runnable interface.

1. By extending Thread class

The java contains a built-in class Thread inside the java.lang package. The Thread class contains all the methods that are related to the threads.

To create a thread using Thread class, follow the step given below.

Step-1: Create a class as a child of Thread class. That means, create a class that extends Thread class.

Step-2: Override the run( ) method with the code that is to be executed by the thread. The run( ) method must be public while overriding.

Step-3: Create the object of the newly created class in the main( ) method.

Step-4: Call the start( ) method on the object created in the above step.

Example: By extending Thread class

```
class SampleThread extends Thread
{
public void run()
{
System.out.println("Thread is under Running...");
for(int i= 1; i<=10; i++)
{
System.out.println("i = " + i);
}
}
}
public class My_Thread_Test
{
```

```
public static void main(String[] args)
{
SampleThread t1 = new SampleThread();
System.out.println("Thread about to start...");
t1.start();
}
}
```

Output: Thread about to start...

Thread is under Running... i = 1

i = 2

i = 3

i = 4

i = 5

i = 6

i = 7

i = 8

i = 9

i = 10

2. By implementing Runnable interface

- The java contains a built-in interface Runnable inside the java.lang package.
- The Runnable interface implemented by the Thread class that contains all the methods that are related to the threads.

To create a thread using Runnable interface, follow the step given below.

Step-1: Create a class that implements Runnable interface.

Step-2: Override the run( ) method with the code that is to be executed by the thread. The run( )method must be public while overriding.

Step-3: Create the object of the newly created class in the main( ) method.

Step-4: Create the Thread class object by passing above created object as parameter to the Thread class constructor.

Step-5: Call the start( ) method on the Thread class object created in the above step.

Example: By implementing the Runnable interface

```
class SampleThread implements Runnable
{
public void run()
{
System.out.println("Thread is under Running...");
for(int i= 1; i<=10; i++)
{
System.out.println("i = " + i);
}
}
}
public class My_Thread_Test
{
public static void main(String[] args)
{
SampleThread threadObject = new SampleThread();
Thread thread = new Thread(threadObject);
System.out.println("Thread about to start...");
thread.start();
}
```

}

Output:

Thread about to start...

Thread is under Running... i = 1

i = 2

i = 3

i = 4

i = 5

i = 6

i = 7

i = 8

i = 9

i = 10

CONSTRUCTORS OF THREAD CLASS

1. Thread()
2. Thread(String name)
3. Thread(Runnable r)
4. Thread(Runnable r,String name)

METHODS OF THREAD CLASS

1. public void run(): is used to defines actual task of the thread.

2. public void start():It moves the thread from Ready state to Running state by calling run( ) method.

3. public void sleep(long milliseconds): Moves the thread to blocked state till the specified number of milliseconds.

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

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

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 suspend(): is used to suspend the thread(deprecated).

SLEEP() & JOIN()

```
class SampleThread extends Thread
{
public void run()
{
System.out.println("Thread is under Running...");
for(int i= 1; i<=10; i++)
{
try
{
Thread.sleep(1000);
}
catch(Exception e)
{
```

```
System.out.println(e);
}
System.out.println("i = " + i);
}
}
}
public class My_Thread_Test
{
public static void main(String[] args)
{
SampleThread t1 = new SampleThread();
SampleThread t2 = new SampleThread();
SampleThread t3 = new SampleThread();
System.out.println("Thread about to start...");
t1.start();
try
{
t1.join();
}
catch(Exception e)
{
System.out.println(e);
}
t2.start();
```

```
t3.start();
}
}
```

THREAD PRIORITIES

- In a java programming language, every thread has a property called priority.
- Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling).
- The thread with more priority allocates the processor first.
- But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

Three constants defined in Thread class:

1. MIN_PRIORITY
2. NORM_PRIORITY
3. 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.
- The java programming language Thread class provides two methods setPriority(int), and getPriority( ) to handle thread priorities.

setPriority( ) method

The setPriority( ) method of Thread class used to set the priority of a thread.

It takes an integer range from 1 to 10 as an argument and returns nothing (void).

Example

```
threadObject.setPriority(4);
```


or

```
threadObject.setPriority(MAX_PRIORITY);
```

getPriority( ) method

The getPriority( ) method of Thread class used to access the priority of a thread.

It does not takes any argument and returns name of the thread as String.

Example

String threadName = threadObject.getPriority();

Example1

```
class SampleThread extends Thread
{
 public void run()
 {
 System.out.println("Inside SampleThread");
 System.out.println("CurrentThread: " + Thread.currentThread().getName());
 }
}
public class My_Thread_Test {
public static void main(String[] args)
{
SampleThread threadObject1 = new SampleThread();
SampleThread threadObject2 = new SampleThread();
threadObject1.setName("first");
threadObject2.setName("second");
threadObject1.setPriority(4);
threadObject2.setPriority(Thread.MAX_PRIORITY);
threadObject1.start();
threadObject2.start();
}
```

```
}
```

Output:

```
Inside SampleThread
Inside SampleThread
CurrentThread: second
CurrentThread: first
```

Example2

```
class MultiThread extends Thread
{
 public void run()
 {
 System.out.println("running thread name is:"+Thread.currentThread().getName());
 System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
 }
 public static void main(String args[])
 {
 MultiThread m1=new MultiThread ();
 MultiThread m2=new MultiThread ();
 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

SYNCHRONIZING THREADS

SYNCHRONIZATION

- The java programming language supports multithreading.
- The problem of shared resources occurs when two or more threads get execute at the same time.
- In such a situation, we need some way to ensure that the shared resource will be accessed by only one thread at a time, and this is performed by using the concept called synchronization.
- The synchronization is the process of allowing only one thread to access a shared resource at a time.

UNDERSTANDING THE PROBLEM WITHOUT SYNCHRONIZATION

In this example, there is no synchronization, so output is inconsistent.

Example:

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

```
}
catch(Exception e)
{
System.out.println(e);
}
}
}
}
class MyThread1 extends Thread
{ Table t; MyThread1(Table t)
{
this.t=t; } public void run()
{
t.printTable(5); }
}
class MyThread2 extends Thread
{ Table t; MyThread2(Table t)
{
this.t=t; } public void run()
{
t.printTable(100); }
}
class TestSynchronization
{
```

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

Output:

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

Thread Synchronization

In java, the synchronization is achieved using the following concepts.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.

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

Java synchronized method

- If you declare any method as synchronized, it is known as synchronized method.
- When a method created using a synchronized keyword, it allows only one object to access it at a time.
- When an object calls a synchronized method, it put a lock on that method so that other objects or thread that are trying to call the same method must wait, until the lock is released.
- Once the lock is released on the shared resource, one of the threads among the waiting threads will be allocated to the shared resource.
- In the above image, initially the thread-1 is accessing the synchronized method and other threads (thread-2, thread-3, and thread-4) are waiting for the resource (synchronized method).
- When thread-1 completes it task, then one of the threads that are waiting is allocated with the synchronized method, in the above it is thread-3.

Example:

```
class Table
    {
    synchronized void printTable(int n)
    {
    for(int i = 1; i <= 10; i++)
    System.out.println(n + " * " + i + " = " + i*n);
    }
    }
    class MyThread1 extends Thread
```

```
{
Table table = new Table();
int number;
MyThread1(Table table, int number)
{
this.table = table;
this.number = number;
}
public void run()
{
table.printTable(number);
}
}
class MyThread2 extends Thread
{
Table table = new Table();
int number;
MyThread2(Table table, int number)
{
this.table = table;
this.number = number;
}
public void run()
{
```

```
table.printTable(number);
}
}
class ThreadSynchronizationExample
{
public static void main(String[] args)
{
Table table = new Table();
MyThread1 thread1 = new MyThread1(table, 5);
MyThread2 thread2 = new MyThread2(table, 10);
thread1.start();
thread2.start();
}
}
```

Output:

```
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
```

5 * 10 = 50

10 * 1 = 10

10 * 2 = 20

10 * 3 = 30

10 * 4 = 40

10 * 5 = 50

10 * 6 = 60

10 * 7 = 70

10 * 8 = 80

10 * 9 = 90

10 * 10 = 100

Synchronized Block in Java

- The synchronized block is used when we want to synchronize only a specific sequence of lines in a method.
- For example, let's consider a method with 20 lines of code where we want to synchronize only a sequence of 5 lines code, we use the synchronized block.
- If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

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

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 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
```

INTERTHREAD COMMUNICATION

- Inter thread communication is the concept where two or more threads communicate to solve the problem of polling.
- In java, polling is the situation to check some condition repeatedly, to take appropriate action, once the condition is true.
- That means, in inter-thread communication, a thread waits until a condition becomes true such that other threads can execute its task.
- The inter-thread communication allows the synchronized threads to communicate with each other.

Java provides the following methods to achieve inter thread communication.

- void wait( ) It makes the current thread to pause its execution until other thread in the same monitor calls notify( )
- void notify( ) It wakes up the thread that called wait( ) on the same object.
- void notifyAll() It wakes up all the threads that called wait( ) on the same object.

Let's look at an example problem of producer and consumer.

- The producer produces the item and the consumer consumes the same.
- But here, the consumer cannot consume until the producer produces the item, and producer cannot produce until the consumer consumes the item that already been produced.
- So here, the consumer has to wait until the producer produces the item, and the producer also needs to wait until the consumer consumes the same.
- Here we use the inter-thread communication to implement the producer and consumer problem.

Example

```
class ItemQueue
{
int item;
boolean valueSet = false;
synchronized int getItem()
{
while (!valueSet)
try
 {
 wait();
 }
 catch (InterruptedException e)
 {
 System.out.println("InterruptedException  caught");
```

```
 }
System.out.println("Consumed:" + item); valueSet = false;
try
 {
 Thread.sleep(1000);
}
 catch (InterruptedException e)
 {
 System.out.println("InterruptedException  caught");
 }
notify();
return item;
}
synchronized void putItem(int item)
 {
while (valueSet)
try
 {
 wait();
}
 catch (InterruptedException e)
 {
 System.out.println("InterruptedException  caught");
}
```

```
this.item = item; valueSet = true;
System.out.println("Produced: " + item);
try
 {
 Thread.sleep(1000);
}
 catch (InterruptedException e)
{
 System.out.println("InterruptedException  caught");
}
notify();
}
}
class Producer implements Runnable
{
ItemQueue itemQueue;
Producer(ItemQueue itemQueue)
 {
this.itemQueue = itemQueue;
new Thread(this, "Producer").start();
}
public void run()
 {
int i = 0;
```

```
while(true)
 {
 itemQueue.putItem(i++);
}
}
}
class Consumer implements Runnable
{
ItemQueue itemQueue;
Consumer(ItemQueue itemQueue)
 {
this.itemQueue = itemQueue;
new Thread(this, "Consumer").start();
}
public void run()
 {
while(true)
 {
 itemQueue.getItem();
}
}
}
class ProducerConsumer
{
```

```
public static void main(String args[])
 {
ItemQueue itemQueue = new ItemQueue();
new Producer(itemQueue);
new Consumer(itemQueue);
}
}
```

THREADGROUP IN JAVA

- Java provides a convenient way to group multiple threads in a single object. In such a way, we can suspend, resume or interrupt a group of threads by a single method call.
- Java thread group is implemented by java.lang.ThreadGroup class.
- A Thread Group represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.
- A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

ThreadGroup Example

```
public class ThreadGroupDemo implements Runnable
{
 public void run()
 {
 System.out.println(Thread.currentThread().getName());
 }
 public static void main(String[] args)
 {
 ThreadGroupDemo r = new ThreadGroupDemo();
 ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");
```

```
Thread t1 = new Thread(tg1, r,"one");
t1.start();
Thread t2 = new Thread(tg1, r,"two");
t2.start();
Thread t3 = new Thread(tg1, r,"three");
t3.start();
System.out.println("Thread Group Name: "+tg1.getName());
}
}
```

Output :

```
one
two
three
Thread Group Name: Parent ThreadGroup
```

DAEMON THREAD IN JAVA

- In Java, daemon threads are low-priority threads that run in the background to perform tasks such as garbage collection or provide services to user threads.
- The life of a daemon thread depends on the mercy of user threads, meaning that when all user threads finish their execution, the Java Virtual Machine (JVM) automatically terminates the daemon thread.
- To put it simply, daemon threads serve user threads by handling background tasks and have no role other than supporting the main execution. Methods for Java Daemon thread by Thread class The java.lang.Thread class provides two methods for java daemon thread.

1 public void setDaemon(boolean status) is used to mark the current thread as daemon thread or user thread.

2 public boolean isDaemon() is used to check that current is daemon.

Example

```
public class TestDaemonThread1 extends Thread
```

```
{
public void run()
{
if(Thread.currentThread().isDaemon())
{
System.out.println("daemon thread work");
}
else
{
System.out.println("user thread work");
}
}
public static void main(String[] args)
{
TestDaemonThread1 t1=new TestDaemonThread1(); //creating thread
TestDaemonThread1 t2=new TestDaemonThread1();
TestDaemonThread1 t3=new TestDaemonThread1();
t1.setDaemon(true); //now t1 is daemon thread
t1.start(); //starting threads
t2.start();
t3.start();
} }
```

JAVA ENUMS

- The Enum in Java is a data type which contains a fixed set of constants.
- It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc.
- According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.
- Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change).
- The Java enum constants are static and final implicitly.
- Enums are used to create our own data type like classes.
- The enum data type (also known as Enumerated Data Type) is used to define an enum in Java.
- Java Enum internally inherits the Enum class, so it cannot inherit any other class, but it can implement many interfaces.
- We can have fields, constructors, methods, and main methods in Java enum.

Example

```
class EnumExample1

{ public enum Season { WINTER, SPRING, SUMMER, FALL }

public static void main(String[] args)

{

for (Season s : Season.values())

System.out.println(s); }

}
```

AUTOBOXING

- The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing.
- So java programmer doesn't need to write the conversion code.

Advantage

No need of conversion between primitives and Wrappers manually so less coding is required.

Example

```
class BoxingExample1
{
public static void main(String args[])
{
int a=50;
Integer a2=new
Integer(a);
//Boxing Integer
a3=5; //Boxing
System.out.println(
a2+" "+a3);
} }
```

Output: 50 5

JAVA ANNOTATIONS

- Java Annotation is a tag that represents the metadata i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.
- Annotations in Java are used to provide additional information, so it is an alternative option for XML and Java marker interfaces.

Example

```
@Override
@SuppressWarnings @Deprecated
```


@Override

- @Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.

- Sometimes, we does the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurity that method is overridden.