Exception Handling

Exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Exception is an abnormal condition.

In Java, The **Exception Handling** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.

Exception Handling removes runtime error in the java program.It is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, etc.

Advantage of Exception Handling

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

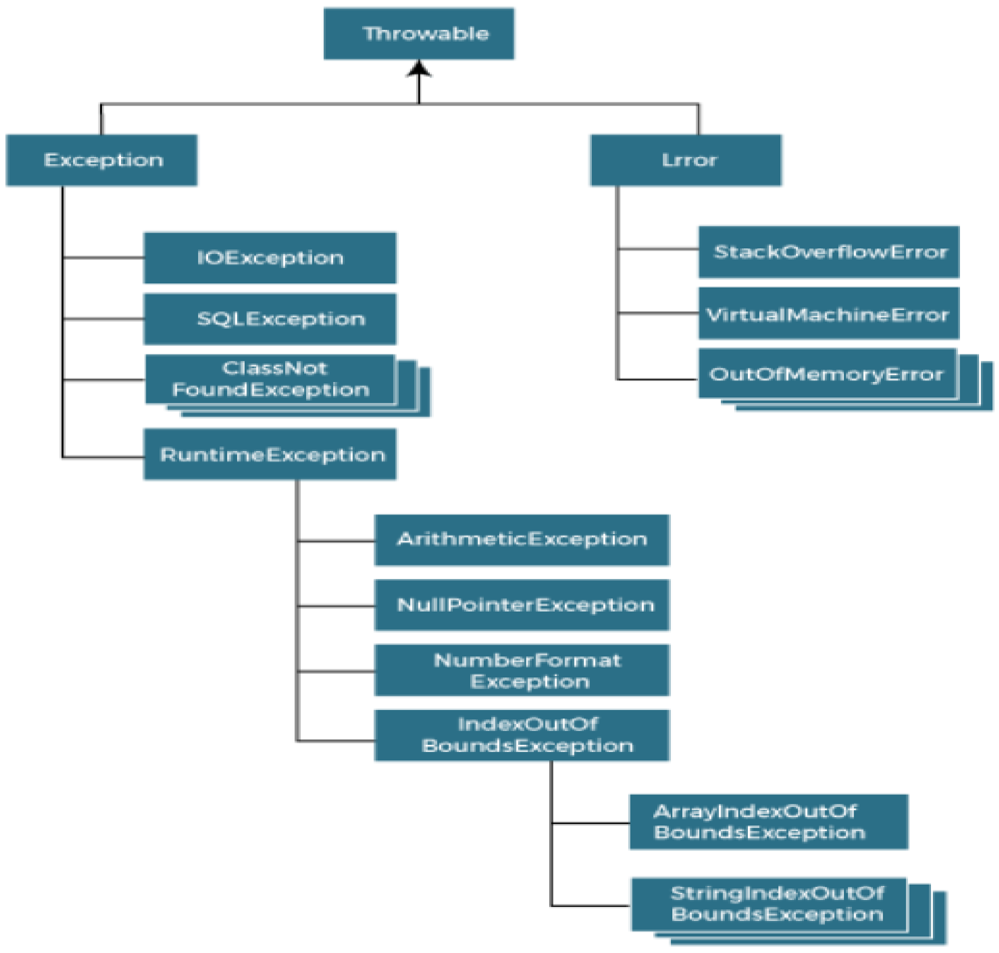
Let's consider a scenario:

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

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

Hierarchy of Java Exception classes

Throwable class is the root class of Java Exception classes in Exception handling. The hierarchy of Java Exception classes is given below:



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked.

1. Checked Exception
2. Unchecked Exception

1) Checked Exception

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

The exceptions which are checked by the compiler at compile time is called Checked Exceptions

2) Unchecked Exception

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

The exceptions which are unchecked by the compiler at compile time is called Checked Exceptions

Keywords used in exception handling

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

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

1) try Block: Used to define a block of code that might throw an exception.

Syntax

```
try {
   // Code that may throw an exception
}
```

2) catch Block: The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.

Syntax:

```
try {
   // Code that may throw an exception
} catch (ExceptionType e) {
   // Code to handle the exception
}
```

3) finally: The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is occurred or not. Typically used for cleanup code.

```
try {
   // Code that may throw an exception
} catch (ExceptionType e) {
   // Code to handle the exception
} finally {
   // Code to be executed regardless of exception
}
```

4) throw: The "throw" keyword is used to generate an exception manually.
Syntax:
throw new ExceptionType("Error message");
ex: throw new ArithmeticException("/zero")

5)throws: The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example

Ex:

```
public class ExceptionDemo {
        public static void main(String args[]) {
                try {
                        int data = 100 / 0;
                } catch (ArithmeticException e) {
                        System.out.println("Division by zero");
                }
                System.out.println("rest of the code...");
        }
}
```

Ex2:

```
public class TryCatchExample {

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

}
```

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

1) A scenario where ArithmeticException occurs
If we divide any number by zero, there occurs an ArithmeticException.

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

2) A scenario where NullPointerException occurs
If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

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

3) A scenario where NumberFormatException occurs
Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

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

4) A scenario where ArrayIndexOutOfBoundsException occurs
When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. Consider the following statements.

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

Java multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Ex:

```
    public class MultipleCatchBlock {

        public static void main(String[] args) {

             try{
                 int a[]=new int[5];

                 System.out.println(a[10]);
                }
                catch(ArithmeticException e)
                  {
                   System.out.println("Arithmetic Exception occurs");
                  }
                catch(ArrayIndexOutOfBoundsException e)
                  {
                   System.out.println("ArrayIndexOutOfBounds Exception occurs");
                  }
                catch(Exception e)
                  {
                   System.out.println("Parent Exception occurs");
                  }
                System.out.println("rest of the code");
        }
    }
```

Output:

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

finally block

Java finally block is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

```
class TestFinallyBlock {
        public static void main(String args[]) {
                try {
                        int data = 25 / 5;
                        System.out.println(data);
                }

                catch (NullPointerException e) {
                        System.out.println("This block will not be executed");
                }

                finally {
                        System.out.println("finally block is always executed");
                }

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

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

throw keyword

The Java throw keyword is used to throw an exception explicitly. We specify the **exception** object which is to be thrown. We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

Syntax:
```
 throw ThrowableInstance;
```
Ex1: ArithmeticException a=new ArithmeticException("/zero");
```
      throw a;
```
Ex2 :
```
public class TestThrow {
        public static void validate(int age) {
                if (age < 18) {
                throw new ArithmeticException("Person is not eligible to vote");
                } else {
                        System.out.println("Person is eligible to vote!!");
                }
        }
        public static void main(String args[]) {
                validate(13);
                System.out.println("rest of the code...");
        }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to vote
```

Exception Propagation

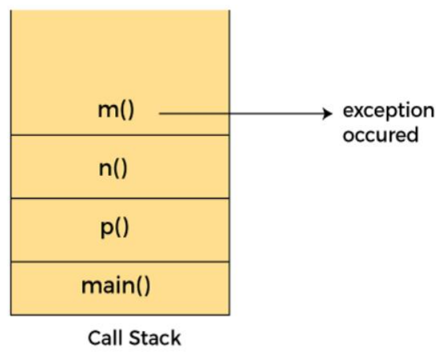
Exception propagation in Java is the process where an exception is thrown from the top of the call stack and, if not caught, moves down to the previous method in the call stack, continuing this way until it is caught or reaches the bottom of the call stack.

```
class TestExceptionPropagation{
  void m(){
    int data=50/0;
  }
  void n(){
    m();
  }
  void p(){
   try{
    n();
   }catch(Exception e){System.out.println("exception handled");}
  }
  public static void main(String args[]){
   TestExceptionPropagation obj=new TestExceptionPropagation();
   obj.p();
   System.out.println("normal flow...");
  }
}
```

Output:

```
exception handled
normal flow…
```

Exception can be handled in any method in call stack either in the main() method, p() method, n() method or m() method.



throws keyword

The **throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained. Exception Handling is mainly used to handle the checked exceptions.

Syntax of Java throws

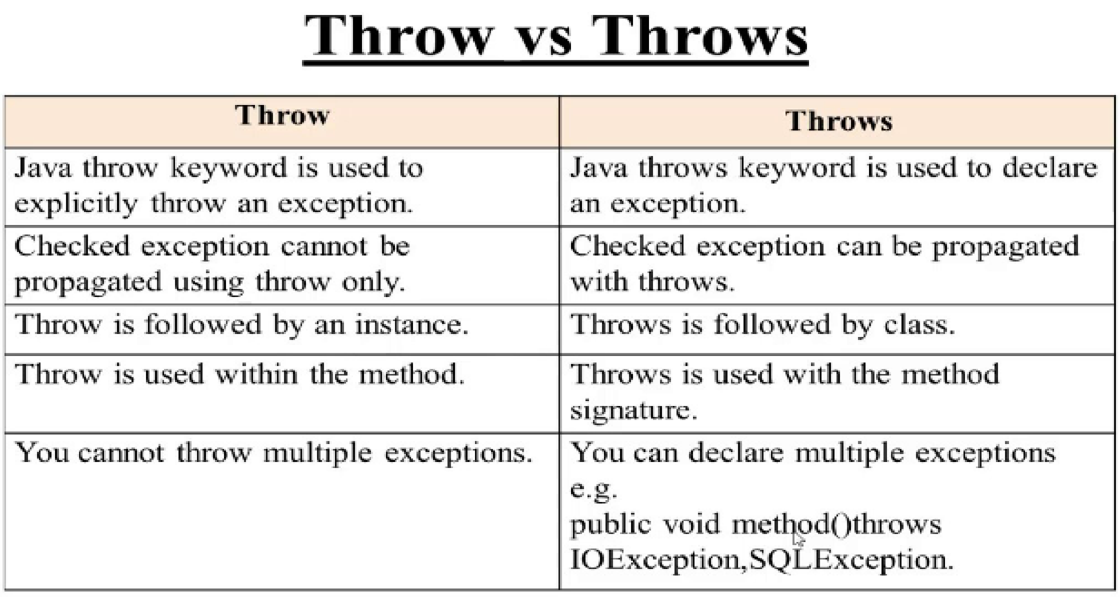
```
return_type method_name() throws exception_class_name{
//method code
}
```

Ex:

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

Output:

```
exception handled
normal flow
```



Throw vs Throws

Throw	Throws
Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
Throw is followed by an instance.	Throws is followed by class.
Throw is used within the method.	Throws is used with the method signature.
You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

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

Ex:
Step 1: Define the Custom Exception

```
public class InvalidAgeException extends Exception {
        public InvalidAgeException(String message) {
                super(message);
        }
}
```

Step 2: Use the Custom Exception

```
public class CustomException {
        public static void main(String[] args) {
                try {
                        checkAge(17);
                } catch (InvalidAgeException e) {
                        System.out.println("Caught InvalidAgeException: " + e.getMessage());
                }
        }

        public static void checkAge(int age) throws InvalidAgeException {
                if (age < 18) {
                        throw new InvalidAgeException("Person is not eligible for vote");
                } else
                        System.out.println("Person is eligible for vote");
        }
}
```

String handling

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.
Creation of a string object can be done two ways.

1. By string literal
2. By new keyword

1) String Literal

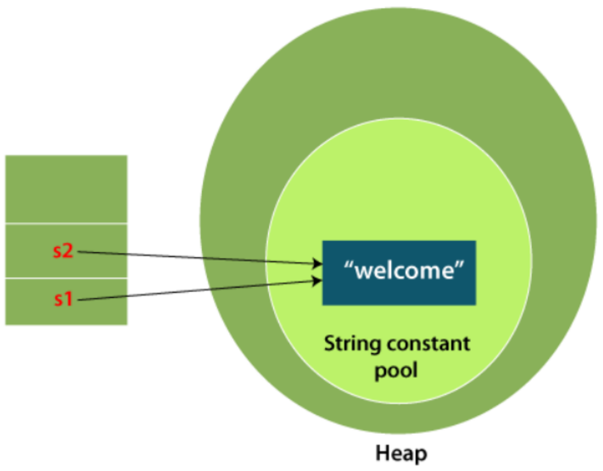
Java String literal is created by using double quotes. For Example:

```
String s="welcome";
```

For every time we are creating a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool.

```
String s1="Welcome";
String s2="Welcome";//It doesn't create a new instance
```

The String class is immutable, means that once a String is declared it’s content cannot be changed at any time.



2) By new keyword
Ex:
```
String s1=new String("example");
String s2=new String("hyderabad”);
```
For each object separate memory allocation will be done.

String class methods

1	char charAt(int index) : It returns char value for the particular index
2	int length() :It returns string length
3	String substring(int beginIndex) : It returns substring for given begin index.
4	String substring(int beginIndex, int endIndex): It returns substring for given begin index and end index.
5	boolean contains() : It returns true or false after matching the sequence of char value.
6	boolean equals(Object another) : It checks the equality of string with the given object.
7	boolean isEmpty() :It checks if string is empty.
8	String concat(String str) :It concatenates the specified string.
9	String replace(char old, char new) : It replaces all occurrences of the specified char value.
10	int indexOf(int ch) It returns the specified char value index.
11	int indexOf(int ch, int fromIndex) : It returns the specified char value index starting with given index.
12	String toLowerCase() :It returns a string in lowercase.
13	String toUpperCase() :It returns a string in uppercase.
14	String trim() :It removes beginning and ending spaces of this string.
15	static String valueOf(int value): It converts given type into string. It is an overloaded method.

Immutable String in Java

In Java, **String objects are immutable**. Immutable means unmodifiable or unchangeable.

Once String object is created its data or state can't be changed but a new String object is created.

```
class Testimmutablestring{
 public static void main(String args[]){
   String s="cse";
   s.concat(" aiml”);//concat() method appends the string at the end
   System.out.println(s);//will print cse because strings are immutable objects
 }
}
```

String comparison

```
class Teststringcomparison1{
 public static void main(String args[]){
   String s1="Hello";
```

```
        String s2="Hello";
        String s3=new String("Hello");
        String s4="Welcome";
        System.out.println(s1.equals(s2));//true
        System.out.println(s1.equals(s3));//true
        System.out.println(s1.equals(s4));//false
        System.out.println(s1==s2);//true (because both refer to same instance)
        System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)

     }
    }
```

StringBuffer

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

Ex: StringBuffer sb=new StringBuffer(“Hello’);

append (): Adds a string to the end.
Ex: sb.append(" World"); // "Hello World"

insert (int offset, String str): Inserts a string at the specified position.
Ex: sb.insert(6, "Beautiful "); // "Hello Beautiful World"

delete (int start, int end): Removes a substring from the specified range.
Ex: sb.delete(6, 16); // "Hello World"

reverse (): Reverses the string.
Ex: sb.reverse(); // "dlroW olleH"

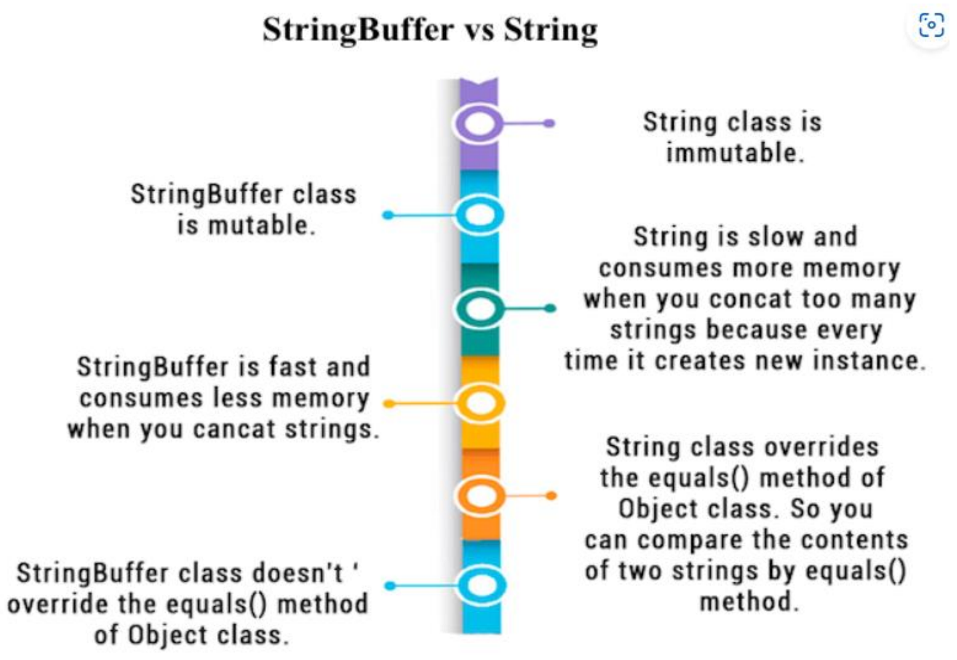
capacity (): The capacity() method of the StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16.
Ex: StringBuffer s1=new StringBuffer()// s1.capacity() returns 16

Java StringBuilder Class

Java StringBuilder class is used to create mutable (modifiable) String. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

the methods of StringBuilder and StringBuffer in Java are essentially the same.

Difference between String and StringBuffer


StringBuffer vs String
String class is immutable.
StringBuffer class is mutable.
String is slow and consumes more memory when you concat too many strings because every time it creates new instance.
StringBuffer is fast and consumes less memory when you cancat strings.
String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.
StringBuffer class doesn't ' override the equals() method of Object class.

StringBuffer	StringBuilder
StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.
StringBuffer was introduced in Java 1.0	StringBuilder was introduced in Java 1.5

Multithreading in Java

Multithreading is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing.

A Thread is a flow of execution, it performs a special task. It is a separate path of execution.

Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Differences between multithreading and multitasking

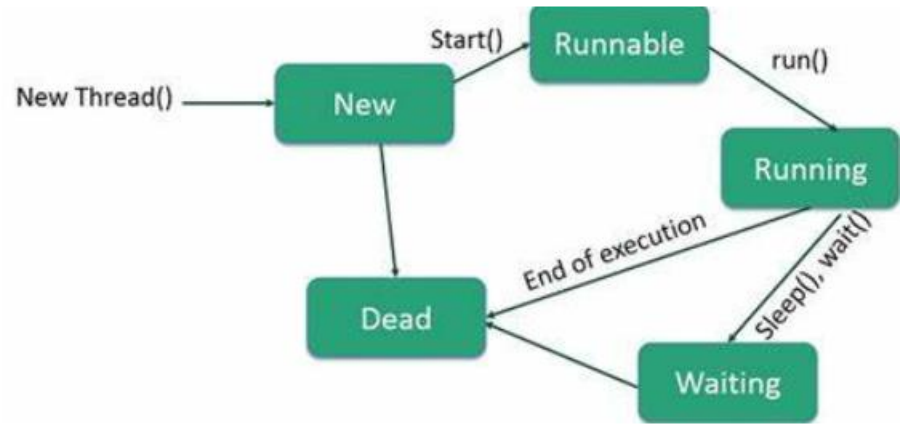
1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

Life Cycle of a Thread



A Thread life cycle contains several states.

New (or Born) State:

when an instance of the Thread class is created Thread will be entered in born state.
Ex : Thread thread = new Thread();

Runnable State:

After invoking the start() method, the thread moves to the runnable state.
Ex: thread.start();

Running State:

When the thread scheduler selects a thread from the runnable pool and the thread starts executing its run() method, it enters the running state.

```
public void run() {
   // task of a Thread
}
```

Blocked (or Waiting) State:

When we call wait() or sleep() methods a thread enters into Blocked state.

Thread.sleep(1000)

Dead State:

A thread enters this state when its run() method completes or when stop() method is invoked. Once terminated, a thread cannot be restarted.

Methods of Thread class

1)void start():It is used to start the execution of the thread.
2)void run() : It is used to perform an action for a thread.
3)static void sleep() :It sleeps a thread for the specified amount of time.
4)static Thread currentThread() :It returns a reference to the currently executing thread object.
5)void join() : It waits for a thread to die.
6)int getPriority(): It returns the priority of the thread.
7)void setPriority(): It changes the priority of the thread.
8)String getName() : It returns the name of the thread.
9)void setName() : It changes the name of the thread.
10)long getId() : It returns the id of the thread.
11)Boolean isAlive() :It tests if the thread is alive.

12)static void yield() :It causes the currently executing thread object to pause and allow other threads to execute temporarily.

13)void suspend() : It is used to suspend the thread.
14)void resume() : It is used to resume the suspended thread.
15)void stop() : It is used to stop the thread.

16) void notify() : It is used to give the notification for only one thread which is waiting for a particular object.
17) void notifyAll() : It is used to give the notification to all waiting threads of a particular object.

Java Threads -How to create a thread in Java

There are two ways to create a thread:

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

1.By extending Thread class

Commonly used Constructors of Thread class:

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

Ex :
```
class MyThread extends Thread {
        public void run() {
                System.out.println("thread is running...");
          }
      }

class ThreadDemo {
        public static void main(String args[]) {
                MyThread t1 = new MyThread();
                t1.start();
        }
}
```

Output: thread is running

By apply sleep() method

```
class MyThread extends Thread {
    public void run() {
        try {
            for (int i = 1; i < =5; i++) {
                System.out.println("thread is running...");
                Thread.sleep(1000); // Sleep for 1 second (1000 milliseconds)
            }
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted.");
        }
    }
}

class ThreadDemo {
    public static void main(String args[]) {
        MyThread t1 = new MyThread();
        t1.start();
    }
}
```

Output:

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

Creation of Multiple Threads

```
class A extends Thread {
	public void run() {
		try {
			for (int i = 1; i <= 5; i++) {
				System.out.println(i);
				Thread.sleep(5000);
			}
		} catch (InterruptedException e) {
			System.out.println("IE handled");
		}
	}
}

class B extends Thread {
	public void run() {
		try {
			for (int i = 101; i <= 105; i++) {
				System.out.println(i);
				Thread.sleep(10000);
			}
		} catch (InterruptedException e) {
			System.out.println("IE handled");
		}
	}
}

class C extends Thread {
	public void run() {
		try {
			for (int i = 201; i <= 205; i++) {
				System.out.println(i);
				Thread.sleep(15000);
			}
		} catch (InterruptedException e) {
			System.out.println("IE handled");
```

```
                    }
          }
}

public class Mutilple {

          public static void main(String[] args) {
                    A a1 = new A();
                    a1.start();
                    B b1 = new B();
                    b1.start();
                    C c1 = new C();
                    c1.start();

          }

}
```

```
Output: 1
           101
           201
           Numbers will be displayed randomly
```

2.By implementing Runnable interface

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


public static void main(String args[]){
MyThread m1=new MyThread();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

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

Can we start a thread twice?

```
public class TestThreadTwice1 extends Thread{
 public void run(){
   System.out.println("running...");
 }
 public static void main(String args[]){
  TestThreadTwice1 t1=new TestThreadTwice1();
```

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

Output:
running
Exception in thread "main" java.lang.IllegalThreadStateException

Thread Priorities

3 constants defined in Thread class:
public static int MIN_PRIORITY -->1
public static int NORM_PRIORITY--->default priority value-->5
public static int MAX_PRIORITY-->10

public final int getPriority(): it returns the priority of the given thread.
public final void setPriority(int newPriority): It sets the new priority to the thread .

```
class MyThread1 extends Thread {
	public void run() {
		System.out.println("Hello");
	}
}

class MyThread2 extends Thread {
	public void run() {
		System.out.println("Welcome");
	}
}

public class ThreadPriorityDemo {

	public static void main(String[] args) {
		MyThread1 a1 = new MyThread1();
		a1.start();
		MyThread2 b1 = new MyThread2();
		b1.start();
	System.out.println("Current Priorities of MyThread1 and MyThread2 Threads");

		System.out.println(a1.getPriority());
		System.out.println(b1.getPriority());

	System.out.println("Setting new priorities to MyThread1 and MyThread2 Threads");

		a1.setPriority(Thread.MAX_PRIORITY);
		b1.setPriority(Thread.MIN_PRIORITY);
System.out.println("after setting Current Priorities of MyThread1 and MyThread2 Threads");

                    System.out.println(a1.getPriority());
                    System.out.println(b1.getPriority());

          }

}
```

Output: Hello

```
Current Priorities of MyThread1 and MyThread2 Threads
5
5
Setting new priorities to MyThread1 and MyThread2 Threads
after setting Current Priorities of MyThread1 and MyThread2 Threads
10
1
Welcome
```

synchronizing threads

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

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

Thread Synchronization can be achieved in two ways.

1.Synchronized method.

2.Synchronized block.

1.Synchronized method.

If we declare any method as synchronized, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource.

synchronized method ensures that only one thread can execute it at a time for a given instance.

Ex:

```
class Table{

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

 } //close printTable
}  //close Table

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

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

public class SynchronizedDemo{
public static void main(String args[]){
Table t1 = new Table();//only one object
A a1=new A(t1);
B b1=new B(t1);
a1.start();
b1.start();
}
}
```

output:

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

2.Synchronized block.

Synchronized block can be used to perform synchronization on any specific resource of the method.Suppose we have 100 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized block.

syntax:

```
synchronized (object reference expression) {

  //code block

}
```

ex:

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

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

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

public class SynchronizedBlockDemo{
public static void main(String args[]){
Table t1 = new Table();//only one object
A a1=new A(t1);
B b1=new B(t1);
a1.start();
b1.start();
}
}
```

output:

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

Inter thread communication

It is a mechanism that allows synchronized threads to communicate with each other.

This is essential for creating complex multi-threaded applications where threads need to share data or synchronize their actions.

For implementing inter thread communication java recommends the following methods.

wait(): Causes the current thread to pause and release the lock until another thread calls notify() or notifyAll().
notify(): Wakes up one waiting thread that is paused on the same object's lock.
notifyAll(): Wakes up all waiting threads that are paused on the same object's lock.
Ex:

```
class Customer {
        int amount = 10000;

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

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

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

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

        }
}
```

Output:

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

ThreadGroups

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.

A ThreadGroup represents a set of threads. A thread group can also include the other thread group.

Constructors:
ThreadGroup(String name): creates a thread group with given name.

ThreadGroup(ThreadGroup parent, String name) :creates a thread group with a given parent group and name.

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

         Thread t1 = new Thread(tg1, runnable,"one");
         t1.start();
         Thread t2 = new Thread(tg1, runnable,"two");
         t2.start();
         Thread t3 = new Thread(tg1, runnable,"three");
         t3.start();

         System.out.println("Thread Group Name: "+tg1.getName());


   }
  }
```

Output:

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

Daemon Threads

A Daemon thread is created to support the user threads. It generallty works in background and terminated once all the other threads are closed.

ex: Garbage collector

Creating a Daemon Thread

To set a thread to be a daemon thread, all we need to do is to call Thread.setDaemon().

```
NewThread daemonThread = new NewThread();
daemonThread.setDaemon(true);
daemonThread.start();
```

ex:

```
class MyDaemonThread extends Thread {
        public void run() {
                try {
```

```
                for(int i=1;i<=5;i++) {
                    System.out.println("Daemon thread is running...");
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
                System.out.println("Daemon thread interrupted.");
            }
        }
}

public class DaemonThreadDemo {
    public static void main(String[] args) {
        MyDaemonThread daemonThread = new MyDaemonThread();
        daemonThread.setDaemon(true);
        daemonThread.start();

    }
}
```

Autoboxing

An automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing.
Boxing is also called as Autoboxing.

ex:
```
int a=50;
Integer a2=new Integer(a);//Boxing
Integer a3=5;//Boxing
```

Unboxing is the reverse process where the object wrapper is converted back into a primitive type.

ex:
```
Integer i=new Integer(50);
int a=i;
System.out.println(a);
```

```
import java.util.ArrayList;

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

    ArrayList<Integer> list = new ArrayList<>();

    list.add(5);
    list.add(6);
    System.out.println("ArrayList: " + list);

    int a = list.get(0);
    System.out.println("Value at index 0: " + a);
  }
}
```

Enum

Enum is a data type which contains a fixed set of constants.

According to the Java naming conventions, we should have all constants in capital letters.

The Java enum constants are static and final implicitly.

Defining an Enum

We define an enum using the enum keyword. Each constant in the enum is a static final instance of the enum type.

Ex:

```
public class EnumExample {
enum Day {
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY
}

   public static void main(String[] args) {
      Day day = Day.MONDAY;

      switch (day) {
         case SUNDAY:
            System.out.println("It's Sunday!");
            break;
         case MONDAY:
            System.out.println("It's Monday!");
            break;
         case TUESDAY:
            System.out.println("It's Tuesday!");
            break;
         case WEDNESDAY:
            System.out.println("It's Wednesday!");
            break;
         case THURSDAY:
            System.out.println("It's Thursday!");
            break;
         case FRIDAY:
            System.out.println("It's Friday!");
            break;
         case SATURDAY:
            System.out.println("It's Saturday!");
            break;
```

```
        }
    }
}
```

Annotations

Java Annotation is a tag that represents the metadata i.e. attached with class, interface, methods or fields to indicate some additional information.
Annotations in Java are used to provide additional information, so it is an alternative option for XML.

Built-In Java Annotations used in Java code
@Override
@Deprecated

1. @Override
@Override annotation assures that the subclass method is overriding the parent class method.

```
class A{
void doSomething(){System.out.println("printing");}
}

class Dog extends Animal{
@Override
void dosomething(){System.out.println("reading");}
}


class AnnotationDemo{
public static void main(String args[]){
A a1=new B();
a1.doSomething();
}
}
```

2.@Deprecated
@Deprecated annoation marks that this method is deprecated so compiler prints warning.
It informs user that it may be removed in the future versions.

```
public class DeprecatedExample {
    public static void main(String[] args) {
        DeprecatedExample example = new DeprecatedExample();
        example.oldMethod(); // This will generate a compiler warning
        example.newMethod();
    }

    @Deprecated
    public void oldMethod() {
        System.out.println("This method is deprecated and may be removed in future versions.");
    }

    public void newMethod() {
        System.out.println("This is the new method to use.");
    }
}
```

Generics

Generics provide a way to define classes, interfaces, and methods with type parameters. It allows for type safety and reduces the need for type casting, making the code more robust and easier to read.

Example of Generic Class

```
public class Box<T> {
    private T t;

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<>();
        Box<String> stringBox = new Box<>();

        integerBox.set(10);
        stringBox.set("Hello Generics");

        System.out.println("Integer value: " + integerBox.get());
        System.out.println("String value: " + stringBox.get());
    }
}
```

Explore Java.util package

The java.util package provides a vast set of utility classes and interfaces, which are essential for many basic operations such as data structures, date and time facilities, and other utility functionalities.

Here’s an overview of some of the key classes and interfaces in the java.util package:

1. Collections Framework
Interfaces
Collection: The root interface of the collections framework.
List: Ordered collection (also known as a sequence).
Set: Collection that cannot contain duplicate elements.
Map<K, V>: Object that maps keys to values, cannot contain duplicate keys.

Classes
ArrayList: Resizable array implementation of the List interface.
LinkedList: Doubly-linked list implementation of the List and Deque interfaces.
HashSet: Hash table-backed implementation of the Set interface.
HashMap<K, V>: Hash table-backed implementation of the Map interface.

2. Date and Time Utilities
Date: Represents a specific instant in time, with millisecond precision.
Calendar: Abstract class for converting between a Date object and a set of integer fields such as YEAR, MONTH, DAY, etc.
TimeZone: Represents a time zone offset and also figures out daylight savings.

Example Usages

Using ArrayList

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Hello");
        list.add("World");
        list.add("Welcome");

        for (String str : list) {
            System.out.println(str);
        }
    }
}
```

Using HashSet

```
import java.util.HashSet;

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

        HashSet<String> set = new HashSet<>();
        set.add("Apple");
        set.add("Banana");
        set.add("Cherry");
        set.add("Apple");
        set.add("Banana");
        System.out.println("HashSet: " + set);
        if (set.contains("Apple")) {
            System.out.println("HashSet contains Apple");
        }
        set.remove("Banana");
        System.out.println("HashSet after removal: " + set);
        System.out.println("Iterating over HashSet:");
        for (String fruit : set) {
            System.out.println(fruit);
        }
    }
```

Using HashMap

```
import java.util.HashMap;
import java.util.Map;

public class HashMapExample {
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "One");
        map.put(2, "Two");
        map.put(3, "Three");

        for (Map.Entry<Integer, String> entry : map.entrySet()) {
            System.out.println(entry.getKey() + " = " + entry.getValue());
        }
    }
}
```

Using Scanner

```
import java.util.Scanner;

public class ScannerExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();
        System.out.println("Hello, " + name + "!");
    }
}
```

Using Calendar

```
import java.util.Calendar;
public class CalendarExample {
    public static void main(String[] args) {
        Calendar calendar = Calendar.getInstance();
        System.out.println("Current Date and Time: " + calendar.getTime());

        calendar.add(Calendar.DATE, 1);
        System.out.println("Tomorrow's Date: " + calendar.getTime());
    }
}
```