Lambda Function in Java Programming

* **Objective:** To implement Functional Programming in Java Programming language.

* **Purpose:**

↳ The biggest advantage of using Lambda Function is concise coding (short coding)

↳ Lambda is a representation of Anonymous method (without name, modifier, return type)

* Introduced in Java 1.8 Version.

Rules of Writing Lambda Expression

```
( ) ⟶ { System.out.println("Welcome to Lambda "); };
```

Representation of Lambda
(without name, Modifier, Return Type)

* If there is only one statement, then No need to specify curly braces

```
( ) ⟶ sop.ln("Welcome to Java");
```

↳ Method with Parameters

In General

```
Public void add(int a, float b)
{
    System.out.println(a+b);
}
```

By Lambda

(int a, float b) ⟶ { s.o.p ln(a+b); };

More concise way

(a, b) ⟶ s.o.p ln(a+b);

↳ { No need to specify data types also, It is the responsibility of compiler to Recognise }

For example:

In General — Public int bonus (int sal)

{ return sal + 2000;

}

By Lambda

(sal) ⟶ { return sal + 2000; };

↳ More Concise Way by Lambda

sal ⟶ { return sal + 2000;

If we have only one argument to pass, then we have no need to specify Parenthesis.

↳ More concise way

sal ⟶ sal + 2000;

(Return value depends on the context)

Rule: If we are applying curly braces then we must mention return statement.

Phase-2: How to Implement Lambda expression in Java Programming language ?

→ By using Functional Interface.

Bonus: Marker Interface

e.g. @Annotations
@FunctionalInterface
interface i1
{
// Functional Interface is that interface which contain only 1 method (Abstract method must be 1).

Public void m1();

}

// However we can create any no of static method or private method inside it using Functional Interface.

// If we create 2 Abstract methods using Annotation @FunctionalInterface then Error will occur.

interface i1
{ Public void m1();
}
interface i2 extends i1
{
// overriding occurs

Marker Interface

The interface which is empty known as Marker Interface.

To extend the functionality of a class, we use Marker Interface

Eg:2

```
interface i1
{
// This is Marker interface (Cloneable)
}
```

⭐ Implementation of Marker Interface in Java

General method

```
interface i1
{ Public abstract void m1():
}
class c1 implements i1
{ @override
  public void m1()
  { s.o.p.ln("Hello");
    }
class Test
{ psvm(string args[])
  { c1 obj = new c1()
      obj.m1()
  }
```

⭐ Representation Using Lambda Expression and Anonymous Class

```
@FunctionalInterface
interface i1
{ public abstract void m1();
}

class Test
{ psvm(string args[])
  { i1 obj = () -> S.O.P("Hello");
    obj.m1()
  };

  obj.m1();
}
```

WRAPPER CLASSES

↳ For all the primitive data types available in Java, there is a corresponding object class representation available which is known as Wrapper classes.

Need for Wrapper classes

1) All Collection classes in Java can store only objects.
2) Primitive data types cannot be stored directly in these classes and hence the primitive values needs to be converted to objects.
3) We have to wrap the primitive data types in a corresponding object, and give them an object representation.

✱ Definition: The process of converting the primitive data types into objects is called Wrapping.

```
Integer iref = new Integer(i);
```

Here, class Integer is the wrapper class wrapping a primitive data type i.

✱ The Java API has provided a set of classes that make the process of wrapping easier. Such classes are called wrapper classes.

✱ For all primitive data types, there are corresponding wrapper classes. Storing primitive types in the form of objects affects the performance in terms of memory and speed.

✱ Representing an Integer via a wrapper class takes about 12-16 bytes, compared to 4 in an actual integer. Also, Retrieving the value of an integer uses the method Integer.intValue().

✱ The wrapper classes are very useful as they enable you to manipulate primitive data types.

For ex: You can take the integer input from the user in the form of a string and convert it into integer type using following statements

```
String str = "100";
int j = Integer.parseInt(str);
```

⊙ The wrapper classes also have constants like :

MAX_VALUE, MIN_VALUE, NaN (Not a Number)

POSITIVE_INFINITY, NEGATIVE_INFINITY

THE INTEGER CLASS

⊙ class Integer is a wrapper for values of type int.

→ Integer objects can be constructed with a int value, or a string containing a int value

↳ The constructors for Integer are shown here :

- ⊙ Integer(int num)
- ⊙ Integer(String str) throws NumberFormatException

↳ Some methods of Integer class :

- ⊙ static int parseInt(String str) throws NumberFormatExcep
- ⊙ int intValue() returns the value of the invoking object as a int value.

⊛ Few more methods of

ByteValue()

DoubleValue()

FloatValue()

LongValue()

ShortValue()

e.g.: Integer i1 = new Integer(20);
double d1 = i1.doubleValue();

import java.util.Scanner;

class DemoBinaryString

{ psvm(String[] args)

{ Scanner x = new Scanner (System.in);

System.out.println("Enter any int b/w 1 and 255");

int i = x.nextInt();

String s = Integer.toBinaryString(i);

System.out.println("The unsigned integer in base 2 of "+i+" is :"+s);

}

}

Autoboxing and Unboxing

Java 5.0 version introduced automatic conversion b/w a primitive type and corresponding wrapper class.

↳ During assignment, the automatic transformation of primitive type to corresponding wrapper type is known as ~~Unboxing~~. Autoboxing.

Note: Primitive types ⟶ wrapper type (Autoboxing)

e.g. Integer i1 = 10;

↳ During assignment, the automatic transformation of wra
type into their primitive equivalent is known as Unboxing.

Note: wrapper Type ⟶ Primitive Type (Unboxing

E.g.→ int i=0;
i = new Integer(10);

Note: Boxing conversion converts values of primitive type to corresponding values of reference type. But the primiti types can't be widened/narrowed to wrapper classes and vice versa.

Note: All the wrapper classes belongs to java.lang package.

EXCEPTIONAL HANDLING MECHANISM IN JAVA

If we use Exceptional Handling mechanism, then our program will be gracefully terminated. (If a exception occurs in any part of the code).

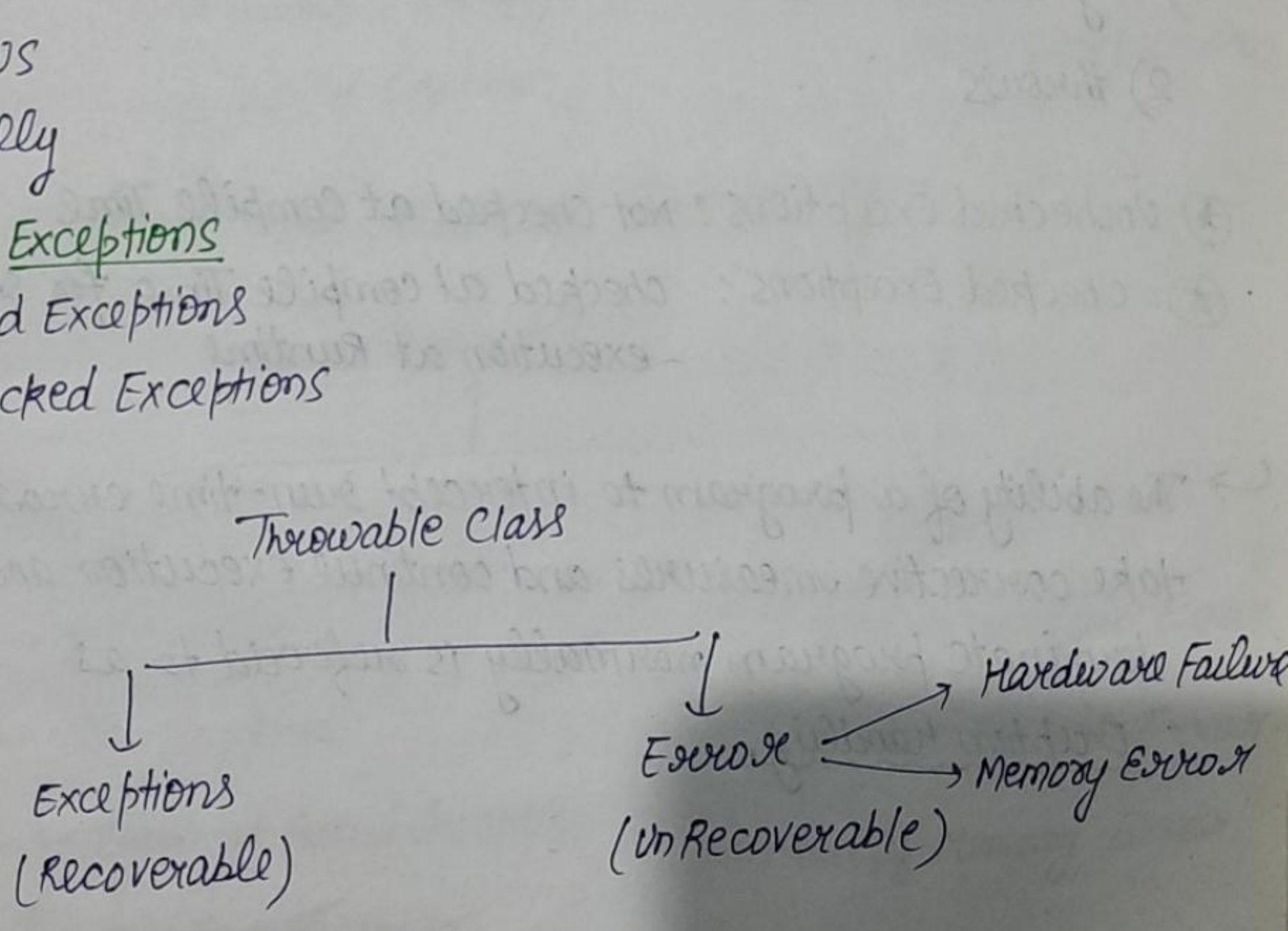
If we don't use Exceptional Handling mechanism, then our program will abnormally terminated.

#Keywords Used in Exceptional Handling Mechanism

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

Types of Exceptions

- Checked Exceptions
- Unchecked Exceptions



Ex: class Test

```
{ public static void main (string [] args)
  { int arr = new int [Integer.MAX-VALUE]
    }
}
```

★ Hardware Failure Exception which is not Recoverable.

Note: All the exceptions which are subclasses of Runtime exceptions are known as Checked Exceptions.

Handling Exceptions

1) try catch throw

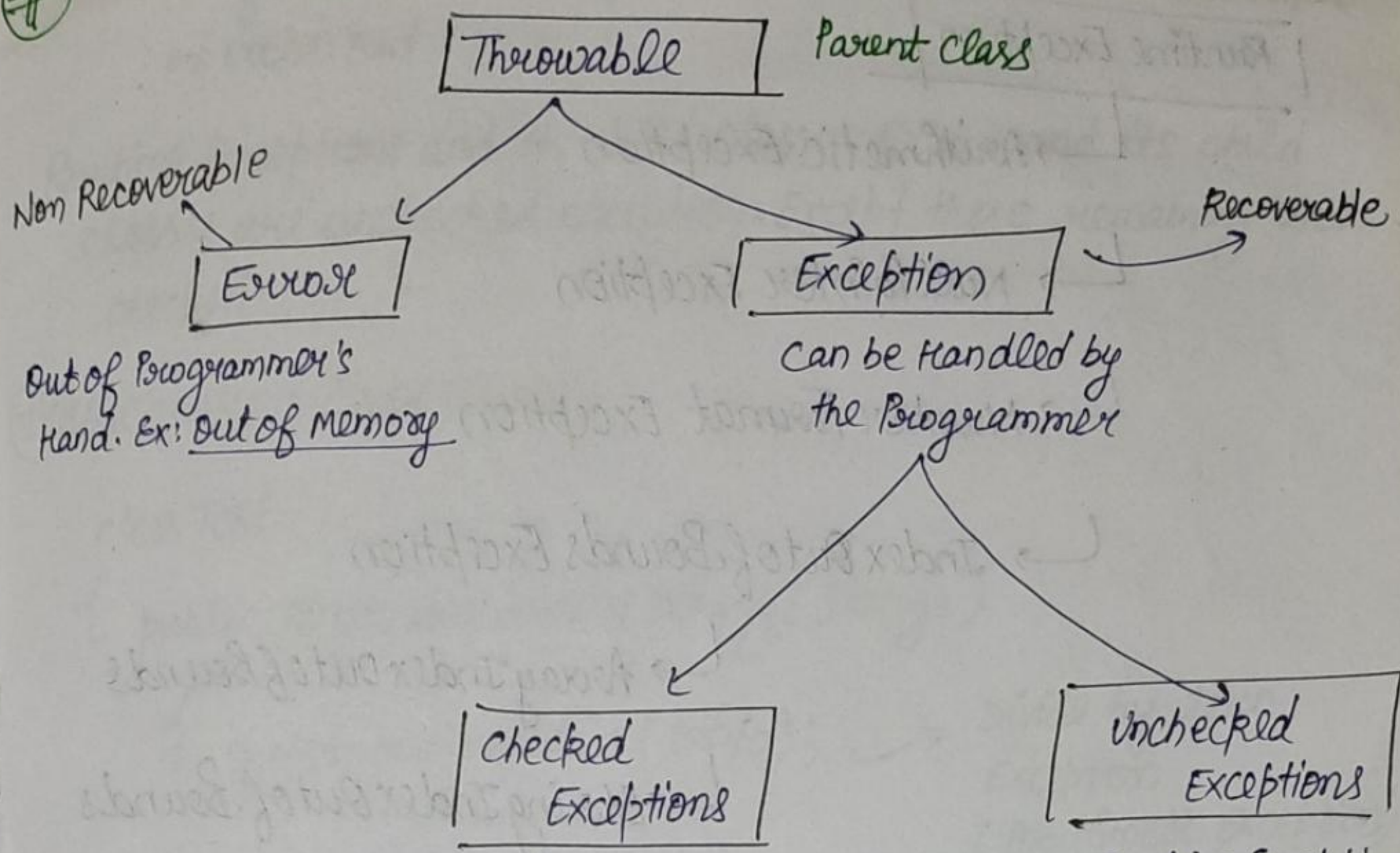
2) throws

★ Unchecked Exceptions : Not checked at Compile Time

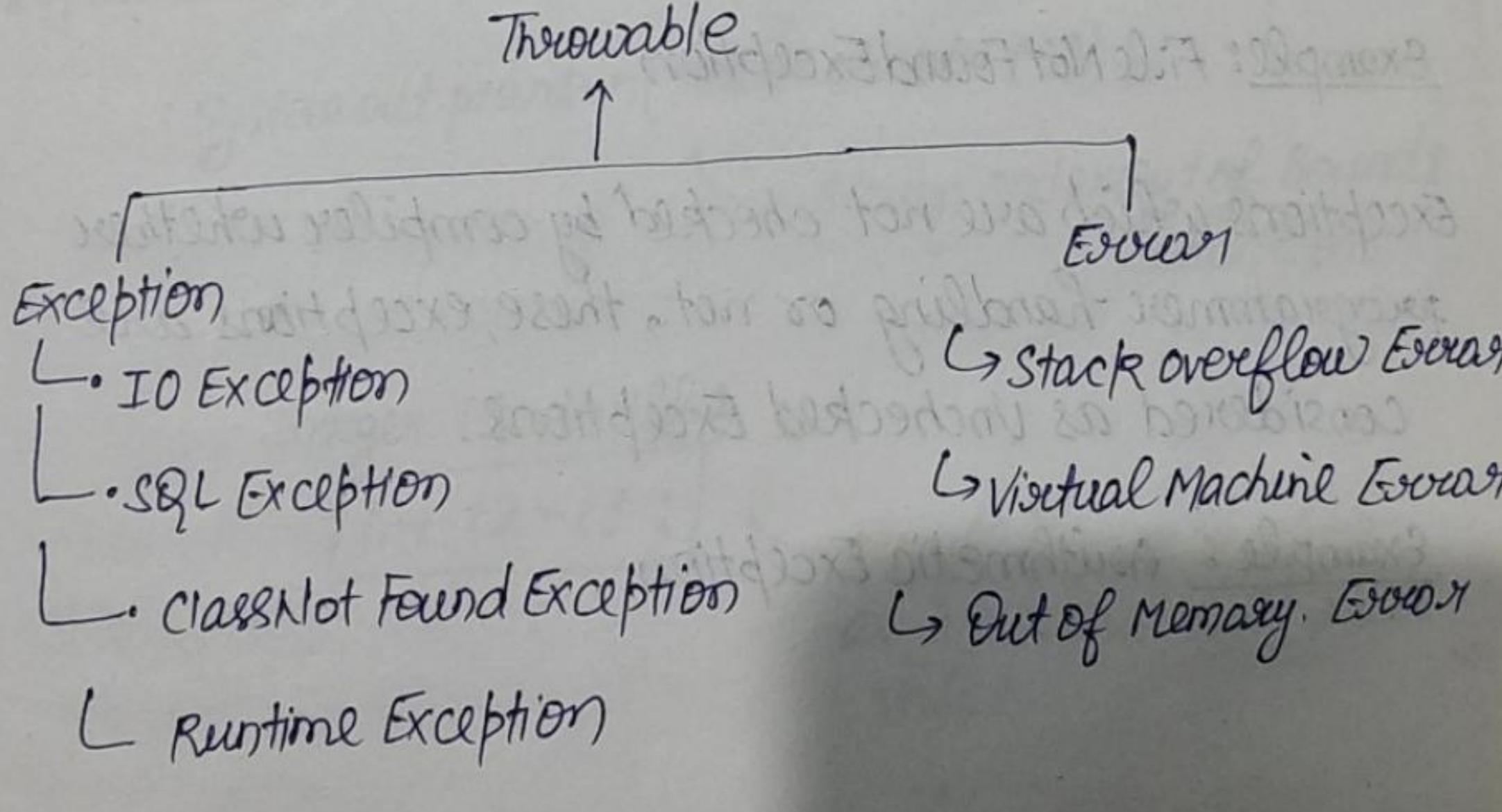
★ checked Exceptions : checked at compile Time for smooth execution at Runtime

↳ The ability of a program to intercept run-time errors, take corrective measures and continue execution and terminate program normally is refered to as Exception Handling.

Exception in Nutshell



JAVA EXCEPTION HIERARCHY



Continued....

- Runtime Exception
  - Arithmetic Exception
  - NullPointer Exception
  - Number Format Exception
  - Index Out of Bounds Exception
    - Array Index Out of Bounds
    - String Index Out of Bounds

Checked and Unchecked Exceptions

Exceptions which are checked by compiler for smooth execution of program at runtime are considered as checked Exceptions.

Example: FileNotFoundException

Exceptions which are not checked by compiler whether programmer handling or not, these exceptions are considered as Unchecked Exceptions.

Example: Arithmetic Exception

Note:- Every exception occurs at Runtime whether it is checked or unchecked

Runtime Exceptions and its child classes, Error and its child classes are unchecked exception. Except these remaining are checked.

#UNCHECKED EXCEPTIONS

```
class Test
{ public static void main(String[] args)
  { System.out.println(100/0); ⟶ Divide by Zero Exception
                                 (Arithmetic Exception)
                                 ↓
                                 Predefined Class

    System.out.println(args[5]);
                    .
                    ↙
IndexOutOfBoundException

    System.out.println("GLA".charAt(5));
                              ↘ String IndexOut of Bounds
                                Exception

    Integer i1 = null;
    int i2 = i1;  ⟶ Null Pointer Exception as int cannot hold null value.
                    ↓
                    Primitive data type.
```

Integer i3 = new Integer ("ABC"); exception Raised

↳ Number Format Exception

Integer i4 = new Integer (" ");

↳ Empty (No Exception)

(##) class Test

{ public static void main (string[] args)

{ int a, b;

a = 10;

try throw new Exception ("/ by Zero")

{ system.out.println (a/0)

}

catch (Arithmetic Exception Obj)

{

System.out.println(e);

}

Note:

try

{

}

catch

{

}

try

{

}

finally

{

}

→ we cannot use catch Handlers after finally.

In this case, error will occur.

Note: One try block can have more than one catch handlers.

In Java 1.7 Version, Single catch handlers can handle multiple exceptions.

Ex:
```
catch (ArithmeticException | IndexOutOfBoundsException e)
    // Three methods
```

Throwable class { toString()
printStackTrace()
getMessage() ⟶ Only description

Note: Nested try blocks are possible

↳ First inner catch handle then outer catch handle.

↳ try without catch is not possible.

↳ In nested try, only outer catch can also do the job

Note: way of catch statements ⟶ First Specific Then Generic

Imp. ## Difference b/w final, finally and finalized method

1) One is used with Exceptional Handling and other one is used for Automatic Garbage Collection.

2) finalized method is used for object clean up activity.

3) finally block is used for try block open resource clean up activity.

3) System.exit(1)

If we forcefully stop our program, by explicitly mentioning above statement then finally block will not execute in this case.

Imp. We can write Return Statement in catch block but it will dominate finally block.

Note:

Interview Question

* Can you close the connection without using finally block?

Two types of Interfaces

- Autocloseable
- closeable

Prototype declaration of method

Public void m1() throws Checked-Exception name

** Note:

1) when we use throws only Checked Exceptions are there.

2) We only use throws statement after method & constructor.

3) If you are going to override the method that contain throws statement, then in child class while overriding we have to write same exception or write child exception name.

* Root class of exception class is throwable.

# class Test
{
public static void main (String[] args)
{ throw new Test ("Hello");
}
}

throw statement can execute only by throwable class.

class Test extends Exception → because Parent class of exception is throwable.
{}

class Test extends Runtime Exception

Runtime Exception
↓
unchecked Exception

CUSTOM EXCEPTION

How to create custom Exception ?

```
class customException extends Exception
{  customException(String obj)
   {  super(obj);
   }
}
```

Example:

```
class TestCustomException
{  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 (InvalidAgeException e)
{ System.out.println("Exception occured "+ m);
  }
    System.out.println("Rest of code")
  }
}
```

★★★★★

✱ try with Resources

→ Introduced in Java 1.7 Version

```
class Test
{ public static void main(String [] args)
  {
    try
{ Scanner sc = new Scanner(System.in);
      int a = sc.nextInt();
    }
      catch (Exception ob)
      { System.out.println(ob);
      }
      finally
      { System.out.println("finally block");
        sc.close();
```

```
class Test
{ public static void main (String [] args)
    { try (Scanner SC = new Scanner (System.in))
        { int a = sc.nextInt ()
        }
    catch (Exception obj)
    { System.out.println (obj)
      }
```

Note:

Only those resources can be specified here, who implement autocloseable interface. and Autocloseable Interface have one method

↓

close() method

and Scanner class which we have implemented in previous example implemented the Autocloseable Interface.

Imp. Point

* If we are using try with resource then we can't use finally block. There is no need of using finally block when we use try with resource which implement Autocloseable Interface

(★) In Java 1.9 version

If we have opened few resources and we want to Autoclose those Resources then we can do it.

From Java 1.9 version, we can implement this thing.

Ex:

```
class Test
{ public static void main (String [] args)
  {Scanner sc = new Scanner(System.in);
                                  ★ Only this Resource can autoclose
   try(sc)
   { int a = sc.nextInt();
   }
```

But if we reinitialize the resource in try block, then it will not autoclose the reference.

Eg:

```
try(sc)
{
  sc = new Scanner(System.in)        ★ This Resource will not autoclose.
     int a = sc.nextInt();
}
```

MULTITHREADING

Multithreading in Java is a process of executing multiple threads simultaneously.

⊙ A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and Multiprogramming & Multithreading are used to achieve multitasking.

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

⊙ Java Multithreading is mostly used in games, animation etc.

✱ Advantages of Java Multithreading

1) It doesn't block the user because threads are independent and you can perform multiple operations at same time.

2) You can perform many operations together, so it saves a lot of time.

3) Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilise the CPU. Multitasking can be achieved in two ways:

- Process Based Multitasking ⟶ Multiprocessing
- Thread Based Multitasking ⟶ Multithreading

What is Multithreading?

- In multithreading, the thread is the smallest unit of code that can be dispatched by the thread scheduler.
- A single program can perform two tasks using two threads.
- Only one thread will be executing at any given point of time given a single-processor architecture.

Uses of Multithreading

- A multithreaded application performs two or more activities concurrently.
- It is accomplished by having each activity performed by a seperate thread.
- Threads are the lightest tasks within a program, and they share memory space and resources with each other.

Q How to create Multithreaded Applications ?

1st method

class Test implements Runnable Interface

2nd method

class multithreadingDemo extends Thread (class)

Thread creation by implementing Runnable Interface

```
class Test implements Runnable
{ @override
    public void run()
    { try
        { System.out.println("Thread " + Thread.currentThread()
                                .getId() + " is running");

        }catch (Exception e)
        {
            System.out.println ("Exception is caught");
        }
    }
}
```

```
class multithread
{  psvm(string[] args)
   {  int a=8;
      for(int i=0;i<a;i++){
↳        Thread obj = new Thread(new Test()); ④
         obj.start();
      }
   }
}
```

(#) Thread creation by extending Thread class

```
class Demo extends Thread{
   public void run()
   {
      //
   }
public class Test
{  psvm(String[] args)
   {  Demo obj = new Demo();
      obj.start();
}}}
```

Thread Class v/s Runnable Interface **

1) If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple Inheritance But if we implement the Runnable interface, our class can still extend other base classes.

2) We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable Interface.

3) Using Runnable will give an object that can be shared amongst multiple threads.

JAVA THREAD METHODS

Method	Modifier and Type	Description
start()	void	It is used to start the execution of Thread.
run()	void	It is used to do an action for a thread
sleep()	static void	It sleeps a thread for specified amt of time
join()	void	It waits for a thread to die.

getPriority()	int	It returns the priority of thread.
setPriority()	void	It changes the priority of thread.
getId()	long	It returns the id of the thread.
yield()	static void	It causes the currently executing thread object to pause and allow other threads to execute temporarily.
stop()	void	It is used to stop the thread.
notify()	void	It is used to give the notification for only one thread which is waiting for a particular object.
notifyall()	void	It is used to give the notification to all waiting threads of a particular object.

Points

* To prevent thread execution we use, sleep(), yield() and join method.

  For inter thread communication we use wait(), notify() and notifyAll() method.

* All interthread communication methods are defined inside the object class.

* All Interthread communication methods are used inside the synchronized blocks or synchronized methods.

* whenever you call, wait() or notify() method, they will immediately release the block.

Que what is the difference b/w Synchronization block and Synchronization method? what is the advantage of using synchronized block over synchronized methods?

Soln

```
class Account
{
    Public int balance;
    Public account()
    {
        balance = 10000;
    }

Public void withDraw(int val) throws InterruptedException
synchronization (x)
{
    Thread.sleep(1000);
    balance = balance - val
```

Note:
* Because it will degrade the performance of whole application.

```
        System.out.println(balance);
    }
class mythread extends Thread
{   Account obj;
    MyThread(Account tobj)
    {  obj = tobj;
    }
    public void run()
    {  obj.withDraw(500)
    }
}
class Test
{  public static void main(String[] args)
    {   Account obj = new Account();
        MyThread T1 = new MyThread(obj);
        MyThread T2 = new MyThread(obj);
        T1.start();
        T2.start();
    }
}
```

Note: yield, join and sleep methods are used for prevention of execution of Thread.

Synchronization of previous code

(*) withDraw method (making Synchronized block)

```
public void withDraw(int val)
{
  synchronized(this)  ↪ (*) we can also specify the name of particular obj.
  {
    Thread.sleep(1000);
    balance = balance - val;
    System.out.println(balance);
  }
  // - - - -
  // - - - -
  // - - - -
}
```

Note: If we use synchronized(Account class) then all the objects of Account class will be blocked.

(*) → Performance of system will be increased when we will utilise synchronization blocks over synchronized method.

(*) → Every Thread which is created from main class will have same priority.

wait() & notify() { wait() & sleep() / wait() & yield() }

```
class MyThread extends Thread
{ public void run()
   { system.out.println(main.sum);
   }

class Main
{ myThread obj = new MyThread();
    obj.start();

  public static int sum;
   for(int i=1; i<20; i++)
   { sum=sum+i;
   }
   notify();
  }
```

JDBC

⊛ JAVA DATABASE CONNECTIVITY ⊛

⊛ Steps to develop JDBC Application

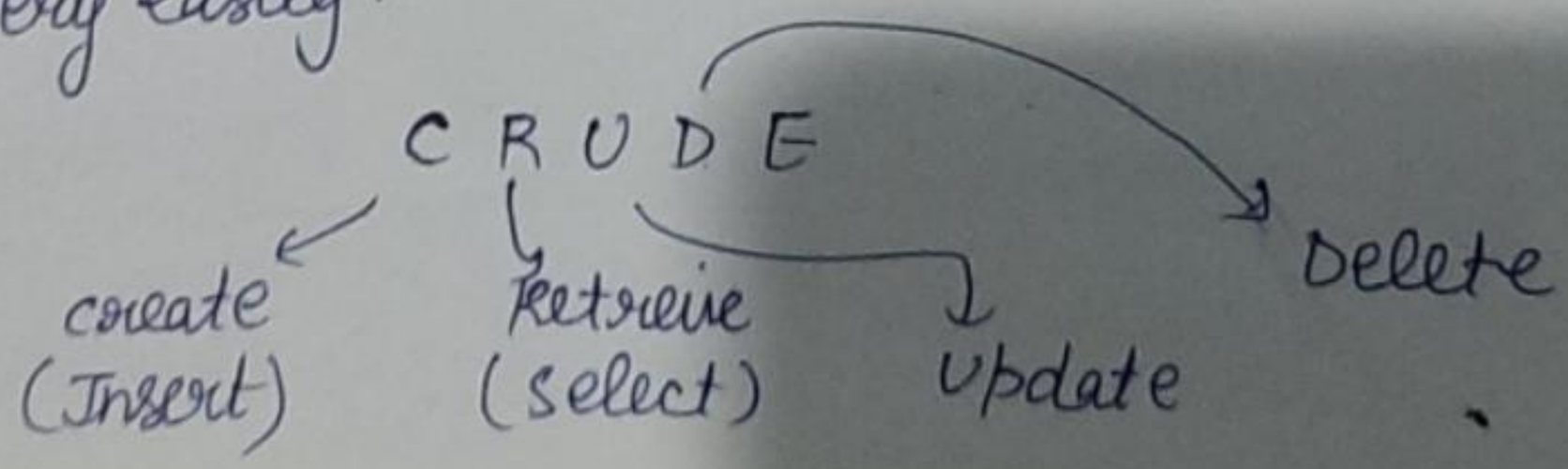
1) Load and Register Driver.
2) Establish connection b/w Java Application and Database
3) Creation of Statement Object.
4) Send & Execute SQL Query.
5) Process results from Resultset.
6) close the connection.

JDBC Features

1) JDBC is a standard API. we can communicate with any database without rewriting our application i.e., it is database independent API.

2) Most of the JDBC drivers are developed in JAVA and hence, JDBC concept can work for any platform i.e., it is a platform independent technology.

3) By using JDBC API, we can perform basic crude operations very easily.



4) There is a big vendor support for JDBC. They developed multiple products on JDBC API.

5) JDBC version is different while JAVA version is different. The JDBC version which we are using is 4.3.

Note: Last update in JDBC API was in Java 9 version, After that No updation is done in JDBC API.

Note: | Driver → oracle.jdbc.oracledriver |

public class yy{

public static void main(String[] args)

{ class.forName("com.mysql.cj.jdbc.Driver")

Connection con = DriverManager.getConnection(url: "", user: "root", password: "");

Statement s = con.createStatement();

ResultSet rs = s.executeQuery(sql: "select * from stu");

while(rs.next())

{

System.out.println(rs.getInt(columnIndex: "1");

OR

Sout(rs.getInt(columnlabel: "rollno" + " " + rs.getString(columnlabel: "name");

}

con.close(); → (*) Because this method is inherited from Autocloseable Interface.

}

}

Note:
<protocol>://<host>:<port>/<database>

JDBC API

we utilise java.sql package. → (*) All basic functions and features are present in this package.

javax.sql → Advanced features are present in this package.
(x → extension)

↳ By using the GUI components, we can make our application very interactive.

((*) → But not in syllabus, study on your own)

(*) GUI components have two packages:

- awt package (Heavy weighted)
- swing package (Java language itself; Light weighted)

major diff. is that awt package contains → button
swing package contains → jbutton

↳ Statement object

```
Statement s = con.createstatement();
Resultset rs = s.executeQuery("select * from stu");
```

1) Result set ⟶ executeQuery

2) execute() ⟶ Boolean (Return type)

3) int executeUpdate

④ s.executeUpdate (insert, update, delete)

Ex: Table name ⟶ student

rollno, name, marks

↳ insert into student values (1, "aa", 78);

Note: Statement objects are not SQL injections safe.
while Prepared Statement objects are safe of SQL injection.

Note: Database engine processes statement obj. again and again every time.

Prepared Statements

prepared Statement obj

```
Statement s = con.createStatement();
Prepared Statement ps = con.createStatement("Query
                        which you want to execute);
```

insert into student values (?, ?, ?);

↳ Represent attribute

while(true)

{

* get some information from Scanner class object.

ps.setValue (1, new Scanner(System.in.nextInt());

ps.executeQuery();

}

int res;

sout("Do you want to continue? Enter yes or No");

res = ps.next();

if (res.equalsIgnoreCase("No")

{

break;

}

Note: Method can be of three types → setValue()
setInt()
setString()

REGULAR EXPRESSIONS

Basically, it is an expression which is used to represent string or group of string according to a specific pattern.

↳ Regular Expressions (Regex) are validated by validation frameworks

Ex:

```
String str = "GLAU";
str.matches ("GLAU");
```

Note: two classes which are used to implement regex in java.

↳ Pattern

↳ Matcher

These classes are present in java.util.regex.*;

```
class Test
{ public static void main (String [] args)
    { Pattern p = Pattern.compile ("Naman");
```

↳ the pattern you want to match will be specified here.

```
String str = "Naman nam nama abc xyz java ajay";
Matcher m = p.matcher(s);
```

↳ Target string

Note: Regular Expressions (regex) are case sensitive.

```
int count =0;
while(m.find())
{ count++;
  System.out.println(m.start()+"  " + m.end());
  }
   System.out.println(count);
  }
}
```

Quantifiers

we use quantifiers to specify no of occurences to match.
There are three types of Quantifiers:

- + (at least one)
- * multiplication
  (Any no of value including zero no also)
- ? (at most one) → either one or zero

GENERICS AND COLLECTION FRAMEWORKS IN JAVA

↳ Collection Frameworks deals only with objects.

↳ To provide 'type safety' and to resolve 'type casting' problems, we utilize the concept of Generics

Representation of Generics

Test<>

↳ Various types of concepts can be passed in Angular braces.

Note: Arrays are by default type safe (*).