SNS COLLEGE OF ENGINEERING

Kurumbapalayam (Po), Coimbatore – 641 107

An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

COURSE NAME : 23CSB101- OBJECT ORIENTED PROGRAMMING

I YEAR /II SEMESTER

Unit III – EXCEPTION HANDLING AND MULTITHREADING

Topic : EXCEPTION HANDLING

EXCEPTION HANDLING

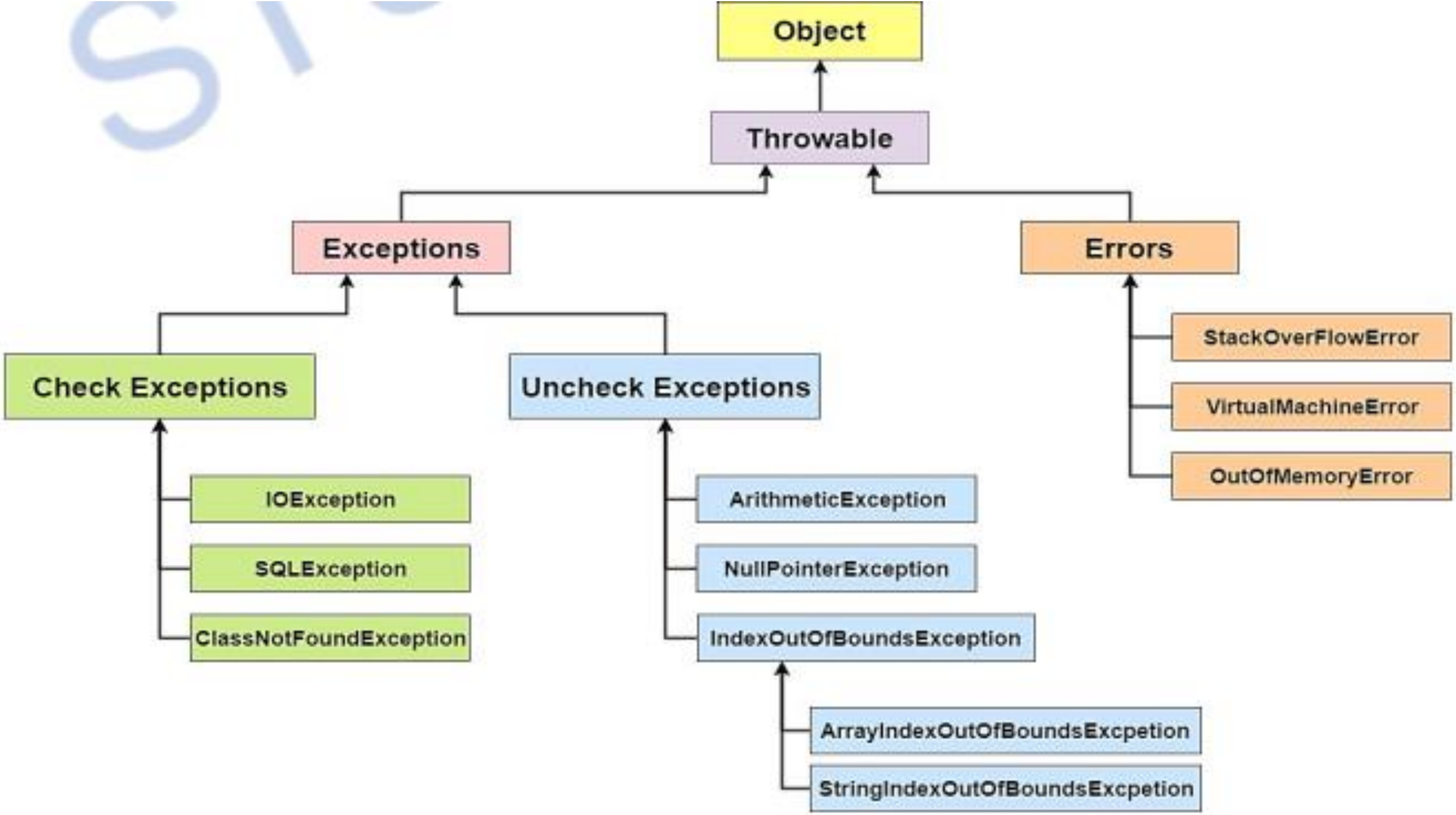
•An **Exception** is an event that occurs during program execution which disrupts the normal flow of a program. It is an object which is thrown at runtime

•**Exception handling** is a mechanism that allows you to handle runtime errors and exceptional conditions in a graceful manner, preventing the program from crashing. This is done using a combination of try, catch, throw, throws, and finally blocks.

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

•The Throwable class is further divided into two classes:

1. Exceptions and
2. Errors.


Object
Throwable
Exceptions
Errors
Check Exceptions
Uncheck Exceptions
StackOverFlowError
VirtualMachineError
OutOfMemoryError
IOException
SQLException
ClassNotFoundException
ArithmeticException
NullPointerException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsExcpetion
StringIndexOutOfBoundsExcpetion

USER DEFINED EXCEPTIONS

•**Built-in exceptions** are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations.

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

USER DEFINED EXCEPTIONS

To create User-defined Exceptions:

1. Pick a self-describing ***Exception** class name.

2. Decide if the exception should be checked or unchecked.

    Checked : extends Exception

    Unchecked: extends RuntimeException

3. Define constructor(s) that call into super class constructor(s), taking message

that can be displayed when the exception is raised.

4. Write the code that might generate the defined exception inside the try-catch block

5. If the exception of user-defined type is generated, handle it using throw clause.

USER DEFINED EXCEPTIONS

To create User-defined Exceptions:

1. Pick a self-describing ***Exception** class name.

2. Decide if the exception should be checked or unchecked.

    Checked : extends Exception

    Unchecked: extends RuntimeException

3. Define constructor(s) that call into super class constructor(s), taking message

that can be displayed when the exception is raised.

4. Write the code that might generate the defined exception inside the try-catch block

5. If the exception of user-defined type is generated, handle it using throw clause.

throw ExceptionClassObject;

USER DEFINED EXCEPTIONS

To create User-defined Exceptions:

1. Pick a self-describing ***Exception** class name.

2. Decide if the exception should be checked or unchecked.

    Checked : extends Exception

    Unchecked: extends RuntimeException

3. Define constructor(s) that call into super class constructor(s), taking message

that can be displayed when the exception is raised.

4. Write the code that might generate the defined exception inside the try-catch block

5. If the exception of user-defined type is generated, handle it using throw clause.

throw ExceptionClassObject;

USER DEFINED EXCEPTIONS

```
public class EvenNoException extends Exception
{
 EvenNoException(String str)
 {
 super(str); // used to refer the superclass constructor
 }
 public static void main(String[] args)
 {
 int arr[]={2,3,4,5};
 int rem;
 int i;
 for(i=0;i<arr.length;i++)
 {
 rem=arr[i]%2;
```

Cont…

USER DEFINED EXCEPTIONS

```
try
 {
    if(rem==0)
   {
     System.out.println(arr[i]+" is an Even Number");
    }
    else
   {
     EvenNoException exp=new EvenNoException(arr[i]+" is not an Even Number");
     throw exp;
   }
```

Cont…

USER DEFINED EXCEPTIONS

```
    }
        catch(EvenNoException exp)
    {
      System.out.println("Exception thrown is "+exp);
     }
 } // for loop
 } // main()
} // class
```

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

Comparison – final-finally-finalize

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

Comparison – final-finally-finalize

Basis for comparison	final	finally	Finalize

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

Avoid finalize()?

✓ Unreliable: No guarantee when (or if) it runs.

✓ Deprecated: Not recommended in Java 9+.

✓ Better Alternatives: Use try-with-resources (AutoCloseable interface) for proper cleanup.

- catch block is not mandatory if you use a finally block or try-with-resources.
- Recommended: Always use catch or throws to handle exceptions properly.

Example: try-with-resources

```
import java.io.*;

class ResourceExample implements AutoCloseable
 {
   public void show()
   {
      System.out.println("Using resource...");
   }

   @Override
   public void close()
   {
      System.out.println("Resource closed.");
   }
}
```

```
public class TryWithResourcesExample
{
   public static void main(String[] args)
{
   try
     (
        ResourceExample res = new ResourceExample()
     )
     {
         res.show();
      } // `close()` is automatically called here
   }
}
```

Example: try-with-resources

Scenario	Is `catch` Required?	Notes
`try` alone	❌ Not allowed	Compilation error
`try` + `catch`	✅ Required	Standard exception handling
`try` + `finally`	❌ Not required	But exception remains unhandled
`try-with-resources`	❌ Not required	Resources auto-close, but exceptions must be handled separately

Lab Program

Implement exception handling and creation of user defined exceptions.

```
import java.io.*;
import java.util.*;
class MyException extends Exception
{
 private int d;
 MyException (int a)
 {
      d = a;
 }
 public String toString()
 {
 return "MyException [" + d + "]";
 }
}
```

Cont…

Lab Program

```
class UserException
{
   static void compute(int a) throws MyException
   {
        System.out.println ("Called Compute(" + a + ")");
        if(a>10)
          Throw new MyException(a); System.out.println ("Normal Exit");
        public static void main(String args[])
       {
        try
            {
             compute(1);
             compute(20);
            }
       catch(MyException e)
         {
            System.out.println("Caught " + e); }
    }
}
```

```
Called Compute(1)
Normal Exit
Called Compute(20)
Caught MyException [20]
```


Thank
you