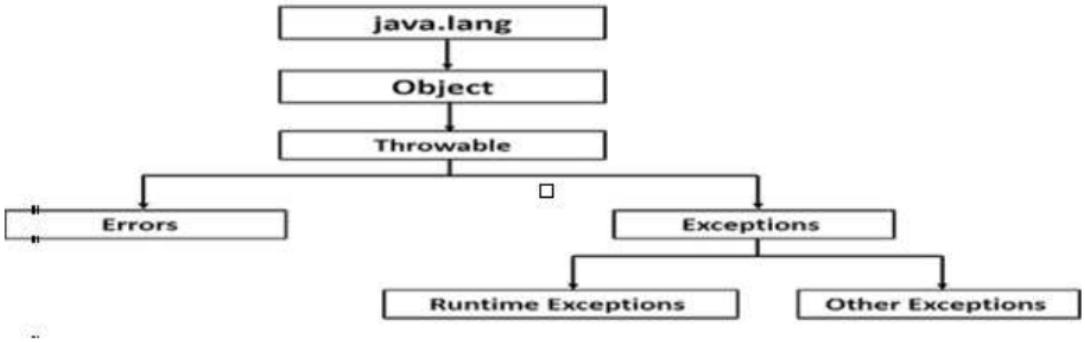
Errors indicate serious problems and abnormal conditions that most applications should not try to handle. Error defines problems that are not expected to be caught under normal circumstances by our program. For example memory error, hardware error, JVM error etc.

Exceptions are conditions within the code. A developer can handle such conditions and take

Exception Hierarchy

- All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class.



Categories of Exceptions

1. **Checked exceptions** – A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.

2. **Unchecked exceptions** – An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime

   exceptions are ignored at the time of compilation.

Common scenarios where exceptions may occur:

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

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

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

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an

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

3) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:
int a[]=new int[5];

a[10]=50; //ArrayIndexOutOfBoundsException

Checked Exception	Unchecked Exception
1, checked exceptions occur at compile time	1, Unchecked Exceptions occurs at run time
2, The compiler checks a checked exception	2, The compiler doesn't check these types of exceptions
3, These exceptions can be handled at time of compilation	3, These types of exceptions can't be handle at time of compilation
4, They are sub-class of the exception class	4, They are runtime exceptions & hence aren't a part of Exception class
Examples (or) Types of Exceptions :- 1, Class Not Found Exception 2, IO Exception 3, SqL Exception 4, File Not Found Exception	Examples (or) Types of Unchecked Exceptions :- 1, Arithmetic Exception 2, Nullpointer Exception 3, Number format Exception 4, Array Index Out of Exception

Array index exception:

```
package p1;
import java.util.Scanner;
public class ExceptionHandling {
public static void  main(String args[])
{
Scanner sc=new Scanner(System.in);
try
{
      int b[]=new int[5];
      System.out.println(b[6]);
      }
catch(ArrayIndexOutOfBoundsException e) {
            System.out.println(e);
      }
        System.out.println("rest of the code...");


       }
      }
```

Null Pointer Exception:

```
package p1;

import java.util.Scanner;

public class NulpointerException
{
```

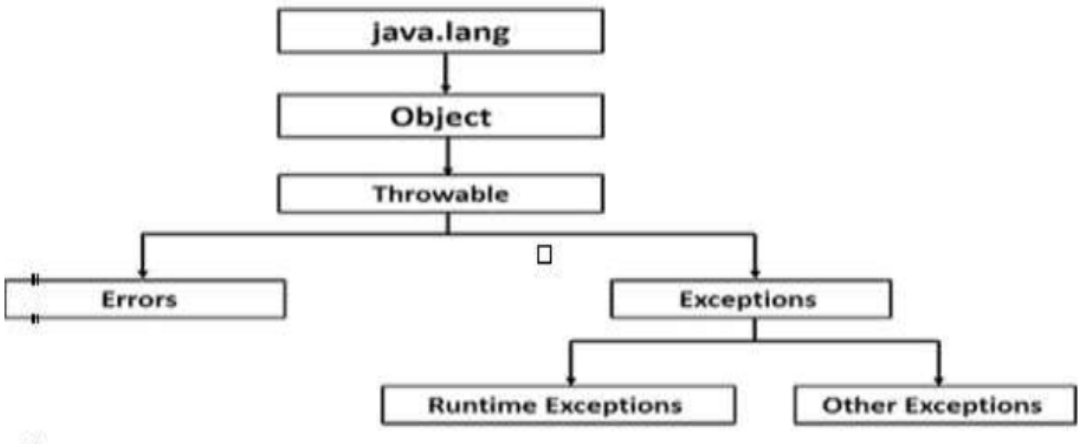
```
    public static void  main(String args[])
{
      Scanner sc=new Scanner(System.in);
    try {
            String s=null;
        System.out.println(s.length());
    }
    catch(NullPointerException e) {
        System.out.println(e);
    }
      System.out.println("rest of the code...");


     }
    }
```

NumberFormatException:

```
package p1;
public  class  Numberformatexception
 {
  public static void  main(String args[])
{
try
{
String s="CSE24STUDENTSARENOTPERFECT";
// NumberFormatException
int  i=Integer.parseInt(s);
      }
      catch(NumberFormatException   e)
      {
            System.out.println(e);
      }
        System.out.println("rest of the code...");
       }
}
```

Exception Hierarchy

- All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class.



Key words used in Exception handling

There are 5 keywords used in java exception handling.

1. try	A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code.
2. catch	A catch statement involves declaring the type of exception we are trying to catch.
3. finally	A finally block of code always executes, irrespective of occurrence of an Exception.
4. throw	It is used to execute important code such as closing connection, stream etc. throw is used to invoke an exception explicitly.
5. throws	throws is used to postpone the handling of a checked exception.

Using try and Catch

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch. A try and its catch statement form a unit.

```
class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
```

```
} catch (ArithmeticException e) { // catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

The call to println( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block.

Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.

The following example traps two different exception types:

```
// Demonstrate multiple catch statements.
class MultiCatch {
public static void main(String args[]) {
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
```

```
}
}
```

Here is the output generated by running it both ways:

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

Nested try Statements

The try statement can be nested. That is, a try statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.

If no catch statement matches, then the Java run-time system will handle the exception.

```
// An example of nested try statements.
class NestTry {
public static void main(String args[]) {
try {
int a = args.length;
/* If no command-line args are present,
the following statement will generate
a divide-by-zero exception. */
int b = 42 / a;
System.out.println("a = " + a);
try { // nested try block
/* If one command-line arg is used,
then a divide-by-zero exception
will be generated by the following code. */
```

```
if(a==1) a = a/(a-a); // division by zero
/* If two command-line args are used,
then generate an out-of-bounds exception. */
if(a==2) {
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);
}
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
```

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException:42
```

throw

it is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:

throw ThrowableInstance;

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions. There are two ways you can obtain a Throwable object: using a parameter in a catch clause, or creating one with the new operator.

,

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace

```
// Demonstrate throw.
class ThrowDemo {
static void demoproc() {
try {
throw new NullPointerException("demo");
} catch(NullPointerException e) {
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
System.out.println("Recaught: " + e);
}
}
}
```

Here is the resulting output:
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo

Throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a

method
can throw must be declared in the throws clause This is the general form of a method declaration that includes a throws clause:
type method-name(parameter-list) throws exception-list
{
// body of method
}
Here, exception-list is a comma-separated list of the exceptions that a method can throw

```
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
}
}
}
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

finally

The finally keyword is designed to address this
contingency.
finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method

returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The finally clause is optional.

```
// Demonstrate finally.
class FinallyDemo {
// Through an exception out of the method.
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
} finally {
System.out.println("procA's finally");
}
}
// Return from within a try block.
static void procB() {
try {
System.out.println("inside procB");
return;
} finally {
System.out.println("procB's finally");
}
}
// Execute a try block normally.
static void procC() {
try {
System.out.println("inside procC");
} finally {
System.out.println("procC's finally");
}
}
public static void main(String args[]) {
try {
procA();
} catch (Exception e) {
System.out.println("Exception caught");
```

```
}
procB();
procC();
}
}
```

Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

User-defined Exceptions

- All exceptions must be a child of **Throwable.**
- If we want to write a checked exception that is automatically enforced by the Handle ,we need to extend the Exception class.
- **User defined exception needs to inherit (extends) Exception class in order to act as an exception.**
- throw keyword is used to throw such exceptions.

In-lab Task:

1. Election committee wants to check whether the voter is eligible to vote or not. The person can vote if his age is greater than 18. Help the Election committee by developing a code which arises exception if the voter age is less than 18 then print the exception and "VOTER IS NOT ELIGIBLE TO VOTE", otherwise print "VOTER IS ELIGIBLE TO VOTE". (Hint: Develop user-defined Exception)

```
package pack1;
import java.util.Scanner;

class MyException extends Exception
```

```
{
    int age;

    MyException(int a)
    {
        age =  a;
    }

    public String toString( )
    {
        return "VOTER IS NOT ELIGIBLE TO
VOTE :" + age;
    }
}

public class ElectionDemo {

    public static void main(String[] args) {

        System.out.println("Enter the age");
        Scanner sobj = new Scanner(System.in);

        int age = sobj.nextInt( );

        try
```

```
        {
            if(age<18)
                throw new MyException(age);
            else
                System.out.println("VOTER IS
ELIGIBLE TO VOTE");
        }
        catch(MyExceptioneobj)
        {
            System.out.println(eobj);
        }

    }

}
```

2. Enhance the Student class of previous problem, such that when Invalid ID (-ve number) is given it must throw InvalidIDException, and InvalidNameException when name has special characters or digits

```
package pack1;

classInvalidIDException extends Exception
{
    private long id;
    InvalidIDException(long i)
```

```
    {
        id = i;
    }
    public String toString( )
    {
        return "Invalid ID :"+id;
    }
}

classInvalidNameException extends Exception
{
    private String name;
    InvalidNameException(String n)
    {
        name = n;
    }
    public String toString( )
    {
        return "Invalid Name :"+name;
    }
}
```

```
class Student
{
	private long id;
	private String name;

	public void setId(long i)
	{
		try
		{
			if(i > 0)
				id =  i;
			else
				throw new InvalidIDException(id);
		}
		catch(InvalidIDExceptioneobj)
		{
			System.out.println(eobj);
		}
	}

	public void setName(String n)
	{
```

```
        boolean mat = n.matches("[a-zA-Z]+");
        try
        {
            if(mat)
                name =  n;
            else
                throw new
InvalidNameException(name);
        }
        catch(InvalidNameExceptioneobj)
        {
            System.out.println(eobj);
        }
    }

    public String getName( ) { return name; }
    public long getId( ) { return id; }

    public String toString( )
    {
        return "\nName :"+getName( )+"\nId
:"+getId( );
```

```
    }
}

public class StudentDemo {

    public static void main(String[] args)
    {

            Student sobj = new Student( );
            sobj.setName("SaiKumar");
            sobj.setId(1000);
            System.out.println(sobj);

    }

}
```

Post-lab Task:

1. Create a Class Engine with attributes engineID (int), engineType (String), horsepower(int), engineWeight (float). add constructors, getter, setters and toString () for Engine class. Enhance the setters in such a way that if invalid data is sent the setter will generate an appropriate exception. Create 2 objects in main () and access the methods using these objects. Display the details.

```
package pack1;

classInvalidEngineIDException extends Exception
{
    privateinteid;
    InvalidEngineIDException(int e)
    {
        eid = e;
    }
    public String toString( )
    {
        return "Invalid Engine ID :"+eid;
    }
}
```

```
classInvalidEngineTypeException extends
Exception
{
    private String eType;
    InvalidEngineTypeException(String e)
    {
        eType = e;
    }
    public String toString( )
    {
        return "Invalid Engine Type :"+eType;
    }
}

classInvalidHorsePowerException extends
Exception
{
    privateinthPower;
    InvalidHorsePowerException(int e)
    {
        hPower = e;
    }
    public String toString( )
    {
```

```
        return "Invalid Horse Power :"+hPower;
    }
}


classInvalidEngineWeightException extends
Exception
{
    private float eWeight;
    InvalidEngineWeightException(float e)
    {
        eWeight = e;
    }
    public String toString( )
    {
        return "Invalid Engine Weight :"+eWeight;
    }
}


class Engine
{
    privateintengineID;
    private String engineType;
    privateinthorsePower;
```

```
    private float engineWeight;

    public void setEngineId(int e)
    {
        Integer obj = e;
        String str = obj.toString( );
        intlen = str.length();
        try
        {
            if(len == 5)
                engineID = e;
            else
                throw new
InvalidEngineIDException(e);
        }
        catch(InvalidEngineIDExceptioneobj)
        {
            System.out.println(eobj);
        }
    }

    public void setEngineType(String e)
    {
        boolean mat = e.matches("[a-zA-Z]+");
        intlen = e.length();
```

```
            try
            {
                if(len == 8 && mat)
                    engineType = e;
                else
                    throw new
InvalidEngineTypeException(e);
            }
            catch(InvalidEngineTypeExceptioneobj)
            {
                System.out.println(eobj);
            }
        }

        public void setHorsePower(int e)
        {
            try
            {
                if(e <= 2000)
                    horsePower = e;
                else
                    throw new
InvalidHorsePowerException(e);
            }
            catch(InvalidHorsePowerExceptioneobj)
```

```
            {
                System.out.println(eobj);
            }
        }

        public void setEngineWeight(float e)
        {
            try
            {
                if(e <= 500)
                    engineWeight = e;
                else
                    throw new
InvalidEngineWeightException(e);
            }
            catch(InvalidEngineWeightExceptioneobj)
            {
                System.out.println(eobj);
            }
        }

        publicintgetEngineId( ) { return engineID; }
        public String getEngineType( ) { return
engineType; }
```

```
    publicintgetHorsePower( ) { return horsePower;
}
    public float getEngineWeight( ) { return
engineWeight; }

    public String toString( )
    {
        String str;
        str = String.format("Engine Id
:%d%nEngine Type :%s%n"
                + "Horse Power :%d%nEngine
Weight :%f%n",
                getEngineId( ),getEngineType(
),getHorsePower( ),
                getEngineWeight( ));
        returnstr;
    }



}

public class EngineDemo {

    public static void main(String[] args) {
        Engine eobj = new Engine( );
```

```
        eobj.setEngineId(51456);
        eobj.setEngineType("normalTy");
        eobj.setHorsePower(1800);
        eobj.setEngineWeight(450);
        System.out.println(eobj);
    }

}
```

2. Create a class Book with BName, BId, BAuthor, and YOP (Year of Publication). Use proper getter and setter methods. BName must not have any special symbols except '-', BId must not have any whitespace and special symbols, BAuthor must not have any special characters and digits, and YOP should contain only a 4-digit number. Use toString () to format the details of the book. And enhance the setter methods to throw an exception is the data passed to setter is not a valid data. Create 2 objects in main () and access the methods using these objects. Display the details

```
package pack1;


classInvalidBNameException extends Exception
{
    private String BName;
    InvalidBNameException(String b)
    {
```

```
        BName = b;
    }
    public String toString( ) { return "Invalid Book
Name :"+BName; }
}

classInvalidBIdException extends Exception
{
    private String BId;
    InvalidBIdException(String b)
    {
        BId = b;
    }
    public String toString( ) { return "Invalid Book
Id :"+BId; }
}

classInvalidBAuthorException extends Exception
{
    private String BAuthor;
    InvalidBAuthorException(String b)
    {
```

```
        BAuthor = b;
    }
    public String toString( ) { return "Invalid Book
Author :"+BAuthor; }
}

classInvalidYOPException extends Exception
{
    privateint YOP;
    InvalidYOPException(int b)
    {
        YOP = b;
    }
    public String toString( ) { return "Invalid Year
of Publication :"+YOP; }
}

class Book
{
    private String BName, BId, BAuthor;
    privateint YOP;
```

```
    public void setBName(String b)
    {
        boolean match = b.matches("^[a-zA-Z\\s-
]*");
        try
        {
            if(match)
                BName = b;
            else
                throw new
InvalidBNameException(b);
        }
        catch(InvalidBNameExceptioneobj)
        {
            System.out.println(eobj);
        }
    }

    public void setBId(String b)
    {
        boolean match = b.matches("^[a-zA-Z0-
9]*");
```

```
        try
        {
            if(match)
                BId = b;
            else
                throw new InvalidBIdException(b);
        }
        catch(InvalidBIdExceptioneobj)
        {
            System.out.println(eobj);
        }
    }

    public void setBAuthor(String b)
    {
        boolean match = b.matches("^[a-zA-
Z\\s]*");
        try
        {
            if(match)
                BAuthor = b;
            else
```

```
                    throw new
InvalidBAuthorException(b);
            }
            catch(InvalidBAuthorExceptioneobj)
            {
                System.out.println(eobj);
            }
        }

        public void setYOP(int b)
        {
            String str = Integer.toString(b);
            intlen = str.length( );
            try
            {
                if(len == 4 && b > 0)
                    YOP = b;
                else
                    throw new
InvalidYOPException(b);
            }
            catch(InvalidYOPExceptioneobj)
```

```
        {
            System.out.println(eobj);
        }
    }

    public String getBName( ) { return BName; }
    public String getBId( ) { return BId; }
    public String getBAuthor( ) { return BAuthor; }
    publicintgetYOP( ) { return YOP; }

    public String toString( )
    {
        String str;
        str = String.format("Book Name
:%s%nBook Id :%s%nBook Author :%s%n"
                + "Book Year of Publication
:%d%n", getBName( ),getBId( ),getBAuthor( ),
                getYOP( ));
        returnstr;
    }

}
```

```
public class BookDemo {

	public static void main(String[] args) {
		Book book1 = new Book( );
		book1.setBName("Java Progamming-
Software Approach");
		book1.setBId("B1234");
		book1.setBAuthor("NarendraBabu");
		book1.setYOP(2009);
		System.out.println(book1);

		Book book2= new Book( );
		book2.setBName("Mastering in Java - A
Constructive Approach");
		book2.setBId("B1891");
		book2.setBAuthor("YaminiAparnaDevika");
		book2.setYOP(2019);
		System.out.println(book2);

	}
```

}