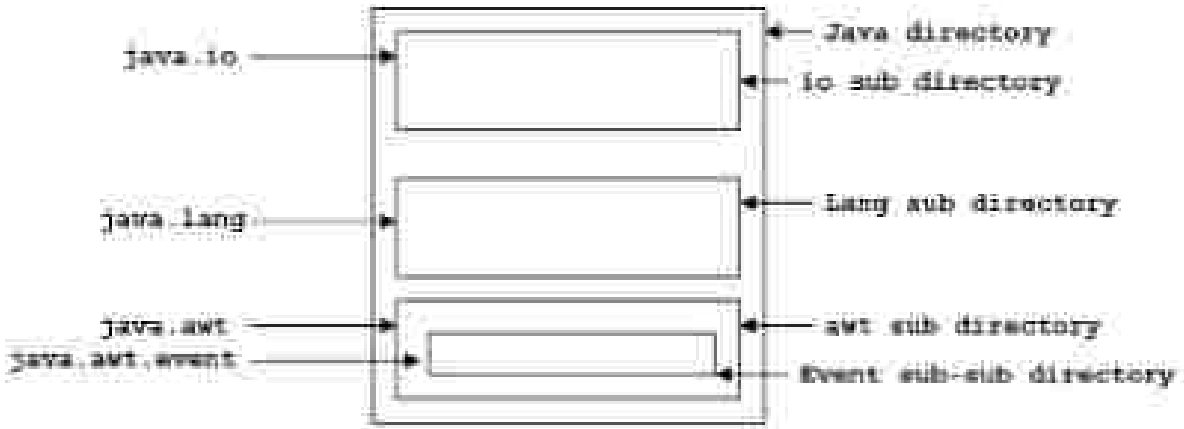
UNIT IV

Packages and Java Library: Introduction, Defining Package, Importing Packages and Classes into Programs, Path and Class Path, Access Control, Packages in Java SE, Java.lang Package and its Classes, Class Object, Enumeration, class Math, Wrapper Classes, Auto-boxing and Auto-unboxing, Java util Classes and Interfaces, Formatter Class, Random Class, Time Package, Class Instant (java.time.Instant), Formatting for Date-Time in Java, Temporal Adjusters Class, Temporal Adjusters Class.

Exception Handling: Introduction, Hierarchy of Standard Exception Classes, Keywords throws and throw, try, catch, and finally Blocks, Multiple Catch Clauses, Class Throwable, Unchecked Exceptions, Checked Exceptions, try-with-resources, Catching Subclass Exception, Custom Exceptions, Nested try and catch Blocks, Rethrowing Exception, Throws Clause.

Packages and Java Library: Introduction

- It is necessary in software development to create several classes and interfaces.
- After creating these classes and interfaces, it is better if they are divided into some groups depending on their relationship.
- So these classes and interfaces are stored in some directory.
- This directory or folder is also known as package.



Advantages of packages:

- packages hide the classes and interfaces in a separate sub directory, so accidental deletion of classes and interfaces will not take place.
- Two classes in two different packages can have the same name.
- A group of packages is called a library. The reusability nature of packages makes programming easy.

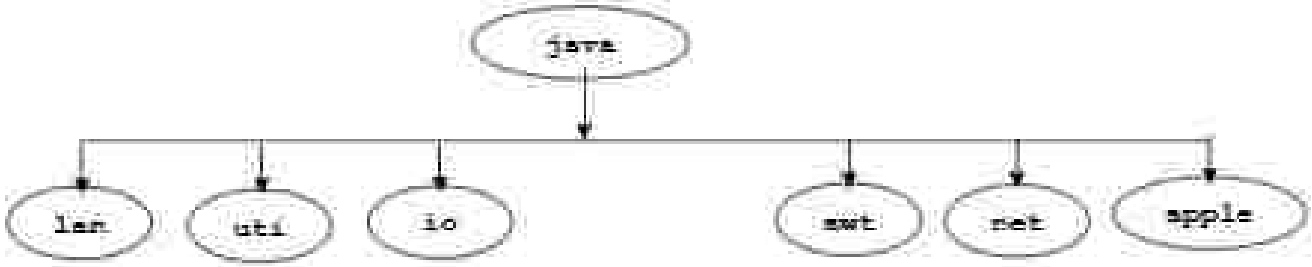
There are two types of packages:

- Built-in packages/Java API packages.
- User Defined Packages.

Java API Packages:

- Java API provides a large number of classes grouped into different packages according functionality.

Packages from java API are



Package name	Contents
java.lang	lang package contains language support classes. There classes are imported by java compiler automatically for its usage. lang package contains classes for primitive types, string, threads, exceptions, etc……
java.util	Utility classes such as vectors, hash tables, etc……
java.io	Contains classes for input/output of data.
java.awt	Contains for graphical user interface, classes for windows, buttons, lists, menus, etc……
java.net	Classes for networking, contains classes for communicating local computers and internet servers.
Java.applet	Classes for creating and implementing applets

Defining Package

- A package is created using keyword package
  - Syntax:
    package <package>;
- Package statement must be the first statement in a java source file.
  - Ex:
    package studentspack; // package declaration
    class Student // class declaration
    {

    }

Create a package which contains Addition class in that.

```
package pack;

public class Addition

{

 private double a,b;
```

```
public Addition(double a,double b)
{
 this.a=a;
 this.b=b;
}
public void sum()
{
 System.out.println("Sum="+(a+b));
}
}
```

Compiling the program:

> javac –d . Addition.java

- -d tells java compiler to create a separate sub directory and place the .class file there.
- Dot (.) indicates that the package should be created in the current directory.

Program to use Addition class from mypack:

```
class UsePack
{
public static void main(String args[])
{
 pack.Addition obj=new pack.Addition(10,20.5);
 obj.sum();
}
}
```

Note:

- Instead of referring the package name every time, we can import the package like this
  - import pack.Addition;
- Then the program can be written as....

```
import pack.Addition;
class UsePack
{
public static void main(String args[])
{
 Addition obj=new Addition(10,20.5);
 obj.sum();
}
}
```

Importing Packages and Classes into Programs

There are two ways of accessing classes stored in a package.

1. Fully qualified class name
2. Using Import Statement

1. Fully qualified class name;

```
double y=java.lang.Math.sqrt(x);
```

2. Using Import Statement;

syntax:-

```
import packagename.classname;
```

or

```
import packagename.*;
```

```
import java.awt.Font;
```

- imports Font class in our program.

(or)

```
import java.awt.*;
```

- imports all classes of awt package in our program.

Path and Class Path

PATH	CLASSPATH
PATH is an environment variable	CLASSPATH is an environment variable that tells the java compiler where to look for class files to import. Generally CLASSPATH is set to a directory or JAR file.
It is used by the operating system to find the executable files (.exe)	It is used by ApplicationClassLoader to locate the .class file
You are required to include the directory which contains .exe files	You are required to include all the directories which contain .class and JAR files.
PATH environment variable once set, cannot be overridden.	The CLASSPATH environment variable can be overridden by using the command line option -cp or -CLASSPATH to both javac and java command

How to Set CLASSPATH in Windows Using Command Prompt

CLASSPATH is an environment variable that tells the java compiler where to look for class files to import. Generally CLASSPATH is set to a directory or JAR file.

Type the following command in your Command Prompt and press enter.

```
set CLASSPATH=%CLASSPATH%;C:\Program Files\Java\jre1.8\rt.jar;
```

In the above command, the set is an internal DOS command that allows the user to change the variable value. CLASSPATH is a variable name. The variable enclosed in percentage sign (%) is an existing environment variable. The semicolon is a separator, and after the (;) there is the PATH of rt.jar file.

Access Control:

Access modifier → / Access Location ↓	Private	Public	Default/ Nomodifier/ Friendly	protected	Private Protected
Same package Same class	Yes	Yes	Yes	Yes	Yes
same package subclass	No	Yes	yes	Yes	Yes
Same package Non-subclass	No	Yes	Yes	Yes	No
Different package Subclass	No	Yes	No	Yes	Yes
Different package Non-subclass	No	Yes	No	No	no

- This table only applies to members of classes
- A class has only two access levels
  - Public - accessible any where
  - Default - accessible with in the same package.

Packages in Java SE:

Java Standard Edition provides 14 packages namely –

- **applet** – This package provides classes and methods to create and communicate with the applets.
- **awt**– This package provides classes and methods to create user interfaces.
- **io**– This package contains classes and methods to read and write data standard input and output devices, streams and files.
- **lang**– This package contains the fundamental classes, methods, and, interfaces of Java language.
- **math**– This package contains classes and methods which helps you to perform arithmetic operations using the Java language.
- **net**– This package provides classes to implement networking applications.
- **rmi**– This package provides classes, methods, and interfaces for Remote Method Invocation.
- **security**– This package provides classes and interfaces for security framework.
- **sql**– This package provides classes and methods to access and process the data stored in a data source.
- **text**– This package provides classes and interfaces to handle text.
- **time**– This package provides API for dates, times, instants, and durations.
- **util**– This package contains collection framework, collection classes, classes related to date and time, event model, internationalization, and miscellaneous utility classes.

Java. lang Package and its Classes:

The most important classes are of lang are

- Object, which is the root of the class hierarchy, and Class, instances of which represent classes at runtime.
    - protected Object clone()
    - boolean equals(Object obj)
    - protected void finalize()
    - Class getClass()
    - int hashCode()
    - void notify()
    - void notifyAll()
    - void wait()
    - String toString()
- The wrapper classes
    - Boolean
    - Character
    - Integer
    - Short
    - Byte
    - Long
    - Float
    - Double
- The classes String, StringBuffer, and StringBuilder similarly provide commonly used operations on character strings.
- Class Throwable encompasses objects that may be thrown by the throw statement. Subclasses of Throwable represent errors and exceptions.

Enumeration in java:

```
class EnumExample
{
        //defining enum within class
        public enum Season { WINTER, SPRING, SUMMER, FALL }
        public static void main(String[] args)
        {
                //printing all enums
                for (Season s : Season.values())
                {
                        System.out.println(s);
                }
                System.out.println("Value of WINTER is: "+Season.valueOf("WINTER"));
                System.out.println("Index of WINTER is: "+Season.valueOf("WINTER").ordinal());
                System.out.println("Index of SUMMER is: "+Season.valueOf("SUMMER").ordinal());

}}
```

class Math:

```
class MathClass
{
	public static void main(String args[])
	{
		System.out.println("Absolute value-"+Math.abs(-90));
		System.out.println("Minimum value-"+Math.min(90,20));
		System.out.println("Maximum value-"+Math.max(90,20));
		System.out.println("round value-"+Math.round(79.52));
		System.out.println("sqrtroot value-"+Math.sqrt(25));
		System.out.println("cuberoot value-"+Math.cbrt(125));
		System.out.println("power value-"+Math.pow(2,3));
		System.out.println("ceil value-"+Math.ceil(2.2));
		System.out.println("ceil value-"+Math.floor(2.8));
		System.out.println("floorDiv value-"+Math.floorDiv(25,3));
		System.out.println("random value-"+Math.random());
		System.out.println("rint value-"+Math.rint(81.65));
		System.out.println("subtractExact value-"+Math.subtractExact(732, 190));
		System.out.println("multiplyExact value-"+Math.multiplyExact(732, 190));
		System.out.println("incrementExact value-"+Math.incrementExact(732));
		System.out.println("decrementExact value-"+Math.decrementExact(722));
		System.out.println("negateExact value-"+Math.negateExact(90));
	}
}
```

Wrapper Classes:

Primitive data types can be converted into object types by using the wrapper classes contained in java.lang package.

Below table shows the simple data types and their corresponding wrapper classes:

Simple type	Wrapper class
int	Integer
char	Character
float	Float
double	Double
long	Long
boolean	Boolean

The wrapper classes have a number of unique methods for handling primitive data types and objects.

Converting primitive type to object type using Constructor methods:

Constructor	Conversion Action
Integer x=new Integer(i)	Primitive integer to Integer object
Float x=new Float(f)	Primitive float to Float object
Double x=new Double(d)	Primitibe double to Double object
Long x=new Long(l)	Primitive long to Long object

Converting object numbers to Primitive numbers using typeValue() method:

Method	Conversion Action
int i=x.intValue()	Object to primitive integer
float f=x.floatValue()	Object to primitive float
long l=x.longValue()	Object to primitive long
double d=x.doubleValue()	Object to primitive double

Converting numbers to string using toString() method:

Method	Conversion Action
str=Integer.toString(i)	primitive integer to String
str=Float.toString(f)	Primitive float to String
str=Double.toString(d)	Primitive double to String
str=Long.toString(l)	Primitive long to String

Converting string Objects to Numeric objects using static method valueOf():

Method	Conversion Action
X=Double.valueOf(str)	Converts string to Double object
X=Float.valueOf(str)	Converts string to Float object
X=Integer.valueOf(str)	Converts string to Integer Object
X=Long.valueOf(str)	Converts string to Long object

Converting Numeric string to primitive type using parsing methods:

Method	Conversion Action

int i=parseInt(Str)	Converts string to integer
long l=parseLong(str)	Converts string to long

Ex:-

A simple integer can be changed to object type:

int x=100;

Integer ob1=new Integer(x);

Now ob1 is an integer object which contains the 100 as its value.

Object can be converted into primitive type:

int x=ob1.intValue();

now x will contain the value 100.

Byte class:

The following are the Byte class constructures

Byte(byte N);

Byte(String str);

Ex:

Byte b1=new Byte(100);

Byte b2=new Byte("Hello");

Note:

Byte class constructor accepts number as well as string as its parameters.

The following are the methods of Byte class.

byte byteValue() – returns byte value of the object.

Int compareTo(Byte b) – it accepts byte object and compares with invoking object.

static byte parseByte(String str) throws NumberFormatException

- it accepts string object and converts into byte. It is static so it can be called directly using the class name.

static Byte valueOf(String str)throws NumberFormatException

- it accepts a String object and converts into Byte class object.

Integer class:

The following are the Integer class constructures

Integer(int N);

Integer(String str) throws NumberFormatException

The following are the methods of Integer class:

int intValue() – To get integer from Integer object.

int compareTo() -

Auto-boxing and Auto-unboxing:

The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing. This is the new feature of Java5. So java programmer doesn't need to write the conversion code.

Advantage of Autoboxing and Unboxing:

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

AutoBoxing Example:

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

UnBoxing Example

```
class UnboxingExample1
{
```

```
    public static void main(String args[])
    {
      Integer i=new Integer(30);
      int a=i;
      System.out.println(a);
    }
}
```

Java util Classes and Interfaces

Classes of util package:

- ArrayDeque
- **ArrayList**
- **Arrays**
- BitSet
- Calendar
- **Collections**
- Currency
- Date
- Dictionary
- EnumMap
- EnumSet
- **Formatter**
- GregorianCalendar
- HashMap
- HashSet
- Hashtable
- IdentityHashMap
- LinkedHashMap
- LinkedHashSet
- LinkedList
- ListResourceBundle
- Locale
- Observable
- PriorityQueue
- Properties
- PropertyPermission
- PropertyResourceBundle
- **Random**
- ResourceBundle
- ResourceBundle.Control
- Scanner
- ServiceLoader
- SimpleTimeZone

- Stack
- StringTokenizer
- Timer
- TimerTask
- TimeZone
- TreeMap
- TreeSet
- UUID
- Vector
- WeakHashMap

Interfaces of util package

- Collection<E>
- Comparator<T>
- Deque<E>
- Enumeration<E>
- EventListener
- Formattable
- Iterator<E>
- List<E>
- ListIterator<E>
- Map<K,V>
- Map.Entry<K,V>
- NavigableMap<K,V>
- NavigableSet<E>
- Observer
- Queue<E>
- RandomAccess()
- Set<E>
- SortedMap<K,V>
- SortedSet<E>

Formatter Class:
The java.util.Formatter class provides support for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output.

Once the Formatter object is created, it may be used in many ways. The format specifier specifies the way the data is formatted.

A few common format specifiers are:

- **%S or %s:** Specifies String
- **%X or %x:** Specifies hexadecimal integer
- **%o:** Specifies Octal integer
- **%d:** Specifies Decimal integer
- **%c:** Specifies character

- **%T or %t:** Specifies Time and date
- **%n:** Inserts newline character
- **%B or %b:** Specifies Boolean
- **%A or %a:** Specifies floating point hexadecimal
- **%f:** Specifies Decimal floating point

Example:

```
Formatter f=new Formatter();
f.format("%3$3s %2$3s %1$3s", "fear","strengthen", "weakness");
System.out.println(f);
```

Random Class:

```
//all the functions of Random class will generate Pseudo random numbers
import java.util.Random;
public class Test
{
   public static void main(String[] args)
   {
      Random random = new Random();
      System.out.println(random.nextInt(10));
      System.out.println(random.nextBoolean());
      System.out.println(random.nextDouble());
      System.out.println(random.nextFloat());
      System.out.println(random.nextGaussian());
   }
}
```

Time Package

Class Instant (java. time. Instant)

Formatting for Date/Time in Java:

```
//Formatting for Date/Time in Java
import java.text.SimpleDateFormat;
import java.util.Date;
class SimpleDateFormatExample2
{
        public static void main(String[] args)
        {
                Date date = new Date();
                SimpleDateFormat formatter = new SimpleDateFormat("MM/dd/yyyy");
                String strDate = formatter.format(date);
                System.out.println("Date Format with MM/dd/yyyy : "+strDate);

                formatter = new SimpleDateFormat("dd-M-yyyy hh:mm:ss");
                strDate = formatter.format(date);
                System.out.println("Date Format with dd-M-yyyy hh:mm:ss : "+strDate);

                formatter = new SimpleDateFormat("dd MMMM yyyy");
                strDate = formatter.format(date);
                System.out.println("Date Format with dd MMMM yyyy : "+strDate);

                formatter = new SimpleDateFormat("dd MMMM yyyy zzzz");
                strDate = formatter.format(date);
```

```
                System.out.println("Date Format with dd MMMM yyyy zzzz : "+strDate);

                formatter = new SimpleDateFormat("E, dd MMM yyyy HH:mm:ss z");
                strDate = formatter.format(date);
                System.out.println("Date Format with E, dd MMM yyyy HH:mm:ss z : "
+strDate);
        }
}
```

Temporal Adjusters Class:

```
import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.temporal.TemporalAdjusters;
public class TemporalAdjusterExample
{
    public static void main(String[] args)
    {

        LocalDate now = LocalDate.now();
        System.out.println("Current date : " +now);

        LocalDate output = null;

        output = now.with(TemporalAdjusters.firstDayOfMonth());
        System.out.println("firstDayOfMonth :: " + output);

        output = now.with(TemporalAdjusters.firstDayOfNextMonth());
        System.out.println("firstDayOfNextMonth :: " + output);

        output = now.with(TemporalAdjusters.firstDayOfNextYear());
        System.out.println("firstDayOfNextYear :: " + output);

        output = now.with(TemporalAdjusters.firstDayOfYear());
        System.out.println("firstDayOfYear :: " + output);

        output = now.with(TemporalAdjusters.lastDayOfYear());
        System.out.println("lastDayOfYear :: " + output);

        output = now.with(TemporalAdjusters.dayOfWeekInMonth(2, DayOfWeek.FRIDAY));
        System.out.println("dayOfWeekInMonth(1, DayOfWeek.FRIDAY) :: " + output);

        output = now.with(TemporalAdjusters.lastDayOfMonth());
        System.out.println("lastDayOfMonth :: " + output);

        output = now.with(TemporalAdjusters.lastDayOfYear());
        System.out.println("lastDayOfYear :: " + output);
    }

}
```

Exception Handling: Introduction

- An exception is an abnormal condition that arises in a code sequence at run time. Exception is a run time error.
- Java and other programming languages have mechanisms for handling exceptions that you can use to keep your program from crashing. In Java, this is known as catching an exception/exception handling.
- When java interpreter encounters an error, it creates an exception object and throws it(informs us that an error occurred).
- If the exception object is not caught and handled properly, the interpreter will display a message and stops the program execution.
- If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions.

Errors are broadly classified into two categories.

1. Compile time exception(error)
2. Run Time exception(error)

Compile-time errors:

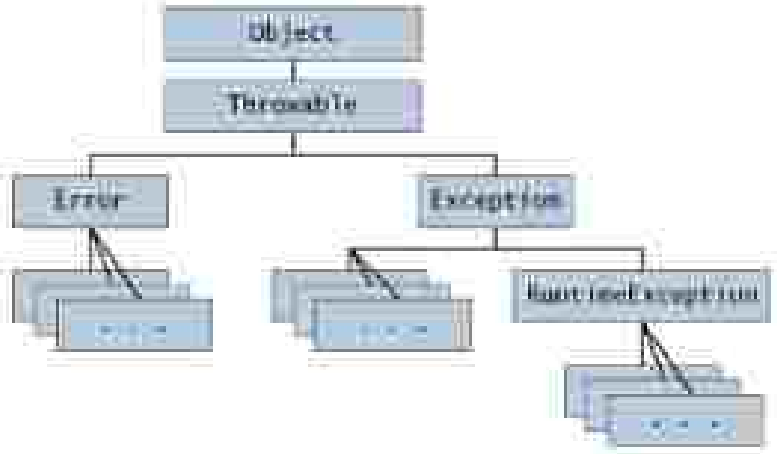
- All syntax errors will be detected and displayed by the java compiler and therefore these errors are known as compile-time errors.
- Whenever the compiler displays an error, it will not create the .class file.

Run-time errors:

- Sometimes a program may compile successfully creating the .class file but may not run properly because of abnormal conditions:
- Common run-time errors are
    - The file you try to open may not exist.
    - Dividing an integer by Zero.
    - Accessing an element that is out of the bonds of an array type.
    - Trying to store a value into an array of an incompatiable class or type.
    - Trying to cast an instance of a class to one of its subclasses.
    - And many more.....

Each exception is a class, part of java.lang package and it is derived from Throwable class.

Hierarchy of Standard Exception Classes:



There are two types of exceptions in Java:

- Unchecked Exceptions
- Checked Exceptions

Unchecked Exceptions:

- Unchecked exceptions are RuntimeExceptions and any of its subclasses.
- Error class and its subclasses are also called unchecked exceptions.
- Compiler does not force the program to catch the exception or declare in a throws clause.
    - Ex: ArithMeticException
- Unchecked exceptions can occur anywhere in a program and in a typical program can be very numerous.

Checked Exceptions:

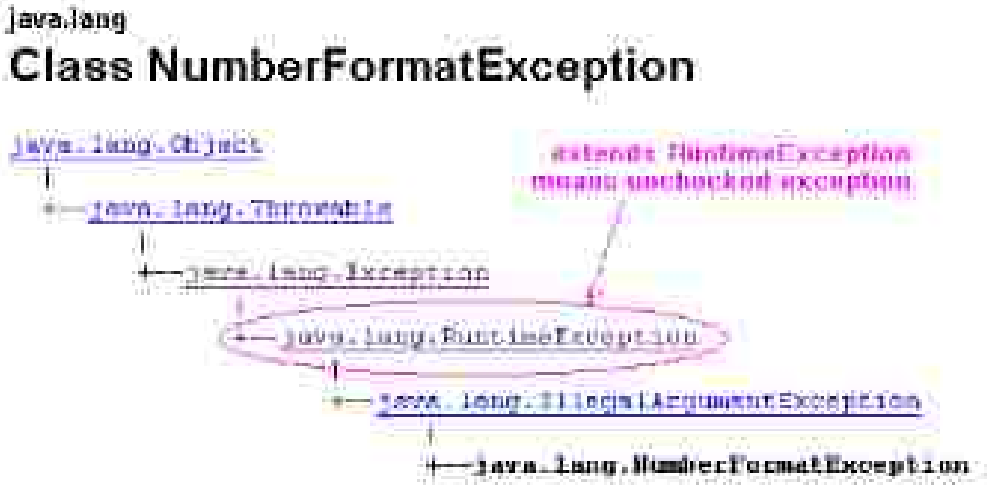
- A checked exception is any subclass of Exception (or Exception class itself), excluding class RuntimeException and its subclasses.
- Checked exceptions must be handled by the programmer to avoid a compile-time error.
- There are two ways to handle checked exceptions.
    - Declare the exception using a throws clause.
    - catch the exception.

The compiler requires a throws clause or a try-catch statement for any call to a method that may cause a checked exception to occur.

- Checked Exceptions are checked at compile-time, where as unchecked exceptions are at runtime.

We can know whether the exception is Checked or Unchecked in two ways:

- Using Java API
- By Experience



Some common exceptions Unchecked (Runtime Exceptions):

Exception type	Cause of exception
ArithmeticException	Caused by math errors such as division by zero
ArrayIndexOutOfBoundsException	Caused by bad array indexes
ArrayStoreException	Caused when a program tries to store the wrong type of data in an array
ClassCastException	Invalid cast
NullPointerException	Invalid use of a null reference

NumberFormatException	Invalid conversion of a string to a numeric format
StringIndexOutOfBounds	Attempt to index outside the bounds of a string

Some common exceptions Checked (Compiletime Exceptions):

Exception type	Cause of exception
ClassNotFound	Class not found
IllegalAccessException	Access to a class denied
InstantiationException	Attempt to create an object of an abstract class or interface
NoSuchFieldException	A requested field does not exist
NoSuchMethodException	A requested method does not exist

Java Exception handling is managed by 5 keywords:

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

If these abnormal conditions are not handled properly, either the program will be aborted or the incorrect result will be carried on causing more and more abnormal conditions.

Example to see what happens if the exceptions are not handled:

```
class NoException
{

      public static void main(String[] args)
      {
       int no=10;
       int r=0;
       r=no/0;
       System.out.println("Result is:"+r);
      }
}
```

The above program gives the following error:

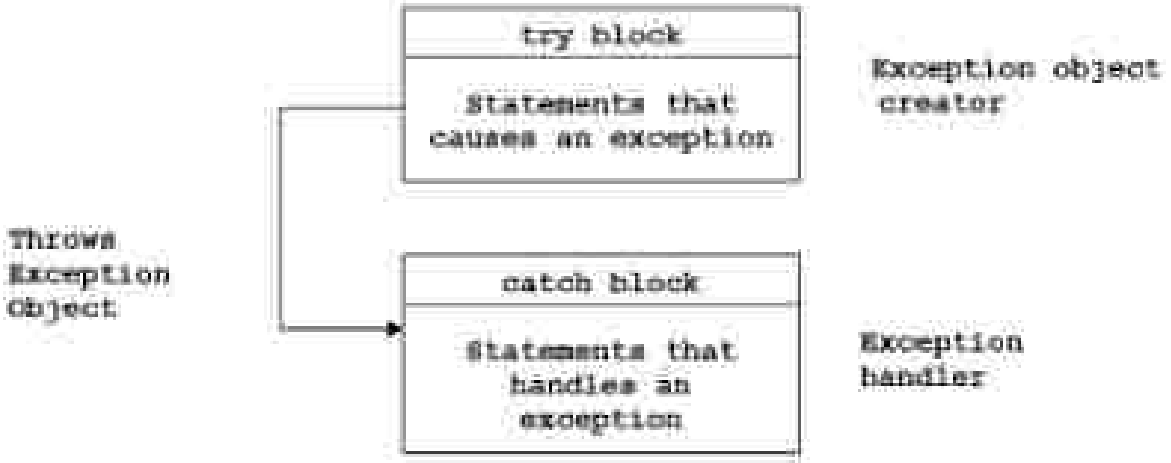
```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at NoException.main(NoException.java:8)
```

In the above program the error is handled by the default exception handler which is provided by java run-time system.

If the exception is handled by the user then it gives two benefits.

1. Fixes the error.
2. Avoids automatic termination of program.

Syntax for exception handling code:



```
try
        {
         statement;    // generates exception
        }
        catch(Exception-type e)
        {
         statement;    // process the exception
        }
```

- Java uses a keyword try to preface a block of code that is likely to cause an error and "throw" an exception.
- A catch block is defined by the keyword catch "catches" the exception "thrown" by the exception block.
- Catch block should be immediately after the try block.
- try block can have one or more statements that could generate an exception.

- If any one statement generates an exception, the remaining statements in the block are skipped and control jumps to the catch block that is placed next to the try block.
- The catch block too can have one or more statements that are necessary to process the exception.
- Every try block should be followed by at least one catch statement.
- Catch statement works like a method definition, contains one parameter, which is reference to the exception object thrown by the try block.

Program:

```
class TryCatch
{
  public static void main(String[] args)
  {
    int a=10;
    int b=5;
    int c=5;
    int x;

    try
    { x=a/(b-c); // exception here
      System.out.println("This will not be executed.....");
    }
    catch(ArithmeticException e)
    { System.out.println("Cannot divide by 0.....");  }
    System.out.println("After catch statement.....");
  }
}
```

Note: Once an exception is thrown, control is transferred to catch block and never returns to try block again.

***Statements after the exception statement(try block) never get executed.

Multiple Catch Clauses:

Syntax:

```
try
{
 statement;
}
catch(Exception-type 1 e)
{
 statement;  // process exception type 1
}
catch(Exception-type 2 e)
{
 statement;  // process exception type 2
}
catch(Exception-type N e)
{
 statement;  // process exception type N
}
```

- When an exception in a try block is generated, java treats the multiple catch statements like cases in switch statement.
- The first statement whose parameter matches with the exception object will be executed, and the remaining statements will be skipped.
- Code in the catch block is not compulsory.
  - catch (Exception e){}
  - the catch statement simply ends with a curly braces({}), which does nothing, this statement will catch an exception and then ignore it.

```
import java.util.*;

class MultiCatch
{
 public static void main(String args[])
  {
   int no,d,r;
   no=d=r=0;

 try
  {
   no=Integer.parseInt(args[0]);
   d=Integer.parseInt(args[1]);
```

```
r=no/d;
System.out.println("Result :"+r);
}
catch (NumberFormatException nf)
{
  System.out.println(nf);
}
catch (ArrayIndexOutOfBoundsException ai)
{
System.out.println(ai);
}
catch (ArithmeticException ae)
{
 System.out.println(ae);
}
}
}
```

try-with-resources:

In Java, the try-with-resources statement is a try statement that declares one or more resources. The resource is as an object that must be closed after finishing the program. The try-with-resources statement ensures that each resource is closed at the end of the statement execution.

Syntax:

```
try(resource code)
{
        use resources
}
catch()
{
 handle exceptions
}
```

Example:

```
import java.io.*;
class TryWithRes
{
        public static void main(String args[])
        {

                try (BufferedReader br=new BufferedReader(new FileReader("Sample.java")))
                {
                        String line;
                        while((line=br.readLine())!=null)
                        {
                                System.out.println(line);
                        }
```

```
            }
            catch(IOException e)
            {
                    System.out.println(e);
            }
        }
}
```

Custom Exceptions:

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, Java custom exceptions are used to customize the exception according to user need.

Example:

```
public class WrongFileNameException extends Exception {
  public WrongFileNameException(String errorMessage) {
  super(errorMessage);
  }
}
```

Nested try and catch Blocks:

In Java, using a try block inside another try block is permitted. It is called as nested try block.

Syntax:

```
...
//main try block
try
{
  statement 1;
  statement 2;
//try catch block within another try block
  try
  {
    statement 3;
    statement 4;
//try catch block within nested try block
    try
    {
      statement 5;
      statement 6;
```

```
        }
          catch(Exception e2)
          {
//exception message
          }

        }
        catch(Exception e1)
        {
//exception message
        }
}
//catch block of parent (outer) try block
catch(Exception e3)
{
//exception message
}
```

—

Example:

```
class NestedExcep
{
        public static void main(String args[])
        {
                try
                {

                        try
                        {

                                System.out.println("going to divide");
                                int b =39/0;
                        }
                catch(ArithmeticException e)
                {
                        System.out.println("A number can't be divide with zero");
                }
```

```
                try
                {
                int a[]=new int[3];
                        a[5]=4;
                }
            catch(ArrayIndexOutOfBoundsException e)
            {
                System.out.println(e);
            }

            }
            catch(Exception e)
            {
                System.out.println("handeled");
            }

        }
}
```

Throws Clause or Rethrowing Exception

```
import java.io.*;
 class Test
 {
  void doWork() throws IOException
  {
   throw new IOException();
  }
 }

 class ThrowsTest1
 {
  public static void main(String args[])  throws IOException
  {
  Test t1=new Test();
  t1.doWork();
  }
  }
```