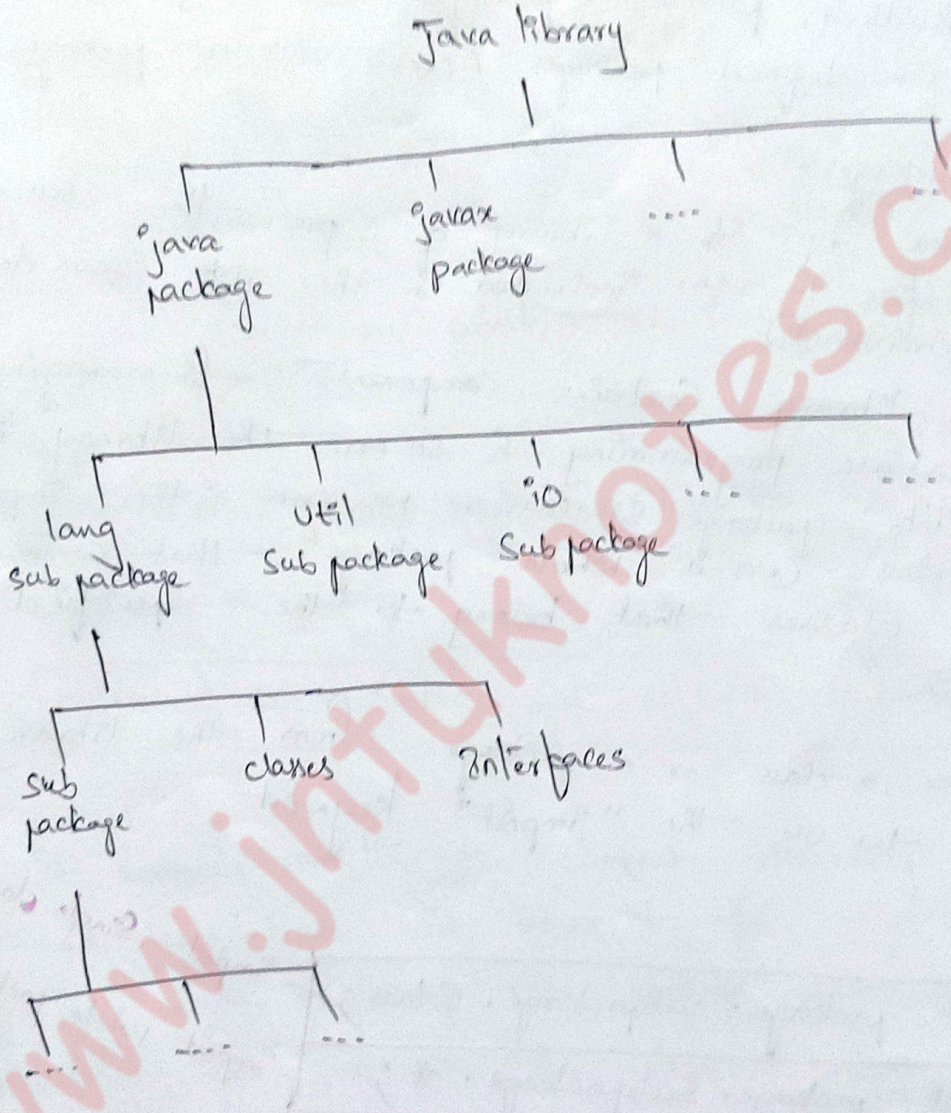
Packages and Java Library

Introduction:

- A package is a collection of Sub packages, classes & Interfaces .. etc Java library organized in the form of packages.



- Java compiler generates ".class" file for every class & every interface & a Source file
- Java compiler generates folder for every package & every Sub package.
- by using "package" keyword we create packages.

Defining a package:

A package in java is used to group related classes. We use packages to avoid name conflicts & to write a better maintable code. Packages are divided into two categories.

1. Built-in packages (packages from the java API)
2. User-defined packages (create our own packages)

Built-in packages:

The java API is a library of prewritten classes, that are free to use, included in the JDE (Java development Environment).

The library contains components for managing input, database programming & so on. The library is divided into packages & classes. We can either import a single class (or a whole package that contains all the classes that belong to the specified package.

To use a class or a package from the library we need to use the "import" keyword

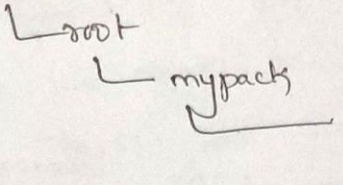
Syntax:

```
import package.subpackage.Class;   // import single class
import package.subpackage.*;       // import whole package.
```

User-defined packages:

To create our own packages, we need to understand that java uses a file system directory to store them. just like folders on our computer.

Ex:



- To create a package, use the "package" keyword

MyPackageClass.java

```
package mypack;
class MyPackageClass
{
    public static void main(String args[])
    {
        System.out.println("this is my package");
    }
}
```

- c:\Users\513> javac MyPackageClass.java

To compile the package

```
c:\Users\ src > javac -d c:\work MypackageClass.java
```

-d: it omits the current directory & specifies the destination for where to save the class file. If we want to keep the package within the same directory, we can use the "." (dot) sign.

```
c:\Users\ src > javac -d . MyPackageClass.java
```

When we compiled the package in the above example, a new folder was created, called "mypack".

importing packages & classes into programs

import a class:

If we want to use a class for example "Scanner" class, which is used to get input then we have to write the statement like

```
import java.util.Scanner;
```

In the above statement, java → is a main package
util – is a sub package
Scanner – is a class name.

- To use the Scanner class, we can create an Object & use any of the available methods found in the Scanner class.

Ex:

```
import java.util.Scanner;
class A
{
public static void main(String args[])
{
Scanner s=new Scanner(System.in);
System.out.println("Enter User name");
String userName= s.nextLine();
System.out.println("Username is:"+userName);
}
}
```

In the above program we use Scanner class & nextLine() method in the Scanner class. so we import only Scanner class in util package

import a package:

If we dont know the class name of particular method, then we can import a whole package to our program. then all classes, interfaces, methods are import to our program.

To import a whole package, end the sentence with an asterisk sign (*).

Ex:

```
import java.util.*;
```

by using the above statement we can use all the classes & interfaces of util package

path and class path

PATH & CLASS PATH are environmental variables on microsoft windows

windows 7: PATH

1. from the desktop, right click on Computer icon
2. choose properties from the menu
3. click the Advanced System Settings link
4. click Environment variables. In the section System variables, find the PATH environment variable & select it. click Edit if the PATH environment variable does not exist click new.
5. In the Edit System variable (or new System Variable) window, specify the value of the PATH Environment variable. click OK. close all remaining windows by clicking OK.

Classpath

```
c:\> set path="%path%";c:\program files\java\jdk1.8.0\bin;
```

(or) `%CLASSPATH%`

⇝ Access control

private: private members of class are not accessible on other classes of the same package or another package. i.e within the class only accessible.

public: public members of a class available in the other classes of the same package or another package i.e from any where we can access.

protected: protected members are accessible in the sub classes of same package but not in another package

default: Default members are available in other classes of same package but not in another package

⇝ Packages in JAVA SE

1. java.lang → basic language functionality & fundamental types
2. java.util → collection of data structure classes
3. java.io → file operations
4. java.math → multiprecision arithmetics
5. java.nio → non-blocking I/O framework for java
6. java.net → networking operations, sockets, DNS lookups
7. java.security → key generation, encryption & decryption
8. java.sql → java database connectivity (JDBC) to access databases
9. java.awt → basic hierarchy of packages for native GUI components
10. java.text → provides classes & interfaces for handling text, dates, numbers & messages in a manner independent of natural languages.

11. java.rmi → provides the Rmi package

12. java.time → the main APIs for dates, times, instants & durations.

13. java.beans → it contains classes & interfaces related to JavaBeans Components

14. java.applet → provides classes & methods to create & communicate with the applets.

java.lang.Object - Super class of all classes

"java.lang" package is a default package. It is available without the use of an import statement.

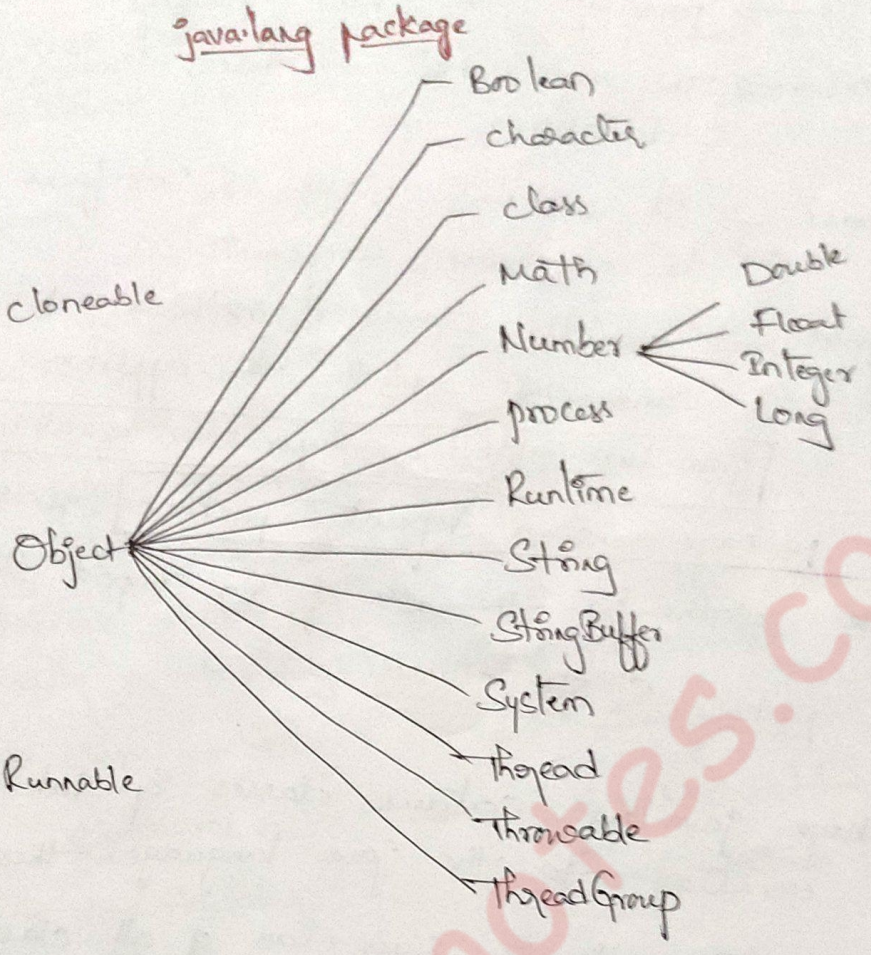
→ java.lang package & its classes

The package java.lang contains classes & interfaces that are essential to the java language. These include

- Object : the ultimate super class of all classes in java.
- Thread : the class that controls each thread in a multi threaded program.
- Throwable : the super class of all error & exception classes in java

- classes that encapsulate the primitive data types in java.
- classes for accessing System resources & other low-level entities.
- Math : a class that provides standard mathematical methods
- String : the class that is used to represent strings

- The java.lang package is implicitly imported by every java source file.



classes:

Object, Thread, ThreadGroup, Throwable

Interfaces: Cloneable, Runnable,

Abstract class: Number, process

Non-Instantiable class: Class, Runtime, System

final classes: Boolean, character, Math, String, StringBuffer, Double, float, Integer, Long

*
**
→ class Object:

Object class is present in java.lang package. Every class in java is directly (or) indirectly derived from the Object class. If a class does not ~~exist~~ extend any other class then it is direct child class of Object. & if extends other class then it is an indirectly

derived. Therefore the object class methods are available to all java classes. hence object class acts as a root of inheritance heirarchy in any java program.

methods

1. toString():

toString() provides string representation of an object & used to convert an object to String. to override toString() method to get our own String representation of object.

- whenever we try to print any object reference, then internally toString() method is called.
- Object class toString() method always return ClassName@HashCode-In-HexaDecimalFormat
- String class toString() method always return content of String object

Ex:

```
class A
{
  public static void main(String args[])
  {
    String s= new String();
    System.out.println(s);
    A a =new A();
    System.out.println(a);
  }
}
```

implicitly s.toString()

o/p: A@3e25a5

implicitly a.toString()

2. hashCode():

for every object, JVM (java virtual machine) generates a unique number which is hashcode. it returns the address of object.

3. Equals(Object obj):

Compares the given object to "this" object (the object on which the method is called). It gives the generic way to compare objects for equality.

4. getClass():

Returns the class object of "this" object & used to get actual run-time class of the object. It can also be used to get metadata of this class.

5. finalize():

This method is called just before an object is garbage collected. It is called by the Garbage Collector. We should override finalize() method to dispose system resources, perform clean-up activities & minimize memory leaks.

finalize() method is called just once in a program.

6. clone():

It returns new object that is exactly the same as this object.

7. wait(), notify(), notifyAll()

These are related to concurrency. For threads these methods are used. These are used in Thread class

⇝ Enumerations

This feature allows to create a new Data type in java. In order to use this feature "enum" keyword introduced in JDK 1.5 version in 2004.

- All enumeration literals are implicitly "static"

Ex:

```
enum Day
{
    Mon, Tue, Wed, Thu, Fri, Sat, Sun
};

class A
{
public static void main(String args[])
{
    Day d = Day.Tue;
    System.out.println(d);
}
}
```

(enumeration literals)

(because all enumeration literals are static by default, so we can call with class Name.)

- In the above program Day is the Data Type.

⇝ class Math

Java Math class provides several methods to perform on math calculations like

min()	tan()
max()	round()
avg()	ceil()
sin()	floor()
cos()	abs()

Methods:

The java.lang.Math class contains various methods for performing basic numeric operations such as logarithm, cube root, trignometric functions etc.

Math.abs(): return Absolute value of given value

Math.max(): return largest of two values

Math.min(): return Smallest of two values

Math.round(): round of the decimal numbers to the nearest value.

Math.Sqrt(): return Squareroot of a number

Math.cbrt(): return cube root of a number

Math.pow(): returns power of a number

Math.signum(): find the sign of a given value

Math.ceil(): find Smallest integer value that is greater than or Equal to the mathematical integer

Math.floor(): find largest integer value which is less than & Equal to the argument & Equal to the mathematical integer of double value

Math.random(): returns a double value with a positive sign greater than Equal to 0.0 & less than 1.0

Math.log(): returns natural logarithm of a double value

Math.log10(): return the base 10 logarithm of a double value

Math.exp(): returns Exponent value Equal to 2.71828

Math.Sin(): return trigonometric Sine value of a double value

Math.cos(): Cosine value

Math.tan(): tangent value

Math.Asin(): arc Sine value

Wrapper classes

Each of java 8 primitive data types has a class & those classes are called wrapper classes. because they wrap the data into an object.

primitive data types	wrapper class
1. byte	1. Byte
2. short	2. Short
3. int	3. Integer
4. long	4. Long
5. float	5. Float
6. double	6. Double
7. char	7. Character
8. boolean	8. Boolean

the above all wrapper classes are the part of "java.lang" package. All wrapper classes are called Reference data types.

1. java.lang.Integer

constructor: `public Integer(int);`

it is used to convert primitive data type to reference data type

method: `public int intValue();`

used to convert reference data type to primitive datatype

Ex1:

primitive

```
int x=5;
```

x → 5

```
Integer ob = new Integer(x);
```

reference data type

ob → (5) Integer object

primitive to Reference

Ex2:

```
Integer ob = new Integer(5);
```

reference

ob → (5) Integer object

```
int x = ob.intValue();
```

primitive

x → 5

Reference to primitive

java.lang.Float

`public Float(float);` - constructor - primitive to reference

`public float floatValue();` - method - reference to primitive

Auto boxing

The process of converting primitive data type to the corresponding reference data type is known as Auto boxing.

Ex:

primitive data type

```
int x=5
Integer ob=x;
```

- Auto boxing

reference data type

here we can assign a int variable to reference data type object reference.

Auto unboxing

The process of converting reference data type to the corresponding primitive data type is known as Auto unboxing

```
Integer ob = new Integer(5);
int x= ob;
```

reference data type

- Auto unboxing

primitive data type

here we can assign a object reference of reference data type to primitive data type variable.

- both Auto boxing & Auto unboxing features are introduced in JDK 1.5 version in 2004.

Ex:

```
float f=1.5;
Float ob=f;
```
} Auto boxing

```
Float ob=new Float(1.5);
float f=ob;
```
} Auto un boxing

⇒ Java Util classes & Interfaces

java.util package classes & interfaces are divided into 2 categories.

1. collection frame work collections
2. legacy collections.

- collections frame work collections are divided into 3 Sub categories

1. Core collection interfaces
2. General purpose implementations
3. More utility collections.

1. Core collection interfaces.

These are the foundations of collections frame work.

1. Collection
2. List
3. Set
4. Map
5. SortedSet
6. SortedMap
7. NavigableSet
8. NavigableMap
9. Queue
10. Deque

1. Collection Interface: it is a root interface in the heirarchy
2. List interface: it extends collection interface & it maintain sequences
3. Set interface: it extends collection interface & it maintain sets
4. Map interface: it is a 2-dimensional collection & it maintain data as a "key/value" pair
5. SortedSet interface: it extends Set Interface & it maintains sorted sets

6. SortedMap interface: It extends Map interface & it maintains sorted maps. here only keys are sorted

7. NavigableSet interface: It extends SortedSet interface & it is used to navigate elements of a set

8. NavigableMap interface: It extends SortedMap interface & it is used to navigate key/value pair of a Map

9. Queue interface: It is called as First In First Out list

10. Dequeue interface: It is called as Double Ended Queue.

2. General purpose implementations

The core collection interfaces implementations classes are called General purpose implementations

1. ArrayList
2. LinkedList
3. HashSet
4. LinkedHashSet
5. TreeSet
6. HashMap
7. LinkedHashMap
8. TreeMap
9. PriorityQueue
10. ArrayDeque

1. ArrayList: It is an implementation of Linear list data Structure & it supports all operations of linear list data structure.

2. LinkedList: It is an implementation of "Double linked list data structure". It supports all operations of double linked list data structure. here data stored in Nodes.

3. HashSet: It is an implementation of "Hashing technique" with array representation.

4. LinkedHashSet: It is an implementation of hashing technique with linked representation. here data stored in Nodes.

5. TreeSet: It is an implementation of "binary search tree technique with linked representation.

6. HashMap: It is the two dimensional collection & it maintains data as a key/value pair. implements hashing with Array representation.

7. LinkedHashMap: It is two dimensional collection & implement with hashing technique with Linked representation.

8. TreeMap: It is two dimensional collection & implementation of Binary Search tree technique with linked representation.

9. PriorityQueue: It is called as Priority In & priority out list It allows insertions at rear end & deletion at front end.

10. ArrayDeque: It is an implementation of double ended Queue data structure with array representation. It allows both insertion & deletion at both ends.

3. More Utility collections:

1. Iterator (Interface)
2. ListIterator (Interface)
3. Arrays (class)
4. Collections (class)
5. Scanner (class)

1. Iterator: used to iterate elements of a collection
2. ListIterator: used to iterate elements of a collection
3. Arrays: It contains several useful methods, used to perform on Arrays
4. Collections: It contains several useful methods, used to perform on collections
5. Scanner: used to accept data from keyboard, file & Network.

legacy collections

1. Enumeration (interface)
2. StringTokenizer (class)
3. Vector (class)
4. Dictionary (abstract class)
5. HashTable (class)
6. Random (class)
7. Stack (class)
8. Date (class)

1. Enumeration (interface): Similar to Iterator interface
2. StringTokenizer : allows to an application to break a String into Tokens
3. Vector : Similar to ArrayList class
4. Dictionary : Similar to Map interface
5. HashTable : Similar to HashMap class
6. Random : This class generates random Integers, floating point numbers & boolean values
7. Stack : called as Last In ~~Fast~~ out List & allows insertion & deletion at top end only
8. Date : used to get System date & Time

⇝ Formatter class

The Formatter is a built-in class in java used for layout justification & alignment, common formats for numeric, String, date/time data. The Formatter class is defined as final class inside the java.util package.

- The Formatter class implements Closeable & Flushable interface.

Constructors:

1. Formatter () : creates new Formatter
2. Formatter(Appendable a) : Creates new formatter with specified destination.
3. Formatter (File file) : creates new formatter with specified file
4. Formatter (File file, String charset) : creates new formatter with specified file & character set
5. Formatter (Locale l) : creates new formatter with specified locale.
6. Formatter (OutputStream os) : creates new formatter with specified output stream.

7. Formatter(PrintStream ps): creates new formatter with specified print stream

8. Formatter(String fileName): creates new formatter with specified file name.

methods:

1. Formatter format(Locale l, String format, Object ... args)
   - it writes a formatted string to the invoking objects destination using the specified locale, format string & arguments

2. void flush(): it flushes the invoking formatter
3. Appendable out(): it returns the destination of the output
4. Locale locale(): returns locale set by the construction of the invoking formatter
5. String toString(): converts invoking objects to String
6. IOException ioException(): returns the IOException last thrown by the invoking formatters Appendable
7. void close(): closes the invoking formatter.

⇝ Random class

It is in java.util package. It generates "random integers, floating point numbers & boolean values".

java.util.Random

Constructor: `public Random();`

methods: 1. public int nextInt();
it returns one number within int range

2. public int nextInt(int);
it returns one number between 0 & specified number

3. public boolean nextBoolean();
4. public float nextFloat();

Ex:

```
import java.util.*;
class A
{
public static void main(String args[])
{
  Random r = new Random();
  boolean b = r.nextBoolean();
  System.out.println(b);
 for(int i=1; i<=10; i++)
 {
   System.out.println(r.nextInt(100));
 }
}
}
```

o/p: 1 80
99 9
87 11
44 97
55
67

→ Time package — java.time

This package is used for all the date, time related operations. the various classes in this package are.

1. Clock :- provides access to current date, time according to the zone specified.
2. Duration :- it is the amount of time spent/recorded etc.
3. Instant :- current time.
4. LocalDate :- it is a date without any specified time zone.
5. LocalDateTime :- it is a date-time without a time zone.
6. LocalTime :- it represents a time without a time zone.
7. MonthDay :- it represents a day of the month in the calendar
8. OffsetDateTime :- the class represents a date time with an offset
9. OffsetTime :- it represents the time with an offset

10. period :- represents a amount of time
11. year :- it represents a year in the calendar system
12. YearMonth :- it represents a combination of the month & the year with respect to the calendar system
13. ZonedDateTime :- represents a date-time with the specified time zone.
14. ZoneId :- gives the idea of any time zone
15. ZoneOffset :- represents the offset of any time zone

=> class Instant

It is in java.times package. it is used to represent the specific moment on the timeline. it inherits the Object class & implements the Comparable interface

java.time.Instant

public final class Instant extends Object implements Temporal, TemporalAdjuster, Comparable, Serializable

methods:

1. Temporal adjustInto(Temporal temporal):- used to adjust the specified temporal object to have this instant.
2. int get(TemporalField field):- used to get the value of the specified field from this instant as an int
3. boolean isSupported(TemporalField field):- used to check if the specified field is supported
4. Instant minus(TemporalAmount amountToSubtract):- used to return a copy of this instant with the specified amount subtracted
5. static Instant now():- used to obtain the current instant from the system clock
6. static Instant parse(CharSequence text):- used to obtain an instance of Instant from a text String such as 2007-12-03T10:15:30.002.

7. Instant plus(TemporalAmount amountToAdd):- used to return a copy of this instant with the specified amount added

8. Instant with(TemporalAdjuster adjuster):- used to return an adjusted copy of this instant

Ex:

```
import java.time.Instant;
public class Instant
{
  public static void main(String args[])
  {
    Instant inst = Instant.parse("2017-02-03T10:37:30.00Z");
    System.out.println(inst);
  }
}
```

o/p: 2017-02-03T10:37:30Z

Ex:

```
import java.time.*;
public class Instant
{
  public static void main(String args[])
  {
    Instant inst = Instant.parse("2017-02-03T11:25:30.00Z");
    Instant inst2 = inst.plus(Duration.ofDays(125));
    System.out.println(inst2);
  }
}
```

o/p: 2017-06-08T11:25:30Z

⇒ Formatting of Date/Time in Java

Formatting is used to separate the date from the time. so we can use DateTimeFormatter class with the ofPattern() method in the same package to format date-time objects. In the above example "T" & nano-seconds are display but by using ofPattern() method we will remove both T & nano-seconds. from the date-Time.

Ex:

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
public class A
{
public static void main(String args[])
{
LocalDateTime obj = LocalDateTime.now();
System.out.println("before formatting:" + obj);
DateTimeFormatter obj2 = DateTimeFormatter.ofPattern
("dd-mm-YYYY HH:mm:ss");
String formattedDate = obj.format(obj2);
System.out.println("After formatting:" + formattedDate);
}}
```

O/P: Before formatting : 2021-04-27T11:54:30.526283
After formatting : 27-04-2021 11:54:30.

The ofPattern() method accepts all sorts of values.

Ex:

YYYY-MM-dd	- 1988-09-29
dd/MM/YYYY	- 29/09/1988
dd-MMM-YYYY	- 29-Sep-1988
E, MMM dd YYYY	- Thu, Sep 29 1988

⇒ Temporal Adjusters class

java.time.temporal.TemporalAdjusters

This class provides Adjusters, which are a key tool for modifying temporal objects. i.e date like "second Saturday of the month" or "next Tuesday" etc.

This class can be used in two ways

1. invoking the method on the interface directly
2. by using Temporal.with(TemporalAdjuster):

but 2nd way is recommended approach.

methods:

1. dayOfWeekInMonth(int ordinal, DayOfWeek dayOfWeek)
2. firstDayOfMonth()
3. firstDayOfNextMonth()
4. firstDayOfNextYear()
5. firstDayOfYear()
6. firstInMonth(DayOfWeek dayOfWeek)
7. lastDayOfMonth()
8. lastDayOfYear()
9. lastInMonth(DayOfWeek dayOfWeek)
10. next(DayOfWeek dayOfWeek)
11. nextOrSame(DayOfWeek dayOfWeek)
12. previous(DayOfWeek dayOfWeek)
13. previousOrSame(DayOfWeek dayOfWeek)

Ex:

```
import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.temporal.TemporalAdjusters;

public class A
{
public static void main(String args[])
{
    A a=new A();
    a.testAdjusters();
}
public void testAdjusters()
{
LocalDate date1=LocalDate.now();
System.out.println("Today's date:"+date1);
LocalDate nextTuesday = date1.with(TemporalAdjusters.
        next(DayOfWeek.MONDAY));
System.out.println("Next Monday is"+nextTuesday);
LocalDate firstInYear=LocalDate.of(date1.getYear(),
        date1.getMonth(),1);
LocalDate SecondSaturday = firstInYear.with(TemporalAdjusters.
    nextOrSame(DayOfWeek.SATURDAY)).with(TemporalAdjusters.
    next(DayOfWeek.SATURDAY));
System.out.println("Second saturday is"+secondSaturday);
```

O/p:

```
Today's date : 2021-02-24
Next Monday is : 2021-03-01
Second Saturday is : 2021-02-13.
```

UNIT-4

I. Packages and Java Library:

1. Introduction
2. Defining Package
3. Importing Packages and Classes into Programs
4. Path and Class Path
5. Access Control
6. Packages in Java SE
7. Java.lang Package and its Classes
8. Class Object
9. Enumeration
10. class Math
11. Wrapper Classes
12. Auto-boxing and Auto- unboxing
13. **java.util** Classes and Interfaces
14. Formatter Class
15. Random Class Time Package
16. Class Instant (java.time.Instant)
17. Formatting for Date/Time in Java
18. Temporal Adjusters Class
19. Temporal Adjusters Class.

II. Exception Handling:

1. Introduction
2. Hierarchy of Standard Exception Classes
3. Keywords throws and throw
4. try catch and finally Blocks
5. Multiple Catch Clauses
6. Class Throwable
7. Unchecked Exceptions
8. Checked Exceptions
9. try-with-resources
10. Catching Subclass Exception
11. Custom Exceptions
12. Nested try and catch Blocks
13. Rethrowing Exception
14. Throws Clause

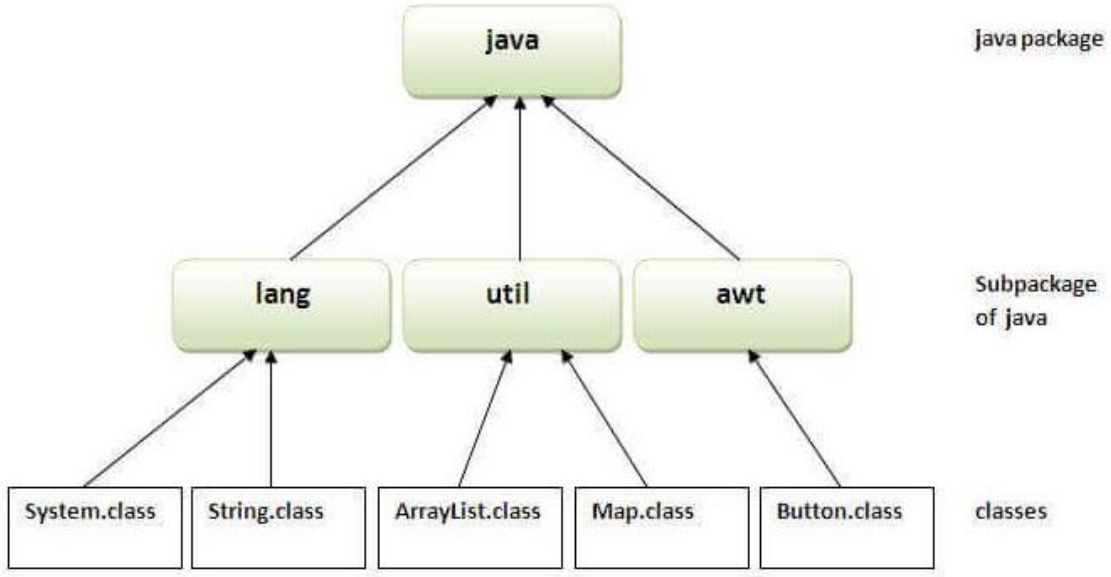
I. Packages and Java Library:

1. Introduction

- A **Java package** is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form,
  i. built-in package and
  ii. user-defined package.
- There are many built-in packages such as java.lang, java.awt, java.javax, java.swing, java.util, etc.
- Here, we will have the detailed learning of creating and using user-defined packages.

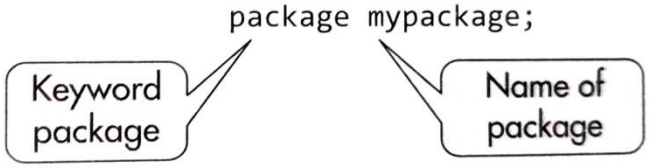
Advantage of Java Package

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.



2. Defining Package

- A package comprises a group of similar type of classes, interfaces, and sub-packages. It is defined with the keyword package followed by the name of the package and a semicolon, as illustrated in the following figure.



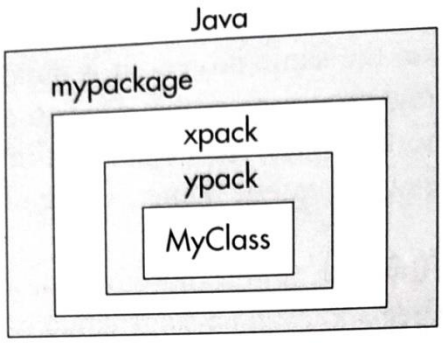


Fig. 10.1 Illustration of hierarchy of packages

```
java.mypackage.xpack.ypack.MyClass;
```

Here different levels in package are seperated by a period (.)

3. Importing Packages and Classes into Programs

- In Java, the **import** statement is used to bring certain classes or the entire packages, into visibility. As soon as imported, a class can be referred to directly by using only its name.
- The import statement is a convenience to the programmer and is not technically needed to write complete Java program. If you are going to refer to some few dozen classes into your application, the import statement will save a lot of time and typing also.
- In a Java source file, the import statements occur immediately following the package statement (if exists) and before any class definitions.
- If a program needs several classes of a package, it is better to import the whole package. This may be done as illustrated in the following example. For importing package awt contained in package java, the code is.

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

Importing Class in Programs

- All the classes defined in Java or by a programmer have to be in a package.
- A class may be include in a program by importing it through its fully qualified name.
- The selection of packages, sub-packages, and classes achieved by dot (.) selection operator, as shown in the following code

```
import java.packageP1.packageP2.package3.ClassName
```

Here

```
java      – Name of the package
packageP1 – Sub Package of Java
packageP2 – Sub Package of packageP1
packageP3 – Sub Package of packageP2
ClassName - Name of the Class in PackageP3.
```

User-defined Packages and Classes

- One of the important features of the Java programming language is that it allows a programmer to build his/her own packages and classes and make use of them whenever the need arises for the use of standard Java library packages and classes.
- A programmer can save his/her own packages and classes in a directory, and from there, it can be imported to the ones required into his/her application program by using import keyword.
- For this, the following points should be taken care of while constructing a folder for a user's own package.

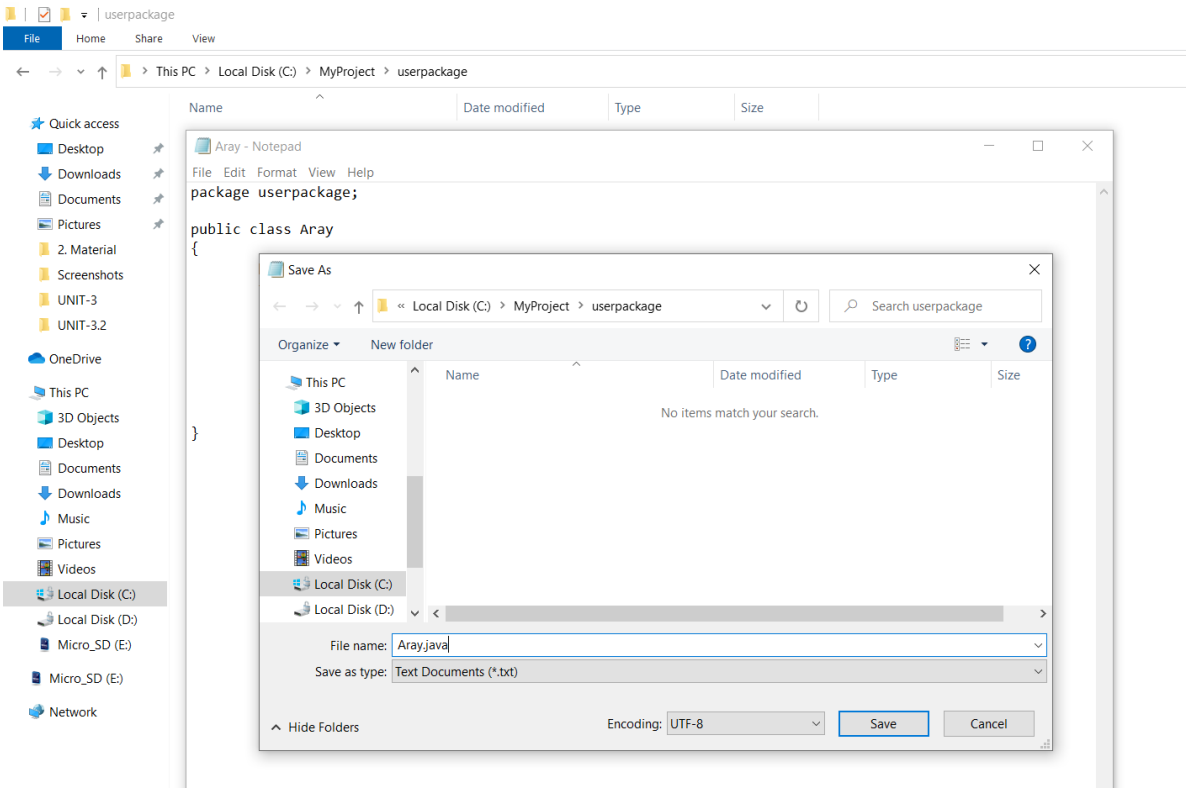
1. The folder name should be same as the name of the package desired to be created,
2. Create the desired class required to imported. Save it in the folder.
3. Compile the folder and class. However, do not run it because the class will not have the main. The application class to which it is imported will have the main method.
4 The application class should be saved outside this folder in any directory of your choice.

Example: The following class is constructed and saved in "userpackage" folder, which is saved in MyProject folder in Local Disk (C)

Program1: Aray.java in userpackage folder

```
package userpackage;

public class Aray
{
      public Aray()
      {
            int[] num= new int []{1,2,3,4,5};
            System.out.println("Array elements are : ");
            for(int x: num)
              System.out.print(x + "   ");
            System.out.println();
      }
}
```

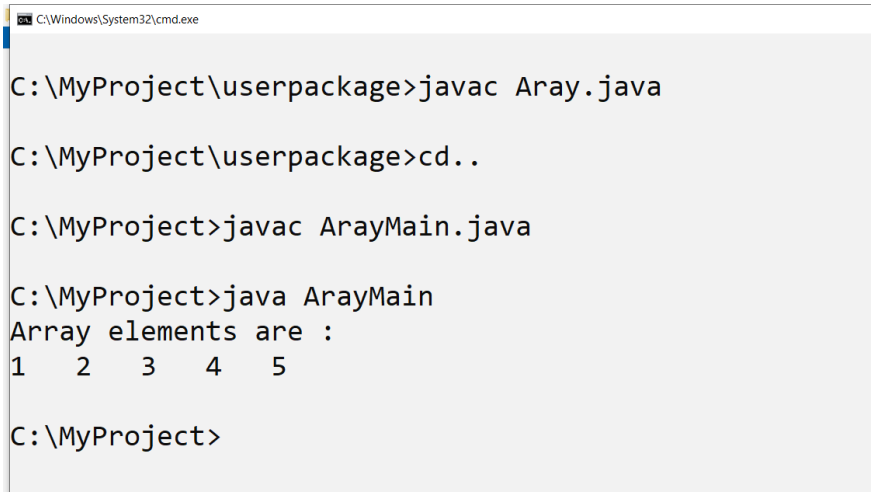


Program2: ArayMain.java in MyProject folder in Local Disk (C)

```
import userpackage.*;

public class ArayMain
{
        public static void main(String args[])
        {
            Aray objAray = new Aray();
        }
}
```

Execution Process:



4. Path and Class Path

- The PATH is an environmental variable used by the operating system to find the executive binary files

  Javac, which compiles the source code into bytecode, and
  java, which interprets the bytecode through for execution of the program
- such commands Path tells the system where to locate the JDK files that contain these commands.
- If the programmer is using classes or packages other than the Java standard their location is specified through classpath.
- The directory tree after the Java Development Kit (JDK) has been installed in Windows OS would look as illustrated in the following Diagram. The path is indicated by arrows

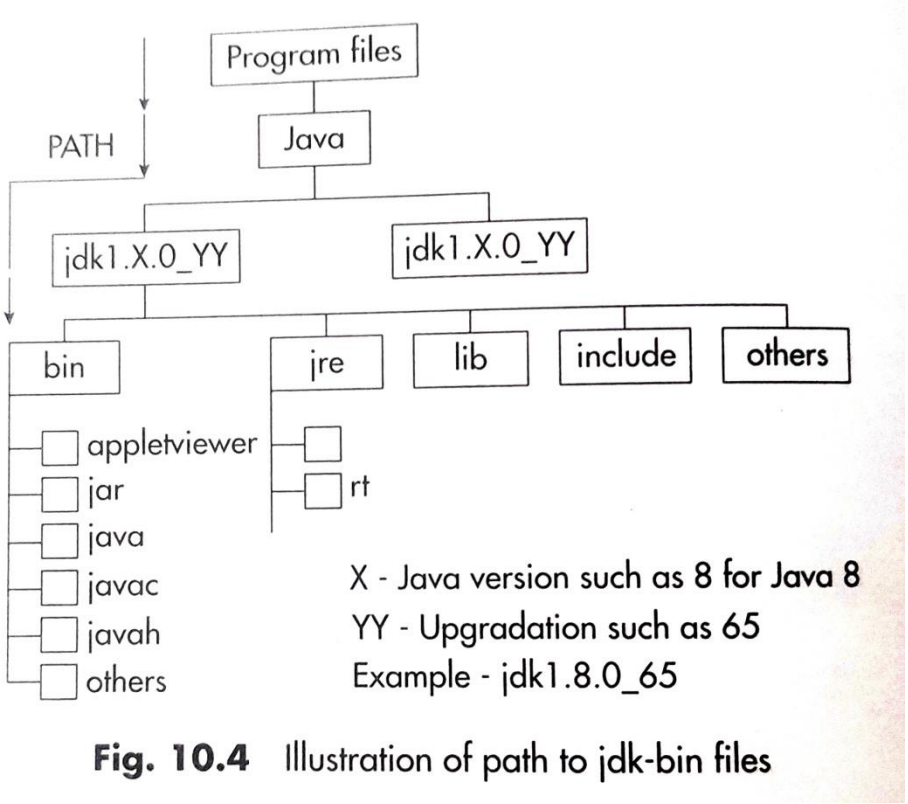


Fig. 10.4 Illustration of path to jdk-bin files

The system has to reach the location of jdk1.x.0_ yy\bin, which contains javac and java commands The path may be set in two ways

1. Set the path on Command Prompt for individual programs.
2. Set the path as value of environmental variable. Once done, you can run any number of programs without bothering about it, as done in running **ArayMain** class.

The path can be set for individual programs. Therefore, the path to JDK files installed in our computer is specified as

```
C:\Program Files\Javaljdk1.8.0_65\bin
```

Path is set by command set as

```
set path=%path%;C:\Program Files\Javaljdk1.8.0_65\bin
```

The important thing about setting path is that there will be no blank in assignment.

CLASSPATH is another environment variable. It is used by System or Application ClassLoader to locate and load **.class** that comprises the compiled Java bytecodes. Like path, the classpath is also not case sensitive and you may type it as CLASSPATH, ClassPath, or classpath.

Enter the path value as indicated in the following figure.

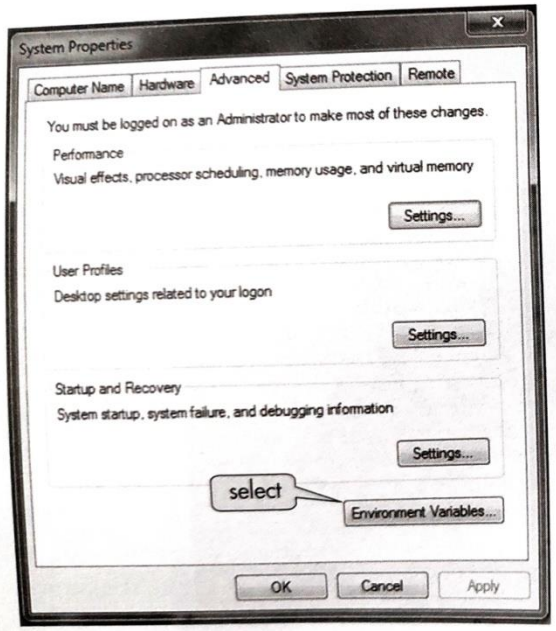


Fig. 10.7 Advanced system setting

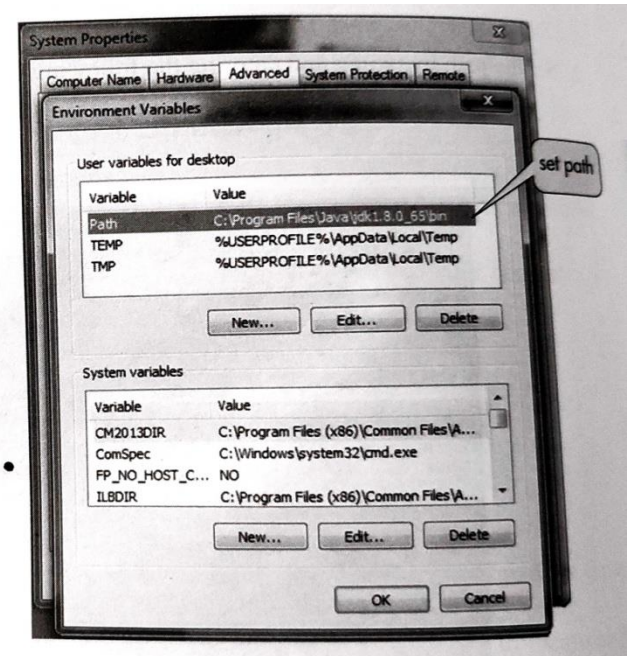


Fig. 10.8 Setting path value

5. Access Control

In Java, There are three access specifiers are permitted:

- public
- protected
- private

The coding with access specifiers for variables is illustrated as

```
Access_specifier type identifier;
```

Detials of Access specifiers are as follows.

Table 10.1 Access specifiers for classes

Access specifier	*Example of code*	*Access permitted*
No access specifier	`class A {}`	Accessible to classes in same package
public	`public class B{}`	Accessible to all classes in all packages
protected	`public class A{ protected class C {} }`	Used with nested classes; members of host (outer) class can access protected class because it is a member of the class
private	`public class A{ private class C {} }`	Used with nested classes; members of host (outer) class can access private class because it is a member of the class

Table 10.2 Access specifiers and permissible access for class members and different packages

Access specifier	*Access*
No access specifier	Access permitted to any other class belonging to the same package
public	Access permitted to any class in any package
protected	Access permitted to its subclasses in any package and also to any class in the same package
private	Access permitted to members of the same class only; no access permitted to any code outside the class

Table 10.3 Access to members permitted by specifiers

Class/Access specifier	*Access permitted Yes/No*			
	No specifier	*Public*	*Protected*	*Private*
Members of same class	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	No
Any other class in same package	Yes	Yes	Yes	No
Subclass in a different package	No	Yes	Yes	No
Any other class in any package	No	Yes	No	No

6. Packages in Java SE

Java has been evolving since its inception as Java 1. The latest version of Java is Java SE 8 (The **Java** Standard Edition 8). Originally all were supposed to be part of Java SE 7; however, it was taking more time, and Java SE 7 was released with some changes/ modifications and the remaining enhancements of Java were released in Java 8. It is worthwhile to at the additions and modifications in Java 7 and Java 8.

7. java.lang Package and its Classes

java.lang package is imported into every Java program by default. The programmer does not have to use the statement `import java.lang;` in the program. This package contains classes and interfaces that are essential for the design of Java programming language and are as follows:

i. Object class is the super class of all other classes in Java. It is the root of the class hierarchy
ii. Classes encapsulate the primitive data types in Java.
iii. Classes access system resources and other low-level entities.
iv. The class Math provides standard mathematical functions, for example, sine, cosine, square root, etc
v. The class Throwable is the super class of all error and exception classes in Java. It deals with the object that can be thrown by the throw statement. Subclasses of Throwable represent errors and exceptions
vi. The class Thread controls each thread in a multithreaded program.
vii. Classes String and StringBuffer provide commonly used operations on character strings.

It is all very necessary for a programmer to know the different classes available

Table 10.4 gives a brief description of the classes available in this package.

Class name	Brief description
Boolean	Wrapper class for Boolean values 'true' and 'false'
Byte	Wrapper class to provide reference to type byte
Character	Wrapper class to provide reference to type char and it supports two byte Unicode characters
Class	Subclass of Object, used for referring classes as objects
Double	Wrapper class to provide reference to type double
Enum	Defines methods that support enumeration
Float	Wrapper class to provide reference to type float
Integer	Wrapper class to provide reference to type int
Long	Wrapper class to provide reference to type long
Math	Contains several mathematical methods
Number	Abstract super class for the Wrapper classes such as Byte, Short, Integer, Float, and Double
Object	Super class to all other classes in Java
Package	For managing packages
String	Subclass of Object; it defines methods for handling strings.

8. Class Object

The class Object is one of the important classes of Java because it is the super class to all other classes. The class has one constructor and defines 11 methods that are available to all the objects. The methods of class are described in following Table.

Table 10.6 Methods defined in Object class

Method	*Description*
`Object clone()`	Creates a new copy of invoking object
`boolean equals(Object obj)`	Returns true if invoking object and argument object are equal
`void finalize()`	Executed when the object is garbage collected and the method needs to be overridden by a class
`final Class getClass()`	Identifies the class of invoking object
`int hashCode`	Returns hash code of the invoking object
`final void notify()`	Notifies the thread waiting on invoking thread
`final void notify all()`	Notifies all the threads waiting on invoking thread
`String toString()`	Returns string equivalent of invoking object
`final void wait()`	Invoking thread waits till notified and throws `InterruptedException`
`final void wait((long milliseconds))`	Makes the invoking thread wait for a specified time in milliseconds and throws `InterruptedException`

Table 10.6 *(Contd)*

Method	*Description*
`final void wait(long milliseconds, int nanoseconds)`	Makes the invoking thread wait for a specified time in milliseconds and nanoseconds, if the system permits nanoseconds and throws `InterruptedException`

9. Enumeration

- An enum is a special "class" that represents a group of **constants** (unchangeable variables, like final variables).
- To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma. Note that they should be in uppercase letters:

The members of an enum type declaration are as follows:

i. Members declared in the body of the enum type declaration.
ii. Members inherited from the super class Enum<E>.
iii. An enum declaration is implicitly final unless at least one declared constant has a class in its body
iv. Constants declared, which are implicitly public, static, and final.
v. Constants' body may have annotations, fields, expressions, or a class.
vi. Enum keyword has the properties of a class; therefore, it can implement interfaces and its member constants can have inner class.

Example:

```
class Enum
{
  enum Level { LOW, MEDIUM, HIGH}
  public static void main(String args[])
  {
  System.out.println("First Value = " + Level.LOW);
  }
}
```

Output:

```
C:\ >javac Enum.java

C:\ >java Enum
First Value =  LOW
```

10. class Math

The Math class is an important class of `java.lang` package. It defines methods for evaluating several mathematical functions such as trigonometry functions (sine, cosines, etc.), exponential functions, rounding off functions, and other miscellaneous mathematical functions. The class is defined as a final class.

`public final class Math extends Object`

The class inherits methods from object class.

`double E` - Returns value of E (the base of natural logarithm) as double number
`double PI` - Returns the PI value which is 3.14159 , (ratio of circumference to diameter of circle as double number
All the methods defined in Math class are static, and thus, they may be called without object; however, the name of the class is required.
For example, the codes for calling sqrt() and pow() methods are

`Math.sqrt(10);` // returns square root of 10

`Math.pow(2,4);` // returns 2 to the power 4

Example: MathDemo.java

```
class MathDemo
{
     public static void main(String args[])
     {
      System.out.println("Value of E = "+ Math.E);
      System.out.println("Value of PI = "+ Math.PI);
      System.out.println("Squar Root of 25 = "+ Math.sqrt(25));
      System.out.println("2 to the power of 4 = "+ Math.pow(2,4));
       }
}
```

Output:

```
C:\ >javac MathDemo.java

C:\ >java MathDemo

Value of E = 2.718281828459045

Value of PI = 3.141592653589793

Squar Root of 25 = 5.0

2 to the power of 4 = 16.0
```

11. Wrapper Classes

- Wrapper class wraps (i.e., encloses) a primitive data type and provides its object representation.
- In Java, a simple data type can also be converted into an object using Wrapper classes.
- Many data structures in Java are designed to operate on objects.
- Wrapper classes that encapsulate a primitive data type within an object.
- **The eight primitive data types**, namely
    i. boolean,
    ii. byte,
    iii. short,
    iv. int,
    v. long,
    vi. float,
    vii. double, and
    viii. char,
- These 8 primitive data types are not objects of classes; hence, they cannot be passed on by references and they are passed on by value only.
- Therefore, in order to provide object representation, **eight Wrapper classes are defined** in java.lang package which is imported by default in all the Java programs.
- The **eight Wrapper classes are**

i. Boolean
ii. Byte,
iii. Short,
iv. Integer,
v. Long,
vi. Float,
vii. Double
viii. Char

Methods of Wrapper Classes :

i. equals()
ii. isInfinite()
iii. isNaN()
iv. byteValue()
v. compareTo(type Object)
vi. doubleValue()
vii. floatValue()
viii. hashCode()
ix. intValue()
x. longValue()
xi. ShortValue()
xii. toHexString(type number)
xiii. valueOf(type number)
xiv. valueOf(String str)
xv. toString(type number)

Example: WrapperClasses.java

```
class WrapperClasses
{
     public static void main(String args[])
     {
      // for int values
      int i = 10;
      Integer in = i;
      System.out.println("Value of in = "+ in);

      //for float values
      float f = 25.6f;
      Float fn = f;
      System.out.println("Value of fn = "+ fn);

       }
}
```

Output:

```
C:\>javac WrapperClasses.java

C:\ >java WrapperClasses
Value of in = 10
Value of fn = 25.6
```

12. Auto-boxing and Auto- unboxing

Auto-boxing

- Auto-boxing involves automatic conversion of the **primitive data types into its corresponding wrapper types** so that the value may be represented by reference as the objects of other classes.
- This includes the conversion of
    - int to Integer,
    - double to Double,
    - float to Float,
    - boolean to Boolean, etc.,

Auto-unboxing

- Auto - unboxing is the reverse process in which the **wrapping class objects are converted into the corresponding primitive data types**.
- This includes conversion of
    - Integer to int,
    - Long to long,
    - Double to double, etc.

Example: AutoBoxing.java

```
class AutoBoxing
{
     public static void main(String args[])
     {
      // Auto Boxing
      int i = 10;
      Integer in = new Integer(i); //Auto Boxing: Primitive to Wrapper
      System.out.println("Value of Wrapper variable in = "+ in);

      //Auto Unboxing
      Integer x = 20;
      int y = x.intValue(); //Auto Unboxing : Wrapper to Primitive
      System.out.println("Value of primitive variable y = "+ y);

       }
}
```

Output:

```
C:\>javac AutoBoxing.java

C:\>java AutoBoxing
Value of Wrapper variable in = 10
Value of primitive variable y = 20
```

13. java.util Classes and Interfaces

java.util is an important package of the Java library. It contains classes and interfaces of collections, event model, and miscellaneous utility classes. The classes and interfaces contained in this package are very important as these are required for many programs. For instance, it comprises Math class that supports Mathematical expressions.

Classes in java.util package

AbstractCollection	AbstractList	AbstractMap	AbstractQueue
AbsractSequentiallist	AbstractSet	ArrayList	Arrays
BitSet	Calendar	Collections	Currency
Date	Dictionary	EnumMap	EnumSet
EventListenerProxy	EventObject	Formatter	HashMap
HashSet	Hashtable	IdentityHashMap	LinkedHashSet
LinkedList	ListSources Bundle	Locale	Observable
PriorityQueue	Properties	PropertyPermission	Property
ResourceBundle	Random	ResourceBundle	Scanner
ServiceLoader	SimpleTimeZone	Stack	StringTokanizer
Timer	TimeTask	TimeZone	TreeMap
TreeSet	UUID	Vector	WeakHashMap

(Write any ten class names in the Examination)

Interfaces in java.util package

Collection	Comparator	Deque	Enumeration
EventListener	Formattable	Iterator	List
ListIterator	Map	Map. Entry	NavigableMap
NavigableSet	Observer	Queue	RandomAccess
Set	SortedMap	SortedSet	

(Write any ten interface names in the Examination)

Scanner Class

Java Scanner class is a text scanner that breaks the input into tokens using delimiter, which is whitespace by default. Delimiter is a character that identifies the beginning or end of character string and it is not a part of the character string.

The received string can then be converted into values of different types using the various next() methods. The application of this class is not thread safe. Java Scanner class extends Object class and implements Iterator and Closeable interfaces. For instance, the following code allows the user to input into the program an integer number through the keyboard.

Sample Code:

```
Scanner sc = new Scanner(System.in);
System.out.println("Enter your roll no.");
int rollno = sc.nextInt();
```

Example : UserInput.java

```
import java.util.Scanner;
public class UserInput
{
      public static void main(String[] args)
      {
      Scanner scaninput = new Scanner (System. in);
      int n;
      int m;
      System.out. print( "Enter the value of n : ");
      n=scaninput.nextInt();
      System.out. print( "Enter the value of m : ");
      m=scaninput.nextInt();

      System.out.println("Sum of two numbers is ="+(n+m));

      }
}
```

Output

```
   C:\>javac UserInput.java

   C:\>java Arithmetic
   Enter the value of n : 10
   Enter the value of m : 3
   Sum of two numbers is =13
```

Radix in java.util package

Radix is another name of base of the number system being referred to.
Radix values specifies the base value of the numbers ie 2 for Binary Numbers, 8 for Octal Numbers, 10 for Decimal Numbers (The default radix number is 10) and 16 for HexaDecimal Numbers

- **`radix()`** method prints the current radix value
- **`useRadix()`** method used to change the radix value

Example : TestRadix.java

```
import java.util.Scanner;
public class TestRadix
{
   public static void main(String[] args)
   {
      int n;

      //for Decimal Numbers
Scanner input = new Scanner(System.in);
System.out.println("For Decimal numbers- radix = "+ input.radix());
System.out.print("Enter an integer number: ");
n = input.nextInt();
System.out.println("The number n = " + n);

      //for HexaDecimal Numbers
input.useRadix(16);
System.out.println("\nFor HexaDecimal numbers radix = "+ input.radix());
System.out.print("Enter a number with base 16: ");
n = input.nextInt ();
System.out.println("The number n= " + n);

      //for Octal Numbers
input.useRadix(8);
System.out.println("\nFor Octal numbers radix = "+ input.radix());
System.out.print("Enter a number with base 8 : ");
n = input.nextInt();
System.out.println("The number n = " + n);

//for Binary Numbers
input.useRadix(2);
System.out.println("\nFor Binar numbers radix = "+ input.radix());
System.out.print("Enter a number with base 2 : ");
n = input.nextInt();
System.out.println("The number n = " + n);

   }
}
```

Output:

```
E:\>javac TestRadix.java
E:\ >java TestRadix
For Decimal numbers- radix = 10
Enter an integer number: 56
The number n = 56

For HexaDecimal numbers radix = 16
Enter a number with base 16: 55
The number n= 85

For Octal numbers radix = 8
Enter a number with base 8 : 41
The number n = 33

For Binar numbers radix = 2
Enter a number with base 2 : 11110
The number n = 30
```

14. Formatter Class

- The class Formatter supports the formatting of the output of characters, strings, and numeric types.
- The declaration of this class is highly inspired by the printf method of C programming language. It is the printf method of C in Java clothing;
- In Java, printf() method returns an object of print stream.
- Class Formatter provides support for layout justification, alignment, formats for numeric, string, and date/time data.
- The class declaration is

```
public final class Formatter
                extends Object
                implements Closeable, Flushable
```

- The format string for output of numeric values comprises a % sign followed by a conversion character.
- The field width and precision are specified between the % sign and the conversion character.

Table 10.17 Conversion characters

Conversion character			Description
Lower case/Upper case		Category	
b	or B	General	Boolean output—true/false in lower case or upper case
c	or C	Character	Represent in Unicode character
h	or H	General	Represent as hexadecimal number
s	or S	General	Represent as string
d		Integral	Output formatted as decimal integer used for byte, short, int, and long
o		Integral	Represent as octal number
x	or X	Integral	Represent as hexadecimal number in lower or upper case
e	or E	Floating point	For representation in scientific notation
f		Floating point	For floating point numbers—float or double
G	or G	Floating point	Normal or exponential representation
n		Separator	Result on right side of n is shifted to next line
%		Percent	Results in % sign
t	or T	Date and time	Conversion character for time and date

Example : FormatterDemo.java

```
import java.util. Formatter;
public class FormatterDemo
{
      public static void main(String[] args)
      {
      int x = 165;
      float f = 432.14159f;
      double d = 5654.87543;
      char ch = 'a';
      String str = "Hello World";

      //Declaration of two objects formt and fmt
      Formatter formt = new Formatter(), fmt= new Formatter();

      //Formatting and printing formt object
      formt.format("x = %d, f= %f, d= %f, ch= %c and str= %s", x, f,
d, ch, str);
      System.out.println(formt);

      //Formatting and printing fmt object
      fmt.format("Upper case of ch = %C and str = %S ", ch, str);
      System.out.println(fmt);
      }
}
```

Output:

```
C:\>javac FormatterDemo.java

C:\>java FormatterDemo
x = 165, f= 432.141602, d= 5654.875430, ch= a and str= Hello World
Upper case of ch = A and str = HELLO WORLD
```

15. Random Class

The class Random of java.util package is generally used for generating random numbers of different types such as int, double, float, long, and byte. The method random() of class Math may also be used for generating double numbers.

Example: Random.java

```
import java.util.Random;
public class RandomNumbers
{
     public static void main(String[] args)
     {
     Random rand = new Random();

     // Random Integer values
     System.out.print("The random numbers are: ");
     for (int i = 0; i<10; i++)
         System.out.print((1 + rand.nextInt(6))+ " ");

     // Random Double values
     System.out.print("\nTHe double random numbers are: ");
     for (int i = 0; i<5; i++)
        System.out.printf(" %.3f", (rand.nextDouble()));


     // Random Boolean values
     System.out.print("\nTHe boolean random values are: ");
     for (int i = 0; i<5; i++)
       System.out.print((rand.nextBoolean())+ " ");
     }
}

Output:
C:\ >javac RandomNumbers.java

C:\ >java RandomNumbers
The random numbers are: 5 1 1 5 2 6 5 2 5 2
THe double random numbers are:  0.289 0.543 0.337 0.243 0.250
THe boolean random values are: false true false false false
```

16. Time Package

- This package provides support for date and time-related features for program developers.
- It comprises classes that represent date/time concepts including instants, duration, date, time, time zones, and periods.
- The date and time framework in Java prior to Java SE 8 consisted of classes such as
    - `java.util.Date,`
    - `java.util.Calendar`, and
    - `java.util.SimpleDateFormatter`

  which posed problems to programmers because of the following reasons:
  These classes are
    i. not thread safe
    ii. present a poor show.
    iii. poor design.
    iv. it was difficult to work with different calendar systems followed in different parts of the world.

- The result is the development of an entirely new package `java.time` that has classes that are thread safe; these classes have robust design and they can be adapted to work with different calendar systems.

Example: TimePrg.java

```
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.LocalDateTime;
import java.time.Period;

// declaring the class.
class TimePrg
{
  public static void main(String[] args)
  {
  //creating an object
  LocalTime time = LocalTime.now();
  System.out.println("The time at present is :" + time);

  // creating another object
  LocalTime newTime;
  newTime = time.plusHours (2);
  System.out.println("The modified time:"+ newTime);

  // creating an object
  LocalDate date = LocalDate.now();
  System.out.println("The date today is : "+date);

  LocalDateTime datetime = LocalDateTime.now();
  System.out.println("The date and time at present are:
"+datetime);
```

```
  //1 year, 3 months and 10 days
  Period p = Period.of(1,3,10);

  LocalDateTime newdate = datetime.plus(p);
  System.out.println("The newdate is:" + newdate);
  }
}
```

Output:

```
C:\ >javac TimePrg.java

C:\ >java TimePrg
The time at present is :07:24:57.342943800
The modified time:09:24:57.342943800
The date today is : 2021-06-12
The date and time at present are: 2021-06-12T07:24:57.354018300
The newdate is:2022-09-22T07:24:57.354018300
```

17. Class Instant (java.time.Instant)

Java Instant class is used to represent the specific moment on the time line. It inherits the Object class and implements the Comparable interface.

Table 10.23 Methods of class Instant

Method	*Description*
isAfter	Compares two time instants `Instant.now() isAfter Instant.now().minusHours(1);`
isBefore	Compares two time instants `Instant.now() is before Instant.now().plusHours(1);`
plus	Adds time to instant. An example of this code is `Instant later = Instant.now().plusMinutes(30);`
minus	Subtracts time from Instant. An example of this code is `Instant before = Instant.now().minusHours(2);`
until	Returns how much time exists between two time Instant objects

Example: TimeInstantDemo.java

```
import java.time.Instant;
public class TimeInstantDemo
{
  public static void main(String[] args)
  {
    Instant now = Instant.now();
    System.out.println("Now Time = "+ now);

    Instant before = Instant.now().minusSeconds(600);
    System.out.println("before Time = "+ before);

    Instant later = Instant.now().plusSeconds(900);
    System.out.println("later Time = "+ later);

  }
}
```

Output:

```
C:\ >javac TimeInstantDemo.java

C:\ >java TimeInstantDemo
Now Time = 2021-06-12T02:25:19.772322600Z
before Time = 2021-06-12T02:15:19.800125300Z
later Time = 2021-06-12T02:40:19.804155800Z
```

Example2: UntilDemo.java

```
import java.time.*;
import java.time.temporal.*;

public class UntilDemo
{
     public static void main(String[] args)
     {
      long td;
           LocalTime time = LocalTime.parse("10:15:30");
           LocalTime time1 = LocalTime.now();

     //Calculating time difference (in hours) thorugh until()
method
     td = time1.until(time, ChronoUnit.HOURS);
           System.out.println("Time  difference  =  "+  td  +  "
Hours");
```

```
      //Calculating time difference (in minutes)thorugh until()
method
      td = time1.until(time, ChronoUnit.MINUTES);
            System.out.println("Time  difference  =  "+  td  +  "
Minutes");

      //Calculating time difference (in seconds)thorugh until()
method
      td = time1.until(time, ChronoUnit.SECONDS);
            System.out.println("Time  difference  =  "+  td  +  "
Seconds");
   }
}
```

```
Output:
C:\>javac UntilDemo.java

C:\>java UntilDemo
Time difference = 2 Hours
Time difference = 169 Minutes
Time difference = 10188 Seconds
```

18.Formatting for Date/Time in Java

The old API for formatting has been revised in Java 8 to a new class for formatting, printing, and parsing **Date/Time** objects through DateTimeFormatter class.

DateTimeFormatter class declaration is as follows.

```
   public final class DateTimeFormatter extends object
```

The class provides the following three ways to achieve the desired results.

1. By using the predefined constants in the language that represent a particular format of formatting date/ time objects.
2. By using patterns defined by the programmer.
3. By using localized format.

Example: DateTimeFormatter.java

```
import java.time.format.DateTimeFormatter;
import java.time.LocalDate;
import java.time.DateTimeException;
import java.time.LocalDateTime;

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

```
      //defining an instance
      LocalDateTime datetime = LocalDateTime.now();
      System.out.println("Default format of LocalDateTime: "+ datetime);
      System.out.println("\nIn 150_LOCAL_DATE_TIME Format: " +
datetime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

      LocalDate date= LocalDate.now();
      System.out.println("\nDefault format of LocalDate= " + date);


      System.out.println("\nIn ISO LOCAL DATE Format: " +
date.format(DateTimeFormatter.ISO_LOCAL_DATE));
      System.out.println("\nUser format:" + datetime. format
(DateTimeFormatter.ofPattern("dd-MMM-yyyy hh:mm:ss")));
      }
}
```

Output:

```
C:\>javac DateTimeFormatDemo.java

C:\>java DateTimeFormatDemo
Default format of LocalDateTime: 2021-06-12T19:24:32.807557700

In 150_LOCAL_DATE_TIME Format: 2021-06-12T19:24:32.8075577

Default format of LocalDate= 2021-06-12

In ISO LOCAL DATE Format: 2021-06-12

User format:12-Jun-2021 07:24:32
```

19. Temporal Adjusters Class

Temporal Adjuster Interface

The java time temporal defines an interface by name TemporalAdjuster that defines methods that take a temporal value and return an adjusted temporal value according to a user's specifications.

Temporal Adjusters class

The package also defines a class TemporalAdjusters, which contains predefined used by a programmer. Some examples of adjusters are as follows.

i. firstDayOfMonth()
ii. lastDayOfMonth()
iii. firstDayOfyear()
iv. lastDayOfYear
v. firstDayOfWeek ()

vi. lastbayOfWeek()
vii. firstInMonth (DayOfWeek. Monday)
viii. lastInMonth (DayOfWeek. FRIDAY)

All these methods are static methods and may be used with static import.
Static import is used to access any static member of a class directly without referring its class name. There are two general forms of declaration of static import statements.

The first form of static import statement involves importing only a single member of a class as shown:

```
import static package.class-name.static-member-name:
```

As for instance,

```
import static java.lang.Math.sqrt;
```

The second form of static import statement involves importing all the static members of a class as

```
import static package.class-type-name.*;
```

This is shown in the following example:

```
import static java.lang.Math.;
```

It is different from import statement that allows the programmer to access classes of a package without providing package name or qualification. It is to be noted that import statement makes the classes and interfaces accessible, whereas the static import provides accessibility to static members of the class only.

Example: TemporalAdjusterDemo.java

```
import java.time.*;
import java.time.temporal.TemporalAdjusters;

public class TemporalAdjusterDemo
{
      public static void main(String[] args)
      {
      // defining an object

      LocalDate date1= LocalDate.of(2015, Month.NOVEMBER, 20);
      DayOfWeek dow1 = date1.getDayOfWeek();

      System.out.println("The day of week on 20th Nov 2015 =" +
dow1);

      System.out.println("The first day of November = "+
date1.with(TemporalAdjusters.firstDayOfMonth()));
```

```
     System.out.println("The last day of November = "+
date1.with(TemporalAdjusters.lastDayOfMonth()));
     }
}
```

Output

```
C:\>javac TemporalAdjusterDemo.java


C:\>java TemporalAdjusterDemo
The day of week on 20th Nov 2015 =FRIDAY
The first day of November = 2015-11-01
The last day of November = 2015-11-30
```

II. Exception Handling:

1. Introduction

- In Java, an exception is an *object* of a relevant exception class. **When an error occurs in a method, it throws out an exception object** that contains the information about where the error occurred and the type of error. **The error (exception)** object is passed on to the runtime system, which searches for the appropriate code that can handle the exception.
- The event handling code is called ***exception handler*.**
- **Exception handling** is a mechanism that is used to handle runtime errors such as classNotFound and IO. This ensures that the normal flow of application is not disrupted and program execution proceeds smoothly.
- **The exception handling code** handles only the type of exception that is specified for it to handle. **The appropriate code** is the one whose specified type matches the type of exception thrown by the method.
- If the runtime system finds such a handler, it passes on the exception object to the handler. **If an appropriate handler is not found, the program terminates**.
- The method that creates an exception may itself provide a code to deal with it.
- **The try and catch blocks are used to deal with exceptions.**
- The code that is likely to create an exception is kept in the try block, which may include statements that may create or throw exceptions.
- The exception object has a data member that keeps information about **the *type* of exception** and it becomes an argument for another block of code that is meant to deal with the exception.
- **There may be more than one catch blocks.**
- There are basically two models of exception handling.

1. Termination Model

- According to **termination model**, when the method encounters an exception, further processing in that method is terminated and control is transferred from the point where an exception occurs to the point where its nearest matching exception handler (i.e., the catch block) is located.
- This model is analogous to the real-life situation when the gas in the cylinder gets over while cooking.
- Under the termination model, the cooking process will stop, whereas in resumption model, you would change the cylinder with a new one and continue with the cooking process.

2. Resumption model

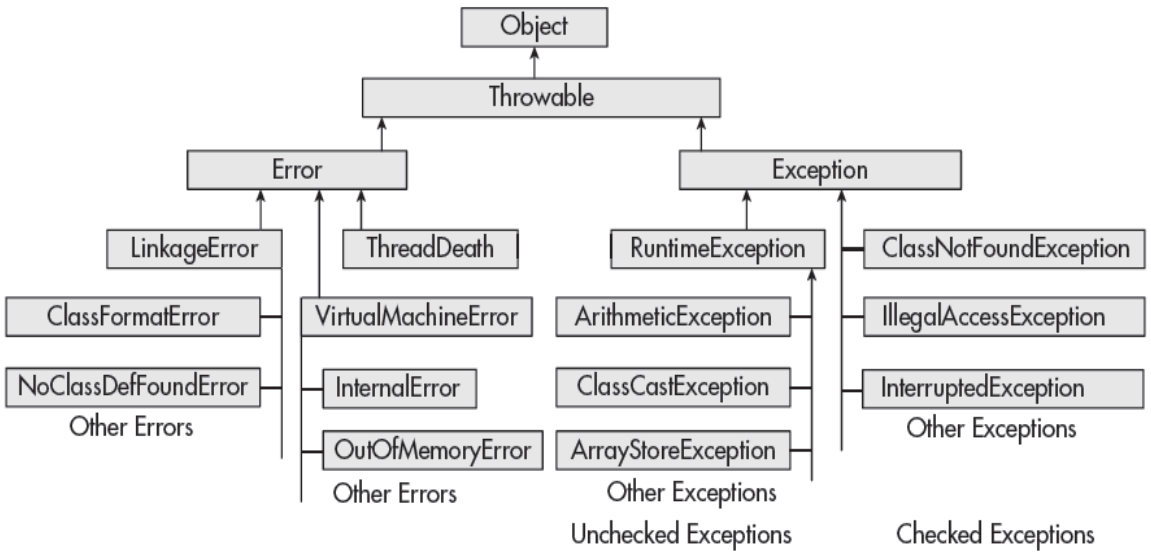
- Thus, the alternative approach is based on the **resumption model** wherein the exception handler tries to rectify the exception situation and resume the program.
- Resumption model was mostly used in earlier languages including PL/I, Mesa, and BETA.
- The implementation of resumption model in contemporary languages such as C++ and

Java is rarely found as the effort to implement this model is quite high.

- This is because the program code becomes quite cumbersome and difficult to understand, and further, it is more error prone.
- However, there are situations where the resumption model is quite useful.

2. Hierarchy of Standard Exception Classes

- In Java, exceptions are instances of classes derived from the class Throwable which in turn is derived from class Object.
- Whenever an exception is thrown, it implies that an object is thrown.
- Only objects belonging to class that is derived from class Throwable can be thrown as exceptions.
- The next level of derived classes comprises two classes: the class Error and class Exception.
- Error class involves errors that are mainly caused by the environment in which an application is running.
- All errors in Java happen during runtime.



- Examples
  OutOfMemoryError occurs when the **JVM runs out of memory** and **StackOverflowError** occurs when the **stack overflows**.
- Exception class represents exceptions that are mainly caused by the application itself.
- **It is not possible to recover from an error using try–catch blocks**.
- **The only option available is to terminate the execution** of the program and **recover from exceptions using either the try–catch block or throwing an exception.**
  - The exception class has several subclasses that deal with the exceptions that are caught and dealt with by the user's program.
  - The Error class includes such errors over which a programmer has less control.

- A programmer cannot do anything about these errors except to get the error message and check the program code.
- A programmer can have control over the exceptions (errors) defined by several subclasses of class Exception.

The subclasses of Exception class are broadly subdivided into two categories.
Unchecked exceptions These are subclasses of class **RuntimeException**, derived from Exception class. For these exceptions, **the compiler does not check** whether the method that throws these exceptions has provided any exception handler code or not.

Checked exceptions These are direct subclasses of the **Exception** class and are not subclasses of the class RuntimeException. These are called so because **the compiler ensures (checks)** that the methods that throw checked exceptions deal with them.

This can be done in two ways:
1. The method provides the exception handler in the form of appropriate try–catch blocks.
2. The method may simply declare the list of exceptions that the method may throw with throws clause in the header of the method.
   - In this case, the method need not provide any exception handler.
   - It is a reminder for those who use the method to provide the appropriate exception handler.
   - If a method does not follow either of the aforementioned ways, the compilation will result in an error.

Methods defined in class Throwable

Table 11.1 Methods defined in class `Throwable`

Method name	*Description*
`getMessage ()`	It returns a string that gives information about the current exception and consists of a fully qualified name of the exception class and a relevant brief description.
`toString()`	The `class Throwable` overrides the method `toString()` of Object class for displaying messages on screen.
`printStackTrace()`	It traces and displays the hierarchy of method calls that resulted in the exception. The information will be displayed on the screen in the case of a console program.

Example: TryCatchDemo.java

```
class TryCatchDemo
{
   public static void main(String args[])
  {
  int i=6,j=0,k;
  try {
        System.out.println("Entered try block");
        k=i/j;
        System.out.println("Exiting try block");
  }
   catch (ArithmeticException e)
   {
        System.out.println("e = " + e);
   }
   System.out.println("End of the Program ");
   }
}
```

Output:

```
C:\ >javac TryCatchDemo.java

C:\ >java TryCatchDemo
Entered try block
e = java.lang.ArithmeticException: / by zero
End of the Program
```

3. Keywords throws and throw

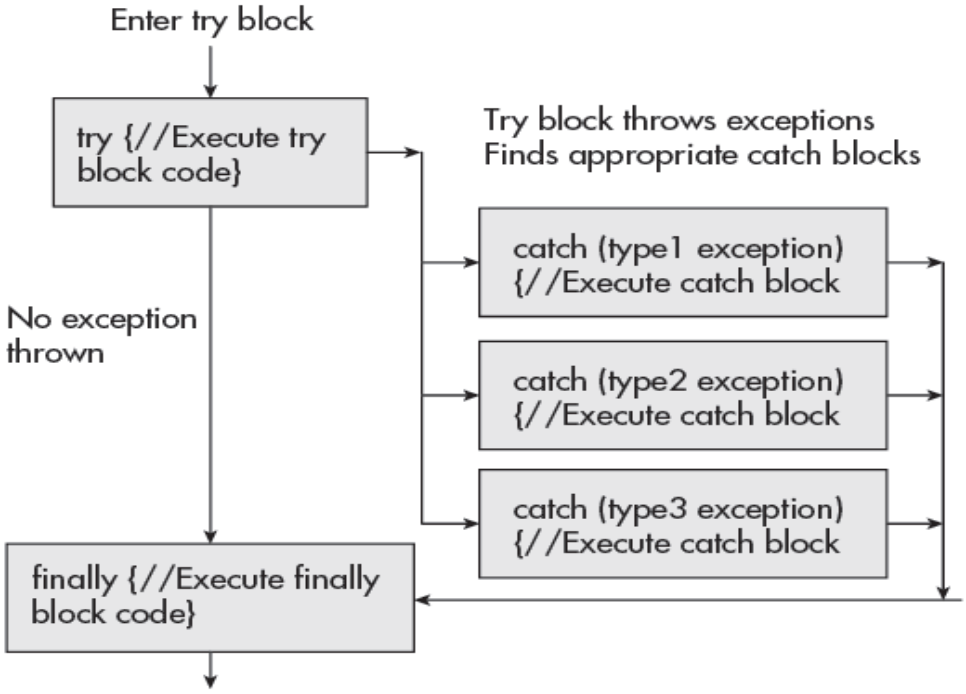
- If a method can cause one or more checked exceptions directly or indirectly by calling other methods and throw exceptions and does not deal with them, it must declare the list of exceptions it can throw by using the *throws* clause.
- By doing this, the caller of the method can ensure that appropriate arrangements are made to deal with the exceptions.
- For this, keep the method in a try block, and provide suitable catch blocks.
- The program will not compile if this is not done.
- **The throws clause comprises the keyword throws followed by the list of exceptions that the method is likely to throw separated by commas.**
- The method declaration with throws clause is

```
type Method_Name(type parameters)  throws  list-of-Exceptions
{
  /* Body of Method*/
 }
```

4. try catch and finally Blocks

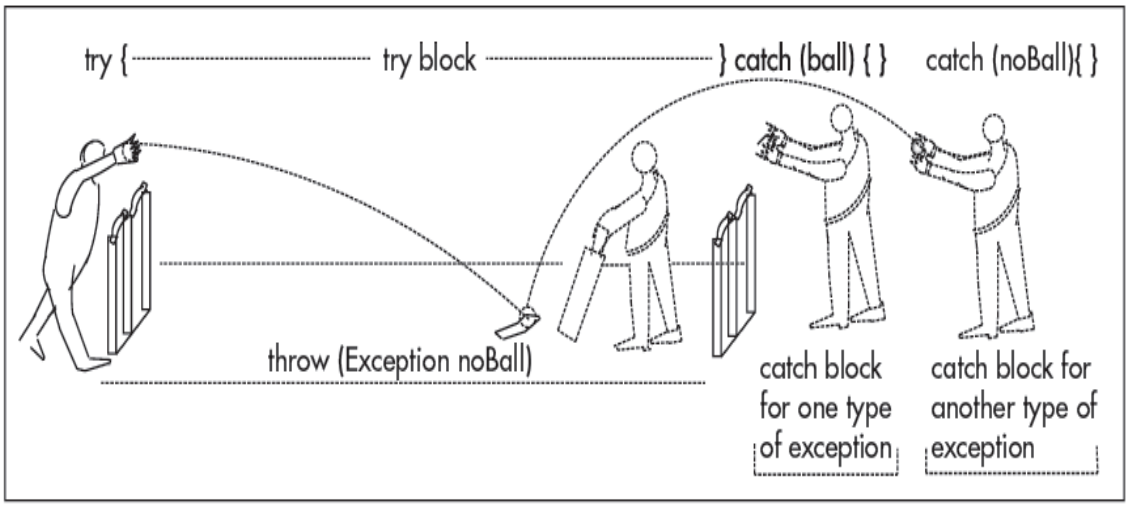
try {} Block

- The program code that is most likely to create exceptions is kept in the try block, which is followed by the catch block to handle the exception.
- In normal execution, the statements are executed and if there are no exceptions, the program flow goes to the code line after the catch blocks.
- However, if there is an exception, an exception object is thrown from the try block.
- Its data members keep the information about the *type* of exception thrown.
- The program flow comes out of the try block and searches for an appropriate catch block with the same *type* as its argument.



catch {} Block

- A catch block is meant to catch the exception if the *type* of its argument matches with the *type* of exception thrown.
- If the *type* of exception does not match the *type* of the first catch block, the program flow checks the other catch blocks one by one.
- If the *type* of a catch block matches, its statements are executed.
- If none matches, the program flow records the *type* of exception, executes the finally block, and terminates the program.



Example: TryCatchDemo2.java

```
class TryCatchDemo2
{
   public static void main(String args[])
   {
   int i=6,j=0,k;
   try {
         System.out.println("Entered try block");
         k=i/j;
         System.out.println("Exiting try block");
   }
   catch (ArithmeticException e)
   {
       System.out.println("e = " + e);
   }

   j=2; // Value of j Changed
   System.out.println("When J =2, i/j = " + i/j);
   System.out.println("End of the Program ");
    }
}
```

Output:

```
C:\ >javac TryCatchDemo2.java

C:\ >java TryCatchDemo2
Entered try block
e = java.lang.ArithmeticException: / by zero
When J =2, i/j = 3
End of the Program
```

finally {} Block

- This is the block of statements that is always executed even when there is an exceptional condition, which may or may not have been caught or dealt with.
- Thus, finally block can be used as a tool for the clean up operations and for recovering the memory resources.
- For this, the resources should be closed in the finally block.
- This will also guard against situations when the closing operations are bypassed by statements such as continue, break, or return.
-

Finalize() in Exception Handling

- finalize() method releases system resources before the garbage collector runs for a specific object.
- JVM allows finalize() to be invoked only once per object
- **The syntax for using finalize() method is given as follows:**

```
@Override //
protected void finalize() throws Throwable
{
   try{
   …….
   } catch(Throwable t)
     {
        throw t;
    }finally{
     super.finalize();
    }
}
```

- Following points need to be taken into consideration when using finalize() method:

1. **super.finalize()** method is always called in finalize() method.
2. Time critical application **logic should not be placed in finalize()** method.

5. Multiple Catch Clauses

- Java allows to write multiple catch block in the program.
- Often the programs throw more than one *type* of exception, at that time more catch block need to be used.
- The exception classes are connected by Boolean operator OR.

Example: MultiCatchExample.java

```
class MultiCatchExample
{
   public static void main(String args[])
   {
  try {
       int a=0, b=56, c;
```

```
            int aray[] = {4,5,6,7};
            System.out.println("The Array elements are : ");

            for(int i=0; i<10;i++)
             System.out.println(aray[i] + " ");
            System.out.println("a = " + a + "  b = " + b);
            c=b/a;
        } catch (ArithmeticException e)
        {
         System.out.println("\nInteger Division by 0, e = " + e);
        } catch (ArrayIndexOutOfBoundsException ae)
        {
System.out.println("\nArray Index Out of Bounds, ae = " + ae);
        }
           System.out.println("End of the Program ");
         }
    }
```

Output:

```
C:\ >javac MultiCatchExample.java

C:\>java MultiCatchExample
The Array elements are :
4
5
6
7

Array Index Out of Bounds, ae =
java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds
for length 4
End of the Program
```

6. Class Throwable

- The class is declared as

```
public class Throwable extends Object
        implements serializable
```

The class Throwable is super class to all the error and exception classes.

- The program code can throw only objects of this class or objects of its subclasses.
- Only the object of Throwable class or its subclasses can be the arguments of catch clause.
- It would be a bad programming practice to throw Throwable or make Throwable as argument of catch blocks because it will hide desirable information.

Table 11.2 Constructors of class Throwable

Constructor	Description
Throwable ()	Constructs a new throwable with no message
Throwable(String message)	Constructs a throwable with specified message
Throwable(String message, Throwable cause)	Constructs a throwable with specified method and cause
Throwable(Throwable cause)	Constructs a throwable with specified cause
protected Throwable(String message, Throwable cause, Boolean enablesuppression)	Constructs a throwable with specified message, cause, enabled/disabled suppression, and enabled/disabled stack trace

Methods in Throwable Class:

void addSuppressed (Throwable exception)

Throwable fill InStackTrace()

Throwable getCause()

String getMessage ()

String getLocalizedMessage()

StackTraceElement[] getStackTrace()

Throwable [] get Suppressed()

Throwable initCause (Throwable cause)

void printStackTrace()

void printStackTrace(printStream strm)

void printStackTrace(printwriter strm)

void setStackTrace(

StackTraceElement[] stackTrace)

String toString()

Example: TryCatchDemo3.java

```
class TryCatchDemo3 extends Throwable
{
   public static void main(String args[])
   {
      int i=6, j=0, k;
      String str;
      TryCatchDemo3 ct = new TryCatchDemo3();
      try {
           System.out.println("Entered in try block");
           k=i/j;
              System.out.println("Entered in try block");
      } catch (ArithmeticException e)
      {
      System.out.println("Entered in catch block");
      System.out.println("ct.getStackTrace() = " +
ct.getStackTrace());
      System.out.println("Exception e = " + e);

      } finally{
      System.out.println("It is finally block");
      System.out.println("The class is : " + ct.getClass());
      }

     System.out.println("End of the Program ");
   }
}
```

Output:

```
C:\1. JAVA\UNIT-4.2>javac TryCatchDemo3.java

C:\1. JAVA\UNIT-4.2>java TryCatchDemo3
Entered in try block
Entered in catch block
ct.getStackTrace() = [Ljava.lang.StackTraceElement;@452b3a41
Exception e = java.lang.ArithmeticException: / by zero
It is finally block
The class is : class TryCatchDemo3
End of the Program
```

7. Unchecked Exceptions

- The unchecked exception classes are subclasses of class
- RuntimeException, which is further a subclass of class Exception.
- The runtime exceptions are not checked by the compiler.
- These exceptions form a subgroup of classes that are subclasses of class Exception.
- It is not checked by the compiler whether the programmer has handled them or not.

Table 11.4 Unchecked subclasses of RuntimeException

Name of class	*Description of exception condition*
`ArithmeticException`	Exceptional arithmetic condition such as division by zero
`ArrayIndexOutOfBoundsException`	Exception arises when out of bound index number is used
`ArrayStoreException`	Exception arises when we try to use an object of wrong *type* in an array
`ClassCastException`	Exception condition arising out of invalid *type* object
`IllegalArgumentException`	Exception arises due to the use of an invalid argument in a method like placing a negative number as argument of sqrt()
`IllegalStateException`	Exception due to incorrect state of an application
`IllegalThreadStateException`	Exception arises when the operation attempted is not compatible with the thread state
`IndexOutOfBoundsException`	When index number used for array, vector, or string element is out of bounds
`NegativeArraySizeException`	Exception due to the use of negative number for size of array
`NumberFormatException`	Exception due to invalid conversion of string into number
`NullPointException`	Exception due to the use of object with null reference
`SecurityException`	When the code does an illegal operation that violates security
`TypeNotPresentException`	When the specified *type* is not found
`UnsupportedOperationException`	Exception due to unsupported operation

Example: UnCheckedDemo.java

```
class UnCheckedDemo
{
   public static void main(String args[])
   {
      int a =12,b = 0, c = 14;
      int sum, d;
      sum = a+b+c;
      System.out.println("Sum = " +sum);

      try {
            System.out.println("Entered try block");
            d=a/b;
            System.out.println("d = " + d);
            System.out.println("Exiting try block");
      }
      catch (ArithmeticException e)
      {
       System.out.println("Do not divide by Zero, Exception =" + e);
```

```
      }
      System.out.println("End of the Program ");
   }
}
```

Output:

```
C:\>javac UnCheckedDemo.java

C:\>java UnCheckedDemo
Sum = 26
Entered try block
Do not divide by Zero, Exception =java.lang.ArithmeticException: /
by zero

End of the Program
```

8. Checked Exceptions

Checked exceptions are direct subclasses of the **Exception** class and are not subclasses of the class RuntimeException. These are called so because **the compiler ensures (checks)** that the methods that throw checked exceptions deal with them

Table 11.5 Checked Exceptions

Exception	*Description*
ClassNotFoundException	Exception because class is not found
FileNotFoundException	Exception if the file does not exist
CloneNotSupportedException	If an object does not implement Cloneable interface, an attempt to clone will result in this exception
IllegalAccessException	Attempt to access a class not permitted access
InstantiationException	Attempt to create objects of an abstract class or of an interface will result in this exception
InterruptedException	When one thread is interrupted by another thread
NoSuchFieldException	Request for a nonexistent field
NoSuchMethodException	Request for a nonexistent method

Example: MyThread.java

```
        class MyThread
        {
           public static void main(String args[])
           {
            MyThread obj = new MyThread();
            obj.method2();
            System.out.println("End of the Program ");
           }

           public static void method2()
           {
            Thread t1 = new Thread();
            Thread t2 = new Thread();
            try{
            System.out.println("Entered try block");

            t2.sleep(200);
            System.out.println("Square   root   of   64   =   "
        +Math.sqrt(64));
            t1.sleep(1000);
            t2.notify();

            System.out.println("Exiting try block");
            }catch (InterruptedException ie)
            {
                System.out.println("Exception =" + ie);
            }
           }
}
```

Output:

```
C:\ >javac MyThread.java

C:\ >java MyThread
Entered try block
Square root of 64 = 8.0
Exception in thread "main" java.lang.IllegalMonitorStateException
        at java.lang.Object.notify(Native Method)
        at MyThread.method2(MyThread.java:20)
        at MyThread.main(MyThread.java:6)
```

9. try-with-resources

- Generally, the resources such as file, data bases, or Internet connections that are opened in try block are closed in finally block, which is always executed.
- This makes a sure-shot method for recovery of sources and memory.
- It is seen that programmers who otherwise are adept at opening sources such as data bases and files often forget to close these sources when their relevance is over.
- These heavyweight sources keep lurking in JVM, and thus, overloading JVM and creating problems in memory allocation.
- In order to redress the problem, Java 7 has added AutoCloseable.

Example: AutoCloseExample.java

```
class SourceX implements AutoCloseable
{
   @Override
   public void close() throws Exception
   {
      System.out.println("The SourceX is closed");
   }
}

public class AutoCloseExample
{
   public static void main(String args[])
   {
      try  {
           System.out.println("Entered in Try Block");
           SourceX objS = new SourceX();
           objS.close();
           System.out.println("Exit from Try Block");
      } catch (Exception ex)
      {
          System.out.println("Exception ex = " + ex);
      }

     System.out.println("End of the Program ");
   }
}

Output:
C:\>javac AutoCloseExample.java

C:\>java AutoCloseExample
Entered in Try Block
The SourceX is closed
Exit from Try Block
End of the Program
```

10. Catching Subclass Exception

If the super class method does not declare any exception, then subclass overridden method cannot declare (or throws) checked exceptions but it can declare (or throws) unchecked exceptions. This is illustrated in the following Programs.

Example1 : SubClass.java - Sub Class cannot throws **Checked** Exception

```
import java.io.*;

class A
{
   void display()
   {
   System.out.println("Super class display method");
   }
}

public class SubClass extends A
{
   //sub class cannot throws Checked Exception
   void display() throws IOException
   {
   System.out.println("Sub class display method");
   }

   public static void main(String[] args)
   {
   A obj1 = new A();
   obj1.display();

   SubClass obj2 = new SubClass();
   obj2.display();
   }
}
```

Output:

```
C:\>javac SubClass.java
SubClass.java:13: error: display() in SubClass cannot override
display() in A
        void display() throws IOException
             ^
  overridden method does not throw IOException
1 error
```

Example2 : SubClass.java - Sub Class can throws **Unchecked** Exception

```
import java.io.*;

class A
{
      void display()
      {
      System.out.println("Super class display method");
      }
}

public class SubClass2 extends A
{
      //sub class can throws UnChecked Exception
      void display() throws ArithmeticException
      {
      System.out.println("Sub class display method");
      }

      public static void main(String[] args)
      {
      A obj1 = new A();
      obj1.display();

      SubClass2 obj2 = new SubClass2();
      obj2.display();
      }
}
```

Output:

```
C:\>javac SubClass2.java

C:\>java SubClass2
Super class display method
Sub class display method
```

11. Custom Exceptions

- It is also called as user defined exception.
- A programmer may create his/her own exception class by extending the exception class and can customize the exception according to his/her needs.
- Using Java custom exception, the programmer can write their own exceptions and messages.

Example: ExceptionEx.java

```
class UserException extends Exception
{
      UserException (String Message)
      {
       super(Message);
      }
}

class ExceptionEx
{
  public static void main(String args[])
  {
   byte a = 4, b = 9;
   try
   {
   if(a/b== 0)
   throw new UserException("It is integer division.");
   }catch (UserException Ue)
   {
    System.out.println("The exception has been caught.");
    System.out.println(Ue.getMessage());
   }
 }
}
Output:

C:\>javac ExceptionEx.java

C:\>java ExceptionEx
The exception has been caught.
It is integer division.
```

12. Nested try and catch Blocks

- In nested try–catch blocks, one try–catch block can be placed within another try's body.
- Nested try block is used in cases where a part of block may cause one error and the entire block may cause another error.
- In such cases, exception handlers are nested.
- If a try block does not have a catch handler for a particular exception, the next try block's catch handlers are inspected for a match.
- If no catch block matches, then the Java runtime system handles the exception.

```
try{// main try block
    //statements
    try {   // try block 1
    //statements
    try {   // try block 2
    /* statement */}
    catch(Exception  e1)
    {// statements. Innermost catch block}
    }
    catch(Exception e2)
    {/*statements. Middle catch block.*/}
    }
catch()
    {/* Statements. outer most catch block*/}
```

Example: NestedTry.java

```
class NestedTry
{
  public static void main (String args[])
  {
    int [] array = {6, 7};
    //outer try block
    try {
       System.out.println("Entered outer try block.");
       // inner try block
       try {
         System.out.println("Entered inner try block.");
         for (int i = 0; i<= 2; i++)
          System.out.println("Array Element["+i+"]= " + array[i]);
          System.out.println ("Exiting try block.");
         }
```

```
          // inner catch block
          catch (ArrayIndexOutOfBoundsException e)
           {
       System.out.println("ArrayIndexOutOfBoundsException caught");
       System.out.println("Exiting inner catch block.");
             }

      int n = 5, j=2 ;
         for (j =2; j>=0; j--)
            System.out.println ("n/j= " + n/j);
        }
      // outer catch block
         catch (ArithmeticException E)
         {
           System.out.println ("Arithmetic Exception caught");
         }
  }
}
```

Output:

```
C:\>javac NestedTry.java

C:\>java NestedTry
Entered outer try block.
Entered inner try block.
Array Element[0]= 6
Array Element[1]= 7
ArrayIndexOutOfBounds Exception caught
Exiting inner catch block.
n/j= 2
n/j= 5
Arithmetic Exception caught
```

13. Rethrowing Exception

- An exception may be thrown and caught and also partly dealt within a catch block, and then, rethrown.
- In many cases, the exception has to be fully dealt within another catch block, and throw the same exception again or throw another exception from within the catch block.
- When an exception is rethrown and handled by the catch block,
  - the compiler verifies that the type of re-thrown exception is meeting the conditions that the try block is able to throw
  - and there are no other preceding catch blocks that can handle it.

Example: Rethrow.java

```
class Rethrow
{
  public static void main (String Str[])
  {
   int[] array = {6, 7};
   try {
   System.out.println ("Entered inner try block.");
   for (int i = 0; i<= 2; i++)
      System.out.println ("Array Element["+i+"]= " + array[i]);
   System.out.println ("Exiting try block.");
   }catch (ArrayIndexOutOfBoundsException e)
   {
 System.out.println ("ArrayIndexOutOfBounds Exception caught");
 System.out.println ("Throwing e and exiting inner catch block.");
 throw e;
   }
  }
}
```

Output:

```
C:\ >javac Rethrow.java

C:\ >java Rethrow
Entered inner try block.
Array Element[0]= 6
Array Element[1]= 7
ArrayIndexOutOfBounds Exception caught
Throwing e and exiting inner catch block.
Exception              in              thread              "main"
java.lang.ArrayIndexOutOfBoundsException: Index 2 out of bounds
for length 2
        at Rethrow.main(Rethrow.java:9)
```

14. Throws Clause

The **throws clause** specifies that the method can throw an exception. **throws** is a keyword.

The **Java 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 normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

We can throw either checked or unchecked exception

Syntax of java throws

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

Example: Computation.java

```
class MyException extends Exception
{
      MyException(String Message)
      {
      super (Message);
      }
}

public class Computation
{
      public void Compute() throws MyException
      {
         int n = 5, m = 2, z, i ;
         for (i = 0; i<5; i++)
         {
            if ((m-i) == 0)
                   throw new MyException ("Denominator is zero.");
               System.out.println ("z = n/(m-i) = " + n/(m-i));
            }
        }
        public static void main (String args[])
        {
         try
            {
            new Computation ().Compute();
         }catch (MyException me)
         {
            System.out.println("Division by zero exception caught.\n
me =" + me);
            }
      }
}
```

Output:

```
C:\ >javac Computation.java

C:\ >java Computation
z = n/(m-i) = 2
z = n/(m-i) = 5
Division by zero exception caught.
 me =MyException: Denominator is zero.
```