

Packages and Java Library: Introduction, Defining Package, Importing Packages and Classes into Programs, Path and Class Path,

Packages in Java SE: Java.lang Package and its Classes: 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.

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.

Packages and Java Library

**Q) Explain about Different types of Packages in Java.
(or)**

Explain about creation and importing of User defined Packages in Java.

Package :Package is the group of classes , interfaces etc. These classes are used by other programs by importing that package.

Types of Packages :In Java, Packages are two types. They are :

- 1.System Defined(Java API) Built-in Packages
2. User Defined Packages

1.System Defined (Java API) Packages: These are provided by Java API(Application Programming Interface). They are :

- **java.lang:** It contains classes for language support.
- **java.util :** It contains Vector classes.
- **java.io :** It contains classes for input – output related operations.
- **java.awt :** It contains classes to develop Buttons , Menus etc.
- **java.net :** It contains classes for Networking Operations.
- **java.applet :** It contains classes to develop Applet programs.

2. User Defined Packages :These packages are created by programmers. The process creating and usage of User Defined packages as follows :

- Creating a Package
- Adding an interface or a class to existing Package
- Importing Packages

→ **Creating a Package :** It is the process of declaring a package name and defining classes for that package.

To declare a package we use a special keyword “**package**” as follows :

Syntax : `package Package_name ;`

```

public class Class_name
{
    variables ;
    methods ;
}

```

Note : open “Notepad” and type the code as follows

Example : `package P1 ;`

```

public class A
{
    public int x ;
    public void displayA( )
    {
        System.out.println(" From class A in pack P1 , x = " + x );
    }
}

```

In the above program, we create an User Defined package ‘ **P1** ’ and it contains a class ‘ **A** ’ .

✓ Saving the Package : To save a Package program , we follow below steps :

- create a new folder with name same as package name (Ex : P1)
- save the file with name same as “public class name.java” in folder P1.
 - Ex : A.java

✓ Compiling the Package : To compile above package program, goto folder P1 and compile as follows at command prompt.

C:\> P1> javac A.java (or) C:\> javac -d . A.java

→ **Adding Interface(or)class to an Existing Package :**

It is possible to add one more class/interface to the same Package. For Example , the above package ‘ **P1** ’ contains class **A**. Now we can add an interface **K** to the package **P1**

For this ,open another notepad file and type as follows :

```

package P1;

public interface K
{
    public void show( ) ;
}

```

Now the User Defined package ‘ **P1** ’ one class **A** and an interface **K** .

- ✓ Saving: To save the above program, go to folder P1 and save as “ K.java”
- ✓ Compiling : To compile above program, goto folder P1and compile as follows at command prompt.

C:\> P1>javac K.java (or) C:\>javac -d . K.java

→ Importing Packages :To use the classes of a package, we need to import the packages into our programs. For this, we use keyword “ **import**” as follows :

Ex : import P1.* ; *//symbol * represents all classes of P1 are imported*

(or)

import P1.A ; *// here A represents only class A of P1 is imported*

The following program shows how to import package P1 into another program.
open new notepad file and type as follows :

```
import P1.A ;
import P1.K ;

public class B implements K    /* using interface K in pack P1*/
{
    public void show()
    {
        System.out.println(“From interface K in pack p1”);
    }
}

class use_pack
{
    public static void main( String args[ ] )
    {
        A a = new A( ); /* using class A in pack P1*/
        a.x = 10 ;
        a.displayA( ) ;
        K i ;
        B b=new B();
        b = i ;
        b.show( ) ;
    }
}
```

Now save the above code as “ use_pack.java “ in the drive where folder P1 exists.
After that compile and execute “use_pack.java “ at command prompt as follows :

```
C:>javac use_pack.java
```

```
C:> java use_pack
```

then output will be as follows :

```
From class A in pack P1 , x = 10
From interface K in pack P1
```

Note 1: In the above program(use_pack.java) , we are not defining classes A and B but we are using those classes by importing package P1 into our program , since P1 contains classes A and B.

Note 2: This is main advantage of creating packages. i.e. the classes/interfaces of one program are used in another program.

Q) Explain Access Specifiers / Access Modifiers(Visibility control) in Java (or)

Explain different levels of Access Protection.

Sometimes we need to restrict the access of the variables and methods at outside of the classes or packages. For this we use some keywords as modifiers before the variables and methods. These are called “Access Modifiers or Access Specifiers” .

Java supports three access modifiers : 1. **public**
2. **protected**
3. **private**

These are used to change the access location of the variables and methods as shown below :

Access modifiers → Access location ↓	Public	Protected	Friendly (default)	Private Protected	private
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same Package	Yes	Yes	Yes	Yes	No
Other classes in same packages	Yes	Yes	Yes	No	No
Subclass in other packages	Yes	Yes	No	Yes	No
Non-subclasses in other packages	Yes	No	No	No	No

public: It is used to giving the access at any where.

protected: It is the access level between public friendly access.

friendly : It is default level access. i.e. no access modifier is used.

private protected :It is limited to that class and its sub class only.

private: It is access level which is limited to that class only.

Path and ClassPath

PATH is an environment variable that is used to find and locate binary files like “java” and “javac” and to locate needed executables from the command line

// To set PATH in the window OS.

set PATH=%PATH%;C:\Program Files\Java\JDK1.5.10\bin

CLASSPATH is an environment variable that is used by the application ClassLoader or system to locate and load the compiled Java bytecodes stored in the .class file.

// To set CLASSPATH in window OS.

set CLASSPATH=%CLASSPATH%;C:\Program Files\Java\JDK1.5.10\lib

1.	An environment variable is used by the operating system to find the executable files.	An environment variable is used by the Java compiler to find the path of classes.
2.	PATH setting up an environment for the operating system. Operating System will look in this PATH for executables.	Classpath setting up the environment for Java. Java will use to find compiled classes.

Java.lang package and its classes in Java

“java.lang” Provides classes that are fundamental to the design of the Java programming language. The most important classes are Object, which is the root of the class hierarchy, and Class, instances of which represent classes at run time.

Following are the Important Classes in Java.lang package :

1. **Boolean**: The Boolean class wraps a value of the primitive type boolean in an object.
2. **Double**: The Double class wraps a value of the primitive type double in an object.
3. **Float**: The Float class wraps a value of primitive type float in an object.
4. **Integer**: The Integer class wraps a value of the primitive type int in an object.
5. **Long**: The Long class wraps a value of the primitive type long in an object.

(Note : The above 5 classes are called “Wrapper Classes”)

6. **Enum**: This is the common base class of all Java language enumeration types.

7. **Math** – The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
 8. **String**: The String class represents character strings.
 9. **StringBuffer**: A thread-safe, mutable sequence of characters.
 10. **System**: The System class contains several useful class fields and methods.
 11. **Thread**: A thread is a thread of execution in a program.
 12. **Throwable**: The Throwable class is the superclass of all errors and exceptions in the Java.
 13. **Exception**: contains Subclasses for different Built-in Exceptions.
-

Q) Explain about Wrapper Classes in Java.

The classes which are used to convert primitive data types into objects and object wraps primitive data type are called “Wrapper Classes”.

Java provides following Wrapper classes which are available in “ java.lang package”.

Primitive Data type	Wrapper class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
Boolean	Boolean

Each of the above wrapper class contains constructors and methods to perform conversion.

Wrapper class constructors : These are used to convert primitive datatype into object. For example declare primitive datatype values as :

```
int i ;
float f ;
long l ;
double d ;
```

- Integer ival =new Integer(i) ;

Integer class converts ‘i’ value belongs to primitive type 'int' into an object ‘ival’ .

- Float fval =new Float (f) ;

Float class converts ‘f’ value belongs to primitive type 'float' into an object ‘fval’ .

- Long lval = new Long (l);

Float class converts 'l' value belongs to primitive type 'long' into an object 'lval' .

- Double dval = new Double (d);

Float class converts 'd' value belongs to primitive type 'double' into an object 'dval' .

Wrapper class methods:

The wrapper classes contains methods for handling primitive data types and objects.

- Converting primitive number to object number

Ex : Integer ival = new Integer(10);

Float fval= new Float(10.5f);

Double dval = new Double(10.5);

- Converting object numbers to Primitive numbers

Ex : int i = ival . intValue();

float f= fval . floatValue();

double d= dval. doubleValue();

- Converting numbers to strings

Ex : str = Integer. toString (i) ;

str = Float . toString(f) ;

str = Double.. toString (d);

- Converting string objects to numeric objects

Ex : intobj = Integer. valueOf(str) ;

fl_obj = Float.valueOf(str) ;

dou_obj = Double.valueOf(str) ;

- Converting Numeric strings to primitive numbers

Ex : int i = Integer.parseInt(str);

double d1= Double.parseDouble(str);

In this way we can convert primitive data type into object type and vice-versa by using Wrapper classes.

Autoboxing and Unboxing:

The automatic conversion of primitive data types into its equivalent Wrapper type is known as "Auto-Boxing " and opposite operation is known as "Auto- unboxing".

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.

Example of Autoboxing :

```
class Boxing
{
    public static void main(String args[])
    {
        int a=50;
        Integer a2 = new Integer( a );    //Boxing
        Integer a3 = 7 ;    //auto Boxing
        System.out.println(a2+" , "+a3);
    }
}
```

In the above example, automatically converted primitive **7** into Integer object **"a3"**

Output: 50 , 7

Example of Unboxing

The automatic conversion of wrapper class type into corresponding primitive type, is known as "Unboxing".

```
class Unboxing
{
    public static void main(String args[ ])
    {
        Integer i=new Integer(50);
        int a=i;    // auto un boxing
        System.out.println(a);
    }
}
```

In the above example, Wrapper class object **"i"** converted into primitive integer **"a"**

Output: 50

****java.lang.Enum(Enumeration)**

- The **Enum** is a class or user defined data type available in "java.lang package" which contains a fixed set of constants.
- The **enum** data type (also known as Enumerated Data Type) is used to define an enum in Java.
- According to the Java naming conventions, we should have all constants in uppercase letters. The Java enum constants are static and final implicitly.

Example Program:

```

class EnumExample
{
    public enum Season { WINTER, SPRING, SUMMER, FALL } //defining enum within class
    public static void main ( String args[ ] )
    {
        for (Season s : Season.values( )) //printing all enum 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());
    }
}

```

Output:

```

WINTER
SPRING
SUMMER
FALL
Value of WINTER is: WINTER
Index of WINTER is: 0
Index of SUMMER is: 2

```

java.lang.Math Class

In Java, the **Math** class, part of the java.lang package, provides methods for performing basic numeric operations such as exponentiation, logarithms, trigonometry, and more.

The methods of the Math class are static, so they can be accessed directly without creating an object of the class.

They are :

1. **abs(double a)**:Returns the absolute value of a number.
Example: Math.abs(-5) → 5.
2. **max(int a, int b)**:Returns the larger of two numbers.
Example: Math.max(3, 7) → 7.
3. **min(int a, int b)**:Returns the smaller of two numbers.
Example: Math.min(3, 7) → 3.
4. **ceil(double a)**: Returns the smallest integer value greater than or equal to a (rounds up).
Example: Math.ceil(3.2) → 4.0.
5. **floor(double a)**:Returns the largest integer value less than or equal to a (rounds down).
Example: Math.floor(3.7) → 3.0.
6. **round(double a)**:Returns the closest long or int (depending on input type) to the argument, with ties rounding up.
Example: Math.round(3.5) → 4.
7. **pow(double a, double b)**:Returns a raised to the power of b.
Example: Math.pow(2, 3) → 8.0.
8. **sqrt(double a)** :Returns the square root of a.
Example: Math.sqrt(16) → 4.0

9.log(double a) : Returns the natural logarithm (base e) of a.

Example: Math.log(2) → 0.639

10.sin(double a) :Returns the sine of an angle (in radians).

Example: Math.sin(0) → 0

11.cos(double a) :Returns the cosine of an angle (in radians).

Example: Math.cos(0) → 1

Constants in Math Class

- **Math.PI**
The value of π (3.14159...).
- **Math.E**
The value of Euler's number (2.718...).

(Note :Euler's number is the constant base of the natural logarithm and has profound applications in calculus, exponential growth, complex analysis)

java.util Package(Formatter class and Random Class)

The **java.util** package is a standard package of Java . This package is very useful and it provides commonly used Collections like ArrayList, HashMap, and other utility classes for event model handlings, date and time operations etc.

This package is imported as follows :

```
import java.lang.*;
```

Following is the list of some important classes in java.util.package:

- Arrays – The ArrayList class contains various methods for manipulating arrays (such as sorting and searching).
- Collections – This class consists methods that operate on or return collections.
- Formatter – This class provides methods for layout justification and alignment, common formats for numeric, string etc.
- Random – This class instance is used to generate a stream of pseudo random numbers.
- Scanner – This class is a simple text scanner which can parse primitive types and strings using regular expressions.
- Vector – Vector implements a dynamic array.

java.util.Formatter class:

- The **Formatter** is a built-in class in java used for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output in java programming.
- The Formatter class is defined inside the **java.util** package.

- The Formatter class in java has the following methods.

1. **format()**: Formats the arguments according to the specified format string.
2. **out()**: Returns the Appendable object (often StringBuilder) that the formatter uses for output.
3. **toString()**: Returns the contents of the formatter as a String.
4. **locale()**: provide information about language and region
5. **flush()** : used to clear any data in output buffer and ensure that the data is written to its intended destination.
6. **Close()** : closes the invoking formtter

Example Program :

```
import java.util.Formatter;
import java.util.Locale;

class FormatterExample {
    public static void main(String[] args) {
        // Creating a Formatter instance with default locale
        Formatter fmt = new Formatter();

        // format() method:
        fmt.format("Hello, %s ! %n", "World");
        fmt.format("Number: %d, Float: %.2f%n", 42, 3.14159);
        System.out.println("Using format():");
        System.out.println(fmt);

        // out() method:
        Appendable out = fmt.out();
        System.out.println("Using out():");
        System.out.println(out);

        // toString() method:
        System.out.println("Using toString():");
        System.out.println(fmt.toString());

        // locale() method:
        Locale locale = fmt.locale();
        System.out.println("Using locale():");
        System.out.println("Locale: " + locale);
    }
}
```

```
}
```

Output :

Using format():

Hello, World!

Number: 42, Float: 3.14

Using out():

Hello, World!

Number: 42, Float: 3.14

Using toString():

Hello, World!

Number: 42, Float: 3.14

Using locale():

Locale: en_IN

java.util.Random Class :

- The Random class in Java is part of the java.util package and is used to generate pseudo random numbers in various data types like integers, doubles, floats, booleans, etc.
- It provides methods to generate random values and is commonly used in applications involving games, simulations, and random data generation.

Commonly Used Methods in the **Random** Class

- **nextInt():** Returns a random int within the full range of integers.
- **nextInt(int bound):** Returns a random int between 0 (inclusive) and the specified bound (exclusive).
- **nextLong():** Returns a random long value.
- **nextFloat():** Returns a random float between 0.0 (inclusive) and 1.0 (exclusive).
- **nextDouble():** Returns a random double between 0.0 (inclusive) and 1.0 (exclusive).
- **nextBoolean():** Returns a random boolean (either true or false).
- **ints(), longs(), doubles():** Stream-generating methods that produce streams of random values of specified types, useful for generating large datasets.
- **Example Program**

```
import java.util.Random;
```

```
class RandomExample
```

```
{
```

```
    public static void main(String[ ] args)
```

```
{
```

```
Random random = new Random();
```

```
System.out.println("Random Integer: " + random.nextInt()); // Any integer
```

```
System.out.println("Random Integer (0 to 9): " + random.nextInt(10)); // Integer between 0 and 9
```

```
System.out.println("Random Float (0.0 to 1.0): " + random.nextFloat());
```

```
System.out.println("Random Boolean: " + random.nextBoolean());
```

```
}
```

```
}
```

Output:

Random Integer: 1749266783

Random Integer (0 to 9): 7

Random Float (0.0 to 1.0): 0.9562389

Random Boolean: true

“java.time “ Package

(Formatting Date and Time ,Instant andTemporal Adjusters classes):

- The java.time package, is used for handling date, time, and time zones in our programs.
- Some of the important classes in “java.time” package are :

1.Date and Time Classes

The following classes in **java.time package** are used to get Date and Time and formatting them

- ❑ a) **LocalDate**: Represents a date without time or time zone (e.g., 2024-11-15).
- ❑ b) **LocalTime**: Represents a time without a date or time zone (e.g., 15:30:45).
- ❑ c) **LocalDateTime**: Represents both date and time without a time zone (e.g., 2024-11-15T15:30:45).
- ❑ **DateTimeFormatter**: For formatting and parsing date-time objects.

Example program for Formatting Date and Time using above classes

We can use the DateTimeFormatter class with the ofPattern() method in the same package to format or parse date-time objects.

```
import java.time.LocalDateTime;
```

```
import java.time.format.DateTimeFormatter;
```

```
class Main
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        LocalDateTime myDateObj = LocalDateTime.now();
```

```
        System.out.println("Before formatting: " + myDateObj);
```

```
        DateTimeFormatter myFormat = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");
```

```
        String    formattedDate = myDateObj.format(myFormat);
```

```
        System.out.println("After formatting: " + formattedDate);
```

}

}

Output : Before Formatting: 2024-11-15 T 20:02:32.566257

After Formatting: 15-11-2024 20:02:32

2.java.time.Instant - Class : This Class is used to represent the specific time instant on the current timeline.Methods of Instant Class:

- a)now():This method obtains the current instant from the system clock.
- b)minus() : This method returns a copy of this instant with the specified amount subtracted.
- c) plus() : This method returns a copy of this instant with the specified amount added.

Example program for Instant class

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

class InstExample
{
    public static void main(String[] args)
    {
        Instant cur = Instant.now();
        System.out.println("Current Instant is " + cur);

        Instant diff = inst1.minus(Duration.ofDays(70));
        System.out.println("Instant after subtraction : "+ diff);

        Instant sum = inst1.plus(Duration.ofDays(10));
        System.out.println("Instant after addition : "+ sum);
    }
}
```

Output : Current Instant is 2021-03-03T16:27:54.378693Z

Instant after subtraction : 2020-12-01T11:19:42.120Z

Instant after addition : 2021-02-19T11:19:42.120Z

3.java.time.TemporalAdjusters- Class

The TemporalAdjusters class in the java.time package is a utility class that provides a collection of predefined methods for common date and time adjustments. They are

Method	Description
firstDayOfMonth()	Adjusts to the first day of the current month.
lastDayOfMonth()	Adjusts to the last day of the current month.
firstDayOfNextMonth()	Adjusts to the first day of the next month.
firstDayOfYear()	Adjusts to the first day of the current year.
lastDayOfYear()	Adjusts to the last day of the current year.

Example Program for TemporalAdjusters class

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

class TemporalAdjustersExample
{
    public static void main(String[] args)
    {
        LocalDate today = LocalDate.now();

        LocalDate firstDayOfMonth = today.with(TemporalAdjusters.firstDayOfMonth());
        LocalDate lastDayOfMonth = today.with(TemporalAdjusters.lastDayOfMonth());

        System.out.println("Today: " + today);
        System.out.println("First Day of Month: " + firstDayOfMonth);
        System.out.println("Last Day of Month: " + lastDayOfMonth);
    }
}
```

Output : Today: 2024-11-15

First Day of Month: 2024-11-01

Last Day of Month: 2024-11-30

-----packages over-----

Exception Handling

Q) Define Exception . Explain Exception Handling process.

(OR)

Explain different types of Errors. Explain how Errors and Exceptions handled in Java.

(OR)

Explain the following statements :

a) try block b) catch block c) finally block

Ans) Error : While developing a program , the programmer may make mistakes and these mistakes are called “Errors”.

These errors are three types :

1. Compile-time (Syntax) Errors
2. Run-time Errors
3. Logical Errors

1. Compile-Time (Syntax) Errors : These errors are found at compilation time of the program. These errors are raised by mistakes in syntax of that programming language. For Example :

- Missing semicolons
- Mismatch or unclosed of brackets in classes and methods
- spelling mistakes of identifiers and keywords ,..etc

Due to these errors , we cannot execute the program.

2. Run-Time Errors : These errors are found at execution time (Run time)of the program. These Errors may raised because of following conditions :

- Dividing an integer by zero.
- Accessing an element that is out of the bounds of an array.
- Trying to store a wrong data type value into an array etc.

Due to these Errors, we can't get the desired output, or program may terminated etc.

3. Logical Errors : These errors raised because of errors in logic of the program. The programmer might be using wrong formula or the design of the program itself is wrong.

➤ **Exception :** The Runtime error condition is called “Exception”.

Due to these Exceptions, we may face problems like System crash, program termination, etc. So we should handle these Exceptions to get desired output .

Exception Handling : The process of catching an exception and continues the execution of remaining code without any disturbance is known as “Exception Handling”.

Due to this process , we overcome problems caused by Exceptions at Runtime.

Java supports the Exception Handling mechanism with the following steps :

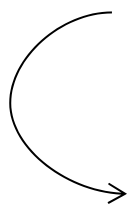
1. *Find* the exception(Find the problem)
2. *throw* the exception (Inform that an error has occurred).
3. *catch* the exception (Receive the error information).
4. *handle* the exceptions (Take corrective actions).

To implement the above steps in a program ,we use the below syntax :

```

try
{
.....
.....
}
catch( ExceptionType e )
{
.....
.....
}

```



Here **try** , **catch** are the keywords and according to concept these are called “Exception Handlers” (Exception Handling statements/blocks).

→ **try block** : In this block, we write the statements which may raises Exceptions. If an Exception is raised, try block throws that Exception to “catch block”.
i.e. try block performs step1 and step2 of Exception Handling process.

→ **catch block** : This block receives an Exception thrown by “try block” , and wewrite the statements to handle that Exception.
i.e. catch block performs step3 and step4 of Exception Handling process.

The following program shows the Exception Handling Process :

```

class ExHandle
{
public static void main( String args [ ] )
{
    int a , b c;

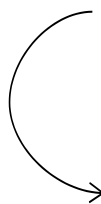
    a = 10 ;
    b = 5 ;
    c = 5 ;
}
}

```

```

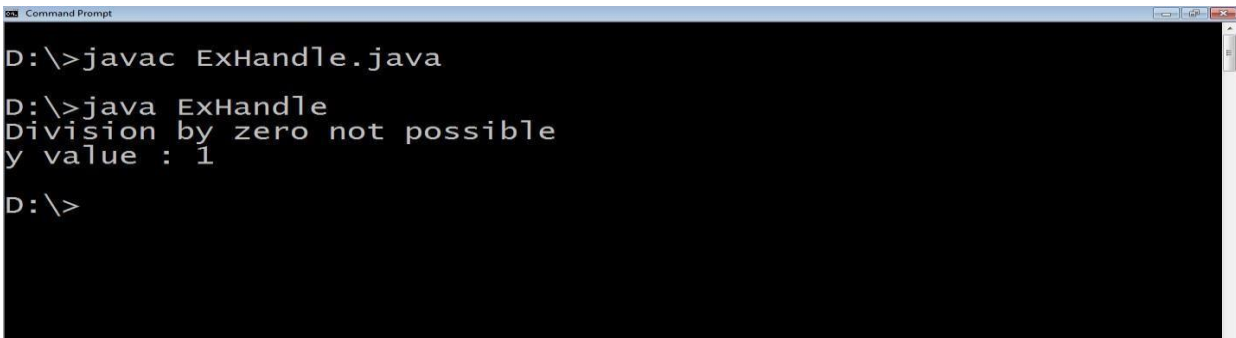
try
{
    int x = a / ( b - c );    /* raise ArithmeticException */
    System.out.println( " x value : " + x );
}
catch( ArithmeticException e )
{
    System.out.println( "Division by zero is not possible " );
}
int y = a / ( b + c );
System.out.println( " y value : " + y );
}
}

```



Explanation : As shown above, the integer 'a' is dividing by zero (b-c=0). Generally it is not possible, so it is Arithmetic Exception which leads to runtime error. This Exception is raised in try block and throws into catch block. Then catch block displays necessary information and continues the execution of remaining code.

Then the output will be :



```

D:\>javac ExHandle.java
D:\>java ExHandle
Division by zero not possible
y value : 1
D:\>

```

Advantages of Exception Handling:

- Separating the statements(code) which have runtime errors in program.
- No disturbance for execution of remaining(correct) code.
- Programmer can identify Error Type and necessary action.

Hierarchy of standard Exceptions/Types of Exceptions (or) Hierarchy Throwable class

Exception : Exception is a RunTime-Error condition. These exceptions are two types :

- a) Checked Exceptions
- b) Unchecked Exceptions

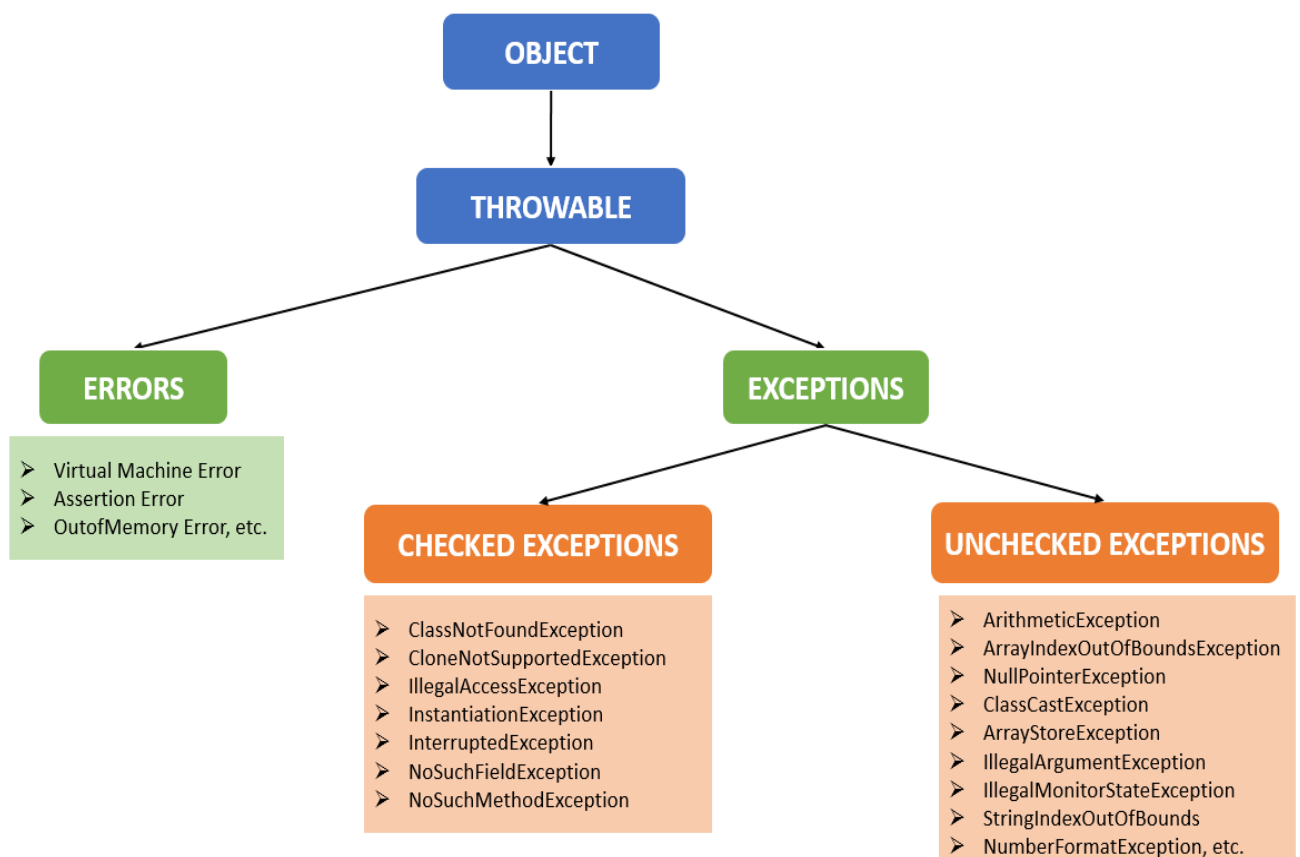
a) Checked Exceptions :

- These should be handled in the code itself using try-catch blocks.
- These exceptions are checked at compilation time by the java compiler.
- These exceptions are extended from the “**java.lang.Exception**” class.

b) Unchecked Exceptions :

- These exceptions are not compulsory handled in the program code, because the JVM (Java Virtual Machine) handles such exceptions.
- These are extended from “**java.lang.RuntimeException**” class.

The following diagram shows Hierarchy of Standard Exceptions and Throwable class



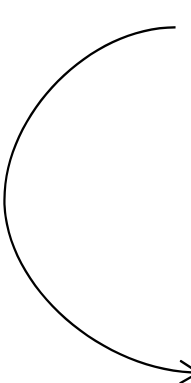
Q) Explain Multiple catch Statements and finally block.

Multiple catch Statements : Sometimes there is need to handle more than oneException in the program, then we use more than one “catch” Statement(block).

finally statement :This is special statement block. It can receive any type of Exception.The statements in this block are guaranteed to execute irrespective of Exception.

Here “**finally**” is the reserved keyword like **try** and **catch** .

The following program shows the working of multiple catch and finally statements:



```

class ExHandle
{
    public static void main( String args [ ] )
    {
        int a[ ] = { 10,5 };
        int b = 5 ;
        try
        {
            int x = a[2] / ( b - a[1] ); /*raise ArrayIndexOutOfBoundsException Exception*/
            System.out.println(“ x value : “ + x ) ;
        }
        catch( ArithmeticException e )
        {
            System.out.println( “Division by zero is not possible “ ) ;
        }
        catch( ArrayIndexOutOfBoundsException e )
        {
            System.out.println(“ You are accessing out of bound element“ ) ;
        }
        catch( ArrayStoreException e )
        {
            System.out.println( “ Wrong Datatype values stored” ) ;
        }
        finally
        {
            int y = a[1] / a[0] ;
            System.out.println(“ y value = “ + y ) ;
        }
    }
}

```

Explanation: In the above program, we are accessing a[2](i.e. third element in Array) but it is not exist because there are two elements in given array “a”. So when Exception raised, the related catch block(2nd catch block) is executed and remaining catch blocks are not executed.
Similarly , finally block is executed irrespective of exceptions.

Output : You are accessing out of bound element
y value : 2

.....

Q) Explain about user defined Exceptions(our own Exceptions)

In Java , it is possible to define our own Exceptions. The user defined Exceptions are defined by extending the predefined class “Exception”.

The following program shows the creation of User Defined Exceptions

```
import java.lang.Exception ;

class myExcep extends Exception
{
    myExcep(String msg)
    {
        super(msg);
    }
}

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

    {
        int x , y ; x=5; y=1000;

        float z = (float) x / (float) y ;

        try
        {
            if (z<0.01)
                throw new myExcep("Number is too small") ;
        }
        catch(myExcep e)
        {
            System.out.println(e.getMessage());
        }
        finally
        {
            System.out.println("z value : "+z);
        }
    }
}
```

```
    }
}
```

Explanation : In the above program, “myExcep” is user defined Exception. When z value is less than 0.01, User Defined Exception is raised and it is caught by catch block. Then object “e” contains the error message “ Number is too small ” and it is displayed by method “getMessage()” .

Output : Number is too small
z value : 0.005

Explain about “ throw “ and “throws” clauses (statements) (or)

Explain how to throw and rethrow an Exception

throw : The **throw** keyword is used to explicitly throw an exception in our code.

- It provides a way to transfer control to an appropriate exception handler.

syntax :

```
throw    new    throwable_Subclass ("message") ;
```

Usage of throw Keyword :

Normally, when exception is raised in try block , it automatically throw or we explicitly throw to catch block by using throw keyword .It is called **throwing** an Exception.

If catch block is unable to handle it, The exception can re-throw using **throw** keyword to another part of code in our program . This process is called as **re- throwing** an exception.

Class ThrowEx

```
{
public void division()
{
    int n1=30;
    int n2=0;
    try
    {
        int x=n1/n2;
        System.out.println("x value "+x);
    }
    catch(ArithmeticException e)
    {
        throw e ; //Exception rethrow to catch block in main class
    }
}
}
```

```

class Test
{
public static void main(String[] args)
{
    ThrowEx r = new ThrowEx();

    try
    {
        r.division( );
    }
    catch ( Exception e )
    {
        System.out.println(e.getMessage());
        System.out.println("This Exception rethrown by division method and caught by main method");
    }
}
}

```

output : / by zero

This Exception rethrown by division method and caught by main method

throws : It is special keyword which is used with methods. It specifies that, the method might throw one or more Exceptions.

Syntax : type method-name(parameter-list) **throws** exceptions-list

```

{
    // body of method
}

```

Example: static void divide() **throws** ArithmeticException

```

{
    int x = a / b ;
}

```

printStackTrace() : This method is a debugging tool that provides detailed information about where an exception occurred, including the method calls that led to the exception.

```

class Demo
{
public static void main(String[] args)
{
    int a=7,b=0;
    try
    {
        int x = a / b; // This will cause ArithmeticException
    } catch (ArithmeticException e) {
        e.printStackTrace();
    }
}
}

```

Output : java.lang.ArithmeticException: / by zero at Demo.main(Demo.java:8)

-----Exception handling over-----

*****Operations on Array Elements(Topic in unit-3)**

The basic operations that we can perform on array elements are as follows:

- **Traverse Array:** Looping through and displaying each element.

```
Ex: int array[]={ 9,10,5,8,12};
    for(int i=0;i<array.length;i++)
        System.out.println(array[i]) ; //9,10,5,8,12
```

- **Insert Elements:** Adding elements to an array at specific indexes.(ex: insert 7 at position 3)

```
// Create a new array with one extra slot
int[] newArray = new int[array.length + 1];
// Copy elements before the position
for (int i = 0; i < position; i++)
{
    newArray[i] = array[i];
}
// Insert the new element
newArray[position] = element;
// Copy the remaining elements after the position
for (int i = position; i < array.length; i++) {
    newArray[i + 1] = array[i];
}
```

After insertion, array elements : 9,10,5,7,8,12

- **Delete Elements:** Removing elements, typically by setting the value to 0 or shifting elements.
(e.g.,delete at index 2)

```
for (int i = index; i < array.length - 1; i++)
{
    array[i] = array[i + 1]; // Shift elements left
}
elements After updating at index 2 : 9,10,7,33,12
```

- **Search Element:** Finding the index of a specific element.

```
int searchKey = 20;
for (int i = 0; i < array.length; i++)
{
    if (array[i] == searchKey) {
        return i; // Return the index if found
    }
    return -1; // Return -1 if not found
}
```

- **Sort Elements:** Arranging elements in ascending or descending order.

```
Arrays.sort(array);
Elements after sorting : 5,7,9,10,12,33
```

Here sort() is the method of “ **Arrays**” class which is available in “ **java.util** ” package

Unit-4 Imp Questions

1. Explain defining and importing user Defined Packages.
 2. What is role of wrapper classes in Java.
 3. Elaborate on java.util(Formatter, **Random** classes), java.time(Date**Time** ,**Instant** Classes).
 4. Define Exception. How to Handle Exceptions in Java.
 5. Creating user defined exceptions.
 6. Discuss about **throw** and **throws** Keywords with examples.
-

Note :following are Extra Questions but related to Packages and Exception Handling . So you go through these after preparing all main questions

Q) Explain the following :

- a) Extending classes from packages(Inheritance in Packages)
- b) Hiding classes from packages

a) Extending classes from packages(Inheritance in Packages) :

Sometimes it is need to extend (derive new classes) from the classes of a package into our program. For this , import that package and derive new classes by using the keyword “**extends** “ (like Inheritance programs).

For example package P1 contains the class A as

```
follows :package P1 ;

public class A
{
    public int x ;
    public void display( )
    {
        System.out.println(“ From class A in pack P1 , x = “ + x );
    }
}
```

Now we derive a new class C from class A in another new program as

```
follows :import P1.A;

class C extends A
{
    void displayC()
    {
        System.out.println(“ C derived from A , x = “ + x ) ;
    }
}
```

```

    }
    class Ext_pack
    {
        public static void main( String args[ ] )
        {
            C obj = new C ( );
            obj.x = 10 ;// class C uses variable of class A
            obj.displayC( ) ;
        }
    }

```

In the above program , class C is derived from class A and this class A existed in Package P1. According to Inheritance property the variables and methods of class A are used by class C. This is Inheritance or extending of packages.

Output :C derived from A , x= 10

b) Hiding classes from packages :

Sometimes there is need to stop the accessing (importing) of classes from the package into another programs. This is called “ hiding of classes “ .

For this process , we declare the classes as “ not public “ ,i.e. the keyword “public” should not use before class definition.

For example the package P1 contains class A and class B as

follows :package P1 ;

```

public class A
{
-----
-----
}

class B
{
-----
-----
}

```

As shown above the class B is not declared with “**public**” , so it is not accessible into other programs . But class A is accessible in other programs.

```

Ex : import P1.A ;    // no error
      import P1.B ;    // gives error

```

Q) Explain creating Sub Packages(packages Hierarchy).

Sometimes a package may contain other sub packages also. i.e. a package is defined in other (already existed) package. Then we can import those sub

packages in Hierarchical manner as follows

Syntax : import Main_pack_name[.Sub_pack_name].Class_Name;

Ex : import java.awt.color ; //System defined
 packageimport P.p1.B; // User defined
 package

Here **java** and **P** are the main packages. **.awt** and **p1** are sub packages. **Color** and **B** are classes defined in sub packages.

Example for Creation of Hierarchy of Packages(User defined) :

a) Creation of Main Package mp :

```
package mp ; public
class A
{
    public void displayA()
    {
        System.out.println("From Main Pack mp ");
    }
}
```

Save the above code as "A.java" in folder **mp**. After that , go to folder mp and compile at command prompt as follows :

```
C:\>mp>javac A.java or C:\> javac -d . A . java
C:\>mp>javac B.java
```

b) Creation of Sub package sp :

```
package mp .sp ;
public class B
{
    public void displayB()
    {
        System.out.println("From Sub Pack sp ");
    }
}
```

Now create a new folder **sp** should as sub folder in folder **mp** which is already created.

Save the above code as "B.java" in folder **sp**. After that go to folder **sp** and compile at command prompt as follows :

```
C:\>mp\cd sp
C:\>mp\sp>javac
```

B.java

c) Importing subpackage and its classes from main package :

```
import mp.sp.*;

class pack_Hier
{
    public static void main(String args[])
    {
        B b=new B();
        b.displayB();
    }
}
```

Above code saved as “pack_Hier.java” in drive where folder P is located.compile as : C:\>javac pack_Hier.java

Run as : C:\>java pack_Hier

Output : From subpackage sp

In the above program , we created sub package **sp** in the main package **mp** and import them into “pack_Hier.java”

****Uncaught Exceptions:**

An Exception that is not caught (uncaught) by our program is called “Uncaught Exception”. These will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred and terminates the program.

The following small program contains an expression that causes a divide-by-zero error:

```
class Test
{
    public static void main(String args[])
    {
        int d = 0;
        int a = 42 / d;
    }
}
```

When the Java run-time system detects the statement that attempts to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of “Test” to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately. In this example, we haven’t supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.

Here is the exception generated when this example is executed:

java.lang.ArithmeticException: / by zero at Test.main(Test.java:4)

The class name Exc0, the method name main(), the filename Exc0.java and the line number 4, are all included in the simple stack trace. Also, notice that the type of exception thrown is a subclass

of Exception called `ArithmeticException`, which more specifically describes what type of error happened.

**** Nested try Statements:**

The try statement can be nested. That is, a try statement can be inside the block of another try statement.

If an inner try statement does not have a catch handler for a particular exception, the next try statement's catch handlers are inspected for a match. This continues until one of the catch statements succeeds or until all of the nested try statements are exhausted. If no catch statement matches then the Java run-time system will handle the exception. Here is an example that uses nested try statements:

```
class NestedTry
{
    public static void main(String args[])
    {
        // outer try block
        try {
            int a[] = { 1, 2, 3, 4, 5 };
            System.out.println(a[5]); /*trying to print element at index 5 */

            // inner try-block
            try
            {
                int x = a[2] / 0; /* performing division by zero */
            }
            catch (ArithmeticException e2)
            {
                System.out.println("division by zero is not possible");
            }
        }

        catch (ArrayIndexOutOfBoundsException e1)
        {
            System.out.println("ArrayIndexOutOfBoundsException");
        }
    }
}
```

Output : `ArrayIndexOutOfBoundsException`

In the above program , outer try -catch block handles the “`ArrayIndexOutOfBoundsException`”and Inner try-catch block handles the “`ArithmeticException`”.
