# MODULE III
# PACKAGES

**STRUCTURE**



**LEARNING OBJECTIVES**

- To understand the concepts of Packages and Interfaces.
- To understand exception handling and to learn the usage of try and catch block.

## 3.1 INTRODUCTION

**Package** in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:

- Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding).

All we need to do is put related classes into packages. After that, we can simply write an import class from existing packages and use it in our program. A package is a container of a group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose. We can reuse existing classes from the packages as many time as we need it in our program.

## 3.2 DEFINITION, ACCESS PROTECTION IMPORTING PACKAGES

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds or *thousands* of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.The Java platform provides an enormous class library (a set of packages) suitable for use in your own applications. This library is known as the "Application Programming Interface", or "API" for short. Its packages represent the tasks most commonly associated with general-purpose programming.

For example, a String object contains state and behavior for character strings; a File object allows a programmer to easily create, delete, inspect, compare, or modify a file on the filesystem; a Socket object allows for the creation and use of network sockets; various GUI objects control

buttons and checkboxes and anything else related to graphical user interfaces. There are literally thousands of classes to choose from. This allows you, the programmer, to focus on the design of your particular application, rather than the infrastructure required to make it work.
The Java Platform API Specification contains the complete listing for all packages, interfaces, classes, fields, and methods supplied by the Java SE platform. Load the page in your browser and bookmark it. As a programmer, it will become your single most important piece of reference documentation.

**Access protection** -Access modifiers define the scope of the class and its members (data and methods). For example, private members are accessible within the same class members (methods). Java provides many levels of security that provides the visibility of members (variables and methods) within the classes, subclasses, and packages.Packages are meant for encapsulating, it works as containers for classes and other subpackages. Class acts as containers for data and methods. There are four categories, provided by Java regarding the visibility of the class members between classes and packages:

1. Subclasses in the same package
2. Non-subclasses in the same package
3. Subclasses in different packages
4. Classes that are neither in the same package nor subclasses

The three main access modifiers private, public and protected provides a range of ways to access required by these categories.

| | Private | No Modifier | Protected | Public |
|---|---|---|---|---|
| Same class | Yes | Yes | Yes | Yes |
| same package subclass | No | Yes | Yes | Yes |
| same package non - subclass | No | Yes | Yes | Yes |
| Different package subclass | No | No | Yes | Yes |
| Different package mon-subclass | No | No | No | Yes |

Simply remember, private cannot be seen outside of its class, public can be access from anywhere, and protected can be accessible in subclass only in the hierarchy.
A class can have only two access modifier, one is default and another is public. If the class has default access then it can only be accessed within the same package by any other code. But if the class has public access then it can be access from any where by any other code.

**Example:**

```
//PCKG1_ClassOne.java
package pckg1;
public class PCKG1_ClassOne{
int a = 1;
private int pri_a = 2;
protected int pro_a = 3;
public int pub_a = 4;
public PCKG1_ClassOne() {
System.out.println("base class constructor called");
System.out.println("a = " + a);
System.out.println("pri_a = " + pri_a);
System.out.println("pro_a "+ pro_a);
System.out.println("pub_a "+ pub_a);
}
}
```

The above file PCKG1_ClassOne belongs to package pckg1, and contains data members with all access modifiers.

*//PCKG1*_ClassTwo.java

```
package pckg1;
class PCKG1_ClassTwo extends PCKG1_ClassOne {
PCKG1_ClassTwo() {
System.out.println("derived class constructor called");
System.out.println("a = " + a);
II accessible in same class only
II System.out.println("pri_a = " + pri_a);
System.out.println("pro_a "+ pro_a);
System.out.println("pub_a =” + pub_a);
}
}
```

The above file PCKG1_ClassTwo belongs to package pckg1, and extends PCKG1_ClassOne, which belongs to the same package.

*//PCKG1*_ClasslnSamePackage

```
package pckg1;
class PCKG1_ClassInSamePackage {
PCKG1_ClassInSamePackage() {
PCKG1_ClassOne co = new PCKG1_ClassOne();
System.out.println("same package class constructor called");
System.out.println("a = " + co.a);
II accessible in same class only
II System.out.println("pri_a = " + co.pri_a);
System.out.println("pro_a "+ co.pro_a);
System.out.println("pub_a = " + co.pub_a);
}
}
```

The above file PCKG1_ClassInSamePackage belongs to package pckg1, and having an instance of PCKG1_ClassOne.

```
package PCKG1;
//Demo package PCKG1
public class DemoPackage1 {
public static void main(String ar[]) {
PCKG1_ClassOne obl = new PCKG1_ClassOne();
PCKG1_ClassTwo ob2 = new PCKG1_ClassTwo();
PCKG1_ClassInSamePackage ob3 = new PCKG1_ClassxnSamePackage();
}
}
```

The above file DemoPackageI belongs to package pckgI, and having an instance of all classes in pckg1.

```
package pckg2;
class PCKG2_ClassOne extends PCKG1.PCKG1_ClassOne {
```

```
PCKG2_ClassOne() {
System.out.println("derived class of other package constructor
called");
II accessible in same class or same package only
II System.out.println("a = " + a);
II accessible in same class only
II System.out.println("pri_a = " + pri_a);
System.out.println("pro_a = " + pro_a);
System.out.println("pub_a = " + pub_a);
}
}
```

The above file PCKG2_ClassOne belongs to package pckg2. extends PCKG I_ClassOne, which belongs to PCKG1, and it is trying to access data members of the class PCKGI_ClassOne.

*IIPCKG2_* ClassInOtherPackage

```
package pckg2;
class PCKG2_ClassInOtherPackage {
PCKG2_ClassInOtherpackage() {
PCKG1.PCKG1_ClassOne co = new PCKG1.PCKG1_ClassOne();
System.out.println("other package constructor");
II accessible in same class or same package only
II System.out.println("a ,= " + co.a);
II accessible in same class only
II System.out.println("pri_a = " + co.pri_a);
II accessible in same class, subclass of same or other package
II System.out.println("pro_a = " + co.pro_a);
System.out.println("pub_a = " + co.pub_a);
}
}
```

The above file PCKG2_ClassInOtherPackage belongs to package pckg2, and having an instance of PCKG LClassOne of package pckg I, trying to access its some data members.

*II* Demo package pckg2.

```
package pckg2;
public class DemoPackage2 {
public static void main(String ar[]) {
PCKG2_ClassOne obl = new PCKG2_ClassOne();
PCKG2_ClassInOtherPackage ob2 = new PCKG2_ClassInOtherPackage();
}
}
```

The above file DemoPackage2 belongs to package pckg2, and having an instance of all classes of pckg2

**Importing packages-**

Java has import statement that allows you to import an entire package (as in earlier examples), or use only certain classes and interfaces defined in the package.

The general form of import statement is:

```
import package.name.ClassName;   // To import a certain class only
import package.name.*   // To import the whole package
```

For example,
import java.util.Date; // imports only Date class
import java.io.*;  // imports everything inside java.io package

The import statement is optional in Java.
If you want to use class/interface from a certain package, you can also use its **fully qualified name**, which includes its full package hierarchy.
Here is an example to import package using import statement.

```
import java.util.Date;

class MyClass implements Date {
   // body
}
```

The same task can be done using fully qualified name as follows:

```
class MyClass implements java.util.Date {
   //body
```

**Example: package and importing package**
Suppose, you have defined a package **com.programiz** that contains a class Helper.

```
package com.programiz;

public class Helper {
   public static String getFormattedDollar (double value){
      return String.format("$%.2f", value);
   }
}
```

Now, you can import Helper class from **com.programiz** package to your implementation class. Once you import it, the class can be referred directly by its name. Here's how:

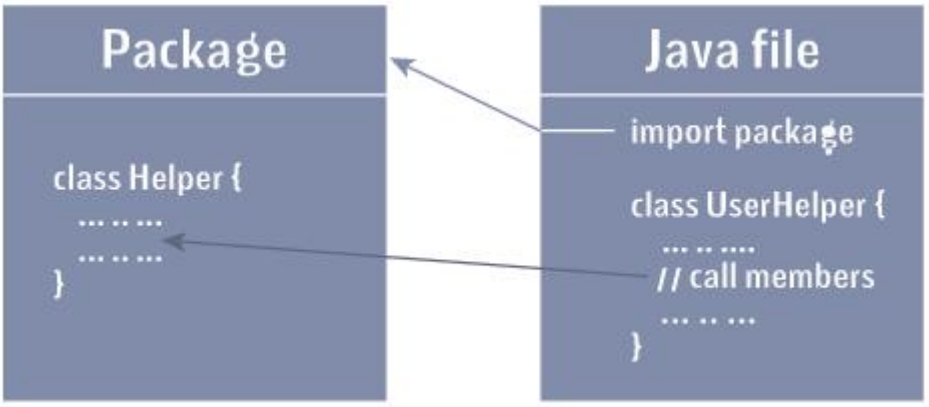
```
import com.programiz.Helper;
Class UseHelper {
   public static void main(String[] args) {
      double value = 99.5;
      String formattedValue = Helper.getFormattedDollar(value);
      System.out.println("formattedValue = " + formattedValue);
   }
}
```

When you run the above program, the output will be:

```
formattedValue = $99.50
```

Here,

1. Helper class is defined in **com.programiz** package.
2. the Helper class is imported to a different file. The file contains UseHelper class.
3. The getFormattedDollar() method of the Helper class is called from inside the UseHelper class.




In Java, import statement is written directly after the package statement (if it exists) and before the class definition.
For example,

```
package package.name;
import package.ClassName; // only import a Class
class MyClass {
   // body
}
```

### 3.3 INTERFACES: DEFINITION IMPLEMENTING INTERFACES

An interface is a reference type in Java. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface. Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types.
Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
- A Java library example is, Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

**Syntax :**

```
interface <interface_name> {

   // declare constant fields
   // declare methods that abstract
   // by default.
}
```

To declare an interface, use interface keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static and final by default. A class that implement interface must implement all the methods declared in the interface. To implement interface use implements keyword.

**A simple interface**

```
interface Player
```

```
{
   final int id = 10;
   int move();
}
```
To implement an interface we use keyword: implement
```
// Java program to demonstrate working of
// interface.
import java.io.*;

// A simple interface
interface In1
{
   // public, static and final
   final int a = 10;

   // public and abstract
   void display();
}

// A class that implements the interface.
class TestClass implements In1
{
   // Implementing the capabilities of
   // interface.
   public void display()
   {
      System.out.println("Geek");
   }

   // Driver Code
   public static void main (String[] args)
   {
      TestClass t = new TestClass();
      t.display();
      System.out.println(a);
   }
}
```
Output:
Geek
10

**A real-world example:**
Let's consider the example of vehicles like bicycle, car, bike……, they have common functionalities. So we make an interface and put all these common functionalities. And lets Bicycle, Bike, car ….etc implement all these functionalities in their own class in their own way.
```
import java.io.*;

interface Vehicle {

   // all are the abstract methods.
   void changeGear(int a);
```

```
    void speedUp(int a);
    void applyBrakes(int a);
}

class Bicycle implements Vehicle{

    int speed;
    int gear;

     // to change gear
    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){

        speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){

        speed = speed - decrement;
    }

    public void printStates() {
         System.out.println("speed: " + speed
             + " gear: " + gear);
    }
}

class Bike implements Vehicle {

    int speed;
    int gear;

    // to change gear
    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){
```

```
        speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){

        speed = speed - decrement;
    }

    public void printStates() {
         System.out.println("speed: " + speed
            + " gear: " + gear);
    }

}
class GFG {

    public static void main (String[] args) {

        // creating an inatance of Bicycle
        // doing some operations
        Bicycle bicycle = new Bicycle();
        bicycle.changeGear(2);
        bicycle.speedUp(3);
        bicycle.applyBrakes(1);

        System.out.println("Bicycle present state :");
        bicycle.printStates();

        // creating instance of the bike.
        Bike bike = new Bike();
        bike.changeGear(1);
        bike.speedUp(4);
        bike.applyBrakes(3);

        System.out.println("Bike present state :");
        bike.printStates();
    }
}
```

Output;

Bicycle present state :

speed: 2 gear: 2

Bike present state :

speed: 1 gear:

### 3.4 EXCEPTION HANDLING

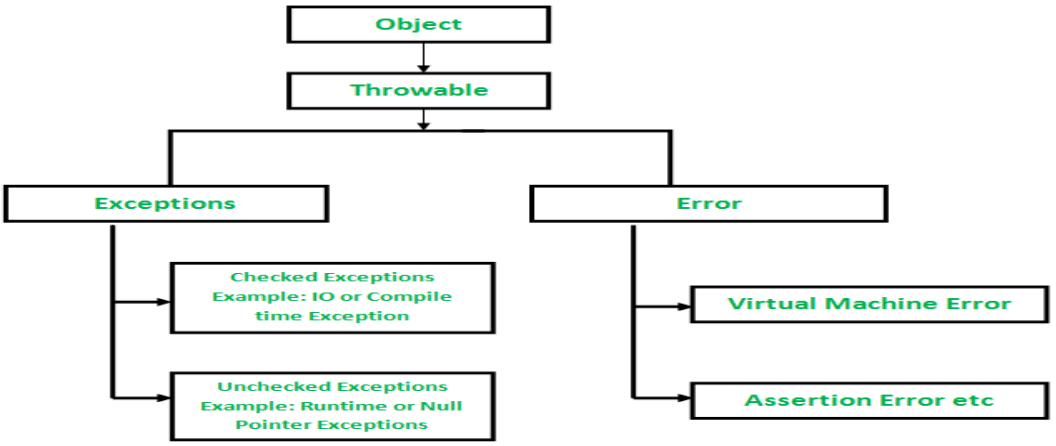
The Exception Handling in Java is one of the powerful mechanism *to handle the runtime errors* so that normal flow of the application can be maintained.In this page, we will learn about Java

exceptions, its type and the difference between checked and unchecked exceptions. Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

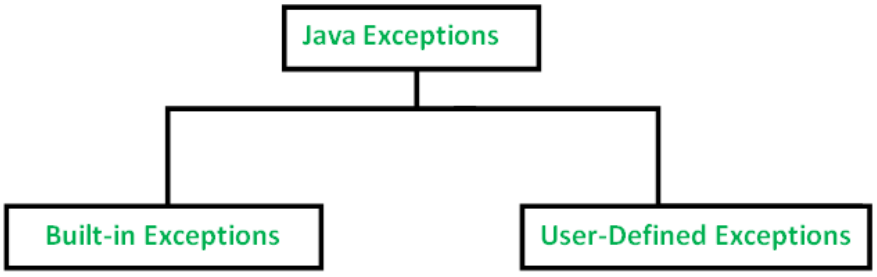
## 3.5 FUNDAMENTAL, EXCEPTION TYPES, USING TRY AND CATCH, MULTIPLE CATCH CLAUSES

### Exception Hierarchy

All exception and errors types are sub classes of class Throwable, which is base class of hierarchy. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. Null Pointer Exception is an example of such an exception. Another branch,Error are used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). Stack Overflow Error is an example of such an error.




### Types of Exception in Java -




1. **Arithmetic Exception-**It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundsException-**It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **ClassNotFoundException-**This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException-**This Exception is raised when a file is not accessible or does not open.
5. **IOException-**It is thrown when an input-output operation failed or interrupted
6. **InterruptedException-**It is thrown when a thread is waiting , sleeping , or doing some processing , and it is interrupted.

7. **NoSuchFieldException-**It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException-**It is thrown when accessing a method which is not found.
9. **NullPointerException-**This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException-**This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException-**This represents any exception which occurs during runtime.
12. **StringIndexOutOfBoundsException-**It is thrown by String class methods to indicate that an index is either negative than the size of the string

**Examples of Built-in Exception:**

**Arithmetic exception**

```
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
   public static void main(String args[])
   {
      try {
         int a = 30, b = 0;
         int c = a/b;  // cannot divide by zero
         System.out.println ("Result = " + c);
      }
      catch(ArithmeticException e) {
         System.out.println ("Can't divide a number by 0");
      }
   }
}
```

**Output:**

Can't divide a number by 0

**NullPointer Exception**

```
//Java program to demonstrate NullPointerException
class NullPointer_Demo
{
   public static void main(String args[])
   {
      try {
         String a = null; //null value
         System.out.println(a.charAt(0));
      } catch(NullPointerException e) {
         System.out.println("NullPointerException..");
      }
   }
}
```

**Output:**

NullPointerException..

**StringIndexOutOfBound Exception**

```
// Java program to demonstrate StringIndexOutOfBoundsException
class StringIndexOutOfBound_Demo
{
   public static void main(String args[])
   {
      try {
         String a = "This is like chipping "; // length is 22
         char c = a.charAt(24); // accessing 25th element
         System.out.println(c);
      }
      catch(StringIndexOutOfBoundsException e) {
         System.out.println("StringIndexOutOfBoundsException");
      }
   }
}
```

**Output:**

StringIndexOutOfBoundsException

**FileNotFound Exception**

```
//Java program to demonstrate FileNotFoundException
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
 class File_notFound_Demo {

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

         // Following file does not exist
         File file = new File("E://file.txt");

         FileReader fr = new FileReader(file);
      } catch (FileNotFoundException e) {
        System.out.println("File does not exist");
      }
   }
}
```

**Output:**

File does not exist

**NumberFormat Exception**

```
// Java program to demonstrate NumberFormatException
class  NumberFormat_Demo
{
   public static void main(String args[])
   {
      try {
         // "akki" is not a number
         int num = Integer.parseInt ("akki") ;

         System.out.println(num);
```

```
            } catch(NumberFormatException e) {
                System.out.println("Number format exception");
            }
        }
    }
```

**Output:**

Number format exception
ArrayIndexOutOfBounds Exception

```
    // Java program to demonstrate ArrayIndexOutOfBoundException
    class ArrayIndexOutOfBound_Demo
    {
        public static void main(String args[])
        {
            try{
                int a[] = new int[5];
                a[6] = 9; // accessing 7th element in an array of
                        // size 5
            }
            catch(ArrayIndexOutOfBoundsException e){
                System.out.println ("Array Index is Out Of Bounds");
            }
        }
    }
```

**Output:**

Array Index is Out Of Bounds

**User-Defined Exceptions**-Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, user can also create exceptions which are called 'user-defined Exceptions'. Following steps are followed for the creation of user-defined Exception.

The user should create an exception class as a subclass of Exception class. Since all the exceptions are subclasses of Exception class, the user should also make his class a subclass of it. This is done as:
class MyException extends Exception
We can write a default constructor in his own exception class.
MyException(){}
We can also create a parameterized constructor with a string as a parameter.
We can use this to store exception details. We can call super class(Exception) constructor from this and send the string there.

**Using try and catch-**

- Java try-catch block is used to handle exceptions in the program.
- The code in the try block is executed and if any exception occurs, catch block is used to process them.
- If the catch block is not able to handle the exception, it's thrown back to the caller program.
- If the program is not able to process the exception, it's thrown back to the JVM which terminates the program and prints the exception stack trace to the output stream.
- A try block must be followed by either catch or finally block.

**Java try-catch Example**

Let's look at a simple code where we can get an exception.

```
package com.journaldev.examples;
public class JavaTryCatch {
```

```
	public static void main(String[] args) {
		System.out.println(divide(20,5));
		System.out.println(divide(20,0));
	}

	public static double divide(int x, int y) {
		return x/y;
	}
}
4.0
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at com.journaldev.examples.JavaTryCatch.divide(JavaTryCatch.java:11)
	at com.journaldev.examples.JavaTryCatch.main(JavaTryCatch.java:7)
```

**Multiple catch clauses-**In java, when we handle more than one exceptions within a single try {} block then we can use multiple catch blocks to handle many different kind of exceptions that may be generated while running the program. However every catch block can handle only one type of exception. This mechanism is necessary when the try block has statement that raises different type of exceptions.

**The syntax for using multiple catch clause is given below:**

```
try{
	???
	???
}
catch(<exceptionclass_1><obj1>){
	//statements to handle the
	exception
}
catch(<exceptionclass_2><obj2>){
	//statements to handle the
	exception
}
catch(<exceptionclass_N><objN>){
	//statements to handle the
	exception
}
```

When an exception is thrown, normal execution is suspended. The runtime system proceeds to find a matching catch block that can handle the exception. If no handler is found, then the exception is dealt with by the default exception handler at the top level.
Let's see an example given below which shows the implementation of multiple catch blocks for a single try block.

```
public class Multi_Catch
{
	public static void main(String args[])
	{
		int array[]={20,10,30};
		int num1=15,num2=0;
		int res=0;
```

```
            try
            {
                    res = num1/num2;
                    System.out.println("The result is" +res);
                    for(int ct =2;ct >=0; ct--)
                    {
                            System.out.println("The value of array are" +array[ct]);
                    }
            }
            catch (ArrayIndexOutOfBoundsException e)
            {
                    System.out.println("Error?. Array is out of Bounds");
            }
            catch (ArithmeticException e)
            {
                    System.out.println ("Can't be divided by Zero");
            }
        }
}
```

In this example, ArrayIndexOutOfBoundsExceptionand ArithmeticExceptionare two catch clauses we have used forcatching the exception in which the statements that may cause exception are kept within the try block. When the program is executed, an exception will be raised. Now that time the first catch block is skipped and the second catch block handles the error.

### 3.6 NESTED TRY STATEMENTS, THROW, THROWS, FINALLY, JAVA'S BUILT - IN EXCEPTION, USING EXCEPTIONS

A try-catch-finally block can reside inside another try-catch-finally block that is known as nested try statement in Java. In case a try-catch block can not handle the exception thrown, the exception goes up the hierarchy (next try-catch block in the stack) and the next try statement's catch block handlers are inspected for a match. If no catch block, with in the hierarchy, handles the thrown exception then the Java run-time system will handle the exception.

**Java nested try statement example**

```
public class NestedTryDemo {
   public static void main(String[] args) {
      try{
         System.out.println("In Outer try block");
         try{
            System.out.println("In Inner try block");
            int a = 7 / 0;
         }catch (IllegalArgumentException e) {
            System.out.println("IllegalArgumentException caught");
         }finally{
            System.out.println("In Inner finally");
         }
      }catch (ArithmeticException e) {
         System.out.println("ArithmeticException caught");
      }finally {
         System.out.println("In Outer finally");
```

```
      }
    }
}
```

**Output**

```
In Outer try block
In Inner try block
In Inner finally
Arithmetic Exception caught
In Outer finally
```

In this example, one try catch finally sequence has been nested within another try catch finally sequence. The inner try block throws an Arithmetic Exception which it could not handle, as the inner catch block catches Illegal Argument Exception. Hence, this exception moves up the hierarchy and handled in the outer try-catch-finally block.

**throw, throws and finally-throw keyword-** is used to throw an exception explicitly. Only object of Throwable class or its sub classes can be thrown. Program execution stops on encountering throw statement, and the closest catch statement is checked for matching type of exception.

**throw ThrowableInstance**

Creating Instance of Throwable class

There are two possible ways to create an instance of class Throwable,

Using a parameter in catch block.

Creating instance with new operator.

new NullPointerException("test");

This constructs an instance of NullPointerException with name test.

Example demonstrating throw Keyword

```
class Test
{
  static void avg()
  {
    try
    {
      throw new ArithmeticException("demo");
    }
    catch(ArithmeticException e)
    {
      System.out.println("Exception caught");
    }
  }

 public static void main(String args[])
 {
    avg();
 }
}
```

In the above example the avg() method throw an instance of ArithmeticException, which is successfully handled using the catch statement and thus, the program outputs "Exception caught".

**Throws Keyword**
Any method that is capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions are to be handled. A method can do so by using the throws keyword.

Syntax:

```
type method_name(parameter_list) throws exception_list
{
  // definition of method
}
```

**Java's Built in exception-**Java defines several exception classes inside the standard package java.lang.
The most general of these exceptions are subclasses of the standard type RuntimeException. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available.
Java defines several other types of exceptions that relate to its various class libraries. Following is the list of Java Unchecked RuntimeException.

| Sr.No. | Exception & Description |
|---|---|
| 1 | **ArithmeticException**<br>Arithmetic error, such as divide-by-zero. |
| 2 | **ArrayIndexOutOfBoundsException**<br>Array index is out-of-bounds. |
| 3 | **ArrayStoreException**<br>Assignment to an array element of an incompatible type. |
| 4 | **ClassCastException**<br>Invalid cast. |
| 5 | **IllegalArgumentException**<br>Illegal argument used to invoke a method. |
| 6 | **IllegalMonitorStateException**<br>Illegal monitor operation, such as waiting on an unlocked thread. |
| 7 | **IllegalStateException**<br>Environment or application is in incorrect state. |
| 8 | **IllegalThreadStateException**<br>Requested operation not compatible with the current thread state. |
| 9 | **IndexOutOfBoundsException**<br>Some type of index is out-of-bounds. |
| 10 | **NegativeArraySizeException**<br>Array created with a negative size. |

| | |
|---|---|
| 11 | **NullPointerException**<br>Invalid use of a null reference. |
| 12 | **NumberFormatException**<br>Invalid conversion of a string to a numeric format. |
| 13 | **SecurityException**<br>Attempt to violate security. |
| 14 | **StringIndexOutOfBounds**<br>Attempt to index outside the bounds of a string. |
| 15 | **UnsupportedOperationException**<br>An unsupported operation was encountered. |

## 3.7 SUMMARY

A java package is a group of similar types of classes, interfaces and sub-packages.
Package in java can be categorized in two form, built-in package and user-defined package.
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Exception Handling is a mechanism to handle runtime errors such as Class Not Found Exception, IO Exception, SQL Exception, Remote Exception, etc. The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application that is why we use exception handling

## 3.8 SELF ASSESSMENT QUESTION

1. Explain access protection
2. Explain importing packages
3. Explain implementing interfaces
4. Explain exception using try and catch
5. Explain throws, finally
6. Explain Built - in exception

## 3.9 SUGGESTED READINGS

1. The complete reference Java –2: V Edition By Herbert Schildt Pub. TMH.
2. SAMS teach yourself Java – 2: 3rd Edition by Rogers Cedenhead and Leura Lemay Pub. Pearson Education
3. Java: A Beginner's Guide by Herbert Schildt
4. Java: Programming Basics for Absolute Beginners by Nathan Clark.