MODULE-4

Packages: Defining Package, Built in packages, accessing Packages, Creating packages, accessing Protection. **Exception Handling**: Exception handling Fundamentals, exception types, Built-in Exceptions, Using try-catch-finally throw-throws keywords, creating your own Exceptions.

PACKAGES

A package represents a directory that contains related group of classes and interfaces. For example, when we write statements like: **import java.io.*;**
We are importing the classes of **java.io** package. Here, java is a directory name, and **io** is another sub directory within it. And the '*' represents all the classes and interfaces of that **io** sub directory.

Advantages of packages:

1. Packages are useful to arrange related classes and interfaces into a group. This makes all the classes and interfaces performing the same task to put together in the same package. For example, in java, all the classes and interfaces which perform input and output operations are stored in **java.io** package.
2. Packages hide the classes and interfaces in a separate sub directory, so that accidental deletion of classes and interfaces will not take place.
3. The classes and interfaces of a package are isolated from the classes and interfaces of another package. This means that we can use same names for classes of two different classes. For example, there is **Date** class in **java.util** package and also there is another **Date** class available in **java.sql** package.
4. A group of packages is called a library. The classes and interfaces of a package are like books in a library and can be reused several times. This reusability nature of packages makes programming easy.

Different types of packages:

There are 2 different types of packages in java. They are:

1. Built – in packages / pre – defined packages
2. User – defined packages

Built – in / pre – defined packages

These are the packages which are already available in java language. These packages provide all most all necessary classes, interfaces and methods for the programmer to perform any task in his programs.

1. **java.lang:** lang stands for language. This package got primary classes and interfaces essential for developing a basic java program. It consists wrapper classes, String, StringBuffer, Thread and Runtime and System classes.
2. **java.util:** util stands for utility. This package contains useful classes and interfaces like Stack, LinkedList, Hashtable, Arrays, Vector etc. These classes are called collections. There are also classes for handling date and time operations.
3. **java.io:** io stands for input and output. This package contains streams are useful to store data in the form of files and also to perform input – output related tasks.
4. **java.awt:** awt stands for abstract window toolkit. This package helps to develop GUI where programs with colorful screens, paintings and images etc. can be developed. It consists an important sub package, **java.awt.event,** which is useful to provide action for components like push buttons, radio buttons, menus etc.
5. **javax.swing:** this package helps to develop GUI like java.awt. The 'x' in javax represents that it is an extended package which means it is a package developed from another package by adding new features to it. javax.swing package is an extended package of java.awt.
6. **java.net:** net stands for network. Client – Server programming can be done by using this package.
7. **java.applet:** Applets are programs which come from a server into a client and get executed on the client machine on a network. **Applet** class of this package is useful to create and use applets.
8. **java.text:** this package has 2 important classes, **DateFormat** to format dates and times, and **NumberFormat** which is useful to format numeric values.

9. **java.sql:** sql stands for structured query language. This package helps to connect to databases like Oracle, Sybase, retrieve the data from them and use it in java program.

User – defined packages: Just like the Built – in packages, the users of the java language can also create their own packages. They are called user – defined packages. User – defined packages can also be imported into other classes and used exactly in the same way as the built – in packages.

Defining / creating a user defined package

To create a package of our own and use it in any other class. To create a package the keyword **package used** as:

package packagename; //to create a package

package packagename.subpackagename //to create a sub package within a package

The preceding statements in fact create a directory with the given package name. We should add our classes and interfaces to this directory. While creating the class in a package to declare all the members and the class itself as **public,** except the instance variables. The reason is only that the public members are available outside the package to other programs.

E.g.:

```
//step1: creating a package pack with Addition class
package pack; //pack is the package name
public class Addition
{
private double d1,d2;
public Addition(double a,double b)
{
d1=a;
d2=b;
}
public void sum()
{
System.out.println("sum=" +(d1+d2));
}
}
```

O/P: **G:\>javac -d . Addition.java**

G:\>

To compile a java program, we have written as:

javac –d . Addition.java

The **–d** option tells the java compiler to create a separate sub directory and place the .class file there. The **dot (.)** operator after **–d** indicates that the package should be created in the current directory. So the java compiler creates a directory in **G:** with name as **pack** and stores **Addition.class** there.

So, our package with **Addition** class is ready. The next step is to use the **Addition** class and its **sum()** method in a program. For this purpose, we write another class **Use**. In this program, we can refer to the **Addition** class of package **pack** using membership operator (.) as:

p**ack.Addition**

Now, to create an object to **Addition** class, we can write as:

pack.Addition obj=new pack.Addition(10,15.5);

E.g.:

```
// step2: using the package pack
class Use
{
public static void main(String args[])
{
//create Addition class object
pack.Addition obj=new pack.Addition(10,15.5);
obj.sum();
}
}
```

O/P: **G:\>javac Use.java**

G:\>java Use

sum=25.5

Accessing / importing a package

Every time we refer to a class of a package, we should write the package name before the class name as **pack.Addition**. This is inconvenient for the programmer. To overcome this, we can use import statement only once in the beginning of the program, as:

import pack.Addition;

Once the import statement is written, we need not to use the package name before the class name and we can create the object to Addition class, in a normal way as:

Addition obj=new Additon(10,15.5);

E.g.:

```
// step2: using the package pack
import pack.Addition;
class Use
{
public static void main(String args[])
{
//create Addition class object
Addition obj=new Addition(10,15.5);
obj.sum();
}
}
```

O/P: **G:\>javac Use.java**
G:\>java Use
sum=25.5

Similarly, we can add another class **Subtraction** with **sub()** method. To add this class to the package, the same procedure should be repeated.

E.g.:

```
//adding another class to the package: pack
package pack; //pack is the package name
public class Subtraction
{
private double d1,d2;
public Subtraction(double a,double b)
{
d1=a;
d2=b;
}
public void sub()
{
System.out.println("sub=" +(d1-d2));
}
}
```

O/P: **G:\>javac -d . Subtraction.java**
G:\>

Here, the java compiler checks whether the package with the name **pack** already exists or not. If it is existing, then it adds **Subtraction.class** to it. If the package **pack** does not exist then java compiler creates a sub directory with the name **pack** in the current directory and adds **Subtraction.class** to it.

E.g.:

```
//using the package pack
import pack.Addition;
import pack.Subtraction;
class Use
{
public static void main(String args[])
{
Addition obj=new Addition(10,15.5);
```

```
obj.sum();
Subtraction obj1=new Subtraction(10,15.5);
obj1.sub();
}
}
```

O/P: **G:\>javac Use.java**
G:\>java Use
sum=25.5
sub=-5.5

When we want to use classes of the same package, we need not write separate import statements. We can write a single import statement as:

import pack.*;

Here, '*' represents all the classes and interfaces of a particular package.

In this case, please be sure that any of the **Addition.java** and **Subtraction.java** programs will not exist in the current directory. Delete them from the current directory as they cause confusion for the java compiler.

E.g.:

```
//using the package pack
import pack.*;
class Use
{
public static void main(String args[])
{
Addition obj=new Addition(10,15.5);
obj.sum();
Subtraction obj1=new Subtraction(10,15.5);
obj1.sub();
}
}
```

O/P: **G:\>javac Use.java**
G:\>java Use
sum=25.5
sub=-5.5

CLASS PATH

Suppose our program is running in **D:\JAVAMCA** and the package **pack** is available in the directory **E:\Chakry**. In this case, the compiler should be given information regarding the package location by mentioning the directory name of the package in class path.

Class path represents an operating system's environment variable which stores active directory path such that all the files in those directories are available to any programs in the system. Now our package **pack** exists in **E:\Chakry.** This information should be provided to the java compiler by setting the class path to **E:\Chakry,** as shown here:

D:\JAVAMCA>set classpath=E:\Chakry;.;%classpath%

Then execute our program which is in **D:\JAVAMCA,** by typing:

D:\JAVAMCA>javac program name.java;
D:\JAVAMCA>java Class name;

Alternatively, you can mention the class path at the time of executing the program at command line using **-cp** option, as:

D:\JAVAMCA>javac –cp E:\Chakry;. Program name.java;
D:\JAVAMCA>java –cp E:\Chakry;. Class name;

Remember, here our java program is in the current directory and the package **pack** is available in **E:\Chakry** directory. So, it is possible to use the package by setting the class path to **E:\Chakry;.** and execute the program.

Interfaces in a package

It is also possible to write interfaces in a package. But whenever, we create an interface the implementation classes are also should be created. We cannot create an object to the interface but we can create objects for implementation classes and use them.

E.g.:

```
//create MyDate interface in the package mypack
package mypack;
public interface MyDate
{
void showDate();
}
```

O/P: **G:\>javac -d . MyDate.java**
G:\>

Compile the preceding code and observe that the java compiler creates a sub directory with the name **mypack** and stores **MyDate.class** file there. The next step is to create implementation class for **MyDate** interface.

E.g.:

```
//this is the implementation class of MyDate interface
package mypack; //store DateImpl class also in mypack
import mypack.MyDate;
import java.util.*;
public class DateImpl implements MyDate
{
public void showDate()
{
//Date class object by default stores system date and time
Date d=new Date();
System.out.println(d);
}
}
```

O/P: **G:\>javac -d . DateImpl.java**
G:\>

When the preceding code is compiled, **DateImpl.class** file is created in the same package **mypack**. **DateImpl** class contained **showDate()** method which can be called and used in any other program.

E.g.:

```
//using the DateImpl of mypack
import mypack.DateImpl;
class DateDisplay
{
public static void main(String args[])
{
//create DateImpl object
DateImpl obj=new DateImpl();
//call showDate
obj.showDate();
}
}
```

O/P: **G:\>javac DateDisplay.java**
G:\>java DateDisplay
Sat Jan 04 22:55:23 IST 2014

An alternative approach where **MyDate** interface reference can be used to refer the **DateImpl** class object to access all the methods of the **DateImpl** class.

E.g.:

```
import mypack.MyDate;
import mypack.DateImpl;
```

```
class DateDisplay
{
public static void main(String args[])
{
/*MyDate interface reference is used to
refer to DateImpl object*/
MyDate obj=new DateImpl();
//call showDate
obj.showDate();
}
}
```

O/P: **G:\>javac DateDisplay.java**
G:\>java DateDisplay
Sat Jan 04 22:58:16 IST 2014

Creating sub package in a package

We can create sub package in a package in the format:

package pack1.pack2;

Here, we are creating **pack2** which is created in **pack1**. To use the classes and interfaces of **pack2**, we can write import statement as:

import pack1.pack2;

E.g.:

```
//creating a sub package tech in the package dream
package dream.tech;
public class Sample1
{
public void show()
{
System.out.println("Welcome to dream tech");
}
}
```

O/P: **G:\>javac -d . Sample1.java**
G:\>

When the preceding program is compiled, the java compiler creates a sub directory with the name **dream.** Inside this, there would be another sub directory with the name **tech** is created. In this **tech** directory, **Sample1** class is created. Suppose the user wants to use the **Sample1** class of **dream.tech** package, he can write a statement as:

import dream.tech.Sample1;

E.g.:

```
//using the package dream.tech
import dream.tech.Sample1;
class Use
{
public static void main(String args[])
{
Sample1 s=new Sample1();
s.show();
}
}
```

O/P: **G:\>javac Use.java**
G:\>java Use
Welcome to dream tech

Exception Handling

What is an Exception?

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time that disrupts the normal flow of the program's instructions.

Error vs Exception

- **Error:** An Error indicates serious problem that a reasonable application should not try to catch.
- **Exception:** Exception indicates conditions that a reasonable application might try to catch.

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. NullPointerException 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). StackOverflowError is an example of such an error.

There are basically 3 types of errors in the java program:

1. Compile – time errors: These are syntactical errors found in the code, due to which a program fails to compile. For example, forgetting a semicolon at the end of a java statement, or writing a statement without proper syntax will result in compile – time error.

E.g.:

```
//compile - time error
class Error
{
public static void main(String a[])
{
System.out.println("Hai NECN")
System.out.println("Hello MCA")
}
}
```

O/P: F:\JAVA>javac Error.java

```
compile.java:6: error: ';' expected
System.out.println("Hai NECN")
                   ^
compile.java:7: error: ';' expected
System.out.println("Hello MCA")
                   ^
2 errors
```

In this program, we are not writing a semicolon at the end of the statements. This is will be detected by the java compiler.

2. Run – time errors: These errors represent inefficiency of the computer system to execute a particular statement. For example, inefficient memory to store something or inability of the microprocessor to execute some statement come under run – time errors.

E.g.:

```
//run - time error
class Error
{
public static void main()
{
```

```
System.out.println("Hello");
System.out.println("where is error");
}
}
```

O/P: F:\JAVA>javac Error.java

F:\JAVA>java Error
Error: Main method not found in class Error, please define the main method as:
public static void main(String[] args)

Run – time errors are not detected by the java compiler. They are detected by the JVM, only at runtime.

3. Logical errors: These errors represent error in the logic of the program. The programmer might be using a wrong formula or the design of the program itself is wrong. Logical errors are not detected either by a java compiler or JVM. The programmer is only responsible for them.

E.g.:

```
//logical error
class Error
{
public static void main(String args[])
{
double sal=5000.00;
sal = sal * 15/100;      // wrong. Use: sal+=sal*15/100;
System.out.println("Incremented salary= "+sal);
}
}
```

O/P: F:\JAVA>javac Error.java
F:\JAVA>java Error
Incremented salary= 750.0

By comparing the output of a program with manually calculated results, a programmer can guess the presence of a logical error.

Benefits of Exception Handling

1. Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run – time characteristics.
2. By handling exceptions, a programmer can make his programs 'robust'. It means the java program will not abnormally terminate if all the possible exceptions have been handled properly.
3. To provide user understandable messages when exception is raised, so that user can take decision without developer's help.

THE CLASSIFICATION OF EXCEPTIONS – EXCEPTION HIERARCHY

If the source code is syntactically incorrect, java compiler reports compilation errors. During program execution, if run – time environment rules are violated, JVM reports run – time errors known as exceptions. Object oriented representation of an abnormal event occurred during program execution is known as an exception. When an exception is raised program gets terminated abnormally. Abnormal termination of the program causes the following problems.

1. End user loses previous work and data.
2. End user doesn't have proper information about what went wrong.
3. Any resources allocated to the application cannot be resubmitted.

Exception is thrown by the JVM and is caught by the handler. Pre – created exception classes are known as **standard exceptions.** Our own created exceptions are known as **user defined exceptions.** In java,

exception is an object that is an instance of some sub class of java.lang.Throwable. java.lang.Throwable class is the super class of all exceptions. It has two main sub classes:

1. Error
2. Exception

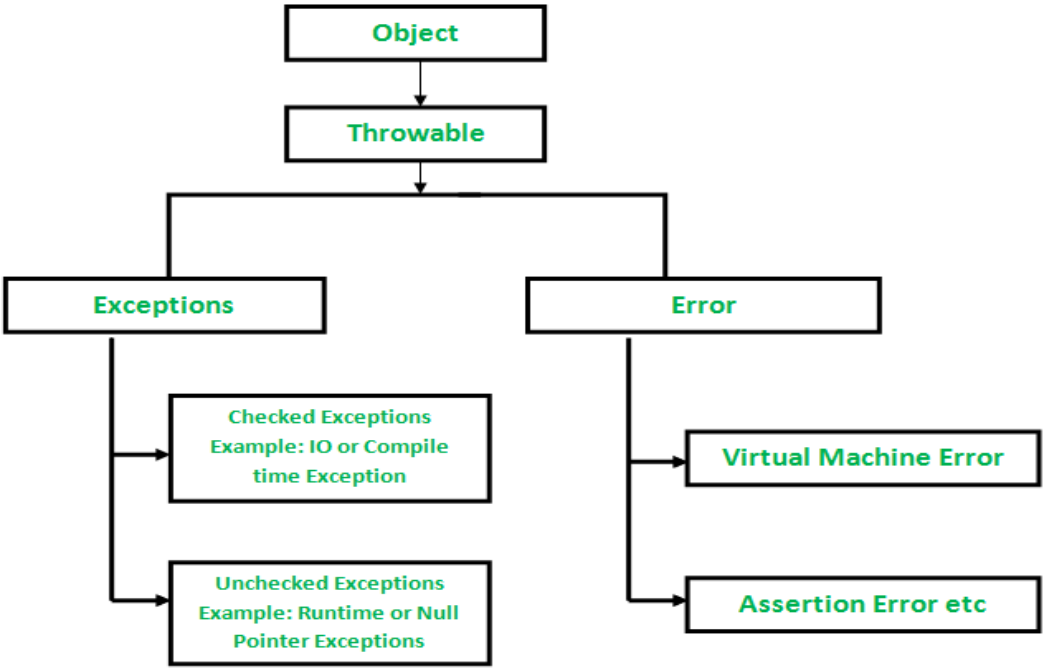
The differences between error and exception:

1. Error type exceptions are thrown due the problem occurred inside JVM logic, like
   - If there is no memory in JSA to create, new stack frame to execute method, JVM process is killed by throwing Error type exception "java.lang.StackOverFlowError".
   - If there is no memory HA to create, new object, then JVM process is killed by throwing Error type exception "java.lang.OutOfMemoryError".

   Exception type exceptions are thrown due to the problem that occurred in java program logic execution like,
   If we divide an integer number with zero, then JVM terminates the program execution by throwing Exception type exception "java.lang.ArithmeticException".
2. We cannot catch Error type exception, because Error type exception is not thrown in our application, and once this Error type exception is thrown JVM is terminated.

   We can catch Exception type exceptions, because Exception type exception is thrown in our application, and more over JVM is not directly terminated. JVM is terminated only if the thrown exception is not caught.

The following diagram shows "java.lang" package exception classes hierarchy.



CHECKED EXCEPTIONS AND UNCHECKED EXCEPTIONS

java.lang.Exception class has 2 kinds of sub classes:

1. Checked exception classes: The exceptions that are checked at compilation time by the java compiler are called 'checked exceptions'. The checked exceptions are recognized by the compiler. In the exception hierarchy, except the RuntimeException and its sub classes all other sub classes of Exception class are called checked exceptions.

E.g.: FileNotFoundException, IOException, InterruptedException etc.

Let us now consider a statement.

public static void main(String a[])throws IOException

Here, IOException is an example for checked exception. So we threw it out of main() method without handling it. We can also handle it. Suppose, we are not handling it and not even throwing it, then the java compiler will give an error.

2. Unchecked exception classes: The exceptions that are checked by the JVM are called 'unchecked exceptions'. In the exception hierarchy, the RuntimeException and its sub classes are called as unchecked exceptions.

E.g.: NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException etc.

```
//An exception example
class Ex
{
public static void main(String args[])
{
System.out.println("open the files");
int n=args.length;
System.out.println("n= "+n);
int a=45/n;
System.out.println("a= "+a);
System.out.println("close the files");
}
}
```

O/P: F:\JAVA>javac Ex.java
F:\JAVA>java Ex 1 2 3
open the files
n= 3
a= 15
close the files

F:\JAVA>java Ex
open the files
n= 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Ex.main(exception.java:10)

Please observe the output of this program. When we passed 3 command line arguments, then the program executed without any problem. But when we run the program without passing any arguments, then n value become zero. So there will be an exception at runtime. In this case, JVM displays exception details and then terminates the program abnormally. The sub sequent statements in the program are not executed. When there is an exception, the files may not be closed, or the threads may abnormally terminate or the memory may not be freed properly.

EXCEPTION HANDLING

If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions by using below keywords. This task is known as Exception handling.

Keyword	Description
Try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
Catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
Finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
Throw	The "throw" keyword is used to throw an exception.
Throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

The mechanism suggests incorporation of a separate error handling code that performs the following tasks.
(1) Find the problem (Hit the exception)
(2) Inform that an error has occurred (Throw the exception)
(3) Receive the error information (Catch the exception)
(4) Take corrective actions (Handle the exception)
The error handling code basically consists of two segments, one to detect errors and to throw exceptions and the other to catch exceptions and to take appropriate actions.

The process of catching the exception for converting JVM given exception message to programmer understandable message or for stopping abnormal termination of the program is called exception handling. For this, we should perform the following steps:

- The programmer should observe the statements in his program where they may be a possibility of exceptions. Such statements should be written inside a **try** block. A **try** block looks like as follows:

```
try
{
        statements;
}
```

The greatness of **try** block is that even if some exception arises inside it, the program will not be terminated. When JVM understands that there is an exception, it stores the exception details in an exception stack and then jumps into a catch block.
The programmer should write the **catch** block where he should display the exception details to the user. This helps the user to understand that there is some error in the program. The programmer should also display a message regarding what can be done to avoid this error. **Catch** block looks like as follows:

```
catch(Exception class ref)
{
        Statements;
}
```

The reference ref is automatically refer to the exception stack where the details of the exception are available. So, we can display the exception details using any one of the following ways:

1. Using print() or println() methods, such as System.out.println(ref);
2. Using printStackTrace() method of **Throwable** class, which fetches exception details from the exception stack and displays them.

Lastly, the programmer should perform cleanup operations like closing the files and terminating the threads. The programmer should write this code in the **finally** block like as follows:

```
finally
{
        Statements;
}
```

The specialty of the **finally** block is that the statements inside the **finally** block are executed irrespective of whether there is an exception or not. This ensures that all the opened files are properly closed and all the running threads are properly terminated.
Performing the above tasks is called ‘exception handling’.
E.g.:

```
//Exception handling using try, catch and finally blocks
class Ex
{
public static void main(String args[])
{
try
{
System.out.println("open the files");
```

```
int n=args.length;
System.out.println("n= "+n);
int a=45/n;
System.out.println("a= "+a);
}

catch(ArithmeticException ae)
{
//display the exception details
System.out.println(ae);
System.out.println("please pass data while running this program");
}
finally
{
//close the files
System.out.println("close files");
}
}
}
```

O/P: F:\JAVA>javac Ex.java
F:\JAVA>java Ex
open the files
n= 0
java.lang.ArithmeticException: / by zero
please pass data while running this program
close files

try:
It is a keyword used to create a block of statements. In our java application what ever code is doubtful for raising exceptions, that code is placed in the try block. A **try** block looks like as follows:

```
try
{
        statements;
}
```

The greatness of **try** block is that even if some exception arises inside it, the program will not be terminated. When JVM understands that there is an exception, it stores the exception details in an exception stack and then jumps into a catch block.

catch:
It is a keyword used to create a block of statements. Catch block is a method block. In the catch block we write exception handling code. So, catch block is the exception handler. The programmer should write the **catch** block where he should display the exception details to the user. This helps the user to understand that there is some error in the program. The programmer should also display a message regarding what can be done to avoid this error. **Catch** block looks like as follows:

```
catch(Exception class ref)
{
        Statements;
}
```

The reference ref is automatically refer to the exception stack where the details of the exception are available. So, we can display the exception details using any one of the following ways:

1. Using print() or println() methods, such as System.out.println(ref);
2. Using printStackTrace() method of **Throwable** class, which fetches exception details from the exception stack and displays them.

- Between try block and catch block no other statement is allowed.

- One try block can have any number of catch blocks.
- If exception is not raised in the try block, control never goes to catch block.
- If exception is raised in the try block, try block gets terminated instead of program getting terminated.
- JVM calibrates the kind of abnormal event, instantiates the appropriate exception class object and throws it. Catch block is implicitly called.
- To the catch block the reference of the exception object is supplied as argument. Application user can use that reference in the catch block to find where about us and what about us of the exception.

E.g.:

```
//Exception handling using try, catch and finally blocks
class Ex
{
public static void main(String args[])
{
try
{
System.out.println("open the files");

int n=args.length;
System.out.println("n= "+n);
int a=45/n;
System.out.println("a= "+a);
}
catch(ArithmeticException ae)
{
//display the exception details
System.out.println(ae);
System.out.println("please pass data while running this program");
}
}
}
```

O/P: F:\JAVA>javac Ex.java

F:\JAVA>java Ex 1 2 3
open the files
n= 3
a= 15

F:\JAVA>java Ex
open the files
n= 0
java.lang.ArithmeticException: / by zero
please pass data while running this program

throw:

It is a keyword used to throw exception explicitly. We need to create the exception object and throw it. Generally JVM creates the exception object and throws it.

Syntax:

```
someexception e=new someexception();
throw e;
```

E.g.:

```
//using throw
```

```
class Throw
{
static void demo()
{
try
{
System.out.println("Inside demo()");
throw new NullPointerException("Exception data");
}
catch(NullPointerException ne)
{
System.out.println(ne);
}
}
}
class ThrowDemo
{
public static void main(String args[])
{
Throw.demo();
}
}
```

O/P: F:\JAVA>javac ThrowDemo.java

F:\JAVA>java ThrowDemo
Inside demo()
java.lang.NullPointerException: Exception data

throws:

This keyword is used to pass exceptions to the next level instead of handling it in a method. This keyword is associated with method declaration. For a method we can associate any number of exceptions with throws clause. It is associated with a method declaration means the method has doubtful code but not handled it. It is the responsibility of the calling method to handle the exceptions.

Syntax:

```
Return type mehod() throws exception1, exception2, exception3
{
        //body of the method
}
```

Even if the programmer is not handling runtime exceptions, the java compiler will not give any error related to runtime exceptions. But the rule is that the programmer should handle checked exceptions. Incase the programmer does not want to handle the checked exceptions; he should throw them out using **throws** clause. Otherwise, there will be an error flagged by java compiler.

E.g.:

```
//Not handling the exception
import java.io.*;
class Sample
{
private String name;
void accept()
{
BufferedReader br=new BufferedReader (new InputStreamReader(System.in));
System.out.println("enter name: ");
name=br.readLine();
}
void display()
```

```
{
System.out.println("Name: "+name);
}
}
class Ex1
{
public static void main(String args[])
{
Sample s=new Sample();
s.accept();
s.display();
}
}
```

O/P: F:\JAVA>javac Ex1.java
throws.java:11: error: unreported exception IOException; must be caught or declared to be thrown
name=br.readLine();
^
1 error

In this program, the java compiler expects the programmer to handle the **IOException** using **try** and **catch** blocks; else, he should throw out the **IOException** without handling it. But the programmer is not performing either. So there is a compile time error displayed.

```
//Not handling the exception
import java.io.*;
class Sample
{
private String name;
void accept()throws IOException
{
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
System.out.println("enter name: ");
name=br.readLine();
}
void display()
{
System.out.println("Name: "+name);
}
}
class Ex1
{
public static void main(String args[])throws IOException
{
Sample s=new Sample();
s.accept();
s.display();
}
}
```

O/P: F:\JAVA>javac Ex1.java

F:\JAVA>java Ex1
enter name:
DRJSB
Name: DRJSB

finally:

It is a keyword used to create a block of statements that execute definitely in both the cases i.e. in exception generated case and no generated case also. It cannot be written individually. It succeeds either try block or catch block.

Syntax:
```
finally
{
        Statements;
}
```

E.g.:
```
//Exception handling using try, catch and finally blocks
class Ex
{
public static void main(String args[])
{
try
{
System.out.println("open the files");

int n=args.length;
System.out.println("n= "+n);
int a=45/n;
System.out.println("a= "+a);
}
catch(ArithmeticException ae)
{
//display the exception details
System.out.println(ae);
System.out.println("please pass data while running this program");
}
finally
{
//close the files
System.out.println("close files");
}
}
}
```

O/P: F:\JAVA>javac Ex.java

F:\JAVA>java Ex
open the files
n= 0
java.lang.ArithmeticException: / by zero
please pass data while running this program
close files

RETHROWING EXCEPTIONS

When an exception occurs in a try block, it is caught by a catch block. This means that the thrown exception is available to the catch block. The following code shows how to re-throw the same exception out from the catch block.

```
try
{
throw exception;
}
catch(Exception obj)
{
```

```
throw exception; //re-throw the exception out
}
```

Suppose there are two classes A and B. If an exception occurs in A, we want to display some message to the user and then we want to re-throw it. This re-thrown exception can be caught in class B where it can be handled. Hence re-throwing exceptions is useful especially when the programmer wants to propagate the exception details to another class. In this case the exception details are sent from class A to class B where some appropriate action may be performed.

E.g.:

```
//rethrowing an exception
class A
{
void method1()
{
try
{
//take a string with 5 chars. Their index is from 0 to 4
String str="Hello";
//exception is thrown in below statement because there is no index with value 5
char ch=str.charAt(5);
}
catch(StringIndexOutOfBoundsException sie)
{
System.out.println("Please see the index is within the range");
throw sie; //rethrow the exception
}
}
}
class B
{
public static void main(String args[])
{
//create an object to A and call method1()
A a=new A();
try
{
a.method1();
}
//the rethrown exception is caught by the below catch block
catch(StringIndexOutOfBoundsException sie)
{
System.out.println("I caught rethrown exception");
}
}
}
```

O/P: F:\JAVA>javac B.java
F:\JAVA>java B
Please see the index is within the range
I caught rethrown exception

TYPES OF EXCEPTIONS

Following are the exceptions available in java:

1. Built – in exceptions
2. User – defined exceptions

1. Built – in Exceptions:

Built – in exceptions are the exceptions which are available in java. These exceptions are suitable certain error situations. The following table lists the important built – in exceptions:

Exception class	Meaning
ArithmeticException	Thrown when an exceptional condition has occurred in an arithmetic operation.
ArrayIndexOutOfBoundsException	Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or equal to the size of the array.
ClassNotFoundException	This exception is raised when we try to access a class whose definition is not found.
FileNotFoundException	Raised when a file is not accessible or does not open
IOException	Thrown when an input – output operation failed or interrupted.
InterruptedException	Thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
NoSuchFieldException	Thrown when a class does not contain the field or variable specified.
NoSuchMethodException	Thrown when accessing a method which is not found.
NullPointerException	Raised when referring to the members of null object.
NumberFormatException	Raised when a method could not convert a string into a number format.
RuntimeException	This represents any exception which occurs doing runtime.
StringIndexOutOfBoundsException	Thrown by String class methods to indicate that an index is either negative or greater than the size of the string.

User – defined exceptions / creating own exception sub classes:

Sometimes, the built – in exceptions in java are not able to describe a certain situation. In such cases, like the built – in exceptions, the programmer can also create his own exceptions which are called user – defined exceptions. Sometimes input is given to the application may not violate runtime rules but it violates my application business rules. Such violation of business rule JVM will not treat as an abnormal event and hence it will not throw any exception. As a developer we want to stop the task processing as the business rule is violated. The best way to do so is throw the user defined exception explicitly. In order to create user defined exception class, the role is that the class must extends java.lang.Exception or its sub class. The following steps are followed in creation of user – defined exceptions:

- The user should create an exception class as a subclass to Exception class. Since all exceptions are subclasses of Exception class, the user should make his class a subclass to it. This is done as:

  class MyException extends Exception

- The user can write a default constructor in his own exception class. He can use it, in case he does not want to store any exception details. If the user does not want to create an empty object to his exception class, he can eliminate writing the default constructor.

  MyException() {}

- The user can create a parameterized constructor with a string as a parameter. He can use this to store exception details. He can call super class (Exception) constructor from this and send the string there.

```
MyException(String str)
    {
    Super(str);
    }
```

- When the user wants to raise his own exception, he should create an object to his exception class and throw it using throw clause, as:

```
MyEXception me=new MyException("Exception details");
throw me;
```

Ex:- (1) throw new ArithmeticException ( ):
(2) throw new NumberFormatException ( ):

The following program demonstrates the use of a user defined sub class of throwable class. Here exception is a sub class of throwable and therefore My Exception is a sub class of throwable class.

Example program:-

```
class MyExp extends Exception
{
   MyExp (String message )
  {
            super(message)
  }
}

class Test
{
   public static void main(String ar[ ])throw IOException
  {
int x=5,y=1000;
    try
    {
   float z=float(x)/(float)y;
        if(z<0.01)
       {
throw new MyExp("Number is too small ");
        }
    }
    catch(MyExp e)
  {
        System.out.println("caught my exception");
        System.out.println(e.getMessage());
  }
 finally{
      System.out.println("Iam always here");
}
     }
 }
```

/* Define an exception called "Marks Out Of Bound" Exception,
that is thrown if the entered marks are greater than 100.*/

```
import java.io.*;
class MyException extends Exception
{
 MyException(String message)
 {
 super(message);
 }
}
class UserExcep
{
 public static void main(String[] args)throws Exception
 {
 DataInputStream dis=new DataInputStream(System.in);
 System.out.println("Enter name of the student ");
 String name=dis.readLine();
```

```
 int i,sum=0,j;
 int[] a=new int[6];
 System.out.println("Enter six subject marks ");
  for(j=0;j<6;j++)
  {
    try
     {
     a[j]=Integer.parseInt(dis.readLine());
      if(a[j]>100 || a[j]<0)
       {
        throw new MyException("wrong data");
       }
     }
   catch(MyException e)
   {
    System.out.println(e.getMessage());
    System.out.println("please enter marks which are below 100");
    System.out.println("enter correct marks");
    a[j]=Integer.parseInt(dis.readLine());
   }
  sum=a[j]+sum;
  }
  System.out.println("marks of students are");
  for(i=0;i<6;i++)
  {
  int k=i+1;
  System.out.println("marks of "+k+" subject is="+a[i]);
  }
   System.out.println("name of student is="+name);
   System.out.println("total is="+sum);

 }
}
```

****