UNIT 3
INTERFACE, PACKAGES AND EXCEPTION HANDLING

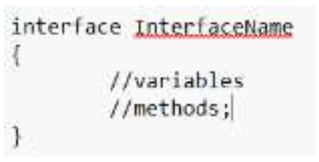
Interface

- Interfaces is similar to a class (or) it looks like a class but it is not a class.
- An interface can have methods & variables just like a class. but the methodis declared in interface are by default public and abstract (only method signature no body). also the variables declared in interface are public, Static & final by default.

Uses:

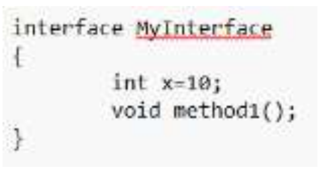
- ✓ It is used to achieve abstraction. Since Methods in interface donot have any body, they have to be implemented by class before you can access them.
- ✓ By interface we can support functionality of multiple inheritance

Syntax:



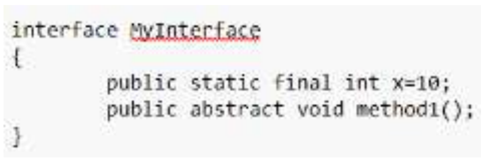
```
interface InterfaceName
{
        //variables
        //methods;
}
```

Ex: After



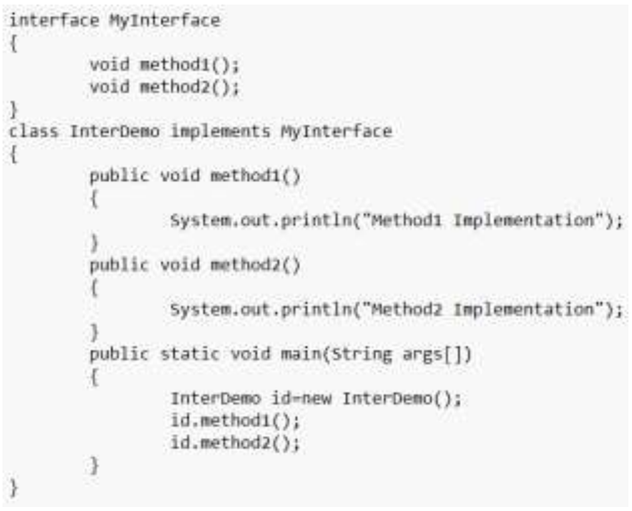
```
interface MyInterface
{
        int x=10;
        void method1();
}
```

Comilation

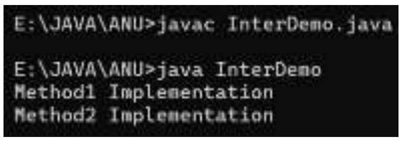


```
interface MyInterface
{
        public static final int x=10;
        public abstract void method1();
}
```

Example program

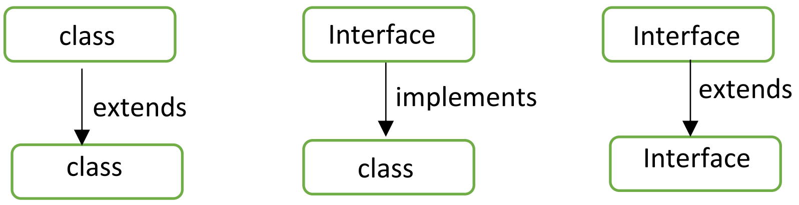


```
interface MyInterface
{
        void method1();
        void method2();
}
class InterDemo implements MyInterface
{
        public void method1()
        {
                System.out.println("Method1 Implementation");
        }
        public void method2()
        {
                System.out.println("Method2 Implementation");
        }
        public static void main(String args[])
        {
                InterDemo id=new InterDemo();
                id.method1();
                id.method2();
        }
}
```



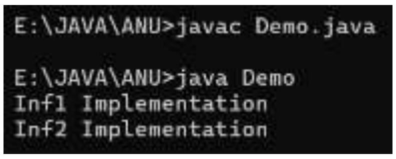
```
E:\JAVA\ANU>javac InterDemo.java

E:\JAVA\ANU>java InterDemo
Method1 Implementation
Method2 Implementation
```

Interface and inheritance



Program on Extending Interface

```
interface Inf1
{
        void method1();
}
interface Inf2 extends Inf1
{
        void method2();
}
class Demo implements Inf2
{
        public void method1()
        {
                System.out.println("Inf1 Implementation");
        }
        public void method2()
        {
                System.out.println("Inf2 Implementation");
        }
        public static void main(String args[])
        {
                Demo d=new Demo();
                d.method1();
                d.method2();
        }
}
```



```
E:\JAVA\ANU>javac Demo.java

E:\JAVA\ANU>java Demo
Inf1 Implementation
Inf2 Implementation
```

- In the above program an interface can not implement another interface . It has to extend the other interface. Here we have two interfaces Inf1 and Inf2.
- Inf2 extends Inf1 so the class implemets the Inf2 it has to provide implementation of all the methods of interface Inf2 as well as Inf1.

Keypoints

Here are the some keypoints to remember about the interfaces

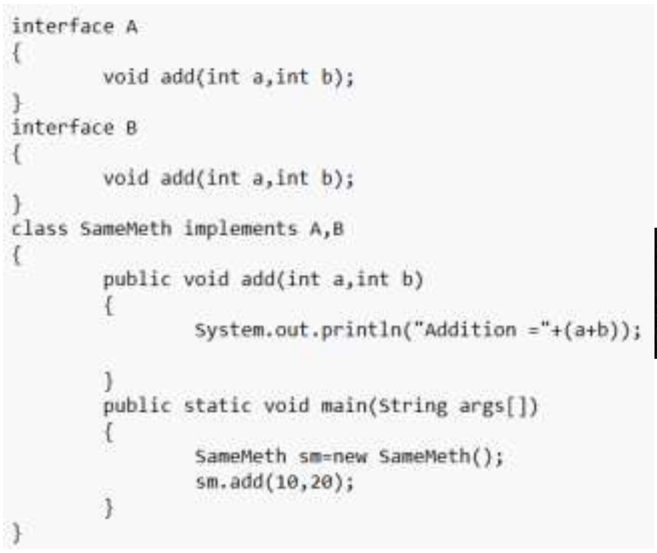
- We can not instantiate an Interface in Java (can't creates any object in interface).
- Interface provides full abstraction as non of its methods have body. In Other hand abstract class provides partial abstraction (0% to 100%) & interface provides full abstraction (100%).
- An Interface can Extend any interface but cannot implement. Class implements interface and interface Extends interface.
- implements keyword is used by class to implement an interface.
- A Class can implements any number of interfaces (Multiple inheritance can be achieve by interfaces).
- All the interface Method are by default Abstract & public.
- While providing implementation in class of any method of an interface it needs to be mentioned as public.
- Variable declared in interface are public, State & final by default.

Ex

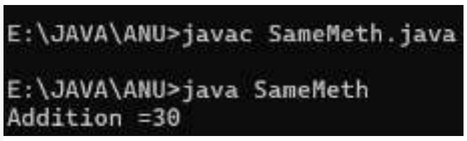
```
interface Inter
{
        public static final float PI = 3.14;
}
```

- Interface Variables must be initialized at the time of declaration otherwise Compiles will raises an error. If there are two or more same methodes in two interfaces and a class implements both interfaces, implementation of method Once is enough.

Program for Multiple Inheritance

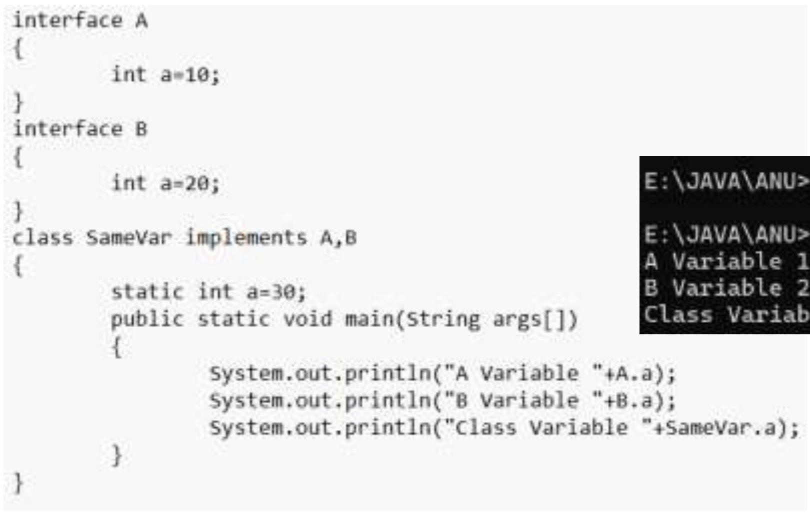


```
interface A
{
        void add(int a,int b);
}
interface B
{
        void add(int a,int b);
}
class SameMeth implements A,B
{
        public void add(int a,int b)
        {
                System.out.println("Addition ="+(a+b));

        }
        public static void main(String args[])
        {
                SameMeth sm=new SameMeth();
                sm.add(10,20);
        }
}
```

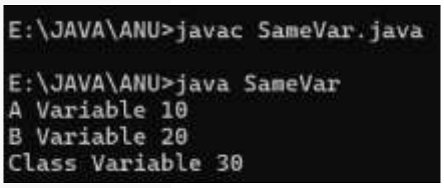


```
E:\JAVA\ANU>javac SameMeth.java

E:\JAVA\ANU>java SameMeth
Addition =30
```

- Variable name conflits are resolved by interface name.



```
interface A
{
        int a=10;
}
interface B
{
        int a=20;
}
class SameVar implements A,B
{
        static int a=30;
        public static void main(String args[])
        {
                System.out.println("A Variable "+A.a);
                System.out.println("B Variable "+B.a);
                System.out.println("Class Variable "+SameVar.a);
        }
}
```



```
E:\JAVA\ANU>javac SameVar.java

E:\JAVA\ANU>java SameVar
A Variable 10
B Variable 20
Class Variable 30
```

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.
But there are many differences between abstract class and interface those are as:

Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.

3) Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
4) Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
6) An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
7) An abstract class can be extended using keyword "extends".	An interface can be implemented using keyword "implements".
8) A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)Example:
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

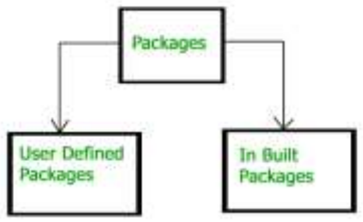
PACKAGES

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.
Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

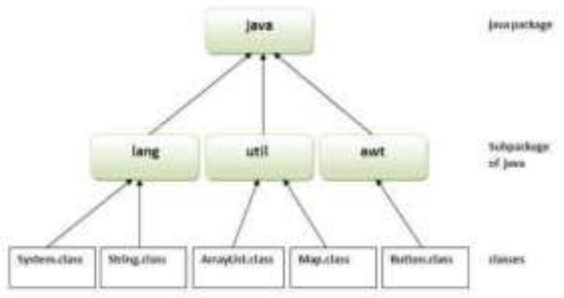
1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.

Types of Packages



Built-in Packages

These packages consist of a large number of classes which are a part of Java API.Some of the commonly used built-in packages are:



- **java.lang**: Contains language support classes(e.g classes which defines primitive data types, math operations). This package is automatically imported.
- **java.io**: Contains classes for supporting input / output operations.
- **java.util**: Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
- **java.applet**: Contains classes for creating Applets.
- **java.awt**: Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).

User defined Packages

A Java package is a group of similar types of classes, interfaces and sub-packages. A package, in the context of computing and software development, typically refers to a collection of files and resources bundled together for distribution, installation, and use. Packages are often used to organize and distribute software libraries, frameworks, applications, or modules. They typically contain executable code, configuration files, documentation, and other necessary resources. In various programming languages and environments, such as Python, Java, and JavaScript, packages are used to manage dependencies, facilitate code reuse, and streamline development processes. They are often distributed via package managers, which automate the process of downloading, installing, and updating software components. Packages are used for:

- Preventing naming conflicts. For example, there can be two classes with the name Employee in two college.staff.ee.Employee packages, college.staff.cse.Employee and
- 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)

Creating Packages

- Step 1: Simply include a package command that has the first statement in the Java source file. Any class you declare within that file will belong to the specified package.
  Syntax:
  ```
  package packagename;
  ```
  Example:
  ```
  package mypack;
  ```
- Step 2: Next define the class that is to be put in the package and declare it as public.
- Step 3: Now store the classname.java file in the directory having the name same as package name.
- Step 4: The file is to be compiled, which creates class file in the directory java also supports the package hierarchy, which allows to group of related classes into a package and then group related packages into a larger package. We can achieve this by specifying multiple names in a package statement, separated by a dot (.).

Syntax:

    package outerpackagename.innerpackagename;

Example:

    package dilkk.parni;

Accessing package: A Java system package can be accessed either using a fully qualified classname or using an import statement. We generally use import statements.

Syntax:

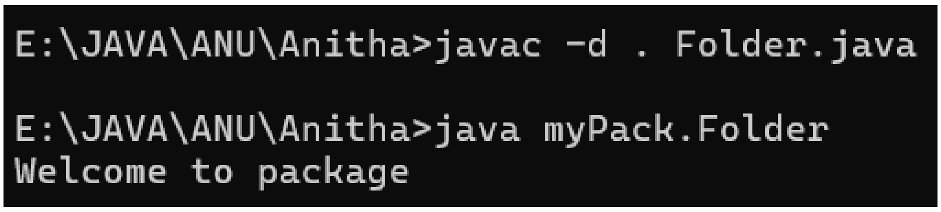
            import package.classname;
                    or
            import package.*;

Java program on creating a package

```
package myPack;
class Folder
{
        public static void main(String args[])
        {
                System.out.println("Welcome to package");
        }
}
```

To compile: javac –d . ClassName.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).



```
E:\JAVA\ANU\Anitha>javac -d . Folder.java

E:\JAVA\ANU\Anitha>java myPack.Folder
Welcome to package
```

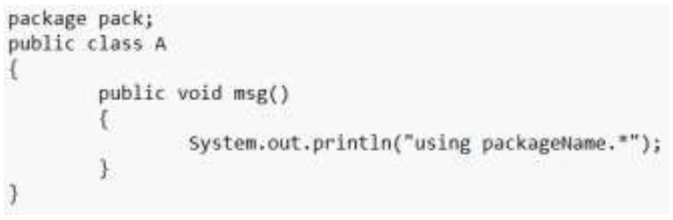
Accessing package from another package:

There are three ways to access the package from outside the package.

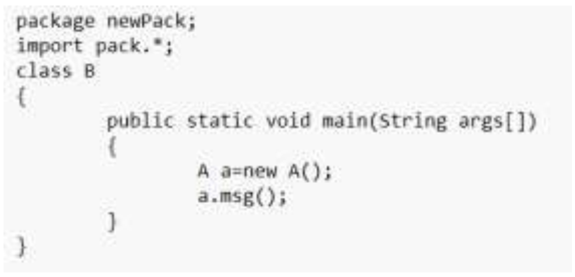
1. import package.*;
2. import package.classname;
3. fully qualified name.

1) **Using packagename.***

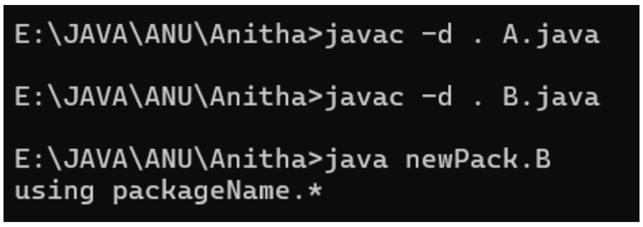
- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.



```
package pack;
public class A
{
        public void msg()
        {
                System.out.println("using packageName.*");
        }
}
```



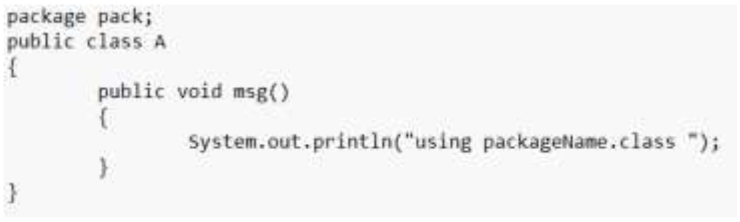
```java
package newPack;
import pack.*;
class B
{
        public static void main(String args[])
        {
                A a=new A();
                a.msg();
        }
}
```



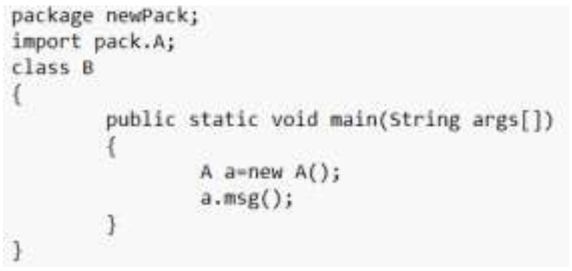
```
E:\JAVA\ANU\Anitha>javac -d . A.java

E:\JAVA\ANU\Anitha>javac -d . B.java

E:\JAVA\ANU\Anitha>java newPack.B
using packageName.*
```

2) Using packagename.classname

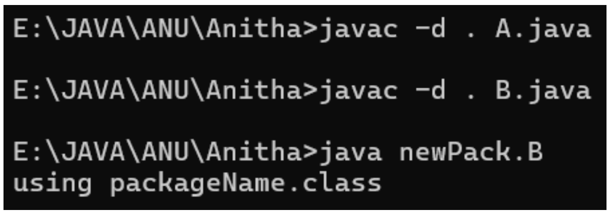
If you import package.classname then only declared class of this package will be accessible.



```java
package pack;
public class A
{
        public void msg()
        {
                System.out.println("using packageName.class ");
        }
}
```



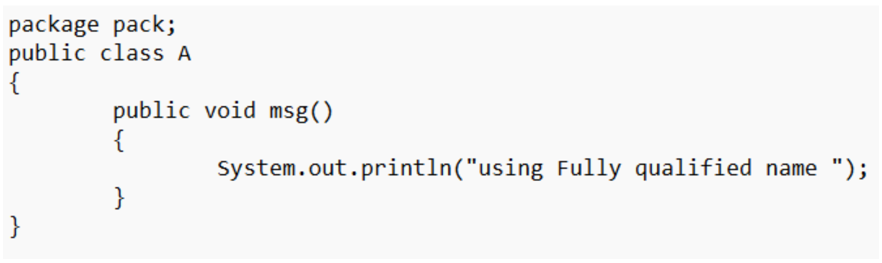
```java
package newPack;
import pack.A;
class B
{
        public static void main(String args[])
        {
                A a=new A();
                a.msg();
        }
}
```



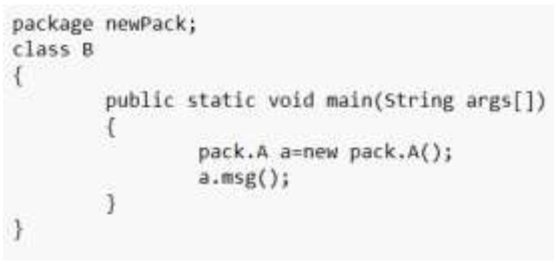
```
E:\JAVA\ANU\Anitha>javac -d . A.java

E:\JAVA\ANU\Anitha>javac -d . B.java

E:\JAVA\ANU\Anitha>java newPack.B
using packageName.class
```

3) Using fully qualified name

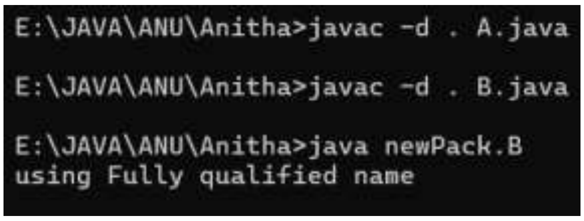
- If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.



```java
package pack;
public class A
{
        public void msg()
        {
                System.out.println("using Fully qualified name ");
        }
}
```



```java
package newPack;
class B
{
        public static void main(String args[])
        {
                pack.A a=new pack.A();
                a.msg();
        }
}
```



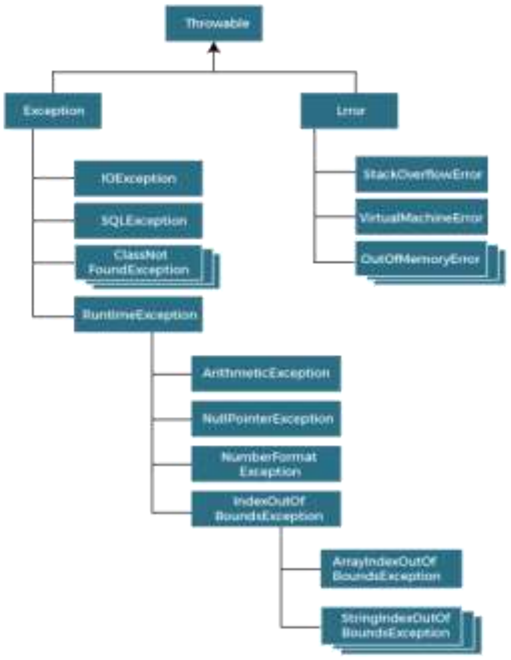
```
E:\JAVA\ANU\Anitha>javac -d . A.java

E:\JAVA\ANU\Anitha>javac -d . B.java

E:\JAVA\ANU\Anitha>java newPack.B
using Fully qualified name
```

Exception Handling

- The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.
- Exception is an abnormal condition.
- An exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

- Checked Exception
- Unchecked Exception
- Error

Checked Exception

- The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

Unchecked Exception

- The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

Error : Error is irrecoverable.

some scenarios where unchecked exceptions may occur

ArithmeticException

If we divide any number by zero, there occurs an ArithmeticException.

Ex : int a=50/0;//ArithmeticException

NullPointerException

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException

Ex : String s=null;
System.out.println(s.length());//NullPointerException

NumberFormatException

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

Ex : String s="abc";
int i=Integer.parseInt(s);//NumberFormatException

ArrayIndexOutOfBoundsException

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

Ex : int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException

Java Exception Keywords

Java provides five keywords that are used to handle the exception.

- **Try**

The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.

Syntax:

```
try
{
    //Exceptional code
}
```

- **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. We can use multiple catch blocks with a single try block if multiple exceptions raised.

Syntax:

```
try
{
    //Exceptional code
}
catch(Exception_Name arg)
{
    //Exception handling code
}
```

- **finally**

The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.

Syntax:

```
try
{
    //Exceptional code
}
catch(Exception_Name arg)
{
    //Exception handling code
}
finally
{
    //important code
}
```

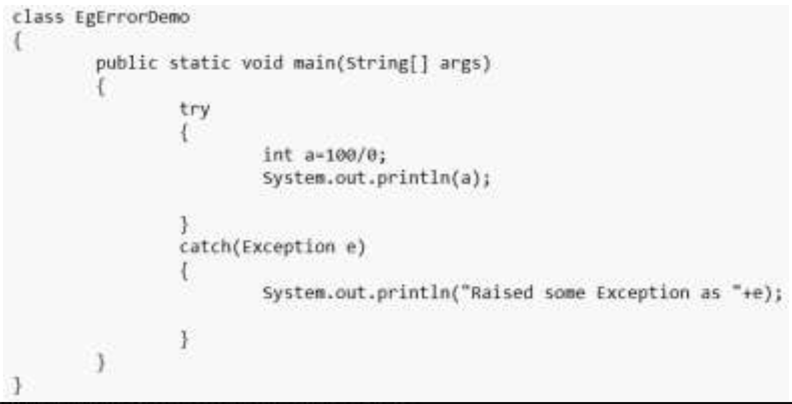
- **throw**

The "throw" keyword is used to throw an exception.

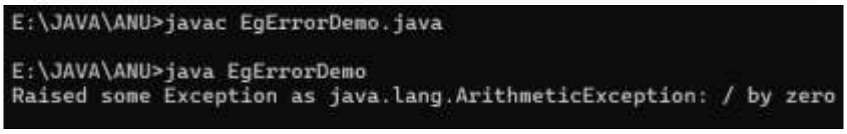
- **throws**

The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java program to illustrate Exception Handling

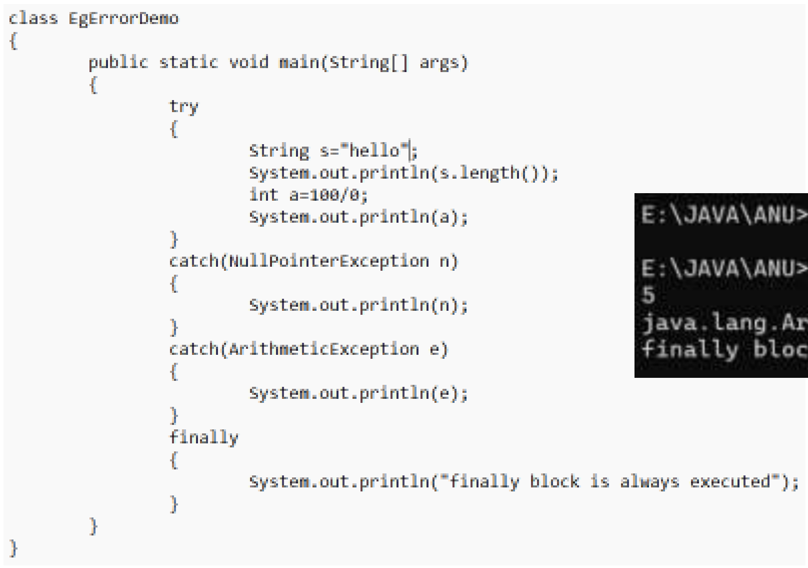


```java
class EgErrorDemo
{
        public static void main(String[] args)
        {
                try
                {
                        int a=100/0;
                        System.out.println(a);

                }
                catch(Exception e)
                {
                        System.out.println("Raised some Exception as "+e);

                }
        }
}
```

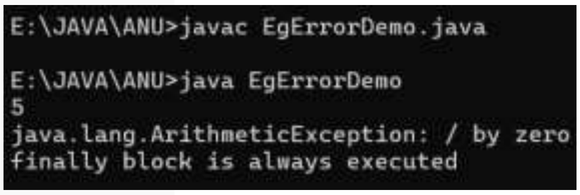


```
E:\JAVA\ANU>javac EgErrorDemo.java

E:\JAVA\ANU>java EgErrorDemo
Raised some Exception as java.lang.ArithmeticException: / by zero
```

Java program to illustrate Exception Handling using multiple catch blocks and finally block

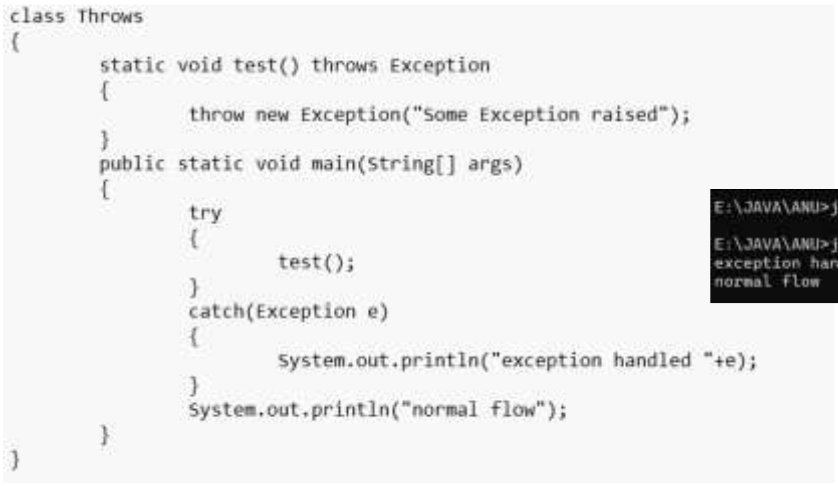


```java
class EgErrorDemo
{
        public static void main(String[] args)
        {
                try
                {
                        String s="hello";
                        System.out.println(s.length());
                        int a=100/0;
                        System.out.println(a);
                }
                catch(NullPointerException n)
                {
                        System.out.println(n);
                }
                catch(ArithmeticException e)
                {
                        System.out.println(e);
                }
                finally
                {
                        System.out.println("finally block is always executed");
                }
        }
}
```

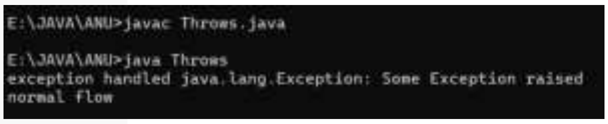


```
E:\JAVA\ANU>javac EgErrorDemo.java

E:\JAVA\ANU>java EgErrorDemo
5
java.lang.ArithmeticException: / by zero
finally block is always executed
```

Java program on handling Exceptions that are raised from methods using throws keyword



```java
class Throws
{
        static void test() throws Exception
        {
                throw new Exception("Some Exception raised");
        }
        public static void main(String[] args)
        {
                try
                {
                        test();
                }
                catch(Exception e)
                {
                        System.out.println("exception handled "+e);
                }
                System.out.println("normal flow");
        }
}
```



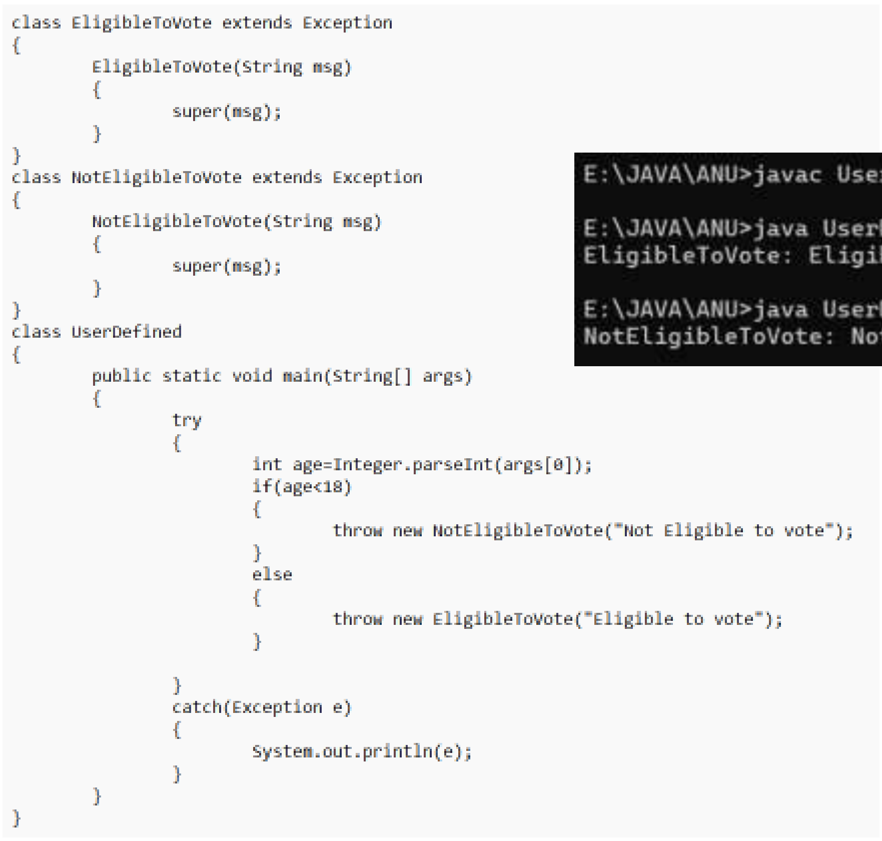
```
E:\JAVA\ANU>javac Throws.java

E:\JAVA\ANU>java Throws
exception handled java.lang.Exception: Some Exception raised
normal flow
```

Difference between throw and throws in Java

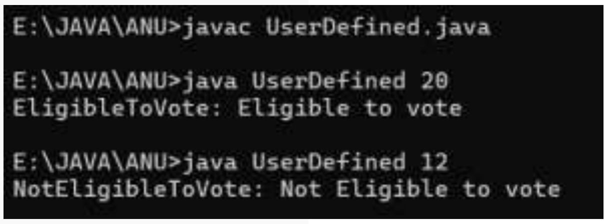
throw	throws
Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.
The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
throw is used within the method.	throws is used with the method signature.
We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

User-defined Exceptions

Java provides us the facility to create our own exceptions which are basically derived classes of Exception. Creating our own Exception is known as a custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user needs. In simple words, we can say that a User-Defined Exception or custom exception is creating your own exception class and throwing that exception using the 'throw' keyword.



```
class EligibleToVote extends Exception
{
        EligibleToVote(String msg)
        {
                super(msg);
        }
}
class NotEligibleToVote extends Exception
{
        NotEligibleToVote(String msg)
        {
                super(msg);
        }
}
class UserDefined
{
        public static void main(String[] args)
        {
                try
                {
                        int age=Integer.parseInt(args[0]);
                        if(age<18)
                        {
                                throw new NotEligibleToVote("Not Eligible to vote");
                        }
                        else
                        {
                                throw new EligibleToVote("Eligible to vote");
                        }

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



```
E:\JAVA\ANU>javac UserDefined.java

E:\JAVA\ANU>java UserDefined 20
EligibleToVote: Eligible to vote

E:\JAVA\ANU>java UserDefined 12
NotEligibleToVote: Not Eligible to vote
```