Module 3

Inheritance

It is the mechanism in java by which one object acquires the properties of the other object. This is important because it supports the concept of hierarchical classification. The class whose features are inherited is known as super class (or a base class or a parent class). The class that inherits the other class is known as sub class (or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the super class fields and methods. Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class.

Defining a sub class:

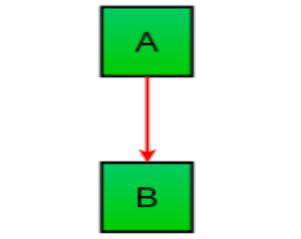
```
class subclassname extends superclassname
{
  Variables    declarartion;
  Methods declarartion;
  }
```

The keyword extends specifies that the properties of the superclassname are extended to the subclassname.

Types of Inheritance

1. Single inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance

Single Inheritance: Deriving a new class from a single class is called single inheritance



Single Inheritance

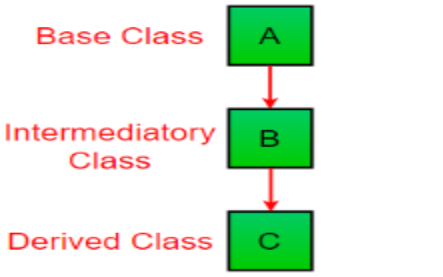
Example:

```
import java.util.*;
        class A
        {  int m.n;
          void getData()
          {
          Scanner s=new Scanner(System.in);
          System.out.println("Enter the values of m and n");
          m=s.nextInt();
          n=s.nextInt();}
```

```
        void putData()
        {
                System.out.println("m="+m+"n="+n);
        }
    }
class B extends A
{
  int sum;
 void add()
 {
 sum=m+n;
  System.out.println("  SUM="+sum);
 }
}
        class SingleInherit
        {
          public static void main(String args[])
          {
           B ob=new B();
           ob.getData();
           ob.putData();
           ob.add();
          }
        }
```

Multilevel Inheritance

Deriving a class from another derived class is called multilevel inheritance.



Multilevel Inheritance

Example:

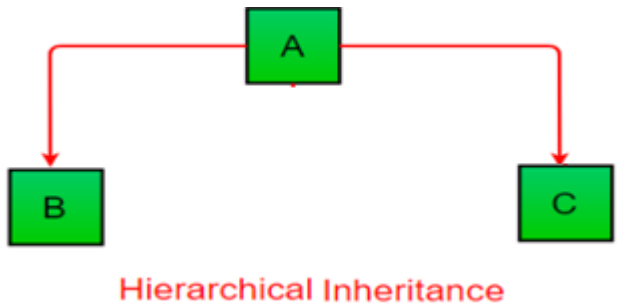
```
import java.util.*;
        class A
        {  int m.n;
          void getData()
          {
          Scanner s=new Scanner(System.in);
          System.out.println("Enter the values of m and n");
          m=s.nextInt();
          n=s.nextInt();}
        void putData()
        {
```

```
                System.out.println("m="+m+"n="+n);
            }
        }
class B extends A
{
   int sum;
  void add()
  {
  sum=m+n;
   System.out.println("  SUM="+sum);
  }
}
class C extends B
{
   int prod;
  void product()
  {
  prod=m*n;
   System.out.println("  Product="+prod);
  }
}

            class Multilevel
            {
              public static void main(String args[])
              {
               C ob=new C();
               ob.getData();
               ob.putData();
               ob.add();
               ob.product();
              }
            }
```

Hierarchical Inheritance

Deriving many classes from a single super class is called hierarchial inheritance.



Hierarchical Inheritance

Example:
```
import java.util.*;
        class A
        {  int m.n;
          void getData()
          {
          Scanner s=new Scanner(System.in);
          System.out.println("Enter the values of m and n");
          m=s.nextInt();
          n=s.nextInt();}
        void putData()
        {
                System.out.println("m="+m+"n="+n);
        }
    }
class B extends A
{
   int sum;
  void add()
  {
  sum=m+n;
   System.out.println("  SUM="+sum);
  }
}
class C extends A
{
   int prod;
  void product()
  {
  prod=m*n;
   System.out.println("  Product="+prod);
  }
}

        class Hierachial
        {
          public static void main(String args[])
          {
                B ob1=new B();
                ob1.getData();
                ob1.putData();
                ob1.add();
                C ob=new C();
                  ob.getData();
                  ob.putData();
                  ob.product();
          }
        }
```

Access Specifiers / Modifiers

Sometimes, we may not like the objects of a class directly altering the value of a variable or accessing a method. This is achieved by applying visibility modifiers to the instance variables and methods. These visibility modifiers are called access modifiers.

public access : A variable or method declared as public has the widest possible visibility and accessable everywhere i.e., within the package as well as outside the package.
Example : public int a;

```
public void sum()
 {
    // statements
 }
```

Friendly access : when no modifier is specified, the member defaults to a limited version of public accessibility known as "friendly " level of access. The difference between the public access and friendly access is that, the public modifier makes fields visible in all classes regardless of their packages, while the friendly access makes fields visible only in the same package, but not in other packages.
protected access: The visibility of a protected field lies in between the public and friendly access i.e., the protected modifier makes the fields visible in the same package but also to sub classes in other packages.
private access : private fields are accessible only within their classes. A method declared as private behaves like a method declared final. It prevents the method from being sub classed.

Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**. For example, consider the following simple class hierarchy:

```
/* In a class hierarchy, private members remain private to their class.
*/
This program contains an error and will not
compile.
// Create a superclass.
class A {
int i; // public by default
private int j; // private to A
void setij(int x, int y) {
i = x;
j = y;
}
}
// A's j is not accessible here.
class B extends A {
int total;
void sum() {
total = i + j; // ERROR, j is not accessible here
}
class Access {
```

```
        public static void main(String args[]) {
        B subOb = new B();
        S  ubOb.setij(10, 12);
        subOb.sum();
        System.out.println("Total is " + subOb.total);
         }
        }
```

Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. In the following program, **obj** is a reference to **NewData** object, Since **NewData** is a subclass of **Data**, it is permissible to assign **obj** a reference to the **NewData** object. When a reference to a subclass object is assigned to a superclass reference variable, you will have access only to those parts of the object defined by the superclass. This is way **obj** can't access **data3** and **data4** even it refers to a **NewData** object.

```
class Data{
        int data1;
        int data2;
        }
class NewData extends Data{
{       int data3;
        int data4;
}
public class Javaapp{
    public static void main(String args[]){
        Data obj=new NewData();
        obj.data1=50;
        obj.data2=100;
        System.out.println("obj.data1 = "+obj.data1);
        System.out.println("obj.data2 = "+obj.data2);
        }
    }
```

Super

The super keyword is a reference variable that is used to refer parent class objects

Usage of super keyword

1. super() can be used to invoke immediate parent class constructor.
2. super can be used to refer immediate parent class instance variable.
3. super can be used to refer immediate parent class method.
4. Super() must be the first statement in the sub class constructor to invoke the super class constructor.

// Program to call immediate super class constructor using super

```
class A{
        A{
          System.out.println("Super class Constructor");
          }
}
class B extends A {
        B()
          {
                super();
                System.out.println("Sub class Constructor");
        }
}
class supereg1{
        public static void main(String args[])
        {
          B ob=new B();
        }
}
```

Ouput

Super class Constructor
Sub class Constructor

// Program to refer super class instance variable

```
class A {
int i;
  }
class B extends A{
  int i;
  B(int a, int b)
{
  super.i=a;
    i=b;
}
void show()
{
System.out.println(" i of super class"+ super.i);
System.out.pr intln(" i of sub class"+ i);
}
}
  class supereg2
  {
```

```
        public static void main(String args[])
        {
          B ob=new B(10,20);
           ob.show();
        }}
```

// Program to invoke super class method using super

```
        class A{
                void display()
                {
                  System.out.println("Display of Super class");
                }
           }
        class B extends A{
                void display()
                {
                  System.out.println("Display of Sub class");
                }
                void printMsg()
                {
                display(); // overriding method is called
                super.display();  // overridden method is called
                }
          public static void main(String args[])
        {
                B ob=new B();
                ob.printMsg();
        }
    }
```

Method overloading

It is possible to define two or more methods within the same class that share the same name but differ in the signature i.e., difference in return type or number of parameters passed or data type of the parameters passed or the order of the parameters passed. Method overloading is used when we perform similar kind of operations on different data type. This is called **static polymorphism** because, which method to be invoked is decided at the time of compilation

Example:

```
class Geometry{
                void area(double r){
                double a=3.142*r*r;
                System.out.println("Area of circle is"+a);
                }
                void area(int b, int h){
                double a=0.5*b*h;
```

```
            System.out.println("Area of triangle is"+a);
            }}
class demo{
   public static void main(String args[])
            {
              Geometry g=new Geometry();
              g.area(2.5);
              g.area(10,20);
            }
}
```

Method overriding

When a method in a sub class has the same name and type signature as a method of super class, then the method in the sub class is said override the method in the super class. When a overridden method is called from within a subclass, it always refer to the version of that method defined by the sub class. The version of the method defined in the super class is hidden.

Method overriding allows a sub class to provide a specific implementation of a method that is already provided by one of its super classes or parent classes. The call to overriden method is resolved at runtime, thus called **runtime**/**dynamic** polymorphism.

Example:

```
class A{
            void display()
            {
              System.out.println("Display of Super class");
            }
   }
class B extends A{
            void display()
            {
              System.out.println("Display of Sub class");
            }
class MethodOverride{
{
  public static void main(String args[]) {
   {
            B ob=new B();
    }
}
```

Dynamic Method Dispatch (DMD)

It is the mechanism by which a call to an overridden method is resolved at run time rather than compile time. It is important because this is how java implements run time polymorphism

A super class reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a super class reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.

```
// Program to demonstrate Dynamic Method Dispatch
class A
{
        void display()
        {
          System.out.println(“ welcome to Nokia World”);
        }
}
class B extends A
{
        void display()
        {
          System.out.println(“ welcome to Samsung World”);
        }
}
class DMD
{
        public static void main(String args[])
        {
          A x=new A( );
          x.display( );  // invokes super class display( )
          B y=new B( );
          x=y;
          x.display( );  // invokes sub class display( )
        }
}
```

Abstract Classes

There are situations in which you will want to define a super class that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a super class that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement.

- Abstract class is an incomplete class that contains one or more abstract methods.
- A abstract method is a method that does not contain any implementation
- An abstract class may contain concrete method(with implementation) along with abstract method

- Abstract classes can be inherited
- Abstract class can't be instantiated
- Abstract constructors and abstract static method can't be declared
- An abstract class must have at least one subclass which overrides the abstract methods. If sub class does not override the methods, then it also becomes abstract.

Example:

```
abstract class A
{
        void display()
        {
          System.out.println("Hello");
        }
    abstract void show( );
}
class B extends A
{
        void show( )
        {
          System.out.println(" Show() implemented");
        }
}
class demo
{
        public static void main(String args[])
        {
          B Ob=new B();
          Ob.display();
          Ob.show();
        }
}
```

final Keyword

There are three uses of final

1. If a variable is declared as final, its contents can not be modified, it becomes a constant
2. It is used to prevent method overriding.
3. It is used to prevent inheritance.

// A final variable is a constant

```
class A
{
        int a=10;
        final int b=20;
```

```
        void change()
        {
          a=a+10;
        //b=b+10; final variable cant be modified
        System.out.println(a);
        System.out.println(b)
        }
  }
class finalvar
 {
        public static void main(String args[])
        {
                A ob=new A();
                ob.change():
        }
}
```

//A final method cannot be overridden

```
class A
{
        void display()
        {
          System.out.println("display of super method");
        }
        final void show( )
        {
          System.out.println(" Show() implemented");
        }
}
class B extends A
{
        void display()
        {
          System.out.println("display of sub method");
        }
        /* void show( )
        {
          System.out.println(" Show() implemented");
        }
        cant be inherited
       */
}
```

```
//A final class can't be inherited
final class A
{
        void display()
        {
          System.out.println("Hello");
        }
   }
/*final class A can't be inherited
class B extends A
{
        void show( )
        {
          System.out.println(" Show() implemented");
        }
}
 A final class can't be inherited
*/
```

Differences between a concrete class and an abstract class

Concrete Class	abstract Class
A regular class will have all implemented methods	An abstract class will have at least one abstract method
A normal class need not have a sub class	An abstract class must have at least one sub class which implements all the abstract methods
A regular class can be instantiated	An abstract class can not be instantiated as it as an incomplete class

Differences between an abstract class and a final class

abstract Class	final Class
An abstract class will have at least one abstract method	A final class does not contain any abstract methods rather it contains only concrete methods
An abstract class can not be instantiated as it as an incomplete class	A final class can be instantiated
An abstract class must be inherited	A final class cannot be inherited

Significance of main method in Java

public static main(String args[])

public : public is an access specifier. When a class is preceded by public, then the members of the class may be accessed by outside the class in which it is declared. main() method must be public since it must be called by code outside of its class when the program is started.
static : The keyword static allows main() to be called without having to create a particular instance (object). This is necessary since main() is called before any objects are created.
void : The keyword void simply tells the compiler that main() does not return a value to the operating system.
String args[] :This declares a parameter named args which is an array of instances of the class String. Objects of type String store character streams. args receives any command line arguments supplied when the program is executed.

Significance of System.out.println(): System is a predefined class that provides access to the system. out is the output stream that is connected to the console. println() displays the data on the screen.

Differences between method overloading and method overriding

Method Overlaoding	**Method overriding**
Method overloading occurs at compile time	Method overriding occurs at run time
It supports static polymorphism	It supports dynamic/run time polymorphism
Different number of parameters can be passes to methods	Same number of parameters are passed to the methods
The overloaded methods may have different return types	All methods will have the same return types
Method overloading is performed within the same class	Method overriding is performed between two classes that have inheritance relationship

INTERFACES

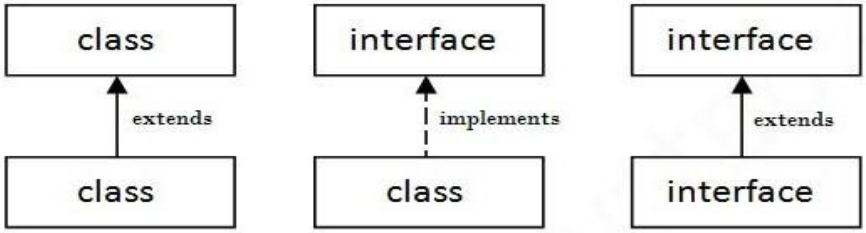
Interfaces are syntactically similar to class but defines only abstract method and final fields (must be static). Once interface is defined, any number of classes can implement. However, each class is free to determine the details of its own implementation. Java allows us to fully utilize the " one interface multiple forms" aspect of polymorphism. If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared abstract.
Syntax:

```
interface interfacename
{
        Variable declaration;// variables declared must be static and final
        Method declaration;
}
```

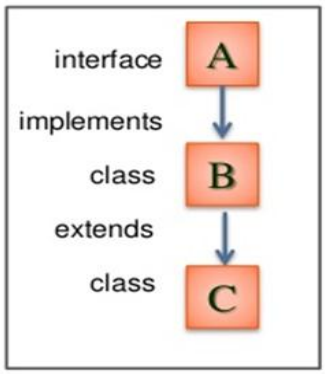
Here, the access is either public or not used. When no access specifier is included, then the default access specifier results and this interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code. The methods declared inside an interface are abstract by default, the variables declared must be final and static.

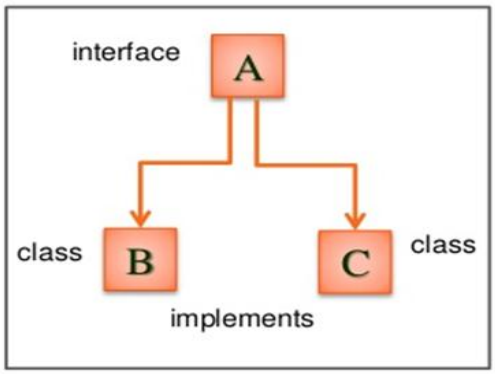
The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Various forms of Interfaces





//Program to demonstrate interface

```
interface printable
        {
        void print();
        }
class A implements printable{
        public void print()
        {
        System.out.println("Hello");
        }
class demo
{
public static void main(String args[]){
A ob = new A();
```

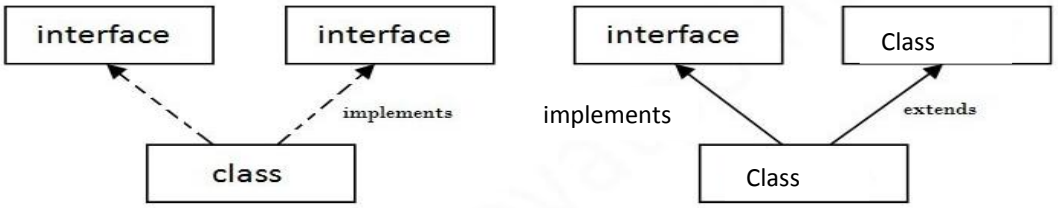
```
        ob.print();
        }
    }
```

// Interfaces can be extended

```
        interface inter1
        {
          void show();
        }
        interface inter2 extends inter1
        {
          void display();
        }
        class A implements inter2 // class A must override all the methods of inter1 and inter2
        {
                    public void show()
                   {
                     System.out.println(" Hai");
                   }
                  public void display()
                   {
                     System.out.println(" Hello");
                   }
        }
        class interfaceextends
        {
                    public static void main(String args[]){
                    A ob = new A();
                    ob.show();
                    ob.display();
                  }
        }
```

Multiple Inheritance

Multiple Inheritance is a feature of object oriented concept, where a class can inherit properties of more than one parent class.



Multiple inheritance in java is implemented using interfaces as shown above.

Example 1: To implement Multiple Inheritance

```
interface Printable
{
        void print();
}
interface Showable
{
        void show();
}
class A implements Printable,Showable
{
        public void print()
        {
        System.out.println("Hello");
        }
        public void show()
        {
        System.out.println("Welcome");
        }

        public static void main(String args[])
        {
                A obj = new A();
                obj.print();
                obj.show();
         }
}
```

Example 2: To implement Multiple Inheritance

```
interface Printable
{
        void print();
}
class B
{
        void show()
```

```
        {
        System.out.println("Welcome");
        }

}
class A extends B implements Printable
{
        public void print()
        {
        System.out.println("Hello");
        }

        public static void main(String args[])
        {
                A obj = new A();
                obj.print();
                obj.show();
         }
}
```

Differences between class and interface

Class	interface
A class is instantiated to create objects	An interface can never be instantiated as the methods are unable to perform any action on invoking
The members of a class can be private, public or protected	The members of an interface are always public
The methods of a class are defined to perform a specific action	The methods in an interface are purely abstract
A class can have constructors to initialize the variables	An interface can never have a constructor as there is hardly any variable to initialize
A class can implement any number of interface and can extend only one class	An interface can extend multiple interfaces but can not implement any interface

Differences between abstract class and interface

Abstract Class	interface
Abstract class can have abstract and non-abstract methods	Interface can have only abstract methods
An abstract class may contain non-final variables	Variables declared in a Java interface are by default final
Abstract class can provide the implementation of interface	Interface can't provide the implementation of abstract class
abstract class can be extended using keyword	A Java interface can be implemented using

"extends"	keyword "implements"
an abstract class can extend another Java class and implement multiple Java interfaces	An interface can extend another Java interface only

Exception Handling

Errors are broadly classified into two categories

- **Compile-time Errors**: All syntax errors will be detected and displayed by the java compiler and therefore these errors are called compile time errors. When there is compile-time error, .class file will not be generated.
  Examples: Missing semicolons, Missing braces, Use of undeclared variable etc.,
- **Run-time Errors**: A program may compile successfully creating .class file, but may not run properly. Such programs may produce wrong results due to wrong logic or even may terminate due to errors such as stack overflow. Exception in java is an indication of some unusual event.
  Examples: Dividing by zero
  ArrayIndexOutOfBounds
  NullPointerException
  Class not found Exception etc.,

Exceptions

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled. An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

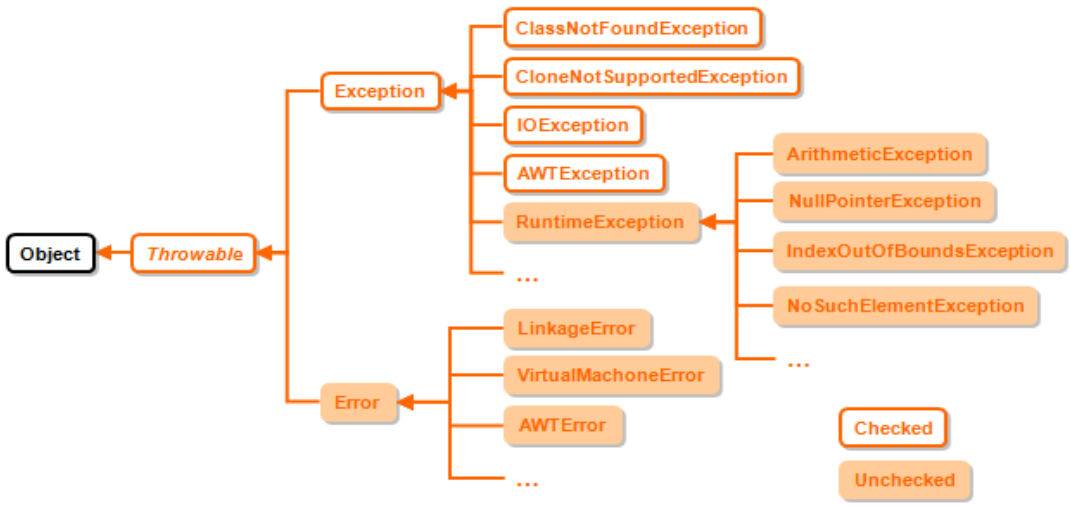
- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime. If the exception object is not caught and handled properly, the java interpreter will display an error message and will terminate the program. 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. This task is known as **Exception Handling**.

Hierarchy of Java Exception classes

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



Types of Exceptions

There are three categories of Exceptions. They are

1. Checked Exception
2. Unchecked Exception
3. Error

1.Checked exceptions − A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions. The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions.
E.g. IOException, SQLException etc. Checked exceptions are checked at compile-time

2.Unchecked exceptions − An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation. he classes which inherit RuntimeException are known as unchecked exceptions.

E.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime

3.Errors − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation. Error is irrecoverable.
E.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java

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.

Using try and catch

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following

```
try{
        // protected code
  }catch(Exception e){
        // catch block
  }
```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block.

// Using try and catch for Exception Handling

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e)
          {
          System.out.println(e);
          }
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Output

Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...

// Catching invalid Command line arguments

```
// Integer values to be given as command line arguments, if not, exception is raised and handled
  public class Example1{
  public static void main(String args[]){
          int invalid=0;
          int number, count=0;
   for(int i=0;i<args.length;i++)
          {
            try{
                  number=Integer.parseInt(args[i]);
                }catch(NumberFormatException e)
                  {
          invalid=invalid+1;
          System.out.println(“Invalid Number”+ args[i]);
          continue;
          }
          count=count+1;
    }
     System.out.println("Valid Nos"+count);
    System.out.println("Invalid Nos"+invalid);
   }
  }
```

Nested try statements

When a try catch block is present in another try block then it is called the nested try catch block. Each time a try block does not have a catch handler for a particular exception, then the catch blocks of parent try block are inspected for that exception, if match is found that that catch block executes.

Example: //To demonstrate nested try statements

```
class Example2
{       public static void main(String args[])
        {
          try
            {    int a=10,b=5, c=5, d;
                 int p[ ]={2,4};
                 p[4]=10;
                 try{
                    d=a/(b-c);
                  }catch(ArithmeticException e)
                   {
                         System.out.println("Division by Zero');
                   }
        }catch(ArrayIndexOutofBoundsException e){
                 System.out.println("ArrayIndexOutofBoundsException ');
          }
        }
 }
```

Multiple catch Statements

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following

```
try{
        //protected code
 } catch(ExceptionType1 e1){
        // catch block
} catch(ExceptionType2 e2){
        // catch block
}
```

Example: // To demonstrate multiple catch statements

```
        public class TestMultipleCatchBlock{
          public static void main(String args[]){
              try{
                  int a[]=new int[5];
                  a[5]=30/0;
                 }
                 catch(ArithmeticException e) {
                System.out.println("task1 is completed");
                 }
                 catch(ArrayIndexOutOfBoundsException e) {
```

```
                System.out.println("task 2 completed");
                }
                 catch(Exception e)  {
                System.out.println("common task completed");
                }
          System.out.println("rest of the code...");
        }
```

The finally block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception. Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code. A finally block appears at the end of the catch blocks and has the following syntax.

```
try {
   // Protected code
} catch (ExceptionType1 e1) {
   // Catch block
} catch (ExceptionType2 e2) {
   // Catch block
} catch (ExceptionType3 e3) {
   // Catch block
}finally {
   // The finally block always executes.
}
```

Example:// To demonstrate the finally block

```
public class ExcepTest {
  public static void main(String args[]) {
    int a[] = new int[2];
    try {
      System.out.println("Access element three :" + a[3]);
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception thrown  :" + e);
    }finally {
      a[0] = 6;
      System.out.println("First element value: " + a[0]);
      System.out.println("The finally statement is executed");
    }
  }
}
```

Output

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. We do this by including a **throws**

clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw.

This is the general form of a method declaration that includes a **throws** clause:

```
type method-name(parameter-list) throws exception-list
{
// body of method
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Example: // Program to demonstrate use of throws clause

```
class Example{
    static void divide( ) throws ArithmeticException
    {
        int x=22;
        int y=0, z ;
        z=x/y;
    }
    public static void main(String args[])
    {
      try{
            divide( );
         }
          catch(ArithmeticException ae)
           {
             System.out.println("Caught Exception " +ae);
           }
       }
  }
```

throw

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exceptions.

Example:

```
throw new ArithmeticException("/ by zero");
```

But this exception i.e, *Instance* must be of type **Throwable** or a subclass of **Throwable**. For example Exception is a sub-class of Throwable and user defined exceptions typically extend Exception class.

// Java program that demonstrates the use of throw

```
class ThrowExcep
{
   static void fun()
   {
      try
      {
         throw new NullPointerException("demo");
      }
```

```
        catch(NullPointerException e)
        {
            System.out.println("Caught inside fun().");
            throw e; // rethrowing the exception
        }
    }
     public static void main(String args[])
    {
        try
        {
            fun();
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught in main.");
        }
    }
}
```

User Defined Exceptions

In java we can create our own exception class and throw that exception using throw keyword. These exceptions are known as user-defined or custom exceptions. This can be done by extending the class Exception.

For example MyException in below code extends the Exception class. We pass the string to the constructor of the super class- Exception which is obtained using "getMessage()" function on the object created. In the below code, constructor of MyException requires a string as its argument. The string is passed to parent class Exception's constructor using super(). The constructor of Exception class can also be called without a parameter.

```
// A Class that represents use-defined expception

class MyException extends Exception
{
    public MyException(String s)
    {
        super(s);  // Call constructor of parent Exception
    }
}
  // A Class that uses above MyException
public class Main
{
     public static void main(String args[])
    {
        try
        {
            throw new MyException("GeeksGeeks"); // Throw an object of user defined exception
```

```
        }
        catch (MyException ex)
        {
            System.out.println("Caught");     // Print the message from MyException object
            System.out.println(ex.getMessage());
        }
    }
}
```

Output

Caught
GeeksGeeks

Methods defined by Throwable class

Exception and all of it's subclasses doesn't provide any specific methods and all of the methods are defined in the base class Throwable.

1. **String getMessage()** – This method returns the message String of Throwable and the message can be provided while creating the exception through it's constructor.
2. **String getLocalizedMessage()** – This method is provided so that subclasses can override it to provide locale specific message to the calling program. Throwable class implementation of this method simply use getMessage() method to return the exception message.
3. **synchronized Throwable getCause()** – This method returns the cause of the exception or null id the cause is unknown.
4. **String toString()** – This method returns the information about Throwable in String format, the returned String contains the name of Throwable class and localized message.
5. **void printStackTrace()** – This method prints the stack trace information to the standard error stream, this method is overloaded and we can pass PrintStream or PrintWriter as argument to write the stack trace information to the file or stream.

Java's Checked Exceptions Defined in java.lang

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Java's Unchecked RuntimeException Subclasses Defined in java.lang

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.