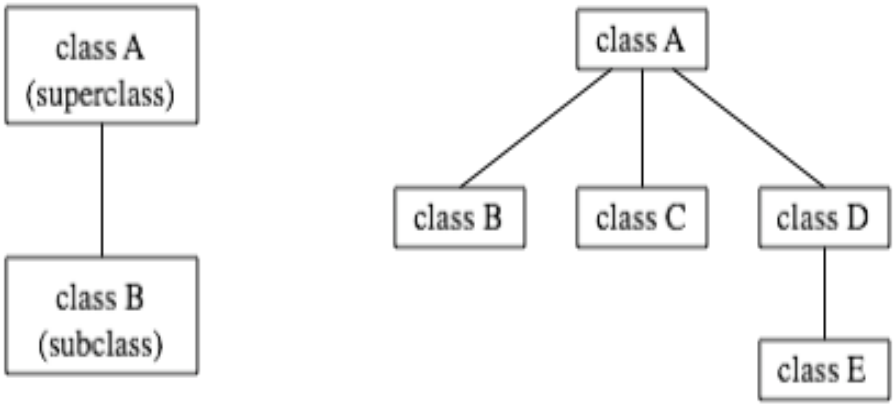
Inheritance

→→**Inheritance:** Inheritance is the process by which objects of one class acquire the properties of objects of another class. Inheritance supports the concept of hierarchical classification. A deeply inherited subclass inherits all of the attributes from each of its ancestors in the class hierarchy.

Most people naturally view the world as made up of objects that are related to each other in a hierarchical way.

Inheritance: A new class (subclass, child class) is derived from the existing class (base class, parent class).



Main uses of Inheritance: 1. Reusability 2. Abstraction

Syntax:

```
class Sub-classname extends Super-classname
{
        //Declaration of variables;
        //Declaration of methods;
}
```

- **Super class:** In Java a class that is inherited from is called a super class.
- **Sub class:** The class that does the inheriting is called as subclass. It inherits all of the instance variables and methods defined by the superclass and add its own, unique elements.
- The "**extends**" keyword indicates that the properties of the super class name are extended to the subclass name. The sub class now contains its own variables and methods.

```
// A simple example of inheritance.
// create a superclass.
class A
 {
int i, j;
void showij()
 {
System.out.println("i and j: " + i + " " + j);
}
}
// create a subclass by extending class A.
class B extends A
{
int k;
void showk()
{
System.out.println("k: " + k);
}
void sum() {
System.out.println("i+j+k: " + (i+j+k));
}
}
class SimpleInheritance
{
public static void main(String args[])
 {
A superOb = new A();
B subOb = new B();
// The superclass may be used by itself.
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb: ");
superOb.showij();
System.out.println();
/* The subclass has access to all public members of
its superclass. */
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of subOb: ");
subOb.showij();
subOb.showk();
System.out.println();
System.out.println("Sum of i, j and k in subOb:");
subOb.sum();
}
}
```

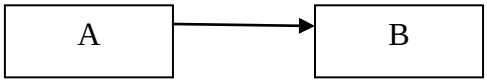
Output:

```
D:\>javac SimpleInheritance.java
D:\>java SimpleInheritance
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

→ Types of Inheritance are used to show the Hierarchical abstractions. They are:

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance.

Single Inheritance: Simple Inheritance is also called as single Inheritance. Here one subclass is deriving from one super class.



Super Class Sub Class

Example:

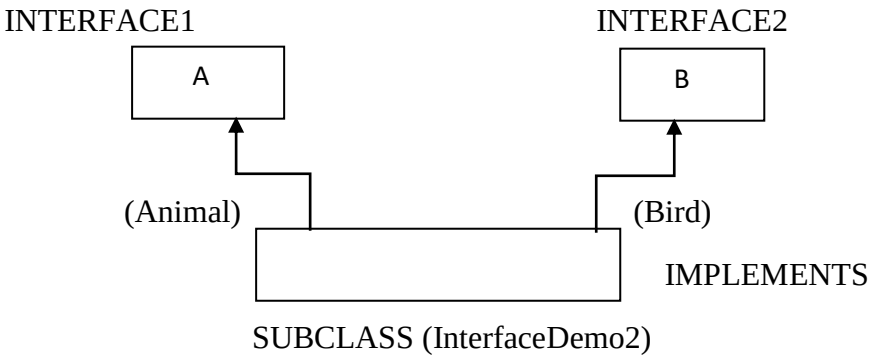
```
import java.io.*;
class A
{
void display()
        {
                System.out.println("hi”);

        }
}
class B extends A
{
        void display()
        {
                System.out.println("hello");

        }
}
```

```
class Inh
{
        public static void main(String args[])
        {
                B b=new B();
                b.display();

        }
}
```

Output:

```
D:\>javac Inh.java
D:\>java Inh
hello
```

Multiple Inheritance: Deriving one subclass from more than one super classes is called multiple inheritance.



→In multiple inheritance, sub class is derived from multiple super classes. If two super classes have same name for their members, then which member is inherited into the sub class is the main confusion in multiple inheritance. This is the reason; ***Java does not support the concept of multiple inheritance***. This confusion is reduced by using ***multiple interfaces to achieve the concept of multiple inheritance***.

Interface: An interface is a class containing a group of constants and method declarations that does not provide implementation. In essence, an interface allows you to specify what a class must do, but not how to do.

Interface syntax:

```
access interface name
 {
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type varname1 = value;
type varname2 = value;
// ...
return-type method-nameN(parameter-list);
type varnameN = value;
}
```

- ***By default all the methods in an interface must be abstract and public. If we do not mention these keywords the JVM will treat all these methods as public and abstract implicitly.***
- ***All the constants are treated as public, final and static.***

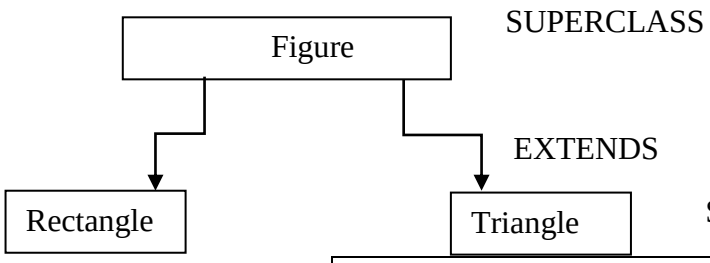
Example:

```
interface Animal
{
  public abstract void moves();
}

interface Bird
{
   void fly();
}

public class InterfaceDemo2 implements Animal, Bird
{
 public void moves()
{
 System.out.println("animal move on land");
}
 public void fly()
{
  System.out.println("birds fly in air");
}

public static void main(String args[])
{
InterfaceDemo2 id=new InterfaceDemo2();
id.moves();
id.fly();
}
}
```

Output:

```
 D:\>javac InterfaceDemo2.java
 D:\>java InterfaceDemo2
animal move on land
birds fly in air
```

Hierarchical Inheritance: Only one base class but many derived classes.



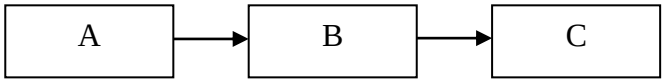
Example:

```
class Figure
 {
double dim1;
double dim2;

Figure(double a, double b)
{
dim1 = a;
dim2 = b;
}

double area()
{
System.out.println("Inside Figure");
}
}

class Rectangle extends Figure
 {

Rectangle(double a, double b)
{
super(a, b);
}

// override area for rectangle
double area()
{
System.out.println("Inside Area for
Rectangle.");
return dim1 * dim2;
}
}

class Triangle extends Figure
{

Triangle(double a, double b)
 {
super(a, b);
}

// override area for right triangle
double area()
 {
System.out.println("Inside Area for
Triangle.");
return dim1 * dim2 / 2;
}
}

class AreaDemo
 {
public static void main(String args[])
{
Figure f = new Figure(10, 10);
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref;
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
}
}
```

Output:

```
D:\>javac AreaDemo.java
D:\>java AreaDemo
Inside Area for Rectangle.
Area is 45.0
Inside Area for Triangle.
Area is 40.0
```

Multilevel Inheritance: In multilevel inheritance the class is derived from the derived class.



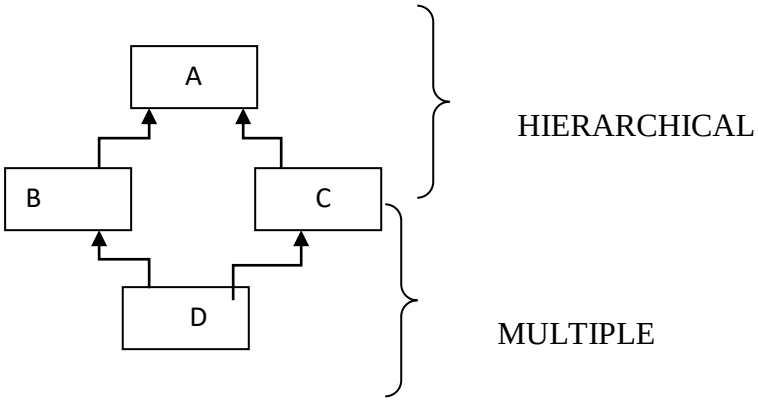
Example:

```
// Create a super class.
class A
{
A()
{
System.out.println("Inside A's constructor.");
}
}
// Create a subclass by extending class A.
class B extends A
{
B()
{
System.out.println("Inside B's constructor.");
}
}
// Create another subclass by extending B.
class C extends B
{
C()
{
System.out.println("Inside C's constructor.");
}
}
class CallingCons
{
public static void main(String args[])
{
C c = new C();
}
}
```

Output :

```
D:\>javac CallingCons.java
D:\>java CallingCons
Inside A's constructor.
Inside B's constructor.
Inside C's constructor.
```

Hybrid Inheritance: It is a combination of multiple and hierarchical inheritance.



→→**Super Uses:** Whenever a subclass needs to refer to its immediate super class, it can do so by the use of the keyword ***super***.

Super has the two general forms.

- super (args-list) : calls the Super class's constructor.
- super.member: To access a member of the super class that has been hidden by a member of a subclass. Member may be variable or method.

The keyword 'super':

- super can be used to refer super class variables as: **super.variable**
- super can be used to refer super class methods as: **super.method ()**
- super can be used to refer super class constructor as: **super (values)**

1. Accessing the super class constructor: super (parameter_list) calls the super class constructor. The statement calling super class constructor should be the first one in sub class constructor.

Example program for super can be used to refer super class constructor as: super (values)

```
class Figure
{
double dim1;
double dim2;
Figure(double a, double b)
{
dim1=a;
dim2=b;
}
}
class Rectangle extends Figure
{
Rectangle(double a,double b)
{
super(a,b);//calls super class constructor
}
double area()
{
System.out.println("Inside area for rectangle");
return dim1*dim2;
}
}
class Triangle extends Figure
{
Triangle(double a,double b)
{
super(a,b);
}
double area()
{
System.out.println("Inside area for triangle");
return dim1*dim2/2;
}
}
class FindAreas
{
public static void main(String args[])
{
 Rectangle r=new Rectangle(9,5);
 Triangle t=new Triangle(10,8);
 System.out.println("area is"+r.area());
 System.out.println("area is"+t.area());
}
}
```

Output:

```
D:\>javac FindAreas.java
D:\>java FindAreas
Inside area for rectangle
area is45.0
Inside area for triangle
area is40.0
```

2. **Accessing the member of a super class:** The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used.

Syntax: super**.** member;

Here, member can be either a method or an instance variable. This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:

```
// Using super to overcome name hiding.
class A
{
int i;
}
// Create a subclass by extending class A.
class B extends A
{
int i; // this i hides the i in A
B(int a, int b)
 {
super.i = a; // i in A
i = b; // i in B
}
void show()
 {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
}
}
class UseSuper
{
public static void main(String args[])
 {
B subOb = new B(1, 2);
subOb.show();
}
}
```

Output:

```
D:\>javac UseSuper.java
D:\>java UseSuper
i in superclass: 1
i in subclass: 2
```

Super uses: super class's method access

```
import java.io.*;
class A
{
    void display()
  {
System.out.println("hi");
   }
}
class B extends A
{
void display()
{
super.display();// calls super class display()
System.out.println("hello");
}
static public void main(String args[])
{
B b=new B();
b.display();
}
}
```

Output:

```
D:\>javac B.java
D:\ >java B
hi
hello
```

Note: Super key word is used in sub class only.

→→Abstract classes:

- A method with body is called concrete method. In general any class will have all concrete methods.
- A method without body is called ***abstract method***.

Syntax: *abstract datatype methodname(parameter-list);*

- A class that contains abstract method is called ***abstract class***.
- It is possible to implement the abstract methods differently in the subclasses of an abstract class.
- These different implementations will help the programmer to perform different tasks depending on the need of the sub classes. Moreover, the common members of the abstract class are also shared by the sub classes.
- The abstract methods and abstract class should be declared using the keyword ***abstract***.
- We cannot create objects to abstract class because it is having incomplete code. Whenever an abstract class is created, subclass should be created to it and the abstract methods should be implemented in the subclasses, then we can create objects to the subclasses.
- An abstract class is a class with zero or more abstract methods
- An abstract class contains instance variables & concrete methods in addition to abstract methods.
- It is not possible to create objects to abstract class.
- But we can create a reference of abstract class type.
- All the abstract methods of the abstract class should be implemented in its sub classes.
- If any method is not implemented, then that sub class should be declared as ***'abstract'.***
- Abstract class reference can be used to refer to the objects of its sub classes.
- Abstract class references cannot refer to the individual methods of sub classes.
- A class cannot be both 'abstract' & 'final'.

  e.g.: final abstract class A // invalid

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of attributes and methods to operate on these attributes. They encapsulate all the essential features of the objects that are to be created since the classes use the concept of data abstraction they are known as Abstract Data Types.

→An abstract class can be sub classed and can't be instantiated.

Example: // Using abstract methods and classes.

```
abstract class Figure
{
double dim1,dim2;
Figure (double a, double b)
{
dim1 = a;
dim2 = b;
}
abstract double area();//an abstract method
}
class Rectangle extends Figure
{
Rectangle (double a, double b)
{
super (a, b);
}
double area ()    // override area for rectangle
{
System.out.println ("Inside Area of Rectangle.");
return dim1 * dim2;
}
}
class Triangle extends Figure
{
Triangle (double a, double b)
{
super (a, b);
}
double area()     // override area for triangle
{
System.out.println ("Inside Area of Triangle.");
return dim1 * dim2 / 2;
}
}
class AbstractAreas
{
public static void main(String args[])
{
// Figure f = new Figure(10, 10); // illegal now
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
System.out.println("Area is " + r.area());
System.out.println("Area is " + t.area());
}
}
```

Output:

```
D:\>javac AbstractAreas.java
D:\>java AbstractAreas
Inside area for Rectangle.
Area is 45.0
Inside are for Triangle.
Area is  40.0
```

→→ Base class Object or The Object class:

Object class: Super class for all the classes in java including user defined classes directly or indirectly.

Importing Object class:

java Library
↓
lang package
↓
Object class

Object class is implicitly (automatically) imported into our source code, because it is in ***"lang"*** package. lang package is also implicitly imported into every java program.

Object class reference can store any reference of any object. This means that a reference variable of type Object can refer to an object of any other class.

→→ **final keyword:** Final is a keyword in Java which generically means, cannot be changed once created. Final behaves very differently for variables, methods and classes.

- A final variable cannot be reassigned once initialized.
- A final method cannot be overridden.
- A final class cannot be extended.

→ Classes are usually declared final for either performance or security reasons.

Final with variables: Final variables work like constants of C-language that can't be altered in the whole program. That is, final variables once created can't be changed and they must be used as it is by all the program code.

Example program:

```
import java.io.*;
class FinalVar
{
  int x=10;
  final int y=20;
 System.out.println("x is:"+x);
 System.out.println("y is:"+y);
 x=30;
 y=40;//error.
 System.out.println("x is:"+x);
 System.out.println("y is:"+y);
}
}
```

Output: [Error]
Cannot assign a value to final variable y

Final with methods: Generally, a super class method can be overridden by the subclass if it wants a different functionality or it can call the same method if it wants the same functionality. If the super class desires that the subclass should not override its method, it declares the method as final. That is, methods declared final in the super class can not be overridden in the subclass (else it is compilation error). But, the subclass can access with its object as usual.

Example program:

```
import java.io.*;
class A
{
final void display()
{
System.out.println("hi");
}
}
class B extends A
{
void display() //cannot override final method
{
super.display();
System.out.println("hello");
}
public static void main(String args[])
{
B b=new B();
b.display();
}
}
```

Output: [Error]
Display() in B cannot override display() in A; overridden method is final.

Final with classes: If we want the class not be sub-classed (or extended) by any other class, declare it final. Classes declared final can not be extended. That is, any class can use the methods of a final class by creating an object of the final class and call the methods with the object (final class object).

Example program:

```
import java.io.*;
final class Demo1
{
public void display()
{
System.out.println("hi");
}
}
public class Demo3 extends Demo1
{
public static void main(String args[])
{
Demo1 d=new Demo1();
d.display();
}
}
```

Output:

```
D:\>javac Demo3.java
Demo3.java:9: cannot inherit from final Demo1
public class Demo3 extends Demo1
                           ^
1 error
```

→→ **Polymorphism:** Polymorphism came from the two Greek words 'poly' means many and 'morphs' means forms. If the same method has ability to take more than one form to perform several tasks then it is called polymorphism. It is of two types:

- Dynamic polymorphism(Runtime polymorphism)
- Static polymorphism(Compile time polymorphism)

→→ **Dynamic Polymorphism:** The polymorphism exhibited at run time is called dynamic polymorphism. In this dynamic polymorphism a method call is linked with method body at the time of execution by JVM. Java compiler does not know which method is called at the time of compilation. This is also known as dynamic binding or run time polymorphism. **Method overloading and method overriding are examples of Dynamic Polymorphism in Java.**

→ **Method Overloading:** Writing two or more methods with the same name with different parameters is called method over loading. In method overloading JVM understands which method is called depending upon the difference in the method parameters. The difference may be due to the following:

Ø There is a difference in the no. of parameters.

void add (int a, int b)	void add (int a, int b, int c)

Ø There is a difference in the data types of parameters.

void add (int a, float b)	void add (double a, double b)

Ø There is a difference in the sequence of parameters.

void swap (int a, char b)	void swap (char a, int b)

```
// overloading of methods --------- Dynamic polymorphism
class Sample
{
void add(int a, int b)
{
System.out.println ("sum of two="+ (a+b));
}
void add(int a, int b, int c)
{
System.out.println ("sum of three="+ (a+b+c));
}
}
class OverLoad
{
public static void main(String[] args)
{
Sample s=new Sample ( );
s.add (20, 25);
s.add (20, 25, 30);
}
}
```

Output:

```
D:\>javac OverLoad.java
D:\java OverLoad
sum of two=45
sum of three=75
```

→→ **Method Overriding:** Writing two or more methods in super & sub classes with same name and same type and same no. of parameters is called method overriding. In method overriding JVM executes a method depending on the type of the object.

```
//overriding of methods ---------- Dynamic polymorphism
class Animal
{
void move()
{
System.out.println ("Animals can move");
}
}
class Dog extends Animal
{
void move()
{
System.out.println ("Dogs can walk and run");
}
}
public class OverRide
{
public static void main(String args[])
{
Animal a = new Animal (); // Animal reference and object
Animal b = new Dog (); // Animal reference but Dog object
a.move (); // runs the method in Animal class
b.move (); //Runs the method in Dog class
}}
```

Output:

```
D:\>javac  OverRide.java
D:\>java OverRide
Animals can move
Dogs can walk and run
```

Achieving method overloading & method overriding using instance methods is an example of dynamic polymorphism.

→→ **Static Polymorphism:** The polymorphism exhibited at compile time is called Static polymorphism. Here the compiler knows which method is called at the compilation. This is also called compile time polymorphism or static binding.

Achieving method overloading & method overriding using static methods is an example of Static Polymorphism.

```
//Static Polymorphism
class Animal
{
static void move ()
{
System.out.println ("Animals can move");
}
}
class Dog extends Animal
{
static void move ()
{
System.out.println ("Dogs can walk and run");
}
}
public class StaticPoly
{
public static void main(String args[])
{
Animal.move ();
Dog.move ();
}
}
```

Output:

```
D:\>javac StaticPoly.java
D:\>java StaticPoly
Animals can move
Dogs can walk and run
```

→→ **Interfaces:** An interface is defined much like as an abstract class (it contains only method declarations).It contains constants and method declarations.

Syntax:

accessspecifier interface interface*name*
{
return-type method-name1(*parameter-list*)*;*
return-type method-name2(*parameter-list*)*;*
datatype varname1 = value;
datatype varname2 = value;
// ...
return-type method-nameN(*parameter-list*)*;*
datatype varnameN = value;
}

When no access specifier is included, then default access results, and the 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. In this case, the interface must be the only public interface.

Note: that the methods are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly **public**.

Example:

```
public interface animal
{
   void display();
}
```

→→ **Implementing Interfaces:**

- Once an **interface** has been defined, one or more classes can implement that interface.
- To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.

Syntax:

class *classname* [extends *superclass*] [implements *interface* [,*interface...*]]

{

// class-body

}

If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Example:

```
public interface Animal
{
void eat();
}
class Dog  implements Animal
{
void eat()
{
System.out.println("Dog eats Chicken");
}
}
public class InterfacDemo
{
public static void main(String args[])
{
Dog d=new Dog();
d.eat();
Animal a;
a=d;
a.eat();
}
```

Output::

```
D:\javac InterfaceDemo.java
D:\java InterfaceDemo
Dog eats Chicken
Dog eats Chicken
```

→→ Variables in Interfaces:

- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.
- When you implement that interface in a class all of those variable names will be in scope as constants. (This is similar to using a header file in C to create a large number of **#define** constants or **const** declarations.)
- If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything.
- It is as if that class was importing the constant fields into the class name space as **final** variables.

Example:

```
interface SharedConstants
{
int NO=0;
int YES=1;
int MAYBE=2;
int LATER=3;
int SOON=4;
int NEVER=5;
}
class Question implements SharedConstants
{
java.util.Random r=new java.util.Random();
int ask()
{
int prob=(int)(100*r.nextDouble());
if(prob<30)
return NO;
else
if(prob<60)
return YES;
else
if(prob<75)
return LATER;
else
if(prob<98)
```

```
return SOON;
else
return NEVER;
}
public static void main(String[] args)
{
Question q=new Question();
System.out.println("First
Question\t::"+q.ask());
System.out.println("Second
Question\t::"+q.ask());
System.out.println("Third
Question\t::"+q.ask());
System.out.println("Fourth
Question\t::"+q.ask());
}
}
```

Output:

```
D:\>javac Question.java
D:\>java Question
First Question      ::1
Second Question     ::1
Third Question      ::3
Fourth Question     ::4
```

→→ **Interfaces Can Be Extended:** One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Example:

```
interface A
{
void meth1();
void meth2();
}
interface B extends A
 {
void meth3();
}
class MyClass implements B
 {
public void meth1()
 {
System.out.println("Method1
Implementation");
}
public void meth2()
{
System.out.println("Method2
Implementation");
}
public void meth3()
{
System.out.println("Method3
Implementation");
}
}
class IFExtend
{
public static void main(String arg[])
 {
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}
}
```

Output:

```
D:\javac IFExtend.java
D:\java IFExtend
Method1 Implementation
Method2 Implementation
Method3 Implementation
```

→→ **Difference between Interface and Class:**

An interface is used to allow unrelated objects to interact with one another, by implementing an agreed upon system of behavior. When a class implements an interface, the class agrees to implement all of the methods defined in the interface. Interfaces are useful since they capture similarity between unrelated objects without forcing a class relationship. Furthermore, interfaces may consist of abstract methods. One class uses an interface by using the **"implements"** keyword.

Classes in Java have its particular fields and methods. Each object is an instance of a class, and follows the class prototype that defines the variables and methods common to all objects of a certain kind. Each instance of a class must be instantiated, after it is declared, unlike in C++. This is usually done with the keyword **"new"**. Classes may have inheritance from other classes, as they do in C++, meaning they inherit all the properties of the parent class. This is usually done with the keyword **"extends"**.

Property	Class	Interface
Instantiation	Can Be Instantiated	Can not be instantiated
Inheritance	A Class can inherit only one Class and can implement many interfaces	An Interface cannot inherit any classes while it can extend many interfaces
Variables	All the variables are instance by default unless otherwise specified	All the variables are static final by default, and a value needs to be assigned at the time of definition
Methods	All the methods should be having a definition unless decorated with an abstract keyword	All the methods are abstract by default and they will not have a definition.

→→ **Interfaces vs. Abstract Classes:** 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 that are given below.

Abstract class	Interface
Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods.
Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
Abstract class can have static methods, main method and constructor.	Interface can't have static methods, main method or constructor.
Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.

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

→→ **Package:** *Packages* are containers for classes that are used to keep the class name space compartmentalized. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions. Packages are of two types Built-in packages and User-defined packages.

Built-in packages are already available in java language which contains classes, interfaces and methods for the programmer to perform any task. Ex: java.lang, java.util etc.

User-defined packages can be created by the users of java which can be imported to other classes and used exactly in the same way as the Built-in packages.

→→ **Creating a Package:** To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will ***belong*** to the specified package.

- The **package** statement defines a name space in which classes are stored.
- If you omit the **package** statement, the class names are put into the default package, which has no name.

Syntax: *package packagename;*

Example: package MyPackage;

- Java uses file system directories to store packages.
- For example, the **class** files for any classes you declare which are part of **MyPackage** must be stored in a directory called **MyPackage**.
- More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package.
- You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period.

The **general form of a multileveled package** statement: *package pkg1[.pkg2[.pkg3]];*

Example:

```
package pack;
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));
}
public static void main(String args[])
{
Addition a=new Addition(23.123,123.23);
a.sum();
}
```

D:\Sub>javac –d . Addition.java

The preceding command means create a package (-d) in the current directory (.) and store Addition.class file there in the package. The package name is specified in the program as pack.

So the Java compiler creates a directory in D:\Sub with the name as pack and stores Addition.class there. Observe it by going to pack sub directory which is created in D:\Sub. So, our package with Addition class is ready.

D:\Sub>java pack.Addition

The preceding command is used for executing the Additon class file which is in pack package.

Sum= 125.3530000

→→ **Finding Packages and CLASSPATH:** ***How does the Java run-time system know where to look for packages that you create?*** The answer has three parts.

*****First,*** by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found.

Example:

```
package pack;
public class Subtraction
{
private double d1,d2;
public Subtraction(double a,double b)
{
d1=a; d2=b;
}
public void subtract()
{
System.out.println("Difference= "+(d1-d2));
}
public static void main(String args[])
{
Subtraction s=new Subtraction(123.123,23.123);
a.sum();
}
}
```

D:\Sub>javac pack/Subtraction.java

The preceding command is used for compiling the Subtraction.java program which is in pack package. This command generates a class file and stores in pack package.

D:\Sub>java pack.Subtraction

The preceding command is used for executing the Subtraction class file which is in pack package.

Difference= 100.00

***Second,** you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.

***Third,** you can use the **-classpath** option with **java** and **javac** to specify the path to your classes.

Example: package MyPack;

In order for a program to find **MyPack,** one of three things must be true. Either the program can be executed from a directory immediately above **MyPack**, or the **CLASSPATH must** be set to include the path to **MyPack**, or the **-classpath** option must specify the path to **MyPack** when the program is run via **java**.

- When the second two options are used, the class path *must not* include **MyPack**, itself.It must simply specify the *path to* **MyPack**.

Example for second case: In a Windows environment, if the path to **MyPack** is C:\MyPrograms\Java\MyPack then the class path to **MyPack** is C:\MyPrograms\Java

The CLASSPATH is an environment variable that tells the Java compiler where to look for class files, to import. CLASSPATH is generally set to a directory.

To see what is there currently in the CLASSPATH variable in your system, you can type in Windows98/2000/Me/NT/XP/Vista:

C:\>echo %CLASSPATH%

rnr; .

Suppose, preceding command has disp1ayed class path as: rnr. This means the current class path is set to rnr directory in C: \ and also to the current directory is represented by dot (.). Our package pack does not exist in either rnr or current directory. Our package exists in D: \sub. This information should be provided to the Java compiler by setting the class path to D: \Sub, as shown here:

C:\>set CLASSPATH=D:\Sub;.;%CLASSPATH%

In the preceding command, we are setting the class path to **Sub** directory and current directory (**.**). And then we typed %CLASSPATH% which means retain the already available class path as it is. This is necessary especially when the class path in your system is already set to an important application that should not be disturbed.

Example:

```
import pack.*;
class Use
{
public static void main(String args[])
{
Addition obj=new Addition obj(10,23.0);
Subtraction s=new Subtraction(213,200.0)
obj.sum();
}
}
C:\>javac Use.java
C:\>java Use
Sum= 33.0
Difference= 13.00
```

→→ Accessing a Package

- Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.
- Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods.
- Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code.
- The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses

- The four access specifiers, **private**, **public**, **protected and default**, provide a variety of ways to produce the many levels of access required by these categories.

	Private	**Default**	**Protected**	**Public**
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Example:

Save as Protection.java in package p1:

```
package p1;
public class Protection
{
int n = 1;
private int n_pri = 2;
protected int n_pro = 3;
public int n_pub = 4;
public Protection()
{
System.out.println("base constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

Save as Derived.java in package p1:

```
package p1;
public class Derived extends Protection
{
public Derived()
 {
System.out.println("derived constructor");
System.out.println("n = " + n);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

Save as SamePackage.java in package p1:

```
package p1;
public class SamePackage
 {
public SamePackage()
{
Protection  p = new Protection();
System.out.println("same package constructor");
System.out.println("n = " + p.n);
System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
} }
```

Main method class for package p1: save in current directory

```
import  p1.*;
public class Demo
{
public static void main(String args[])
{
Protection ob1 = new Protection();
Derived ob2 = new Derived();
SamePackage ob3 = new SamePackage();
}
}
```

Output:

```
D:\>javac p1/Protection.java
D:\>javac p1/Derived.java
D:\>javac p1/SamePackage.java
D:\>javac Demo.java
D:\>java Demo
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
derived constructor
n = 1
n_pro = 3
n_pub = 4
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
same package constructor
n = 1
n_pro = 3
n_pub = 4
```

Save as Protection2.java in package p2:

```
package p2;
public class Protection2 extends p1.Protection
 {
public Protection2()
{
System.out.println("derived other package constructor");
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

Save as OtherPackage.java in package p2:

```
package p2;
public class OtherPackage
 {
public OtherPackage()
{
p1.Protection p = new p1.Protection();
System.out.println("other package constructor");
System.out.println("n_pub = " + p.n_pub);
}
}
```

Main method class for package p2: save in current directory

```
import p2.*;
public class Demo
{
public static void main(String args[])
 {
Protection2 ob1 = new Protection2();
OtherPackage ob2 = new OtherPackage();
}
}
```

Output:

```
D:\>javac p2/OtherPackage.java
D:\>javac p2/Protection2.java
D:\>javac DemoTwo.java
D:\>java DemoTwo
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
derived other package constructor
n_pro = 3
n_pub = 4
base constructor
n = 1
n_pri = 2
n_pro = 3
n_pub = 4
other package constructor
n_pub = 4
```

→→ Importing Packages

- Java includes the **import** statement to bring certain classes, or entire packages, into visibility.
- Once imported, a class can be referred to directly, using only its name.
- The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program.
- If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.
- In a Java source file, **import** statements occur immediately following the **package** statement (If it exists) and before any class definitions.
- This is the general form of the **import** statement:
  import *pkg1*[.*pkg2*].(*classname*|*);

Example:

```
import java.util.Date;
import java.io.*;
import java.util.*;
```

1) *import pack.Addition;*
2) *import pack.*;*

In statement 1, only the Addition class of the package pack is imported into the program and in statement 2, all the classes and interfaces of the package pack are available to the program.

If a programmer wants to import only one class of a package say BufferedReader of java.io package, he can write:

import java.io.BufferedReader;

This is straight and the Java compiler links up the BufferedReader of java.io package with the program. But, if he writes import statement as:

import java.io.;*

In this case, the Java compiler conducts a search for BufferedReader class in java.io package, every time it is used in the rest of the program. This increases load on the compiler and hence compilation time increases. However, there will not be any change in the runtime.

Exception Handling

→→ **Errors in a Java Program:** There are two types of errors, compile time errors and run time errors.

- 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. Detecting and correcting compile-time errors is easy as the Java compiler displays the list of errors with the line numbers along with their description. The programmer can go to the statements, check them word by word and line by line to understand where he has committed the errors.
- Run time errors: The run time errors are the errors that occur at the run-time of the program and cause the abnormal termination of the program. The run time errors are called exceptions. There are the three types of runtime errors.
  - Input errors: Input errors occur if the user provides unexpected inputs to the program. For example, if the program wants an integer and the user provides it the string value. These errors can be prevented from occurring by prompting the user to enter the correct type of values.
  - System errors: System errors or hardware errors occur rarely. These errors occur due to unreliable system software or hardware malfunctions. These errors are beyond a programmer's control.
  - Logical errors: logical errors occur if the program is logically incorrect. These errors either generate incorrect results or terminate program abnormally. For example, a program for adding two numbers requires an addition operator (+), if the program supplies subtraction operator (-) then this generates the incorrect results. To debug these errors the program must be scanned to check the logical statements.

→ **Introduction to Exception Handling:** Java applications are used in embedded system software, which runs inside specialized devices like hand held computers, cellular phones, and etc. in those kinds of applications, it's especially important that software errors be handled strongly. Java offers a solution to these problems with exception handling.

→→ **Concepts of Exception Handling:** An Exception is an abnormal condition that arises during the execution of a program that causes to deviate from the normal flow of execution path. When an exception occurs, it makes the further execution of the program impossible. Thus, an exception can be defined as an event that may cause abnormal termination of the program during its execution.

Exception handling means to handle the exceptions by the programmer to recover the computer from malfunction due to exceptions.

In Java, exception handling is managed via ***five*** keywords: try, catch, throw, throws, and finally.

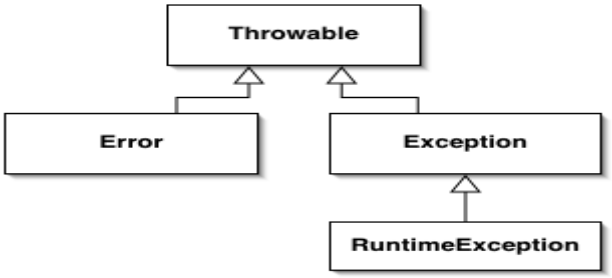
- **try:** The try block is said to monitor the statements enclosed within it and defines the scope of any exception associated with it. It detects the exceptions.
- **catch:** The catch clause contains a series of legal Java statements. These statements are executed if and when the exception handler is invoked. It holds an exception. Catch is known as exception handler which is a piece of code used to deal with the exceptions, either to fix the error or abort execution in a sophisticated way.
- **throw:** To manually throw an exception ,use the keyword throw.
- **throws:** Any exception that is thrown out of a method must be specified as such by a throws clause.
- **finally:** Any code that absolutely must be executed after a try block completes is put in a finally block. After the exception handler has run, the runtime system passes control to the finally block.

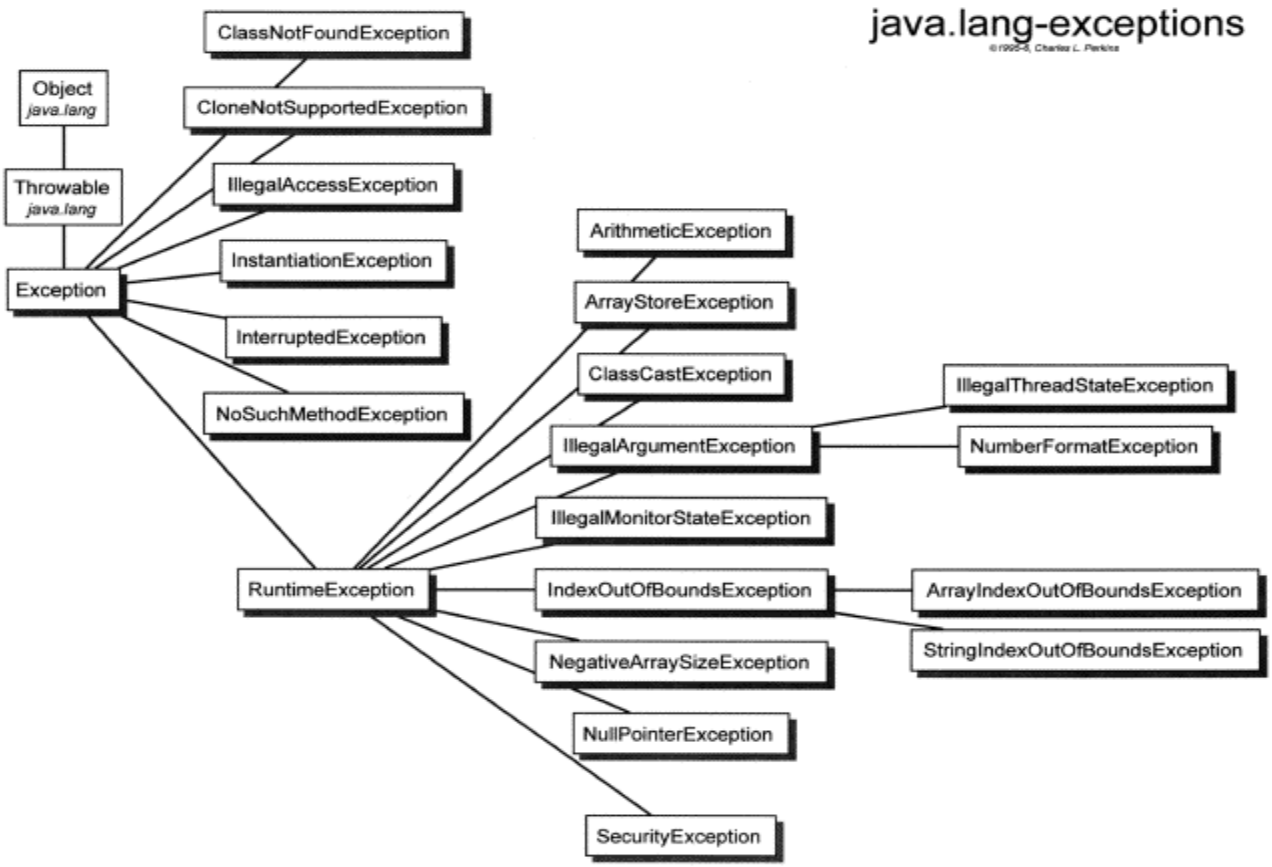
Syntax:

```
try
{
        // block of code to monitor for errors
}
catch (ExceptionType1 id1 )
{
       // exception handler for ExceptionType1
}
catch ( ExceptionType2 id2 )
{
       // exception handler for ExceptionType2
}    .
     .
     .
finally
{
       // statements to execute every time after try block executes
}
```

Here, ExceptionType is the type of exception that has occurred.

→→ **Exception Hierarchy:** All exception types are subclasses of the built-in class *Throwable*, where Throwable is subclass for *'Object'*. Thus, Throwable is at the top of the exception class hierarchy. Immediately below Throwable, there are two subclasses that partition exceptions into two distinct branches.





One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own user defined exception types. There is an important subclass of Exception, called RuntimeException. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by Error, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type Error are used by the Java run-time system

to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

- **java:** JAVA API is a library contains the packages.
- **lang:** lang is a package included in java library. And it is considered as a default package named as language. Implicitly it is imported into every java programs.
- **Object:** Object is a super class of all classes (user defined, pre-defined classes) directly or indirectly. Because it is included in the lang package.
- **Throwable:** Throwable is super class of Errors and Exceptions in java. Throwable is derived from the object class.
- **Error:** Error is a class. This type of errors cannot be handled.
- **Exception:** An abnormal event in a program is called Exception.

There are basically **two** types of Exceptions in the Java program:

Checked Exceptions: Checked exceptions are the exceptions thrown by a method, if it encounters a situation which is not handled by it. All classes that are derived from 'Exception' class, but not RuntimeException' class are checked exceptions. Whenever a method is declared or called it is checked by compiler to determine whether it throws checked exceptions or not.

Programmer should compulsorily handle the checked exceptions in code, otherwise code will not be compiled i.e. the code which may cause checked exception must be specified in try-catch block or throws clause containing the list of checked exception is provided to the method declaration.

- "Checked" means they will be checked at compile time itself.

The example of Checked Exceptions is IOException which should be handled in code compulsorily or else code will throw a Compilation Error.

Example:

```
import java.io.*;
class CEDemo
{
public static void main (String args[]) throws IOException
{
BufferedReader br=new BufferedReader (new InputStreamReader (System.in));
System.out.print ("enter ur name: ");
String name=br.readLine ( );
System.out.println ("Hai "+name);
}
}
```

```
Output:
D:\Seshu\Except>javac CEDemo.java
D:\Seshu\Except>java CEDemo
enter ur name: Seshu
Hai Seshu
```

Unchecked Exceptions: Exceptions which are checked at run time. A java method does not require declaring that it will throw any of the run-time exception. Unchecked exceptions are RuntimeException and any of its subclasses and Class Error and its subclasses also are unchecked. Unchecked runtime exceptions represent conditions that reflect errors in program's logic and cannot be reasonably recovered from at run time. With an unchecked exception, compiler doesn't force programmers either to catch the exception or declare it in a throws clause.

Example:

```
public class REDemo
{
static public void main(String args[])
{
int d[]={1,2};
d[3]=99;
int a=5,b=0,c;
c=a/b;
System.out.println("c is:"+c);
System.out.println("okay");
}
}
```

```
Output:
D:\Seshu\Except>javac CEDemo.java
D:\Seshu\Except>java REDemo
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
        at REDemo.main(CEDemo.java:6)
```

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 Checked Exceptions 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.

Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

→→ Benefits of Exception Handling:

- First, it allows you to fix the error.
- Second, it prevents the program from automatically terminating.
- Third, it adopts the robustness to program.

Exception handling provides the following advantages over "traditional" error management techniques:

- **Separating Error Handling Code from ``regular'' one:** provides a way to separate the details of what to do when something out-of-the-ordinary happens from the normal logical flow of the program code;
- **Propagating Errors up the Call Stack:** lets the corrective action to be taken at a higher level. This allows the corrective action to be taken in the method that calling that one where an error occurs;
- **Grouping Error Types and Error Differentiation: A**llows to create similar hierarchical structure for exception handling so groups them in logical way.

→ **Uncaught Exceptions:**

The following program illustrates what happens when you don't handle the exception when it is raised. This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0
{
public static void main(String args[])
{
int d = 0;
int a = 42 / d;
}
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, no exception handlers are specified, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:6)
```

Notice how the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, **4**, are all included in the simple stack trace. Also, notice that the type of exception thrown is a subclass of **Exception** called **ArithmeticException**, which more specifically describes what type of error happened.

The stack trace will always show the sequence of method invocations that led up to the error. For example, here is another version of the preceding program that introduces the same error but in a method separate from **main( )**:

```
class Exc1
{
static void subroutine()
{
int d = 0;
int a = 10 / d;
}
public static void main(String args[])
{
Exc1.subroutine();
}
}
```

The resulting stack trace from the default handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:6)
at Exc1.main(Exc1.java:10)
```

As you can see, the bottom of the stack is **main**'s line 7, which is the call to **subroutine( )**, which caused the exception at line 4. The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error.

→→ Usage of try and catch:

try: Although the default exception handler provided by the Java run-time system is useful for debugging, a programmer will usually want to handle an exception by themself. Doing so provides two benefits.

- First, it allows you to fix the error.
- Second, it prevents the program from automatically terminating.

To handle run-time errors and monitor the results, simply enclose the code inside a *try* block. If an exception occurs within the *try* block, it is handled by the appropriate exception handler (catch block) associated with the *try* block. If there are no exceptions to be thrown, then try will return the result executing block. A *try* should have one (or more) *catch* blocks or one *finally* block or both. If neither is present, a compiler error occurs which says *try* without *catch* or *finally*.

catch: A catch clause is a group of java statements, enclosed in braces { } which are used to handle a specific exception that has been thrown. Catch clauses should be placed after the try block i.e. catch clause follows immediately after the try block.

A catch clause is specified by the keyword catch followed by a single argument within parenthesis (). The argument type in a catch clause is form the Throwable class or one of its subclasses.

- Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.
- A try and its catch statement form a unit. The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement. A catch statement cannot catch an exception thrown by another try statement (except in the case of nested try statements)
- The statements that are protected by try must be surrounded by curly braces. (That is, they must be within a block.) We cannot use try on a single statement.
- The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.

To illustrate how easily this can be done, the following program includes a *try* block and a *catch* clause that processes the ArithmeticException generated by the division-by-zero error:

```
class Exc2
{
public static void main(String args[])
{
int d, a;
try
{ // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
}
catch (ArithmeticException e)
{ // catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

- Notice that the call to println( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block.

→ **Displaying a Description of an Exception:** Throwable overrides the toString ( ) method (defined by Object). So that it returns a string containing a description of the exception. We can display this description in a println ( ) statement by simply passing the exception as an argument. For example, the catch block in the preceding program can be rewritten like this:

```
catch (ArithmeticException e)
{
System.out.println("Exception: " + e);
a = 0; // set a to zero and continue
}
```

When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:

```
Exception: java.lang.ArithmeticException: / by zero
```

→→ **Multiple catch Statements:**

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, we can specify two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block. The following example traps two different exception types:

Example:

```
class MultiCatch
{
public static void main(String args[])
{
try
{
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
```

```
int c[] = { 1 };
c[42] = 99;
}
catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e);
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

Here is the output generated by running it both ways:

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

This program will cause a division-by-zero exception if it is started with no command-line arguments, since a will equal zero. It will survive the division if we provide a command-line argument, setting a value to something larger than zero. But it will cause an ArrayIndexOutOfBoundsException, since the int array c [] has a length of 1, yet the program attempts to assign a value to c [42].

When we use multiple catch statements, it is important to remember that exception subclasses must come before any of their super classes. This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses.

Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error. For example, consider the following program:

/* this program contains an error. A subclass must come before its superclass in a series of catch statements. If not, unreachable code will be created and a compile-time error will result.*/

```
class SuperSubCatch
```

```
{
```

```
public static void main(String args[])
{
try
{
int a = 0;
int b = 42 / a;
}
catch(Exception e)
{
System.out.println("Generic Exception catch.");
}
/* This catch is never reached because ArithmeticException is a subclass of Exception. */
catch(ArithmeticException e) { // ERROR - unreachable
System.out.println("This is never reached.");
}
}}
```

If we try to compile this program, we will receive an error message stating that the second catch statement is unreachable because the exception has already been caught. Since ArithmeticException is a subclass of Exception, the first catch statement will handle all Exception-based errors, including ArithmeticException. This means that the second catch statement will never execute. To fix the problem, reverse the order of the catch statements.

→→ **Nested try Statements:** The try statement can be nested. That is, a try statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception. Here is an example that uses nested try statements:

Example:

```
class NestTry
{
public static void main(String args[])
{
try
{
int a = args.length;
/* If no command-line args are present, the following statement will generate a divide-by-zero exception. */
int b = 42 / a;
System.out.println("a = " + a);
try
{ // nested try block
/* If one command-line arg is used,
```

```
then a divide-by-zero exception
will be generated by the following code. */
if(a==1) a = a/(a-a); // division by zero
/* If two command-line args are used, then generate an out-of-bounds exception. */
if(a==2)
{
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index out-of-bounds: " + e);
}
}
catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e);
}
}
}
```

As you can see, this program nests one try block within another. The program works as follows.

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException:42
```

When we execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer try block. Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested try block. Since the inner block does not catch this exception, it is passed on to the outer try block, where it is handled. If we execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block.

→→ **throw:** ‘throw’ is a java keyword used in exception handling. Generally a try block checks for arrival of error and when an error occurs it throws the error and it is caught by the catch statement and

then appropriate action will take place. Only the expressions thrown by the java run-time system are being caught, but throw keyword allows a program to throw an exception explicitly.

Syntax: throw ThrowableInstance;

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable. Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.

There are two ways you can obtain a Throwable object:

- using a parameter in a catch clause:: catch(NullPointerException e) { throw e; }
- Creating one with the new operator:: throw new ArithmeticException ();

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.

The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

Example:

```
class ThrowDemo
{
static void demoproc()
 {
try
 {
throw new NullPointerException("demo");
}
catch(NullPointerException e)
 {
System.out.println("Caught inside demoproc.");
throw e; // rethrow the exception
}
}
public static void main(String args[])
 {
```

```
try
{
demoproc();
}
catch(NullPointerException e)
{
System.out.println("Recaught: " + e);
}
}
}
```

Here is the resulting output:
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo

This program gets two chances to deal with the same error. First, main( ) sets up an exception context and then calls demoproc( ). The demoproc( )method then sets up another exception-handling context and immediately throws a new instance of NullPointerException, which is caught on the next line. The exception is then rethrown.

throw new NullPointerException("demo");

Here, new is used to construct an instance of NullPointerException. Many of Java's built- in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to print( ) or println( ). It can also be obtained by a call to getMessage( ), which is defined by Throwable.

→→ 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 necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

Syntax:

```
returntype 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. Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a throws clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.
class ThrowsDemo
{
static void throwOne()
{
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[])
{
ThrowsDemo.throwOne();
}
}
```

To make this example compile, you need to make two changes.

- First, you need to declare that throwOne( ) throws IllegalAccessException.
- Second, main( ) must define a try/catch statement that catches this exception.

The corrected example is shown here:

```
// This is now correct.
class ThrowsDemo
{
static void throwOne() throws IllegalAccessException
{
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[])
{
try
{
throwOne();
```

```
}
catch (IllegalAccessException e)
{
System.out.println("Caught " + e);
}
}
}
```

Here is the output generated by running this example program:
inside throwOne
caught java.lang.IllegalAccessException: demo

→What is the difference between throws and throw?

throws clause is used when the programmer does not want to handle the exception and throw it out of a method. throw clause is used when the programmer wants to throw an exception explicitly and wants to handle it using catch block. Hence, throws and throw are contradictory.

→→ finally:

Whenever an exception occurs, the exception affects the flow of execution of the program. Sometimes some blocks may be bypassed by exception handling. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The finally keyword is designed to address this contingency.

- finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block.
- The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception.
- Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns.
- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.
- The finally clause is optional. However, each try statement requires at least one catch or a finally clause.

→Here is an example program that shows three methods that exit in various ways, none without executing their finally clauses:

```
// Demonstrate finally.
class FinallyDemo
 {
// Through an exception out of the method.
static void procA()
{
try
{
System.out.println("inside procA");
throw new RuntimeException("demo");
}
finally
{
System.out.println("procA's finally");
}
}
// Return from within a try block.
static void procB()
 {
try
{
System.out.println("inside procB");
return;
}
finally
{
System.out.println("procB's finally");
}
}


// Execute a try block normally.
static void procC()
 {
try
 {
System.out.println("inside procC");
}
finally
 {
System.out.println("procC's finally");
}
}
public static void main(String args[])
 {
try
{
procA();
}
catch(Exception e)
{
System.out.println(“Exception Caught”);
}
}
procB();
procC();
}
}
```

Output:
inside procA
procA’s finally
Exception caught
inside procB
procB’s finally
inside procC
procC’s finally

- In this example, procA( ) prematurely breaks out of the try by throwing an exception.
- The finally clause is executed on the way out. procB( )’s try statement is exited via a return statement.
- The finally clause is executed before procB( ) returns. In procC( ), the try statementexecutes normally, without error. However, the finally block is still executed.

NOTE: If a finally block is associated with a try, the finally block will be executed upon conclusion of the try.

→→ **Creating own Exception Sub Classes:** Although Java's built-in exceptions handle most common errors, we will probably want to create our own exception types to handle situations specific to our applications. This is quite easy to do: just define a subclass of Exception (which is, of course, a subclass of Throwable). Our subclasses don't need to actually implement anything—it is their existence in the type system that allows us to use them as exceptions.

The Exception class does not define any methods of its own. It does, of course, inherit those methods provided by Throwable. Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them. They are shown in Table.

We may also wish to override one or more of these methods (shown in table) in exception classes that you create.

Exception defines 2 constructors.

- Exception( )
- Exception(String msg)

The first form creates an exception that has no description. The second form lets us specify a description of the exception. Although specifying a description when an exception is created is often useful, sometimes it is better to override toString( ). Here's why: The version of toString( ) defined by Throwable (and inherited by Exception) first displays the name of the exception followed by a colon, which is then followed by your description. By overriding toString( ), we can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.

Method	Description
Throwable fillInStackTrace( )	Returns a **Throwable** object that contains a completed stack trace. This object can be rethrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of **StackTraceElement**. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The **StackTraceElement** class gives your program access to information about each element in the trace, such as its method name.
Throwable initCause(Throwable *causeExc*)	Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream *stream*)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter *stream*)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement *elements*[ ])	Sets the stack trace to the elements passed in *elements*. This method is for specialized applications, not normal use.
String toString( )	Returns a **String** object containing a description of the exception. This method is called by **println( )** when outputting a **Throwable** object.

The Methods Defined by **Throwable**

The following example declares a new subclass of Exception and then uses that subclass to signal an error condition in a method. It overrides the toString( ) method, allowing a carefully tailored description of the exception to be displayed.

```
// This program creates a custom exception type.
class MyException extends Exception
{
private int detail;
MyException(int a)
{
detail = a;
}
public String toString()
{
return "MyException[" + detail + "]";
}
}
class ExceptionDemo
 {
static void compute(int a) throws MyException
{
System.out.println("Called compute(" + a + ")");
if(a > 10)
throw new MyException(a);
System.out.println("Normal exit");
}
public static void main(String args[])
{
try
{
compute(1);
```

```
compute(20);
}
 catch (MyException e)
{
System.out.println("Caught " + e);
}
}
```

```
}

Here is the result:
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

This example defines a subclass of Exception called MyException. This subclass is quite simple: it has only a constructor plus an overloaded toString( ) method that displays the value of the exception. The ExceptionDemo class defines a method named compute( ) that throws a MyException object. The exception is thrown when compute( )'s integer parameter is greater than 10. The main( ) method sets up an exception handler for MyException, then calls compute( ) with a legal value (less than 10) and an illegal one to show both paths through the code.