Inheritance in Java

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.

The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

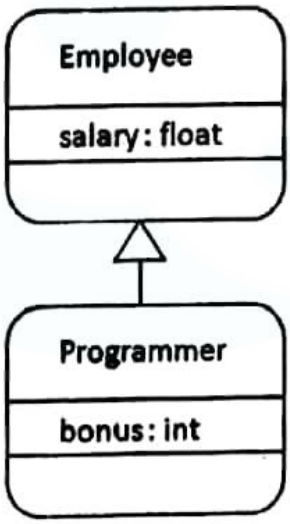
Syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called parent or super class and the new class is called child or subclass.

Java Inheritance Example



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. Relationship between two classes is **Programmer IS-A Employee**.It means that Programmer is a type of Employee.

ANUSREE K ,CSE DEPT.
anusreek@sahrdaya.ac.in

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Programmer salary is:40000.0
Bonus of programmer is:10000

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

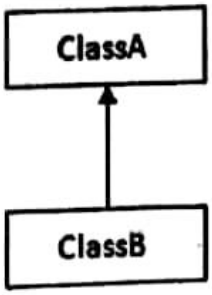
Types of inheritance in java

In Java we have 5 different types of Inheritance. Namely,

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance

Note: In java programming, multiple and hybrid inheritance is supported through interface only.

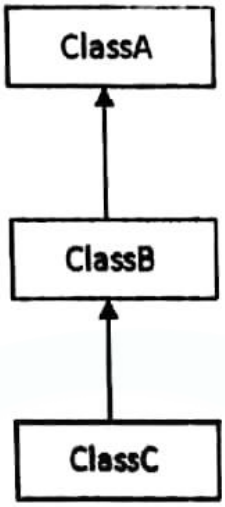
Single inheritance: In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



1) Single

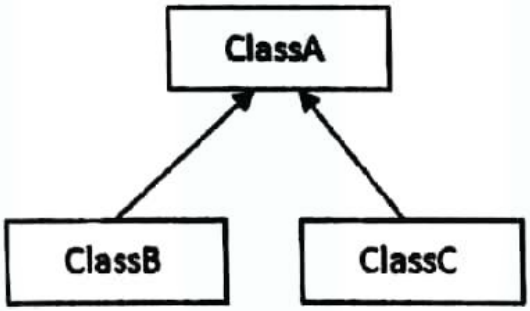
anusreek@sahrdaya.ac.in

Multilevel inheritance: In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



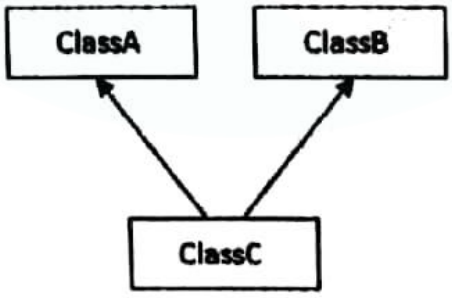
2) Multilevel

Hierarchical inheritance: In this type of inheritance, multiple derived classes inherits from a single base class.



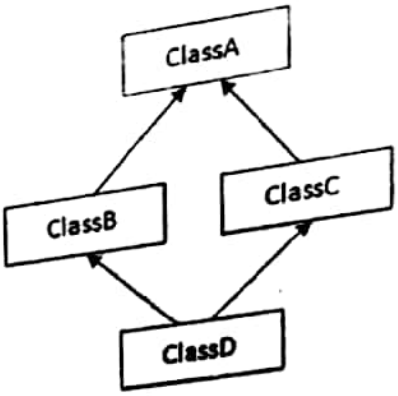
3) Hierarchical

Multipleinheritance: In this type of inheritance a single derived class may inherit from two or more than two base classes.Multiple inheritance is not supported in java through class.



4) Multiple

Hybrid inheritance: Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.



5) Hybrid

Single Inheritance

```
class Animal
{
void eat()
{
System.out.println("eating...");
}
}
class Dog extends Animal
{
void bark()
{
System.out.println("barking...");
}
}
class TestInheritance
{
public static void main(String args[])
{
Dog d=new Dog();
d.bark();
d.eat();
}}
```

Multilevel Inheritance

```
class Animal
{
void eat()
{
System.out.println("eating...");
}
}
class Dog extends Animal
{
void bark()
{
System.out.println("barking...");
}
}
class BabyDog extends Dog
{
void weep()
{
System.out.println("weeping...");
}
}
class TestInheritance2
{
public static void main(String args[])
{
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}
}
```

Hierarchical Inheritanc

```
class Animal
{
void eat()
{
System.out.println("eating...");
}
}
class Dog extends Animal
{
void bark()
{
System.out.println("barking...");
}
}
class Cat extends Animal
{
void meow()
{
System.out.println("meowing...");
}
}
class TestInheritance3
{
public static void main(String args
{
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}
}
```

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.

Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 Public Static void main(String args[]){
  C obj=new C();
  obj.msg();//Now which msg() method would be invoked?
}
}
```

Output:Compile Time Error

This keyword in java

There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.

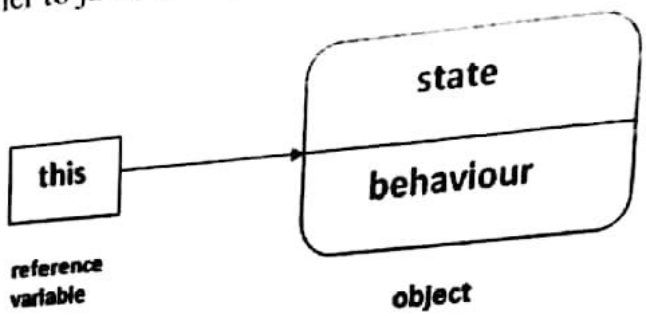
Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable..
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

anusreek@sahrdaya.ac.in

Suggestion: If you are beginner to java, lookup only three usage of this keyword.



This: to refer current class instance variable

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Super keyword in java

The **super** keyword in java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of java super Keyword

1. **super can be used to refer immediate parent class instance variable.**

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Animal{String color="white";}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}}
```

2. **super can be used to invoke immediate parent class method.**

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
void work(){  super.eat();
bark();
}
}
```

3. **super() can be used to invoke immediate parent class constructor.**

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}}
```

super example: real use

Let's see the real use of super keyword. Here, Emp class inherits Person class so all the property of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

```
class Person{
int id;
String name;
Person(int id,String name){
this.id=id;
this.name=name; } }
class Emp extends Person{
float salary;
Emp(int id,String name,float salary){
super(id,name);//reusing parent constructor
this.salary=salary; }
void display(){System.out.println(id+" "+name+" "+salary);} }
class TestSuper5{
public static void main(String[] args){
Emp e1=new Emp(1,"ankit",45000f);
e1.display();
}}
```

Output:

1 ankit 45000

Final Keyword in Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. Java final variable: If you make any variable as final, you cannot change the value of final variable(It will be constant)

   ```
   final int speedlimit=90;//final variable
   ```

2. Java final method: If you make any method as final, you cannot override it.

   ```
   final void run()
   ```

3. Java final class:If you make any class as final, you cannot extend it.

   ```
   final class Bike{}
   ```

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.

In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method is same and there is IS-A relationship between the classes, so there is method overriding.

```
class Vehicle{
void run(){System.out.println("Vehicle is running");}
}
class Bike2 extends Vehicle{
void run(){System.out.println("Bike is running safely");}
public static void main(String args[]){
Bike2 obj = new Bike2();
obj.run();
}
```

Output: Bike is running safely

Can we override static method?

No, static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

anusreek@sahrdaya.ac.in

ANUSREE K ,CSE DEPT.

Why we cannot override static method?

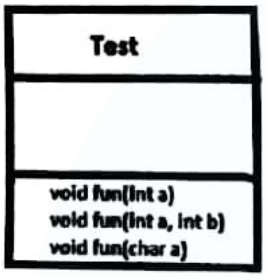
because static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area.

Can we override java main method?

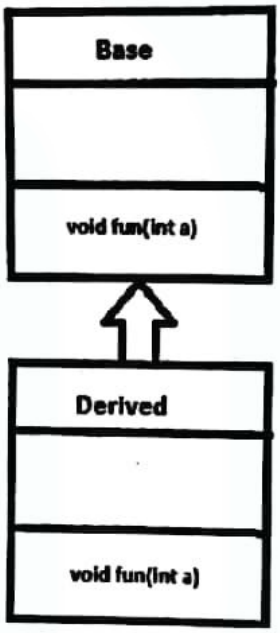
No, because main is a static method.

Difference between method overloading and method overriding in java

- Overloading is about same function have different signatures. Overriding is about same function, same signature but different classes connected through inheritance.



Overloading



Overriding

Method Overloading	Method Overriding
Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the special implementation* of the method that is already provided by its super class.
Method overloading is performed *within class.*	Method overriding occurs *in two classes* that have IS (inheritance) relationship.
In case of method overloading, *parameter must be different.*	In case of method overriding, *parameter must be same*

Method overloading is the example of *compile time polymorphism.*

Method overriding is the example of *run time polymorphism.*

In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.

Return type must be same or covariant in method overriding.

Java Method Overloading example

```
class OverloadingExample{
static int add(inta,int b){return a+b;}
static int add(inta,intb,int c){return a+b+c;}
}
```

Java Method Overriding example

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
}
```

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

Abstract class

A class that is declared as abstract is known as **abstract class.** It needs to be extended and its method implemented. It cannot be instantiated.

abstract class A{}

Abstract method

A method that is declared as abstract and does not have implementation is known as abstract method.

```
abstract void printStatus();//no body and abstract
```

File: TestBank.java

```
abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
int getRateOfInterest(){return 8;}
}
class TestBank{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}}
```

Output:

Rate of Interest is: 7 %
Rate of Interest is: 8 %

Interface in Java

An **interface in java** is a blueprint of a class. It has static constants and abstract methods.

The interface in java is **a mechanism to achieve abstraction.** There can be only abstract methods in the java interface not method body. It is used to achieve abstraction and multiple inheritance in Java.

Java Interface also **represents IS-A relationship.**

It cannot be instantiated just like abstract class.

Why use Java interface?

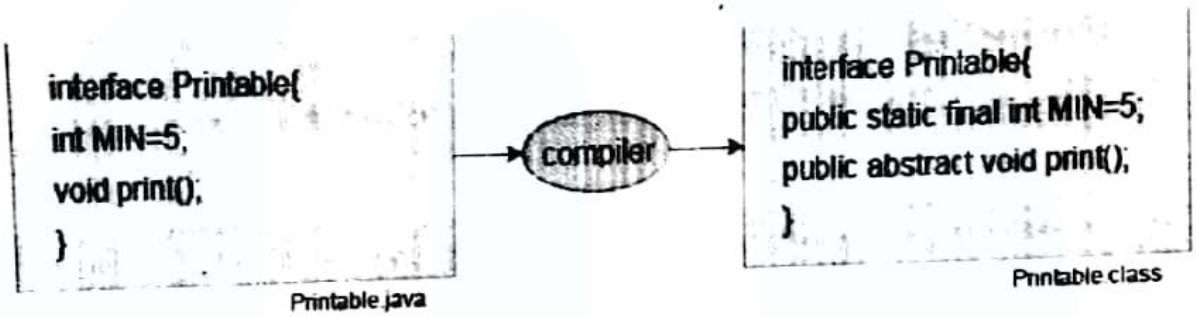
There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

Internal addition by compiler

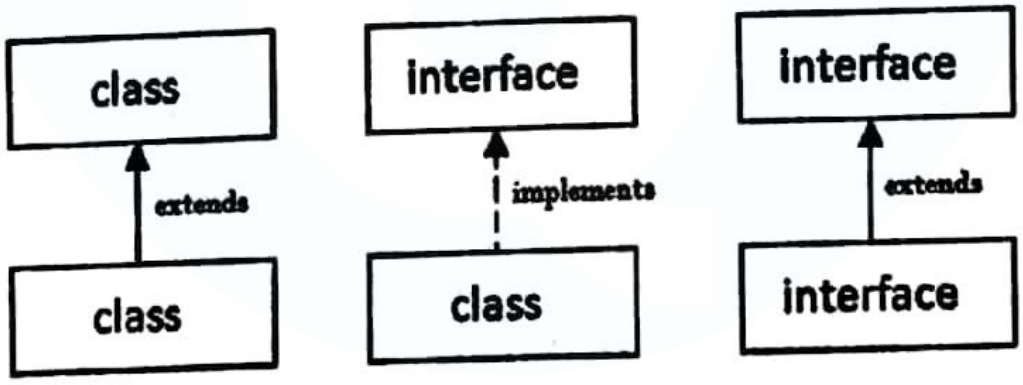
> ***The java compiler adds public and abstract keywords before the interface method. More, it adds public, static and final keywords before data members.***

In other words, Interface fields are public, static and final by default, and methods are public and abstract.



Understanding 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.**



Java Interface Example: Drawable

In this example, Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In real scenario, interface is defined by someone but implementation is provided by different implementation providers. And, it is used by someone else. The implementation part is hidden by the user which uses the interface.

File: TestInterface1.java

```
//Interface declaration: by first user
```

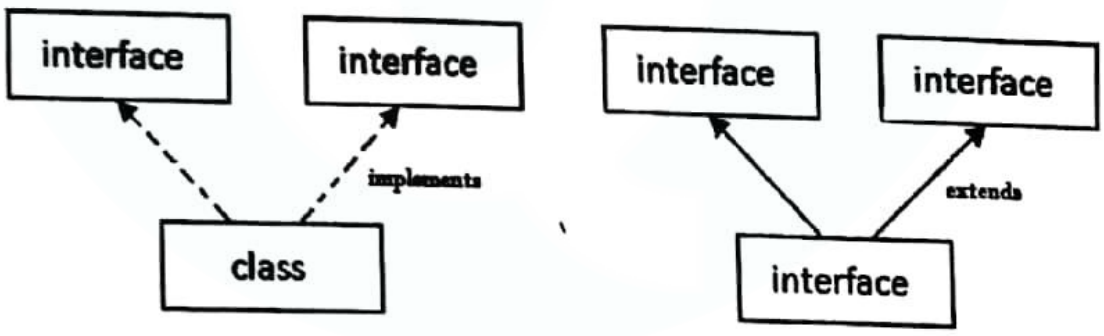
```
interface Drawable{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
d.draw();
}}
```

Output:

drawing circle

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Multiple Inheritance in Java

```
interface Printable{
void print();
}
```

```
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

Output: Hello
Welcome

Q) Multiple inheritance is not supported through class in java but it is possible by interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class because of ambiguity. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

```
interface Printable{
void print();
}
interface Showable{
void print();
}
class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
 }
}
```

Output:Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can have **abstract and non-abstract**methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance.**	Interface **supports multiple inheritance.**
3) Abstract class **can have final, non-final, static and non-static variables.**	Interface has **only static and final variables.**
4) Abstract class **can provide the implementation of interface.**	Interface **can't provide the implementation of abstract class.**
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

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

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

```
//Creating interface that has 4 methods
interface A{
```


anusreek@sahrdaya.ac.in

```
void a();//bydefault, public and abstract
void b();
void c();
void d();
}

//Creating abstract class that provides the implementation of one method of A interface
abstract class B implements A{
public void c(){System.out.println("I am C");}
}

//Creating subclass of abstract class, now we need to provide the implementation of rest of the m
ethods
class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}

//Creating a test class that calls the methods of A interface
class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
}}
```

Output:

```
I am a
I am b
I am c
I am d
```

Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two form, built-in package and user-defined package.

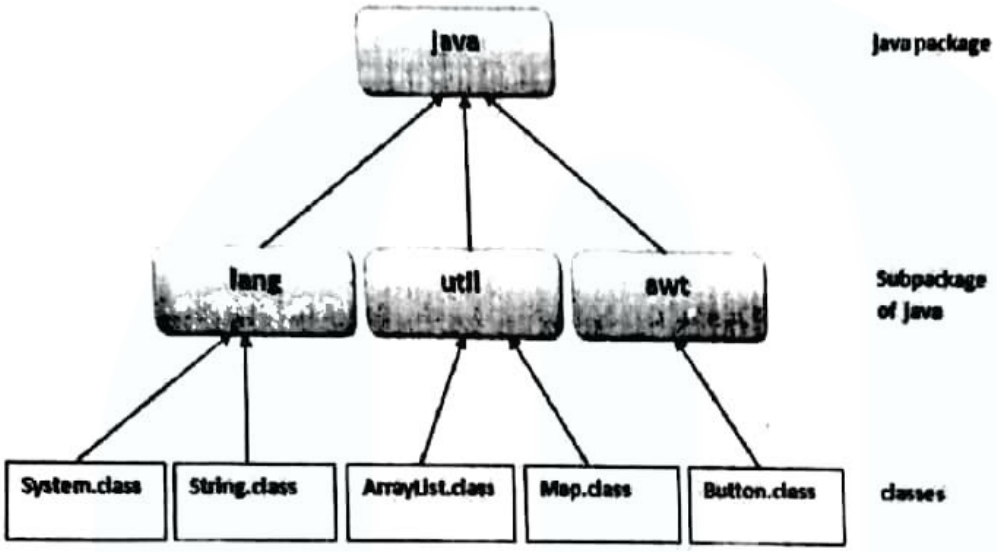
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



Simple example of java package

The **package keyword** is used to create a package in java.

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
   System.out.println("Welcome to package");
  }
}
```

How to access package from another package?

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

1. import package.*;

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

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

Output: Hello

2) Using packagename.classname

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

Example of package by import package.classname

```
//save by A.java
        package pack;
        public class A{
          public void msg(){System.out.println("Hello");}
        }
```

```
//save by B.java
package mypack;
import pack.A;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
 }
```

Output:Hello

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

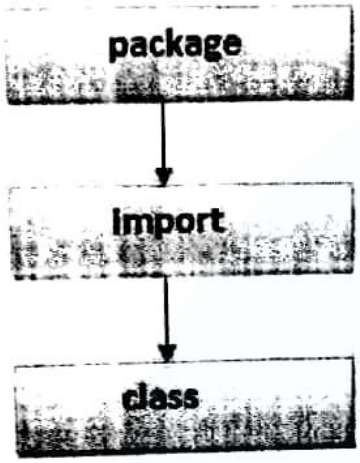
```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
 }
```

Output:Hello

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.



Exception Handling in Java

Exception
An exception 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.

What is exception handling

The **exception handling in java** is one of the powerful *mechanisms to handle the runtime errors* so that normal flow of the application can be maintained.

The purpose of exception handling mechanism is to provide a means to detect and report an "exceptional circumstance"so that appropriate action can be taken.

The error handling code that performs the following tasks:

1. Find the problem(**Hit** the exception)
2. inform that an error has occurred(**Throw** the exception)
3. Receive the error information(**Catch** the exception)
4. Take corrective actions(Handle the exception)

The error handling code basically consists of two segments, one to detect errors and to throw exceptions and the other to catch exceptions and to take appropriate actions.

Checked vs Unchecked Exceptions in Java

In Java, there two types of exceptions:

- **Checked exceptions –**
  - A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions.
  - These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions with the help of try-catch blocks.
  - Checked exceptions are extended from the java.lang.Exception class.
- **Unchecked exceptions –**
  - Unchecked exceptions are not checked at compile-time rather they are checked at runtime, These are also called as **Runtime Exceptions**
  - These exceptions are not essentially handled in the program code; instead the JVM handles such exceptions.

Built-in Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.

Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

1. **ArithmeticException**
   It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundException**
   It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **ClassNotFoundException**
   This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException**
   This Exception is raised when a file is not accessible or does not open.
5. **IOException**
   It is thrown when an input-output operation failed or interrupted
6. **InterruptedException**
   It is thrown when a thread is waiting , sleeping , or doing some processing , and it is interrupted.
7. **NoSuchFieldException**
   It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException**
   It is thrown when accessing a method which is not found.
9. **NullPointerException**
   This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException**
    This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException**
    This represents any exception which occurs during runtime.

12. StringIndexOutOfBoundsException
It is thrown by String class methods to indicate that an index is either negative than the size of the string

Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws

Java try-catch

Java try block is used to enclose the code that might throw an exception. It must be used within the method.Java try block must be followed by either catch or finally block.

Java catch block is used to handle the Exception. It must be used after the try block only.

Syntax of java try-catch

```
try
{
//code that may throw exception
}
catch(ExceptionName e)
{
// Catch block
}
```

Syntax of try-finally block

```
try
{
//code that may throw exception
}
finally
{
// finally block
}
```

Java finally block

- **Java finally block** is a block that is used *to execute important code* such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.

For each try block there can be zero or more catch blocks, but only one finally block.

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.
        }
public class TestFinallyBlock2{
 public static void main(String args[]){
 try{
  int data=25/0;
  System.out.println(data);
 }
 catch(ArithmeticException e){System.out.println(e);}
 finally{System.out.println("finally block is always executed");}
 System.out.println("rest of the code...");
 }
}
```

Output:Exception in thread main java.lang.ArithmeticException:/ by zero
finally block is always executed
rest of the code...

Multiple Catch Blocks

You can use multiple catch block with a single try.

Syntax

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


anusreek@sahrda

```
    // Catch block
  } catch (ExceptionType3 e3) {
    // Catch block
  }
```

Example of java multi-catch block:

```
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...");
}
}
```

Output:
task1 completed
rest of the code...

Java Nested try block

The try block within a try block is known as nested try block in java.

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
....
try
{
  statement 1;
  statement 2;
  try
  {
    statement 1;
    statement 2;
  }
```

```
        catch(Exception e)
        {
        }
    }
    catch(Exception e)
    {
    }
    ....
```

Java nested try example

```
class Excep6{
public static void main(String args[]){
try{
try{
System.out.println("going to divide");
int b =39/0;
}catch(ArithmeticException e){System.out.println(e);}
try{
int a[]=new int[5];
a[5]=4;
}catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}
System.out.println("other statement);
}catch(Exception e){System.out.println("handeled");}
System.out.println("normal flow..");
}
}
```

User-Defined Exceptions

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, user can also create exceptions which are called 'user-defined Exceptions'.

Throwing Our Own Exceptions

throw our own Exceptions by using the keyword throw as follows

throw new Throwable_subclass;

Examples:
throw new ArithmeticException();
throw new NumberFormatException();

Java program to demonstrate user defined exception

```
import java.lang.Exception;
class MyException extends Exception
{
MyException(String message)
{
super(message);
}
}
class Test
{
    public static void main(Strings args[])
    {
        int x=5,y=1000;
try
{
float z=(float)x/(float)y;
if(z<0.01)
{
throw new MyException("Number is too small");
}
}
        catch(MyException e)
{
System.out.println(e);
}
finally
{
System.out.println("I am always here");
}
    }
}
```

Java throw keyword

- The Java throw keyword is used to explicitly throw an exception.
- We can throw either checked or unchekedexception in java by throw keyword.

Java throw keyword example

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow
{
        static void validate(int age)
        {
                if(age<18)
                        throw new ArithmeticException("not valid");
```

```
            else
                System.out.println("welcome to vote");
    }
    public static void main(String args[])
    {
        validate(13);
        System.out.println("rest of the code...");
    }
}
```

Output:
Exception in thread main java.lang.ArithmeticException:not valid

Java throws keyword

- The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.
- It provides information to the caller of the method about the exception.

Syntax of java throws

```
return_type method_name() throws exception_class_name
{
//method code
}
```

Java throws keyword example

```
public class Example1
{
int division(int a, int b) throws ArithmeticException
{
        int t = a/b;
        return t;
}
public static void main(String args[])
{
        Example1 obj = new Example1();
        try
{
        System.out.println(obj.division(15,0));
        }
        catch(ArithmeticException e)
{
        System.out.println("You shouldn't divide number by zero");
        }
}
}
```

Output:
You shouldn't divide number by zero

Important points to remember about throws keyword:

- throws keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless.
- throws keyword is required only to convince compiler and usage of throws keyword does not prevent abnormal termination of program.
- By the help of throws keyword we can provide information to the caller of the method about the exception.

Note the following –

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.
- We can't have catch or finally clause without a try statement.
- A try statement should have either catch block or finally block, it can have both blocks.
- We can't write any code between try-catch-finally block.
- We can have multiple catch blocks with a single try statement.
- try-catch blocks can be nested similar to if-else statements.
- We can have only one finally block with a try-catch statement.
- In a method, there can be more than one statements that might throw exception, So put all these statements within its own **try** block and provide separate exception handler within own **catch** block for each of them.
- If an exception occurs within the **try** block, that exception is handled by the exception handler associated with it. To associate exception handler, we must put **catch** block after it. There can be more than one exception handlers. Each **catch** block is a exception handler that handles the exception of the type indicated by its argument. The argument, ExceptionType declares the type of the exception that it can handle and must be the name of the class that inherits from **Throwable** class.
- For each try block there can be zero or more catch blocks, but **only one** finally block.
- The finally block is optional. It always gets executed whether an exception occurred in try block or not . If exception occurs, then it will be executed after **try and catch blocks.** And if exception does not occur then it will be executed after the **try** block. The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

Java Exception Handling Keywords

Java provides specific keywords for exception handling purposes, we will look after them first and then we will write a simple program showing how to use them for exception handling.

throw – We know that if any exception occurs, an exception object is getting created and then Java runtime starts processing to handle them. Sometime we might want to generate exception explicitly in our code, for example in a user authentication program we should throw exception to client if the password is null. throw keyword is used to throw exception to the runtime to handle it.

throws – When we are throwing any exception in a method and not handling it, then we need to use throws keyword in method signature to let caller program know the exceptions that might be thrown by the method. The caller method might handle these exceptions or propagate it to it's caller method using throws keyword. We can provide multiple exceptions in the throws clause and it can be used with main() method also.

try-catch – We use try-catch block for exception handling in our code. try is the start of the block and catch is at the end of try block to handle the exceptions. We can have multiple catch blocks with a try and try-catch block can be nested also. catch block requires a parameter that should be of type Exception.

finally – finally block is optional and can be used only with try-catch block. Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use finally block. finally block gets executed always, whether exception occurred or not.

How Programmer handles an exception?

Customized Exception Handling: Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Briefly, here is how they work. Program statements that you think can raise exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.