

UNIT-II

INHERITANCE

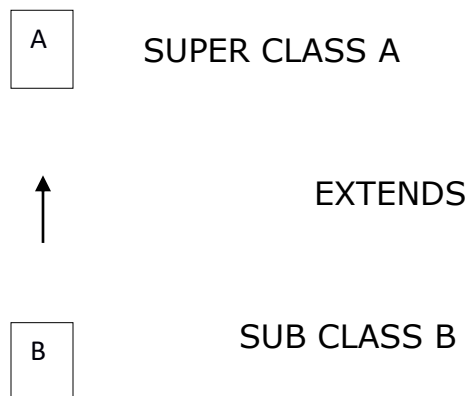
Inheritance is the mechanism of deriving new class from old one, old class is known as superclass and new class is known as subclass. The subclass inherits all of its instances variables and methods defined by the superclass and it also adds its own unique elements. Thus we can say that subclass are specialized version of superclass.

Types of Inheritance:

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

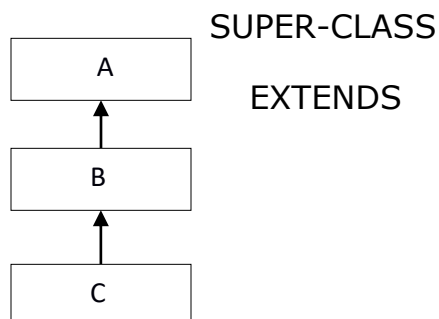
1. Single inheritance:

Here One subclass is deriving from one super class.



2. Multilevel Inheritance:

In multilevel inheritance the class is derived from the derived class.



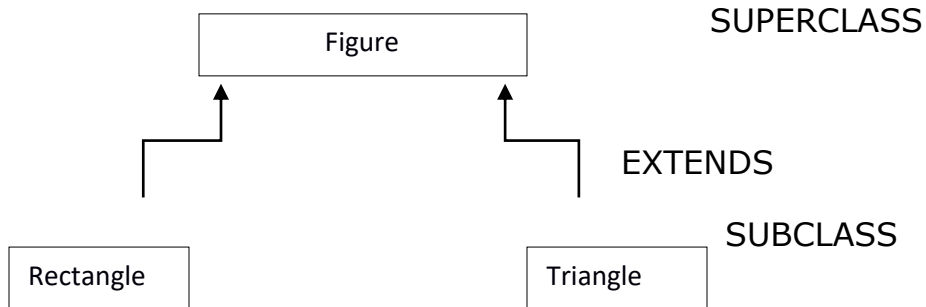
SUB-CLASS

EXTENDS

SUB-SUBCLASS

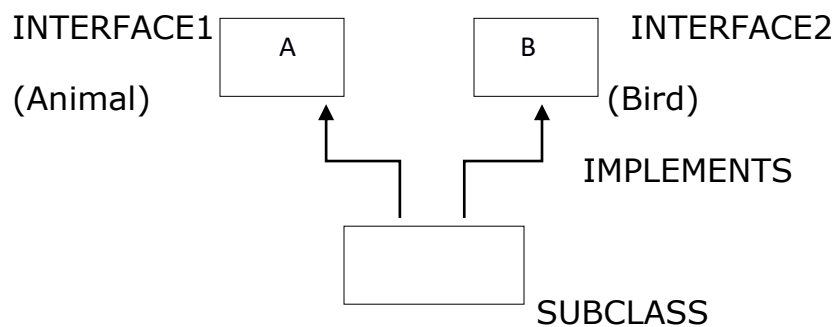
3. Hierarchical Inheritance:

Only one base class but many derived classes.



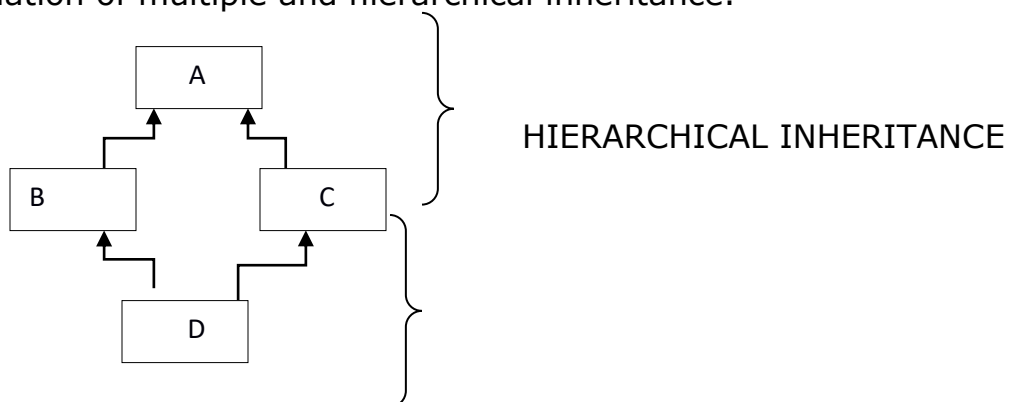
4. Multiple Inheritance:

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



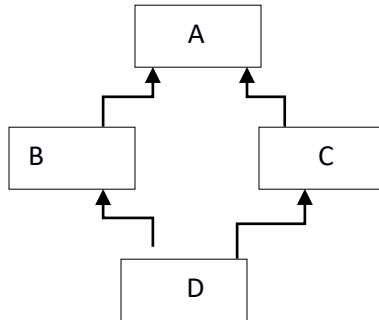
5. Hybrid Inheritance:

It is a combination of multiple and hierarchical inheritance.



MULTIPLE INHERITANCE

6. Multipath Inheritance:



Benefits of Inheritance:

The benefits of inheritance are as follows:

- Increased reliability
- Software reusability
- Code sharing
- To create software components
- Consistency of interface
- Polymorphism
- Information hiding
- Rapid prototyping

Increased Reliability: If a code is frequently executed then it will have very less amount of bugs, compared to code that is not frequently executed.(error free code)

Software reusability: properties of a parent class can be inherited by a child class. But, it does not require to rewrite the code of the inherited property in the child class. In OOPs, methods can be written once but can be reused.

Code sharing: At one level of code sharing multiple projects or users can use a single class.

Software components: Programmers can construct software components that are reusable using inheritance.

Consistency of interfaces: when multiple classes inherit the behavior of a single super class all those classes will now have the same behavior.

Polymorphism: OOPS follows the bottom-up approach. And abstraction is high at top, these exhibit the different behaviors based on instances.

Information hiding: interfaces's or abstracts classes's methods definition is in super class, methods implementation is in subclasses. Means we know what to do but not how to do.

Rapid prototyping: By using the same code for different purposes, we can reduce the lengthy code of the program.

Cost of inheritance:

Following are the costs of inheritance:

- Program size: If the cost of memory decreases, then the program size does not matter. Instead of limiting the program sizes, there is a need to produce code rapidly that has high quality and is also error-free.
- Execution speed: The specialized code is much faster than the inherited methods that manage the random subclasses.
- Program complexity: Complexity of a program may be increased if inheritance is overused.
- Message-passing: The cost of message passing is very less when execution speed is considered.

Member access rules:

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

	Private	No Modifier	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

Table : class member access

We can protect the data from unauthorized access. To do this ,we are using access specifiers.

An access specifier is a keyword that is used to specify how to access a member of a class or the class itself. There are four access specifiers in java:

private: private members of a class are not available outside the class.

public: public members of a class are available anywhere outside the class.

protected: If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.

default: if no access specifier is used then default specifier is used by java compiler. Default members are available outside the class. Specification, it is visible to subclasses as well as to other classes in the same package. This is the default access.

Class Hierarchy

- Good class design puts all common features as high in the hierarchy as reasonable
- The class hierarchy determines how methods are executed
- inheritance is transitive
 - An instance of class Parrot is also an instance of Bird, an instance of Animal, ..., and an instance of class Object

Base Class Object

- In Java, all classes use inheritance.
- If no parent class is specified explicitly, the base class **Object** is implicitly inherited.
- All classes defined in Java, is a child of **Object** class, which provides
- minimal functionality guaranteed to be common to all objects.

Methods defined in Object class are;

- 1.equals(Object obj) Determine whether the argument object is the same as the receiver.
- 2.getClass() Returns the class of the receiver, an object of type Class.
- 3.hashCode() Returns a hash value for this object. Should be overridden when the equals method is changed.
- 4.toString() Converts object into a string value. This method is also often overridden.

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.

1. `super (args-list)` : calls the Super class's constructor.
2. `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.

Use: Overridden methods allow Java to support Run-time polymorphism. This leads to Robustness by Reusability.

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)`

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;  
    }  
    double area()  
    {
```


```
    System.out.println("Area for figure is undefined");  
    return 0;  
}
```

```
}
```

```
class Rectangle extends Figure
```

```
{
```

```
    Rectangle(double a,double b)
```

```
    {  
        calling super class constructor  
        super(a,b);  
    }
```

```
}
```

```
    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[])
    {
        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());
        figref=f;
        System.out.println("area is"+figref.area());
    }
}

```

OUTPUT:

Inside area for rectangle

area is 45

Inside area for triangle

area is 40

Inside area for figure is undefined

area is 0

1. Accessing the member of a super class:

The second form of super acts somewhat like this, except that it always refers to the super class of the subclass in which it is used. This usage has the following general form:

```
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();
}
}

```

This program displays the following:

i in superclass: 1

i in subclass: 2

Although the instance variable i in B hides the i in A, super allows access to the i defined in the superclass. As you will see, super can also be used to call methods that are hidden by a subclass.

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();
        System.out.println("hello");
    }
    static public void main(String args[])
    {
        B b=new B();
        b.display();
    }
}

```

method

calling super class



Output: hi

Hello

Base class

- 1) a class obtains variables and methods from another class
- 2) the former is called subclass, the latter super-class (Base class)
- 3) a sub-class provides a specialized behavior with respect to its

super-class

4) inheritance facilitates code reuse and avoids duplication of data

- One of the pillars of object-orientation.
- A new class is derived from an existing class:
 - 1) existing class is called super-class
 - 2) derived class is called sub-class
- A sub-class is a specialized version of its super-class:
 - 1) has all non-private members of its super-class
 - 2) may provide its own implementation of super-class methods
- Objects of a sub-class are a special kind of objects of a super-class.

extends

Is a keyword used to inherit a class from another class allows to extend from only one class

```
class One
{
    int a=5;
}
```

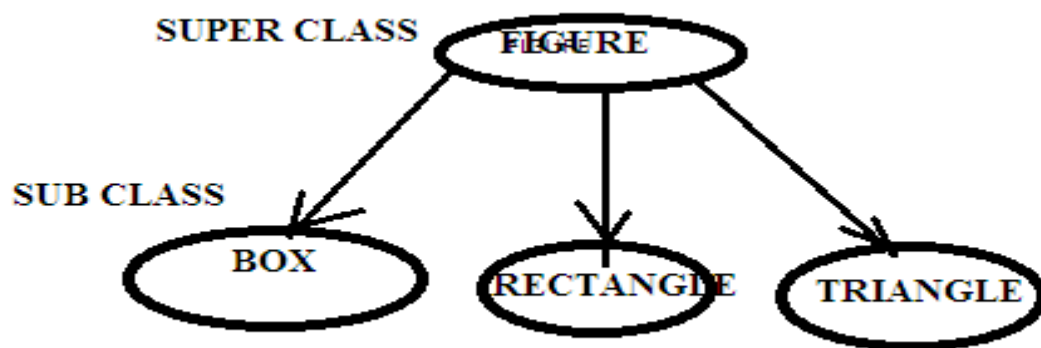
```
class Two extends One
{
    int b=10;
}
```

- **One baseobj**;// base class object.
- super class object **baseobj** can be used to refer its sub class objects.
- For example, **Two subobj=new Two**;
- **Baseobj=subobj** // now its pointing to sub class

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.

Therefore, a subclass is a specialized version of a super class. It inherits all of the instance variables and methods defined by the super class and add its own, unique elements.

Extends: To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword.

The “extends” keyword indicates that the properties of the super class name are extended to the subclass name. The sub class now contain its own variables and methods as well those of the super class. This kind of situation occurs when we want to add some more properties to an existing class without actually modifying the super class members.

To see how, let’s begin with a short example. The following program creates a super class called A and a subclass called B. Notice how the keyword extends is used to create a subclass of A.

```
// 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();  
}  
}
```

The output from this program is shown here:

Contents of superOb:

i and j: 10 20

Contents of subOb:

i and j: 7 8

k: 9

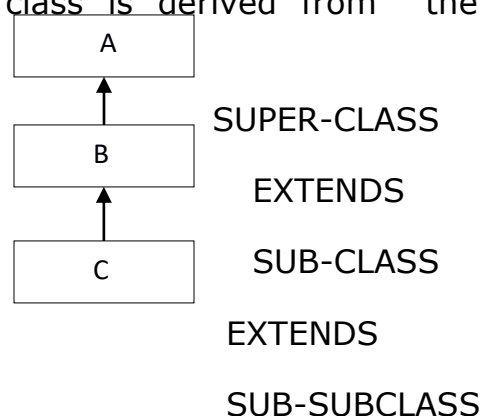
Sum of i, j and k in subOb:

i+j+k: 24

As you can see, the subclass B includes all of the members of its super class, A. This is why *subOb* can access i and j and call *showij()*. Also, inside *sum()*, i and j can be referred to directly, as if they were part of B. Even though A is a super class for B, it is also a completely independent, stand-alone class. Being a super class for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a super class for another subclass.

Multilevel Inheritance:

In multilevel inheritance the class is derived from the derived class.



Given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, **C** inherits all aspects of **B** and **A**. To see how a multilevel hierarchy can be useful, consider the following program. In it, the subclass **BoxWeight** is used as a superclass to create the subclass called **Shipment**. **Shipment** inherits all of the traits of **BoxWeight** and **Box**, and adds a field called **cost**, which holds the cost of shipping such a parcel.

```
// Extend BoxWeight to include shipping costs.
```

```
// Start with Box.
```

```
class Box {
private double width;
private double height;
private double depth;

// construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}

// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}

// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}

// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}

// compute and return volume
double volume() {
return width * height * depth;
}
}

// Add weight.
class BoxWeight extends Box {
```

```

double weight; // weight of box
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
}
// constructor when all parameters are specified
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
}
// default constructor
BoxWeight() {
super();
weight = -1;
}

// constructor used when cube is created
BoxWeight(double len, double m) {
super(len);
weight = m;
}
}

// Add shipping costs.
class Shipment extends BoxWeight {
double cost;
// construct clone of an object
Shipment(Shipment ob) { // pass object to constructor
super(ob);
cost = ob.cost;
}
// constructor when all parameters are specified
Shipment(double w, double h, double d,
double m, double c) {
super(w, h, d, m); // call superclass constructor
cost = c;
}
// default constructor
Shipment() {
super();
cost = -1;
}
// constructor used when cube is created
Shipment(double len, double m, double c) {

```

```

super(len, m);
cost = c;
}
}
class DemoShipment {
public static void main(String args[]) {
Shipment shipment1 =
new Shipment(10, 20, 15, 10, 3.41);
Shipment shipment2 =
new Shipment(2, 3, 4, 0.76, 1.28);
double vol;
vol = shipment1.volume();
System.out.println("Volume of shipment1 is " + vol);
System.out.println("Weight of shipment1 is "
+ shipment1.weight);
System.out.println("Shipping cost: $" + shipment1.cost);
System.out.println();

vol = shipment2.volume();
System.out.println("Volume of shipment2 is " + vol);
System.out.println("Weight of shipment2 is "
+ shipment2.weight);
System.out.println("Shipping cost: $" + shipment2.cost);
}
}

```

The output of this program is shown here:

```

Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41
Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28

```

When Constructors Are Executed

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy executed? For example, given a subclass called **B** and a superclass called **A**, is **A**'s constructor executed before **B**'s, or vice versa? The answer is that in a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass.

Example: As mentioned, it is perfectly acceptable to use a subclass as a superclass of another. For example, given three classes called A, B, and C, C can be a subclass of B, which is a subclass of A. When this type of situation

occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, C inherits all aspects of B and A. To see how a multilevel hierarchy can be useful, consider the following program.

```
// 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();  
    }  
}
```

```
}
```

The output from this program is shown here:

Inside A's constructor

Inside B's constructor

Inside C's constructor

As you can see, the constructors are called in order of derivation. If you think about it, it makes sense that constructors are executed in order of derivation. Because a super class has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must be executed first.

Method Overriding: Writing two or more methods in super & sub classes with same name and same signatures is called method overriding. In method overriding JVM executes a method depending on the type of the object.

Write a program that contains a super and sub class which contains a method

with same name and same method signature, behavior of the method is dynamically decided.

//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: Animals can move

Dogs can walk and run

Dynamic Method Dispatch

Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch*. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

A super class reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a super class reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the

reference variable) that determines which version of an overridden method will be executed.

Therefore, if a super class contains a method that is overridden by a subclass, then when different types of objects are referred to through a super class reference variable, different versions of the method are executed.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch
class A {
void callme() {
System.out.println("Inside A's callme method");
}
}
class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's callme method");
}
}
class C extends A {
// override callme()
void callme() {
System.out.println("Inside C's callme method");
}
}
class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C

A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}
```

The output from the program is shown here:

Inside A's callme method

Inside B's callme method

Inside C's callme method

Abstract classes:

A method with method body is called concrete method. In general any class will have all concrete methods. A method without method body is called abstract method. 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 essential element of object-oriented programming is abstraction. Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to drive to the grocery store without being overwhelmed by the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work. Instead they are free to utilize the object as a whole.

A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units.

For instance, the sound system consists of a radio, a CD player, and/or a tape player. The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions.

Hierarchical abstractions of complex systems can also be applied to computer programs. The data from a traditional process-oriented program can be transformed by abstraction into its component objects. A sequence of process steps can become a collection of messages between these objects. Thus, each of these objects describes its own unique behavior. You can treat

these objects as concrete entities that respond to messages telling them to do something. This is the essence of object-oriented programming.

Object-oriented concepts form the heart of Java just as they form the basis for human understanding. It is important that you understand how these concepts translate into programs. As you will see, object-oriented programming is a powerful and natural paradigm for creating programs that survive the inevitable changes accompanying the life cycle of any major software project, including conception, growth, and aging. For example, once you have well-defined objects and clean, reliable interfaces to those objects, you can gracefully decommission or replace parts of an older system without fear.

Abstract class: Any class that contains one or more abstract methods must also be declared abstract.

To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the super class, or be itself declared abstract.

There are situations in which you will want to define a super class that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a super class that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a super class is unable to create a meaningful implementation for a method. This is the case with the class Figure used in the preceding example. The definition of area() is simply a placeholder. It will not compute and display the area of any type of object. As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its super class. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually

appropriate. You may have methods which must be overridden by the subclass in order for the subclass to have any meaning. Consider the class Triangle. It has no meaning if area() is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the abstract method.

Abstract method: A method that is declared but not implemented (no body). Abstract methods are used to ensure that subclasses implement the method.

You can require that certain methods be overridden by subclasses by specifying the abstract type modifier. These methods are sometimes referred to as subclasses responsibility because they have no implementation specified in the super class. Thus, a subclass must override them—it cannot simply use the version defined in the super class. To declare an abstract method, use this general form:

```
abstract type name(parameter-list);
```

As you can see, no method body is present.

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

Write an example program for abstract class.

```
// Using abstract methods and classes.
```

```
abstract class Figure
```

```
{ double dim1;
```

```
double dim2;
```

```
Figure (double a, double b)
```

```
{ dim1 = a;
```

```
dim2 = b;
```

```
}
```

```
abstract double area (); // area is now 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 right 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:

Inside area for Rectangle.

Area is 45.0

Inside are for Triangle.

Area is 40.0

As the comment inside `main()` indicates, it is no longer possible to declare objects of type `Figure`, since it is now abstract. And, all subclasses of `Figure` must override `area()`. To prove this to yourself, try creating a subclass that does not override `area()`. You will receive a compile-time error.

Although it is not possible to create an object of type `Figure`, you can create a reference variable of type `Figure`. The variable `figref` is declared as a reference to `Figure`, which means that it can be used to refer to an object of any class derived from `Figure`. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

Using final with inheritance:

`Final` is a keyword in Java which generically means, cannot be changed once created. `Final` behaves very differently when variables, methods and classes. Any `final` keyword when declared with variables, methods and classes specifically means:

- 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 methods work like inline code of C++.

- Final with variables

Final variables work like const 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
{
    static
    {
        int x=10;
        final int y=20;
        System.out.println("x is:"+x);
        System.out.println("y is:"+y);
        x=30;
        y=40;
        System.out.println("x is:"+x);
        System.out.println("y is:"+y);
    }
}
```

Output:

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()
    {
        super.display();
        System.out.println("hello");
    }

    static public void main(String args[])
    {
```



```
B b=new B();
```

```
b.display();
```

```
}
```

```
}
```

Output:

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:

Cannot inherit from final Demo1

PACKAGES AND INTERFACES

DEFINING PACKAGE:

This chapter examines two of Java's most innovative features: packages and interfaces.

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.

A package is a container of classes and interfaces. A package represents a directory that contains related group of classes and interfaces. For example, when we write statements like:

```
import java.io.*;
```

Here we are importing classes of java.io package. Here, java is a directory name and io is another sub directory within it. The '*' represents all the classes and interfaces of that io sub directory. We can create our own packages called user-defined packages or extend the available packages. User-defined packages can also be imported into other classes and used exactly in the same way as the Built-in packages.

USE: Packages provide reusability.

ADVANTAGES OF PACKAGES:

- Packages are useful to arrange related classes and interfaces into a group. This makes all the classes and interfaces performing the same task to put together in the same package. For example, in Java, all

the classes and interfaces which perform input and output operations are stored in java.io. package.

- Packages hide the classes and interfaces in a separate sub directory, so that accidental deletion of classes and interfaces will not take place.
- The classes and interfaces of a package are isolated from the classes and interfaces of another package. This means that we can use same names for classes of two different classes. For example, there is a Date class in java.util package and also there is another Date class in java.sql package.
- A group of package called a library. The classes and interfaces of a package are like books in a library and can be reused several times. This reusability nature of packages makes programming easy. Just think, the package in Java are created by JavaSoft people only once, and millions of programmers all over the world are daily by using them in various programs.



Different Types of Packages:

There are two types of packages in Java. They are:

- Built-in packages
- User-defined packages

Built-in packages:

These are the packages which are already available in Java language. These packages provide all most all necessary classes, interfaces and methods for the programmer to perform any task in his programs. Since, Java has an extensive library of packages, a programmer need not think about logic for doing any task. For everything, there is a method available in Java and that method can be used by the programmer without developing the logic on his

own. This makes the programming easy. Here, we introduce some of the important packages of Java SE:

Java.lang: lang stands for language. This package got primary classes and interfaces essential for developing a basic Java program. It consists of wrapper classes(Integer, Character, Float etc), which are useful to convert primitive data types into objects. There are classes like String, StringBuffer, StringBuilder classes to handle strings. There is a Thread class to create various individual processes. Runtime and System classes are also present in java.lang package which contain methods to execute an application and find the total memory and free memory available in JVM.

Java.util: util stands for utility. This package contains useful classes and interfaces like Stack, LinkedList, Hashtable, Vector, Arrays, etc. These classes are collections. There are also classes for handling Date and Time operations.

Java.io: io stands for input and output. This package contains streams. A stream represents flow of data from one place to another place. Streams are useful to store data in the form of files and also to perform input-output related tasks.

Java.awt: awt stands for abstract window toolkit. This helps to develop GUI(Graphical user Interfaces) where programs with colorful screens, paintings and images etc., can be developed. It consists of an important sub package, java.awt.event, which is useful to provide action for components like push buttons, radio buttons, menus etc.

Java.swing: this package helps to develop GUI like java.awt. The 'x' in javax represents that it is an extended package which means it is a package developed from another package by adding new features to it. In fact, javax.swing is an extended package of java.awt.

Java.net: net stands for network. Client-Server programming can be done by using this package. Classes related to obtaining authentication for network, creating sockets at client and server to establish communication between them are also available in java.net package.

Java.applet: applets are programs which come from a server into a client and get executed on the client machine on a network. Applet class of this package is useful to create and use applets.

Java.text: this package has two important classes, DateFormat to format dates and times, and NumberFormat which is useful to format numeric values.

Java.sql: sql stands structured query language. This package helps to connect to databases like Oracle or Sybase, retrieve the data from them and use it in a Java program.

Package	Primary Function
java.text	Formats, searches, and manipulates text.
java.text.spi	Supports service providers for text formatting classes in java.text . (Added by Java SE 6.)
java.util	Contains common utilities.
java.util.concurrent	Supports the concurrent utilities.
java.util.concurrent.atomic	Supports atomic (that is, indivisible) operations on variables without the use of locks.
java.util.concurrent.locks	Supports synchronization locks.
java.util.jar	Creates and reads JAR files.
java.util.logging	Supports logging of information related to a program's execution.
java.util.prefs	Encapsulates information relating to user preference.
java.util.regex	Supports regular expression processing.
java.util.spi	Supports service providers for the utility classes in java.util . (Added by Java SE 6.)
java.util.zip	Reads and writes compressed and uncompressed ZIP files.

java.rmi	Provides remote method invocation.
java.rmi.activation	Activates persistent objects.
java.rmi.dgc	Manages distributed garbage collection.
java.rmi.registry	Maps names to remote object references.
java.rmi.server	Supports remote method invocation.
java.security	Handles certificates, keys, digests, signatures, and other security functions.
java.security.acl	Manages access control lists.
java.security.cert	Parses and manages certificates.
java.security.interfaces	Defines interfaces for DSA (Digital Signature Algorithm) keys.
java.security.spec	Specifies keys and algorithm parameters.
java.sql	Communicates with a SQL (Structured Query Language) database.

Package	Primary Function
java.applet	Supports construction of applets.
java.awt	Provides capabilities for graphical user interfaces.
java.awt.color	Supports color spaces and profiles.
java.awt.datatransfer	Transfers data to and from the system clipboard.
java.awt.dnd	Supports drag-and-drop operations.
java.awt.event	Handles events.

javax.swing	javax.swing.border	javax.swing.colorchooser
javax.swing.event	javax.swing.filechooser	javax.swing.plaf
javax.swing.plaf.basic	javax.swing.plaf.metal	javax.swing.plaf.multi
javax.swing.plaf.synth	javax.swing.table	javax.swing.text
javax.swing.text.html	javax.swing.text.html.parser	javax.swing.text.rtf
javax.swing.tree	javax.swing.undo	

User-Defined packages:

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

CREATING AND IMPORTING PACKAGES:

```
package packagename; //to create a package
```

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

e.g.: package pack;

- The first statement in the program must be package statement while creating a package.
- While creating a package except instance variables, declare all the members and the class itself as public then only the public members are available outside the package to other programs.

Program 1: Write a program to create a package pack with Addition class.

//creating a package

```
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 of two given numbers is : " + (d1+d2) );
```

```
    }
```

```
}
```

Compiling the above program:

A screenshot of a Windows command prompt window. The title bar shows 'C:\WINDOWS\system32\cmd.exe'. The command prompt shows the following text: 'D:\JQR>javac -d . Addition.java' followed by a new line 'D:\JQR>'. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac -d . Addition.java
D:\JQR>
```

The -d option tells the Java compiler to create a separate directory and place the .class file in that directory (package). The (.) dot after -d indicates that the package should be created in the current directory. So, our package pack with Addition class is ready.

Program 2: Write a program to use the Addition class of package pack.

//Using the package pack

```
import pack.Addition;
```

```
class Use
```

```
{ public static void main(String args[])
```

```

    { Addition ob1 = new Addition(10,20);
      ob1.sum();
    }
}

```

Output:



```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Use.java
D:\JQR>java Use
Sum of two given numbers is : 30.0
D:\JQR>

```

Program 3: Write a program to add one more class Subtraction to the same package pack.

//Adding one more class to package pack:

```

package pack;

public class Subtraction
{
    private double d1,d2;

    public Subtraction(double a, double b)
    {
        d1 = a;
        d2 = b;
    }

    public void difference()
    {
        System.out.println ("Sum of two given numbers is : " + (d1 - d2) );
    }
}

```

Compiling the above program:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac -d . Subtraction.java
D:\JQR>
```

Program 4: Write a program to access all the classes in the package pack.

//To import all the classes and interfaces in a class using import pack.*;

import pack.*;

class Use

{ public static void main(String args[])

{ Addition ob1 = new Addition(10.5,20.6);

ob1.sum();

Subtraction ob2 = new Subtraction(30.2,40.11);

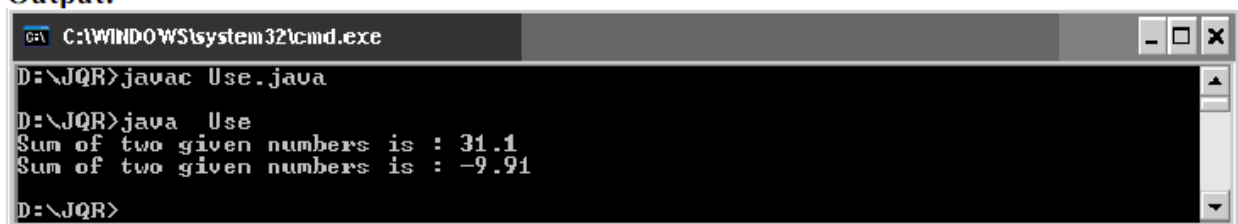
ob2.difference();

}

}

In this case, please be sure that any of the Addition.java and Subtraction.java programs will not exist in the current directory. Delete them from the current directory as they cause confusion for the Java compiler. The compiler looks for byte code in Addition.java and Subtraction.java files and there it gets no byte code and hence it flags some errors.

Output:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Use.java
D:\JQR>java Use
Sum of two given numbers is : 31.1
Sum of two given numbers is : -9.91
D:\JQR>
```

UNDERSTANDING CLASSPATH:

If the package pack is available in different directory, in that case the compiler should be given information regarding the package location by mentioning the directory name of the package in the classpath.

The CLASSPATH is an environment variable that tells the Java compiler where to look for class files to import.

If our package exists in e:\sub then we need to set class path as follows:



We are setting the classpath to e:\sub directory and current directory (.) an %CLASSPATH% means retain the already available classpath as it is.

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

```
package packagename.subpackagename;
```

e.g.: package pack1.pack2;

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

```
import pack1.pack2;
```

Program 5: Program to show how to create a subpackage in a package.

//Creating a subpackage in a package

```
package pack1.pack2;
```

```
public class Sample
```

```
{ public void show ()
```

```
{
```

```
System.out.println ("Hello Java Learners");
```

```
}
```

}

Compiling the above program:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac -d . Sample.java
D:\JQR>
```

ACCESSING A PACKAGES:

Access Specifier: Specifies the scope of the data members, class and methods.

- private members of the class are available with in the class only. The scope of private members of the class is "CLASS SCOPE".
- public members of the class are available anywhere . The scope of public members of the class is "GLOBAL SCOPE".
- default members of the class are available with in the class, outside the class and in its sub class of same package. It is not available outside the package. So the scope of default members of the class is "PACKAGE SCOPE".
- protected members of the class are available with in the class, outside the class and in its sub class of same package and also available to subclasses in different package also.

Class Member Access	private	No Modifier	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

Program 6: Write a program to create class A with different access specifiers.

```
//create a package same
```

```
package same;
```

```
public class A
```

```
{ private int a=1;
```

```
    public int b = 2;
```

```
    protected int c = 3;
```

```
    int d = 4;
```

```
}
```

Compiling the above program:

A screenshot of a Windows command prompt window. The title bar shows 'C:\WINDOWS\system32\cmd.exe'. The command prompt shows the directory 'D:\JQR' and the command 'javac -d . A.java' being executed. The prompt then shows 'D:\JQR>'.

Program 7: Write a program for creating class B in the same package.

```
//class B of same package
```

```
package same;
```

```
import same.A;
```

```
public class B
```

```
{
```

```
    public static void main(String args[])
```

```
    {
```

```
        A obj = new A();
```

```
        System.out.println(obj.a);
```

```
        System.out.println(obj.b);
```

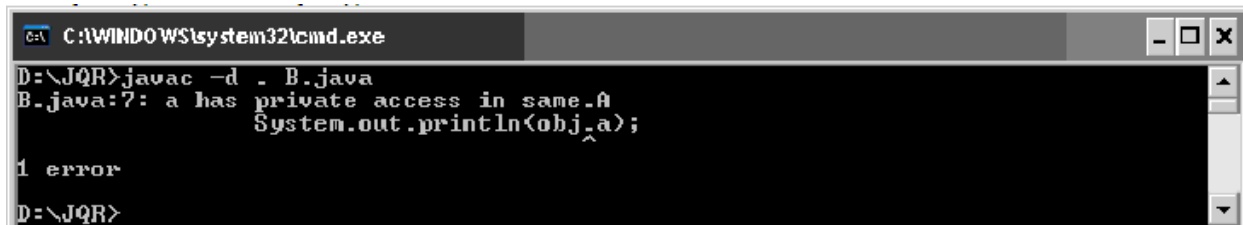
```

        System.out.println(obj.c);

        System.out.println(obj.d);
    }
}

```

Compiling the above program:



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The user is at the directory "D:\JQR". They have entered the command "javac -d . B.java". The output shows an error: "B.java:7: a has private access in same.A System.out.println(obj.a);". Below the error message, it says "1 error". The prompt "D:\JQR>" is visible at the bottom.

```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac -d . B.java
B.java:7: a has private access in same.A
        System.out.println(obj.a);
                             ^
1 error
D:\JQR>

```

Program 8: Write a program for creating class C of another package.

```

package another;

import same.A;

public class C extends A
{
    public static void main(String args[])
    {
        C obj = new C();

        System.out.println(obj.a);

        System.out.println(obj.b);

        System.out.println(obj.c);

        System.out.println(obj.d);
    }
}

```

Compiling the above program:

```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac -d . C.java
C:\java:8: a has private access in same.A
    System.out.println(obj.a);
                        ^
C:\java:11: d is not public in same.A; cannot be accessed from outside package
    System.out.println(obj.d);
                        ^
2 errors
D:\JQR>
```

DIFFERENCES BETWEEN CLASSES AND INTERFACES:

Classes	Interfaces
Classes have instances as variables and methods with body	Interfaces have instances as abstract methods and final constants variables.
Inheritance goes with extends keyword	Inheritance goes with implements keywords.
The variables can have any access specifier.	The Variables should be public, static, final
Multiple inheritance is not possible	It is possible
Classes are created by putting the keyword class prior to classname.	Interfaces are created by putting the keyword interface prior to interfacename(super class).
Classes contain any type of methods. Classes may or may not provide the abstractions.	Interfaces contain mostly abstract methods. Interfaces exhibit the fully abstractions

DEFINING AN INTERFACE AND IMPLEMENTING AN INTERFACE :

A programmer uses an abstract class when there are some common features shared by all the objects. A programmer writes an interface when all the features have different implementations for different objects. Interfaces are written when the programmer wants to leave the

implementation to third party vendors. An interface is a specification of method prototypes. All the methods in an interface are abstract methods.

- An interface is a specification of method prototypes.
- An interface contains zero or more abstract methods.
- All the methods of interface are public, abstract by default.
- An interface may contain variables which are by default public static final.
- Once an interface is written any third party vendor can implement it.
- All the methods of the interface should be implemented in its implementation classes.
- If any one of the method is not implemented, then that implementation class should be declared as abstract.
- We cannot create an object to an interface.
- We can create a reference variable to an interface.
- An interface cannot implement another interface.
- An interface can extend another interface.
- A class can implement multiple interfaces.

An interface is defined much like a class. This is the general form of an interface:

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

```
type final-varnameN = value;  
}
```

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. The general form of a class that includes the implements clause looks like this:

```
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.

Partial Implementations

If a class includes an interface but does not fully implement the methods defined by that

interface, then that class must be declared as abstract. For example:

```
abstract class Incomplete implements Callback {  
int a, b;  
void show() {  
System.out.println(a + " " + b);  
}  
// ...  
}
```

Here, the class Incomplete does not implement callback() and must be declared as abstract.

Any class that inherits Incomplete must implement callback() or be declared abstract itself.

Nested Interfaces

An interface can be declared a member of a class or another interface. Such an interface is

called a member interface or a nested interface. A nested interface can be declared as public,

private, or protected. This differs from a top-level interface, which must either be declared

as public or use the default access level, as previously described. When a nested interface is

used outside of its enclosing scope, it must be qualified by the name of the class or interface

of which it is a member. Thus, outside of the class or interface in which a nested interface is

declared, its name must be fully qualified.

Here is an example that demonstrates a nested interface:

```
// A nested interface example.
```

```
// This class contains a member interface.
```

```
class A {
```

```
// this is a nested interface
```

```
public interface NestedIF {
```

```
boolean isNotNegative(int x);
```

```
}
```

```
}
```

```
// B implements the nested interface.
```

```
class B implements A.NestedIF {
```

```

public boolean isNotNegative(int x) {
    return x < 0 ? false : true;
}

}

class NestedIFDemo {
    public static void main(String args[]) {
        // use a nested interface reference
        A.NestedIF nif = new B();
        if(nif.isNotNegative(10))
            System.out.println("10 is not negative");
        if(nif.isNotNegative(-12))
            System.out.println("this won't be displayed");
    }
}

```

Notice that A defines a member interface called NestedIF and that it is declared public. Next, B implements the nested interface by specifying implements

A.NestedIF

Notice that the name is fully qualified by the enclosing class' name. Inside the main() method, an A.NestedIF reference called nif is created, and it is assigned a reference to a B object. Because B implements A.NestedIF, this is legal.

Program 1: Write an example program for interface

interface Shape

```

{ void area ();
  void volume ();
}

```

```

    double pi = 3.14;
}
class Circle implements Shape
{
    double r;

    Circle (double radius)
    {
        r = radius;
    }

    public void area ()
    {
        System.out.println ("Area of a circle is : " + pi*r*r );
    }

    public void volume ()
    {
        System.out.println ("Volume of a circle is : " + 2*pi*r);
    }
}

class Rectangle implements Shape
{
    double l,b;

    Rectangle (double length, double breadth)
    {
        l = length;
        b = breadth;
    }

    public void area ()
    {
        System.out.println ("Area of a Rectangle is : " + l*b );
    }

    public void volume ()

```

```

    { System.out.println ("Volume of a Rectangle is : " + 2*(l+b));
    }
}

class InterfaceDemo
{
    public static void main (String args[])
    {
        Circle ob1 = new Circle (10.2);

        ob1.area ();

        ob1.volume ();

        Rectangle ob2 = new Rectangle (12.6, 23.55);

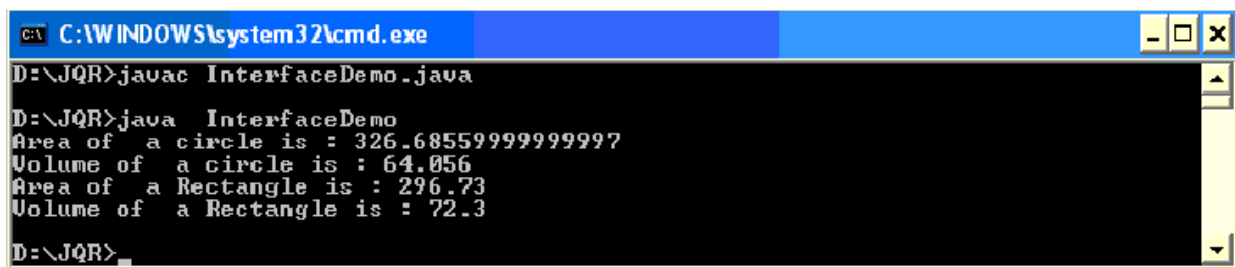
        ob2.area ();

        ob2.volume ();

    }
}

```

Output:



```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac InterfaceDemo.java
D:\JQR>java InterfaceDemo
Area of a circle is : 326.68559999999997
Volume of a circle is : 64.056
Area of a Rectangle is : 296.73
Volume of a Rectangle is : 72.3
D:\JQR>

```

APPLYING INTERFACE:

To understand the power of interfaces, let's look at a more practical example. In earlier chapters, you developed a class called Stack that implemented a simple fixed-size stack. However, there are many ways to implement a stack. For example, the stack can be of a fixed size or it can be "growable." The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same. That is, the methods `push()` and `pop()` define the interface to the stack independently of the details of the implementation.

Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics. Let's look at two examples.

First, here is the interface that defines an integer stack. Put this in a file called `IntStack.java`.

This interface will be used by both stack implementations.

```
// Define an integer stack interface.
```

```
interface IntStack {  
  
    void push(int item); // store an item  
  
    int pop(); // retrieve an item  
  
}
```

Applications are:

- Abstractions
- Multiple Inheritance

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 include that interface in a class (that is, when you "implement" the interface), all of those variable names will be in scope as **constants**. (This is similar to using a header file in C/C++ to create a large number of `#defined` 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 were importing the constant fields into the class name space as **final** variables. The next example uses this technique to implement an automated "decision maker":

```
import java.util.Random;
```

```
interface SharedConstants {
```

```
    int NO = 0;
```

```

int YES = 1;
int MAYBE = 2;
int LATER = 3;
int SOON = 4;
int NEVER = 5;
}

class Question implements SharedConstants {
    Random rand = new Random();

    int ask() {
        int prob = (int) (100 * rand.nextDouble());
        if (prob < 30)
            return NO;          // 30%
        else if (prob < 60)
            return YES;         // 30%
        else if (prob < 75)
            return LATER;       // 15%
        else if (prob < 98)
            return SOON;        // 13%
        else
            return NEVER;       // 2%
    }
}

class AskMe implements SharedConstants {
    static void answer(int result) {

```

```
switch(result) {  
    case NO:  
        System.out.println("No");  
        break;  
    case YES:  
        System.out.println("Yes");  
        break;  
    case MAYBE:  
        System.out.println("Maybe");  
        break;  
    case LATER:  
        System.out.println("Later");  
        break;  
    case SOON:  
        System.out.println("Soon");  
        break;  
    case NEVER:  
        System.out.println("Never");  
        break;  
}  
  
public static void main(String args[]) {  
    Question q = new Question();  
    answer(q.ask());  
}
```

```
answer(q.ask());  
answer(q.ask());  
answer(q.ask());  
}  
}
```

Notice that this program makes use of one of Java's standard classes: Random. This class provides pseudorandom numbers. It contains several methods that allow you to obtain random numbers in the form required by your program. In this example, the method `nextDouble()` is used. It returns random numbers in the range 0.0 to 1.0.

In this sample program, the two classes, Question and AskMe, both implement the SharedConstants interface where NO, YES, MAYBE, SOON, LATER, and NEVER are defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly. Here is the output of a sample run of this program. Note that the results are different each time it is run.

Later

Soon

No

Yes

EXTENDING INTERFACES:

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. Following is an example:

```
// One interface can extend one or more interfaces..
```



```
interface A {  
    void meth1();  
    void meth2();  
}  
  
// B now includes meth1() and meth2() -- it adds meth3().  
interface B extends A {  
    void meth3();  
}  
  
// This class must implement all of A and B  
class MyClass implements B {  
    public void meth1() {  
        System.out.println("Implement meth1().");  
    }  
    public void meth2() {  
        System.out.println("Implement meth2().");  
    }  
    public void meth3() {  
        System.out.println("Implement meth3().");  
    }  
}  
  
class IFExtend {  
    public static void main(String arg[]) {  
        MyClass ob = new MyClass();  
        ob.meth1();  
    }  
}
```

```
ob.meth2();  
ob.meth3();  
}  
}
```

As an experiment, you might want to try removing the implementation for `meth1()` in `MyClass`. This will cause a compile-time error. As stated earlier, any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

Although the examples we've included in this book do not make frequent use of packages or interfaces, both of these tools are an important part of the Java programming environment.

Virtually all real programs that you write in Java will be contained within packages. A number will probably implement interfaces as well. It is important, therefore, that you be comfortable with their usage.

Default Interface Methods

Prior to JDK 8, an interface could not define any implementation whatsoever. This meant that for all previous versions of Java, the methods specified by an interface were abstract, containing no body. This is the traditional form of an interface and is the type of interface that the preceding discussions have used. The release of JDK 8 has changed this by adding a new capability to **interface** called the *default method*. A default method lets you define a default implementation for an interface method. In other words, by use of a default method, it is now possible for an interface method to provide a body, rather than being abstract. During its development, the default method was also referred to as an *extension method*, and you will likely see both terms used.

A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code. If a new method were added to a popular, widely used interface, then the addition of that method would break existing code because no implementation would be found for that new method. The default method solves this problem by supplying an implementation that will be used if no other implementation is

explicitly provided. Thus, the addition of a default method will not cause preexisting code to break.

Another motivation for the default method was the desire to specify methods in an interface that are, essentially, optional, depending on how the interface is used. For example, an interface might define a group of methods that act on a sequence of elements.

One of these methods might be called **remove()**, and its purpose is to remove an element from the sequence. However, if the interface is intended to support both modifiable and nonmodifiable sequences, then **remove()** is essentially optional because it won't be used by nonmodifiable sequences. In the past, a class that implemented a nonmodifiable sequence would have had to define an empty implementation of **remove()**, even though it was not needed. Today, a default implementation for **remove()** can be specified in the interface that does nothing (or throws an exception). Providing this default prevents a class used for nonmodifiable sequences from having to define its own, placeholder version of **remove()**.

Thus, by providing a default, the interface makes the implementation of **remove()** by a class optional.

An interface default method is defined similar to the way a method is defined by a **class**. The primary difference is that the declaration is preceded by the keyword **default**. For example, consider this simple interface:

```
public interface MyIF {  
    // This is a "normal" interface method declaration.  
    // It does NOT define a default implementation.  
    int getNumber();  
    // This is a default method. Notice that it provides  
    // a default implementation.  
    default String getString() {  
        return "Default String";  
    }  
}
```

MyIF declares two methods. The first, **getNumber()**, is a standard interface method declaration. It defines no implementation whatsoever. The second method is **getString()**, and it does include a default implementation. In this case, it simply returns the string "Default String". Pay special attention to the way **getString()** is declared. Its declaration is preceded by the **default** modifier. This syntax can be generalized. To define a default

method, precede its declaration with **default**.

Because **getString()** includes a default implementation, it is not necessary for an implementing class to override it. In other words, if an implementing class does not provide its own implementation, the default is used. For example, the **MyIFImp** class shown next is perfectly valid:

```
// Implement MyIF.
class MyIFImp implements MyIF {
// Only getNumber() defined by MyIF needs to be implemented.
// getString() can be allowed to default.
public int getNumber() {
return 100;
}
}
```

The following code creates an instance of **MyIFImp** and uses it to call both **getNumber()** and **getString()**.

```
// Use the default method.
class DefaultMethodDemo {

public static void main(String args[]) {
MyIFImp obj = new MyIFImp();
// Can call getNumber(), because it is explicitly
// implemented by MyIFImp:
System.out.println(obj.getNumber());
// Can also call getString(), because of default
// implementation:
System.out.println(obj.getString());
}
}
```

The output is shown here:

100

Default String

As you can see, the default implementation of **getString()** was automatically used. It was not necessary for **MyIFImp** to define it. Thus, for **getString()**, implementation by a class is optional. (Of course, its implementation by a class will be *required* if the class uses **getString()**

for some purpose beyond that supported by its default.)

It is both possible and common for an implementing class to define its own implementation of a default method. For example, **MyIFImp2** overrides **getString()**:

```
class MyIFImp2 implements MyIF {
// Here, implementations for both getNumber( ) and getString( ) are
// provided.
public int getNumber() {
return 100;
}

public String getString() {
return "This is a different string.";
}
```

```
}  
}
```

Now, when **getString()** is called, a different string is returned.

Use static Methods in an Interface

JDK 8 added another new capability to **interface**: the ability to define one or more **static**

methods. Like **static** methods in a class, a **static** method defined by an interface can be

called independently of any object. Thus, no implementation of the interface is necessary,

and no instance of the interface is required, in order to call a **static** method. Instead, a

static method is called by specifying the interface name, followed by a period, followed by

the method name. Here is the general form:

InterfaceName.staticMethodName

Notice that this is similar to the way that a **static** method in a class is called.

The following shows an example of a **static** method in an interface by adding one

to **MyIF**, shown in the previous section. The **static** method is **getDefaultNumber()**. It returns zero.

```
public interface MyIF {  
    // This is a "normal" interface method declaration.  
    // It does NOT define a default implementation.  
    int getNumber();  
    // This is a default method. Notice that it provides  
    // a default implementation.  
    default String getString() {  
        return "Default String";  
    }  
}
```

```
// This is a static interface method.
```

```
static int getDefaultNumber() {  
    return 0;  
}  
}
```

The **getDefaultNumber()** method can be called, as shown here:

```
int defNum = MyIF.getDefaultNumber();
```

As mentioned, no implementation or instance of **MyIF** is required to call **getDefaultNumber()** because it is **static**.

One last point: **static** interface methods are not inherited by either an implementing

class or a subinterface.