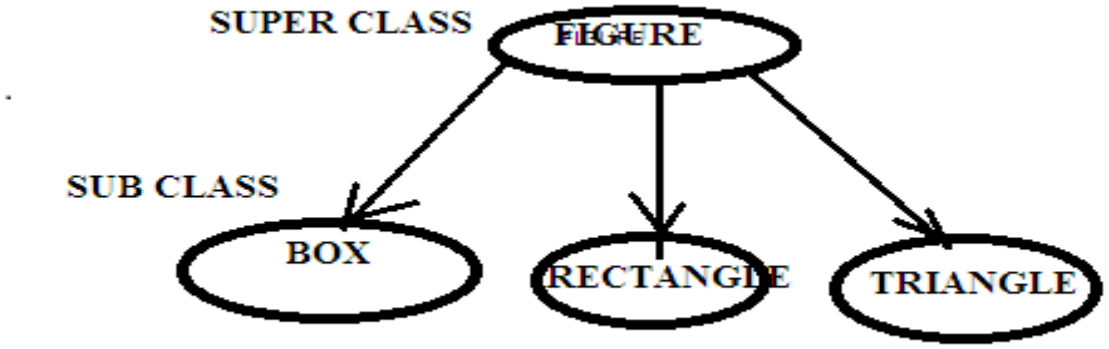
Unit-III

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 ojects 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.

Hierarchical abstractions:

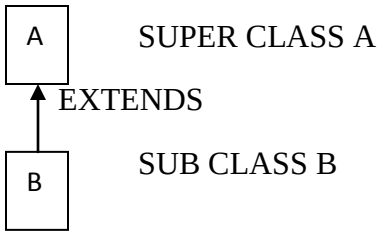
Hierarchical abstractions of complex systems can also be applied to computer programs. The data from a traditional process-oriented program can be transformed by abstraction into its component objects. A sequence of process steps can become a collection of messages between these objects. Thus, each of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

Up to this point, we have been using simple class hierarchies that consist of only a super class and a subclass. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a super class 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 super classes. In this case, C inherits all aspects of B and A.

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

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

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



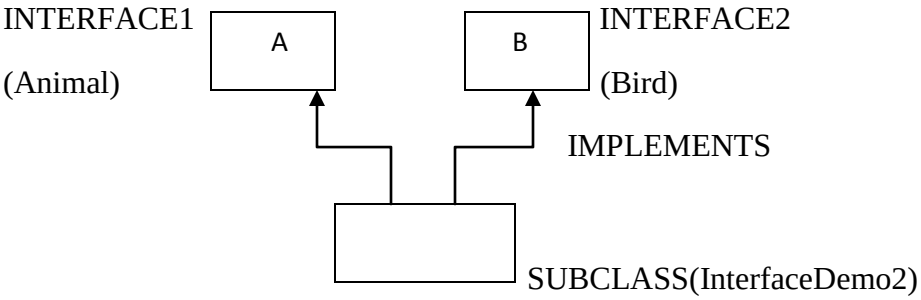
Example:

```
import java.io.*;
 abstract class A
{
   abstract void display();


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


        }
        public static void main(String args[])
        {
                B b=new B();
                b.display();
                super.display();


        }
}
```

Output:

Hello

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



We know that in multiple inheritance, sub class is derived from multiple super classes. If two super classes have sane names for their members then which member is inherited into the sub class is the main confusion in multiple inheritance. This is the reason, Java does not support the concept of multiple inheritance,. This confusion is reduced by using multiple interfaces to achieve multiple inheritance.

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

Interface syntax:

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;

}

All the methods in an interface must be abstract and public. If we not mention these keywords then JVM will treats all methods as public and abstract implicitly.

All the constants are treated as public, final and static.

Example:

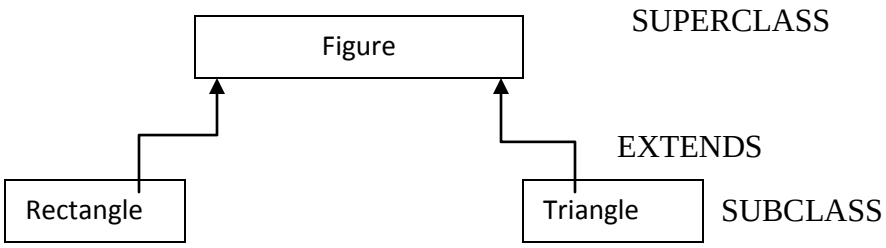
```
interface Animal
{
                                        public abstract void moves();
}
interface Bird
{
                                        void fly();
}
public class InterfaceDemo2 implements Animal,Bird
{
                                        public void moves()
                                        {
                                            System.out.println("animal move on land");
                                        }
                                        public void fly()
                                        {
                                            System.out.println("birds fly in air");
                                        }
                                        public static void main(String args[])
                                        {
                                            InterfaceDemo2 id=new InterfaceDemo2();
                                            id.moves();
                                            id.fly();
                                        }
}
```

Output:

animal move on land

birds fly in air

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



Example:

```
abstract class Figure
 {
        double dim1;
        double dim2;
        Figure(double a, double b)
        {
                dim1 = a;
                dim2 = b;
        }
// area is now an abstract method
        abstract double area();
}
class Rectangle extends Figure
 {
        Rectangle(double a, double b)
         {
                super(a, b);
```

```
        }


// override area for rectangle
        double area()
        {
                System.out.println("Inside Area for Rectangle.");
                return dim1 * dim2;
        }
}
class Triangle extends Figure
{
        Triangle(double a, double b)
         {
          super(a, b);
        }
// override area for right triangle
        double area()
         {
                System.out.println("Inside Area for Triangle.");
                return dim1 * dim2 / 2;
        }
}
class 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);
        Figure figref; // this is OK, no object is created
        figref = r;
        System.out.println("Area is " + figref.area());
        figref = t;
        System.out.println("Area is " + figref.area());
    }
}
```

output:

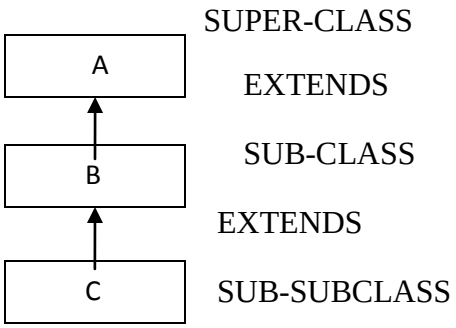
Inside area for Rectangle.

Area is 45.0

Inside are for Triangle.

Area is 40.0

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



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:

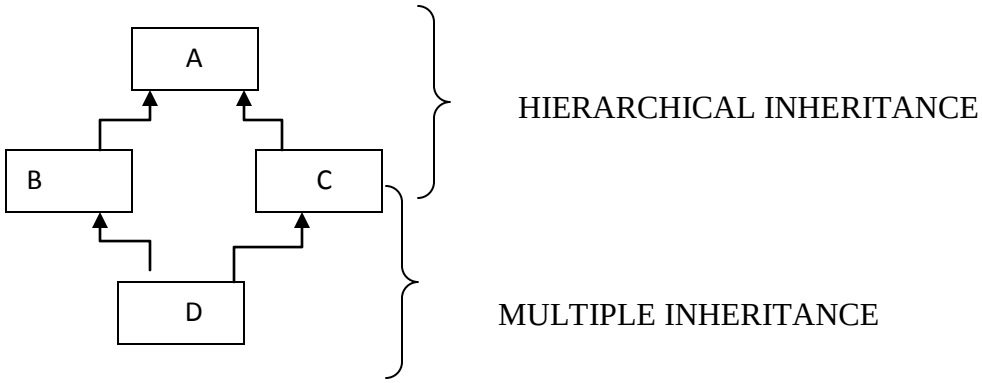
Inside A's constructor

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.

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



Example:

Base class Object or The Object class:

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

Importing Object class: Java Library ↓ Lang package ↓ Object class

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

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

Advantage: When we want to write a method that needs to handle objects if unknown type. If we define a parameter of object type, any class object can be passed to the method. Thus the method can receive any type of object and handle it.

Method	**Description**
boolean equals(Object obj)	This method compares the references of two objects and if they are equal, it returns true, otherwise false.
String toString()	This method returns a string representation of an object.
Class getClass()	This method gives an object that contains the name of a class to which an object belongs. Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object. This method returns hash code number of an object.
void notify( )	This method sends a notification to a thread which is witing for an object.
void notifyAll( )	This method sends a notification for all waiting threads for the object.
void wait( )	This method causes a thread to wait till a notification is received from a notify() or notifyAll() methods.
Object clone( )	This method creates a bitwise exact copy of an existing object. Creates a new object that is the same as the object being cloned.
void finalize( )	Called before an unused object is recycled. This method is called by the garbage collector when an object is removed from memory.

Note: The methods getClass( ), notify( ), notifyAll( ), and wait( ) are declared as final. You may override the others.

Example programs:

Example program for equal():

```
import java.io.*;

import java.util.*;

class Myclass

{

                    int x;

                    Myclass(int x)

                    {

                                        this.x=x;

                     }

}
```

```
class Compare
{
                    public static void main(String args[])
                    {
                    Myclass obj1=new Myclass(15);
                    Myclass obj2=new Myclass(15);
                    Integer obj3=new Integer(15);
                    Integer obj4=new Integer(15);
                    if(obj1.equals(obj2))
                    System.out.println("obj1 and obj2 are same");
                    else
                    System.out.println("obj1 and obj2 are not same");
                    if(obj3.equals(obj4))
                    System.out.println("obj1 and obj2 are same");
                    else
                    System.out.println("obj1 and obj2 are not same");
                    }
}
```

Output: obj1 and obj2 are not same

obj1 and obj2 are same

Example program for getClass():

```
import java.io.*;
import java.util.*;
class Myclass
{
                    int x;
                    Myclass(int x)
```

```
                    {
                                this.x=x;
                    }
}
class GetName
{
                    static void printName(Object obj)
                    {
                    Class c=obj.getClass();
                    String name=c.getName();
                    System.out.println("the class name:"+name);
                    }
}
class Getclass
{
                    static
                    {
                                Myclass obj=new Myclass(10);
                                GetName.printName(obj);
                                System.exit(0);
                    }

}
```

Output: the class name: Myclass

Example program for cloning:

The process of creating an exact copy of creating an exact copy of an existing object is called ‘cloning’.

In cloning, already an object should exist and when we clone the object, a bitwise copy of the object will result. The original and the cloned object will be exactly the same bit to bit. If the original object has some data in it, it also automatically comes into cloned object.

types of cloning:

1.when the cloned object is modified, same modification will alsoaffect the original object. This is called ‘shallow cloning’.

2. when the cloned object is modified, if the original object is not modified, then it is called ‘deep cloning’.

When we have new operator to create the objects, why do we need the cloning technology?

- Using new operator, we can create another object. But, when new operator is used to create the object, the object is created by using the initial values as object1. So,the same processing should be repeated on this object to get the intermediate object,i.e.,object2.
- The other way is to clone the object2, so that we get exact copy of the object. This preserves a copy of the intermediate object and hence the original object and the cloned objects can be processed separately. This method is easy because, we can avoid a lot of processing to be done on the object.

Program:

```
import java.io.*;

import java.util.*;

class Employee implements Cloneable

{

                int id;

                String name;

                Employee(int id,String name)

                {

                                this.id=id;

                                this.name=name;

                }

                void getData()

                {

                System.out.println("id:"+id);
```

```
                                        System.out.println("name:"+name);
                                        }
                                        public Object myClone() throws CloneNotSupportedException
                                        {
                                                                return super.clone();
                                        }
}
class CloneDemo
{
public static void main(String args[])throws CloneNotSupportedException
                                        {
                                        Employee e1=new Employee(10,"cnu");
                                        System.out.println("original object:");
                                        e1.getData();
                                        Employee e2=(Employee)e1.myClone();
                                        System.out.println("cloned object:");
                                        e2.getData();
                                        System.exit(0);
                                        }
}
Output: original object:id:10
          Name:cnu
         cloned object:id:10
          Name:cnu
```

Example program for hashCode:

```
import java.io.*;
class HashCode
{
  static
{
  String str1="hello";
 String str2="Hello";
 System.out.println("str1 is:"+str1.hashCode());
 System.out.println("str2 is:"+str2.hashCode());
 System.exit(0);
 }
}
```

Output: str1 is: 99162322

Str2 is:69609650

Example program for wait() and notify():

```
// A correct implementation of a producer and consumer.
class Q {
int n;
boolean valueSet = false;
synchronized int get() {
while(!valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
```

```
System.out.println("Got: " + n);
valueSet = false;
notify();
return n;
}
synchronized void put(int n) {
while(valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();
}
}
class Producer implements Runnable {
Q q;
Producer(Q q) {
this.q = q;
new Thread(this, "Producer").start();
}
public void run() {
int i = 0;
while(true) {
```

```
q.put(i++);
}
}
}
class Consumer implements Runnable {
Q q;
Consumer(Q q) {
this.q = q;
new Thread(this, "Consumer").start();
}
public void run() {
while(true) {
q.get();
}
}
}
class PCFixed {
public static void main(String args[]) {
Q q = new Q();
new Producer(q);
new Consumer(q);
System.out.println("Press Control-C to stop.");
}
}
```

output:

Put: 1

Got: 1

Put: 2

Got: 2

Put: 3

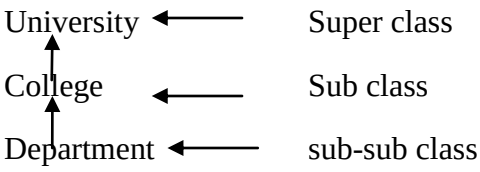
Subclass: when a new class is constructed using inheritance and contains an 'extends' keyword in the program's source description then that class is said to be subclass. Generally, a subclass is similar to a subtype.

Inheritance is a concept where subclass (new classes) can be produced from existing classes (super class).The newly created subclass acquires all the features of existing classes from where it is derived.

Class subclassname extends superclassname{}

Referenced Data Types: class is a references data type. It is used to store the several values. Converting a class into another class type is also possible through casting. But the classes should have some relationship between them by the way of inheritance.

Example: You cannot convert a Dog class into a Horse class, as those classes do not have any relationship between them. But you can convert a College class into a University class, since College is derived from University. And you can convert a Department class into a College,since Department is subclass of College class.



Type Casting: Converting one data type into another data type is called 'type casting' or simply 'casting'. Whenever we assign a value to a variable using assignment operator, the Java compiler checks for uniformity and hence the data types at both sides should be same. If the data types are not same, then we should convert the types to become same at the both sides.

To convert the data type, we use 'cast' operator. Cast operator means writing the data type between simple braces, before a variable or method whose value is to be converted.

Subtype: A subtype describes the relationship between different types that will lead to explicit recognition of substitution principle,

A type y will be a subtype(subclass) of x(superclass) only if it holds the following two conditions,

1. A variable of type x can be assigned an instance of y.
2. The resulting value can e used by the variable without changing the behavior,

Substitutability: Means the type of a variable can differ from the type of the value stored in that variable.

When interfaces are used, there will be a possibility of the occurrence of substitutability.

Types of Castings:

1. Widening or Implicit or Up-casting:

Converting lower data type into higher data type is widening. And it is performed by system, here is no need to put cast operator hence it is implicit casting. Lower datatype (subtype or subclass) is promoted to up(supertype or superclass), hence it is called as up-casting.

Note: In inheritance it is Generalization.

Example:

```
class One
{
 public void show1()
{
                                        System.out.println("hi");
}
}
class Two extends One
{
                                        public void show2()
{
                                        System.out.println("hello");
}
}
class WCast
{
 public static void main(String args[])
                                        {
```

```
                                        One  o=new Two();
                                        o.show1();

                                }
}
```

Output: hi

2. Narrowing or Explicit or Down-casting:

Converting Higher data type(super class) into lower (subclass) data type is called narrowing. And it is performed by the programmer hence it is explicit casting. Here must and should we have to put cast operator. Higher data type(super class) is dragged to down, hence it is down casting.

Note: In Inheritance it is Specialization.

Example:

```
class One
{
 public void show1()
{
                        System.out.println("hi");
}
}
class Two extends One
{
                        public void show2()
{
                        System.out.println("hello");
}
}
class NCast
```

```
{
 public static void main(String args[])
                                        {
                                                                One  o=(One) new Two();
                                                                Two t=(Two)o;
                                                                t.show1();
                                                                t.show2();
                                        }
}
```

Output: hi

Hello

Two types of casting in one program:

```
class Flower
{
 public void smell()
{
                                        System.out.println("flowery smell");
}
}
class Rose extends Flower
{
                                        public void smell()
{
                                        System.out.println("rose smell");

}
```

```
public static void main(String args[])
                                    {
                                                    Flower  f=new Flower();
                                                    Rose r=new Rose();
                                                    f=r;                    } Implicitly substituting
                                                    f.smell();              }
                                                    //r=f;  destination  source
                                                    r=(Rose)f;              } Explicitly substituting
                                                    f.smell();              }
                                    }                                       cast operator
}
```

Output: rose smell

Rose smell

Forms of Inheritance:

The various forms of inheritance are,

- Specialization
- Specification
- Construction
- Extension
- Limitation
- Combination

Specialization: Inheritance is commonly used for specialization purpose. Here, a child class or a new class is a specialized form of the parent class and it conforms to all specifications of the parent. Thus, a sub type(sub class) is created using this form and the substitutability is also maintained explicitly.

Example program:

Narrow casting or explicit casting program.

```
class One
{
 public void show1()
{
```

```
                    System.out.println("hi");
}
}
class Two extends One
{
                    public void show2()
{
                    System.out.println("hello");
}
}
class NCast
{
 public static void main(String args[])
                    {
                                        One  o=(One) new Two();
                                        Two t=(Two)o;
                                        t.show1();
                                        t.show2();
                    }
}
```

Output: hi

Hello

Specification: Inheritance can also be used to allow the classes to implement those methods that have the same names. The parent class can define operations with or without implementations. The operation whose implementation is not defined in the parent class will be defined in the child class. Such kind of parent class that defines abstract methods is also called as “abstract specification class”.

To support inheritance of specification, java language provides two different techniques. They are,

1. Using interfaces 2. Using classes

Example program:

```
interface Animal
{
                    public abstract void moves();
}
interface Bird
{
                    void fly();
}
public class InterfaceDemo2 implements Animal,Bird
{
                    public void moves()
                    {
                    System.out.println("animal move on land");
                    }
                    public void fly()
                    {
                    System.out.println("birds fly in air");
                    }
                    public static void main(String args[])
                    {
                    InterfaceDemo2 id=new InterfaceDemo2();
                                        id.moves();
                                        id.fly();
                    }
}
```

Output: animal move on land

Birds fly in air

2.using classes example program:

```
import java.io.*;
 abstract class A
{
   abstract void display();

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

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


                                    }
}
```

Output:hello

Construction: A child class can inherit most of the properties from its parent class, even if these classes do not have abstract concept in common.

Example program: single inheritance

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

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

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


                                }
}
```

Output:hello

Extension: The sub classification for extension is achieved if a child class adds an additional behavior to the parent class without modifying the attributes that are inherited from that parent class. The parent class functionality is made available to its child class without any modifications. Thus, such classes are also the subtypes because the subclassification for extension also supports the substitutability principle.

Example program is Multi level inheritance:

```
// 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

Limitation: If a subclass restricts few behaviors to be used that are inherited from the parent class, then the sub classification for limitation was occurred.

Example program for Limitation is using final methods with inheritance:

```
import java.io.*;
  class A
{
    final void display();



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


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

```
                    }
}
```

Output: display() in B cannot be overridde display() in a; overridden method is final.

Combination: Java language does not allow a subclass to inherit more than one class. Thus, solution for this problem is to extend a parent class and implement any number of interfaces.

Example program is multiple inheritance.

```
interface Animal
{
                    public abstract void moves();
}
interface Bird
{
                    void fly();
}
public class InterfaceDemo2 implements Animal,Bird
{
                    public void moves()
                    {
                        System.out.println("animal move on land");
                    }
                    public void fly()
                    {
                        System.out.println("birds fly in air");
                    }
                    public static void main(String args[])
```

```
    {
        InterfaceDemo2 id=new InterfaceDemo2();
        id.moves();
        id.fly();
    }
}
```

Output:

animal move on land

birds fly in air

Summary of Forms of Inheritance:

- Specialization. The child class is a special case of the parent class; in other words, the child class is a subtype of the parent class.
- Specification. The parent class defines behavior that is implemented in the child class but not in the parent class.
- Construction. The child class makes use of the behavior provided by the parent class, but is not a subtype of the parent class.
- Generalization. The child class modifies or overrides some of the methods of the parent class.(widening.up-casting)
- Extension. The child class adds new functionality to the parent class, but does not change any inherited behavior.
- Limitation. The child class restricts the use of some of the behavior inherited from the parent class.
- Variance. The child class and parent class are variants of each other, and the class-subclass relationship is arbitrary.
- Combination. The child class inherits features from more than one parent class. This is multiple inheritance and will be the subject of a later chapter.

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 are 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:

Note: Here Unit II-access control, unit IV-accessing packages are also covered in this topic.

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.

Note: A non-nested class has only two possible access levels: default and public.

When a class is declared as public, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

The following example shows all combinations of the access control modifiers. This example

has two packages and five classes. Remember that the classes for the two different

packages need to be stored in directories named after their respective packages—in this

case, p1 and p2.

The source for the first package defines three classes: Protection,Derived, and SamePackage.

The first class defines four int variables in each of the legal protection modes. The variable n

is declared with the default protection, n_pri is private, n_pro is protected, and n_pub is

public.

Each subsequent class in this example will try to access the variables in an instance

of this class. The lines that will not compile due to access restrictions are commented out.

Before each of these lines is a comment listing the places from which this level of protection

would allow access.

The second class, Derived, is a subclass of Protection in the same package, p1. This

grants Derived access to every variable in Protection except for n_pri, the private one. The

third class, SamePackage, is not a subclass of Protection, but is in the same package and

also has access to all but n_pri.

This is file Protection.java:

```
package p1;

public class Protection {

int n = 1;
```

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

This is file Derived.java:

```
package p1;
class Derived extends Protection {
Derived() {
System.out.println("derived constructor");
System.out.println("n = " + n);
//  class only
//  System.out.println("n_pri = "4 + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file SamePackage.java:

```
package p1;
class SamePackage {
SamePackage() {
```

```
Protection p = new Protection();
System.out.println("same package constructor");
System.out.println("n = " + p.n);
//  class only
//  System.out.println("n_pri = " + p.n_pri);
System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

Following is the source code for the other package, p2. The two classes defined i
cover the other two conditions that are affected by access control. The first class, Protecti
a subclass of p1.Protection. This grants access to all of p1.Protection's variables exce
n_pri (because it is private) and n, the variable declared with the default protection. Rem
the default only allows access from within the class or the package, not extra-pack
subclasses. Finally, the class OtherPackage has access to only one variable, n_pub, w
was declared public.

This is file Protection2.java:

```
package p2;
class Protection2 extends p1.Protection {
Protection2() {
System.out.println("derived other package constructor");
//  class or package only
//  System.out.println("n = " + n);
//  class only
//  System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
```

```
}
}
```

This is file OtherPackage.java:

```
package p2;
class OtherPackage {
OtherPackage() {
p1.Protection p = new p1.Protection();
System.out.println("other package constructor");
//  class or package only
//  System.out.println("n = " + p.n);
//  class only
//  System.out.println("n_pri = " + p.n_pri);
//  class, subclass or package only
//  System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

If you wish to try these two packages, here are two test files you can use. The one for package p1 is shown here:

```
// Demo package p1.
package p1;
// Instantiate the various classes in p1.
public class Demo {
public static void main(String args[]) {
Protection ob1 = new Protection();
Derived ob2 = new Derived();
SamePackage ob3 = new SamePackage();
```

```
}
}
```

The test file for p2 is shown next:

```
// Demo package p2.
package p2;
// Instantiate the various classes in p2.
public class Demo {
public static void main(String args[]) {
Protection2 ob1 = new Protection2();
OtherPackage ob2 = new OtherPackage();
}
}
```

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: Overriden 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

2.Accessing the member of a super class:

The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used. 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()                  calling super class method
                                {
                                                        super.display();
                                                        System.out.println("hello");



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




                                }
}
```

Output: hi

Hello

The keyword 'this': There will be situations where a method wants to refer to the object which invoked it. To perform this we use 'this' keyword. There are no restrictions to use 'this' keyword

we can use this inside any method for referring the current object. This keyword is always a reference to the object on which the method was invoked. We can use 'this' keyword wherever a reference to an object of the current class type is permitted. 'this' is a key word that refers to present class object. It refers to

Present class instance variables

Present class methods.

Present class constructor.

Program : Write a program to use 'this' to refer the current class parameterized constructor

and current class instance variable.

```
//this demo
class Person
{  String name;
Person ( )
{  this ("Theressa"); // calling present class parameterized constructor
  this.display ( ); // calling present class method
}
Person (String name)
{  this.name = name; // assigning present class variable with parameter "name"
}
void display( )
{  System.out.println ("Person Name is = " + name);
}
}
class ThisDemo
{  public static void main(String args[])
{
```

```
  Person p = new Person ( );
}
}
```

Output: Person Name is = Theressa

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

The statement calling super class constructor should be the first one in sub class constructor.

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

Polymorphism-Method overriding:

Polymorphism came from the two Greek words ‘poly’ means many and morphos means forms.

If the same method has ability to take more than one form to perform several tasks then it is called polymorphism.

It is of two types: Dynamic polymorphism and Static polymorphism.

Dynamic Polymorphism:

The polymorphism exhibited at run time is called dynamic polymorphism. In this dynamic polymorphism a method call is linked with method body at the time of execution by JVM. Java compiler does not know which method is called at the time of compilation. This is also known as dynamic binding or run time polymorphism.

Method overloading and method overriding are examples of Dynamic Polymorphism in Java.

o Method Overloading: Writing two or more methods with the same name, but with a difference in the method signatures is called method over loading. Method signature represents the method name along with the method parameters. In method over loading JVM understands which method is called depending upon the difference in the method signature. The difference may be due to the following:

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

void add (int a,int b)

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

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

void add (int a,float b)

void add (double a,double b)

Ø There is a difference in the sequence of parameters.

void swap (int a,char b)

void swap (char a,int b)

Write a program to create a class which contains two methods with the same

name but with different signatures.

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

```
{  Sample s=new Sample ( );
 s.add (20, 25);
 s.add (20, 25, 30);
}
 }
```

Output: sum of two=45

Sun of three=75

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

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

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

Achieving method overloading & method overriding using private, static and final methods is an example of Static Polymorphism.

Write a program to illustrate static polymorphism.

```
//Static Polymorphism
  class Animal
  {    static void move ()
{  System.out.println ("Animals can move");
}
   }
   class Dog extends Animal
   {       static void move ()
```

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

Output: Animals can move

Dogs can walk and run

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.