Fundamentals of Object Oriented Programming

Dr. Ayesha Hakim

Module 4

4	**Inheritance and Polymorphism**		11	CO 1, CO 4
	4.1	Inheritance: Types of Inheritance, inheritance using inversion of control		
	4.2	Final class, abstract class with constructor, abstract and non-abstract methods, Method Overriding.		
	4.2	Interface, final keyword, Implementing interfaces, extending interfaces, Difference between an Abstract class and an Interface		

Inheritance

- Inheritance can be defined as the process where **one class acquires the properties (methods and fields) of another.**
- With the use of inheritance the **information is made manageable in a hierarchical order.**
- The class which inherits the properties of other is known as **subclass (derived class, child class)** and the class whose properties are inherited is known as **superclass (base class, parent class).**
- It uses '**extends'** keyword.
- Child class may use the methods and variables of the Parent class.
- Inheritance enables **code reusability** and **robust coding.**

```java
// Java Program to illustrate Inheritance (concise)

import java.io.*;

// Base or Super Class
class Employee {
    int salary = 60000;
}

// Inherited or Sub Class
class Engineer extends Employee {
    int benefits = 10000;
}

// Driver Class
class Gfg {
    public static void main(String args[])
    {
        Engineer E1 = new Engineer();
        System.out.println("Salary : " + E1.salary
                           + "\nBenefits : " + E1.benefits);
    }
}
```

Output

```
Salary : 60000
Benefits : 10000
```

Inheritance

Inheritance Basics

what is inheritance ?

✓Object-oriented programming allows classes to inherit commonly used state and behavior from other classes

✓The mechanism of deriving a new class from existing old one is called inheritance.

✓The old class is known as super class or base class or parent class

✓The new class is known as child class or subclass or derived class.

✓To inherit properties of base class to sub class we use extends keyword in the inherited class i.e. in sub class.

Using super

Method Overriding

Using abstract class and final with inheritance

INHERITANCE

- **Inheritance** is the mechanism in java by which one class is allowed to **inherit the features of another class.**
- ***SYNTAX***: class subclass_name **extends** superclass_name {}
- It is used to provide **4 types** of inheritance. (multi-level, simple, hybrid and hierarchical inheritance)
- Methods can be defined inside the class in case of Inheritance.
- A class that inherits from another class can **reuse the methods** and fields of that class. In addition, you can add new fields and methods to your current class as well.

Inheritance

- To tailor a derived class, the programmer can add new variables or methods, or can modify the inherited ones
- ***Software reuse*** is at the heart of inheritance
- By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software

Inheritance

- Inheritance relationships often are shown graphically in a class diagram, with an arrow with an open arrowhead pointing to the parent class

Inheritance should create an *is-a relationship*, meaning the child *is a* more specific version of the parent

When To Employ Inheritance

- If you notice that **certain behaviors or data is common** among a group of related classes.
- **The commonalities may be defined by a superclass.**
- What is unique may be defined by particular subclasses.

Why Do We Need Java Inheritance?

- **Code Reusability:** The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
- **Method Overriding:** Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
- **Abstraction:** The concept of abstract where we do not have to provide all details is achieved through inheritance. Abstraction only shows the functionality to the user.

Advantages Of Inheritance in Java:

1. Code Reusability: Inheritance allows for code reuse and reduces the amount of code that needs to be written. The subclass can reuse the properties and methods of the superclass, reducing duplication of code.
2. Abstraction: Inheritance allows for the creation of abstract classes that define a common interface for a group of related classes. This promotes abstraction and encapsulation, making the code easier to maintain and extend.
3. Class Hierarchy: Inheritance allows for the creation of a class hierarchy, which can be used to model real-world objects and their relationships.
4. Polymorphism: Inheritance allows for polymorphism, which is the ability of an object to take on multiple forms. Subclasses can override the methods of the superclass, which allows them to change their behavior in different ways.

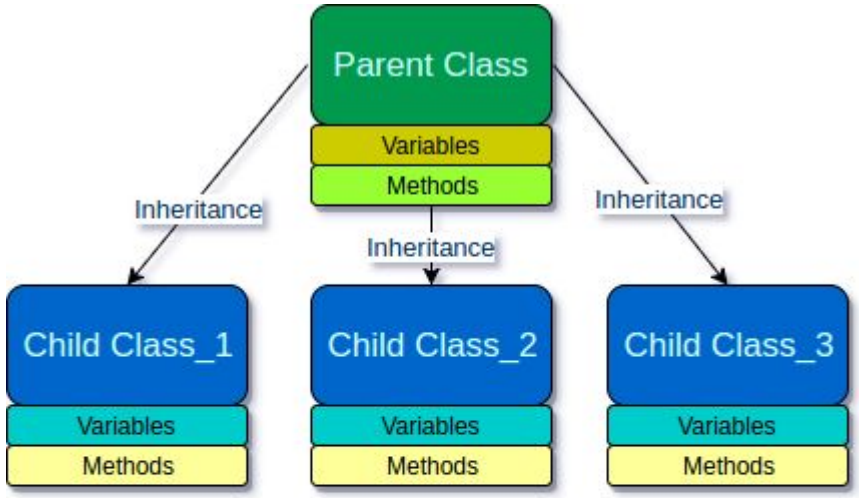
Disadvantages of Inheritance in Java:

1. Complexity: Inheritance can make the code more complex and harder to understand. This is especially true if the inheritance hierarchy is deep or if multiple inheritances is used.
2. Tight Coupling: Inheritance creates a tight coupling between the superclass and subclass, making it difficult to make changes to the superclass without affecting the subclass.

PARENT CHILD ANALOGY

In a parent-child analogy, child inherits parents variables(money, house, etc.,) and methods(behaviors from genetics). The same way, **an object of a child class**, which extends a Parent class, **can access the variables and methods of Parent class as of its own.**

- Inheritance is an Object Oriented Programming(OOP) concept.
- A Child class can inherit only one Parent class. (A child can have only one parent)
- Multiple (sub) classes can inherit a same (super) class. (A parent can have multiple children)
- Child class may use the methods and variables of the Parent class.
- **A child class can inherit all methods of parent class <u>except those which are private.</u>**



What happens when you call a method of Parent class from object of Child class?

When a Child class inherits a Parent class, compiler does not copy the methods of Parent class to Child class. But, the compiler makes a reference to the methods in the instance of Parent class.

Ex: we shall create a Super Class, `MobilePhone.java` and a Sub Class `SmartPhone.java`.

MobilePhone.java

```
package ……

public class MobilePhone {
    public int price;

    public void makeCall(){
        System.out.println("Calling...");
    }

    public void getCharge(){
        System.out.println("Charging...");
    }

    // private method is not visible to other classes
    private void check(){
        System.out.println("This is private to MobilePhone");
    }
}
```

SmartPhone.java

```
package ……

public class SmartPhone extends MobilePhone {

    public String name;

    public static void main(String[] args) {
        SmartPhone sonyXZ =new SmartPhone();
        // variable of SmartPhone (Sub Class)
        sonyXZ.name = "Sony XZ";

        // variable of MobilePhone (Super Class)
        sonyXZ.price = 40500;

        // method of SmartPhone (SubClass)
        sonyXZ.playVideo();

        // methods of MobilePhone (Super Class)
        sonyXZ.makeCall();
        sonyXZ.getCharge();;
    }

    public void playVideo(){
        System.out.println("Playing video..");
    }
}
```

Output at console

```
Playing video..
Calling...
Charging...
```

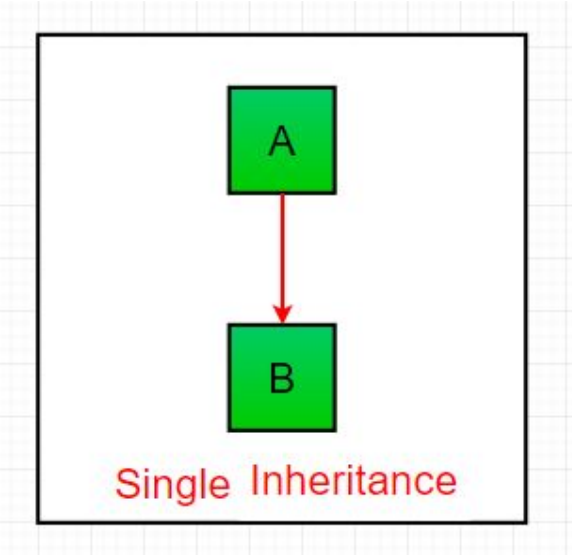
Deriving Subclasses

- In Java, we use the reserved word **`extends`** to establish an inheritance relationship

```
class Car extends Vehicle
{
    // class contents
}
```

Types of Inheritance

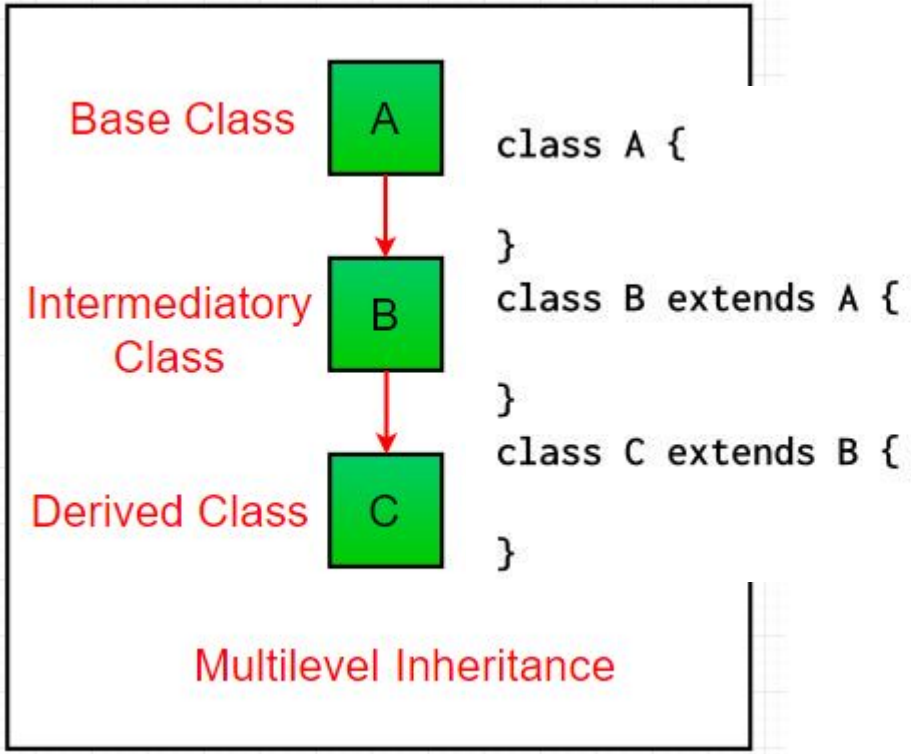
Single Inheritance: In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.



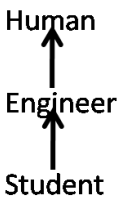
Single Inheritance example program in Java

```
Class A
{
        public void methodA()
        {
                System.out.println("Base class method");
        }
}
Class B extends A
{
        public void methodB()
        {
                System.out.println("Child class method");
        }
        public static void main(String args[])
        {
                B obj = new B();
                obj.methodA(); //calling super class method
                obj.methodB(); //calling local method
        }
}
```

Multilevel Inheritance: In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In the image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. **In Java, a class cannot directly access the grandparent's members.**



```
class Human
{
        void work()
        {
        System.out.println("Human can work");
        }
}
 class Engineer extends Human {
        void eat()
        {
        System.out.println("Engineer can eat");
        }
}
public class Student extends Engineer
{
         public static void main(String[] args)
         {
        Student S=new Student();
         S.work();
         S.eat();
        }
}
```

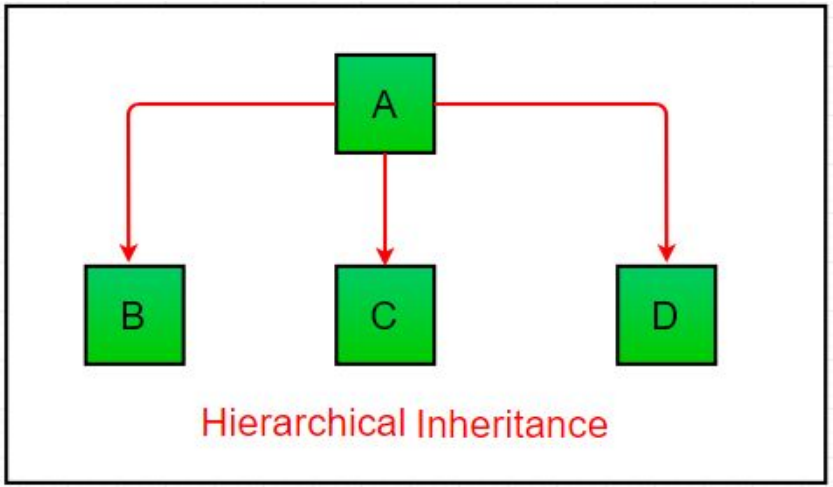


OUTPUT:

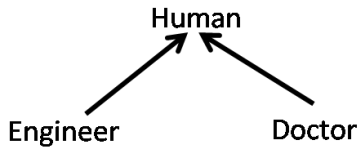
Human can work

Engineer can eat

Hierarchical Inheritance: In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the image, class A serves as a base class for the derived classes B, C and D.



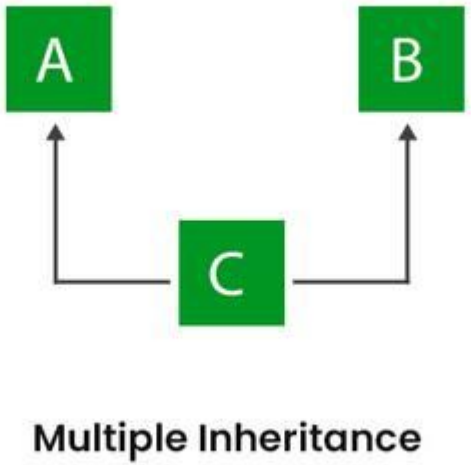
```
class Human {
        void work()
        {
        System.out.println("Human can work");
        }
        void eat()
        {
        System.out.println("Human can eat");
        }
}
 class Engineer extends Human
{
}
public class Doctor extends Human
{
         public static void main(String[] args)  {
        Doctor D=new Doctor();
        Engineer E=new Engineer();
         D.work();
         E.eat();
        }
}
```



OUTPUT:

Human can work

Human can eat



Multiple Inheritance

Java does not support multiple inheritance with classes (causes compile time error). This means that a class cannot extend more than one class.

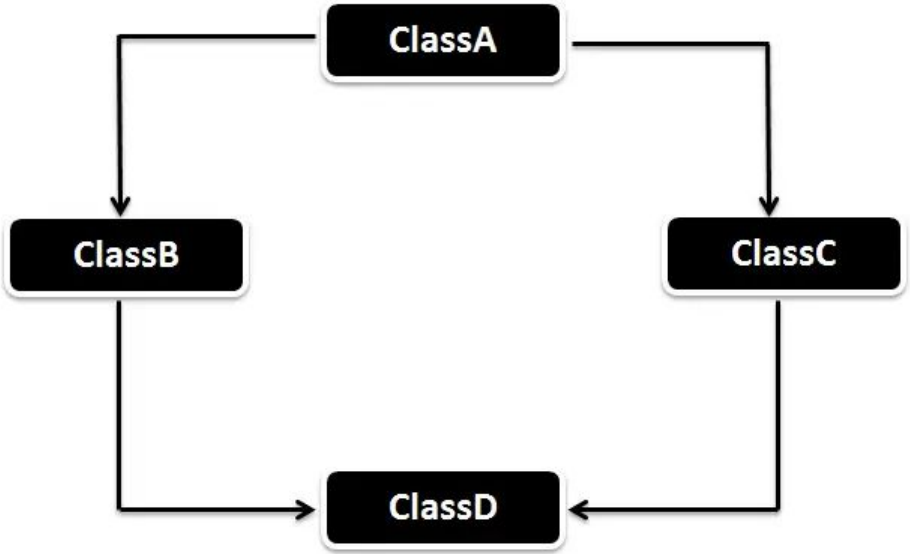
In Java, we can achieve multiple inheritances only through Interfaces.

Multiple Inheritance

- Java supports *single inheritance*, meaning that a derived class can have only one parent class
- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- Java does not support multiple inheritance. In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead

Hybrid Inheritance in Java

Hybrid Inheritance is a combination of both **Single Inheritance** and **Multiple Inheritance.** Since in **Java Multiple Inheritance** is not supported directly we can achieve Hybrid inheritance also through **Interfaces** only.



ClassA is the Parent for both **ClassB** and **ClassC** which is Single Inheritance and again **ClassB** and **ClassC** again act as Parent for **ClassC**(Multiple Inheritance which is not supported by Java)

Hybrid inheritance does not necessarily require the use of Multiple Inheritance exclusively. It can be achieved through a combination of Multilevel Inheritance and Hierarchical Inheritance with classes, Hierarchical and Single Inheritance with classes.

Therefore, it is indeed possible to implement Hybrid inheritance using classes alone, without relying on multiple inheritance type.

Implementation of Hybrid Inheritance with Classes & Interfaces

https://www.javatpoint.com/hybrid-inheritance-in-java

Interfaces in Java

An **Interface in Java** is defined as an abstract type used to specify the **behavior** of a class. An interface in Java is a **blueprint** of a behavior. A Java interface contains s**tatic constants and abstract methods (methods without a body).**

Uses of Interfaces in Java

Uses of Interfaces in Java are mentioned below:

- *It is used to achieve total abstraction.*
- *Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.*
- *Any class can extend only 1 class, but can any class implement an infinite number of interfaces.*
- *It is also used to achieve loose coupling.*
- *Interfaces are used to implement abstraction.*

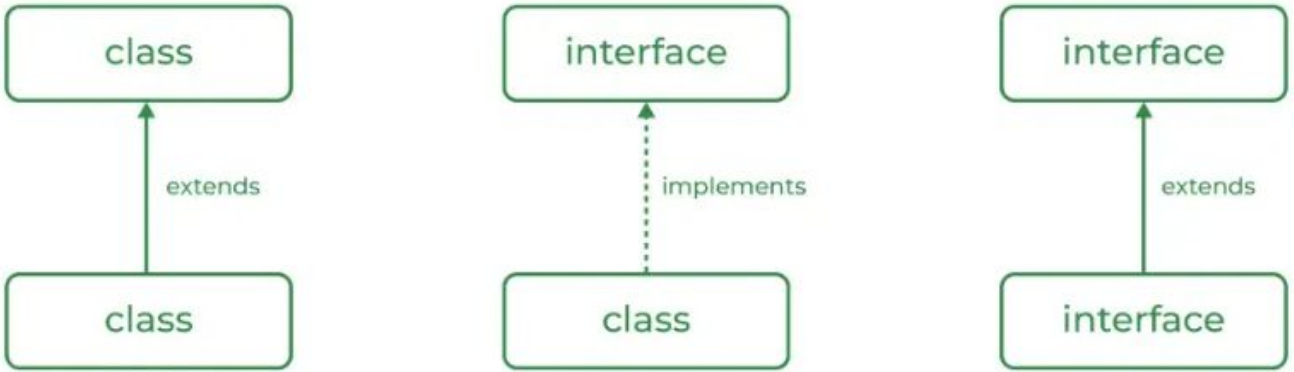
So, the question arises why use interfaces when we have abstract classes?

The reason is, abstract classes may contain non-final variables, whereas variables in the interface are final, public, and static.

```
// A simple interface
interface Player
{
    final int id = 10;
    int move();
}
```

Relationship Between Class and Interface

A class can extend another class similar to this an interface can extend another interface. But only a class can extend to another interface, and vice-versa is not allowed.



Difference Between Class and Interface

Although Class and Interface seem the same there have certain differences between Classes and Interface. The major differences between a class and an interface are mentioned below:

Class	Interface
In class, you can instantiate variables and create an object.	In an interface, you can't instantiate variables and create an object.
A class can contain concrete(with implementation) methods	The interface cannot contain concrete(with implementation) methods
The access specifiers used with classes are private, protected, and public.	In Interface only one specifier is used- Public.

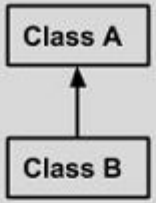
Implementation: *To implement an interface we use the keyword* ***implements***

Advantages of Interfaces in Java

The advantages of using interfaces in Java are as follows:

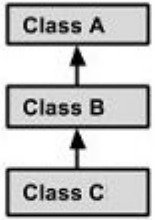
- Without bothering about the implementation part, we can achieve the security of the implementation.
- In Java, multiple inheritances are not allowed, however, you can use an interface to make use of it as you can implement more than one interface.

Single Inheritance



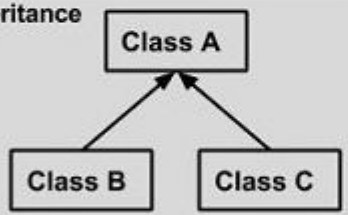

```
public class A {
   .......
}
public class B extends A {
   .........
}
```

Multi Level Inheritance



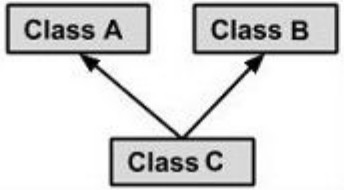

```
public class A { ..................}
public class B extends A {.....................}
public class C extends  B {..................... }
```

Hierarchical Inheritance




```
public class A { ..................}
public class B extends A {.....................}
public class C extends A {..................... }
```

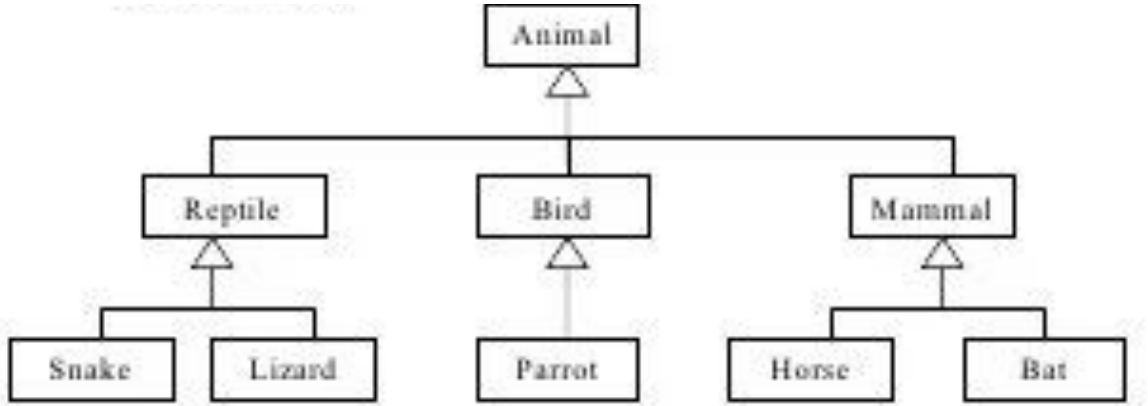
Multiple Inheritance




```
public class A { ..................}
public class B {.....................}
public class C extends  A,B {
   .....................
} // Java does not support mutiple Inheritance
```

Class Hierarchies

- A child class of one parent can be the parent of another child, forming a *class hierarchy*



Class Hierarchies

- Two children of the same parent are called *siblings*
- Common features should be put as high in the hierarchy as is reasonable
- An inherited member is passed continually down the line
- Therefore, a child class inherits from all its ancestor classes
- There is no single class hierarchy that is appropriate for all situations

The protected Modifier

- Visibility modifiers determine which class members are inherited and which are not
- **Variables and methods declared with `public` visibility are inherited; those with `private` visibility are not**
- But `public` variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

The protected Modifier

- The `protected` modifier allows a member of a base class to be inherited into a child.

- A class has members (variables or methods) marked as protected, they are accessible to subclasses, even if the subclass is in a different package. This behavior supports inheritance by ensuring that subclasses can use the protected members as if they were their own.

- Protected visibility provides more encapsulation than public visibility does.

- However, protected visibility is not as tightly encapsulated as private visibility.

The super Reference

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor (overriding)

The super Reference

- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the `super` reference to call the parent's constructor
- The `super` reference can also be used to reference other variables and methods defined in the parent's class

Using super

super has two general forms.

✓**The first calls the superclass constructor.**

✓**The second is** used to access a member of the superclass that has been hidden by a member of a subclass

Inheritance

Using super

Using abstract class and final with inheritance

A first Use for super

```
class a
{
a(){System.out.println("A");}
}
class b extends a
{
b()
{
super();
System.out.println("B");}
}
class c extends b
{
c(){System.out.println("c");}
}
class last
{
public static void main(String args[])
{
c obj = new c();
}
}
```

A Second Use for super

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.*

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:

A Second Use for super

```
// 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
```

```java
// Java code to show use of super
// keyword with variables

// Base class vehicle
class Vehicle {
    int maxSpeed = 120;
}

// sub class Car extending vehicle
class Car extends Vehicle {
    int maxSpeed = 180;

    void display()
    {
        // print maxSpeed of base class (vehicle)
        System.out.println("Maximum Speed: "
                           + super.maxSpeed);
    }
}

// Driver Program
class Test {
    public static void main(String[] args)
    {
        Car small = new Car();
        small.display();
    }
}
```

Output

```
Maximum Speed: 120
```

```java
// Java program to show use of super with methods

// superclass Person
class Person {
    void message()
    {
        System.out.println("This is person class\n");
    }
}
// Subclass Student
class Student extends Person {
    void message()
    {
        System.out.println("This is student class");
    }
    // Note that display() is
    // only in Student class
    void display()
    {
        // will invoke or call current
        // class message() method
        message();

        // will invoke or call parent
        // class message() method
        super.message();
    }
}
// Driver Program
class Test {
    public static void main(String args[])
    {
        Student s = new Student();

        // calling display() of Student
        s.display();
    }
}
```

Output

```
This is student class
This is person class
```

In the above example, we have seen that if we only call method message() then, the current class message() is invoked but with the use of the super keyword, message() of the superclass could also be invoked.

```java
// Java Code to show use of
// super keyword with constructor

// superclass Person
class Person {
    Person()
    {
        System.out.println("Person class Constructor");
    }
}

// subclass Student extending the Person class
class Student extends Person {
    Student()
    {
        // invoke or call parent class constructor
        super();

        System.out.println("Student class Constructor");
    }
}

// Driver Program
class Test {
    public static void main(String[] args)
    {
        Student s = new Student();
    }
}
```

Output

```
Person class Constructor
Student class Constructor
```

Advantages of Using Java ‘super’ keyword

The super keyword in Java provides several advantages in object-oriented programming:

- **Enables reuse of code**: Using the super keyword allows subclasses to inherit functionality from their parent classes, which promotes the reuse of code and reduces duplication.
- **Supports polymorphism**: Because subclasses can override methods and access fields from their parent classes using super, polymorphism is possible. This allows for more flexible and extensible code.
- **Provides access to parent class behavior**: Subclasses can access and use methods and fields defined in their parent classes through the super keyword, which allows them to take advantage of existing behavior without having to reimplement it.
- **Allows for customization of behavior:** By overriding methods and using super to call the parent implementation, subclasses can customize and extend the behavior of their parent classes.
- **Facilitates abstraction and encapsulation:** The use of super promotes encapsulation and abstraction by allowing subclasses to focus on their own behavior while relying on the parent class to handle lower-level details.

Overriding Methods

- A child class can *override* the definition of an inherited method in favor of its own
- The new method must have the **same signature** as the parent's method, but can have a **different body**
- The type of the object executing the method determines which version of the method is invoked

Overriding

- A parent method can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it **cannot** be overridden
- The concept of overriding can be applied to data and is called ***shadowing variables*** (Attributes of the subclass have the same name as attributes of the superclass)
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

Shadowing

- Local variables in a method or parameters to a method have the same name as instance fields.
- Attributes of the subclass have the same name as attributes of the superclass.

Method Overriding

-The method is **implemented differently** between the parent and child classes.

-Each method has the same return value, name and parameter list (identical signatures).

-The method that is actually called is **determined at program run time** (late binding).

-i.e., <reference name>.<method name> (parameter list);

The type of the reference (implicit parameter "this") distinguishes overridden methods

- Example of method overriding:

```
public class Foo
{
    public void display () { … }
        : :
}
public class FooChild extends Foo
{
     public void display () { … }
}

Foo f = new Foo ();
f.display();

FooChild fc = new FooChild ();
fc.display ();
```

```java
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output:

```
Bike is running safely
```

https://www.javatpoint.com/method-overriding-in-java

Accessing The Unique Attributes And Methods Of The Parent

- All protected or public attributes and methods of the parent class can be accessed directly in the child class

Accessing The Non-Unique Attributes And Methods Of The Parent

- An attribute or method exists in both the parent and child class (has the same name in both)
- The method or attribute has public or protected access
- Must prefix the attribute or method with the "super" keyword to distinguish it from the child class.
- **Format**:

  super.*methodName* ()
  super.*attributeName*

Note: If you don't preface the method attribute with the keyword "super" then by default the attribute or method of the child class will be accessed.

Accessing The Non-Unique Attributes And Methods Of The Parent: An Example

```
public class P
{
    protected int num;
    protected void method ()
    {
        :
    }
}
```

```
public class C extends P
{
    protected int num;
    public void method ()
    {
        num = 2;
        super.num = 3;
        super.method();
    }
}
```

Levels Of Access Permissions

- **Private "-"**
  - -Can only access the attribute/method in the methods of the class where the attribute is originally defined.
- **Protected "#"**
  - -Can access the attribute/method in the methods of the class where the attribute is originally defined or the subclasses of that class.
- **Public "+"**
  - -Can access attribute/method anywhere in the program.

Summary: Levels Of Access Permissions

Access level	Accessible to		
	Same class	Subclass	Not a subclass
Public	Yes	Yes	Yes
Protected	Yes	Yes	No
Private	Yes	No	No

Levels Of Access Permission: An Example

```
public class P
{
    private int num1;
    protected int num2;
    public int num3;
    // Can access num1, num2 & num3 here.
}

public class C extends P
{
    // Can't access num1 here
}

public class Driver
{
    // Can't access num1 here.
}
```

General Rules Of Thumb

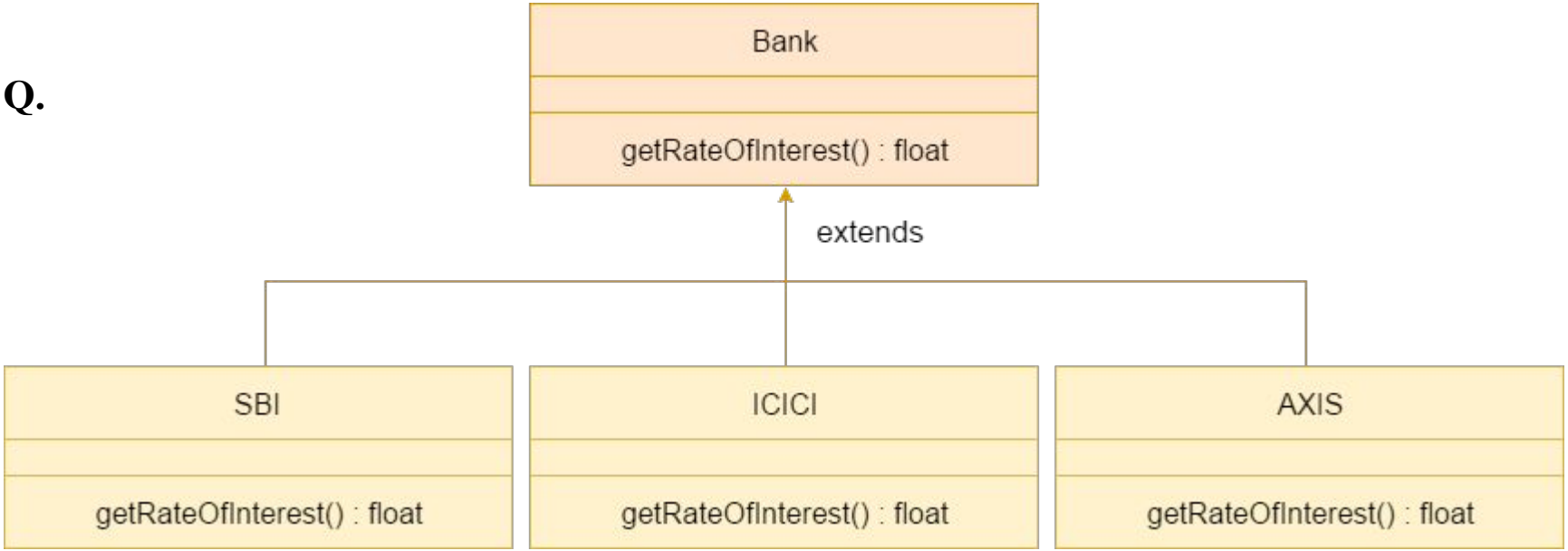
- Variable attributes should not have protected access but instead should be private.
- Most methods should be public.
- Methods that are used only by the parent and child classes should be made protected.

The Final Modifier (Inheritance)

- Methods preceded by the final modifier cannot be overridden
  e.g., public ***final*** void displayTwo ()
- Classes preceded by the final modifier cannot be extended
  -e.g., ***final*** public class ParentFoo

https://www.geeksforgeeks.org/using-final-with-inheritance-in-java/

Q.



Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.

Write a Java Program to demonstrate the real scenario of Java Method Overriding where three classes are overriding the method of a parent class using diagram shown above.

```
//Creating a parent class.
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
```

```
//Test class to create objects and call the methods
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}
}
```

Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9

Note: **a static method cannot be overridden. It is because the static method is bound with class** whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

Coding Practise

https://docs.google.com/document/d/1A8M2U99jZwziCeFDYdZH2OVqzLwLo0EMa8e5woJv7o4/edit?usp=sharing

#1. WAJP to understand concept of different access modifiers and keyword ‘super’

#2. WAJP to define abstract class shape with abstract method area() and volume() and calculate area and volume of cone and cylinder

https://www.codingninjas.com/studio/library/hybrid-inheritance-in-java

https://docs.google.com/document/d/1tM6La9VpnZNpBbQ9tcV11IMaogQTkrpxjKrWp1_O1EU/edit?usp=sharing

Write a Java program for creating one base class 'Personal' for student personal details and inherit those details into the subclass 'Education' of student Educational details to display complete student information.

CODE

Q. You are tasked with designing a simple class hierarchy for a zoo management system. Create the following classes:
Animal: A base class that has the following attributes: name (string), age (integer) It should have a method: describe() that returns a string describing the animal (e.g., "This is a [name], and it is [age] years old.")
Mammal: A subclass of Animal that adds an additional attribute: is_domestic (boolean). It should override the describe() method to include whether the mammal is domestic.
Bird: Another subclass of Animal that adds an attribute: can_fly (boolean). It should also override the describe() method to indicate whether the bird can fly.

Create instances of Mammal and Bird, and call their describe() methods to demonstrate that they work correctly by using super() to access the base class's method.

CODE

Class

- Class can instantiate variable & create object
- Class can contain concrete methods
- The access specifiers used with classes are private, protected and public

Interface

- Interface can't instantiate variable & create an object
- The interface can't contain concrete methods
- In interface only one public specifier is used

Java Interfaces

- Similar to a class
- Provides a design guide/ blueprint rather than implementation details. Without bothering about the implementation part, we can achieve the security of the implementation.
- Specifies what methods should be implemented but not how
  - -An important design tool and agreement for the interfaces should occur very early before program code has been written.
  - -(Specify the signature of methods so each part of the project can proceed with minimal coupling between classes).
- It's a design tool so they cannot be instantiated
  - Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.
  - Any class can extend only 1 class but can any class implement an infinite number of interface.

Interfaces: Syntax

Format for defining an interface

public interface <*name of interface*>

{

constants

*methods **to be** implemented by the class that realizes this interface*

}

Format for realizing / implementing the interface

public class <*name of class*> implements <*name of interface*>

{

attributes

*methods **actually** implemented by this class*

}

Interface

https://www.geeksforgeeks.org/interfaces-in-java/

https://docs.google.com/presentation/d/1WcOJ30qpIvjQt16y5UdpO4_2ac2Lqpws/edit#slide=id.p1

Overriding methods and its purpose

✓Overridden methods allow Java to support run-time polymorphism .

✓Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods

✓Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.

Final variables , methods and classes

- ✓All methods and variables can be Overridden by default in subclass.
- ✓If we wish to prevent the subclass from overriding the members of the superclass.
- ✓We can declare tem as final using the keyword final as a modifier.
- ✓Ex : final int SIZE=100;

  final void showstatus(){…….}
- ✓Making a method final ansures that the functionality defined in this method will never be altered in anyway.

Final variables , methods and classes

- ✓A class that cannot be subclassed is called a final class
- ✓Ex : final class Aclass{…….}
  final class Bclass extends Cclass{……..}
- ✓Any attempt to inherit these classes will cause an error and the compiler will not allow it.
- ✓Declaring a class final prevents any unwanted extension to the class.
- ✓It also allows the complier to perform some optimization when a method of a final class is invoked.

Accessing The Unique Attributes And Methods Of The Parent

- All protected or public attributes and methods of the parent class can be accessed directly in the child class

```
public class P
{
      protected int num;
 }

public class C extends P
{
      public void method ()
      {
       num = 2;
        }
}
```

Accessing The Non-Unique Attributes And Methods Of The Parent

- An attribute or method exists in both the parent and child class (has the same name in both)
- The method or attribute has public or protected access
- Must prefix the attribute or method with the "super" keyword to distinguish it from the child class.
- **Format**:

  super.*methodName* ()
  super.*attributeName*

  - Note: If you don't preface the method attribute with the keyword "super" then by default the attribute or method of the child class will be accessed.

CODING ACTIVITY

#2. WAJP to understand concept of different types of inheritance, different access modifiers and keyword ‘super’

Example: Sample program for inheritance

```
class SuperDemo
{
    int result;
    public void square(int a)
    {
        result = a * a;
        System.out.println("The square of the "+a+" is: "+result);
    }
}
public class SubDemo extends SuperDemo
{
    public void cube(int a)
    {
        result = a * a * a;
        System.out.println("The cube of the "+a+" is: "+result);
    }
     public static void main(String args[])
    {
        int a = 25;
        SubDemo sub = new SubDemo();
        sub.square(a);
        sub.cube(a);
    }
}
```

Output:
The square of the 25 is: 625
The cube of the 25 is: 15625

Example: Sample program for single level inheritance

```
class Parent
{
   public void m1()
   {
      System.out.println("Class Parent method");
   }
}
public class Child extends Parent
{
  public void m2()
  {
    System.out.println("Class Child method");
  }
  public static void main(String args[])
  {
    Child obj = new Child();
    obj.m1();
    obj.m2();
  }
}
```

Output:
Class Parent method
Class Child method

Example: Sample program for multilevel inheritance

```
class Grand
{
  public void m1()
  {
   System.out.println("Class Grand method");
  }
}
class Parent extends Grand
{
  public void m2()
  {
     System.out.println("Class Parent method");
   }
}
class Child extends Parent
{
  public void m3()
  {
   System.out.println("Class Child method");
  }
  public static void main(String args[])
  {
   Child obj = new Child();
   obj.m1();
   obj.m2();
   obj.m3();
  }
}
```

Output:

Class Grand method
Class Parent method
Class Child method

Example: Sample program for hierarchical inheritance

```
class A
{
  public void m1()
  {
    System.out.println("method of Class A");
  }
}
class B extends A
{
  public void m2()
 {
    System.out.println("method of Class B");
  }
}
class C extends A
{
  public void m3()
  {
    System.out.println("method of Class C");
  }
}
class D extends A
{
  public void m4()
  {
    System.out.println("method of Class D");
  }
}
public class MainClass
{
  public void m5()
  {
    System.out.println("method of Class MainClass");
  }
  public static void main(String args[])
  {
    A obj1 = new A();
    B obj2 = new B();
    C obj3 = new C();
    D obj4 = new D();
    obj1.m1();
    obj2.m1();
    obj3.m1();
    obj4.m1();
  }
}
```

Output:

method of Class A
method of Class A
method of Class A
method of Class A

Example: Sample program for multiple inheritance

```
class X
{
  void display()
  {
     System.out.println("class X dispaly method ");
  }
}
class Y
{
  void display()
  {
     System.out.println("class Y display method ");
  }
}
public class Z extends X,Y
{
   public static void main(String args[])
   {
      Z obj=new Z();
      obj.display();
  }
}
```

Output:
Compile time error

Access Modifiers

Access modifiers are simply a keyword in Java that provides accessibility of a class and its member. They set the access level to methods, variable, classes and constructors.

There are 4 types of access modifiers available in Java.

- public
- default
- protected
- private

public

The member with public modifiers can be accessed by any classes. The public methods, variables or class have the **widest scope.**

Example: Sample program for public access modifier

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

{

   // code

}
```

default

When we do not mention any access modifier, it is treated as default. It is accessible only within same package.

Example: Sample program for default access modifier

```
int a = 25;

String str = "Java";

boolean m1()

{

    return true;

}
```

protected

The protected modifier is used within same package. It **lies between public and default access modifier**. It can be accessed outside the package **but through inheritance only.**
A class cannot be protected.

Example: Sample program for protected access modifier

```
class Employee
{
   protected int id = 101;
   protected String name = "Jack";
}
public class ProtectedDemo extends Employee
{
   private String dept = "Networking";
   public void display()
   {
      System.out.println("Employee Id : "+id);
      System.out.println("Employee name : "+name);
      System.out.println("Employee Department : "+dept);
   }
   public static void main(String args[])
   {
      ProtectedDemo pd = new ProtectedDemo();
      pd.display();
   }
}
```

Output:

```
Employee Id : 101
Employee name : Jack
Employee Department : Networking
```

private

The private methods, variables and constructor are not accessible to any other class. It is the **most restrictive access modifier.**

A class except a nested class cannot be private.

Example: Sample program for private access modifier

```
public class PrivateDemo
{
   private int a = 101;
   private String s = "TutorialRide";
   public void show()
   {
      System.out.println("Private int a = "+a+"\nString s = "+s);
   }
   public static void main(String args[])
   {
      PrivateDemo pd = new PrivateDemo();
      pd.show();
      System.out.println(pd.a+" "+pd.s);
   }
}
```

Output:

```
Private int a = 101
String s = TutorialRide
101 TutorialRide
```

Table for Access Modifier

Access modifier	In class	In package	Outside package by subclass	Outside package
public	Yes	Yes	Yes	No
protected	Yes	Yes	Yes	No
default	Yes	Yes	No	No
private	Yes	No	No	No

The super keyword in Java

- super keyword is similar to this keyword in Java.
- It is used to refer to the immediate parent class of the object.

Use of super keyword in Java

- super () calls the parent class constructor with no argument.
- super.methodname calls method from parents class.
- It is used to call the parents class variable.

Example: Sample program for invoking parents class constructor

```
class Animal
{
   public Animal(String str)
   {
      System.out.println("Constructor of Animal: " + str);
   }
}
class Lion extends Animal
{
   public Lion()
   {
      super("super call Lion constructor");
      System.out.println("Constructer of Lion.");
   }
}
public class Test
{
   public static void main(String[] a)
  {
      Lion lion = new Lion();
   }
}
```

Output:

Constructor of Animal: super call from Lion constructor

Constructor of Lion.

Example: Sample program for calling parents class variable

```
class Base
{
   int a = 50;
}
public class Derive extends Base
{
   int a = 100;
   void display()
   {
      System.out.println(super.a);
      System.out.println(a);
   }
   public static void main(String[] args)
   {
      Derive derive = new Derive();
      derive.display();
   }
}
```

Output:

50

100

The final keyword in Java

The **final** keyword in Java indicates that no further modification is possible. Final can be Variable, Method or Class

Final Variable

Final variable is a constant. We cannot change the value of final variable after initialization.

Example: Sample program for final keyword in Java

```
class FinalVarDemo
{
    final int a = 10;
    void show()
    {
        a = 20;
        System.out.println("a : "+a);
    }
    public static void main(String args[])
    {
        FinalVarDemo var = new FinalVarDemo();
        var.show();
    }
}
```

Output:
Compile time error

The Final Modifier (Inheritance)

- Methods preceded by the final modifier cannot be overridden

  e.g., public ***final*** void displayTwo ()

- Classes preceded by the final modifier cannot be extended

  -e.g., ***final*** public class ParentFoo

Final method

When we declare any method as final, it cannot be override.

Example: Sample program for final method in Java

```
class Animal
{
   final void eat()
  {
      System.out.println("Animals are eating");
   }
}
public class Dear extends Animal
{
   void eat()
   {
      System.out.println("Dear is eating");
   }
   public static void main(String args[])
   {
      Dear dear = new Dear();
      dear.eat();
   }
}
```

Output:
Compile time error

Final Class

When a class is declared as a final, it cannot be extended.

Example: Sample program for final class in Java

```
final class Animal
{
    void eat()
    {
        System.out.println("Animals are eating");
    }
}
class Dear extends Animal
{
    void eat()
    {
        System.out.println("Dear is eating");
    }
    public static void main(String args[])
   {
        Dear dear = new Dear();
        dear.eat();
    }
}
```

Output:
Compile time error

Q1.

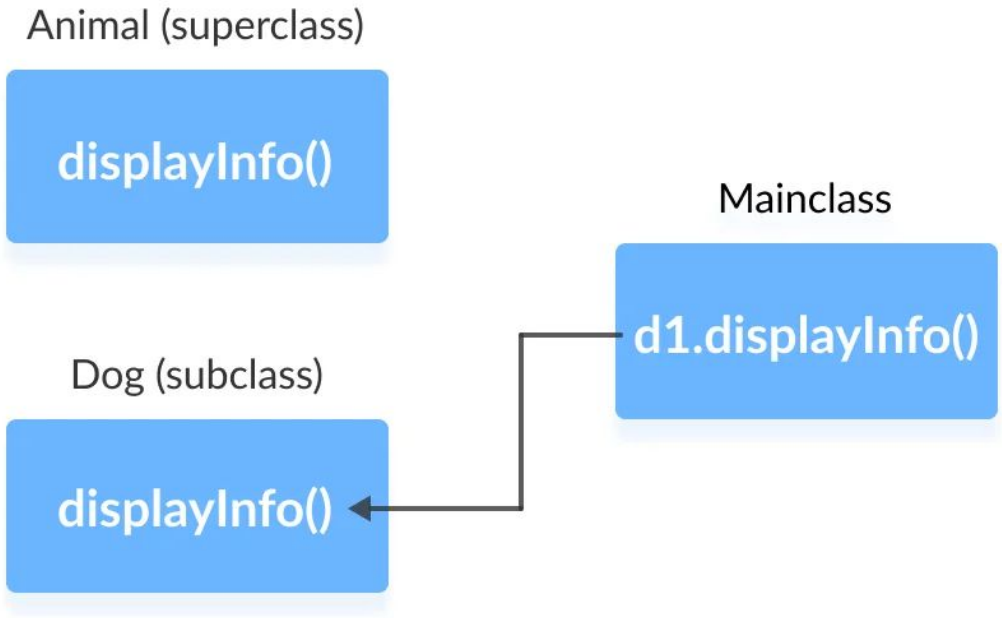
```
class Animal {
   public void displayInfo() {
      System.out.println("I am an animal.");
   }
}

class Dog extends Animal {
   @Override
   public void displayInfo() {
      System.out.println("I am a dog.");
   }
}

class Main {
   public static void main(String[] args) {
      Dog d1 = new Dog();
      d1.displayInfo();
   }
}
```

Output:
I am a dog.

WHY?

In the above program, the displayInfo() method is present in both the Animal superclass and the Dog subclass.

When we call displayInfo() using the d1 object (object of the subclass), the method inside the subclass Dog is called. The displayInfo() method of the subclass overrides the same method of the superclass.



Here, the @Override annotation specifies the compiler that the method after this annotation overrides the method of the superclass. It is not mandatory to use @Override.

Java Overriding Rules

- Both the superclass and the subclass must have the same method name, the same return type and the same parameter list.
- We cannot override the method declared as final and static.

Q2.

```
class Animal {
   public void displayInfo() {
      System.out.println("I am an
animal.");
   }
}

class Dog extends Animal {
   public void displayInfo() {
      super.displayInfo();
      System.out.println("I am a
dog.");
   }
}

class Main {
   public static void main(String[]
args) {
      Dog d1 = new Dog();
      d1.displayInfo();
   }
}
```

Output:

```
I am an animal.
I am a dog.
```

In the above example, the subclass Dog overrides the method displayInfo() of the superclass Animal.

When we call the method displayInfo() using the d1 object of the Dog subclass, the method inside the Dog subclass is called; the method inside the superclass is not called.
Inside displayInfo() of the Dog subclass, we have used super.displayInfo() to call displayInfo() of the superclass.

Access Specifiers in Method Overriding

The same method declared in the superclass and its subclasses can have different access specifiers. However, there is a restriction.

We can only use those access specifiers in subclasses that provide larger access than the access specifier of the superclass. For example,

Suppose, a method `myClass()` in the superclass is declared `protected`. Then, the same method `myClass()` in the subclass can be either `public` or `protected`, but not `private`.

```
class Animal {
  protected void displayInfo() {
    System.out.println("I am an animal.");
  }
}

class Dog extends Animal {
  public void displayInfo() {
    System.out.println("I am a dog.");
  }
}

class Main {
  public static void main(String[] args) {
    Dog d1 = new Dog();
    d1.displayInfo();
  }
}
```

Output:
I am a dog.

- The subclass Dog overrides the method displayInfo() of the superclass Animal.
- Whenever we call displayInfo() using the d1 (object of the subclass), the method inside the subclass is called.
- The displayInfo() is declared protected in the Animal superclass. The same method has the public access specifier in the Dog subclass. This is possible because the public provides larger access than the protected.

Inheritance

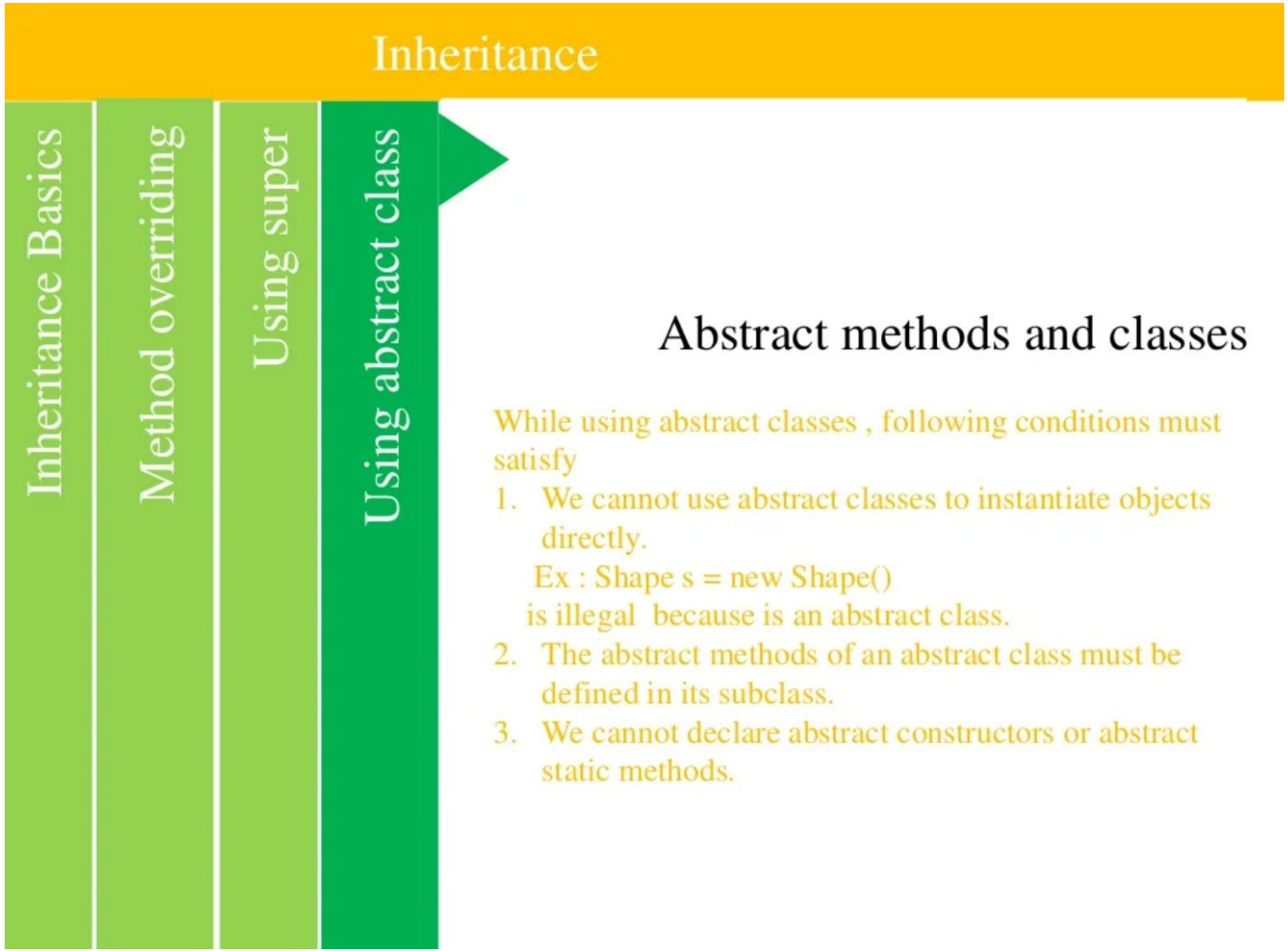
- Inheritance Basics
- Abstract methods and classes
- Using super
- Using abstract class

Abstract methods and classes

✓A final class can never be subclassed.
✓Java allows us to do somethin exactly opposite to this i.e. we can indicate that te method must always be redefined in a subclass , thus making overriding compulsory.
✓This is done by using the modifier keyword abstract.
✓Ex : abstract class shape

```
{
    ……...
    ……...
abstract void draw();
    ……
    ……
}
```

✓ A class with more than one abstract methods should be declared as abstract class


Inheritance
Inheritance Basics
Method overriding
Using super
Using abstract class
Abstract methods and classes
While using abstract classes , following conditions must satisfy
1. We cannot use abstract classes to instantiate objects directly.
Ex : Shape s = new Shape()
is illegal because is an abstract class.
2. The abstract methods of an abstract class must be defined in its subclass.
3. We cannot declare abstract constructors or abstract static methods.

Abstract Classes

- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- We use the modifier **`abstract`** on the class header to declare a class as abstract:

```
public abstract class Whatever
{
   // contents
}
```

Abstract Classes

- An abstract class often contains abstract methods with no definitions (like an interface does)
- Unlike an interface, the `abstract` modifier must be applied to each abstract method
- An abstract class typically contains non-abstract methods (with bodies), further distinguishing abstract classes from interfaces
- A class declared as abstract does not need to contain abstract methods

Abstract Classes

- The child of an abstract class must override the abstract methods of the parent, or it too will be considered abstract
- An abstract method cannot be defined as `final` (because it must be overridden) or `static` (because it has no definition yet)
- The use of abstract classes is a **design decision** – it helps us establish common elements in a class that is too general to instantiate

Interface Hierarchies

- Inheritance can be applied to interfaces as well as classes
- One interface can be derived from another interface
- The child interface inherits all abstract methods of the parent
- A class implementing the child interface must define all methods from both the ancestor and child interfaces
- All members of an interface are public

Abstract Classes

- Classes that cannot be instantiated
- A hybrid between regular classes and interfaces
- Some methods may be implemented while others are only specified
- **Used when the parent class cannot define a complete default implementation (implementation must be specified by the child class).**
- **Format:**

```
public abstract class <class name>
{
        <public/private/protected> abstract method ();
}
```

Abstract Classes (2)

- Example:

```
public abstract class BankAccount
{
    protected float balance;
    public void displayBalance ()
    {
        System.out.println("Balance $" + balance);
    }
    public abstract void deductFees () ;
}
```

The difference between abstract and non abstract methods:

The abstract methods cannot have implementations (like interface methods).

```
1  public void abstractMethod();
```

The non-abstract/concrete methods must have implementations (method body).

```
1  public void nonAbstractMethod() {
2  System.out.println("Non-abstract method");
3  }
```

1. For **abstract methods**, you have to provide an implementation in by creating a derived class.
2. For **non-abstract methods**, there is no need to provide an implementation in derived class.

Code

Abstract Methods:

1. **Definition**: An abstract method is declared without any implementation (i.e., without a body).
2. **Purpose**: Abstract methods are meant to be overridden in subclasses. They define a contract that any subclass must follow by providing an implementation.
3. **Class Context**: Abstract methods can only exist in abstract classes or interfaces.
4. **Syntax (Java Example)**:

   java Copy code

   ```java
   abstract class Shape {
       abstract void draw();  // No implementation
   }
   ```

5. **Usage**: Used when you want to force subclasses to provide specific behavior, but you don't know or don't want to specify that behavior in the parent class.

Non-Abstract Methods:

1. **Definition**: A non-abstract method is a method that has a full implementation (i.e., a method body).
2. **Purpose**: Non-abstract methods can be used directly by objects of that class or inherited by subclasses, though subclasses can choose to override them.
3. **Class Context**: Non-abstract methods can exist in both abstract and non-abstract (concrete) classes.
4. **Syntax (Java Example)**:

   java Copy code

   ```java
   class Shape {
       void draw() {
           System.out.println("Drawing shape");
       }
   }
   ```

5. **Usage**: Used when you want to define a specific behavior in the base class, which may or may not be overridden by subclasses.

In a nutshell, **abstract methods** define what a subclass should do, while **non-abstract methods** define how it should be done (or may provide a default behavior).

So the question arises why use interfaces when we have abstract classes?

The reason is, abstract classes may contain non-final variables, whereas variables in the interface are final, public, and static.

```
// A simple interface

interface Player
{
    final int id = 10;
    int move();
}
```

Note: If you make an interface final, you cannot implement its methods which defies the very purpose of the interfaces. Therefore, **you cannot make an interface final in Java**. Still if you try to do so, a compile time exception is generated saying “illegal combination of modifiers − interface and final”.

Q. WAJP to define abstract class shape with abstract methods area() and volume() and calculate area and volume of cone and cylinder

Volume and Area of Cone and Cylinder using abstract class

```
import java.util.*;
abstract class shape1
{
final float pi=3.14f;
abstract double area(int h,int r);
abstract double volume(int h,int r);
}
class cone extends shape1
{
double area(int h ,int r)
{
return (pi*r*(Math.sqrt((h*h)+(r*r))));
}
double volume(int h,int r)
{
return (pi*r*r*h/3);
}
}
class cylinder extends shape1
{
double area(int h ,int r)
{
return ((2*pi*r*h)+(2*pi*r*r));
}
double volume(int h,int r)
{
return (pi*r*r*h);
}
}
```

```
class shape
{
public static void main (String args[])
{
Scanner sc=new Scanner(System.in);
cone obj1= new cone();
cylinder obj2=new cylinder();
int r1,r2,h1,h2;
System.out.println("Enter radius of cone:");
r1=sc.nextInt();
System.out.println("Enter height of cone:");
h1=sc.nextInt();
System.out.println("Enter radius of cylinder:");
r2=sc.nextInt();
System.out.println("Enter height of cylinder:");
h2=sc.nextInt();
System.out.println("Area of cone is"+obj1.area(h1,r1));
System.out.println("Volume of cone is"+obj1.volume(h1,r1));
System.out.println("Area of cylinder is"+obj2.area(h2,r2));
System.out.println("Volume of cylinder is"+obj2.volume(h2,r2));
}
}
```

OUTPUT

```
 java shape
Enter radius of cone:
25
Enter height of cone:
70
Enter radius of cylinder:
20
Enter height of cylinder:
100
Area of cone is5834.931983322514
Volume of cone is45791.66796875
Area of cylinder is15072.0009765625
Volume of cylinder is125600.0
```

Constructor in Java Abstract Class

Constructor is always called by its class name in a class itself. A constructor is used to initialize an object not to build the object. As we all know abstract classes also do have a constructor. So if we do not define any constructor inside the abstract class then JVM (Java Virtual Machine) will give a default constructor to the abstract class

PROCEDURE

- If you define your own constructor without arguments inside an abstract class but forget to call your own constructor inside its derived class constructor then JVM will call the constructor by default.
  - So if you define your single or multi-argument constructor inside the abstract class then make sure to call the constructor inside the derived class constructor with the **super keyword**.

Implementation: Here in this program, we are going to multiply two numbers by using the following above approach as mentioned.

Step 1: We create an abstract class named '*Content*' and define a user define a constructor with one argument, variable with name '*a*', and an abstract method named as '*multiply*'

Step 2: We create a class that must be derived from this abstract class 'Content' named 'GFG'. Inside GFG class we are going to define a constructor and inside the method call the parent class constructor by using the super keyword and define the abstract method of its parent class in it.

Step 3: Now in the main class of our function that is '*GeeksforGeeks*' here, where we will create an object of abstract class 'Content' by reference to its derived class object. Then we call the method of the abstract class by its object.

Step 4: Inside the method, we multiply both the value stored in the different variable names where one of the variables is the variable of an abstract class. We can access the variable of the abstract class by its derived class object.

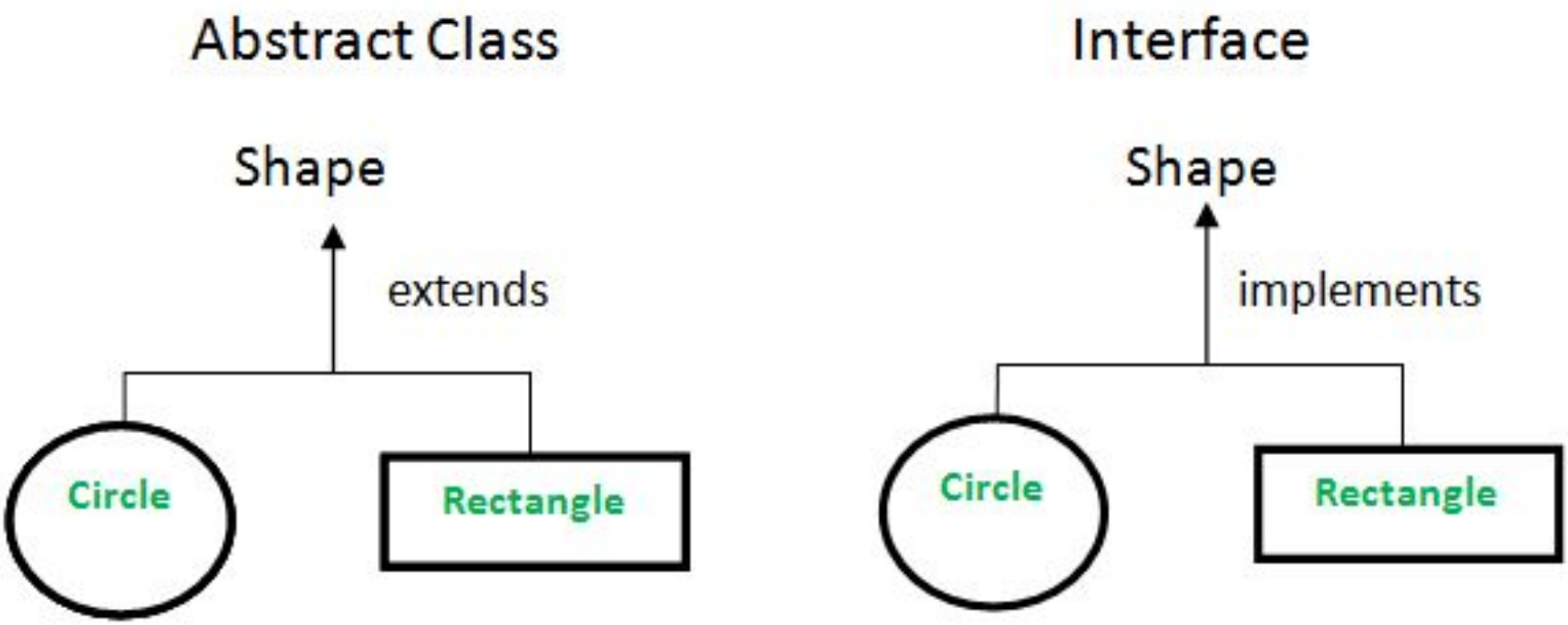
Code:
https://docs.google.com/document/d/1tM6La9VpnZNpBbQ9tcV11IMaogQTkrpxjKrWp1_O1EU/edit?usp=sharing

OOPs interface vs abstract class

Interface	Abstract class
Interface support multiple implementations.	Abstract class does not support multiple inheritance.
Interface does not contain Data Member	Abstract class contains Data Member
Interface does not contain Constructors	Abstract class contains Constructors
An interface Contains only incomplete member (signature of member)	An abstract class Contains both incomplete (abstract) and complete member
An interface cannot have access modifiers by default everything is assumed as public	An abstract class can contain access modifiers for the subs, functions, properties
Member of interface can not be Static	Only Complete Member of abstract class can be Static

Difference between Abstract class and Interface

Abstract class	Interface
It is collection of abstract method and concrete methods.	It is collection of abstract method.
There properties can be reused commonly in a specific application.	There properties commonly usable in any application of java environment.
It does not support multiple inheritance.	It support multiple inheritance.
Abstract class is preceded by abstract keyword.	It is preceded by Interface keyword.
Which may contain either variable or constants.	Which should contains only constants.
The default access specifier of abstract class methods are default.	There default access specifier of interface method are public.


Abstract Class
Shape
extends
Circle
Rectangle
Interface
Shape
implements
Circle
Rectangle

CODING ACTIVITY

#1. WAJP to find area of circle and rectangle using method overriding

Area of rectangle and circle using method overriding

```
class OverloadDemo
{
void area(float x,float y)
{
System.out.println("The area of the rectangle is "+x*y+"sq
units");
}
void area(double x)
{
double z= 3.14*x*x;
System.out.println("The area of the circle is "+z+"sq units");
}
}
class Overload
{
public static void main(String args[])
{
OverloadDemo ob= new OverloadDemo();
ob.area(10,20);
ob.area(2.5);
}
}
```

```
OUTPUT:
java Overload
The area of the rectangle is 200.0sq units
The area of the circle is 19.625sq units
```

Thank you