

Unit-2

Inheritance and Polymorphism:-

1. Types of inheritance
2. Member access rules
3. Super uses
4. Using final with inheritance
5. The object class and its methods
6. Method Overriding
7. Dynamic Binding
8. Abstract Classes
9. Abstract Methods

Inheritance:-

Definition:- When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

- We can achieve inheritance by using the **extends** keyword.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Terms used in Inheritance

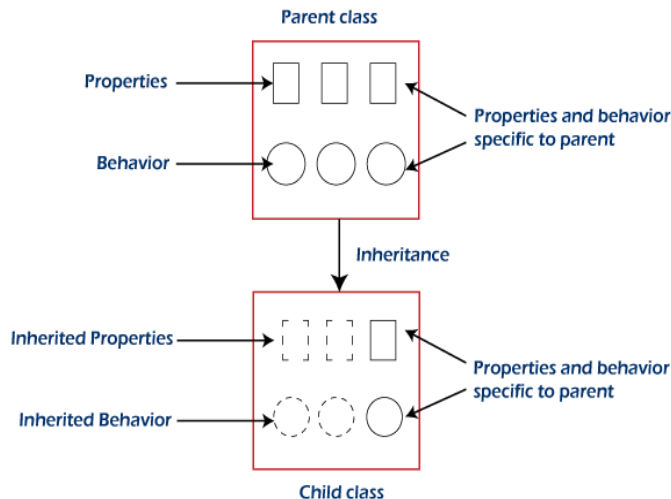
- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

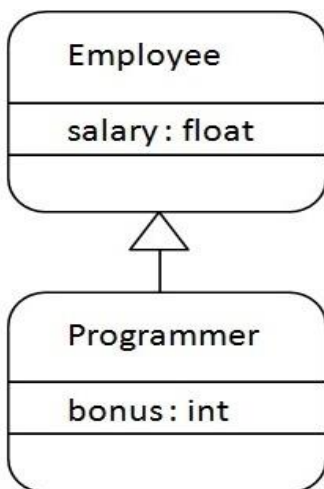
```
class Subclass-name extends Superclass-name
{
    //methods and fields
}
```

- A class whose properties are inherited is known as **parent class** and a class that inherits the properties of the parent class is known as **child class**. Thus, it establishes a relationship between parent and child class that is known as parent-child or **Is-a** relationship.

- Suppose, there are two classes named **Father** and **Child** and we want to inherit the properties of the Father class in the Child class. We can achieve this by using the **extends** keyword.



Inheritance Example



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

Example program:-

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

```
}  
}
```

Output:-

```
Programmer salary is:40000.0  
Bonus of programmer is:10000
```

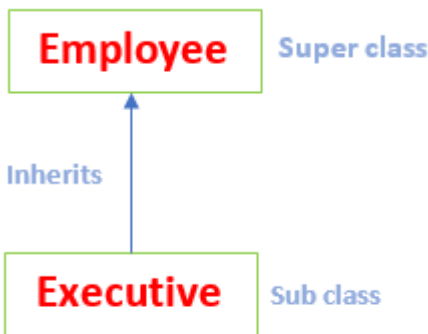
1.Types of Inheritance:-

- Single Inheritance
- Multi-level Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance
- Multiple Inheritance

Note: Multiple inheritance is not supported in Java.

1.Single Inheritance

In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behavior of a single-parent class. Sometimes it is also known as **simple inheritance**.



Single Inheritance

In the above figure, Employee is a parent class and Executive is a child class. The Executive class inherits all the properties of the Employee class.

Let's implement the single inheritance mechanism in a Java program.

Executive.java

```

class Employee
{
float salary=34534*12;
}
public class Executive extends Employee
{
float bonus=3000*6;
public static void main(String args[])
{
Executive obj=new Executive();
System.out.println("Total salary credited: "+obj.salary);
System.out.println("Bonus of six months: "+obj.bonus);
}
}

```

Output:

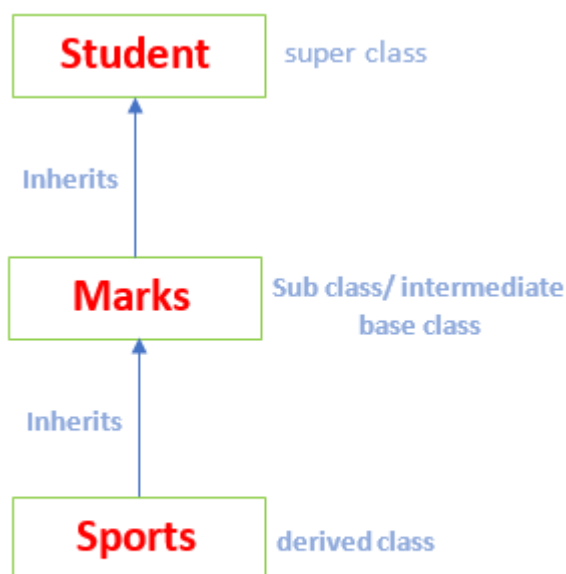
```

Total salary credited: 414408.0
Bonus of six months: 18000.0

```

2.Multi-level Inheritance

In **multi-level inheritance**, a class is derived from a class which is also derived from another class is called multi-level inheritance. In simple words, we can say that a class that has more than one parent class is called multi-level inheritance. Note that the classes must be at different levels. Hence, there exists a single base class and single derived class but multiple intermediate base classes.



Multi-level Inheritance

In the above figure, the class Marks inherits the members or methods of the class Students. The class Sports inherits the members of the class Marks. Therefore, the Student class is the parent class of the class Marks and the class Marks is the parent of the class Sports. Hence, the class Sports implicitly inherits the properties of the Student along with the class Marks.

Let's implement the multi-level inheritance mechanism in a Java program.

MultilevelInheritanceExample.java

```
//super class
class Student
{
    int reg_no;
    void getNo(int no)
    {
        reg_no=no;
    }
    void putNo()
    {
        System.out.println("registration number= "+reg_no);
    }
}

//intermediate sub class
class Marks extends Student
{
    float marks;
    void getMarks(float m)
    {
        marks=m;
    }
    void putMarks()
    {
        System.out.println("marks= "+marks);
    }
}

//derived class
class Sports extends Marks
{
    float score;
    void getScore(float scr)
    {
        score=scr;
    }
}
```

```

void putScore()
{
    System.out.println("score= "+score);
}

public class MultilevelInheritanceExample
{
    public static void main(String args[])
    {
        Sports ob=new Sports();
        ob.getNo(0987);
        ob.putNo();
        ob.getMarks(78);
        ob.putMarks();
        ob.getScore(68.7);
        ob.putScore();
    }
}

```

Output:

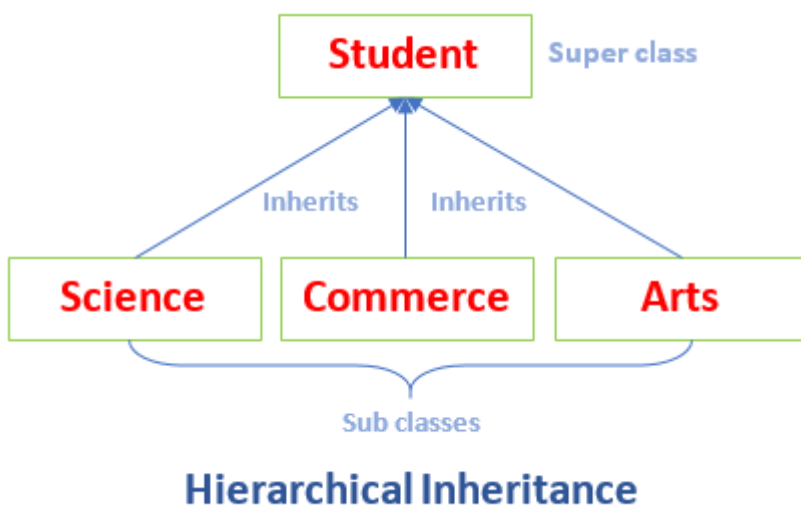
```

registration number= 0987
marks= 78.0
score= 68.7

```

3.Hierarchical Inheritance

If a number of classes are derived from a single base class, it is called **hierarchical inheritance**.



In the above figure, the classes Science, Commerce, and Arts inherit a single parent class named Student.

Let's implement the hierarchical inheritance mechanism in a Java program.

HierarchicalInheritanceExample.java

```
//parent class
class Student
{
    public void methodStudent()
    {
        System.out.println("The method of the class Student invoked.");
    }
}

class Science extends Student
{
    public void methodScience()
    {
        System.out.println("The method of the class Science invoked.");
    }
}

class Commerce extends Student
{
    public void methodCommerce()
    {
        System.out.println("The method of the class Commerce invoked.");
    }
}

class Arts extends Student
{
    public void methodArts()
    {
        System.out.println("The method of the class Arts invoked.");
    }
}

public class HierarchicalInheritanceExample
{
    public static void main(String args[])
    {
        Science sci = new Science();
        Commerce comm = new Commerce();
        Arts art = new Arts();
        //all the sub classes can access the method of super class
        sci.methodStudent();
    }
}
```

```

comm.methodStudent();
art.methodStudent();
}
}

```

Output:

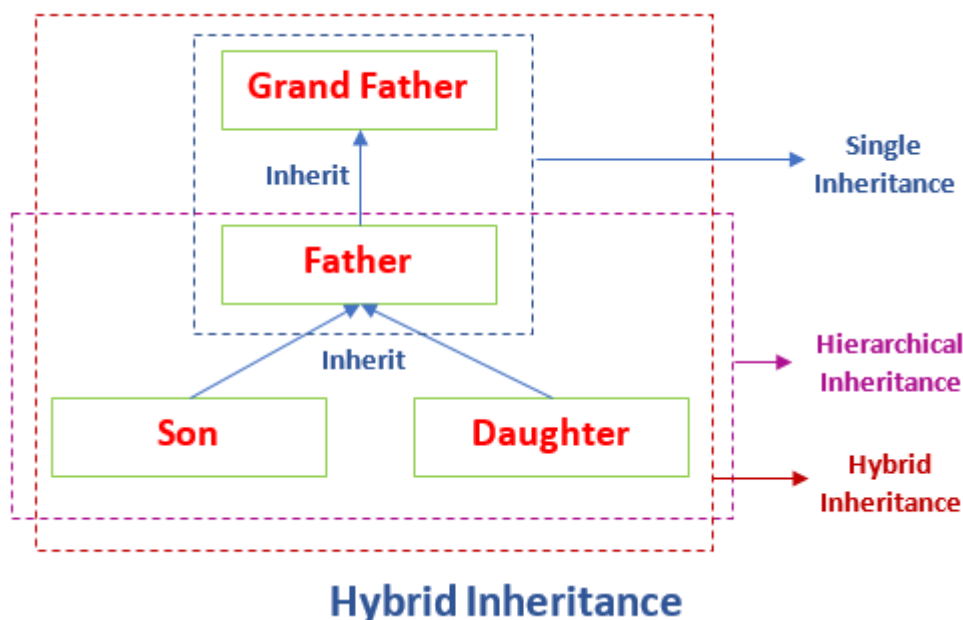
```

The method of the class Student invoked.
The method of the class Student invoked.
The method of the class Student invoked.

```

4. Hybrid Inheritance

Hybrid means consist of more than one. Hybrid inheritance is the combination of two or more types of inheritance.



- In the above figure, GrandFather is a super class. The Father class inherits the properties of the GrandFather class. Since Father and GrandFather represents single inheritance. Further, the Father class is inherited by the Son and Daughter class. Thus, the Father becomes the parent class for Son and Daughter. These classes represent the hierarchical inheritance. Combinedly, it denotes the hybrid inheritance.

Let's implement the hybrid inheritance mechanism in a Java program.

Daughter.java

```

//parent class
class GrandFather
{
    public void show()
    {

```

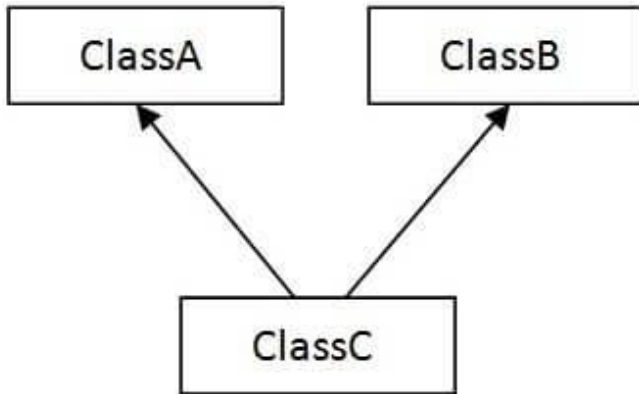


```
System.out.println("I am grandfather.");
}
}
//inherits GrandFather properties
class Father extends GrandFather
{
public void show()
{
System.out.println("I am father.");
}
}
//inherits Father properties
class Son extends Father
{
public void show()
{
System.out.println("I am son.");
}
}
//inherits Father properties
public class Daughter extends Father
{
public void show()
{
System.out.println("I am a daughter.");
}
public static void main(String args[])
{
Daughter obj = new Daughter();
obj.show();
}
}
```

Output:

```
I am daughter.
```

5. Multiple Inheritance (not supported)



4) Multiple

- Java does not support multiple inheritances due to ambiguity. For example, consider the following Java program.

Demo.java

```
class Wishes
{
void message()
{
System.out.println("Best of Luck!!");
}
}
class Birthday
{
void message()
{
System.out.println("Happy Birthday!!");
}
}
public class Demo extends Wishes, Birthday //considering a scenario
{
public static void main(String args[])
{
Demo obj=new Demo();
//can't decide which classes' message() method will be invoked
obj.message();
}
```

}

- The above code gives error because the compiler cannot decide which message() method is to be invoked. Due to this reason, Java does not support multiple inheritances at the class level but can be achieved through an **interface**.

2.Member Access Rules:-

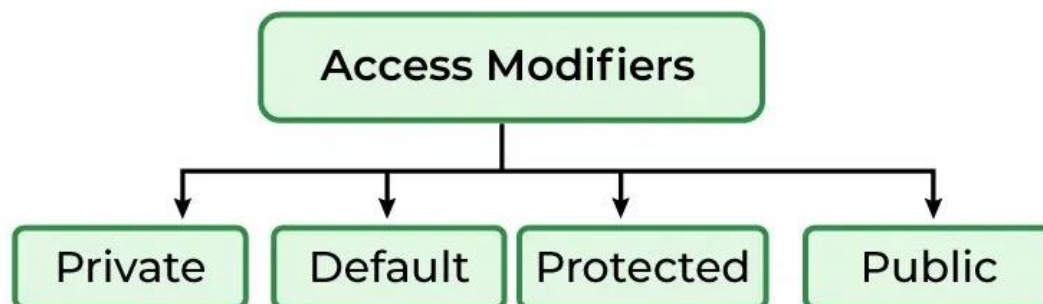
Member Access Rules in Inheritance

The following rules for inherited methods are enforced –

- Methods declared public in a superclass also must be public in all subclasses.
- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
- Methods declared private are not inherited at all, so there is no rule for them.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers :



	Default	Private	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package Subclass	Yes	No	Yes	Yes
Same Package Non-Subclass	Yes	No	Yes	Yes
Different Package Subclass	No	No	Yes	Yes
Different Package Non-Subclass	No	No	No	Yes

1) Private

The access level of a private modifier is only within the class. It cannot be accessed from outside the class and sub class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
public static void main(String args[]){
A obj=new A();
System.out.println(obj.data);//Compile Time Error
obj.msg();//Compile Time Error
}
}
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A{
private A(){ }//private constructor
void msg(){System.out.println("Hello java");}
}

public class Simple{
public static void main(String args[]){
A obj=new A();//Compile Time Error
}
}
```

Note: A class cannot be private or protected except nested class.

2) Default

The access level of a default modifier is only within the sub classes, class and package . It cannot be accessed from outside the sub class, class and package. If you do not specify any keyword, it will be the default.

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

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

//save by B.java
package mypack;
import pack.*;
class B{
    public static void main(String args[]){
        A obj = new A();//Compile Time Error
        obj.msg();//Compile Time Error
    }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) Protected

The **protected access modifier** is accessible within sub class, class and package and outside the sub class package but through inheritance only.

- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.

- The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifier.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

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

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

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

Output:Hello

4) Public

The **public access modifier** is accessible everywhere. It can be accessed from within the class, outside the class, within the package and outside the package. It has the widest scope among all other modifiers.

- Methods declared public in a superclass also must be public in all subclasses.

Example of public access modifier

```
//save by A.java
```

```
package pack;
public class A{
```

```
public void msg(){System.out.println("Hello");}
}
```

//save by B.java

```
package mypack;
import pack.*;
```

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

Output:Hello

Java Access Modifiers with Method Overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

```
class A{
protected void msg(){System.out.println("Hello java");} //used protected
}

public class Simple extends A{
void msg(){System.out.println("Hello java");} //C.T.Error //assigned default
public static void main(String args[]){
  Simple obj=new Simple();
  obj.msg();
}
}
```

The default modifier is more restrictive than protected. That is why, there is a compile-time error.

3.Super Uses:-

Super Keyword in Java

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

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

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

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

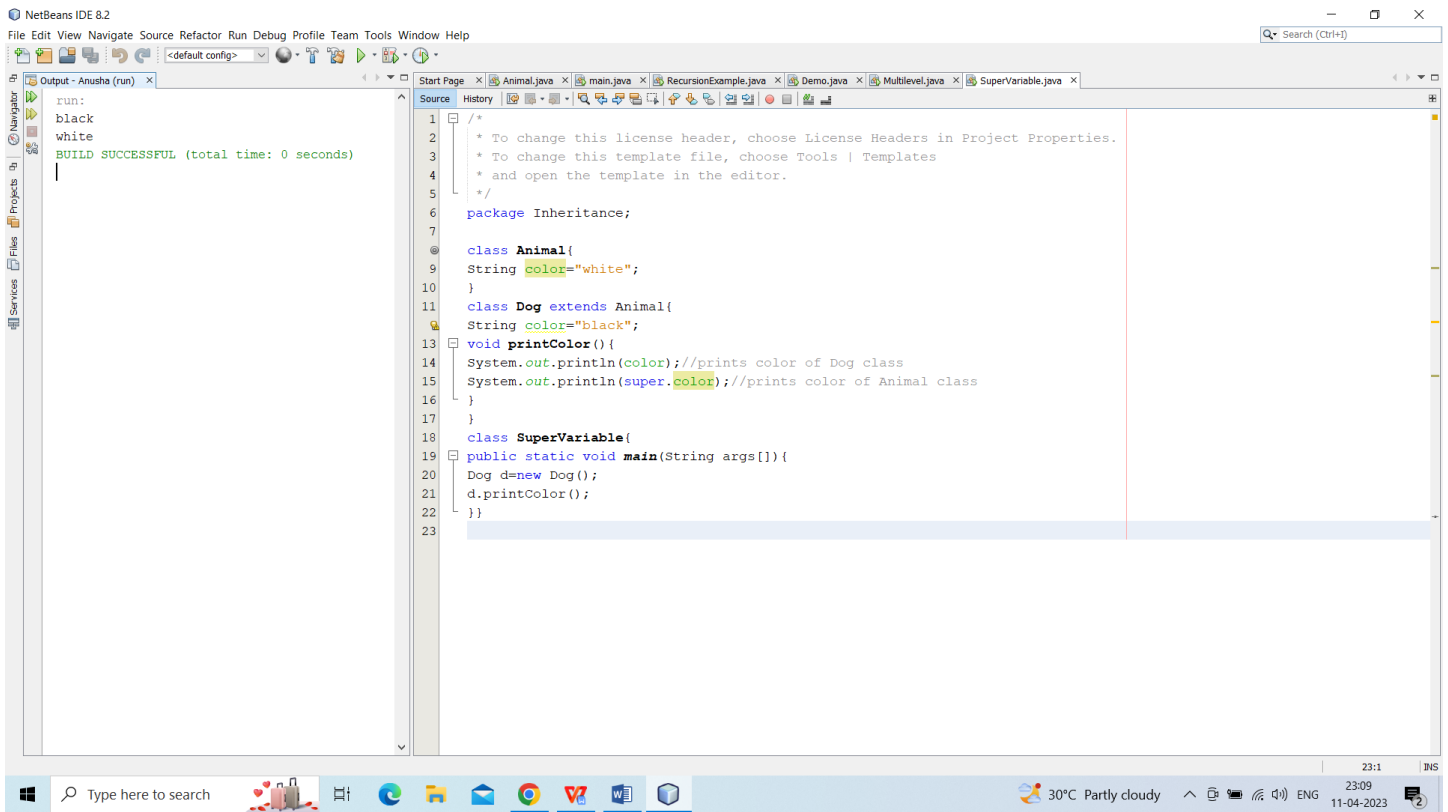
Example program:-

```
1. class Animal{
2. String color="white";
3. }
4. class Dog extends Animal{
5. String color="black";
6. void printColor(){
7. System.out.println(color);//prints color of Dog class
8. System.out.println(super.color);//prints color of Animal class
9. }
10. }
11. class TestSuper1{
12. public static void main(String args[]){
13. Dog d=new Dog();
14. d.printColor();
15. }}
```

Output:-

black

white



In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

2) super can be used to invoke parent class method

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

Example Program:-

```
1. class Animal1{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog1 extends Animal1{
5. void eat(){System.out.println("eating bread...");}
6. void bark(){System.out.println("barking...");}
7. void work(){
8. super.eat();
9. bark();
10. }
11. }
12. class TestSuper2{
13. public static void main(String args[]){
14. Dog1 d=new Dog1();
15. d.work();
16. }}
```

Output:-

eating...

barking...

```
run:
eating...
barking...
BUILD SUCCESSFUL (total time: 0 seconds)

package Inheritance;

class Animal{
    void eat(){System.out.println("eating...");}
}

class Dog1 extends Animal{
    void eat(){System.out.println("eating bread...");}
    void bark(){System.out.println("barking...");}
    void work(){
        super.eat();
        bark();
    }
}

class SuperMethod{
    public static void main(String args[]){
        Dog1 d=new Dog1();
        d.work();
    }
}
```

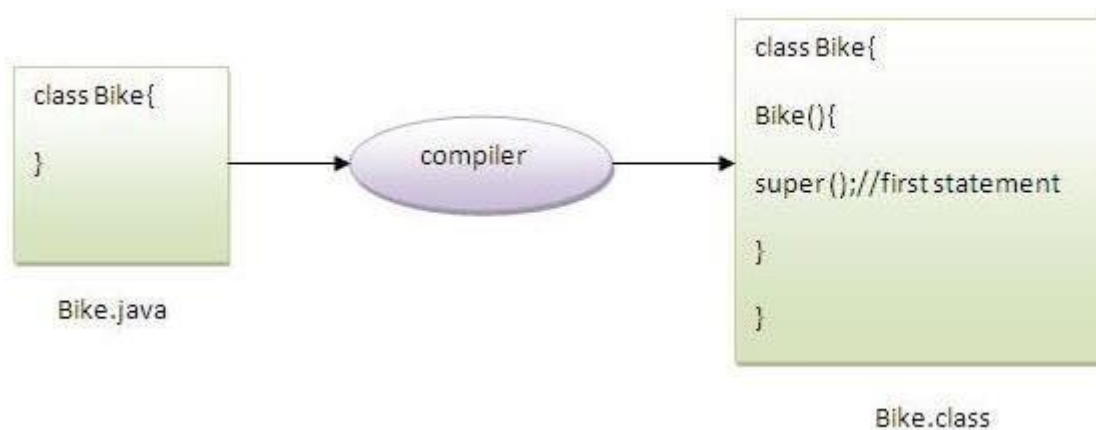
In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

To call the parent class method, we need to use super keyword.

3) super is used to invoke parent class constructor.

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

Note: super() is added in each class constructor automatically by compiler if there is no super() or this().



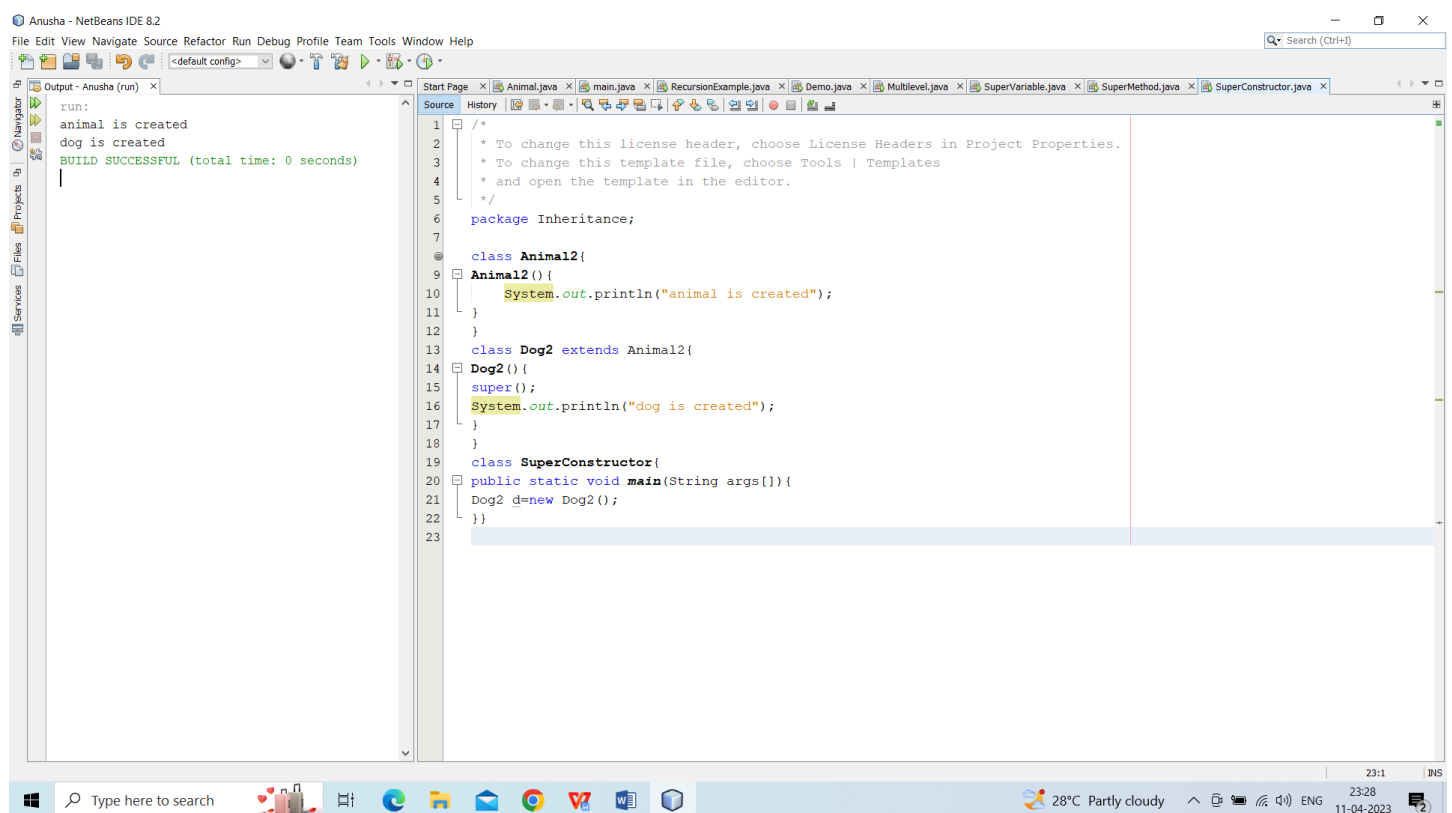
Example program:-

1. **class** Animal2{
2. Animal2(){System.out.println("animal is created");}
3. }
4. **class** Dog2 **extends** Animal2{
5. Dog2(){
6. **super**();//without use of super keyword java compiler automatically executed
7. System.out.println("dog is created");
8. }
9. }
10. **class** TestSuper3{
11. **public static void** main(String args[]){
12. Dog2 d=**new** Dog2();
13. }}

Output:-

animal is created

dog is created



Does a super class reference be used to refer sub class object?justify

Ans:-

- So difference between referencing using superclass reference and referencing using subclass reference is use superclass referencing can holds object of subclass and could only access the methods which are defined/overridden by subclass while use subclass referencing can not hold object of superclass and could access the methods of both superclass and subclass.
- If you assign an object of the subclass to the reference variable of the superclass then the subclass object is converted into the type of superclass and this process is termed as widening (in terms of references).

```
class Animal{  
    void eat(){System.out.println("animal is eating...");}  
}
```

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

```
public static void main(String args[]){  
    Animal a=new Dog();  
    a.eat();  
}
```

Output:-

Dog is eating

4.Using Final With Inheritance:-

Final Keyword In Java

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

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

1) Java final variable

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

Example of final variable

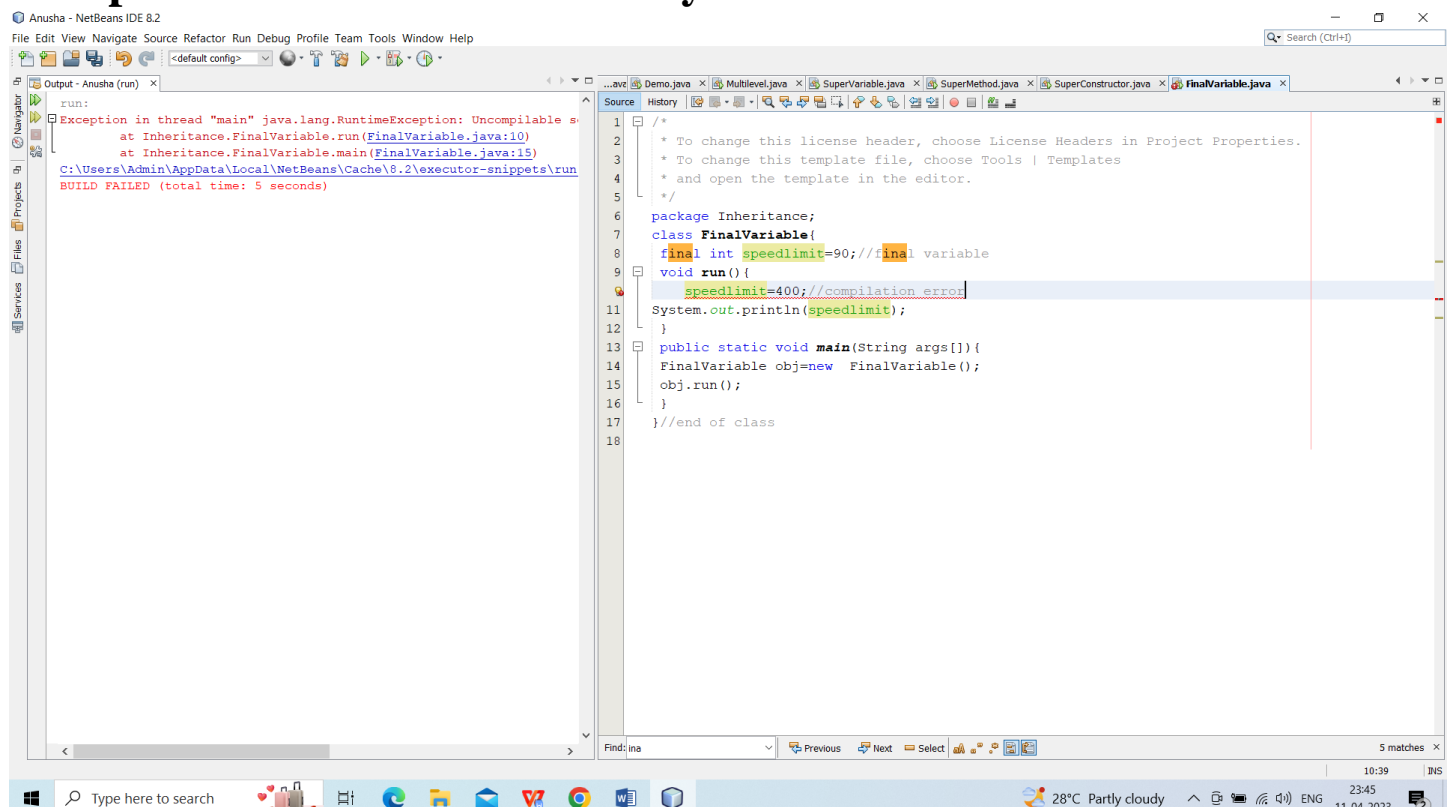
There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

Example Program Final Variable:-

1. **class** Bike9{
2. **final int** speedlimit=90;//final variable
3. **void** run(){
4. speedlimit=400;
5. }
6. **public static void** main(String args[]){
7. Bike9 obj=new Bike9();
8. obj.run();
9. }
10. }//end of class

OutPut:-

Compile time error //can't modify final variable value



2) Java final method

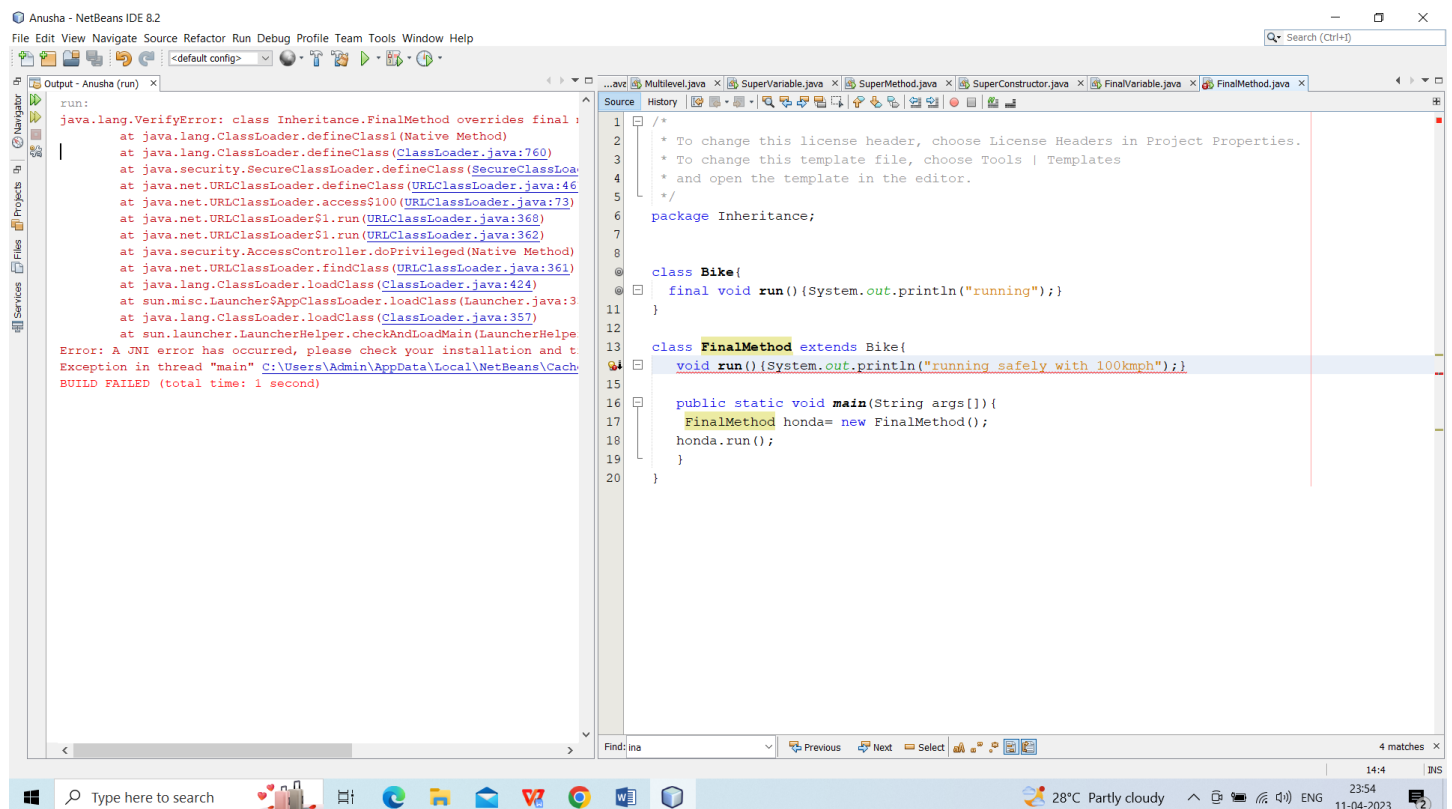
If you make any method as final, you cannot override it.

Example of final method

```
1. class Bike{
2.     final void run(){System.out.println("running");}
3. }
4.
5. class Honda extends Bike{
6.     void run(){System.out.println("running safely with 100kmph");}
7.
8.     public static void main(String args[]){
9.         Honda honda= new Honda();
10.        honda.run();
11.    }
12. }
```

OutPut:-

Compile time error//can't override method



3) Java final class

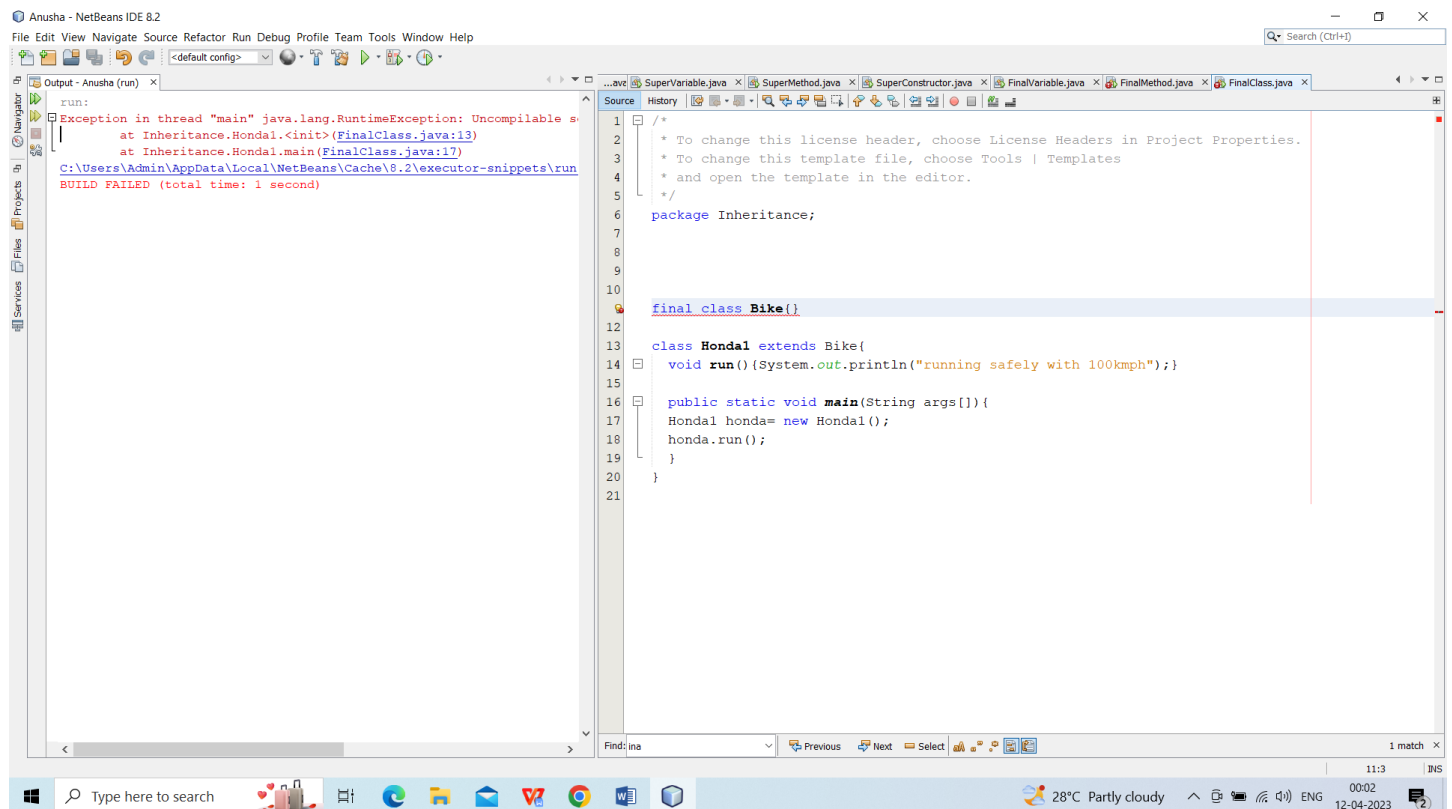
If you make any class as final, you cannot extend it.

Example of final class

1. **final class** Bike{ }
- 2.
3. **class** Honda1 **extends** Bike{
4. **void** run(){System.out.println("running safely with 100kmph");}
- 5.
6. **public static void** main(String args[]){
7. Honda1 honda= **new** Honda1();
8. honda.run();
9. }
10. }

OutPut:-

Compile time error //can't inherited



5.Object class in Java

Object Class in Java:-

Object Class in Java is the topmost class among all the classes in Java. We can also say that the Object class in Java is the parent class for all the classes. It means that all the classes in Java are derived classes and their base class is the Object class. **Object** class is present in **java.lang** package.

All the classes directly or indirectly inherit from the Object class in Java.

Ex:-public class Object{

//Object class body

/*

1. toString()
 2. hashCode()
 3. equals(Object o)
 4. clone()
 5. finalize()
 6. getClass()
 7. wait()
 8. wait(long timeout)
 9. wait(long timeout,int nanos)
 - 10.notify()
 - 11.notifyAll()
- */**

}

/*Consider the example-

class Vehicle{

//body of class Vehicle

}

Here we can see that class Vehicle is not inheriting any class, but it inherits the Object class. It is an example of the direct inheritance of object class.

Now, look at this example-

class Vehicle{

//body of class Vehicle

}

class Car extends Vehicle{

//body of class Car

}

Here we can see that the Car class directly inherits the Vehicle class. The extends keyword is used to establish inheritance between two classes. But we have seen in the above example that if we define a class without inheriting it from some other class, it directly inherits the Object class. Since Car inherits the Vehicle and the Vehicle inherits the Object class, Car indirectly inherits the Object class. */

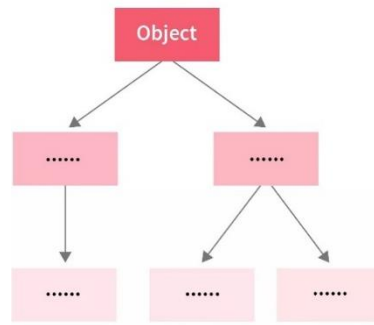


Fig.1.Object Class

Methods of Object class

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

Object class provides many methods. They are as follows:

6.Method Overriding:-

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**. In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

Rules for Java Method Overriding

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

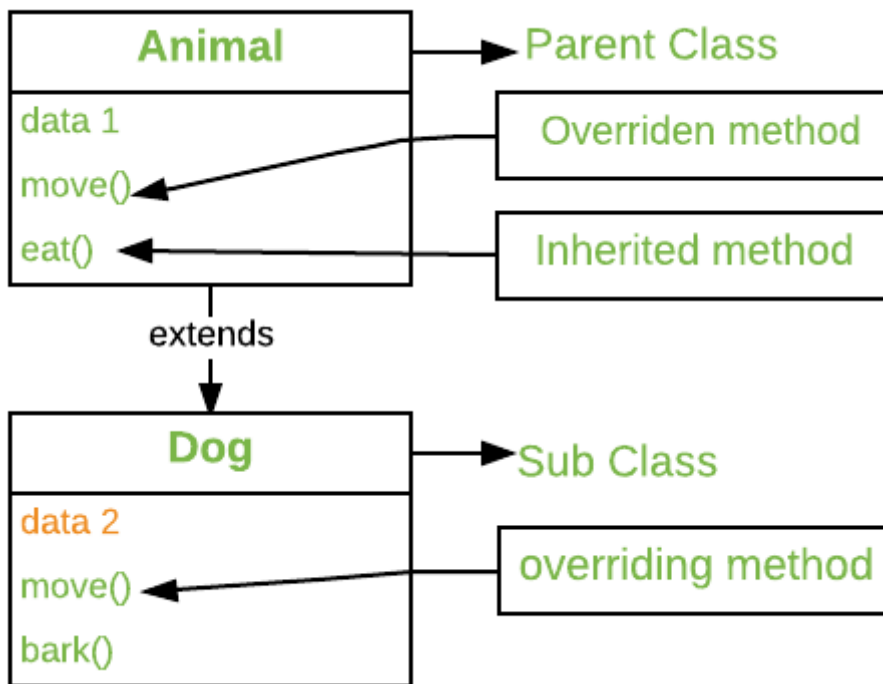


Fig.1.Method overriding process

Syntax:-class A

```
{  
Void show()  
{  
// block of code  
}
```

```
}
```

Class B extends A

```
{
```

Void show()

```
{
```

//block of code

```
}
```

```
}
```

Example of method overriding:-

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

//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");}  
}
```

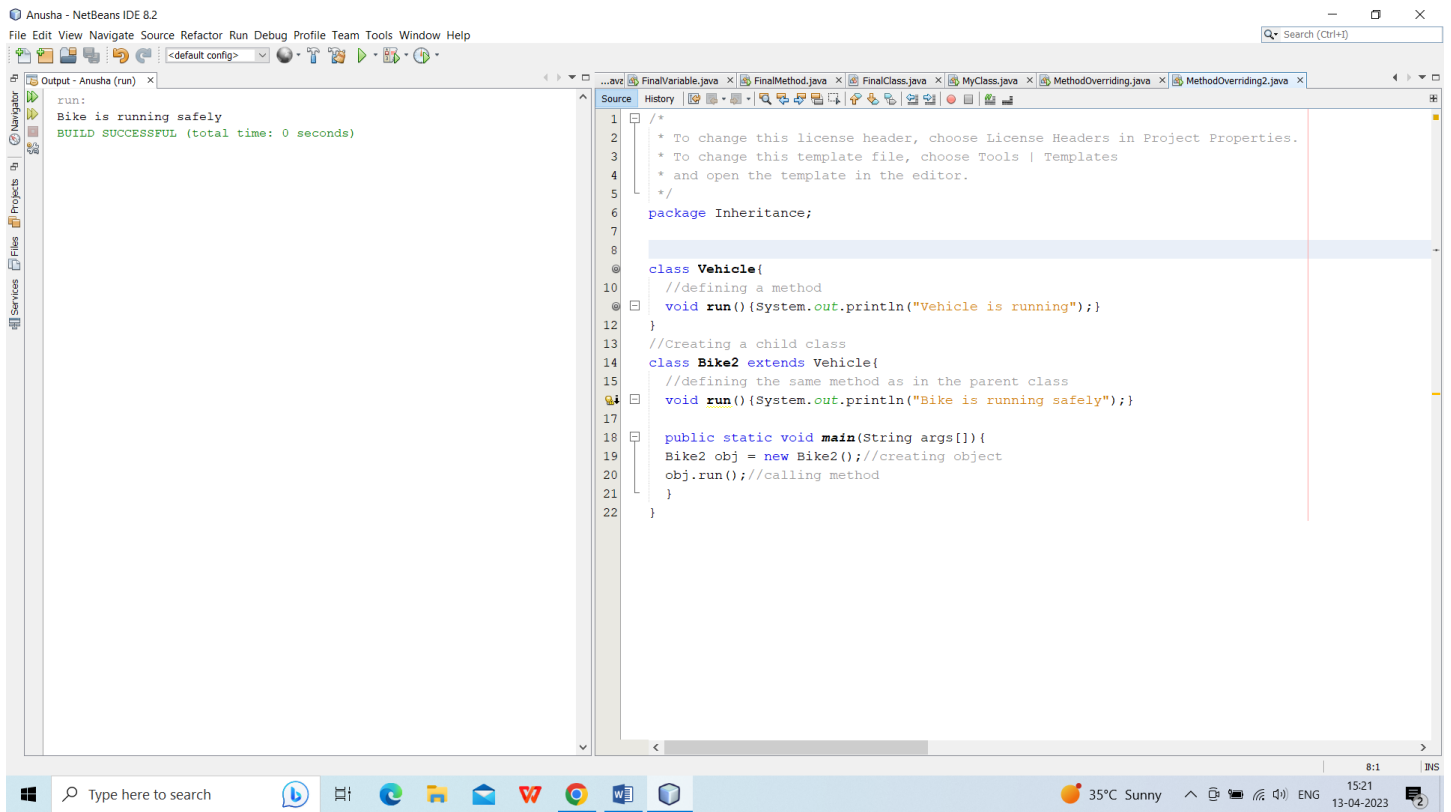
//Creating a child class

```
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



Example 2:-

A real example of Java Method Overriding

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.

```
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());

}

}

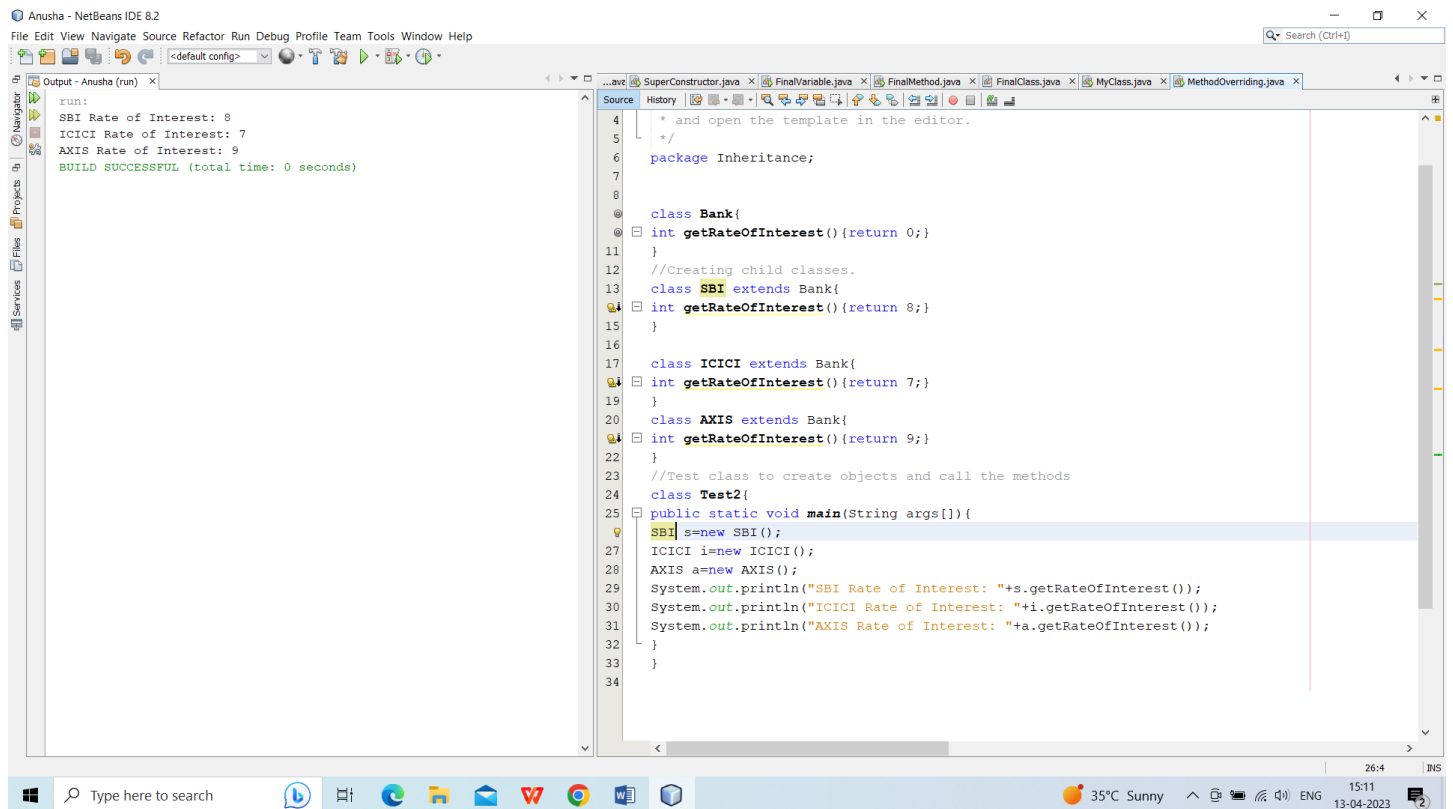
```

output:

```

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

```



7.Dynamic Binding:-

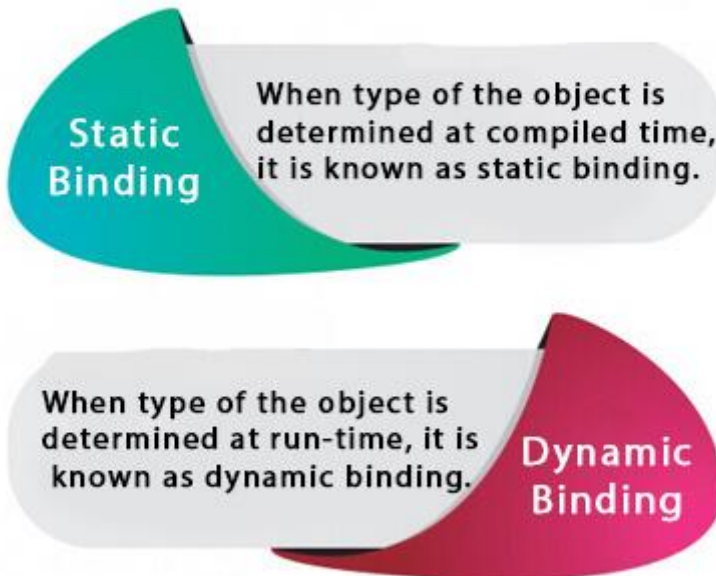
Connecting a method call to the method body is known as binding.

There are two types of binding

1. Static Binding (also known as Early Binding).
2. Dynamic Binding (also known as Late Binding).

Dynamic Binding is also known as **runtime polymorphism** or dynamic method dispatch.

Static vs Dynamic Binding



Understanding Type

Let's understand the type of instance.

1) variables have a type

Each variable has a type, it may be primitive and non-primitive.

Ex:- `int data=30;`

Here data variable is a type of int.

2) References have a type

```
class Dog{  
    public static void main(String args[]){  
        Dog d1;//Here d1 is a type of Dog  
    }  
}
```

3) Objects have a type

An object is an instance of particular java class, but it is also an instance of its superclass.

```
class Animal{ }
```

```
class Dog extends Animal{  
    public static void main(String args[]){
```

```
Dog d1=new Dog();
```

```
}
```

```
}
```

Here d1 is an instance of Dog class, but it is also an instance of Animal.

static binding

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

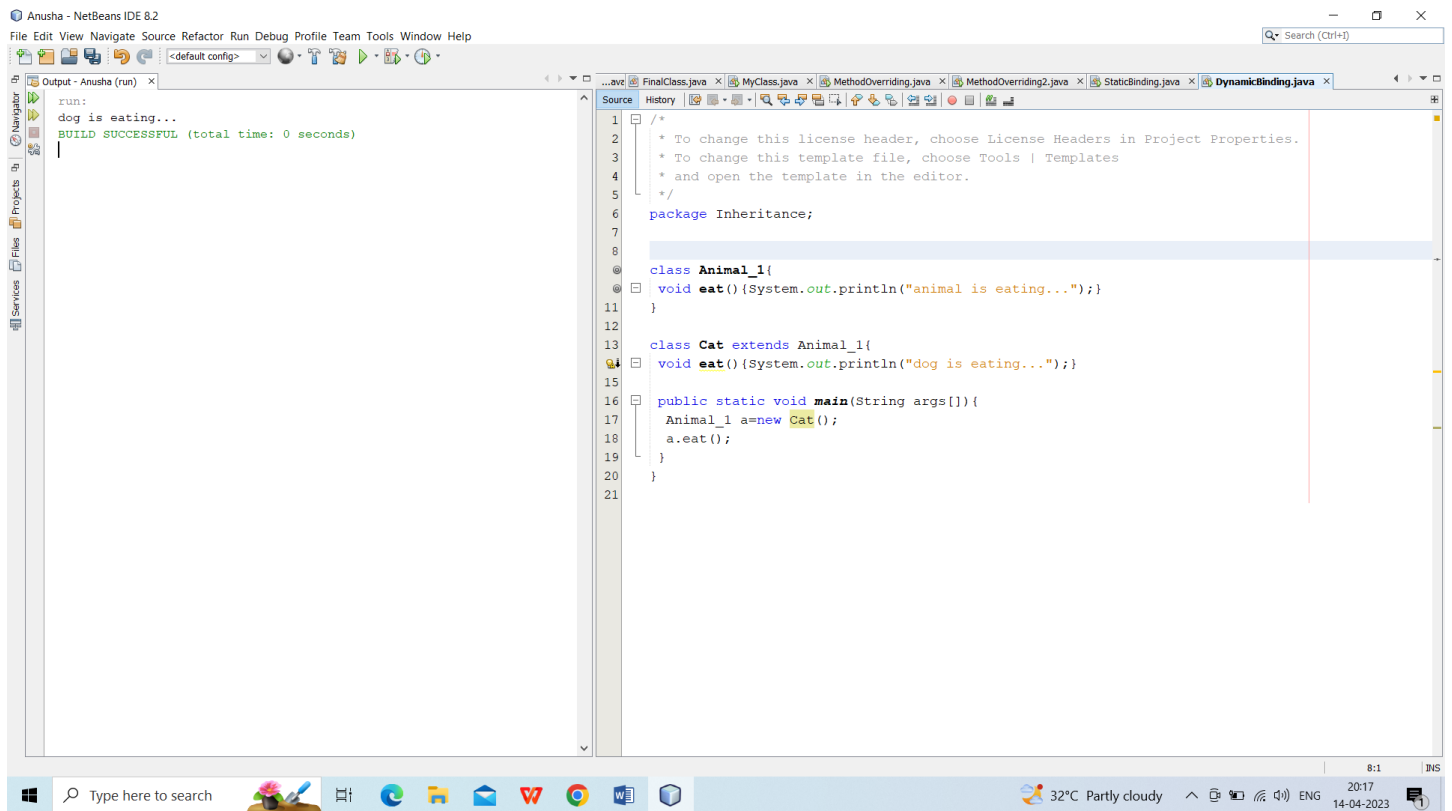
```
class Animal{  
    void eat(){System.out.println("animal is eating...");}  
}
```

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

```
public static void main(String args[]){  
    Animal a=new Dog();  
    a.eat();  
}  
}
```

//In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal. So compiler doesn't know its type, only its base type.

Output:-
dog is eating...



8.Abstraction:-

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

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

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

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

Abstract class:-

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Rules For Java Abstract Class:-

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have [constructors](#) and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

- To use an abstract class, you have to inherit it from sub classes.
- If a class contain partial implementation, then we should declare a class as abstract.

Syntax:-

```
abstract class class_name{  
  
//abstract methods and non-abstract methods  
  
}
```

Abstract Method

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

Syntax of abstract method

```
abstract access modifier return type method_name();//no method body and abstract method
```

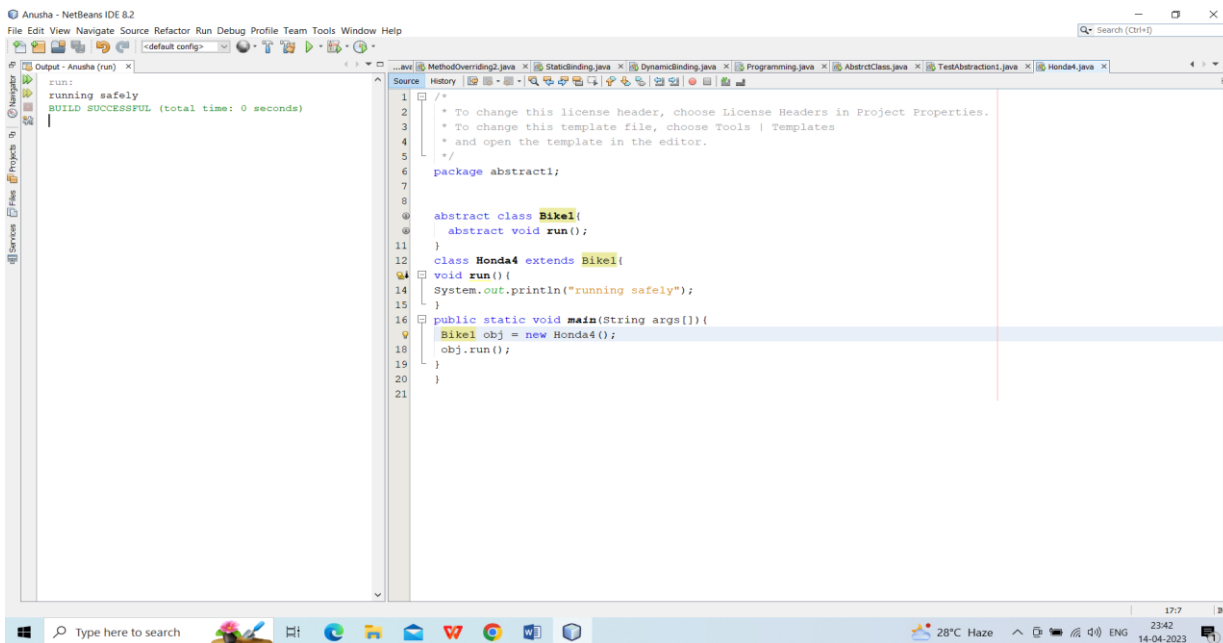
Abstract class Example1:-

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike{  
    abstract void run();  
}  
class Honda4 extends Bike{  
    void run(){  
        System.out.println("running safely");  
    }  
    public static void main(String args[]){  
        Bike obj = new Honda4();  
        obj.run();  
    }  
}
```

Output:-

running safely



Abstract Method Using Real Scenario:-Example 2

```
abstract class Shape{  
abstract void draw();  
}
```

//In real scenario, implementation is provided by others i.e. unknown by end user

```
class Rectangle extends Shape{  
void draw(){System.out.println("drawing rectangle");}  
}  
class Circle1 extends Shape{  
void draw(){System.out.println("drawing circle");}  
}
```

//In real scenario, method is called by programmer or user

```
class TestAbstraction1 {  
public static void main(String args[]){
```

Shape s=**new** Circle1();//In a real scenario, object is provided through method

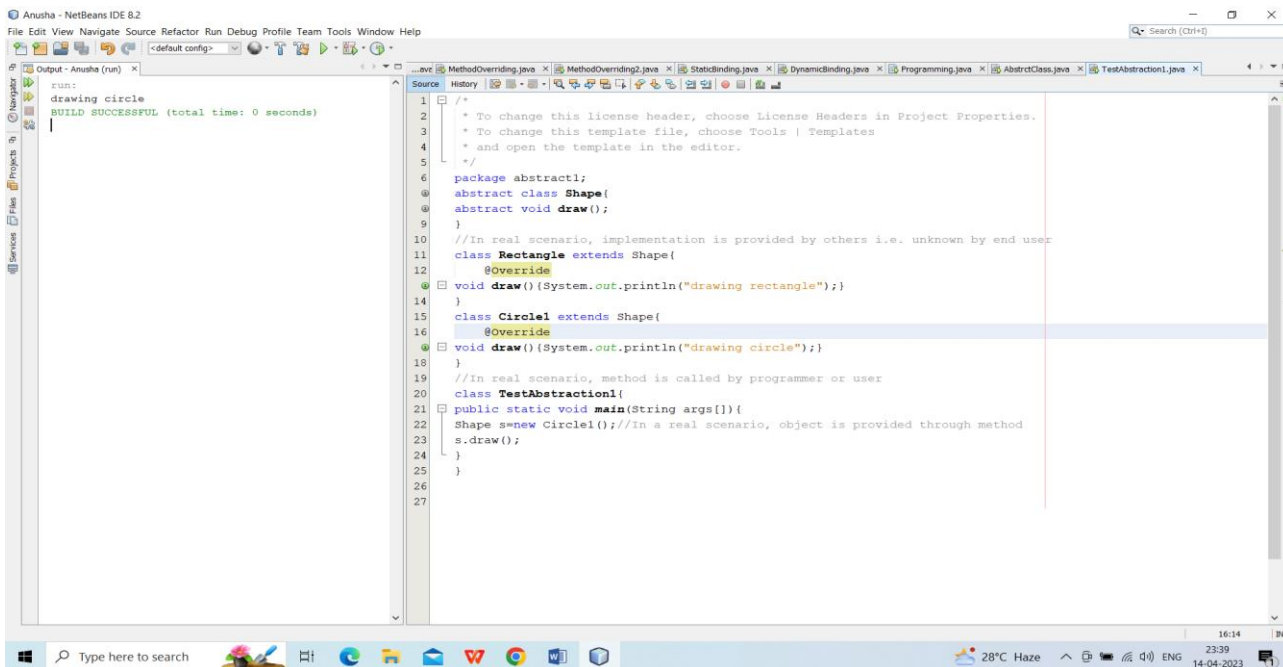
```
s.draw();
```

```
}
```

```
}
```

Output:-

Drawing circle



Abstract class having constructor, data member and methods

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

Example 3:-

File: *TestAbstraction2.java*

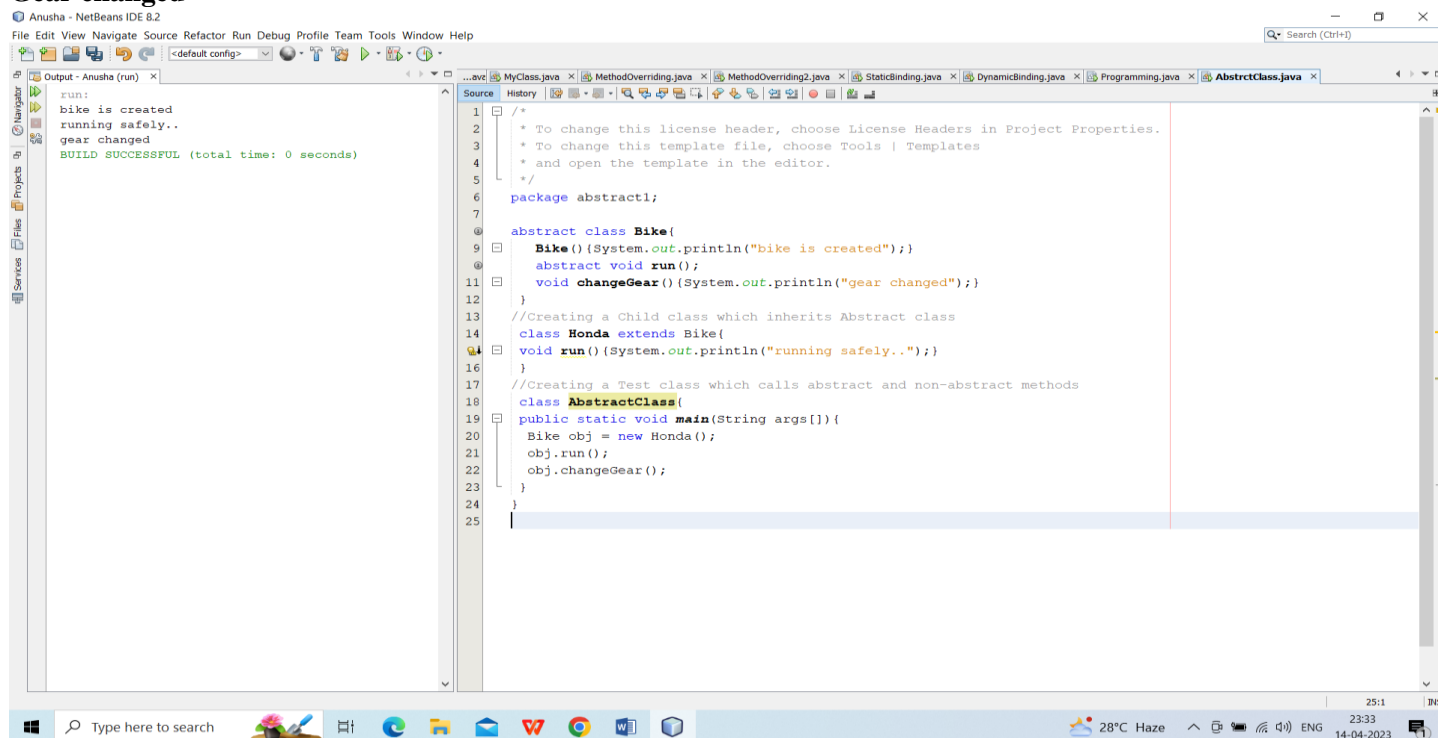
1. //Example of an abstract class that has abstract and non-abstract methods
2. **abstract class** Bike{
3. Bike(){System.out.println("bike is created");}
4. **abstract void** run();
5. **void** changeGear(){System.out.println("gear changed");}
6. }
7. //Creating a Child class which inherits Abstract class
8. **class** Honda **extends** Bike{
9. **void** run(){System.out.println("running safely..");}
10. }
11. //Creating a Test class which calls abstract and non-abstract methods
12. **class** TestAbstraction2{
13. **public static void** main(String args[]){
14. Bike obj = **new** Honda();
15. obj.run();
16. obj.changeGear();
17. }
18. }

OutPut:-

Bike is created

Running safely..

Gear changed



The screenshot shows the NetBeans IDE interface. On the left, the 'Output' window displays the following text: 'run: bike is created', 'running safely..', 'gear changed', and 'BUILD SUCCESSFUL (total time: 0 seconds)'. The main editor window shows the source code of a Java program. It defines an abstract class 'Bike' with methods 'run()' and 'changeGear()'. A subclass 'Honda' extends 'Bike' and implements these methods. A test class 'AbstractClass' contains a 'main' method that creates a 'Honda' object and calls its 'run()' and 'changeGear()' methods. The code is as follows:

```
1  /*
2  * To change this license header, choose License Headers in Project Properties.
3  * To change this template file, choose Tools | Templates
4  * and open the template in the editor.
5  */
6  package abstract1;
7
8  abstract class Bike{
9      Bike() {System.out.println("bike is created");}
10     abstract void run();
11     void changeGear() {System.out.println("gear changed");}
12 }
13 //Creating a Child class which inherits Abstract class
14 class Honda extends Bike{
15     void run() {System.out.println("running safely..");}
16 }
17 //Creating a Test class which calls abstract and non-abstract methods
18 class AbstractClass{
19     public static void main(String args[]) {
20         Bike obj = new Honda();
21         obj.run();
22         obj.changeGear();
23     }
24 }
25
```

Note1:-If there is an abstract method in a class, that class must be abstract.

Note2:If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

9.Abstract Method

A method declared using the **abstract** keyword within an abstract class and does not have a definition (implementation) is called an abstract method.

When we need just the method declaration in a super class, it can be achieved by declaring the methods as abstracts.

Abstract method is also called subclass responsibility as it doesn't have the implementation in the super class. Therefore a subclass must override it to provide the method definition.

Syntax for abstract method:

abstract return_type method_name([argument-list]);

Here, the abstract method doesn't have a method body. It may have zero or more arguments.

Steps to Create Abstract Method:-

- It can only be used in abstract class.
- It doesn't contain any body "{ }" and always ends with semicolon ";".
- Abstract method must be overridden in sub-classes otherwise it will also become a abstract class.
- Whenever the action is common but implementation are difficult then we should use abstract method.

Ex:-class fruit

{

abstract class fruit

{

```
public void test()
{
}
}
```

```
public abstract void test();
}
```

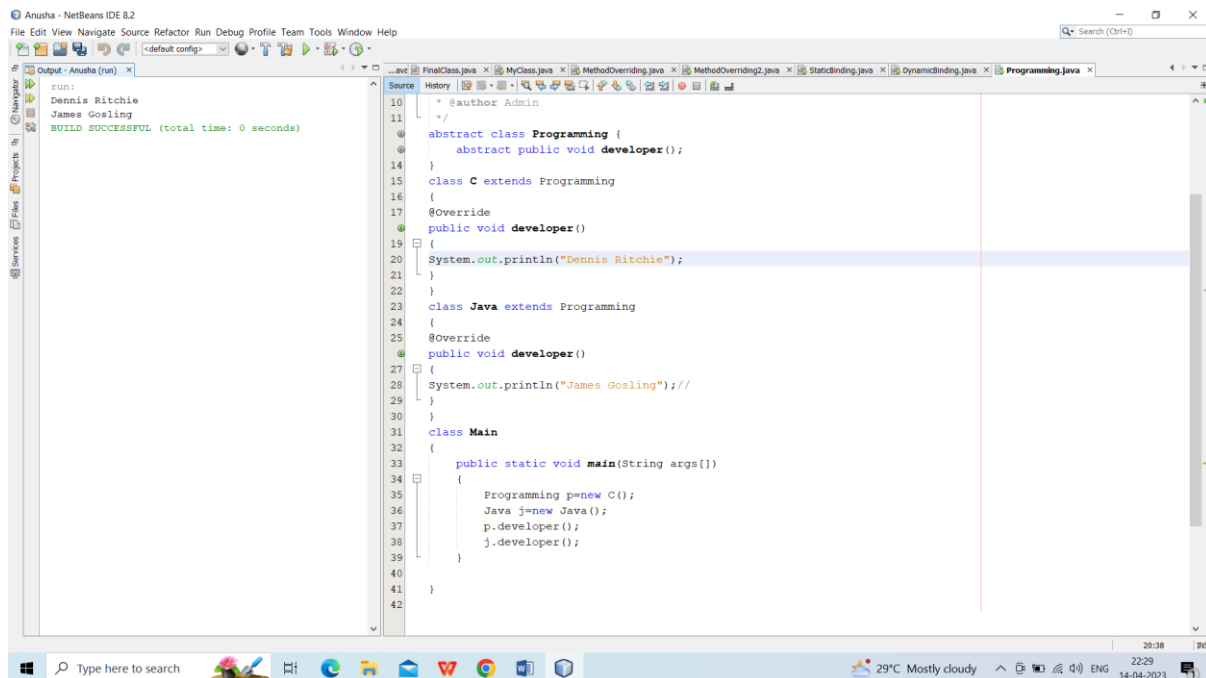
Example Program:-

```
abstract class Programming {
    abstract public void developer();
}
class C extends Programming
{
    @Override
    public void developer()
    {
        System.out.println("Dennis Ritchie");
    }
}
class Java extends Programming
{
    @Override
    public void developer()
    {
        System.out.println("James Gosling");//
    }
}
class Main
{
    public static void main(String args[])
    {
        Programming p=new C();
        Java j=new Java();
        p.developer();
        j.developer();
    }
}
```

Output:-

Dennie Ritchie

James Gosling



Difference between abstract class and interface

Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance .	Interface supports multiple inheritance .
3) Abstract class can have final, non-final, static and non-static variables .	Interface has only static and final variables .
4) Abstract class can provide the implementation of interface .	Interface can't provide the implementation of abstract class .
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
6) An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
7) An abstract class can be extended using keyword "extends".	An interface can be implemented using keyword "implements".
8) A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**

```
public abstract class Shape{  
public abstract void draw();  
}
```

Example:

```
public interface  
void  
}
```