UNIT 2

NHERITANCE, SUB CLASSING, PACKAGE

INHERITANCE, SUB CLASSING, PACKAGE

1. INHERITANCE CONCEPTS
2. DEFINING SUB CLASSES, METHOD OVERRIDING
3. USING SUPER KEYWORD
4. VARIABLE SHADOWING, METHOD AND VARIABLE BINDING
5. USING FINAL KEYWORD
6. ABSTRACT CLASSES AND INTERFACES
7. OBJECT CLASS, PACKAGES - CREATING PACKAGE
8. CLASSPATH ENVIRONMENT VARIABLE
9. ACCESS SPECIFIERS, ACCESS CONTROL / VISIBILITY.

INHERITANCE CONCEPTS

- Inheritance is a fundamental concept in object-oriented programming (OOP) that allows one class to inherit the fields and methods of another class. In Java, it provides a way to establish a relationship between classes and promote code reuse. Here's a breakdown of the key concepts related to inheritance in Java:

1. **Super Class (Parent Class)**: The class whose properties and methods are inherited by another class. It's also called the base class or parent class.
2. **Sub Class (Child Class)**: The class that inherits the properties and methods of another class. It's also known as the derived class or child class

- Syntax

```
class Parent {
    // Parent class code
}

class Child extends Parent {
    // Child class code
}
```

Why use inheritance in java

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

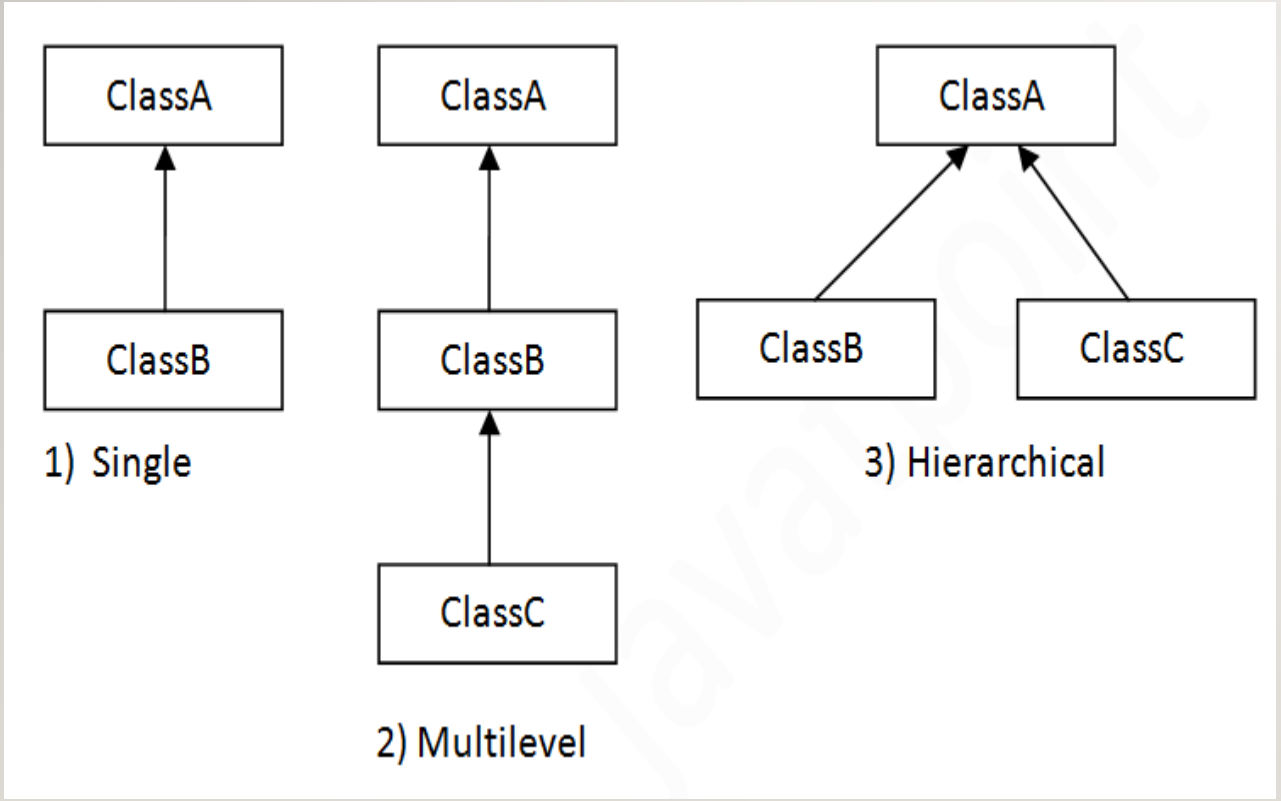
TERMS USED IN INHERITANCE

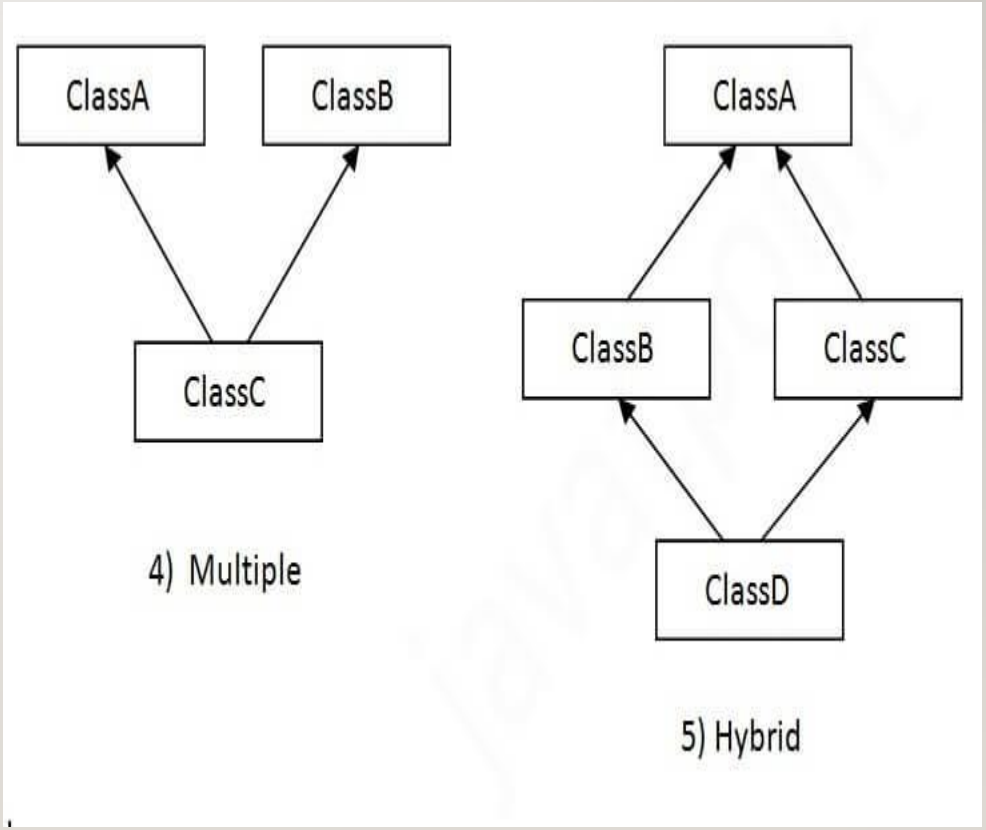
- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent 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.

TYPES OF INHERITANCE

- **Single Inheritance**: A class inherits from only one superclass.
- **Multiple Inheritance**: Java does not support multiple inheritance through classes directly. However, it can be achieved using interfaces.
- **Multilevel Inheritance**: A class is derived from another class, which is also derived from another class.
- **Hierarchical Inheritance**: Multiple classes inherit from a single superclass.
- **Hybrid Inheritance**: A combination of two or more types of inheritance. Java does not support hybrid inheritance through classes but supports it using interfaces
- Note: Multiple inheritance is not supported in Java through class.

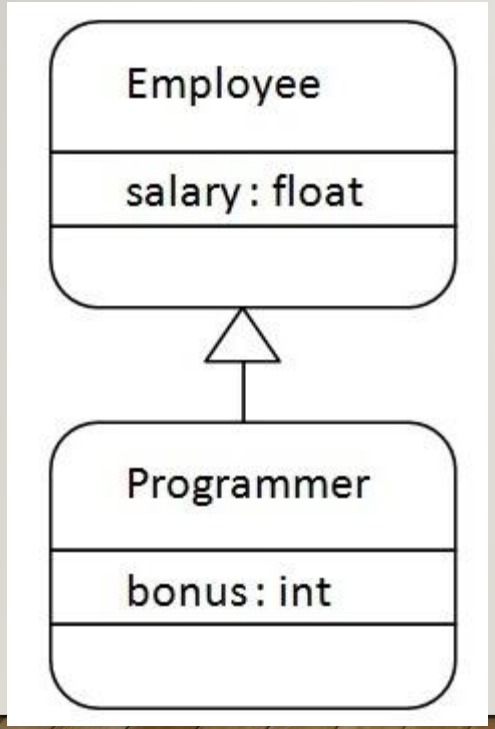
TYPE (CONT.)


ClassA
ClassB
1) Single
ClassA
ClassB
ClassC
2) Multilevel
ClassA
ClassB
ClassC
3) Hierarchical


ClassA
ClassB
ClassC
4) Multiple
ClassA
ClassB
ClassC
ClassD
5) Hybrid

EXAMPLE

- 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.




```
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);
}
}
```

SINGLE INHERITANCE EXAMPLE

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

MULTILEVEL INHERITANCE EXAMPLE

- When there is a chain of inheritance, it is known as *multilevel inheritance*.

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

HIERARCHICAL INHERITANCE EXAMPLE

- When two or more classes inherits a single class, it is known as *hierarchical inheritance*.

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

Q) Why multiple inheritance is not supported in java?

Q) WHY MULTIPLE INHERITANCE IS NOT SUPPORTED IN JAVA?

Answer

- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
- Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method

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

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

}
}
```

HYBRID INHERITANCE

- Hybrid inheritance refers to a combination of two or more types of inheritance (single, multiple, hierarchical, or multilevel).
- However, Java does not support multiple inheritance through classes directly to avoid the **Diamond Problem**, which occurs when two parent classes have the same method, and a child class inherits from both. This can cause ambiguity in which method to call.
- To achieve hybrid inheritance in Java, you can use **interfaces**. Interfaces allow multiple inheritance of behavior, meaning a class can implement multiple interfaces and combine them with other types of inheritance.

EXAMPLE

```
// Interface 1
interface Animal {
    void eat();
}

// Interface 2
interface Mammal {
    void giveBirth();
}

// Parent class
class LivingBeing {
    void breathe() {
```

```
// Child class
class Dog extends LivingBeing implements Animal, Mammal {
    // Implementing methods from interfaces
    public void eat() {
        System.out.println("Dog is eating.");
    }

    public void giveBirth() {
        System.out.println("Dog gives birth to puppies.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.breathe();     // Inherited from LivingBeing
        dog.eat();         // Implemented from Animal interface
        dog.giveBirth();   // Implemented from Mammal interface
    }
}
```

POLYMORPHISM IN JAVA

- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*.
- There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

METHOD OVERLOADING IN JAVA

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.
- Note In Java, Method Overloading is not possible by changing the return type of the method only.

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

- There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1) METHOD OVERLOADING: CHANGING NO. OF ARGUMENTS

- we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

2) METHOD OVERLOADING: CHANGING DATA TYPE OF ARGUMENTS

- we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

Q) Why Method Overloading is not possible by changing the return type of method only?

Now

Q) Why Method Overloading is not possible by changing the return type of method only?

- In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

```
class Adder{
static int add(int a,int b){return a+b;}
static double add(int a,int b){return a+b;}
}
class TestOverloading3{
public static void main(String[] args){
System.out.println(Adder.add(11,11));//ambiguity

}}
```

Can we overload java main() method?

Can we overload java main() method?

- Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

```
class TestOverloading4{
public static void main(String[] args)
{System.out.println("main with String[]");}
public static void main(String args)
{System.out.println("main with String");}
public static void main()
{System.out.println("main without args");}
}
```

o/p

```
main with String[]
```

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**.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

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).

UNDERSTANDING THE PROBLEM WITHOUT METHOD OVERRIDING

Parul® University

- Let's understand the problem that we may face in the program if we don't use method overriding

```
//Java Program to demonstrate why we need meth
od overriding
//Here, we are calling the method of parent class wi
th child
//class object.
//Creating a parent class
class Vehicle{
  void run(){System.out.println("Vehicle is running");
}
}
//Creating a child class
class Bike extends Vehicle{
  public static void main(String args[]){
  //creating an instance of child class
  Bike obj = new Bike();
  //calling the method with child class instance
  obj.run();
   }
}
```

Problem is that I have to provide a spe
implementation of run() method in su
that is why we use method overriding

EXAMPLE OF METHOD OVERRIDING

```
//Java Program to illustrate the use of Jav
a 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
  }
}
```

QUESTIONS

- **Can we override static method?**
- **Why can we not override static method?**
- **Can we override java main method?**

QUESTIONS

- **Can we override static method?**

No, a static method cannot be overridden. It can be proved by runtime polymorphism

- **Why can we not override static method?**

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.

- **Can we override java main method?**

No, because the main is a static method.

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.

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

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.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```

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:

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

SHADOWING IN JAVA

- **Shadowing** is the concept of OOP paradigm. It provides a new implementation to the base member without overriding it. Both shadowing and hiding are the same concepts but uses in different context. These are compile-time process.
- **variable shadowing** occurs when a variable declared in a local scope (e.g., a method or block) hides or "shadows" a variable with the same name declared in an outer scope (such as a class field or an inherited field). This means that the local variable "shadows" the instance variable or field, making it inaccessible in that scope.

KEY POINTS:

- Shadowing happens when a local variable (e.g., in a method or constructor) has the same name as a class-level field.
- In the local scope, the local variable will take precedence over the class-level field

You can access the shadowed class-level field using the **this** keyword

VARIABLE SHADOWING AND VARIABLE HIDING

Java allows us to declare three type of variables:

- Local Variables
- Instance Variables
- Class Variables
- If the instance variable and the local variable declared with the same name and we want to print the instance variable, in this case, it will print the local variable instead of instance variable.
- When we print the variables declared inside the method, the local values will be printed on the console. Hence, we can say that local variables shadowing the instance variables.

LET'S UNDERSTAND THE CONCEPT OF SHADOWING THROUGH A JAVA PROGRAM.

```
public class Shadowing
{
String car_name = "Ferrari;

double price = 50000000;

public void showCar()
{
String car_name = "Bugat;

long price = 43000000;
System.out.println("Car Name: "+ca
r_name);
System.out.println("Price: "+price);
}
public static void main(String arg
s[])
{
new Shadowing().showCar();
}
1.}
```

Car Name: Bugatti Price: 43000000

EXAMPLE

What if want to access instance variables in a method also? We can access instance variables by using this keyword.

```
public class Shadowing
{
String car_name = "Ferrari";
long price = 50000000;
public void showCar()
{
String car_name = "Bugatti";
long price = 43000000;
System.out.println("Car Name: "+car_name);
System.out.println("Price: "+price);
System.out.println("Car Name: "+this.car_name);
System.out.println("Price: "+this.price);
}
public static void main(String args[])
{
new Shadowing().showCar();
}
}
```

Car Name: Bugatti Price: 43000000 Car Name: Ferrari Price: 50000000

EXAMPLE OF SHADOWING IN JAVA:

```
class ShadowingExample {
    int x = 10;  // Class-level field (instance variable)
    void display(int x) {
        // The parameter 'x' shadows the instance variable 'x'
        System.out.println("Local x: " + x);  // Refers to the method parameter 'x'
        System.out.println("Instance x: " + this.x);  // Refers to the class field 'x'
    }
        public static void main(String[] args) {
            ShadowingExample obj = new ShadowingExample();
            obj.display(20);  // Calling display() with local variable value 20
        }
    }
```

o/p
Local x: 20
Instance x: 10

METHOD AND VARIABLE BINDING IN JAVA

- In Java, method and variable binding refer to the mechanisms by which methods and variables are associated with their respective objects or classes during program execution. This concept is crucial in understanding how Java handles polymorphism, inheritance, and variable shadowing

1. Method Binding

- Method binding in Java occurs in two main ways: **static (or early) binding** and **dynamic (or late) binding**.

a. Static Binding

- **Definition**: Static binding occurs at compile-time. The compiler determines which method to invoke based on the reference type.
- **Usage**: It is used with:
  - **Static methods**: Because static methods belong to the class rather than any instance.
  - **Final methods**: Since they cannot be overridden.
  - **Private methods**: Since they are not accessible outside the class.

EXAMPLE

```
class Animal {
    static void sound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    static void sound() {
        System.out.println("Bark");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Reference type is Animal, object is Dog
        animal.sound(); // Outputs: Animal sound (static binding)
    }
}
```

o/p Animal sound

B. DYNAMIC BINDING

- **Definition**: Dynamic binding occurs at runtime. The method to be invoked is determined based on the actual object being referenced, not the reference type.
- **Usage**: It is used with instance methods that can be overridden.

B. DYNAMIC BINDING

```
class Animal {
    void sound() {
        System.out.println("Animal sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Bark");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Reference type is Animal, object is Dog
        animal.sound(); // Dynamic binding occurs
    }
}
```

o/p Bark

2. VARIABLE BINDING

- Variable binding refers to how variables are associated with their values and scopes. In Java, this is closely related to the concept of shadowing.
- **Shadowing**
- **Definition**:Variable shadowing occurs when a variable in a subclass or a method has the same name as a variable in a superclass or an enclosing scope. The variable in the inner scope "shadows" the one in the outer scope.

EXAMPLE

```
class Parent {
    int x = 10; // Parent class variable
}

class Child extends Parent {
    int x = 20; // Child class variable (shadows parent's variable)
    void display() {
        System.out.println("Child's x: " + x);      // Accesses child's variable (x = 20)
        System.out.println("Parent's x: " + super.x); // Accesses parent's variable (x = 10)
    }
}

public class Main {
    public static void main(String[] args) {
        Child child = new Child();
        child.display();
    }
}
```

```
Child's x: 20
Parent's x: 10
```

FINAL KEYWORD IN JAVA

Parul® University

- 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

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

Output:Compile Time Error

2) JAVA FINAL METHOD

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

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

Output:Compile Time Error

3) JAVA FINAL CLASS

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

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

Output:Compile Time Error

Q) Is final method inherited?

Q) IS FINAL METHOD INHERITED?

- Que) Can we initialize blank final variable?

```
class Bike{
  final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
   public static void main(String args[]){
    new Honda2().run();
   }
}
```

Que) Can we initialize blank final variable?

Output:running...

QUE) CAN WE INITIALIZE BLANK FINAL VARIABLE?

- Yes, but only in constructor. For example:

```
class Bike10{
 final int speedlimit;//blank final variable

 Bike10(){
 speedlimit=70;
 }

 public static void main(String args[]){
   new Bike10();
}

 System.out.println(speedlimit);  }
```

STATIC BLANK FINAL VARIABLE

- A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

```
class A{
  static final int data;//static blank final variable
  static{ data=50;}
  public static void main(String args[]){
    System.out.println(A.data);
 }
}
```

ABSTRACT CLASSES

- A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Abstraction in Java

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

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

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

ABSTRACT CLASS IN JAVA

- A class 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.

Points to Remember

- 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.

Rules for Java Abstract class

1. An abstract class must be declared with an abstract keyword.
2. It can have abstract and non-abstract methods.
3. It cannot be instantiated.
4. It can have final methods
5. It can have constructors and static methods also.

ABSTRACT METHOD IN JAVA

- A method which is declared as abstract and does not have implementation is known as an abstract method.
- **Example of abstract method**

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

EXAMPLE OF ABSTRACT CLASS THAT HAS AN ABSTRACT METHOD

- 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();
}
}
```

running safely

Understanding the real scenario of Abstract class

- In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.
- Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.

```
abstract class Shape{
abstract void draw();
}
//In real scenario, implementation is provided by ot
hers 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 p
//rovided through method, e.g., getShape() method
s.draw();
}
}
```

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.

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

```
//Example of an abstract class that has abstract and non-
//abstract methods
 abstract class Bike{
   Bike(){System.out.println("bike is created");}
   abstract void run();
   void changeGear(){System.out.println("gear changed");}
 }
//Creating a Child class which inherits Abstract class
 class Honda extends Bike{
 void run(){System.out.println("running safely..");}
 }
//Creating a Test class which calls abstract and non-abstract methods
 class TestAbstraction2{
 public static void main(String args[]){
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
 }
}
```

bike is created running safely.. gear changed

Interface in Java

Parul® University

- An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
- The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.
- Java Interface also **represents the IS-A relationship**.

Why use Java interface?

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

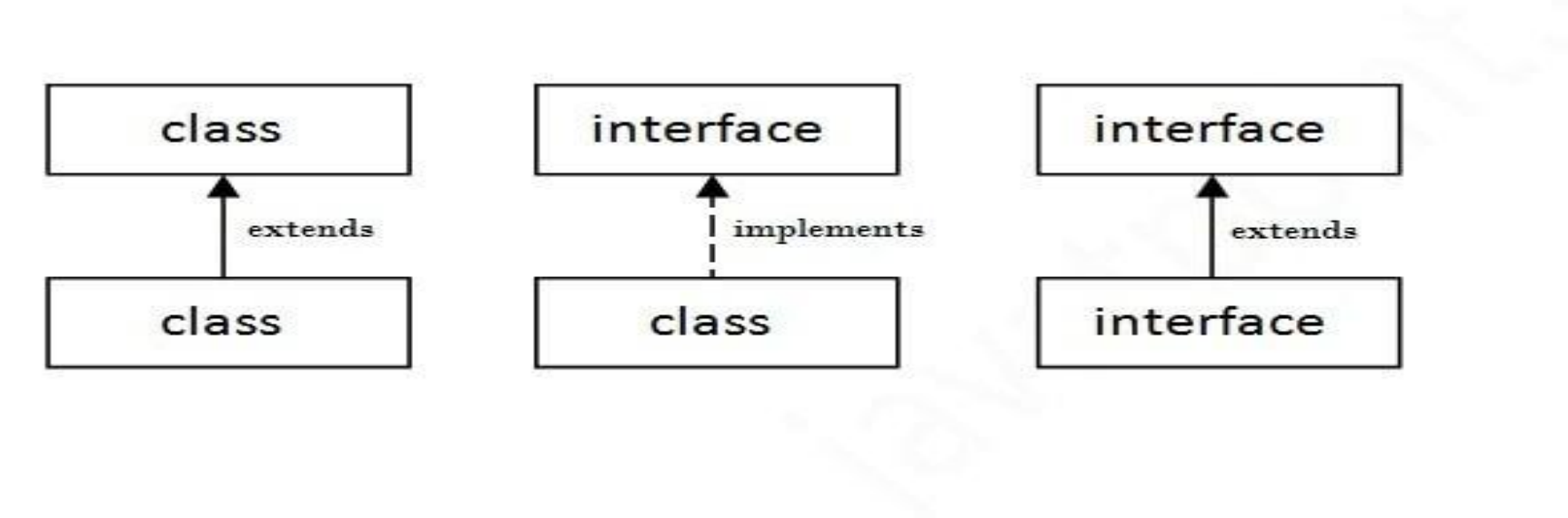
How to declare an interface?

- An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.
- Syntax

```
interface <interface_name>{
    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Java Interface Example

- In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

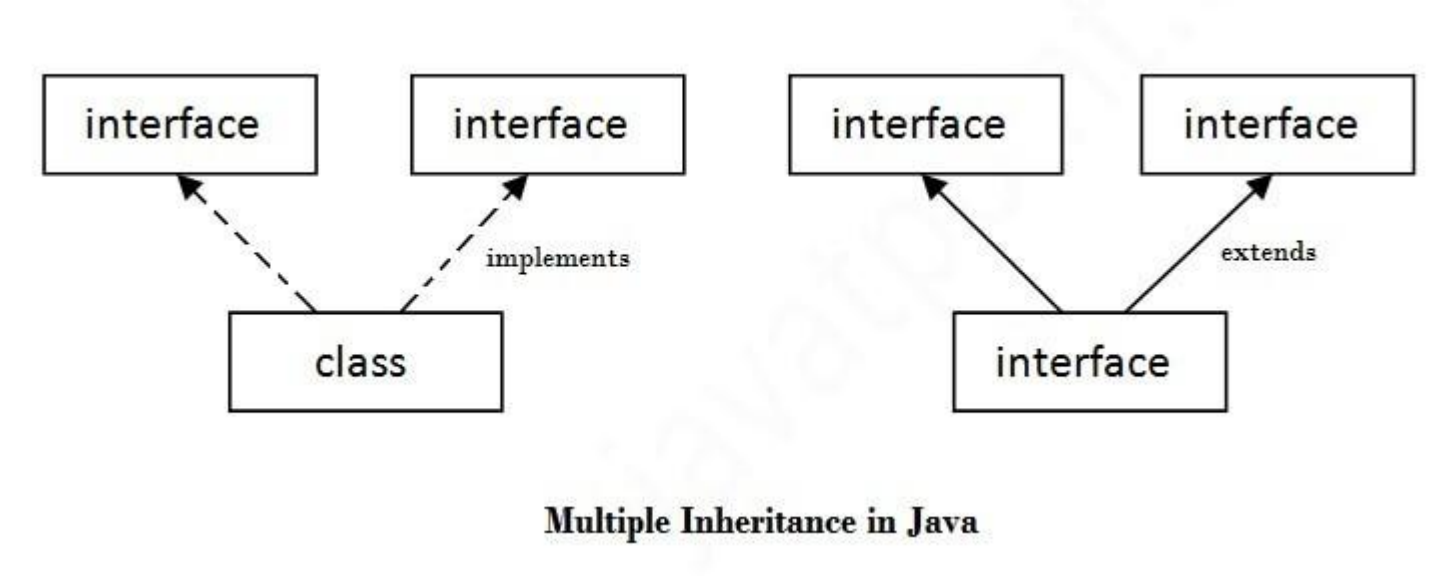
```
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```

Hello

Multiple inheritance in Java by interface

- If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

EXAMPLE

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

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

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

Output:Hello Welcome

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

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

- As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class.

Interface inheritance

- A class implements an interface, but one interface extends another interface.

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
```

```
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
 }
}
```

Java 8 Default Method in Interface

- Since Java 8, we can have method body in interface. But we need to make it default method

```
interface Drawable{
void draw();
default void msg(){System.out.println("default method");}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
```

```
class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
d.msg();
}}
```

drawing rectangle default method

Java 8 Static Method in Interface

- Since Java 8, we can have static method in interface.

```
interface Drawable{
void draw();
static int cube(int x){return x*x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rec
tangle");}
}

class TestInterfaceStatic{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable.cube(3));
}}
```

Encapsulation in Java

- **Encapsulation in Java** is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.
- We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

Advantage of Encapsulation in Java

By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods.

It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.

CONT.

- It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.
- The encapsulate class is **easy to test**. So, it is better for unit testing.
- The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

SIMPLE EXAMPLE OF ENCAPSULATION IN JAVA

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

```
//A Java class which is a fully encapsulated class.
//It has a private data member and getter and setter methods.
package com.javatpoint;
public class Student{
//private data member
private String name;
//getter method for name
public String getName(){
return name;
}
//setter method for name
public void setName(String name){
this.name=name
}
}
```

```
//A Java class to test the encapsulated class.
package com.javatpoint;
class Test{
public static void main(String[] args){
//creating instance of the encapsulated class
Student s=new Student();
//setting value in the name member
s.setName("vijay");
//getting value of the name member
System.out.println(s.getName());
}
}
```

Compile By: javac -d . Test.java Run By: java com.javatpoint.Test

vijay

READ-ONLY CLASS

```
//A Java class which has only getter methods.
public class Student{
//private data member
private String college="AKG";
//getter method for college
public String getCollege(){
return college;
}
}
```

- Now, you can't change the value of the college data member which is "AKG".
- s.setCollege("KITE");//will render compile time error

WRITE-ONLY CLASS

```
//A Java class which has only setter methods.
public class Student{
//private data member
private String college;
//getter method for college
public void setCollege(String college){
this.college=college;
}
}
```

```
1.System.out.println(s.getCollege());//Compile Time
Error, because there is no such method
2.System.out.println(s.college);//Compile Time Error,
 because the college data member is private.
3.//So, it can't be accessed from outside the class
```

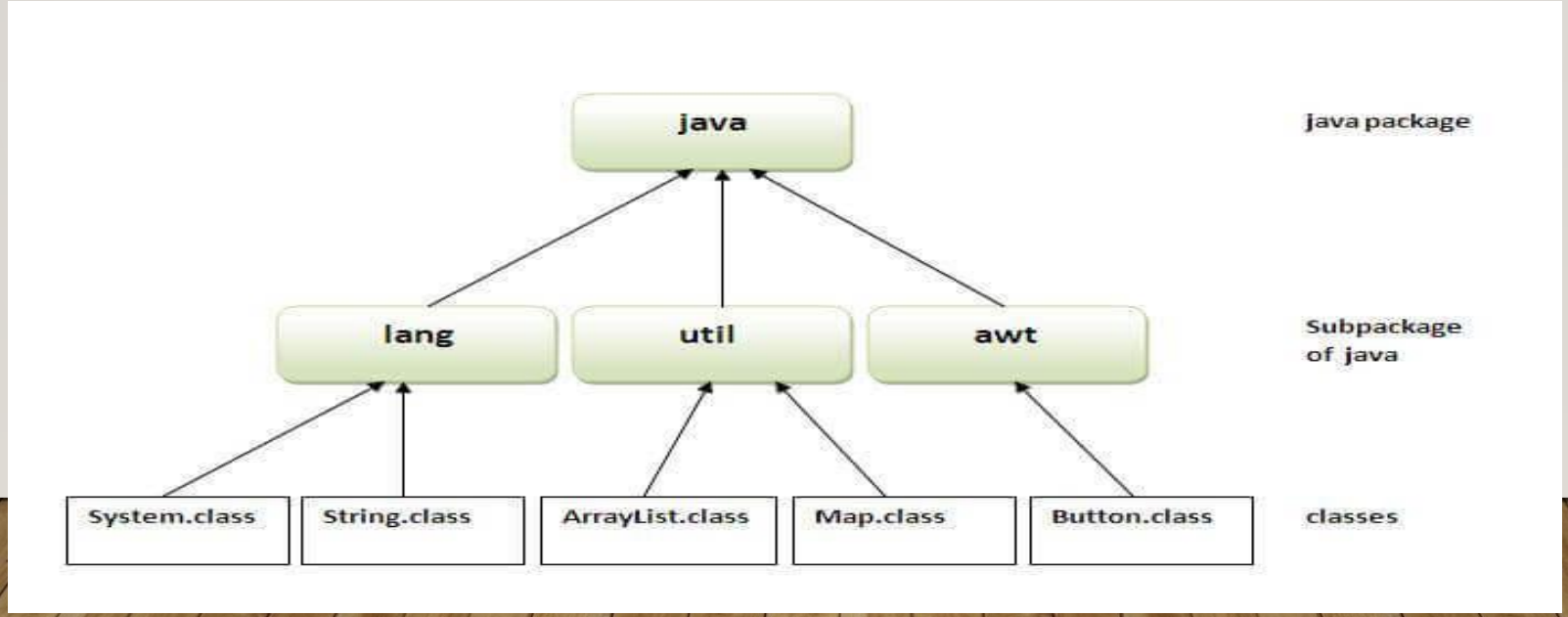
- Now, you can't get the value of the college, you can only change the value of college data member.

Java Package

- A **java package** is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form, built-in package and user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc

Advantage of Java Package

- 1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- 2) Java package provides access protection.
- 3) Java package removes naming collision.



Simple example of java package

Parul® University

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

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

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

javac -d directory javafilename

For example
javac -d . Simple.java

HOW TO RUN JAVA PACKAGE PROGRAM

- You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java
To Run: java mypack.Simple

Output:Welcome to package

HOW TO ACCESS PACKAGE FROM ANOTHER PACKAGE?

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

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

1) Using packagename.*

- If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//save by A.java

package pack;

public class A{

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

}
```

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

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

Output:Hello

2) Using packagename.classname

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

```
//save by A.java

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

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

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

Output:Hello

3) Using fully qualified name

- If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
- It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java

package pack;

public class A{

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

}
```

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

Output:Hello

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

What is CLASSPATH in Java?

The **CLASSPATH** environment variable is used by the Java Virtual Machine (JVM) and the Java compiler (javac) to determine the location of user-defined classes, libraries, and packages. It helps Java programs find and load the required .class files and libraries (JAR files) that are not part of the standard Java runtime.

Key Concepts of CLASSPATH:

- **Directories**: Can contain compiled .class files.
- **JAR files**: External libraries bundled as Java ARchive (JAR) files.
- **Default Value**: If the CLASSPATH is not set, Java looks in the current directory (.) by default.
- **Setting CLASSPATH**: Can be done through environment variables or overridden at runtime using the -cp or -classpath option.

How to Set CLASSPATH?

You can set CLASSPATH in two ways:

1. **As an Environment Variable**: Permanently or temporarily set in the operating system.
2. **At runtime**: Override the CLASSPATH value with the -cp or -classpath flag during program execution.

Example Scenario

- Assume we have the following project structure:

/project

/lib

utility-library.jar

/src

Main.java

/classes

Main.class

•utility-library.jar: External library file located in the lib directory.
•Main.java: Your Java program located in the src directory.
•Main.class: Compiled class file of Main.java located in the classes directory.

Step-by-Step Example

1. Create and Compile a Simple Java Program

Let's create a basic Java program (Main.java):

```
// File: Main.java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello,
CLASSPATH!");
    }
}
```

Navigate to the src directory and compile the program:

```
javac -d ../classes Main.java
```

This command will place the Main.class file in the classes directory.

2. Setting CLASSPATH

Now, we need to set the CLASSPATH so that Java knows where to find the Main.class and any external libraries.

- **On Windows**:
- set CLASSPATH=.;C:\project\classes;C:\project\lib\utility-library.jar;

Here's what we're doing:

- .: Refers to the current directory.
- C:\project\classes: Path to the directory where Main.class is stored.
- C:\project\lib\utility-library.jar: Path to the external JAR file.

3. Running the Program

Once CLASSPATH is set, you can run the Java program using the java command:

```
java Main
```

Output Hello, CLASSPATH!

4. Overriding CLASSPATH at Runtime

You can also override the CLASSPATH temporarily at runtime using the -cp or -classpath flag:
java -cp .;C:\project\classes;C:\project\lib\utility-library.jar Main
This command overrides the globally set CLASSPATH and tells Java where to look for class files and libraries during program execution.

Common Errors Related to CLASSPATH:

- **ClassNotFoundException**: If Java cannot find the class you're trying to run.
- **NoClassDefFoundError**: If the class was available during compile time but is not found during runtime.

Access Modifiers in Java

Parul® University

- There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.
- 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:

- **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
- **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
- **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
- **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Understanding Java Access Modifiers

Parul® University

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

1) Private

- The private access modifier is accessible only within the 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 ja
va");}
}
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

- 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("Hell
o");}
}//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 package and outside the package but through inheritance only.
- 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 modifer.

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 arg
s[]){
   B obj = new B();
   obj.msg();
  }
}
```

Output:Hello

4) Public

- The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.
- **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");}
}


public class Simple extends A{
void msg(){System.out.println("Hello java");}//C.T.Error
 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.

Parul® University

Thank you