OOPS

Tags	
Status	In progress

Classes

- class is like a blue print
- classes Named group of properities and function
- classes is used combined properties and Methods
- classes and object are important object oriented programming oriented Pillars are Encaplusation, inheritance, Abstraction, Ploymorphism
- Inheritance ⇒ Inherit the value form one class to another classes
- Polymorphism ⇒ solving the problem of Inheritance and polymorphism work for constructor value.ploymorphism,are Operator overloading, Function overloading, Overriding
- Abstraction focuses on the external level of implementation, whereas Encapsulation concentrates on the internal level of implementation.
- Abstraction involves gaining information, while Encapsulation involves containing information at a certain level.

Why need Class

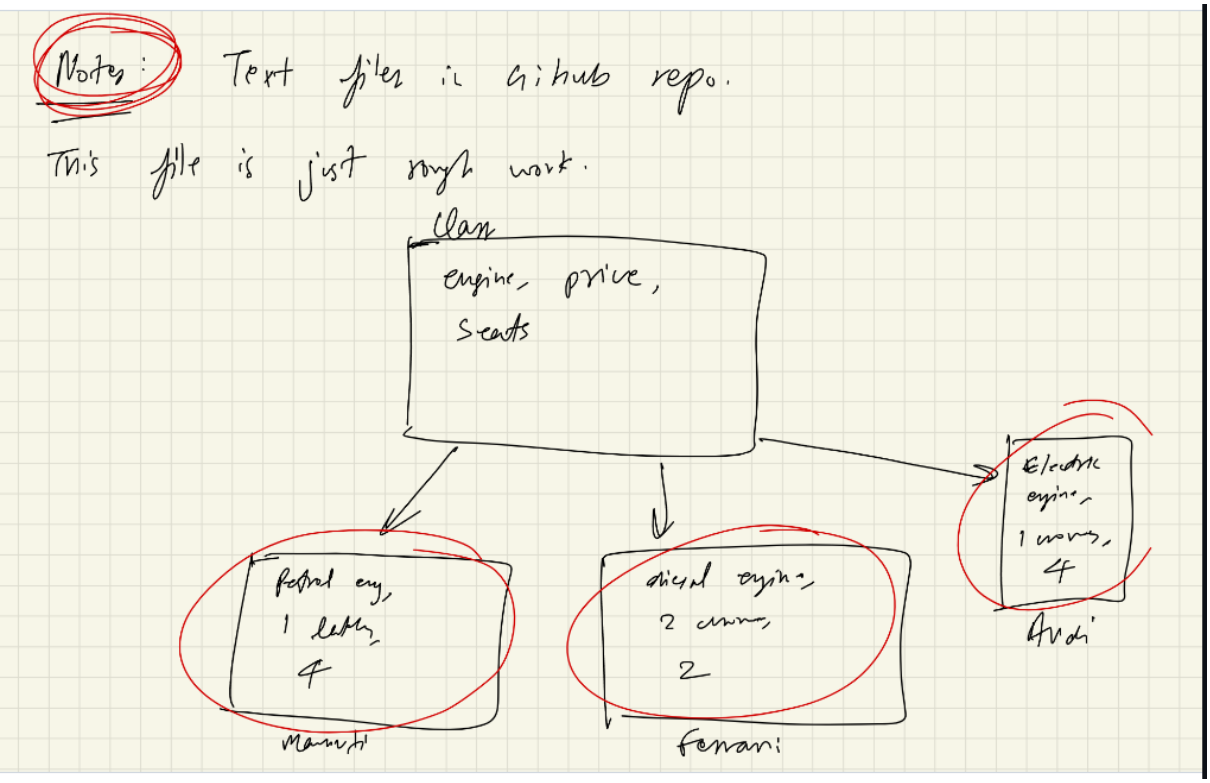
Consider a scenario where we need to manage information about students, such as their roll numbers and names. Initially, we might resort to separate arrays for each piece of data:

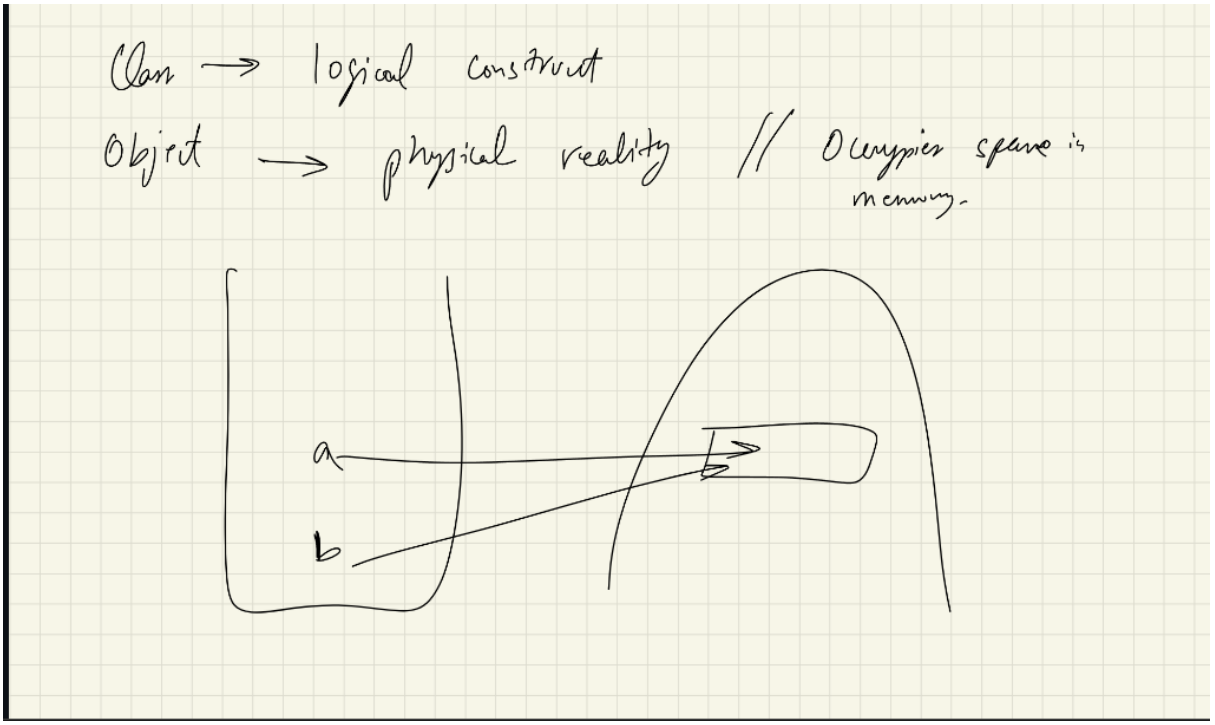
```
javaCopy code
int[] rollNumbers = new int[5]; // Array for storing roll n
umbers
String[] names = new String[5]; // Array for storing studen
```

```
t names
```

While this approach works, it can quickly become unwieldy as the program grows. Every additional piece of data necessitates another array, leading to increased memory consumption and complex data management. Classes offer a more elegant solution by consolidating related data and behavior into a single entity.

Classes





- Classes is created a own purpose contains own **Methods and properties.**
- Class is a template of an Object
- Object is an instance of classes

> Classes ⇒ Logical Value // Not Occpuing Space in Memory
> object ⇒ physical object // Occuping space in memory

Objects

- Object Occuping space in Memory Handle
- Object Access using . dot operator using Reference Variable
- Instance Variable is Object inside the properties
- It is useful to think of an object's identity as the place where its value is stored in memory.
- The behavior of an object is the effect of data-type operations.

- The dot operator links the name of the object with the name of an instance variable. Although commonly referred to as the dot operator, the formal specification for Java categorizes the . as a separator.

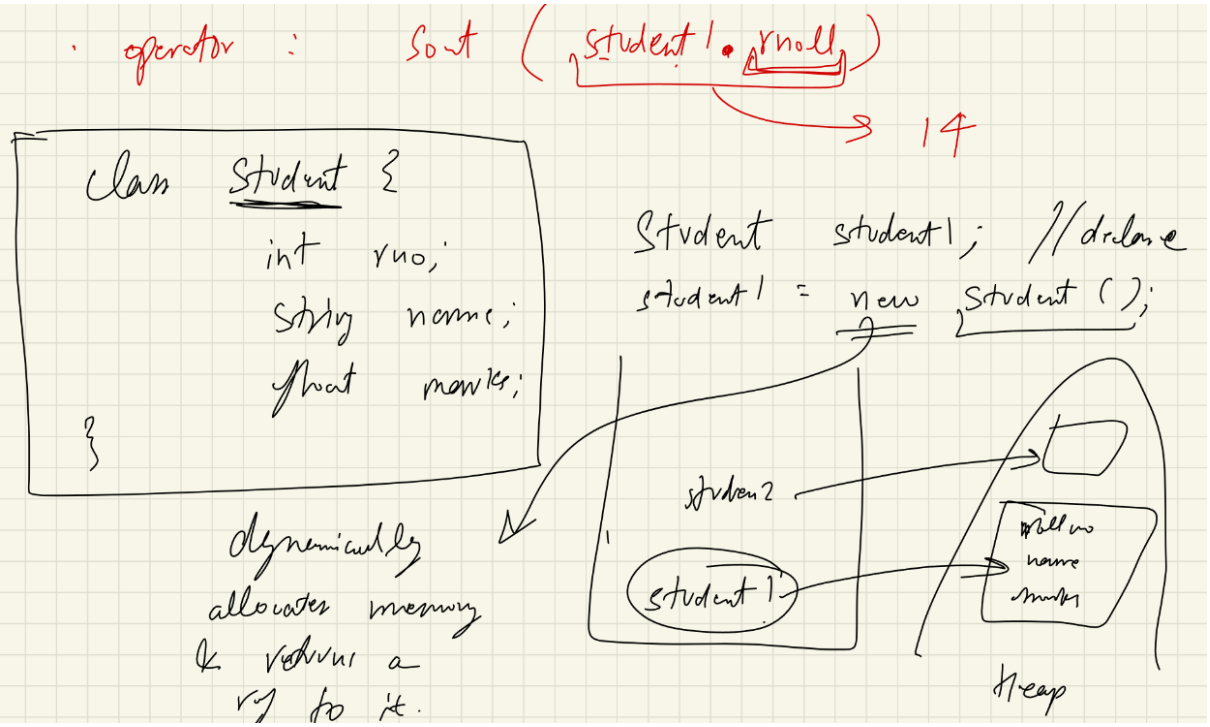
Class and Object declare

Declaring a Class

```
// create a class

Class Student {

int rno;   //rno;
int marks; //Marks
String name;  // Name
```

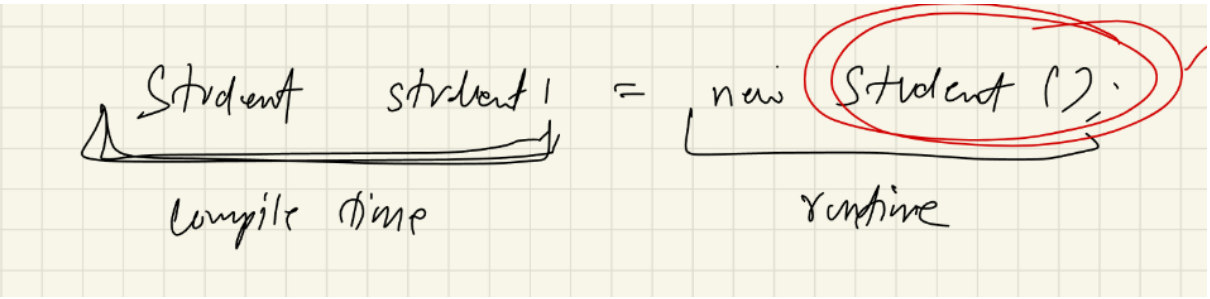
Declaring Object

```
Student student1 = new Student();
```



- New Word is create a Dynamically allocate memory & Return a reference to it

Runtime and compile Time



Complie Time

- Compile time refers to the period during which a program is translated from human-readable source code into machine-readable instructions by a compiler.

- Compile time Excute code Only initialize stack Memory

Run Time

- During runtime, variables are allocated memory, objects are created, and functions/methods are executed based on the flow of the program.
- Java programming Dynamically create object in Run time.

Manuplate the object and classes

```
public class Oops{

      static class Student{

        int rno;
        float  marks;
        String name;
      }
      public static void main(String[] args) {
          Student student1 = new Student();

          System.out.println(student1.rno);
          System.out.println(student1.marks);
          System.out.println(student1.name);
      }
}
```

Constructor

- Constructor is special type of function But Run you create a object and allows the same

  data type variables

- Constructor is same name of Class
- Immediately called object is created

- Constructor initializing the Values
- Constructor No return Type
- Constructor Value No modified.
- The constructor is a useful special type of function responsible for handling object properties' values and ensuring that they cannot be modified from outside the class.
- Must use this keyword

Advantage of constructor

1. **Initialization**: Constructors initialize object attributes, ensuring a defined starting state.
2. **Encapsulation**: They enforce encapsulation by controlling attribute initialization.
3. **Convenience**: Simplify object creation by providing a standardized process.
4. **Default Values**: Offer default values for attributes, enhancing flexibility.
5. **Inheritance and Polymorphism**: Play a key role in inheritance and polymorphism, enabling subclass object creation and specialization.

Types of Constructor

- Default Constructor
- Parameterized Constructor
- Copy Constructor

this Keyword

- The `this` keyword in Java refers to the current object.

- It is typically used to distinguish between instance variables and local variables, to invoke methods or constructors of the current object, or to pass the current object as an argument.
- However, it cannot be used in static methods.
- Understanding its usage is crucial for maintaining clarity and avoiding ambiguity in Java code.

```
public class Oops{

      static class Student{

        int rno;
        float   marks;
        String name;

        Student () {
        // Default values
        this.rno = 10;        // this word reference pointing
        this.marks = 20.0f;
        this.name = "gokil";
        }


    }

      public static void main(String[] args) {
        Student student1 = new Student();
        Student Rahul = new Student();
          System.out.println(student1.rno);
          System.out.println(student1.marks);
          System.out.println(student1.name);
      }
}
```

constructor Value modified

- Constructor only modified passing arugments
- Constructor using Modified Every Object Values using the parameters /Arugments values

```
public class Oops {

    static class Student {

        int rno;
        float marks;
        String name;

        // Constructor with parameters to modify values
        public Student(int rno, float marks, String name) {
            this.rno = rno;
            this.marks = marks;
            this.name = name;
        }
    }

    public static void main(String[] args) {
        // Creating a Student object with specific values usi
        Student student1 = new Student(10, 20.0f, "gokil");

        // Outputting the values of student1
        System.out.println(student1.rno);
        System.out.println(student1.marks);
        System.out.println(student1.name);
    }
}
```

Constructor Overloading

- Constructor overloading is indeed an important concept in polymorphism.

- In object-oriented principles, constructors do indeed consider parameters/arguments (values).
- A constructor with no arguments is called a default constructor.

```
public class Oops {

    public static class Student{

        int rno;
        float marks;
        String name;

    Student() {   //  constructor consider calling the size o

        this.rno=10;
        this.marks=20.0f;
        this.name="gokil";
    }
    Student(int rollno,float marks,String Name) // constructo
    {
        this.rno=rollno;
        this.marks=marks;
        this.name=Name;
    }
}
    public static void main(String[] args) {
         Student gokil = new Student(20,10.0f,"Rahul");

        Student Rahul = new Student();
        System.out.println(Rahul.marks);

    }


}
```

calling construct to another constructor

- call one constructor from another constructor in the same class using the `this()` keyword.
- This is useful when you have multiple constructors in a class and want to avoid code duplication by reusing common initialization logic.

1. `this()` **Keyword**:
   - Use `this()` to call another constructor from within a constructor.
   - It must be the first statement in the constructor.
2. **Avoiding Redundancy**:
   - Helps in avoiding repetition of initialization logic.
   - Centralizes common initialization steps.
3. **Parameter Passing**:
   - Arguments can be passed to the constructor being called using `this()`.
   - Ensure argument types match the parameter list of the target constructor.
4. **Code Readability**:
   - Enhances code readability by providing a clear flow of initialization logic.

```
public class Oops{

 public static class Student{

    int rno;
    String name;

    Student(int rollno,String Name)
    {
      rno =rollno;
```

```
      name=Name;
    }
    Student()
    {
        this(12,"gokil");
    }
 }
   public static void main(String [] args)
   {
      Student cs = new Student();

      System.out.println(cs.name);

   }


}
```

Copy Constructor

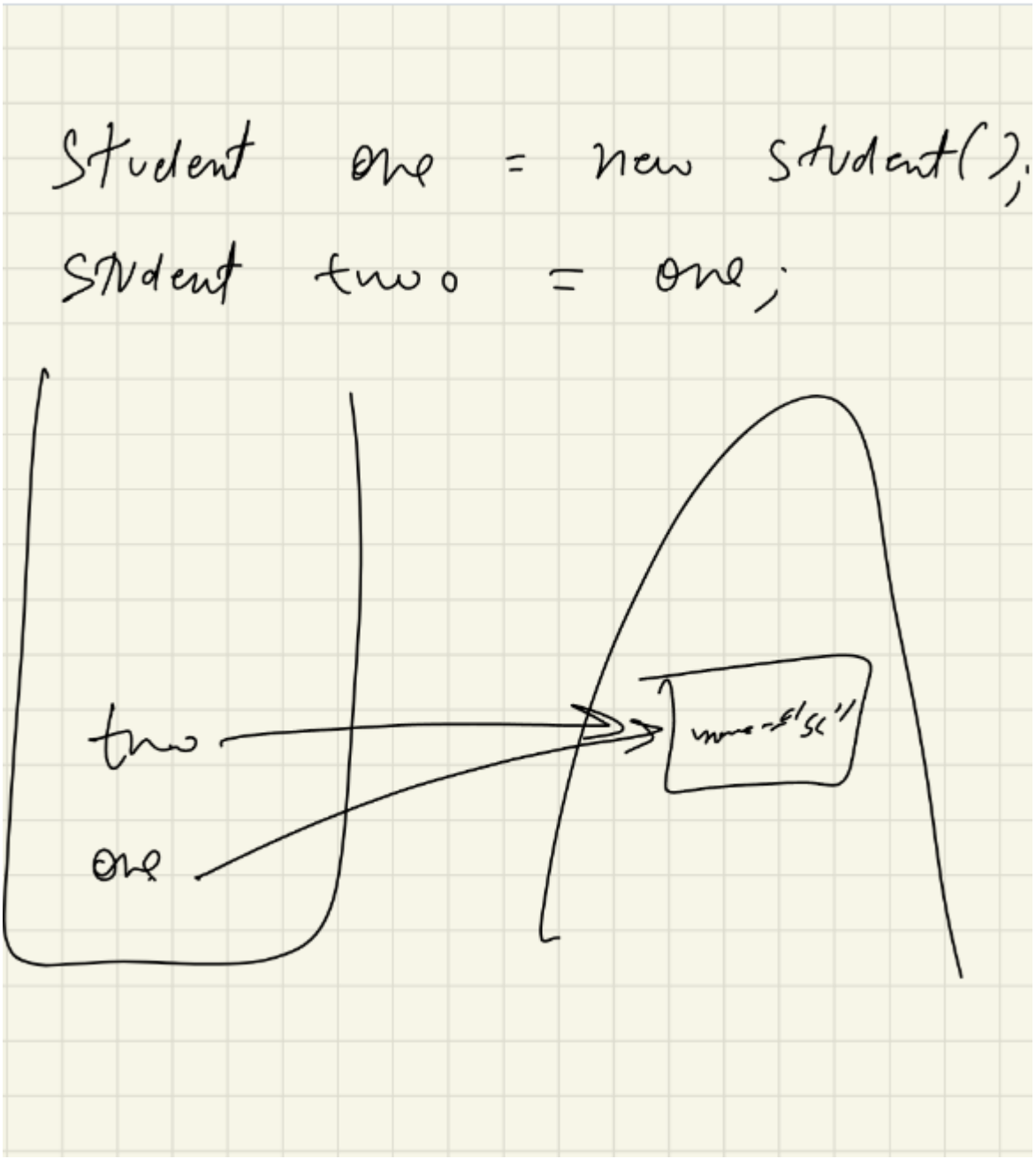
- java copy constructor returns a copy of the specified object by taking the existing object as an argument.
- To create a copy constructor, we need to take the existing object as an argument and initialize the values of instance variables with the values obtained in the object.

```
class Person {
    String name;

    // Copy constructor
    public Person(Person other) {
        this.name = other.name;
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        Person person1 = new Person();
        person1.name = "John";

        // Using the copy constructor to create a new object
        Person person2 = new Person(person1);

        System.out.println("Person 1: " + person1.name);
        System.out.println("Person 2: " + person2.name);
    }
}
```

Memory Allociation New Key Word

- New word also an Objects and stored in Heap Memory
- Heap mermory Object point to another Reference Variable



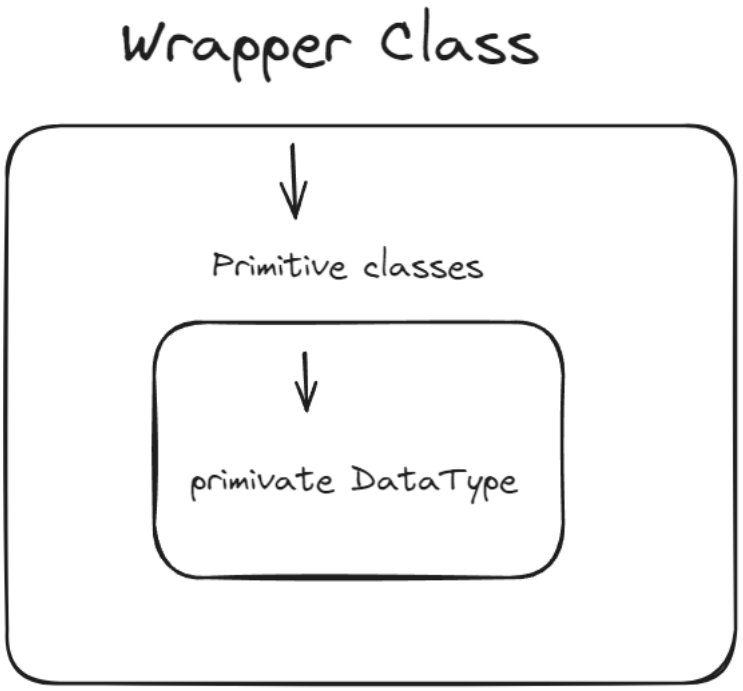
Wrapper Classes

- wrapper classes are used to convert primitive data types into objects, as Java is an object-oriented programming language where everything is treated as an object.
- Wrapper classes is immutable

- wrapper classes in scenarios where objects are required, such as collections (ArrayList, HashMap) which can't directly store primitive types.

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

- Wrapper classes contains internal side



Arritubutes of Wrapper Classes

```
Integer i = Integer.valueOf(value); // Boxing, Wrapping

int j = i.int Value(i) // unboxing

Integer i3 = 30; // auto boxing
int k = i2;

int i = Integer.parseInt(varible);
```

Final Key word

- When a non primitive is final, you cannot reassign cannot the value.
- primitive data type final key word you can't Modified.
- The `final` keyword in Java is used to declare constants, immutable variables, and prevent method overriding or subclassing.
- It makes variables unmodifiable after initialization, methods unoverridable, and classes unextendable. It ensures code robustness, improves performance, and clarifies intent by signaling immutability or constant values.

Garbage Collections

- If there is an object without reference variable then object will be destroyed by "Garbage Collection

Java Packages & API

A package in Java is used to group related classes. Think of it as **a folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

Built in packages

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more. The complete list can be found at Oracles website: https://docs.oracle.com/javase/8/docs/api/.

User-defined Packages

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

Example

```
└── root
  └── mypack
    └── MyPackageClass.java
```

To create a package, use the `package` keyword:

Static Keyword

Static method using an referencing without object

Static Variables: Variables that belong to the class itself, rather than any specific instance of the class. They are shared among all instances of the class.

- **Definition**: Shared variables for all objects of a class.
- **Example**:

```
public class Example {
    static int count = 0; // Static variable
}
```

```
public class Human {
    int age;
    String name;
    int salary;
    boolean married;
    static long population;  // static not referncing an obje

    static void message() {
        System.out.println("Hello world");
//        System.out.println(this.age); // cant use this over
    }

    public Human(int age, String name, int salary, boolean ma
        this.age = age;
        this.name = name;
        this.salary = salary;
        this.married = married;
        Human.population += 1; // class name give the static
    }
}
```

Static Methods: Methods that belong to the class rather than any instance of the class. They can be called without creating an instance of the class.

- **Definition**: Methods that belong to the class, not objects.
- **Example**:

```
public class Example {
    static void display() { // Static method
        System.out.println("Hello, World!");
    }
}
```

```
package com.kunal.staticExample;

public class Main {
    public static void main(String[] args) {
//        Human kunal = new Human(22, "Kunal", 10000, false);
//        Human rahul = new Human(34, "Rahul", 15000, true);
//        Human arpit = new Human(34, "arpit", 15000, true);
//
//        System.out.println(Human.population);
//        System.out.println(Human.population);
//        System.out.println(Human.population);

        Main funn = new Main();
        funn.fun2();

    }

    // this is not dependent on objects
    static void fun() {
//        greeting(); // you cant use this because it require
        // but the function you are using it in does not depe

        // you cannot access non static stuff without referen
        // a static context

        // hence, here I am referencing it
        Main obj = new Main();
```

```
        obj.greeting();
    }

    void fun2() {
        greeting();
    }

    // we know that something which is not static, belongs to
    void greeting() {
//        fun();
        System.out.println("Hello world");
    }
}
```

Static Block: A block of code enclosed in curly braces `{}` inside a class, preceded by the `static` keyword. It is executed when the class is loaded into memory.

- **Definition**: Special code block executed when class is loaded.
- **Example**:

```
public class Example {
    static {
        // Static block
        System.out.println("Static block initialized.");
    }
}
```

```
package com.kunal.staticExample;

// this is a demo to show initialisation of static variables
public class StaticBlock {
    static int a = 4;
    static int b;

    // will only run once, when the first obj is create i.e.
```

```
    static {
        System.out.println("I am in static block");
        b = a * 5;
    }

    public static void main(String[] args) {
        StaticBlock obj = new StaticBlock();
        System.out.println(StaticBlock.a + " " + StaticBlock.

        StaticBlock.b += 3;

        System.out.println(StaticBlock.a + " " + StaticBlock.

        StaticBlock obj2 = new StaticBlock();
        System.out.println(StaticBlock.a + " " + StaticBlock.
    }

}
```

Static Nested Classes: Classes that are declared inside another class and marked as static. They can be accessed without instantiating the outer class.

- **Definition**: Nested class declared inside another class.
- **Example**:

```
public class Outer {
    static class Nested {
        void display() {
            System.out.println("Nested class method");
        }
    }
}
```

```
package com.kunal.staticExample;

import java.util.Arrays;
```

```
public class InnerClasses {

    static class Test {
        String name;
        public Test(String name) {
            this.name = name;
        }

        @Override
        public String toString() {
            return name;
        }
    }

    public static void main(String[] args) {
        Test a = new Test("Kunal");
        Test b = new Test("Rahul");

        System.out.println(a);

//        System.out.println(a.name);
//        System.out.println(b.name);
    }
}

//static class A {
//
//}
```

Singleton Classes

In object-oriented programming, a java singleton class is a class that can have only one object (an instance of the class) at a time. After the first time, if we try to instantiate the Java Singleton classes, the new variable also points to the first instance created.

1. Make a constructor private.
2. Write a static method that has the return type object of this singleton class. Here, the concept of Lazy initialization is used to write this static method.

```
package com.kunal.singleton;

import com.kunal.access.A;

public class Singleton {
    private Singleton () {

    }

    private static Singleton instance;

    public static Singleton getInstance() {
        // check whether 1 obj only is created or not
        if (instance == null) {
            instance = new Singleton();
        }

        return instance;
    }

}
```

```
package com.kunal.singleton;

import com.kunal.access.A;

public class Main {
    public static void main(String[] args) {
        Singleton obj1 = Singleton.getInstance();

        Singleton obj2 = Singleton.getInstance();
```

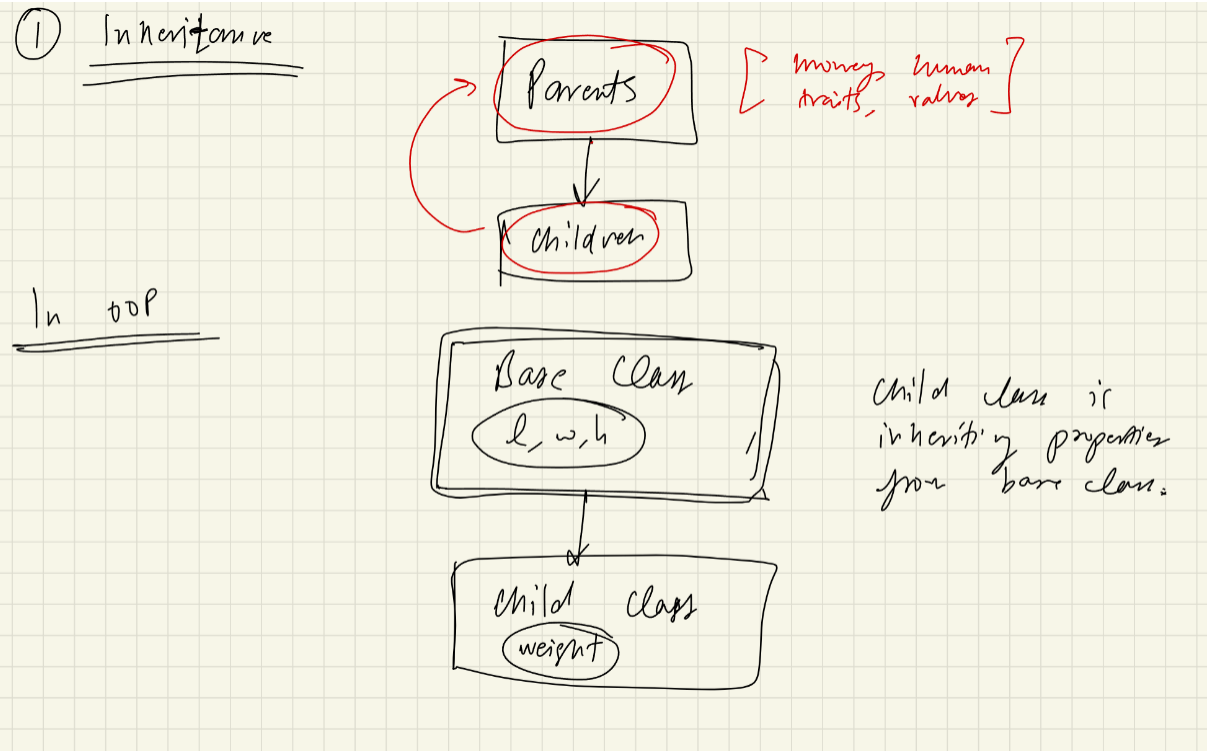
```
        Singleton obj3 = Singleton.getInstance();

        // all 3 ref variables are pointing to just one objec

        A a = new A(10, "Kunal");
        a.getNum();
//        int n = a.num;
    }
}
```

Principle of oops

- Inheritance
- polymorphism
- Encapsulation
- Abstraction

Inheritance

- Inheritance is an important pillar of OOP(Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class
- Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition.
- In heritance is base classes inherit to derived class you can also create a new properties on derived classes and also used existing base class properties .



Syntax

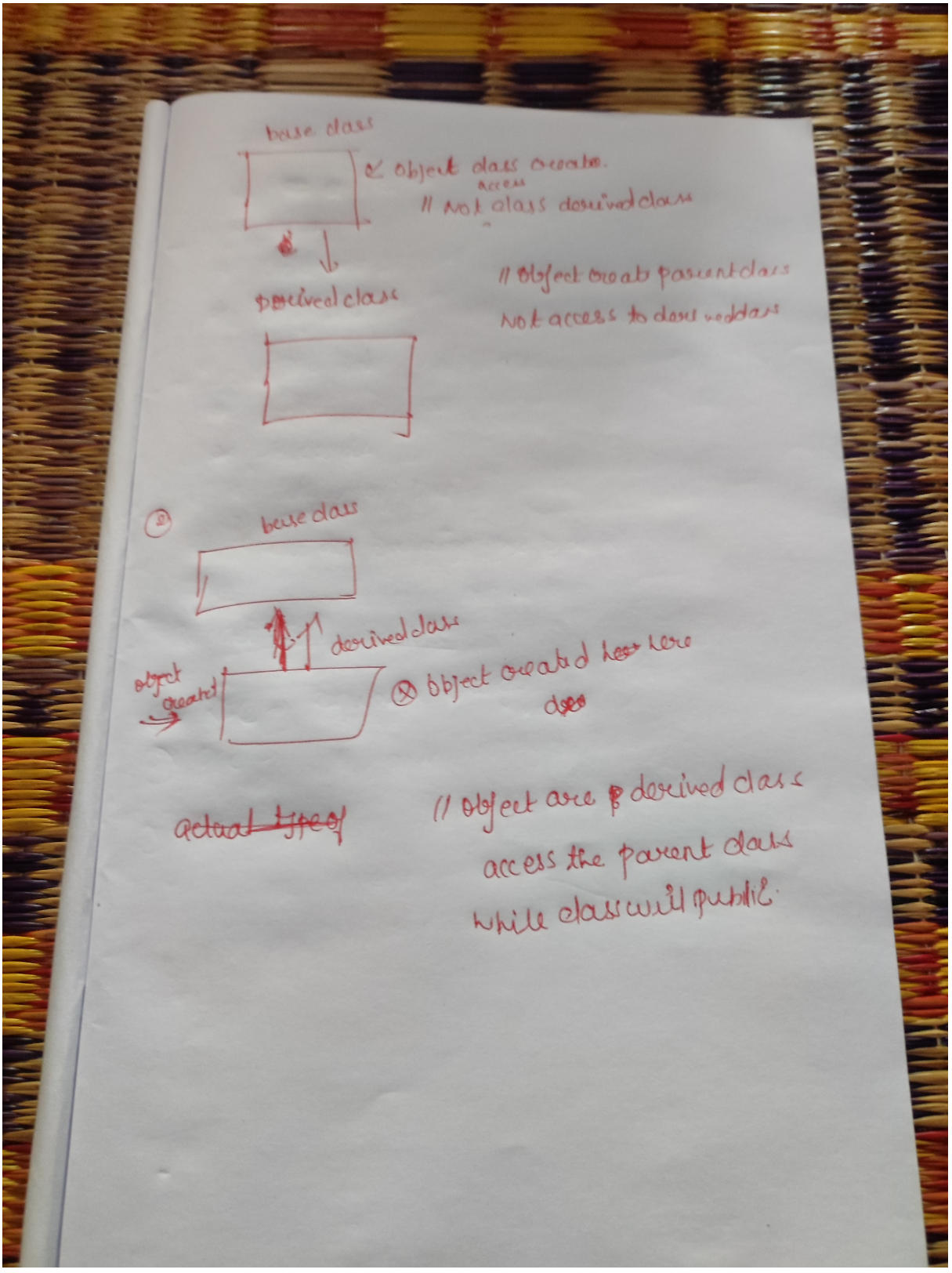
```
class derived-class extends base-class
{
   //methods and fields
}
```

```
public class Oops{


  static class Department{
```

```
    String college;
    String  Department;
    int student_ls;
    int Staff_ls;

    Department()
    {
       college =" IIT Madras";
       Department = "Computer Science";
       Staff_ls=20;
       student_ls=100;
    }
}
  public static class Student extends Department{

    String Mentor_name;
    int  student_rollno;
    int semster;

    Student()
    {
       Mentor_name="ram";
       student_rollno = 01;
       semster = 01;
    }

  }

    public static void main(String[] args) {
        Student student = new Student();

        System.out.println(student.Mentor_name);
        System.out.println(student.Department);
    }
```

```
}
```

Access to object



- In object-oriented programming (OOP), when you create an object of a parent class, you can't directly access the members or methods of derived

classes. The reason is that the object created is of the parent class type, so it only contains the attributes and methods defined in the parent class.

- However, if the methods or attributes in the derived classes are accessible through inheritance (i.e., they are not private or protected), you can access them by creating an object of the derived class and then accessing those members or methods through that object.

```
    Box box5 = new BoxWeight(2, 3, 4, 8);
       System.out.println(box5.w);  // you can inherit devie
                                    // constructor automatic

       // there are many variables in both parent and child
       // you are given access to variables that are in the
       // hence, you should have access to weight variable
       // this also means, that the ones you are trying to a
       // but here, when the obj itself is of type parent cl
       // this is why error
       BoxWeight box6 = new Box(2, 3, 4);
       System.out.println(box6);  // you can  inherit base
                                    // constructor automatic
```

Super class

- Super is reference to the parent class constructor .
- Super classes is always reference to the parent constructors
- `this` keyword: Refers to the current instance of the class, used to access instance variables or methods of the current object.
- `super` keyword: Refers to the parent class, used to call the parent class's constructor or methods from the subclass.

superclass methods and variables:

The `super` keyword can be used to access methods and variables of the superclass from within the subclass. This is particularly useful when the subclass overrides a method of the superclass but still needs to call the superclass's version of that method.

```
class Vehicle {
    int maxSpeed = 120;
}

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

    void display()
    {

        System.out.println("Maximum Speed: "
                           + super.maxSpeed); // calling constru
                                  // print value maxspeed is 120
    }
}

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

Calling the superclass constructor:

When you create an object of a subclass, the constructor of the superclass is called implicitly before the constructor of the subclass. However, you can explicitly call a specific constructor of the superclass using the `super()` keyword. This is useful when the superclass has multiple constructors, and you want to call a specific one

```
class Superclass {
    void display() {
        System.out.println("Superclass method");
    }
}

class Subclass extends Superclass {
    @Override
    void display() {
        super.display(); // Calls the superclass method
        System.out.println("Subclass method");
    }
}

public class Main {
    public static void main(String[] args) {
        Subclass obj = new Subclass();
        obj.display();
    }
}

Output code

Superclass constructor
Subclass constructor
```

Types of Inheritance

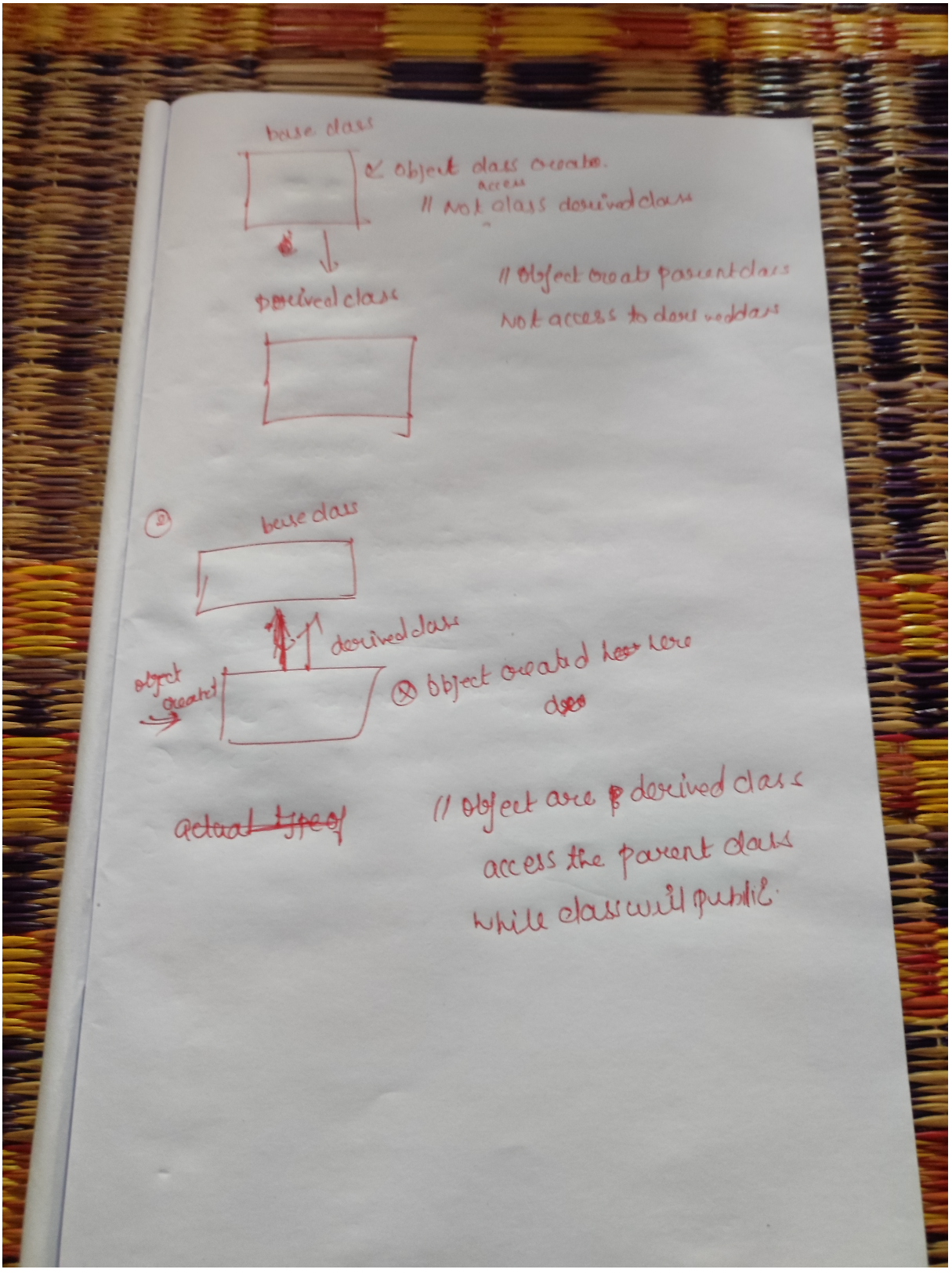
- Single inheritance
- Multiple inheritance
- Multilevel inheritance
- Hierarchical inheritance
- Hybrid inheritance

Single Inheritance

Single Inheritance is one classes extends from another classes

In the oops concept create object dervied classes to base classes

Example



// object class create.
// Not access derived class

// Object create parent class
Not access to derived class

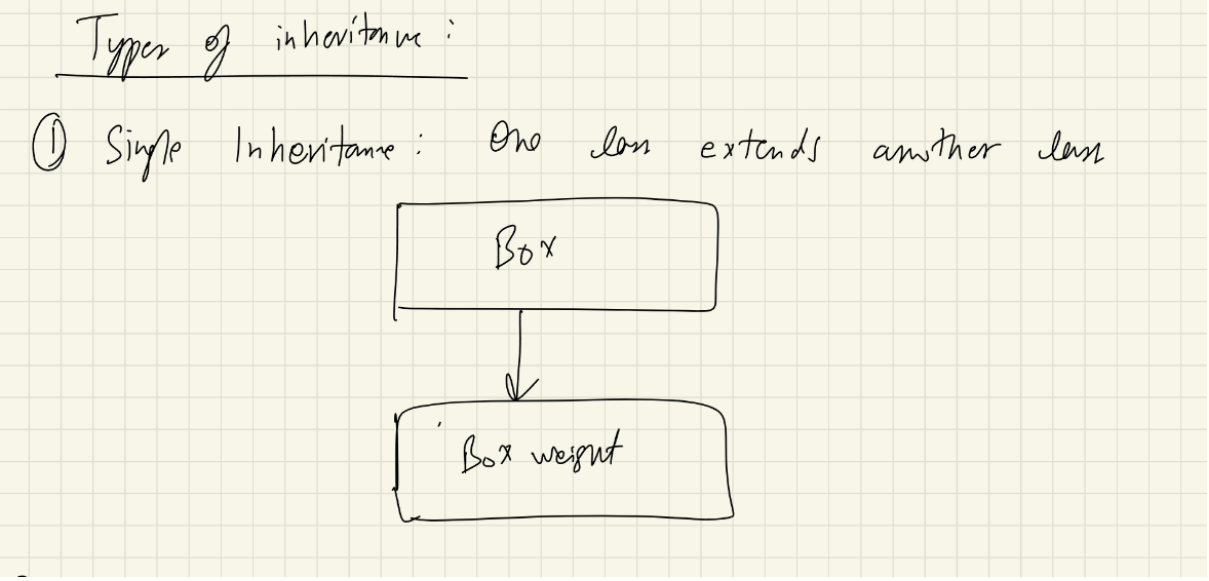
(2)

⊗ Object created here

actual type of

// Object are of derived class
access the parent class
while class will public.

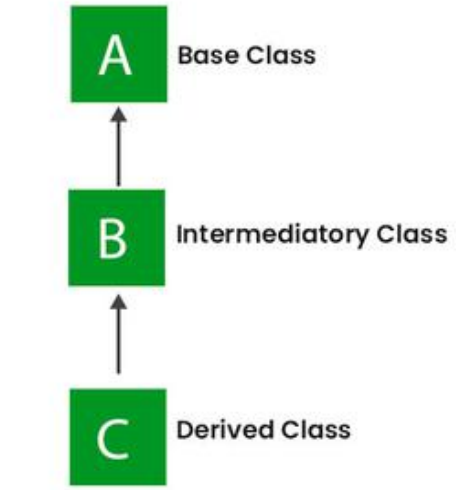
```
class derived-class extends base-class
{
   //methods and fields
}
```



multi level inheritance

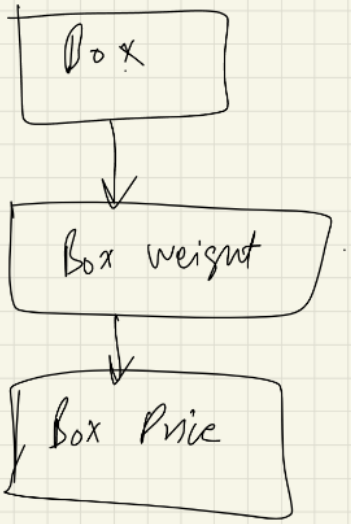
- In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes.
- In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.

Visual Represtation



Multilevel Inheritance

(2) multilevel Inheritance :




```
// Java program to illustrate the
// concept of Multilevel inheritance
import java.io.*;
import java.lang.*;
```

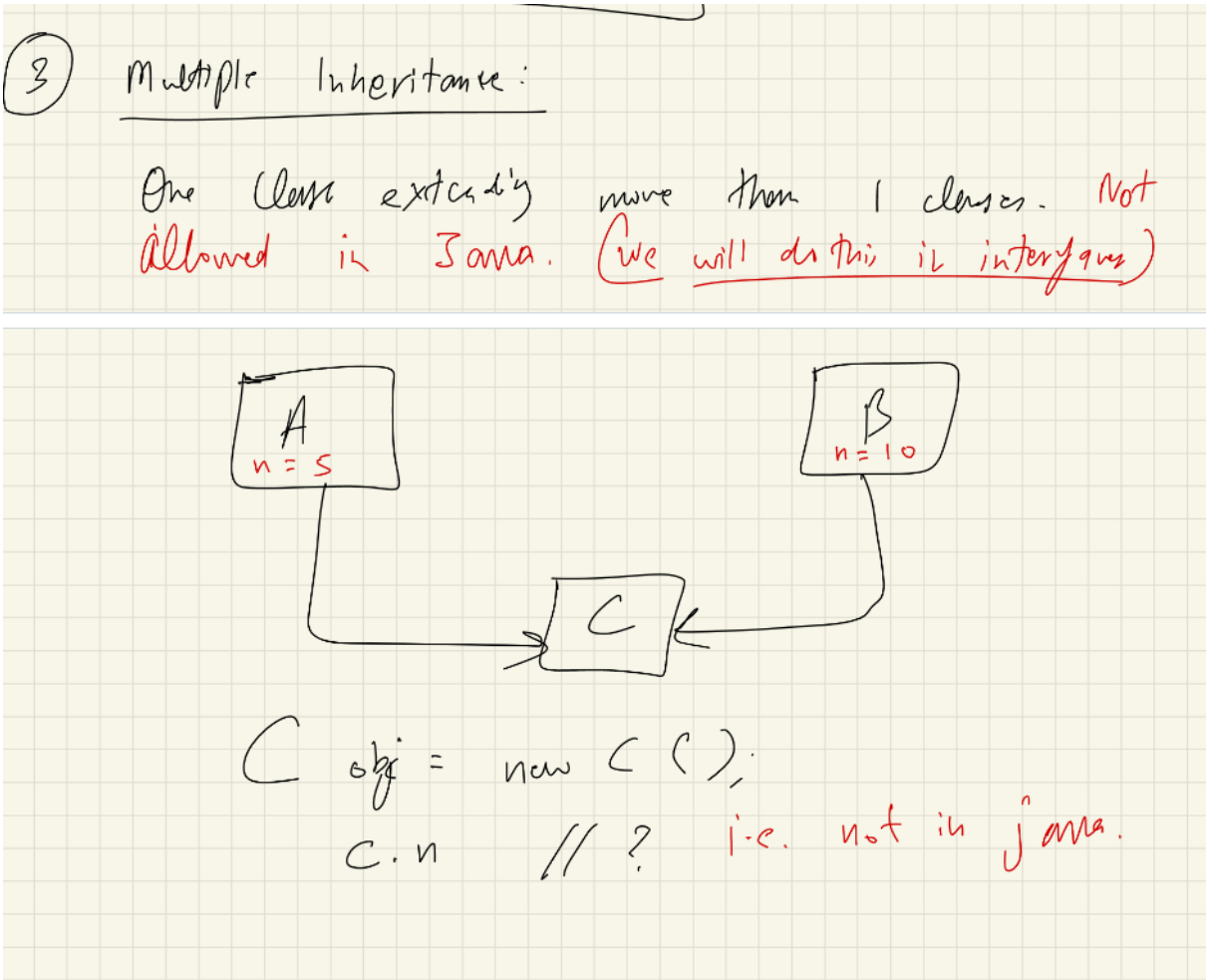
```
import java.util.*;

class One {
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}

class Two extends One {
    public void print_for() { System.out.println("for"); }
}

class Three extends Two {
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}

// Drived class
public class Main {
    public static void main(String[] args)
    {
        Three g = new Three();
        g.print_geek();
        g.print_for();
        g.print_geek();
    }
}
```

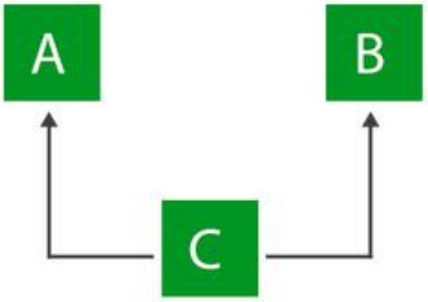
Multiple Inheritance

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes.

Java does not support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces. In the image below, Class

C is derived from interfaces A and B.

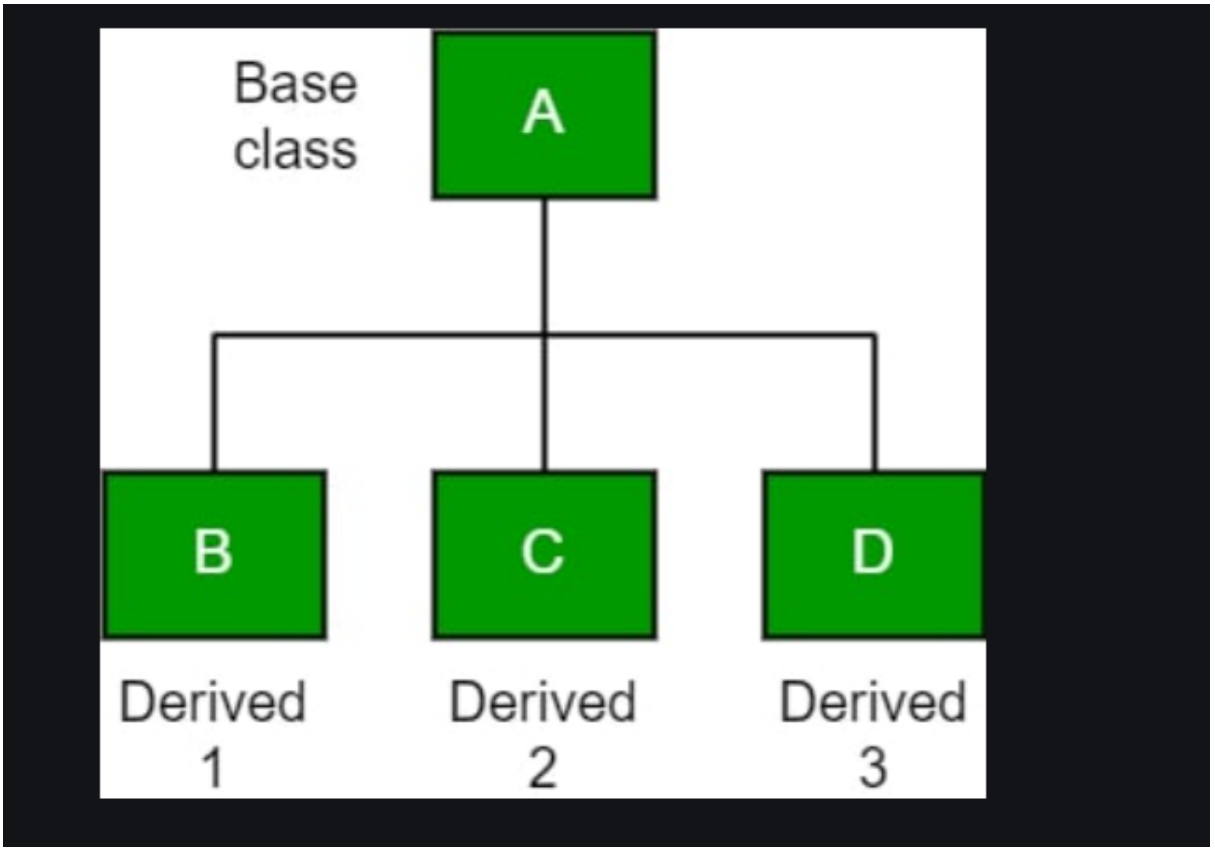




Multiple Inheritance

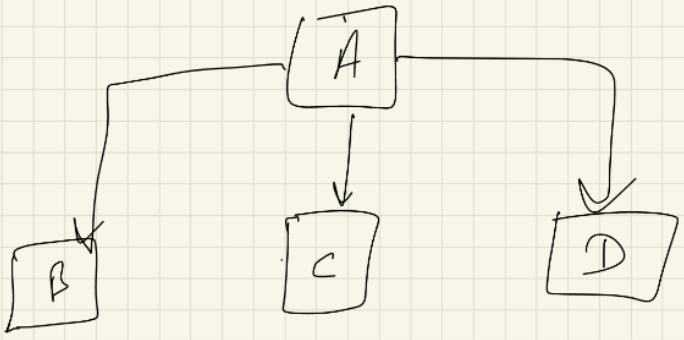
Hierarchical Inheritance

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



(4) Hierarchial Inheritance :

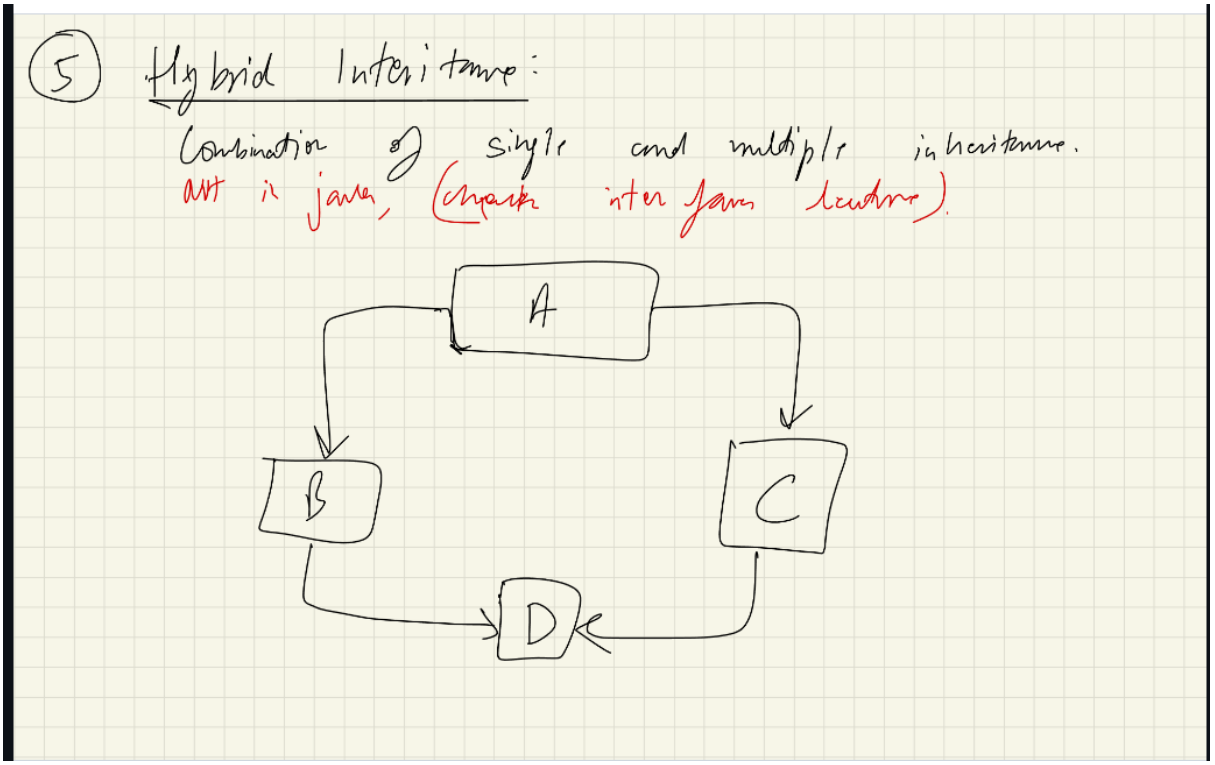
One class is inherited by many classes.



```
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
}}

// it working
```

Hybrid inheritance

- Hybird Inheritance combination of single and Multiple inheritance
- Hybird Inheritance is not access java but only use in interfaces

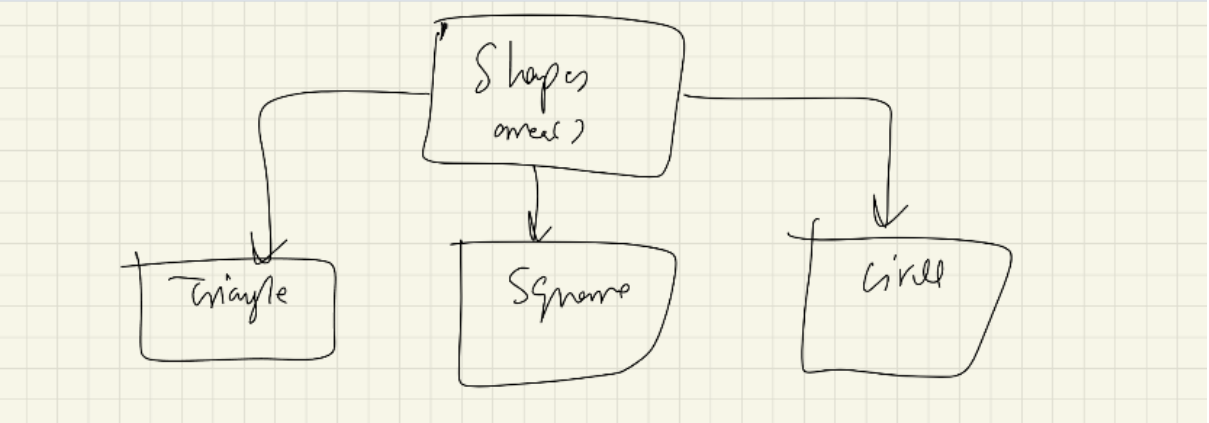


Ploymorphsim

> Poly Means Many

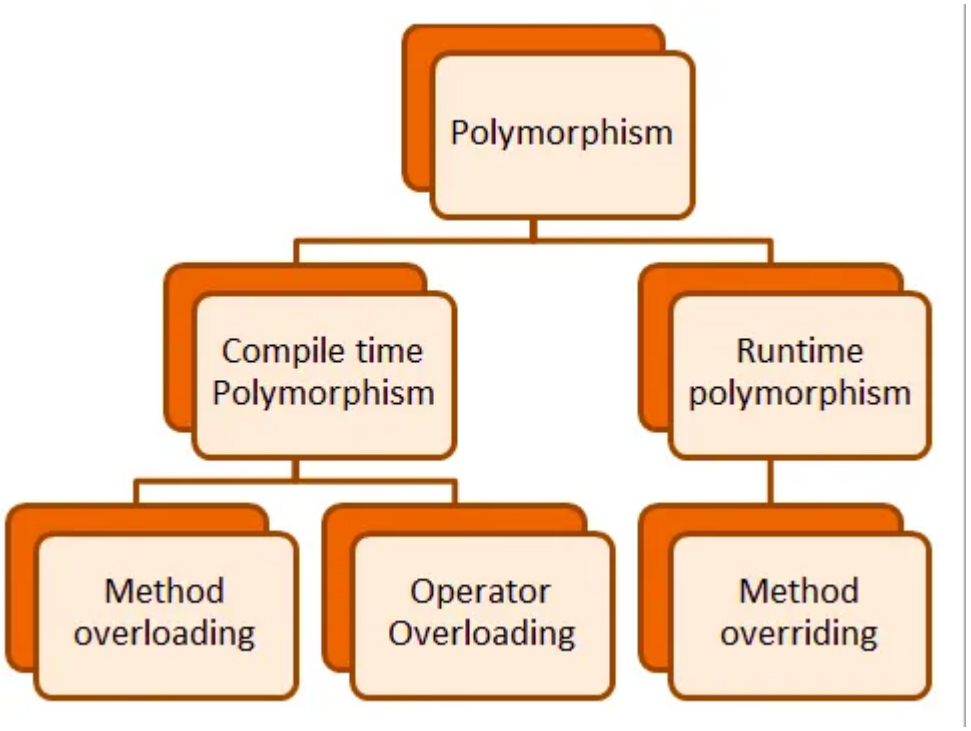
> morphism ⇒ ways to respresent

Syntax of ploymorphism



Types of Polymorphism

- Compile Time polymorphism
- Static polymorphism



Compile time Polymorphism

- Compile Time polymorphism / Static polymorphism Achived via Method Overloading
- Same name methods, Constructor but types Arugmuents,returntypes ordering can be different

Eg: Multiple constructor

```
class Helper {

    // Method with 2 integer parameters
    static int Multiply(int a, int b)
    {
        // Returns product of integer numbers
        return a * b;
    }

    // Method 2
    // With same name but with 2 double parameters
    static double Multiply(double a, double b)
    {
```

```
        // Returns product of double numbers
        return a * b;
    }
}

// Class 2
// Main class
class GFG {
    // Main driver method
    public static void main(String[] args)
    {
        // Ploymorphsim Calling method by passing

        System.out.println(Helper.Multiply(2, 4));
        System.out.println(Helper.Multiply(5.5, 6.3));
    }
}
```

Dynamic polymorphism

Dynamic polymorphism or Runtime Polymorphism

```


// Base Class
class Parent {
    void show() //same method name with  one to access the va

{ System.out.println("Parent's show()"); }
}

// Inherited class
class Child extends Parent {
```

```
 void show()   //same method name with  one to access the var
    {
        System.out.println("Child's show()");
    }
}

// Driver class
class Main {
    public static void main(String[] args)
    {
        Parent obj1 = new Parent();
        obj1.show();
parent object = new child();
   object.show();
//refercing varible on parent class but created
// child classes object is created working only
// Java run a method in referece in child classes only not a
    }
}
```

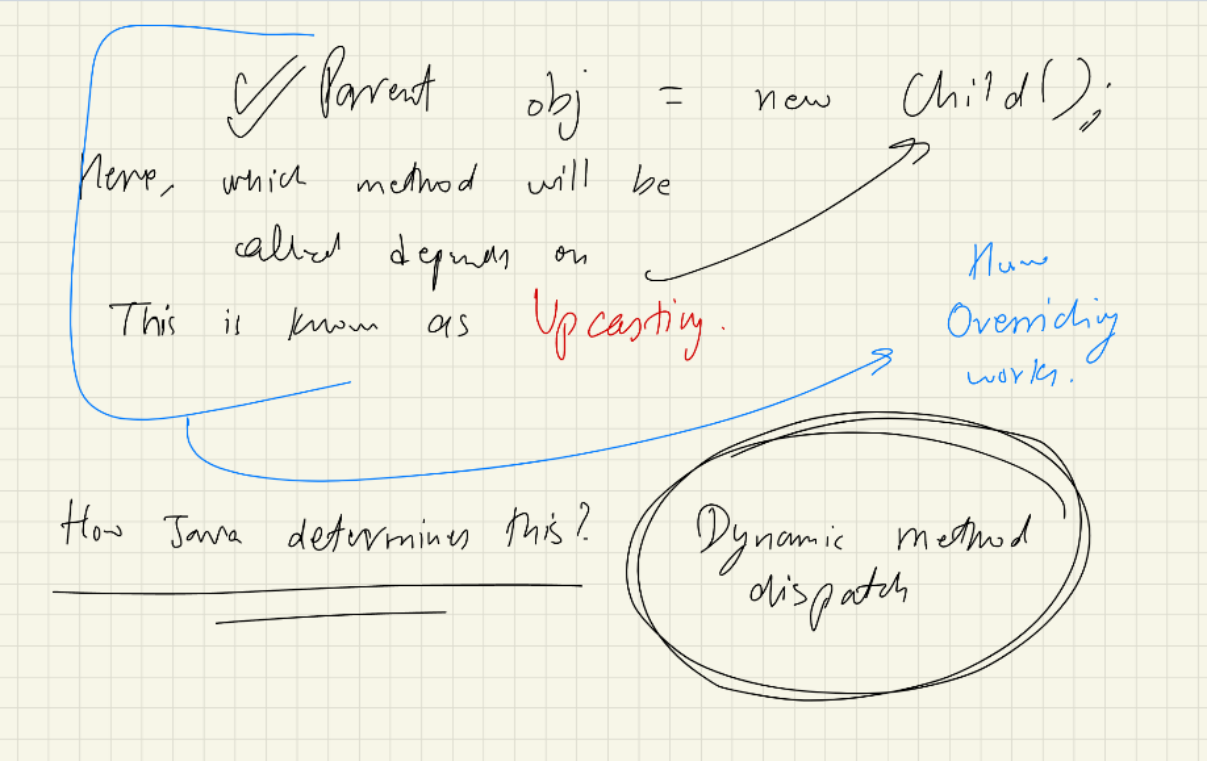
Overiding

- "Overriding involves the use of polymorphism, wherein subclasses inherit methods from their base classes. However, a main disadvantage of inheritance arises when a user calls a method with the same name, resulting in the execution of the method from the parent class."

1. **Overriding and Polymorphism**:
    - Overriding: Subclasses can redefine methods inherited from their parent classes.
    - Polymorphism: Allows a single method call to behave differently based on the object's type.
2. **Disadvantage of Inheritance**:
    - Inheritance: Subclasses inherit properties and behaviors from their parent classes.

- Drawback: Method conflicts arise when a method with the same name exists in both the parent and child classes.

3. **Execution of Parent Class Method**:
   - When a method is called on a subclass object with the same name as a method in the parent class:
   - The parent class method is executed instead of the overridden method in the subclass.
   - This behavior may cause unexpected results, deviating from the intended subclass functionality.



Early Binding and Late binding

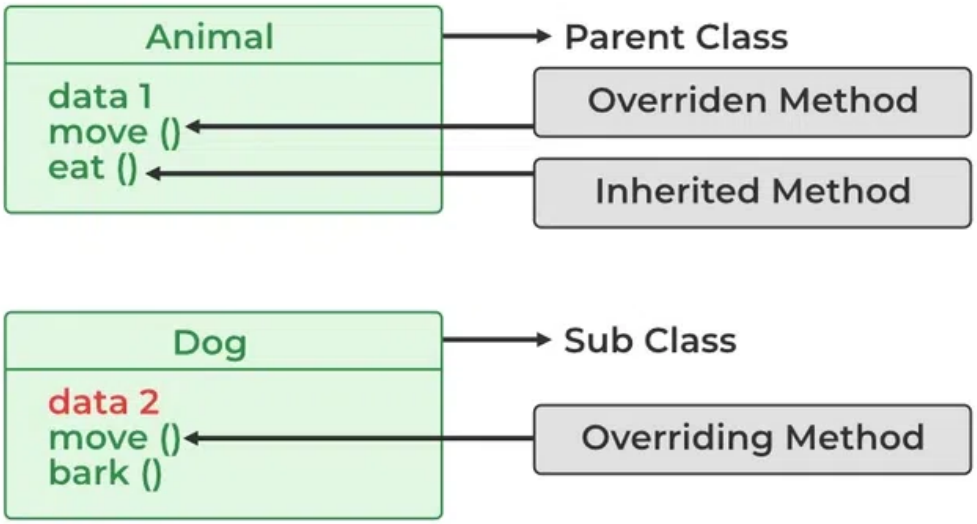
1. **Early Binding**:
   - In early binding, the association between a method call and the method implementation is resolved at compile time.

- This means that the compiler knows exactly which method will be called and can directly link the method call to its implementation.
- Early binding is usually more efficient in terms of performance because the method resolution is done once, during compilation.
- However, it lacks flexibility because it requires that the method to be called is known at compile time.

2. **Late Binding**:
    - In late binding, also known as dynamic binding or runtime binding, the association between a method call and the method implementation is resolved at runtime.
    - This means that the decision about which method to call is deferred until the program is actually running.
    - Late binding allows for greater flexibility because it enables things like polymorphism, where a single method call can behave differently depending on the actual object being referred to.
    - However, late binding typically incurs a performance overhead because the method resolution needs to be done each time the method is called at runtime.

Final Keyword

- Final Keyword prevent overriding at Runtime .
- The final keyword always prevents the overriding of classes and methods. For example, inheriting the parent class with the final keyword will not work because the final keyword does not allow it.
- Using the static keyword prevents the overriding of values because static does not depend on objects. A static method is referenced using the class name or a reference variable; you cannot access it by calling on objects.
- Static methods do not depend on objects, so they cannot be overridden based on objects. Therefore, static methods cannot be overridden.



Encapsulation and Abstraction

- Abstraction solves design-level issues, while Encapsulation addresses implementation issues by hiding the code within a single combined and protected form from the outside world.
- Abstraction focuses on the external level of implementation, whereas Encapsulation concentrates on the internal level of implementation.
- Abstraction involves gaining information, while Encapsulation involves containing information at a certain level.
- Abstraction works with abstract classes and interfaces, whereas Encapsulation handles access modifiers and methods."

Access Modifier’s

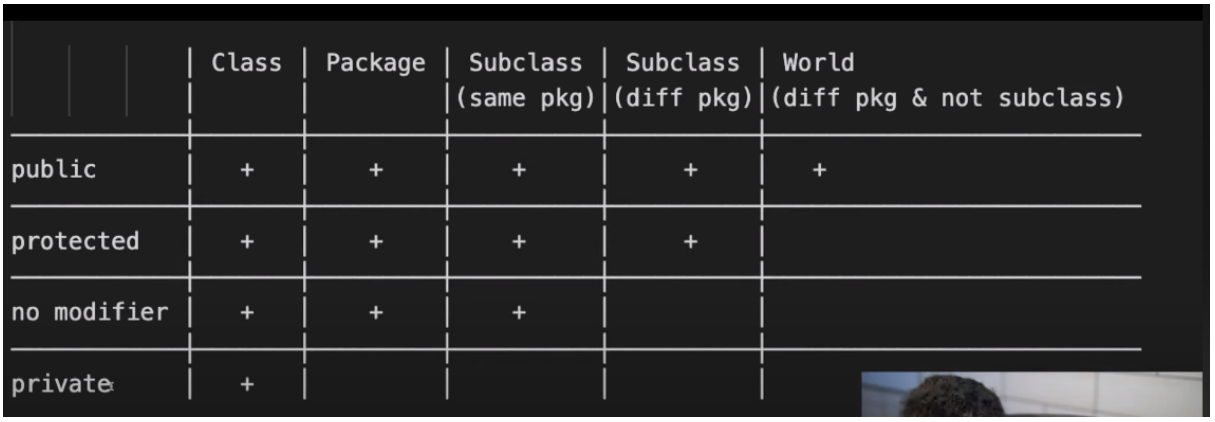
default package not Access out side the packages.

public package is Access in every Where

private package only access inside the class and methods

protected is access classes, packages, Sub class Package in same pkg, Sub class Package in different package

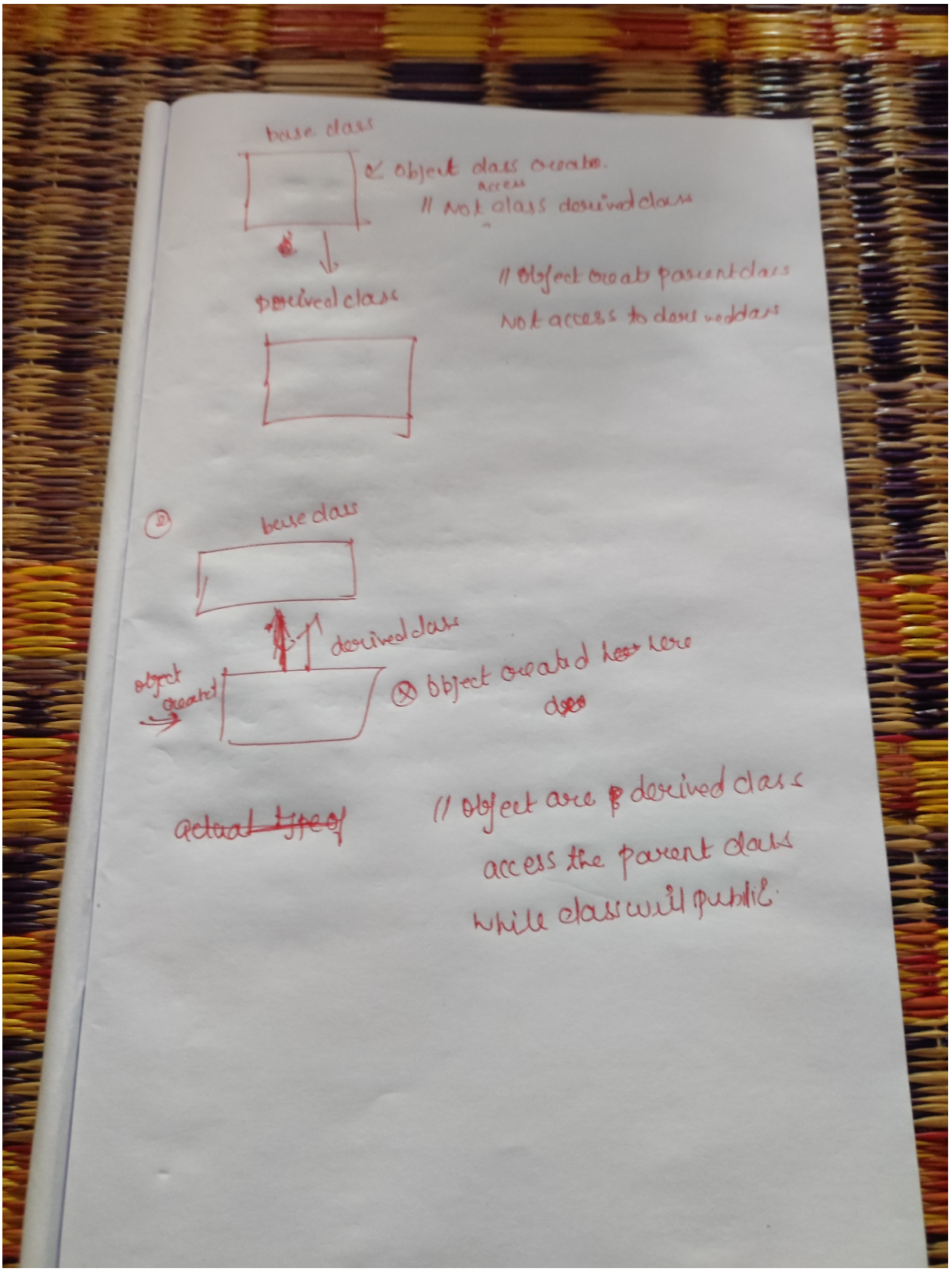
Rules for Access Modifier’s



	Class	Package	Subclass (same pkg)	Subclass (diff pkg)	World (diff pkg & not subclass)
public	+	+	+	+	+
protected	+	+	+	+	
no modifier	+	+	+		
private	+				

When to use the Access modifier’s

- "Private" holds sensitive data and should not be accessed directly within the same class, using only getter and setter methods.
- Getter and setter methods must be declared as "Public."
- "No modifier" signifies the default access level, allowing access only within the same package or class, but not outside of it.
- "Protected" access modifier allows access within the class, its package, and subclasses (inheritance), but not outside of the package or class hierarchy.
- "Protected" is primarily used for inheriting classes in subclasses, where subclasses must only access variable data.
- "Public" allows access from anywhere and everywhere.
- In protected Only you access inherit the Subclasses Not know the parent classes
- Simple protected access only Subclasses only.


base class
// object class create.
// Not access class derived class
derived class
// Object create parent class
Not access to derived class
base class
derived class
object created
// Object create derived class
access the parent class
while class will public.

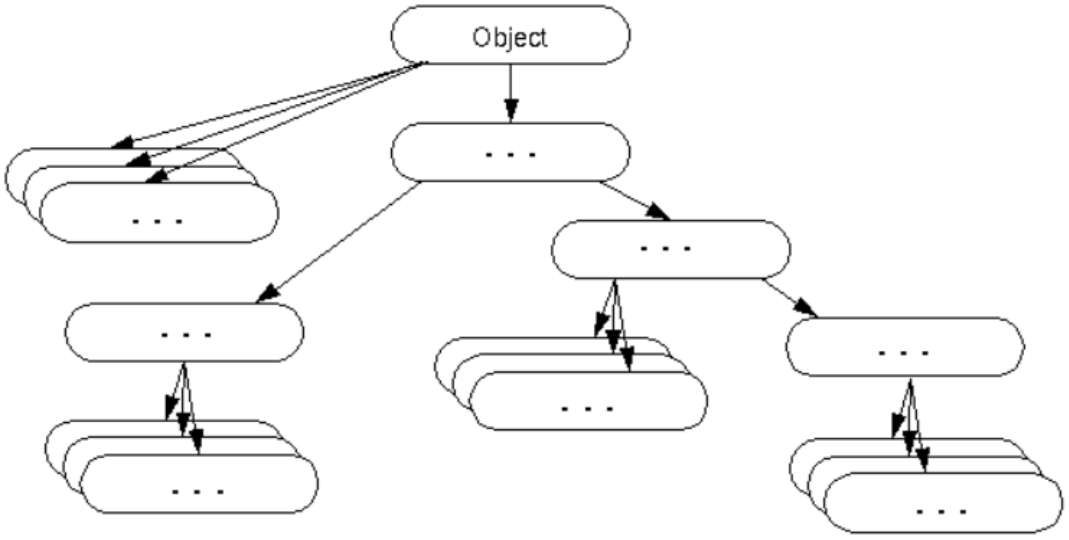
- Protected Key word also refers to the base classes

Common Packages

- lang: Language - Java programming language core package containing fundamental classes and interfaces.
- io: Input/Output - Java package for input and output operations, including file handling.
- util: Utility - Java package containing various utility classes and data structures.
- applet: Application let - A small application program that runs within a larger application or web browser.
- awt: Abstract Window Toolkit - Java package for creating graphical user interfaces (GUIs).
- net: Network - Java package for networking functionality, including client-server communicati

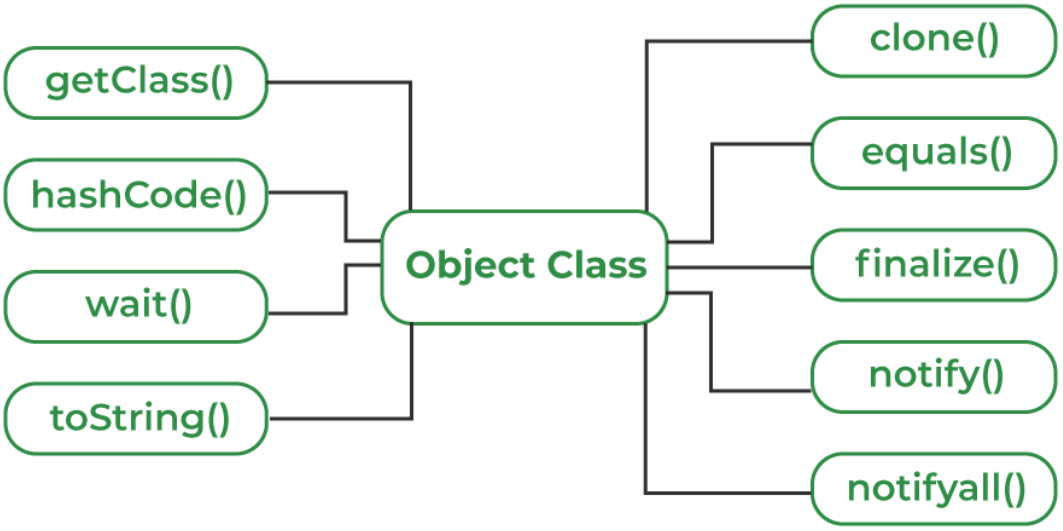
Object classes

- The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
- The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.



Object class Methods

- tostring() method
- hashCode() method
- equals(Object obj) method
- finalize() method
- getClass() method



1. `toString()`

Definition:

The `toString()` method is a method in Java that returns a string representation of an object. This method is automatically called when you try to concatenate an object with a string or when you try to print an object using `System.out.println()`.

Example:

```
javaCopy code
class Car {
    String brand;
    int year;

    public Car(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }

    public String toString() {
        return "Car{" +
                "brand='" + brand + '\'' +
                ", year=" + year +
```

```
                '}';
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", 2020);
        System.out.println(myCar); // Output: Car{brand='To
yota', year=2020}
    }
}
```

2. `hashCode()`

Definition:

The `hashCode()` method in Java returns a hash code value for the object. This method is used by hash-based data structures such as HashMap, HashSet, etc.

Example:

```
javaCopy code
public class Main {
    public static void main(String[] args) {
        String str = "Hello";
        int hashCode = str.hashCode();
        System.out.println("Hash code for 'Hello': " + hash
Code);
    }
}
```

3. `equals(Object obj)`

Definition:

The `equals(Object obj)` method is used to compare two objects for equality. The

default implementation of this method in the Object class compares memory addresses, but it is often overridden in user-defined classes to compare object contents.

Example:

```
javaCopy code
class Student {
    String name;
    int id;

    public Student(String name, int id) {
        this.name = name;
        this.id = id;
    }

    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) re
turn false;
        Student student = (Student) obj;
        return id == student.id &&
                Objects.equals(name, student.name);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Alice", 123);
        Student s2 = new Student("Alice", 123);
        System.out.println(s1.equals(s2)); // Output: true
    }
}
```

4. `finalize()`

Definition:
The
`finalize()` method is called by the garbage collector before reclaiming memory occupied by an object. It is a method that can be overridden to provide cleanup code for an object before it is garbage collected.

Example: This method is rarely used explicitly.

5. `getClass()`

Definition:
The
`getClass()` method returns the runtime class of an object. It is a final method in the Object class, so it is available to all Java objects.

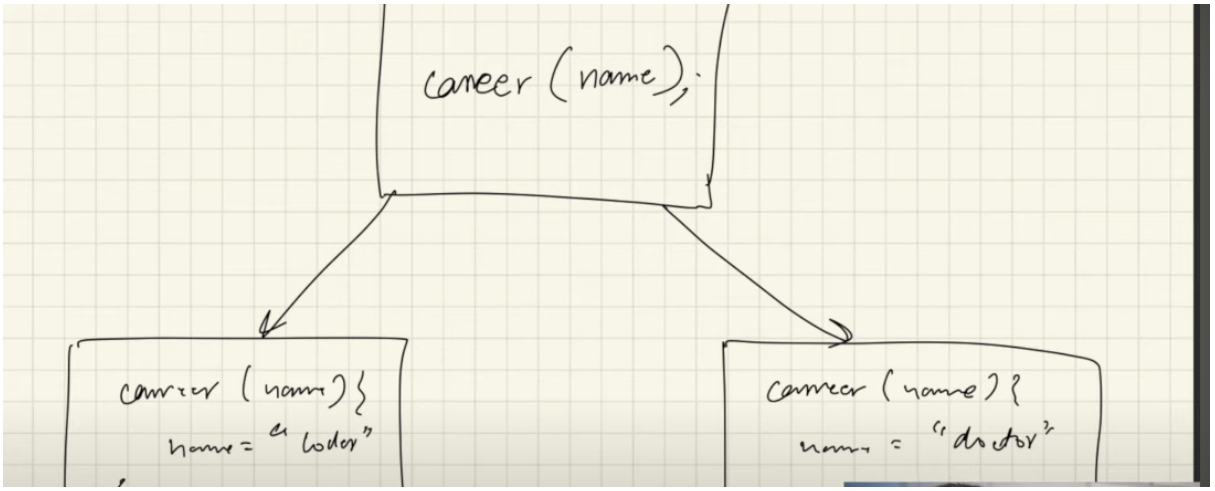
Example:

```
public class Main {
   public static void main(String[] args) {
       String str = "Hello";
       Class<?> cls = str.getClass();
       System.out.println("Class of 'Hello': " + cls.getName
   }
```

Abstract Classes

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.

Abstraction can be achieved with either **abstract classes** or interfaces



- Inheritance conflicts inherit the same Function Name sloving the conflict using Abstract classes
- Overriding sloving the conflicts

The `abstract` keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).
- when you declare an abstract method inside an abstract class, you should include the name add keyword abstract class name.
- Abstract class can't create a object and Abstract class is created in super classes /Parent classes no Body
- Abstract class you can inherit the parent class
- In a method, abstract classes are used, and there may also be abstract classes used within other classes.
- Abstract class you Overiding in using child class object

```
abstract class Animal {
    abstract void makeSound(); // Abstract method
}
```

```
abstract class Animal {
    abstract void makeSound();   //Abstract classes no body
}

class Dog extends Animal {
    void makeSound() {                // Abstarct class only
        System.out.println("Bark");   // Dog is overriding
    }
}

class Cat extends Animal {
    void makeSound() {                  //Abstarct class only
        System.out.println("Meow");     // cat is overiding
    }
}

public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog();
        Animal cat = new Cat();          // only created a Ob

        dog.makeSound();
        cat.makeSound();
    }
}
```

```
public class main{

 static abstract class parent{

   abstract void carrer();
   abstract void age();
 }
```

```
  static public class Son extends parent{

    public void carrer(){
       System.out.println("computer Science");
    }
    public void age(){
       System.out.println("Age is 25");
    }
  }

  static public class Daughter extends parent{

   public void carrer(){
      System.out.println("Doctor");
   }
   public void age(){
       System.out.println("Age is 20");
   }

  }

  public static void main(String[] args) {

   Son son = new Son();

     son.carrer();
     son.age();

   Daughter daughter = new Daughter();
     daughter.carrer();
     daughter.age();
   }


}
```

Abstract class constructor

- Abstract class constructor does created because of Abstract classes does not create a object , so Abstract classes no body no instance variable and methods
- Calling the constructor only the child classes or derived classes

```
public class main{

 static abstract class parent{
    int age;       // Abstract class instance Variable
   abstract void carrer();
   abstract void age();
 }

  static public class Son extends parent{

    public  Son(int age)
    {
       this.age = age;   // only Calling the constructor only
    }
    public void carrer(){
       System.out.println("computer Science");
    }
    public void age(){
       System.out.println("Age is "+age);
    }
  }

  static public class Daughter extends parent{

   public void carrer(){
      System.out.println("Doctor");
   }
   public void age(){
       System.out.println("Age is 20");
```

```
    }

   }

   public static void main(String[] args) {

    Son son = new Son(44);

    son.age = 44;
      son.carrer();
      son.age();

    Daughter daughter = new Daughter();
      daughter.carrer();
      daughter.age();
    }


}
```

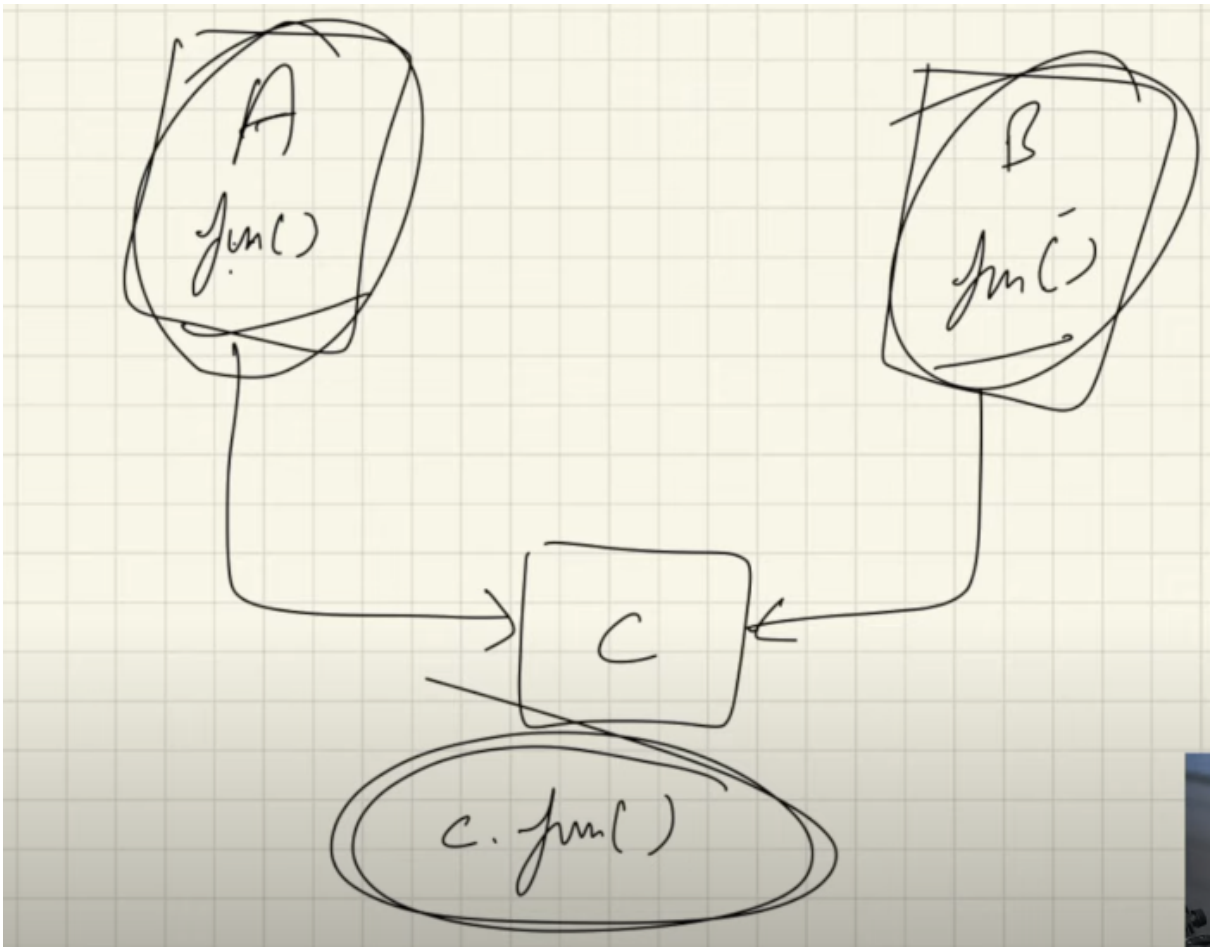
Abstract class Object

Abstract class Object does created because of Abstract classes does not create a object , so Abstract classes no body no instance variable and methods

Static Method

Static method using in Abstract classes because Abstract classes static method not depending on Object

Interfaces



Interfaces

- Interface contains abstract classes
- An `interface` is a completely "**abstract class**" that is used to group related methods
- Like **abstract classes**, interfaces **cannot** be used to create objects
- Interfaces made up on static and final
- Interfaces biggest usage in containing a multiple interface in interfaces classes it also part of Abstract classes
- To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the `implements` keyword (instead

of `extends`). The body of the interface method is provided by the "implement"

- On implementation of an interface, you must override all of its methods
- Interface methods are by default `abstract` and `public`
- Interface attributes are by default `public`, `static` and `final`
- An interface cannot contain a constructor (as it cannot be used to create objects)

Classes

- class is like a blue print
- classes Named group of properities and function
- classes is used combined properties and Methods
- classes and object are important object oriented programming oriented Pillars are Encaplusation, inheritance, Abstraction, Ploymorphism
- Inheritance ⇒ Inherit the value form one class to another classes
- Polymorphism ⇒ solving the problem of Inheritance and polymorphism work for constructor value.ploymorphism,are Operator overloading, Function overloading, Overriding
- Abstraction focuses on the external level of implementation, whereas Encapsulation concentrates on the internal level of implementation.
- Abstraction involves gaining information, while Encapsulation involves containing information at a certain level.

Why need Class

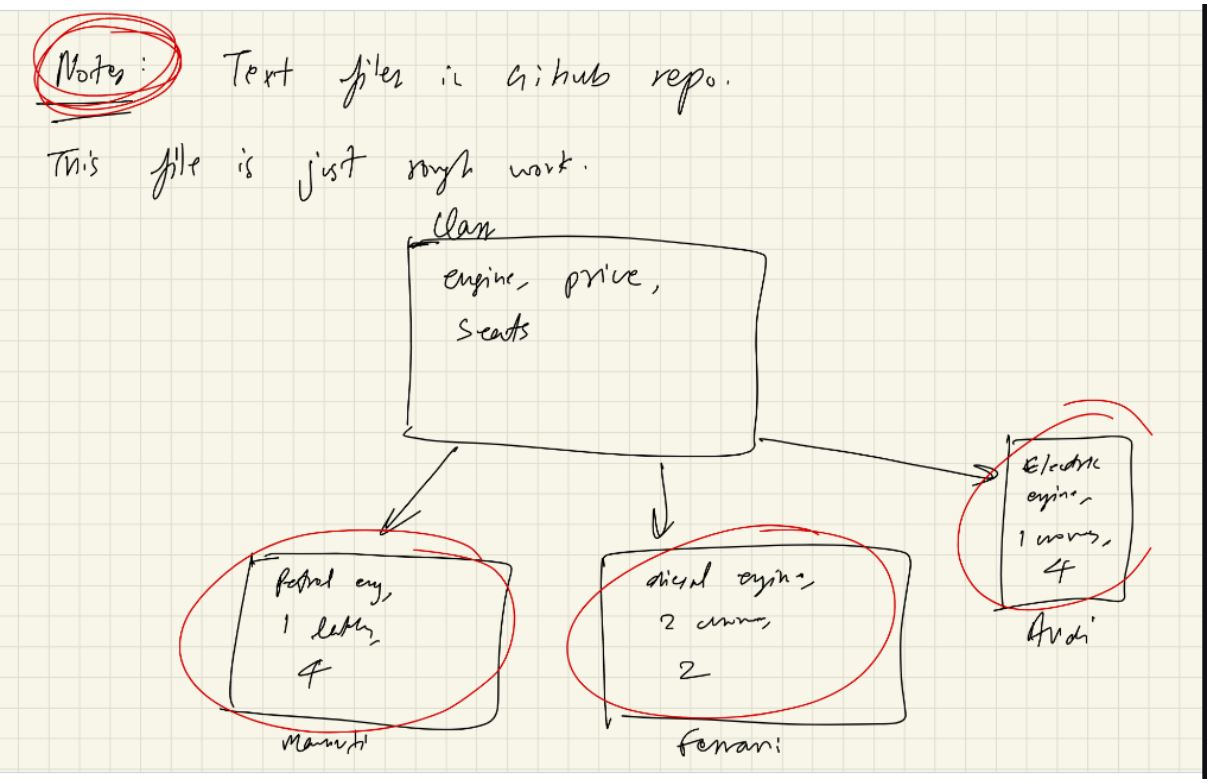
Consider a scenario where we need to manage information about students, such as their roll numbers and names. Initially, we might resort to separate arrays for each piece of data:

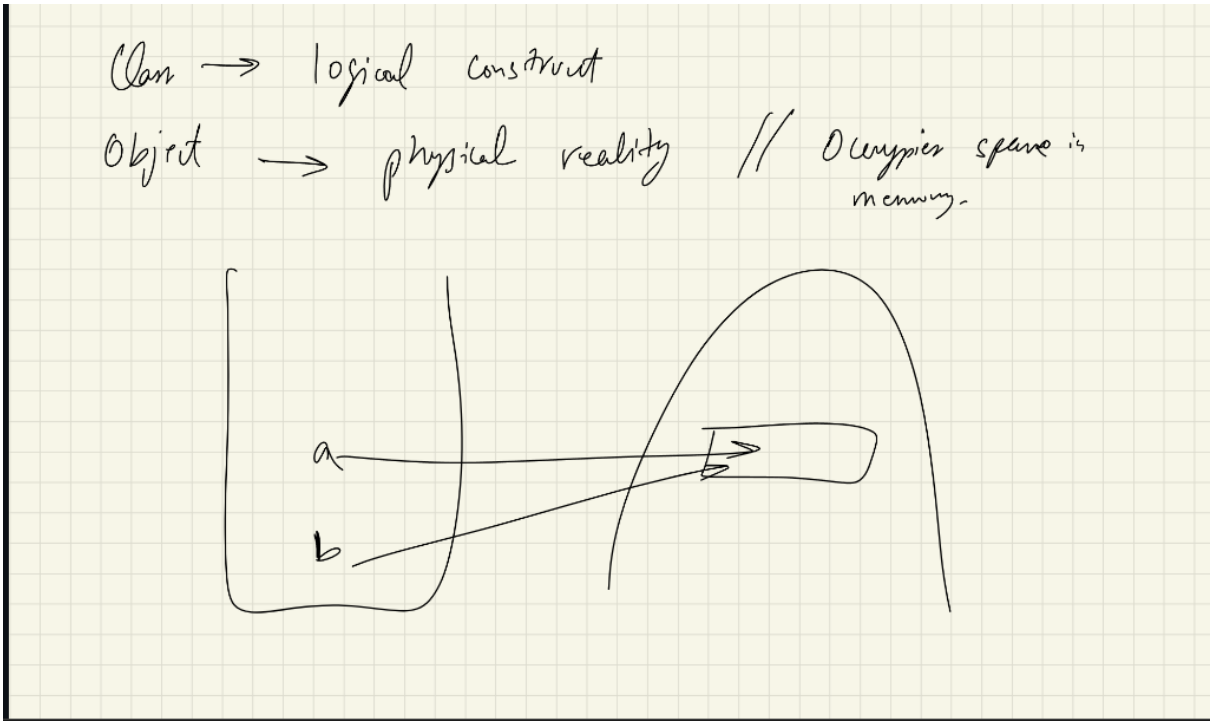
```
javaCopy code
int[] rollNumbers = new int[5]; // Array for storing roll n
umbers
```

```
String[] names = new String[5]; // Array for storing studen
t names
```

While this approach works, it can quickly become unwieldy as the program grows. Every additional piece of data necessitates another array, leading to increased memory consumption and complex data management. Classes offer a more elegant solution by consolidating related data and behavior into a single entity.

Classes





- Classes is created a own purpose contains own **Methods and properties.**
- Class is a template of an Object
- Object is an instance of classes

> Classes ⇒ Logical Value // Not Occpuing Space in Memory
> object ⇒ physical object // Occuping space in memory

Objects

- Object Occuping space in Memory Handle
- Object Access using . dot operator using Reference Variable
- Instance Variable is Object inside the properties
- It is useful to think of an object's identity as the place where its value is stored in memory.
- The behavior of an object is the effect of data-type operations.

- The dot operator links the name of the object with the name of an instance variable. Although commonly referred to as the dot operator, the formal specification for Java categorizes the . as a separator.

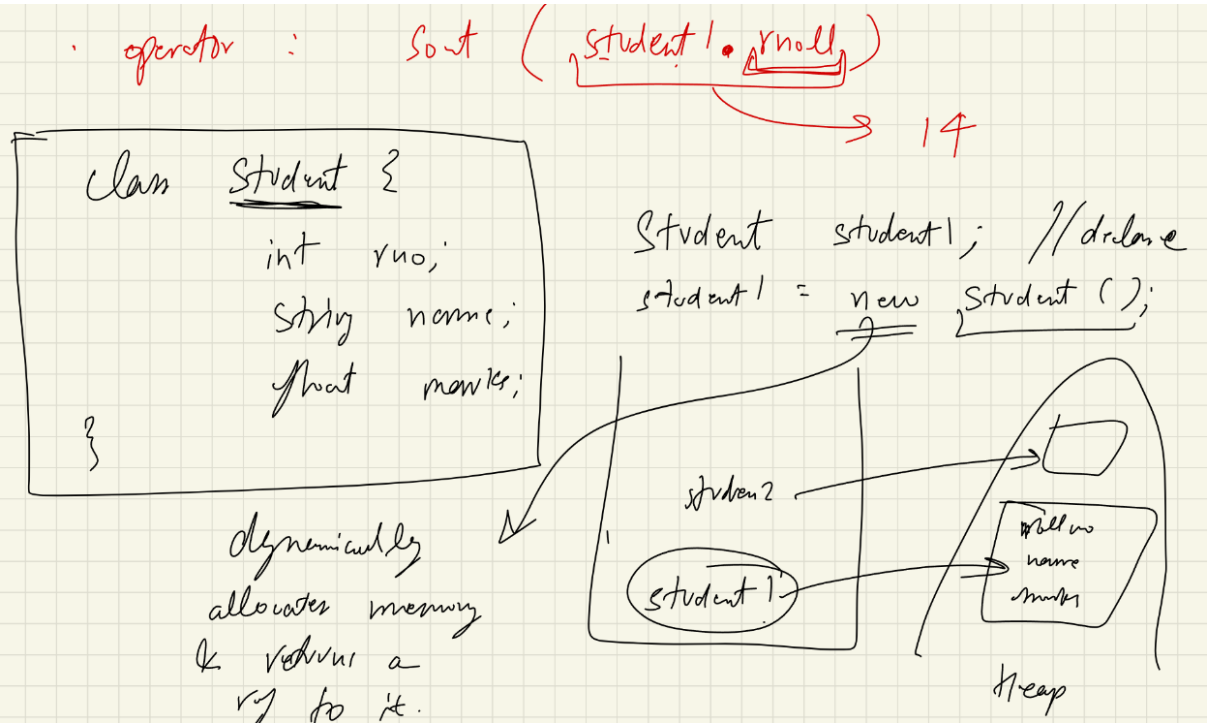
Class and Object declare

Declaring a Class

```
// create a class

Class Student {

int rno;   //rno;
int marks; //Marks
String name;  // Name
```

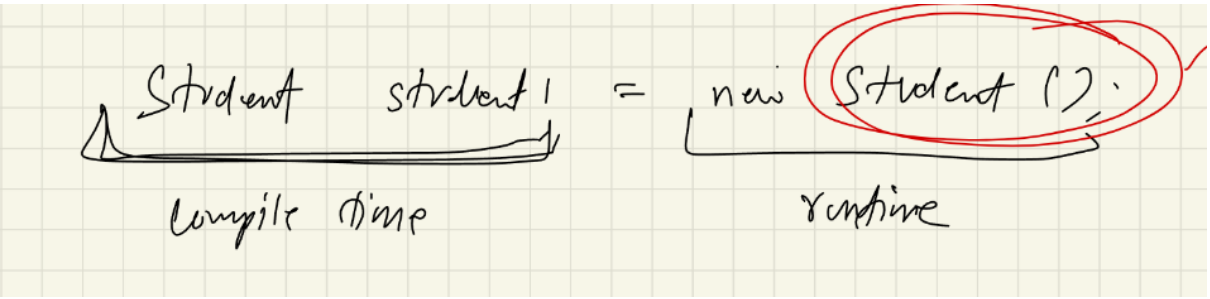
Declaring Object

```
Student student1 = new Student();
```



- New Word is create a Dynamically allocate memory & Return a reference to it

Runtime and compile Time



Complie Time

- Compile time refers to the period during which a program is translated from human-readable source code into machine-readable instructions by a compiler.

- Compile time Excute code Only initialize stack Memory

Run Time

- During runtime, variables are allocated memory, objects are created, and functions/methods are executed based on the flow of the program.
- Java programming Dynamically create object in Run time.

Manuplate the object and classes

```
public class Oops{

      static class Student{

        int rno;
        float  marks;
        String name;
      }
      public static void main(String[] args) {
          Student student1 = new Student();

          System.out.println(student1.rno);
          System.out.println(student1.marks);
          System.out.println(student1.name);
      }
}
```

Constructor

- Constructor is special type of function But Run you create a object and allows the same

  data type variables
- Constructor is same name of Class
- Immediately called object is created

- Constructor initializing the Values
- Constructor No return Type
- Constructor Value No modified.
- The constructor is a useful special type of function responsible for handling object properties' values and ensuring that they cannot be modified from outside the class.
- Must use this keyword

Advantage of constructor

1. **Initialization**: Constructors initialize object attributes, ensuring a defined starting state.
2. **Encapsulation**: They enforce encapsulation by controlling attribute initialization.
3. **Convenience**: Simplify object creation by providing a standardized process.
4. **Default Values**: Offer default values for attributes, enhancing flexibility.
5. **Inheritance and Polymorphism**: Play a key role in inheritance and polymorphism, enabling subclass object creation and specialization.

Types of Constructor

- Default Constructor
- Parameterized Constructor
- Copy Constructor

this Keyword

- The `this` keyword in Java refers to the current object.

- It is typically used to distinguish between instance variables and local variables, to invoke methods or constructors of the current object, or to pass the current object as an argument.
- However, it cannot be used in static methods.
- Understanding its usage is crucial for maintaining clarity and avoiding ambiguity in Java code.

```
public class Oops{

      static class Student{

        int rno;
        float   marks;
        String name;

        Student () {
        // Default values
        this.rno = 10;        // this word reference pointing
        this.marks = 20.0f;
        this.name = "gokil";
        }


    }

      public static void main(String[] args) {
        Student student1 = new Student();
        Student Rahul = new Student();
          System.out.println(student1.rno);
          System.out.println(student1.marks);
          System.out.println(student1.name);
      }
}
```

constructor Value modified

- Constructor only modified passing arugments
- Constructor using Modified Every Object Values using the parameters /Arugments values

```
public class Oops {

    static class Student {

        int rno;
        float marks;
        String name;

        // Constructor with parameters to modify values
        public Student(int rno, float marks, String name) {
            this.rno = rno;
            this.marks = marks;
            this.name = name;
        }
    }

    public static void main(String[] args) {
        // Creating a Student object with specific values usi
        Student student1 = new Student(10, 20.0f, "gokil");

        // Outputting the values of student1
        System.out.println(student1.rno);
        System.out.println(student1.marks);
        System.out.println(student1.name);
    }
}
```

Constructor Overloading

- Constructor overloading is indeed an important concept in polymorphism.

- In object-oriented principles, constructors do indeed consider parameters/arguments (values).
- A constructor with no arguments is called a default constructor.

```
public class Oops {

    public static class Student{

        int rno;
        float marks;
        String name;

    Student() {   //  constructor consider calling the size o

        this.rno=10;
        this.marks=20.0f;
        this.name="gokil";
    }
    Student(int rollno,float marks,String Name) // constructo
    {
        this.rno=rollno;
        this.marks=marks;
        this.name=Name;
    }
}
    public static void main(String[] args) {
         Student gokil = new Student(20,10.0f,"Rahul");

        Student Rahul = new Student();
        System.out.println(Rahul.marks);

    }


}
```

calling construct to another constructor

- call one constructor from another constructor in the same class using the `this()` keyword.
- This is useful when you have multiple constructors in a class and want to avoid code duplication by reusing common initialization logic.

1. `this()` **Keyword**:
   - Use `this()` to call another constructor from within a constructor.
   - It must be the first statement in the constructor.
2. **Avoiding Redundancy**:
   - Helps in avoiding repetition of initialization logic.
   - Centralizes common initialization steps.
3. **Parameter Passing**:
   - Arguments can be passed to the constructor being called using `this()`.
   - Ensure argument types match the parameter list of the target constructor.
4. **Code Readability**:
   - Enhances code readability by providing a clear flow of initialization logic.

```
public class Oops{

 public static class Student{

    int rno;
    String name;

    Student(int rollno,String Name)
    {
      rno =rollno;
```

```
      name=Name;
    }
    Student()
    {
        this(12,"gokil");
    }
 }
   public static void main(String [] args)
   {
      Student cs = new Student();

      System.out.println(cs.name);

   }


}
```

Copy Constructor

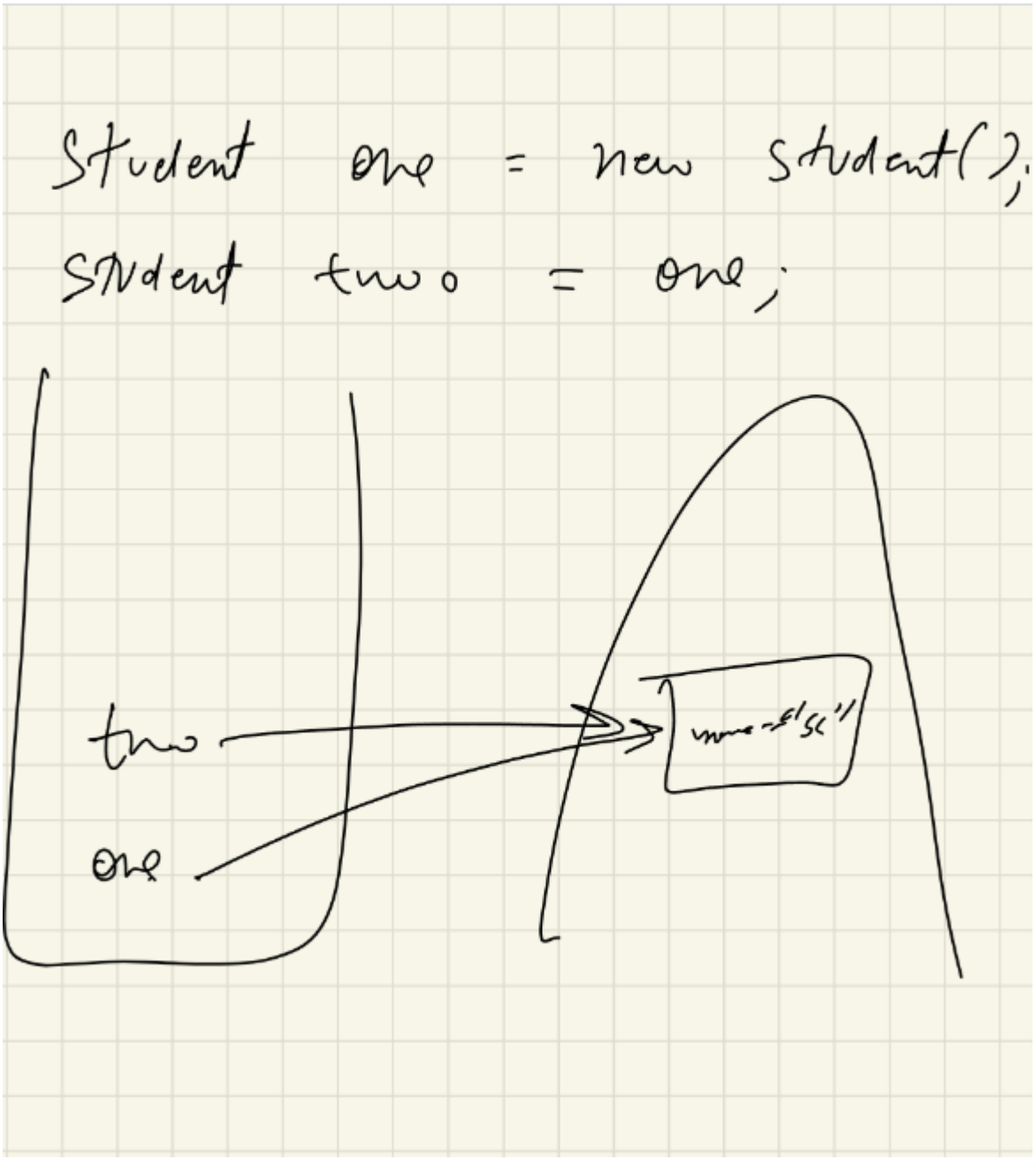
- java copy constructor returns a copy of the specified object by taking the existing object as an argument.
- To create a copy constructor, we need to take the existing object as an argument and initialize the values of instance variables with the values obtained in the object.

```
class Person {
    String name;

    // Copy constructor
    public Person(Person other) {
        this.name = other.name;
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        Person person1 = new Person();
        person1.name = "John";

        // Using the copy constructor to create a new object
        Person person2 = new Person(person1);

        System.out.println("Person 1: " + person1.name);
        System.out.println("Person 2: " + person2.name);
    }
}
```

Memory Allociation New Key Word

- New word also an Objects and stored in Heap Memory
- Heap mermory Object point to another Reference Variable



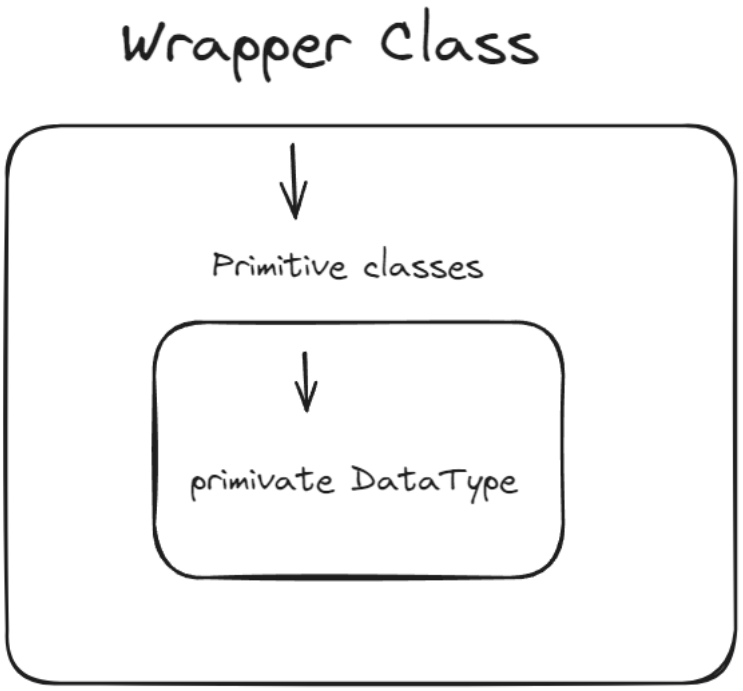
Wrapper Classes

- wrapper classes are used to convert primitive data types into objects, as Java is an object-oriented programming language where everything is treated as an object.
- Wrapper classes is immutable

- wrapper classes in scenarios where objects are required, such as collections (ArrayList, HashMap) which can't directly store primitive types.

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

- Wrapper classes contains internal side



Arritubutes of Wrapper Classes

```
Integer i = Integer.valueOf(value); // Boxing, Wrapping

int j = i.int Value(i) // unboxing

Integer i3 = 30; // auto boxing
int k = i2;

int i = Integer.parseInt(varible);
```

Final Key word

- When a non primitive is final, you cannot reassign cannot the value.
- primitive data type final key word you can’t Modified.
- The `final` keyword in Java is used to declare constants, immutable variables, and prevent method overriding or subclassing.
- It makes variables unmodifiable after initialization, methods unoverridable, and classes unextendable. It ensures code robustness, improves performance, and clarifies intent by signaling immutability or constant values.

Garbage Collections

- If there is an object without reference variable then object will be destroyed by “Garbage Collection

Java Packages & API

A package in Java is used to group related classes. Think of it as **a folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

Built in packages

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more. The complete list can be found at Oracles website: https://docs.oracle.com/javase/8/docs/api/.

User-defined Packages

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

Example

```
└── root
  └── mypack
    └── MyPackageClass.java
```

To create a package, use the `package` keyword:

Static Keyword

Static method using an referencing without object

Static Variables: Variables that belong to the class itself, rather than any specific instance of the class. They are shared among all instances of the class.

- **Definition**: Shared variables for all objects of a class.
- **Example**:

```
public class Example {
    static int count = 0; // Static variable
}
```

```
public class Human {
    int age;
    String name;
    int salary;
    boolean married;
    static long population;  // static not referncing an obje

    static void message() {
        System.out.println("Hello world");
//        System.out.println(this.age); // cant use this over
    }

    public Human(int age, String name, int salary, boolean ma
        this.age = age;
        this.name = name;
        this.salary = salary;
        this.married = married;
        Human.population += 1; // class name give the static
    }
}
```

Static Methods: Methods that belong to the class rather than any instance of the class. They can be called without creating an instance of the class.

- **Definition**: Methods that belong to the class, not objects.
- **Example**:

```
public class Example {
    static void display() { // Static method
        System.out.println("Hello, World!");
    }
}
```

```
package com.kunal.staticExample;

public class Main {
    public static void main(String[] args) {
//        Human kunal = new Human(22, "Kunal", 10000, false);
//        Human rahul = new Human(34, "Rahul", 15000, true);
//        Human arpit = new Human(34, "arpit", 15000, true);
//
//        System.out.println(Human.population);
//        System.out.println(Human.population);
//        System.out.println(Human.population);

        Main funn = new Main();
        funn.fun2();

    }

    // this is not dependent on objects
    static void fun() {
//        greeting(); // you cant use this because it require
        // but the function you are using it in does not depe

        // you cannot access non static stuff without referen
        // a static context

        // hence, here I am referencing it
        Main obj = new Main();
```

```
        obj.greeting();
    }

    void fun2() {
        greeting();
    }

    // we know that something which is not static, belongs to
    void greeting() {
//        fun();
        System.out.println("Hello world");
    }
}
```

Static Block: A block of code enclosed in curly braces `{}` inside a class, preceded by the `static` keyword. It is executed when the class is loaded into memory.

- **Definition**: Special code block executed when class is loaded.
- **Example**:

```
public class Example {
    static {
        // Static block
        System.out.println("Static block initialized.");
    }
}
```

```
package com.kunal.staticExample;

// this is a demo to show initialisation of static variables
public class StaticBlock {
    static int a = 4;
    static int b;

    // will only run once, when the first obj is create i.e.
```

```
    static {
        System.out.println("I am in static block");
        b = a * 5;
    }

    public static void main(String[] args) {
        StaticBlock obj = new StaticBlock();
        System.out.println(StaticBlock.a + " " + StaticBlock.

        StaticBlock.b += 3;

        System.out.println(StaticBlock.a + " " + StaticBlock.

        StaticBlock obj2 = new StaticBlock();
        System.out.println(StaticBlock.a + " " + StaticBlock.
    }

}
```

Static Nested Classes: Classes that are declared inside another class and marked as static. They can be accessed without instantiating the outer class.

- **Definition**: Nested class declared inside another class.
- **Example**:

```
public class Outer {
    static class Nested {
        void display() {
            System.out.println("Nested class method");
        }
    }
}
```

```
package com.kunal.staticExample;

import java.util.Arrays;
```

```
public class InnerClasses {

    static class Test {
        String name;
        public Test(String name) {
            this.name = name;
        }

        @Override
        public String toString() {
            return name;
        }
    }

    public static void main(String[] args) {
        Test a = new Test("Kunal");
        Test b = new Test("Rahul");

        System.out.println(a);

//        System.out.println(a.name);
//        System.out.println(b.name);
    }
}

//static class A {
//
//}
```

Singleton Classes

In object-oriented programming, a java singleton class is a class that can have only one object (an instance of the class) at a time. After the first time, if we try to instantiate the Java Singleton classes, the new variable also points to the first instance created.

1. Make a constructor private.
2. Write a static method that has the return type object of this singleton class. Here, the concept of Lazy initialization is used to write this static method.

```
package com.kunal.singleton;

import com.kunal.access.A;

public class Singleton {
    private Singleton () {

    }

    private static Singleton instance;

    public static Singleton getInstance() {
        // check whether 1 obj only is created or not
        if (instance == null) {
            instance = new Singleton();
        }

        return instance;
    }

}
```

```
package com.kunal.singleton;

import com.kunal.access.A;

public class Main {
    public static void main(String[] args) {
        Singleton obj1 = Singleton.getInstance();

        Singleton obj2 = Singleton.getInstance();
```

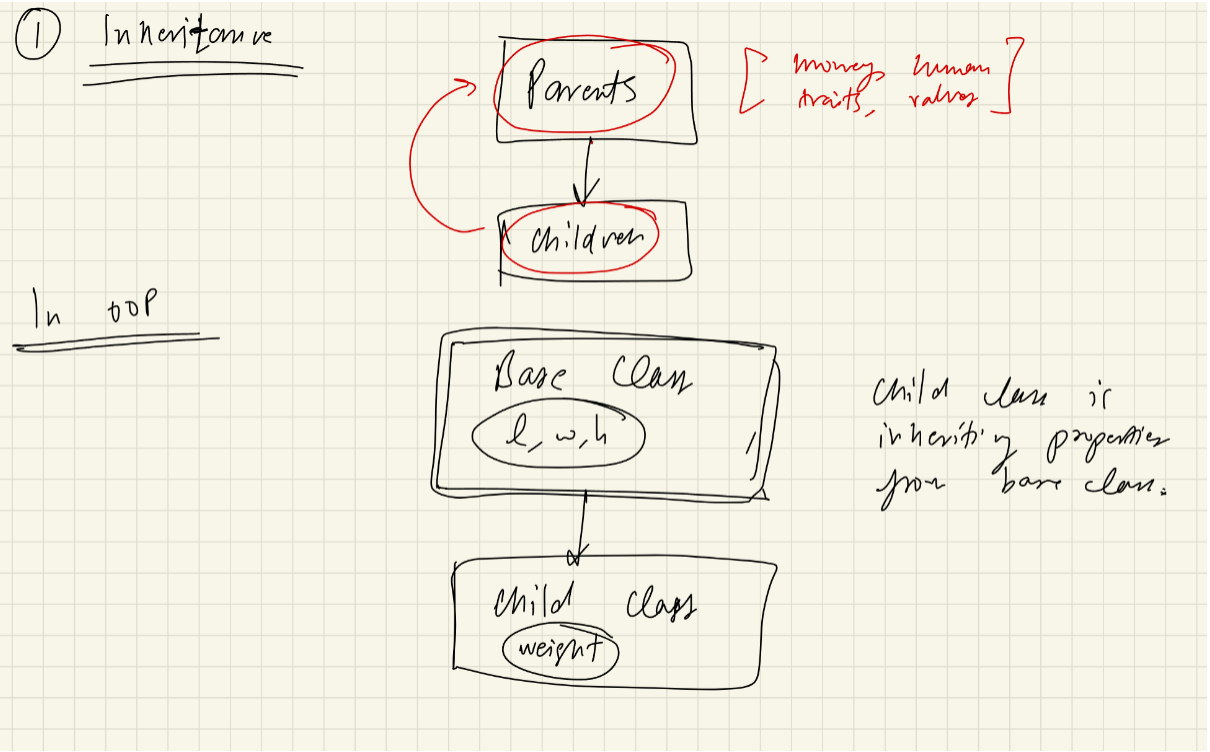
```
        Singleton obj3 = Singleton.getInstance();

        // all 3 ref variables are pointing to just one objec

        A a = new A(10, "Kunal");
        a.getNum();
//        int n = a.num;
    }
}
```

Principle of oops

- Inheritance
- polymorphism
- Encapsulation
- Abstraction

Inheritance

- Inheritance is an important pillar of OOP(Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class
- Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition.
- In heritance is base classes inherit to derived class you can also create a new properties on derived classes and also used existing base class properties .



Syntax

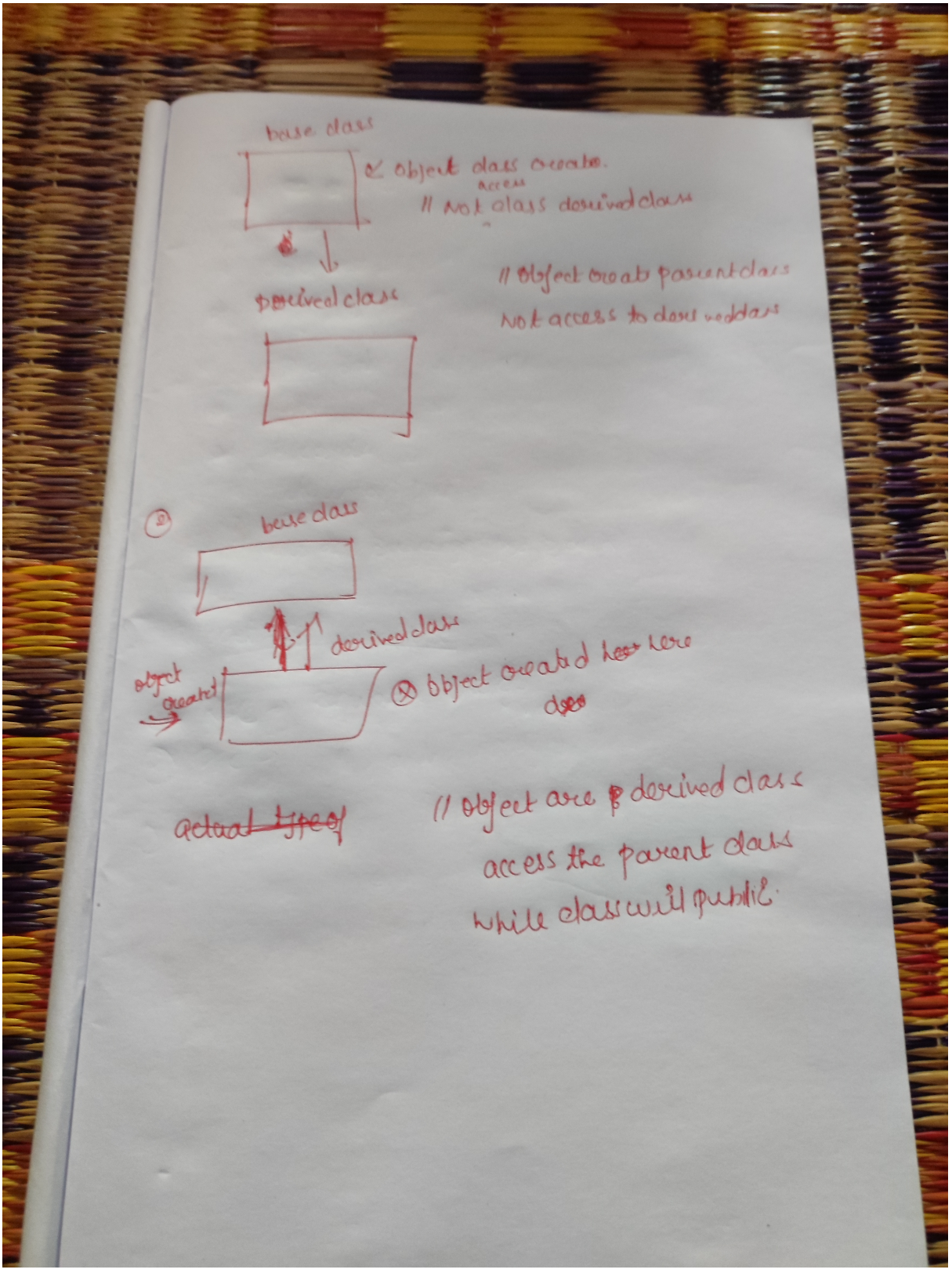
```
class derived-class extends base-class
{
   //methods and fields
}
```

```
public class Oops{


  static class Department{
```

```
    String college;
    String  Department;
    int student_ls;
    int Staff_ls;

    Department()
    {
       college =" IIT Madras";
       Department = "Computer Science";
       Staff_ls=20;
       student_ls=100;
    }
}
  public static class Student extends Department{

    String Mentor_name;
    int  student_rollno;
    int semster;

    Student()
    {
       Mentor_name="ram";
       student_rollno = 01;
       semster = 01;
    }

  }

    public static void main(String[] args) {
        Student student = new Student();

        System.out.println(student.Mentor_name);
        System.out.println(student.Department);
    }
```

```
}
```

Access to object



- In object-oriented programming (OOP), when you create an object of a parent class, you can't directly access the members or methods of derived

classes. The reason is that the object created is of the parent class type, so it only contains the attributes and methods defined in the parent class.

- However, if the methods or attributes in the derived classes are accessible through inheritance (i.e., they are not private or protected), you can access them by creating an object of the derived class and then accessing those members or methods through that object.

```
    Box box5 = new BoxWeight(2, 3, 4, 8);
       System.out.println(box5.w);  // you can inherit devie
                                    // constructor automatic

       // there are many variables in both parent and child
       // you are given access to variables that are in the
       // hence, you should have access to weight variable
       // this also means, that the ones you are trying to a
       // but here, when the obj itself is of type parent cl
       // this is why error
       BoxWeight box6 = new Box(2, 3, 4);
       System.out.println(box6);  // you can  inherit base
                                    // constructor automatic
```

Super class

- Super is reference to the parent class constructor .
- Super classes is always reference to the parent constructors
- `this` keyword: Refers to the current instance of the class, used to access instance variables or methods of the current object.
- `super` keyword: Refers to the parent class, used to call the parent class's constructor or methods from the subclass.

superclass methods and variables:

The `super` keyword can be used to access methods and variables of the superclass from within the subclass. This is particularly useful when the subclass overrides a method of the superclass but still needs to call the superclass's version of that method.

```
class Vehicle {
    int maxSpeed = 120;
}

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

    void display()
    {

        System.out.println("Maximum Speed: "
                           + super.maxSpeed); // calling constru
                                  // print value maxspeed is 120
    }
}

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

Calling the superclass constructor:

When you create an object of a subclass, the constructor of the superclass is called implicitly before the constructor of the subclass. However, you can explicitly call a specific constructor of the superclass using the `super()` keyword. This is useful when the superclass has multiple constructors, and you want to call a specific one

```
class Superclass {
    void display() {
        System.out.println("Superclass method");
    }
}

class Subclass extends Superclass {
    @Override
    void display() {
        super.display(); // Calls the superclass method
        System.out.println("Subclass method");
    }
}

public class Main {
    public static void main(String[] args) {
        Subclass obj = new Subclass();
        obj.display();
    }
}

Output code

Superclass constructor
Subclass constructor
```

Types of Inheritance

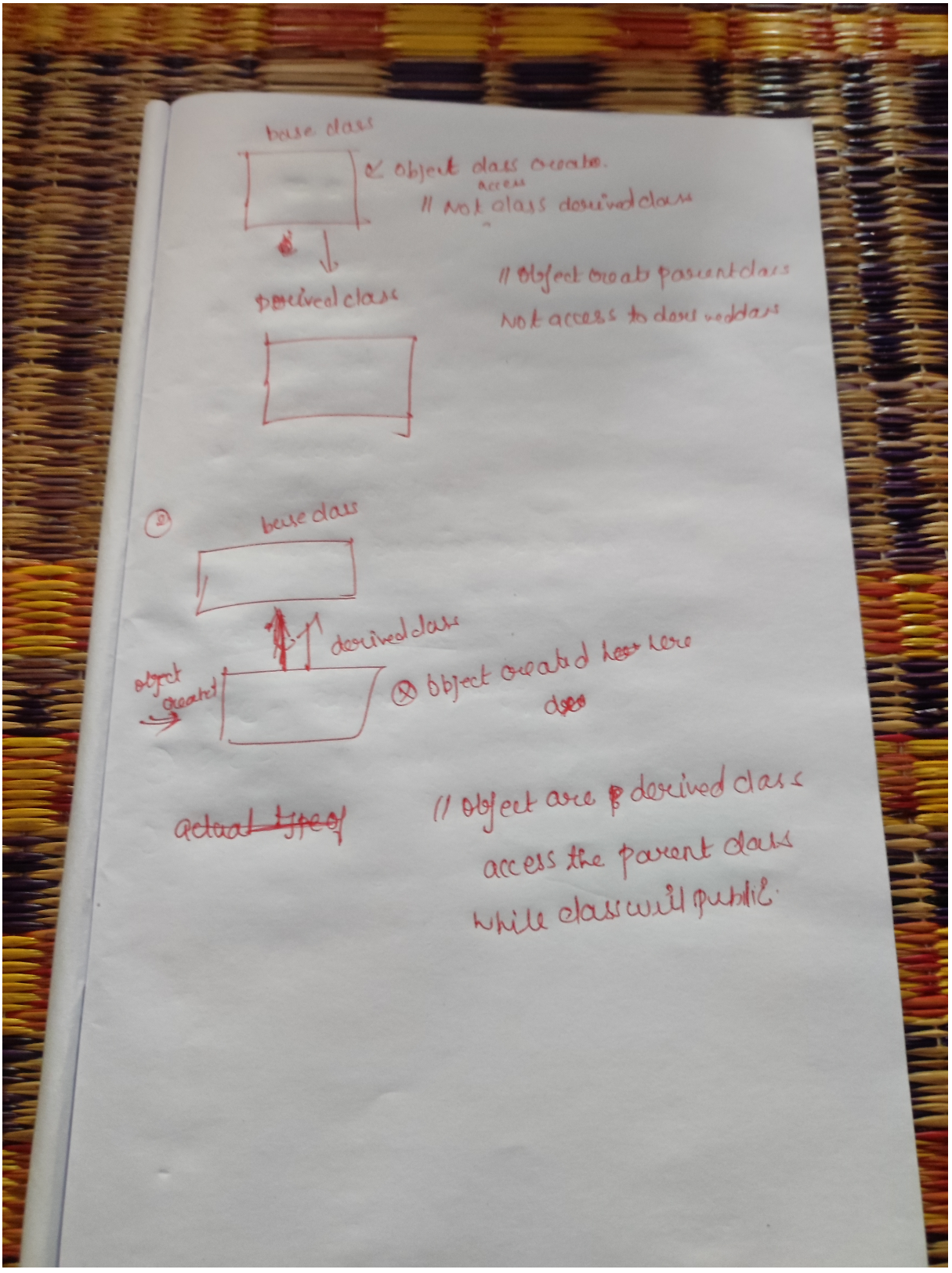
- Single inheritance
- Multiple inheritance
- Multilevel inheritance
- Hierarchical inheritance
- Hybrid inheritance

Single Inheritance

Single Inheritance is one classes extends from another classes

In the oops concept create object dervied classes to base classes

Example



base class
object class create.
// Not class derived class access

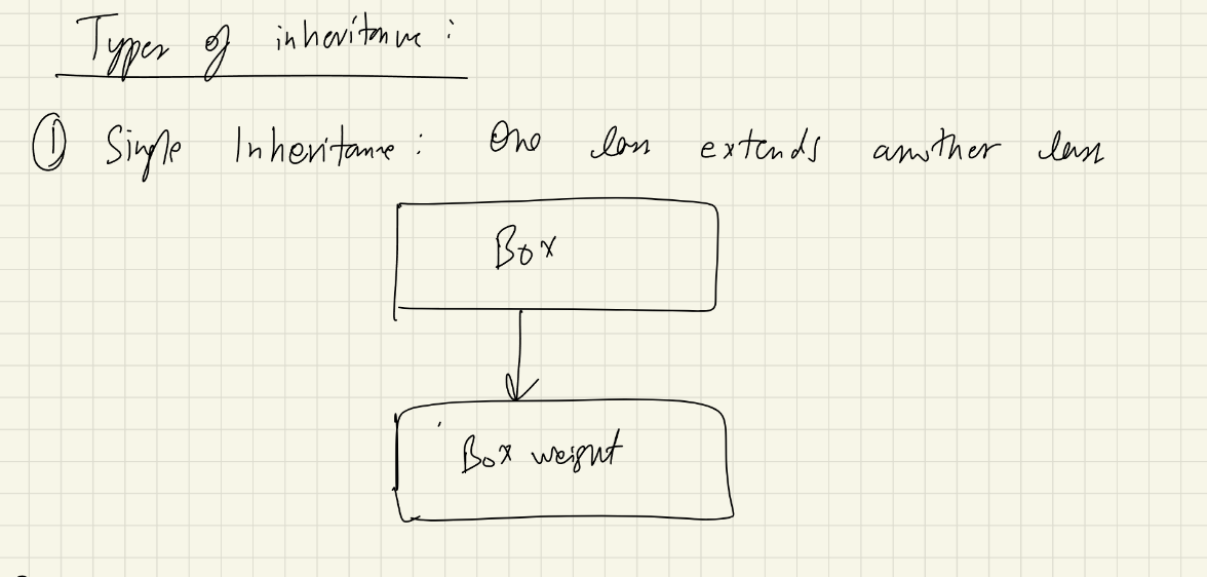
// Object create parent class
Not access to derived class

(2)

derived class

// Object create derived class
access the parent class
while class will public.

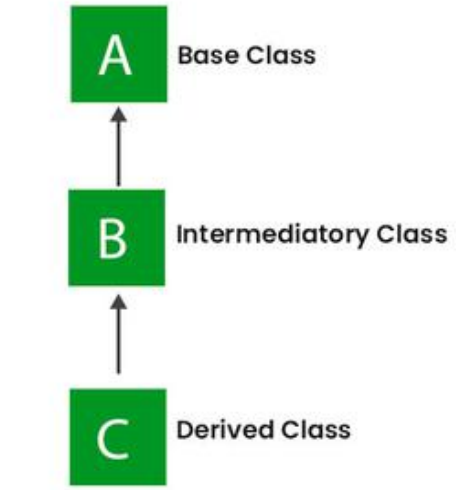
```
class derived-class extends base-class
{
   //methods and fields
}
```



multi level inheritance

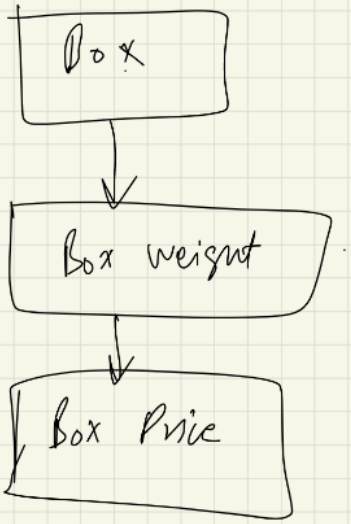
- In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes.
- In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.

Visual Represtation



Multilevel Inheritance

(2) multilevel Inheritance :




```
// Java program to illustrate the
// concept of Multilevel inheritance
import java.io.*;
import java.lang.*;
```

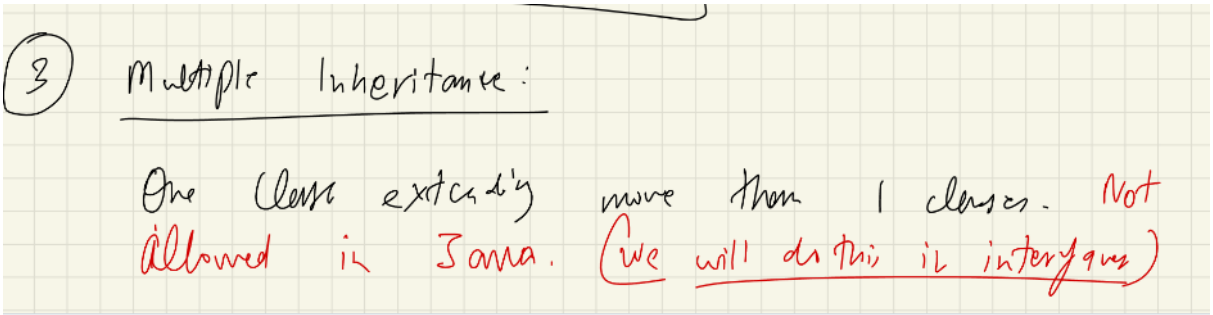
```
import java.util.*;

class One {
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}

class Two extends One {
    public void print_for() { System.out.println("for"); }
}

class Three extends Two {
    public void print_geek()
    {
        System.out.println("Geeks");
    }
}

// Drived class
public class Main {
    public static void main(String[] args)
    {
        Three g = new Three();
        g.print_geek();
        g.print_for();
        g.print_geek();
    }
}
```

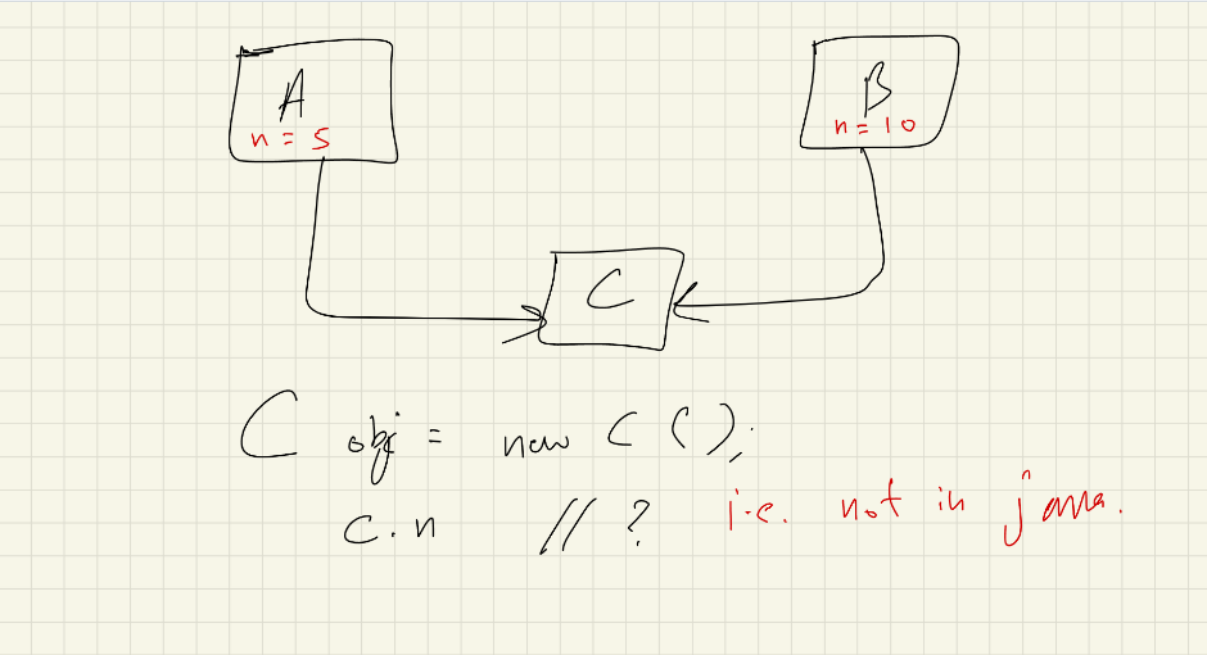
Multiple Inheritance

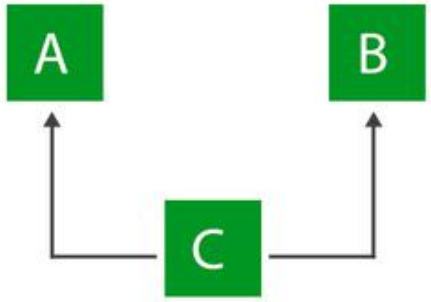
In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes.

Java does not support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces. In the image below, Class

C is derived from interfaces A and B.



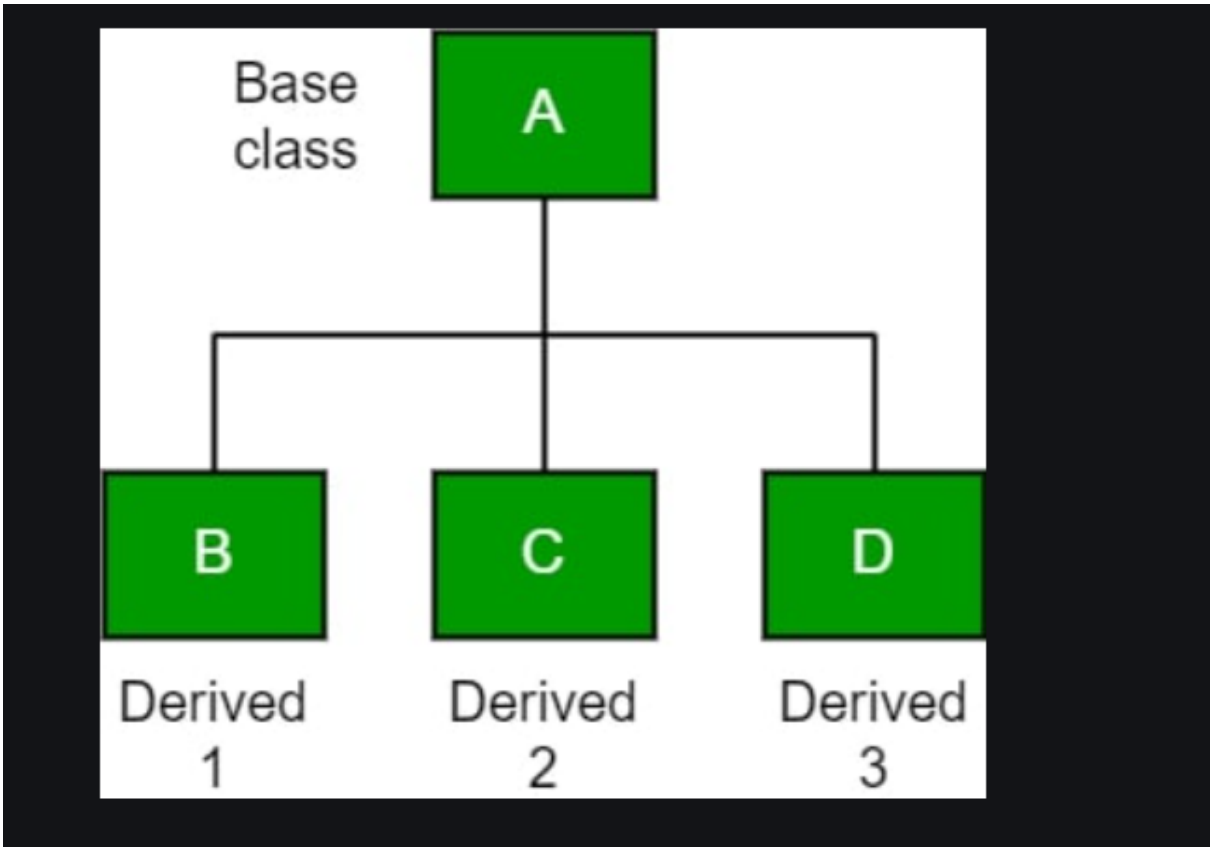




Multiple Inheritance

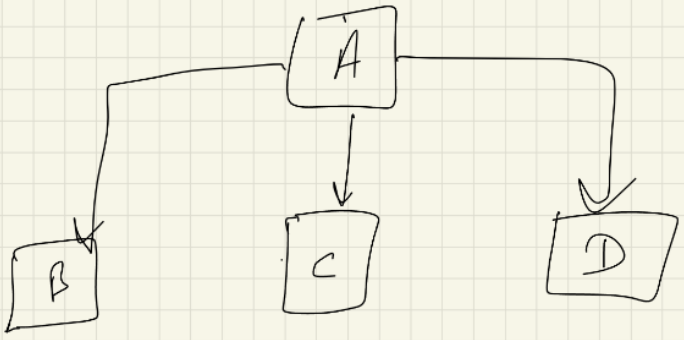
Hierarchical Inheritance

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



(4) Hierarchial Inheritance :

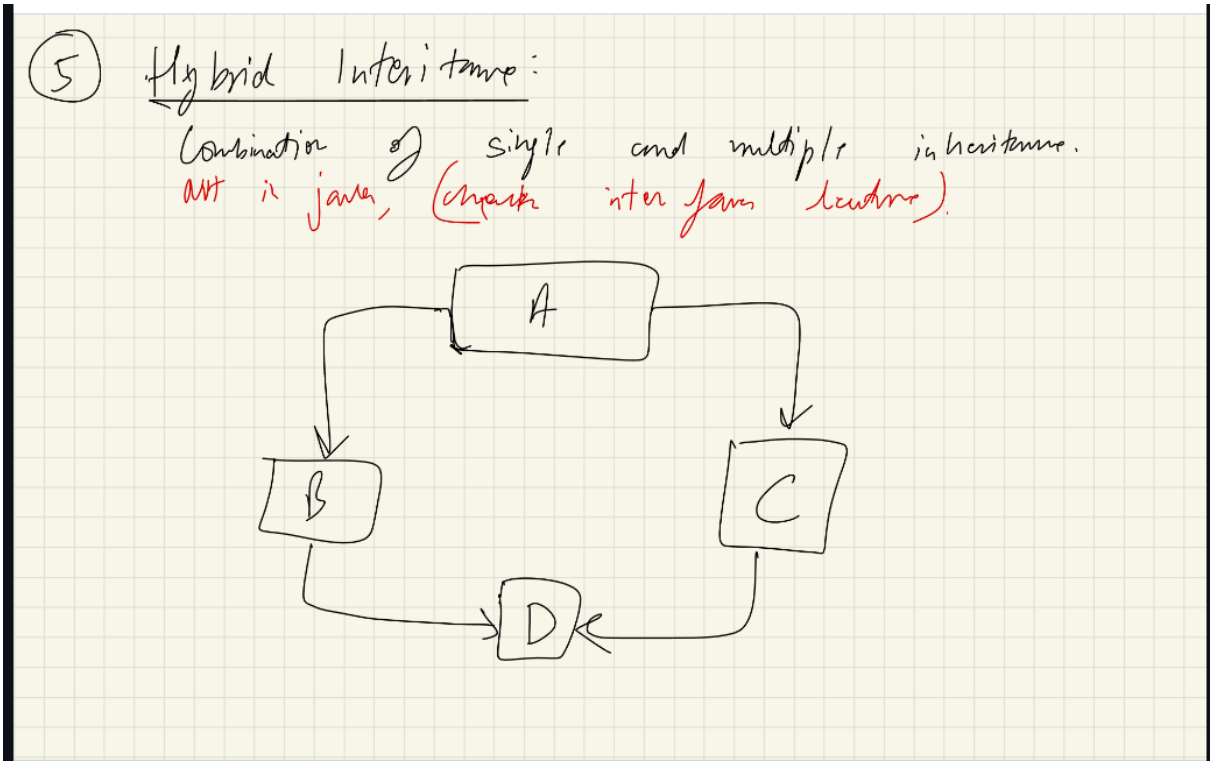
One class is inherited by many classes.



```
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
}}

// it working
```

Hybrid inheritance

- Hybird Inheritance combination of single and Multiple inheritance
- Hybird Inheritance is not access java but only use in interfaces

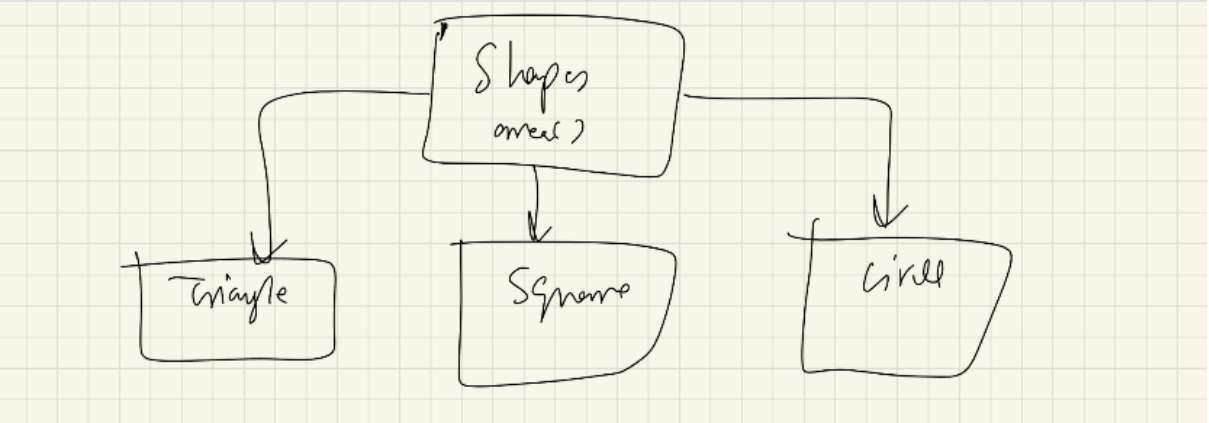


Ploymorphsim

> Poly Means Many

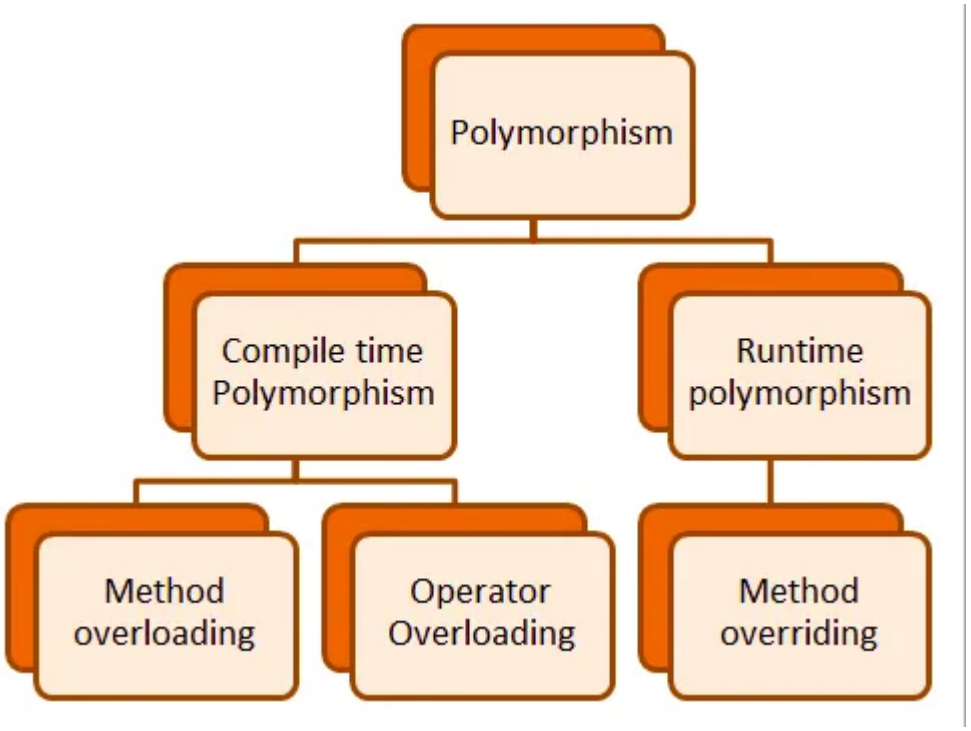
> morphism ⇒ ways to respresent

Syntax of ploymorphism



Types of Polymorphism

- Compile Time polymorphism
- Static polymorphism



Compile time Polymorphism

- Compile Time polymorphism / Static polymorphism Achived via Method Overloading
- Same name methods, Constructor but types Arugmuents,returntypes ordering can be different

Eg: Multiple constructor

```
class Helper {

    // Method with 2 integer parameters
    static int Multiply(int a, int b)
    {
        // Returns product of integer numbers
        return a * b;
    }

    // Method 2
    // With same name but with 2 double parameters
    static double Multiply(double a, double b)
    {
```

```
        // Returns product of double numbers
        return a * b;
    }
}

// Class 2
// Main class
class GFG {
    // Main driver method
    public static void main(String[] args)
    {
        // Ploymorphsim Calling method by passing

        System.out.println(Helper.Multiply(2, 4));
        System.out.println(Helper.Multiply(5.5, 6.3));
    }
}
```

Dynamic polymorphism

Dynamic polymorphism or Runtime Polymorphism

```


// Base Class
class Parent {
    void show() //same method name with  one to access the va

{ System.out.println("Parent's show()"); }
}

// Inherited class
class Child extends Parent {
```

```
 void show()   //same method name with  one to access the var
    {
        System.out.println("Child's show()");
    }
}

// Driver class
class Main {
    public static void main(String[] args)
    {
        Parent obj1 = new Parent();
        obj1.show();
parent object = new child();
   object.show();
//refercing varible on parent class but created
// child classes object is created working only
// Java run a method in referece in child classes only not a
    }
}
```

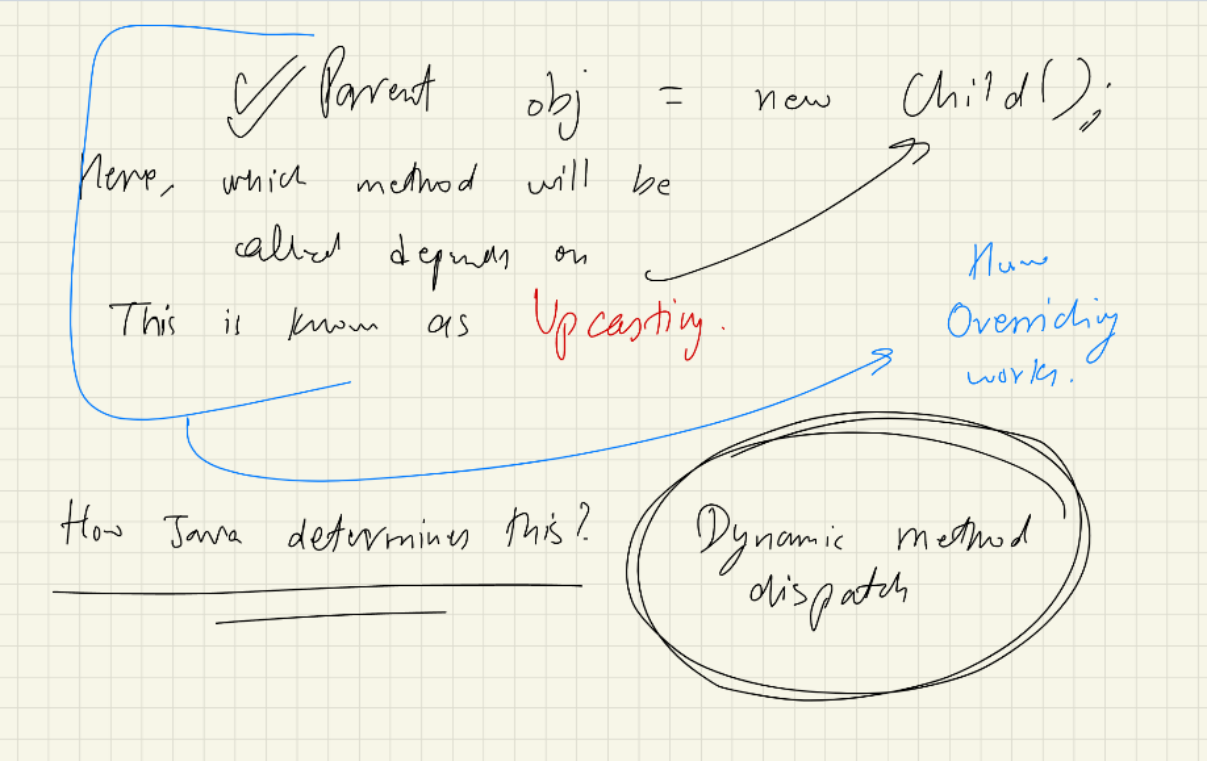
Overiding

- "Overriding involves the use of polymorphism, wherein subclasses inherit methods from their base classes. However, a main disadvantage of inheritance arises when a user calls a method with the same name, resulting in the execution of the method from the parent class."

1. **Overriding and Polymorphism**:
    - Overriding: Subclasses can redefine methods inherited from their parent classes.
    - Polymorphism: Allows a single method call to behave differently based on the object's type.
2. **Disadvantage of Inheritance**:
    - Inheritance: Subclasses inherit properties and behaviors from their parent classes.

- Drawback: Method conflicts arise when a method with the same name exists in both the parent and child classes.

3. **Execution of Parent Class Method**:
   - When a method is called on a subclass object with the same name as a method in the parent class:
   - The parent class method is executed instead of the overridden method in the subclass.
   - This behavior may cause unexpected results, deviating from the intended subclass functionality.



Early Binding and Late binding

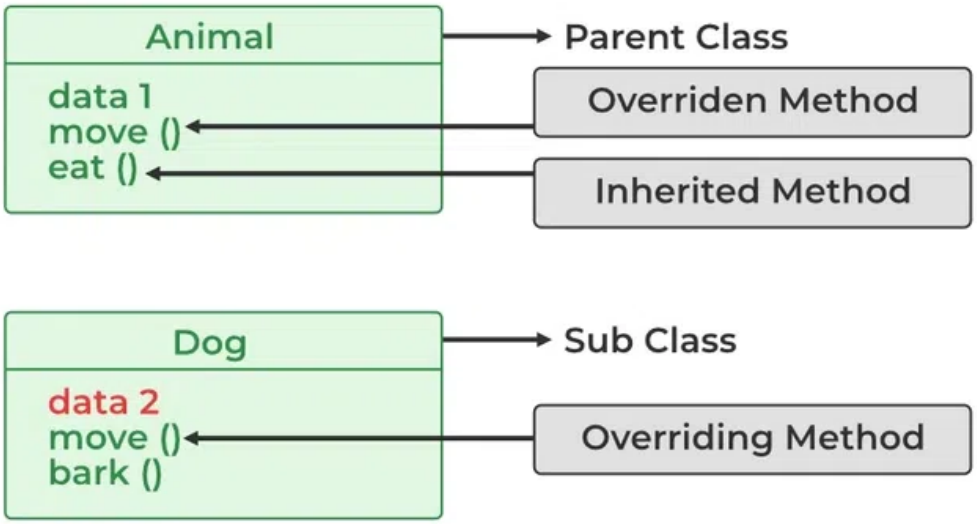
1. **Early Binding**:
   - In early binding, the association between a method call and the method implementation is resolved at compile time.

- This means that the compiler knows exactly which method will be called and can directly link the method call to its implementation.
- Early binding is usually more efficient in terms of performance because the method resolution is done once, during compilation.
- However, it lacks flexibility because it requires that the method to be called is known at compile time.

2. **Late Binding**:
    - In late binding, also known as dynamic binding or runtime binding, the association between a method call and the method implementation is resolved at runtime.
    - This means that the decision about which method to call is deferred until the program is actually running.
    - Late binding allows for greater flexibility because it enables things like polymorphism, where a single method call can behave differently depending on the actual object being referred to.
    - However, late binding typically incurs a performance overhead because the method resolution needs to be done each time the method is called at runtime.

Final Keyword

- Final Keyword prevent overriding at Runtime .
- The final keyword always prevents the overriding of classes and methods. For example, inheriting the parent class with the final keyword will not work because the final keyword does not allow it.
- Using the static keyword prevents the overriding of values because static does not depend on objects. A static method is referenced using the class name or a reference variable; you cannot access it by calling on objects.
- Static methods do not depend on objects, so they cannot be overridden based on objects. Therefore, static methods cannot be overridden.



Encapsulation and Abstraction

- Abstraction solves design-level issues, while Encapsulation addresses implementation issues by hiding the code within a single combined and protected form from the outside world.
- Abstraction focuses on the external level of implementation, whereas Encapsulation concentrates on the internal level of implementation.
- Abstraction involves gaining information, while Encapsulation involves containing information at a certain level.
- Abstraction works with abstract classes and interfaces, whereas Encapsulation handles access modifiers and methods."

Access Modifier’s

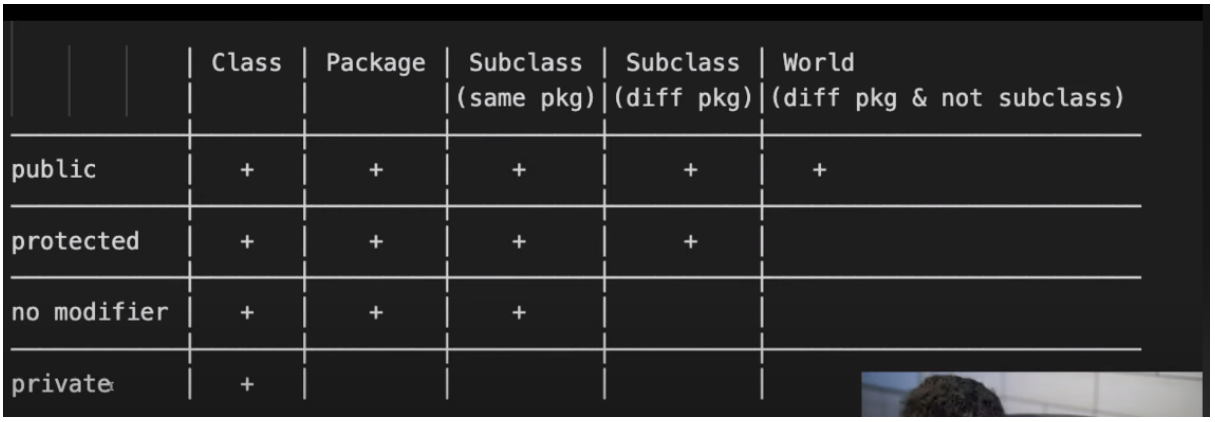
default package not Access out side the packages.

public package is Access in every Where

private package only access inside the class and methods

protected is access classes, packages, Sub class Package in same pkg, Sub class Package in different package

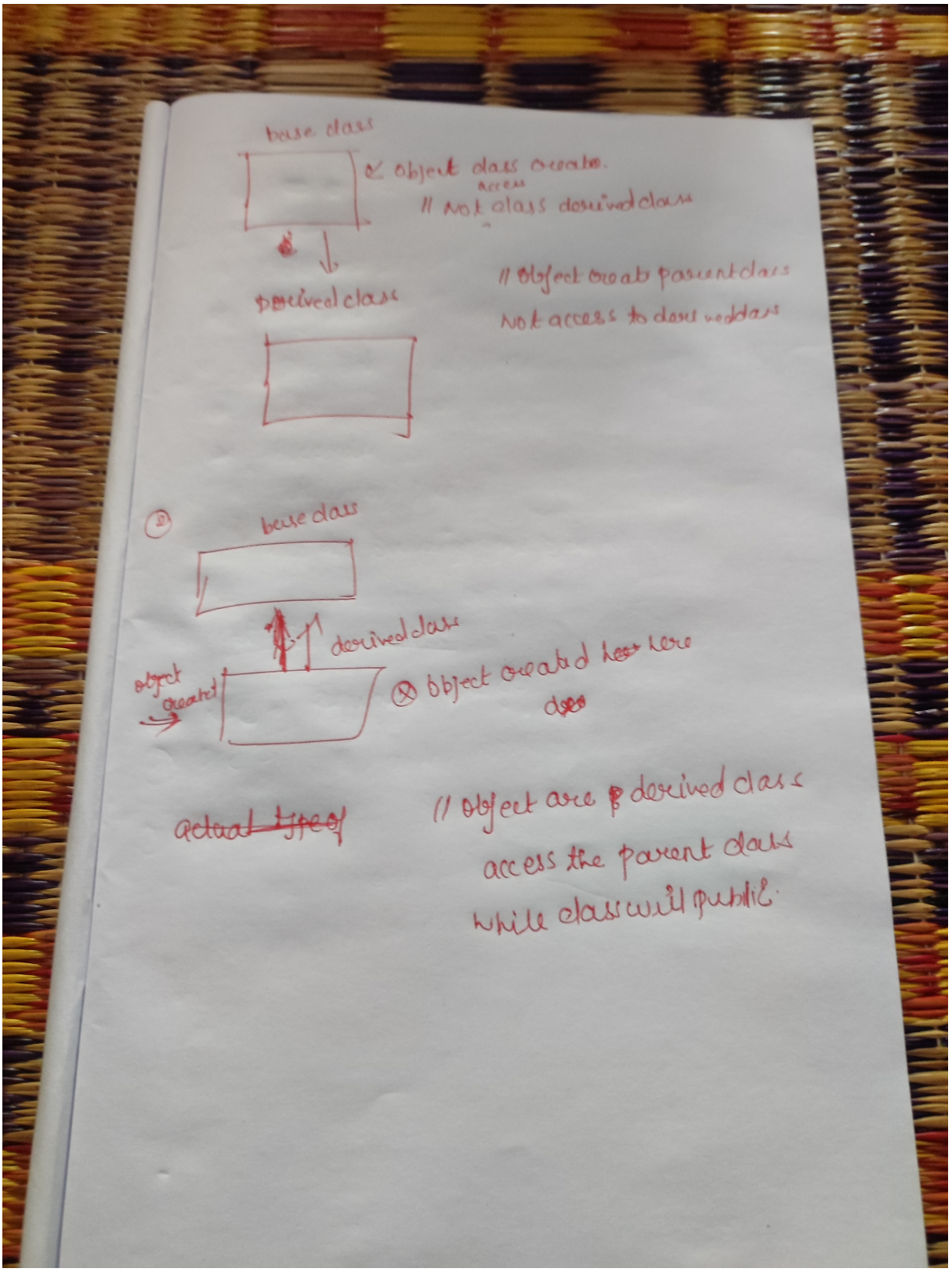
Rules for Access Modifier’s



	Class	Package	Subclass (same pkg)	Subclass (diff pkg)	World (diff pkg & not subclass)
public	+	+	+	+	+
protected	+	+	+	+	
no modifier	+	+	+		
private	+				

When to use the Access modifier’s

- "Private" holds sensitive data and should not be accessed directly within the same class, using only getter and setter methods.
- Getter and setter methods must be declared as "Public."
- "No modifier" signifies the default access level, allowing access only within the same package or class, but not outside of it.
- "Protected" access modifier allows access within the class, its package, and subclasses (inheritance), but not outside of the package or class hierarchy.
- "Protected" is primarily used for inheriting classes in subclasses, where subclasses must only access variable data.
- "Public" allows access from anywhere and everywhere.
- In protected Only you access inherit the Subclasses Not know the parent classes
- Simple protected access only Subclasses only.


base class
object class create.
// Not access class derived class
derived class
// Object create parent class
Not access to derived class
base class
derived class
object created
Object created here
does
actual type
// Object are derived class
access the parent class
while class will public.

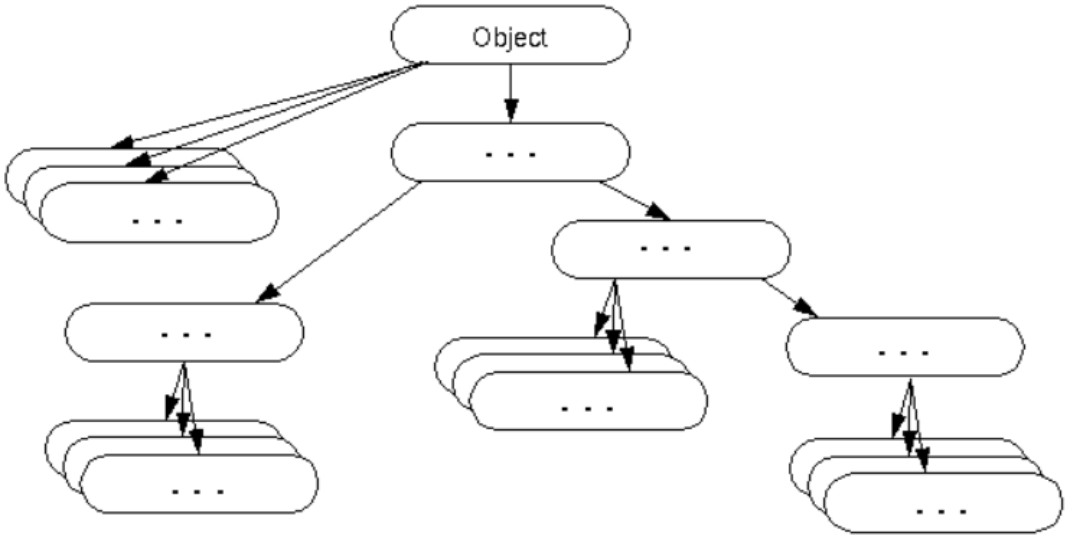
- Protected Key word also refers to the base classes

Common Packages

- lang: Language - Java programming language core package containing fundamental classes and interfaces.
- io: Input/Output - Java package for input and output operations, including file handling.
- util: Utility - Java package containing various utility classes and data structures.
- applet: Application let - A small application program that runs within a larger application or web browser.
- awt: Abstract Window Toolkit - Java package for creating graphical user interfaces (GUIs).
- net: Network - Java package for networking functionality, including client-server communicati

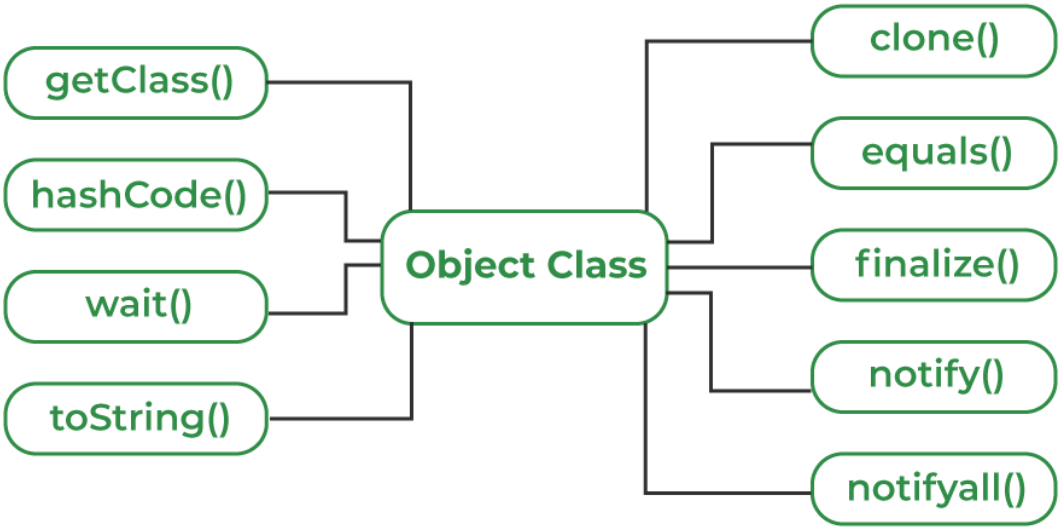
Object classes

- The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
- The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.



Object class Methods

- tostring() method
- hashCode() method
- equals(Object obj) method
- finalize() method
- getClass() method



1. `toString()`

Definition:

The `toString()` method is a method in Java that returns a string representation of an object. This method is automatically called when you try to concatenate an object with a string or when you try to print an object using `System.out.println()`.

Example:

```
javaCopy code
class Car {
    String brand;
    int year;

    public Car(String brand, int year) {
        this.brand = brand;
        this.year = year;
    }

    public String toString() {
        return "Car{" +
                "brand='" + brand + '\'' +
                ", year=" + year +
```

```
                '}';
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", 2020);
        System.out.println(myCar); // Output: Car{brand='To
yota', year=2020}
    }
}
```

2. `hashCode()`

Definition:

The `hashCode()` method in Java returns a hash code value for the object. This method is used by hash-based data structures such as HashMap, HashSet, etc.

Example:

```
javaCopy code
public class Main {
    public static void main(String[] args) {
        String str = "Hello";
        int hashCode = str.hashCode();
        System.out.println("Hash code for 'Hello': " + hash
Code);
    }
}
```

3. `equals(Object obj)`

Definition:

The `equals(Object obj)` method is used to compare two objects for equality. The

default implementation of this method in the Object class compares memory addresses, but it is often overridden in user-defined classes to compare object contents.

Example:

```
javaCopy code
class Student {
    String name;
    int id;

    public Student(String name, int id) {
        this.name = name;
        this.id = id;
    }

    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) re
turn false;
        Student student = (Student) obj;
        return id == student.id &&
                Objects.equals(name, student.name);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Alice", 123);
        Student s2 = new Student("Alice", 123);
        System.out.println(s1.equals(s2)); // Output: true
    }
}
```

4. `finalize()`

Definition:
The
`finalize()` method is called by the garbage collector before reclaiming memory occupied by an object. It is a method that can be overridden to provide cleanup code for an object before it is garbage collected.

Example: This method is rarely used explicitly.

5. `getClass()`

Definition:
The
`getClass()` method returns the runtime class of an object. It is a final method in the Object class, so it is available to all Java objects.

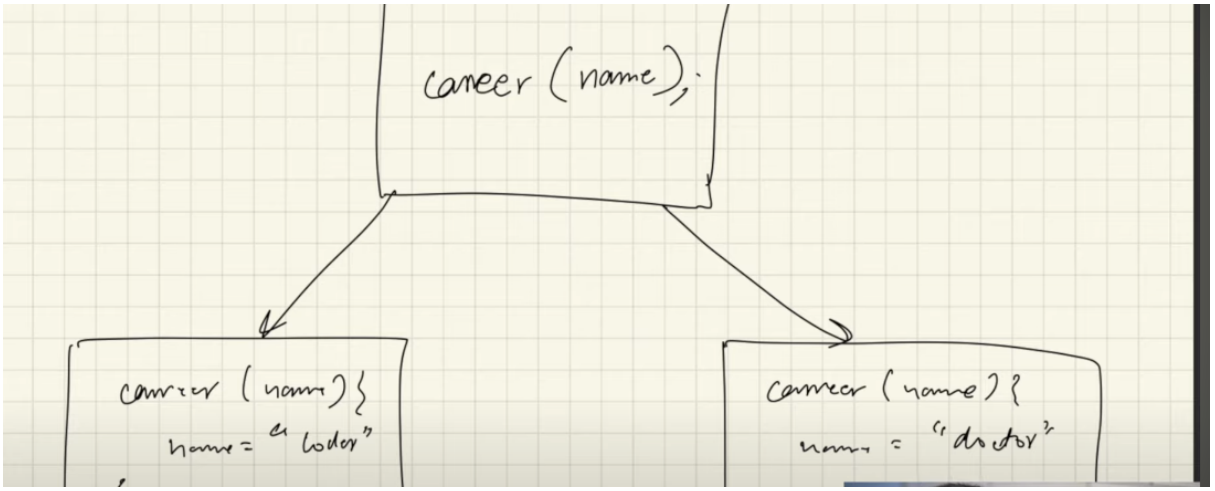
Example:

```
public class Main {
   public static void main(String[] args) {
       String str = "Hello";
       Class<?> cls = str.getClass();
       System.out.println("Class of 'Hello': " + cls.getName
   }
```

Abstract Classes

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.

Abstraction can be achieved with either **abstract classes** or interfaces



- Inheritance conflicts inherit the same Function Name sloving the conflict using Abstract classes
- Overriding sloving the conflicts

The `abstract` keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).
- when you declare an abstract method inside an abstract class, you should include the name add keyword abstract class name.
- Abstract class can't create a object and Abstract class is created in super classes /Parent classes no Body
- Abstract class you can inherit the parent class
- In a method, abstract classes are used, and there may also be abstract classes used within other classes.
- Abstract class you Overiding in using child class object

```
abstract class Animal {
    abstract void makeSound(); // Abstract method
}
```

```
abstract class Animal {
    abstract void makeSound();   //Abstract classes no body
}

class Dog extends Animal {
    void makeSound() {                // Abstarct class only
        System.out.println("Bark");   // Dog is overriding
    }
}

class Cat extends Animal {
    void makeSound() {                  //Abstarct class only
        System.out.println("Meow");     // cat is overiding
    }
}

public class Main {
    public static void main(String[] args) {
        Animal dog = new Dog();
        Animal cat = new Cat();          // only created a Ob

        dog.makeSound();
        cat.makeSound();
    }
}
```

```
public class main{

 static abstract class parent{

   abstract void carrer();
   abstract void age();
 }
```

```
  static public class Son extends parent{

    public void carrer(){
       System.out.println("computer Science");
    }
    public void age(){
       System.out.println("Age is 25");
    }
  }

  static public class Daughter extends parent{

   public void carrer(){
      System.out.println("Doctor");
   }
   public void age(){
       System.out.println("Age is 20");
   }

  }

  public static void main(String[] args) {

   Son son = new Son();

     son.carrer();
     son.age();

   Daughter daughter = new Daughter();
     daughter.carrer();
     daughter.age();
   }


}
```

Abstract class constructor

- Abstract class constructor does created because of Abstract classes does not create a object , so Abstract classes no body no instance variable and methods
- Calling the constructor only the child classes or derived classes

```
public class main{

 static abstract class parent{
    int age;       // Abstract class instance Variable
   abstract void carrer();
   abstract void age();
 }

  static public class Son extends parent{

    public  Son(int age)
    {
       this.age = age;   // only Calling the constructor only
    }
    public void carrer(){
       System.out.println("computer Science");
    }
    public void age(){
       System.out.println("Age is "+age);
    }
  }

  static public class Daughter extends parent{

   public void carrer(){
      System.out.println("Doctor");
   }
   public void age(){
       System.out.println("Age is 20");
```

```
    }

   }

   public static void main(String[] args) {

    Son son = new Son(44);

    son.age = 44;
      son.carrer();
      son.age();

    Daughter daughter = new Daughter();
      daughter.carrer();
      daughter.age();
    }


}
```

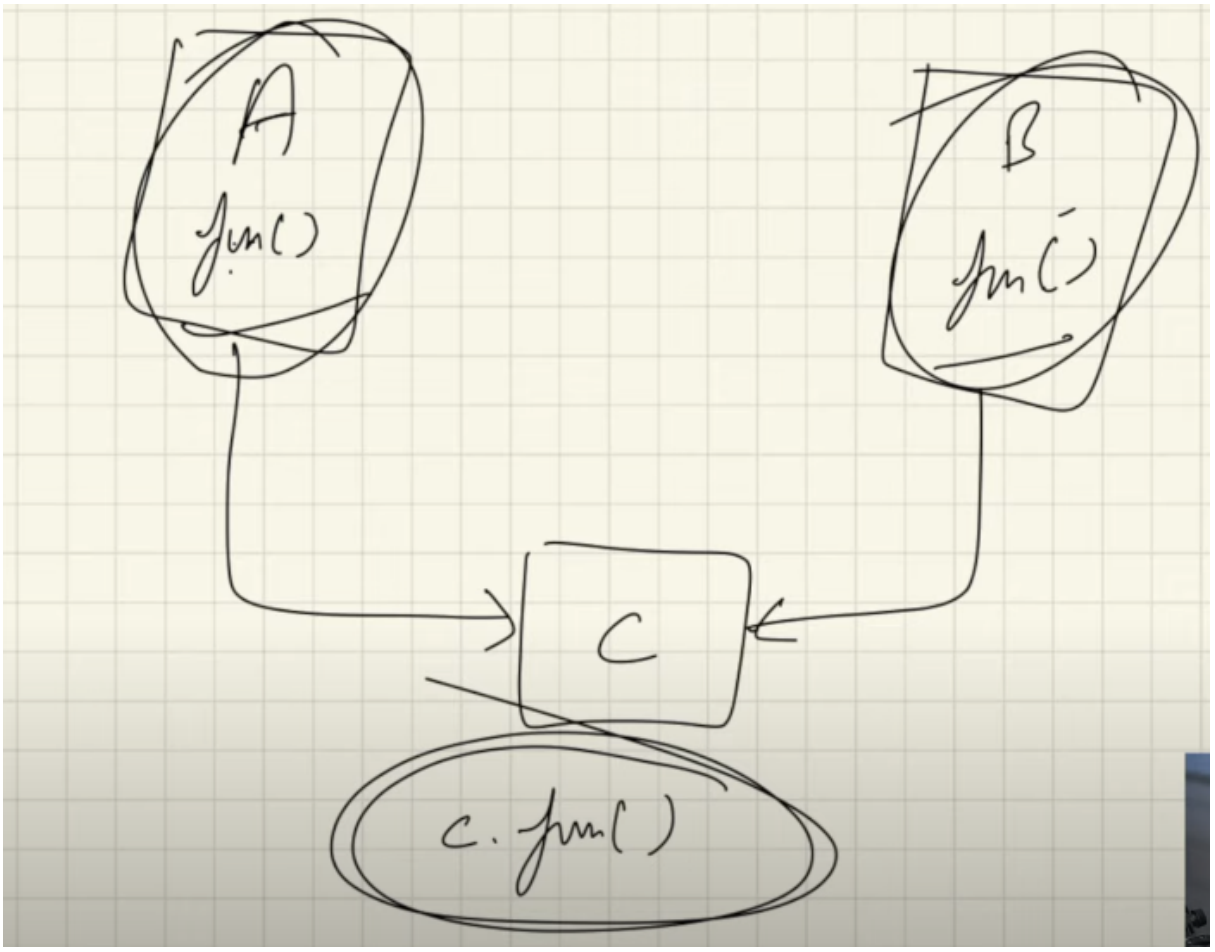
Abstract class Object

Abstract class Object does created because of Abstract classes does not create a object , so Abstract classes no body no instance variable and methods

Static Method

Static method using in Abstract classes because Abstract classes static method not depending on Object

Interfaces



Interfaces

- Interface contains abstract classes
- An `interface` is a completely "**abstract class**" that is used to group related methods
- Like **abstract classes**, interfaces **cannot** be used to create objects
- Interfaces made up on static and final
- Interfaces biggest usage in containing a multiple interface in interfaces classes it also part of Abstract classes
- To access the interface methods, the interface must be "implemented" another class with the `implements` keyword (instead of `extends`). The body of the interface method is provided by the "implement"

- On implementation of an interface, you must override all of its methods
- Interface methods are by default `abstract` and `public`
- Interface attributes are by default `public`, `static` and `final`
- An interface cannot contain a constructor (as it cannot be used to create objects)
- An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).

We use the `interface` keyword to create an interface in Java. For example,

```
interface Language {
  public void getType();

  public void getVersion();
}
```

Difference Class and Interfaces

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.

```
interface Polygon {
  void getArea(int length, int breadth);
}

// implement the Polygon interface
class Rectangle implements Polygon {

  // implementation of abstract method
  public void getArea(int length, int breadth) {
```

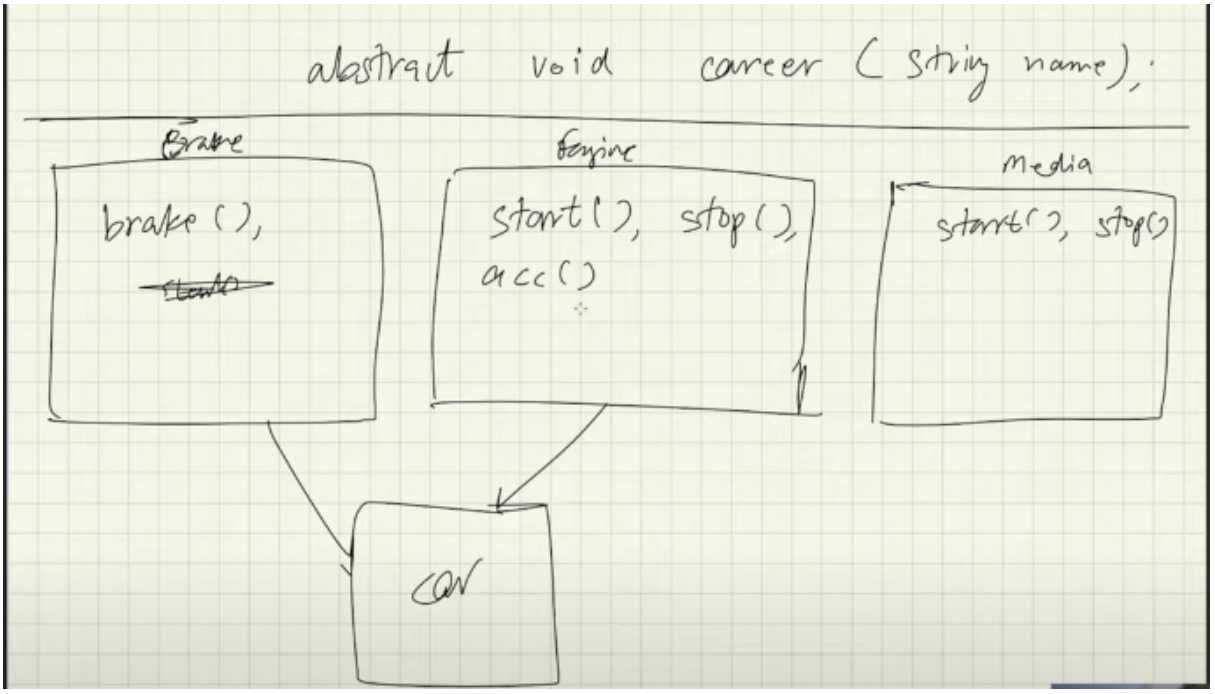
```
    System.out.println("The area of the rectangle is " + (len
  }
}

class Main {
  public static void main(String[] args) {
    Rectangle r1 = new Rectangle();
    r1.getArea(5, 6);
  }
}
```

Example 2

```
// create an interface
interface Language {
  void getName(String name);
}

// class implements interface
class ProgrammingLanguage implements Language {

  // implementation of abstract method
  public void getName(String name) {
    System.out.println("Programming Language: " + name);
  }
}

class Main {
  public static void main(String[] args) {
    ProgrammingLanguage language = new ProgrammingLanguage();
    language.getName("Java");
  }
}
```

Interfaces

- Interface support multiple inheritance classes in one interface files and contains
- Interface using to archive multiple inheritance
- Abstract method run as the Run time method the value



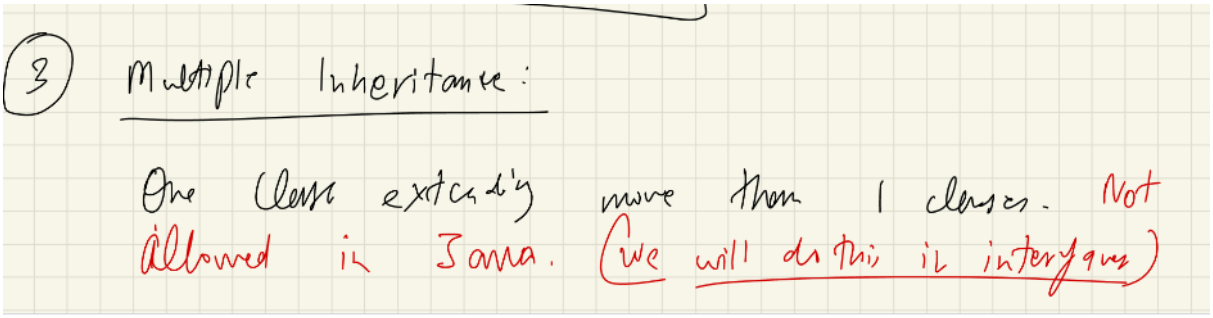
- Parent class no idea of child classes but child class have idea of parent classes when you are calling one function to child to exit on parent classes, parent classes and child class have a present in compile time java complier to check same or not in more child classes get pushed in parent class in higher and higher levels this problem sloved in interfaces they disconnected definition of method hierarchy of inheritance
- Interface give to an main body implement muliple classes
- Interface is not hierarchy class it combine unrelated to each other doesn't care about their parent's classes
- two classes unrelated each other implements or invoke the same interfaces
- class to interface using implements key word , interface to interfaces using extends key word.

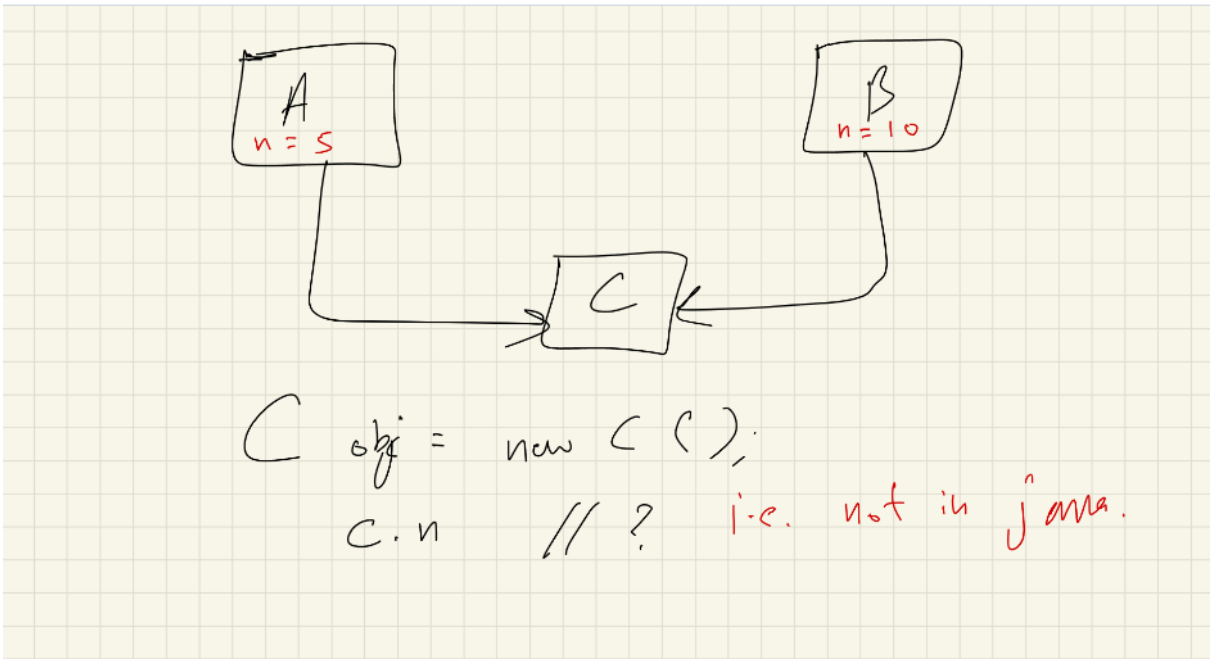
Extends vs Implements

S.No.	Extends	Implements
1.	By using “extends” keyword a class can inherit another class, or an interface can inherit other interfaces	By using “implements” keyword a class can implement an interface
2.	It is not compulsory that subclass that extends a superclass override all the methods in a superclass.	It is compulsory that class implementing an interface has to implement all the methods of that interface.
3.	Only one superclass can be extended by a class.	A class can implement any number of an interface at a time
4.	Any number of interfaces can be extended by interface.	An interface can never implement any other interface

Multiple inheritance

- Multiple inheritance solving only interfaces and Abstract class






```
// Interface for the first parent class
interface Parent1 {
    void method1();
}

// Interface for the second parent class
interface Parent2 {
    void method2();
}

// Concrete class implementing both interfaces
class Child implements Parent1, Parent2 {
    public void method1() {
        System.out.println("Method 1 from Parent1");
    }
```

```
    public void method2() {
        System.out.println("Method 2 from Parent2");
    }
}

public class Main {
    public static void main(String[] args) {
        Child obj = new Child();
        obj.method1();
        obj.method2();
    }
}
```

Static method interfaces

- Static method can overriding in objects and static is independent of object
- static methods are not inherited classes also apply not inherited the Interfaces
- static interface methods you should always have a body
- static method you always declare variable
- static method you should call in the interface name

Nested Interfaces

- We can declare interfaces as members of a class or another interface. Such an interface is called a member interface or nested interface. **Interface in a class** Interfaces (or classes) can have only public and default access specifiers when declared outside any other class

```
Synatax
interface first{
     interface second{
          ...
     }
}
```

```
// Java program to demonstrate working of
// interface inside a class.
import java.util.*;
class Test {
    interface Yes {
        void show();
    }
}

class Testing implements Test.Yes {
    public void show()
    {
        System.out.println("show method of interface");
    }
}

class A {
    public static void main(String[] args)
    {
        Test.Yes obj;
        Testing t = new Testing();
        obj = t;
        obj.show();
    }
}
```

Exception Handling

- The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.

  error ⇒ lacking of resources example Stack Memory is full, out of Memory error, you cannot recoverable

- Expection handling ⇒ flow of program eg divided by Zerom, Null pointer error

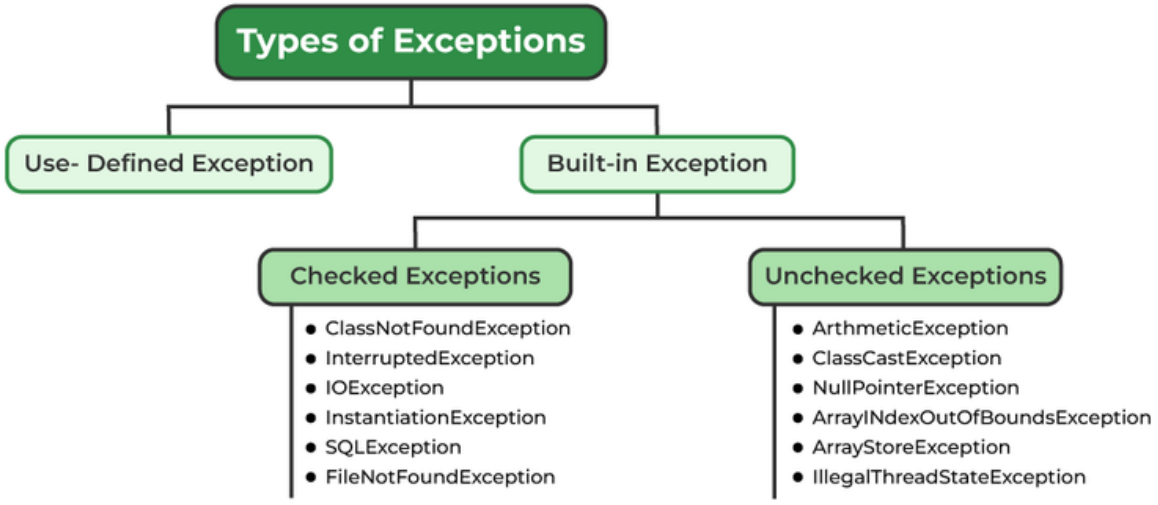
What is Exception Handling?

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions.

Exception Handling Diagram



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

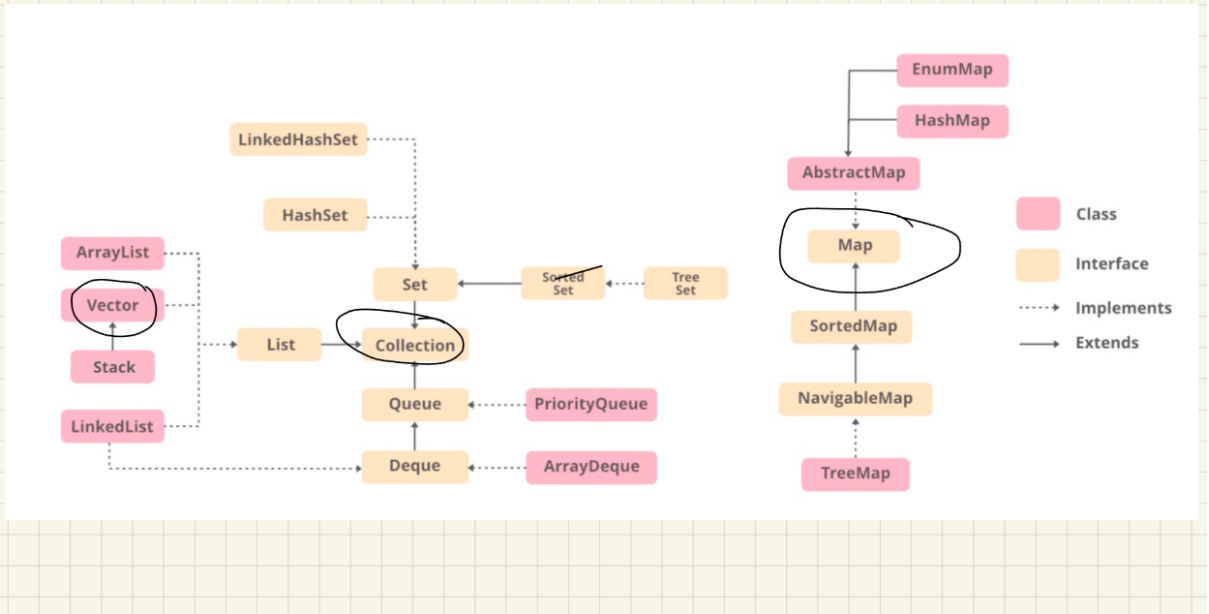
Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.

finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

```
//program to print the exception information using getMessage

import java.io.*;

class GFG1 {
    public static void main (String[] args) {
      int a=5;
      int b=0;
        try{
          System.out.println(a/b);
        }
      catch(ArithmeticException e){
        System.out.println(e.getMessage());
      }
    }
}
```

Collection Frame works

- Any group of individual objects that are represented as a single unit is known as a Java Collection of Objects. In Java, a separate framework named the *“Collection Framework”* has been defined in JDK 1.2 which holds all the Java Collection Classes and Interface in it.



Array List & Vector

- `ArrayList` is not synchronized, meaning it is not thread-safe. Multiple threads can manipulate an ArrayList concurrently, but it is not safe to do so without proper synchronization.
- `Vector`, on the other hand, is synchronized, which means it is thread-safe. This means that multiple threads can safely manipulate a Vector without external synchronization.

Performance:

- Due to the synchronization overhead, Vector operations are generally slower than ArrayList operations.
- Array List is generally preferred in scenarios where thread-safety is not a concern, as it offers better performance.

Growth increment:

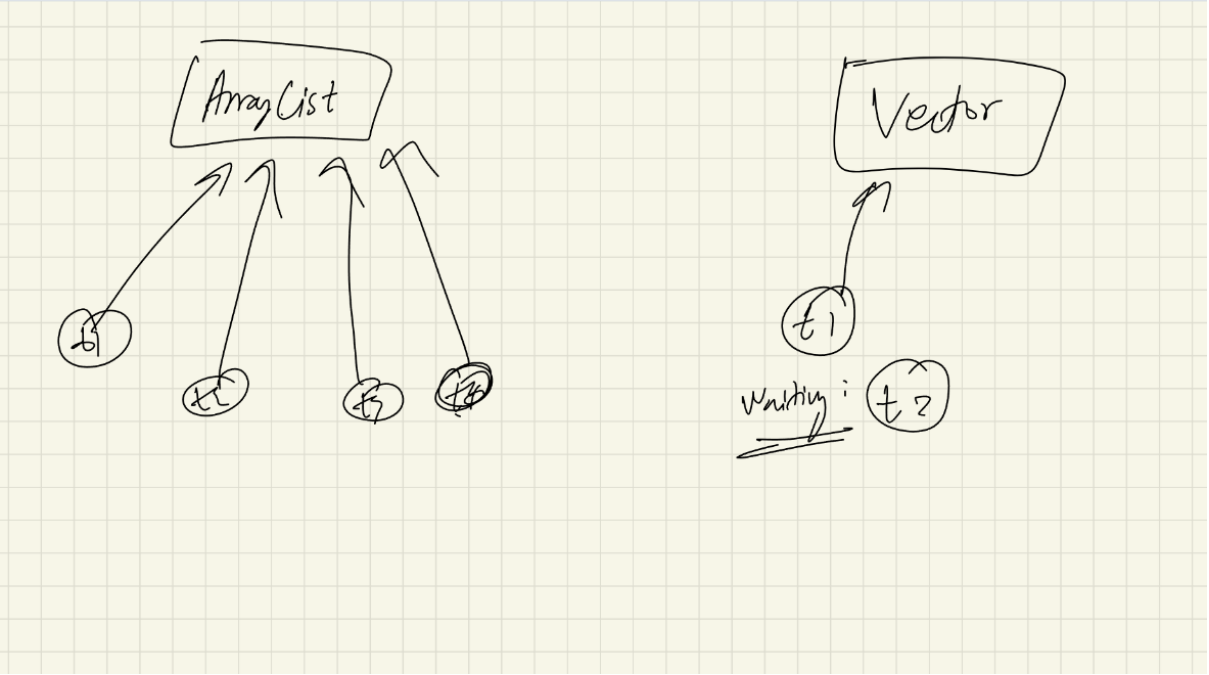
1. **ArrayList**:

- When an ArrayList needs to expand because it has reached its capacity, it allocates a new array with a larger size.
- The new size is typically calculated as 1.5 times (or 50% more) the current capacity.
- For example, if an ArrayList's current capacity is 10 elements and it needs to grow, it will allocate a new array with a capacity of 15 elements.
- This growth strategy helps balance memory usage and performance by avoiding excessive resizing while ensuring sufficient capacity for future elements.

2. **Vector**:
    - Similar to ArrayList, when a Vector needs to expand due to reaching its capacity, it allocates a new array with a larger size.
    - However, the growth strategy for Vector is different; it doubles its size when it needs to expand.
    - For example, if a Vector's current capacity is 10 elements and it needs to grow, it will allocate a new array with a capacity of 20 elements.
    - Doubling the size ensures faster growth compared to ArrayList's 50% increase but may result in more memory being allocated than necessary in some cases.

S. No.	ArrayList	Vector
1.	ArrayList is not synchronized.	Vector is synchronized.
2.	ArrayList increments 50% of the current array size if the number of elements exceeds ts capacity.	Vector increments 100% means doubles the array size if the total number of elements exceeds its capacity.
3.	ArrayList is not a legacy class. It is introduced in JDK 1.2.	Vector is a legacy class.
4.	ArrayList is fast because it is non-synchronized.	Vector is slow because it is synchronized, i.e., in a multithreading environment, it holds the other threads in a runnable or non-runnable state until

		the current thread releases the lock of the object.
5.	ArrayList uses the Iterator interface to traverse the elements.	A Vector can use the Iterator interface or Enumeration interface to traverse the elements.
6	ArrayList performance is high	Vector performance is low
7	Multiple threads is allowed	only one threads are allowed .



Enum

- The **Enum in Java** is a data type which contains a fixed set of constants.
- It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

- Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and final implicitly. It is available since JDK 1.5.
- Enums are used to create our own data type like classes. The **enum** data type (also known as Enumerated Data Type) is used to define an enum in Java. Unlike C/C++, enum in Java is more *powerful*. Here, we can define an enum either inside the class or outside the class.
- Java Enum internally inherits the *Enum class*, so it cannot inherit any other class, but it can implement many interfaces. We can have fields, constructors, methods, and main methods in Java enum.

Points to remember for Java Enum

- Enum improves type safety
- Enum can be easily used in switch
- Enum can be traversed
- Enum can have fields, constructors and methods
- Enum may implement many interfaces but cannot extend any class because it internally extends Enum class

purpose of the values() method in the enum?

The Java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.

purpose of the valueOf() method in the enum?

The Java compiler internally adds the valueOf() method when it creates an enum. The valueOf() method returns the value of given constant enum.

purpose of the ordinal() method in the enum?

The Java compiler internally adds the ordinal() method when it creates an enum. The ordinal() method returns the index of the enum value.

Values()

```
class EnumExample1{
public enum Season { WINTER, SPRING, SUMMER, FALL }
public static void main(String[] args) {
for (Season s : Season.values())  //considering like a for ea
System.out.println(s);
}
}
```

valueof() method and valof ordinal()

```
class EnumExample1{
//defining enum within class
public enum Season { WINTER, SPRING, SUMMER, FALL }
//creating the main method
public static void main(String[] args) {
//printing all enum
for (Season s : Season.values()){
System.out.println(s);
}
System.out.println("Value of WINTER is: "+Season.valueOf("WIN
System.out.println("Index of WINTER is: "+Season.valueOf("WIN
System.out.println("Index of SUMMER is: "+Season.valueOf("SUM

}}
```

Enum data type programming

```
// Define the Fruit enum
enum Fruit {
    APPLE("Red", "Sweet"),
    BANANA("Yellow", "Sweet"),
    ORANGE("Orange", "Tangy"),
    GRAPE("Purple", "Sweet"),
    LEMON("Yellow", "Sour");

    // Properties of each fruit
    private final String color;
    private final String taste;

    // Constructor to initialize properties
    Fruit(String color, String taste) {
        this.color = color;
        this.taste = taste;
    }

    // Getter methods to access properties
    public String getColor() {
        return color;
    }

    public String getTaste() {
        return taste;
    }
}

// Main class to demonstrate the use of the Fruit enum
public class Main {
    public static void main(String[] args) {
        // Iterate over each fruit and display its properties
        for (Fruit fruit : Fruit.values()) {
            System.out.println("Fruit: " + fruit.name());
            System.out.println("Color: " + fruit.getColor());
            System.out.println("Taste: " + fruit.getTaste());
```

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