* **Inheritance:** is – a relationship

**Person**

**Employee**

**Staff**

**Teacher**

**Professor**

**Lecturer**

**Admin**

**Parent class:** Person

**Child Class:** Employee and further classes

* **Syntax Difference:**
* In **C,**

**class Employee: Public Person**

* **Public**: Access Specifier
* **:** shows Imheritance
* In **Java,**

**class Employee extends Person**

* **:** keyword is replaced with **extends** keyword
* All inheritance in java will be **public**. There will no private or protected inheritance.
* **Conceptual Difference:**

1. In java, there is not concept of multiple inheritance (except interfaces).
2. In java, all functions of parent class are by default virtual. (i.e., there is no virtual keyword in java). It means all the functions can be directly over-ride in the child classes.
3. In Java, super keyword will be used for following two functionalities:
4. To invoke the desire constructor of parent class or creation of child class object.
5. To invoke the over-rided functionality of parent class.
6. By default, Java will add a class as parent to every Java class which is not inheriting. (In the above hierarchy, class Person is not inheriting from any other class. So by default, java will add an Object class as its parent).

**Object**

**Person**

**Employee**

**Staff**

**Teacher**

**Professor**

**Admin**

**Lecturer**

1. **Multiple Inheritance:** One class inherits from two class.

In java, there is **no multiple inheritance**. There can’t be two parents of a class. A class will be inherited from only one class at one time.

1. By default, there is **public inheritance** in Java.
2. All functions of parent **are virtual by default**. There is no need to write virtual keyword. If we will write, error will be generated as there is no virtual keyword in java.

If u are trying to over-ride a function, u can write this annotation **@override**. If u wont write this, it will still over-ride. @over-ride this gives error at compile time if we are trying to over-ride a function that is not in parent class.

**More Info:**

In Java, there is no "virtual" keyword like in C++ to explicitly declare a function as virtual. In Java, all non-static methods (functions) are considered to be virtual by default. This means that they can be overridden in derived classes.

In Java, when you define a method in a class, it is assumed to be virtual unless you use the **final** keyword to explicitly mark it as not overrideable, or if the class itself is marked as **final**, indicating that it cannot be subclassed.

**Virtual Functions in Java:**

* + In Java, all non-static methods are implicitly virtual, which means they can be overridden by subclasses.
  + Unlike languages like C++, Java doesn't use the **virtual** keyword; it automatically considers all non-static methods as virtual.

**@Override Annotation:**

* + In Java, the **@Override** annotation is not strictly required to override a method, but it is good practice to use it.
  + If you use **@Override** and the compiler detects that the method does not actually override a method in a superclass, it will generate an error.

1. Person class – function: void show()

Employee class – function: void show() **over-rided**

We want to access the functionality of the overridden method in the parent class. We want to call the parent class function which is over-rided.

**Super.show();** // Immediate call to parent

**More info:**

In Java, you use the **super** keyword to refer to the superclass (or parent class), and you can use it to call the overridden method from the superclass.

Inside the overridden method in the **Employee** class, **super.show()** is used to explicitly call the **show()** method of the parent class (**Person**), allowing you to access the functionality of the overridden method in the parent class.

1. **Inheritance:** Reusability of code

It will be reused if it will be present.

Class is just like a map. Actual thing is object.

* Using functionality of parent class in child class. So when we make an object of child class then first object of parent class will be made because that will be reused to reuse functionality of parent class in child class.

Whenever, an object is made, its constructor is called.

**Q:** When an object of class is made, first constructor of parent class is called. Which constructor of parent class will be called? (Default, 1-paramterized, 2-parameterized)?

**A:** We will determine it through super.

* Constructor of parent call :

In C,

Employee(): Person()

{

}

In Java,

Class Employee extends Person

{

Employee()

{

Super(\_\_);

}

}

Super will determine which constructor of parent class will be called. It will look at the type also. If there are 2 string parameters, then it will call the constructor of parent class which has 2 strings…

**More Info:**

In Java, when you use the super() keyword in a constructor of a subclass (like Employee in your example), it is used to call the constructor of the immediate parent class (superclass). The super() call should be the first statement in the constructor body.

In your example:

**class Employee extends Person {**

**Employee() {**

**super(/\* arguments \*/);**

**}**

**}**

The super() call is used to invoke the constructor of the Person class. If the Person class has multiple constructors, you need to provide the appropriate arguments that match one of the constructors in the Person class.

For example, if there is a constructor in the Person class that takes two string parameters, you would call it like this:

**class Person {**

**Person(String arg1, String arg2) {**

**// constructor logic**

**}**

**}**

**class Employee extends Person {**

**Employee() {**

**super("value1", "value2");**

**}**

**}**

In this case, the super("value1", "value2") statement explicitly calls the constructor in the Person class that takes two string parameters.

Keep in mind that if you don't provide a super() call explicitly in the subclass constructor, Java will automatically insert a call to the parameterless constructor of the superclass. If the superclass does not have a parameterless constructor and you don't provide the super() call with appropriate arguments, a compilation error will occur.

**Question:**

Super statement is inside child constructor. Inheritance says that first constructor of parent class will be made, then constructor of child class will be made. But we are deciding in child constructor about type of parent constructor to be called. Then isn’t the instance/ object of child class is being made first?

**Answer:**

Reason: Super statement must be first statement/ line in child class constructor. Java doesn’t allow us to write like this in front of employee class Employee(): \_\_ . It is allowed in C++ but not in Java.

In first line we are just specifying. First call will be made to parent class. If super statement will not be in first line, then java compiler will generate an error.

* Java adds **parent Object class** to every class which is not being inherited from any class.

Like person class is not being inherited from any other class. So it is being inherited from Object class. Object is indirectly parent of Employee, staff, lecturer and so on.

In java, Object will be parent of any class.

**Object – global object of Java.**

* **C:**

**Person \*p;**

**p = new Employee();**

**p->\_\_()**

This p may contain objects of tchr, staff, lecturer etc class. Person is a reference or global variable because variable of person is containing all lower objects.

* **Java:**

In java, Object is reference variable. It is parent class of every class. So it can contain objects of all classes.

**= > Use of Object:**

1. Streams
2. Session Handling

Use belongs to reference variable of parent class (Object) which will contain object of any class.

* Object is parent class. So it also provides some basic functionality.
* It provides toString() function.
* It provides finalize()
* **toString() function:**

What does it do? Convert to string? Nope

**HW 1**

**More info:**

* In Java, the **toString()** method is a method defined in the **Object** class, which is the root class for all Java classes. The purpose of the **toString()** method is to return a string representation of the object.
* By default, the **toString()** method in the **Object** class returns a string that consists of the class name followed by the "@" character and the object's hashcode. For example, if you have a class named **MyClass** and you call **toString()** on an instance of this class, you might get a string like **MyClass@1a2b3c4d**.
* However, it is common practice for classes to override the **toString()** method to provide a more meaningful and human-readable representation of the object. By doing this, you can customize the string representation of your objects to include relevant information about their state.
* Here's an example of how you might override the **toString()** method in a custom class:

public class MyClass {

private int value;

public MyClass(int value) {

this.value = value;

}

@Override

public String toString() {

return "MyClass{" +

"value=" + value +

'}';

}

public static void main(String[] args) {

MyClass myObject = new MyClass(42);

System.out.println(myObject.toString()); // Output: MyClass{value=42}

}

}

* In this example, the **toString()** method has been overridden to provide a more informative string representation of the **MyClass** object. When you call **toString()** on an instance of **MyClass**, it returns a string like **"MyClass{value=42}"** instead of the default **"MyClass@1a2b3c4d"**.
* **finalize() function:**

Its functionality is **similar to destructor** but not exactly destructor. When we will write it in code, we will get a **warning message – deplicate** (they are going to be delete from library).

**HW 2:** Write in ur class and see in documentation what it is?

**More info:**

* The **finalize()** method in Java is not exactly similar to a destructor in languages like C++ or C#. Instead, it's a method provided by the **Object** class in Java, and it is called by the garbage collector before an object is reclaimed. Its purpose is to allow an object to perform cleanup operations before it is garbage collected.
* The finalize() method in Java is a method of the Object class used to perform cleanup activity before destroying any object. Garbage collector calls it before destroying the objects from memory. finalize method in Java is called by default for every object before its deletion.
* **Type:**
* int i – 4 bytes
* double d – 8 bytes

1. If I write **d= i**, is it allowed?

* Yes. Because double is biggest byte. Integer is smaller byte than double. We can put smaller byte inside a bigger byte. We can assign/ put an integer values inside a double. We don’t need to do something specifically. This is **implicit/ automated typecasting**.

1. **Int i, double d = 10.** 10 is a complete integer value. Is **i = d** allowed?

* Nope, We can’t assign/ put bigger value inside a small. Don’t look at the value. Look at **type** of variables. To put double inside integer, u need to cut its size.
* In C++, we used static\_alloc, malloc etc for typecasting.
* In java, we just write the **target type** with it. Like this: **i = (int) d**. This is called **explicit casting**. We are specifically doing it.

1. If double d = 10.2, then if we do i = (int) d, only 10 will be saved in i not 10.2. U always have a **chance of losing some data** even in typecasting.

* In hierarchy, which is biggest type? – Object is the biggest type.
* Why? Why object is the biggest type?
* Because reference of object class may contain object of Person, Employee etc classes.
* Object of Person class can contain object of any class.
* Employee, teacher, Professor is a person.
* Person may behave like a employee, act as a teacher, as a staff… **multiple behaviors.**
* **Reference variable of Parent class can contain object of any child class.**
* There is **no reference variable** in Java.

**Person P;**

**P = new Employee();**

**P = new Professor();**

This is **implicit type casting**. Person is bigger type and is containing smaller types. P may contain objects of child classes.

Size of p = total size of its attributes

Size of attributes = size of objects in memory.

Object = collection of attributes.

**Q:** Child class contains functionality of all the upper classes. So shouldn’t the child class be the biggest type?

**A:** Nope. Although it has functionality of parent class. **In memory all objects are isolated.** In memory objects of teacher, professor, person all are isolated (separated). Whose functionality will be accessed where? Object doesn’t matter, its accessibility matters.

* Teacher, professor, employee is a Person. But vice versa is not true.

**P = Prof…**

**P = Teacher**

**P = Emp….**

Person P is bigger type because it contains many child objects. Type will be based on the thing which has worth, implementation. Type will be on base of object.

Class -> Actual no worth

Object -> Actual worth

* **Scenarios:**
  + 1. **Person P = new Person();**

**Employee e = new Employee();**

**e = (Employee) P; // i = (int) d**

This **e = (Employee) P is not possible**, not possible even in C++. Person is bigger type, employee is smaller type.

e = (Employee) P; -> It means that every person is an employee which is not possible. Top to bottom relationship of is – a is being disturbed.

* + 1. **How to do downcasting?**

**Person P = new Employee();**

**Employee e = new Employee();**

Both are creating instance of employee.

**e = P; (Nope)**

P is bigger type, so type casting.

**e = (Employee) P;** //i , d = 10; i = (int) d;

This is the only way u can implement **explicit/ downcasting in C++.**

* **Coding:**

1. **Class A:**

Three instance variables: a, b, c

3-parameterized Constructor

**public class A**

**{**

**int a;**

**int b;**

**int c;**

**A(int p, int q, int r)**

**{**

**a = p;**

**b = q;**

**c = r;**

**}**

**}**

1. **Class B (Child class of class A):**

* Inheriting from class A. Inheritance is public by default. So no need to write default keyword.
* Create instance variable d .
* 4-parameterized Constructor. In first line of this child constructor, use super keyword to call 3-parameterized constructor of class A.
* Show function to print all 4 variables.
* In main function, make an object of B and call show function.

class B extends A

{

    //Instance variable

    int d;

    //4-parameterized constructor

    B(int l, int m, int n, int o)

    {

        super(l, m, n); // Immediate call to parent constructor

        d = o;

    }

    //Show function

    void show()

    {

        System.out.println("a: " +a);

        System.out.println("b: " +b);

        System.out.println("c: " +c);

        System.out.println("d: " +d);

    }

    //Main function

    public static void main(String args[])

    {

        //Creating object of class B

        B b = new B(1, 2, 3, 4);

        //Calling show function

        b.show();

    }

}

1. **Class Circle:**

* Instance variable: protected radius.
* **Protected** means it is available in this class and in child class.
* Constructor: 1 –parameterized.
* this-> referring to current object.
* this.radius (instance variable, belongs to current calling object).
* = radius (parameter)
* Math: class of **java.util** package
* PI: Constant variable of Math class.
* There is no constant keyword in Java, final is used.
* Math.PI
* Math – name of class
* PI – **static final** variable
* In Java's **Math** class, the **PI** constant is declared as a **public static final** variable. This means it is both **static** and **final.** Being **static** means that you access it using the class name **(Math.PI**), and being **final** means that its value cannot be changed once it's assigned. It is a constant value representing the mathematical constant π (pi).
* import java.util.\*;
* public class Circle
* {
* //Instance variable
* protected double radius;
* //1-parameterized Constructor
* public Circle(double radius)
* {
* this.radius = radius;
* }
* //getArea function
* public double getArea()
* {
* return Math.PI\*radius\*radius;
* }
* }

1. **Cylinder Class (Child class of Circle class):**

* Protected double length.
* Constructor – first line: **super(radius)** //Immediate call to parent class constructor.
* Super should be in first line
* Area of Cylinder class = 2\*pi\*r^2 + 2Pi\*inch.
* Pi\*r^2 – super.getArea()
* **Super.getArea()** – Function of parent class will be called which has been over-rided. We are accessing functionality of parent class here.

import java.util.\*;

public class Cylinder extends Circle

{

    //Instance variable

    protected double length;

    //Constructor

    public Cylinder(double radius, double length)

    {

        super(radius);  //Immediate call to parent class

        this.length = length;

    }

    //getArea()

    public double getArea()

    {

        return 2\*super.getArea() + 2\*Math.PI\*length\*radius;

    }

}

1. **Test Class:**

* Main function:
* **Scenarios:**

1. **Circle class**

* Create instance of Circle class
* Then get its area

**Circle myCircle = new Circle(1.20);**

**System.out.println(“Area of my circle: “ + myCircle.getArea());**

1. **Cylinder class**

* Create instance of Cylinder class
* Then get its area

**Cylinder myCylinder = new Cylinder(1.20, 2.50);**

**System.out.println(“Area of my cylinder: “ + myCylinder.getArea());**

1. **UpCasting/Implicit Casting**

* Reference variable of Parent class (Circle) is containing object of Child class (Cylinder).

**Circle myCircle1 = new Cylinder(1.20, 2.50);**

**System.out.println(“UpCasting …..”);**

**System.out.println(“Area of myCircle1: “ + myCircle1.getArea());**

1. **DownCasting/ Explicit Casting**

* Reference variable of Parent class is containing object of child class.

**Circle myCircle2 = new Cylinder(1.20, 2.50);**

* Object of child class

**Cylinder myCylinder2;**

* Circle is bigger class than cylinder. So, downcasting

**myCylinder2 = (Cylinder) myCircle2;**

1. **Try (HW)**

* Create instance of circle class.
* Cylinder class instance
* Compile the code. And run

**Circle myCircle3 = new Circle(1.20);**

**Cylinder myCylinder3 = new Cylinder(1.20, 2.50);**

**//myCircle3 = myCylinder3; (Implicit type Casting)**

**myCylinder3 = myCircle3; //Error**

**myCylinder3 = (Cylinder) myCircle3;**

* Error – read it

**More Info:**

1. **myCircle3 = myCylinder3;**: This line is an example of implicit upcasting. It is allowed to assign an instance of a subclass (**Cylinder**) to a variable of the superclass (**Circle**) without explicit casting.
2. **myCylinder3 = myCircle3;**: This line is an example of implicit downcasting. However, it will result in a compilation error because you are trying to assign a superclass instance (**Circle**) to a variable of the subclass type (**Cylinder**) without an explicit cast. This operation is not allowed without an explicit cast.

If you uncomment the line **// myCylinder3 = myCircle3;**, you will see a compilation error indicating that incompatible types are being assigned. The error message will highlight that a cast is required for this assignment to be valid.

1. **myCylinder3 = (Cylinder) myCircle3;**: This line explicitly downcasts the **Circle** instance to a **Cylinder** instance. This is allowed at compile time because the compiler trusts that you know what you're doing. However, if the actual object referred to by **myCircle3** at runtime is not a **Cylinder**, a **ClassCastException** will be thrown at runtime.

public class Test

{

    public static void main(String args[])

    {

        //Circle class

        Circle myCircle = new Circle(1.20);

        System.out.println("Area of my Circle: " + myCircle.getArea()); //4.523893421169302

        //Cylinder class

        Cylinder myCylinder = new Cylinder(1.20, 2.50);

        System.out.println("Area of my Cylinder: " + myCylinder.getArea());  //27.89734276387736

        //Implicit/ Up Casting

        Circle myCircle1 = new Cylinder(1.20, 2.50);

        System.out.println("Up Casting ....");

        System.out.println("Area od myCircle1: " + myCircle1.getArea()); //27.89734276387736

        //Explicit/ Down Casting

        Circle myCircle2 = new Cylinder(1.20, 2.50);

        Cylinder myCylinder2;

        myCylinder2 = (Cylinder) myCircle2;

        System.out.println("Area: " + myCylinder2.getArea());  //27.89734276387736

        //Try

        Circle myCircle3 = new Circle(1.20);

        Cylinder myCylinder3 = new Cylinder(1.20, 2.50);

        //myCircle3 = myCylinder3; //No error Okay

        //System.out.println("Area of my Circle3: " + myCircle3.getArea());  //27.89734276387736

        //System.out.println("Area of my Cylinder3: " + myCylinder3.getArea());  //27.89734276387736

        //myCylinder3 = myCircle3; //error: incompatible types: Circle cannot be converted to Cylinder

        //myCylinder3 = (Cylinder) myCircle3;  //Exception in thread "main" java.lang.ClassCastException: class Circle cannot be cast to class Cylinder (Circle and Cylinder are in unnamed module of loader 'app') at Test.main(Test.java:33)

    }

}

* If u don’t want to over-ride a function in child class, make it final.
* **Q:** What will happen if we will write final with a class? Try writing final with Circle class.

A: **// Error: LinkageError occurred while loading main class Cylinder java.lang. IncompatibleClassChangeError: class Cylinder cannot inherit from final class Circle.**

* When you declare a class as **final** in Java, it means that the class cannot be subclassed or extended.
* **More info:**
* 1. **Inheritance Restriction:** Subclasses cannot be created: If you declare a class as **final**, it indicates that the class is complete and should not be extended. Attempting to create a subclass of a **final** class will result in a compilation error.
* **2. Method Overriding Restriction:** Methods in a **final** class are implicitly **final**: When you declare a class as **final**, all its methods are implicitly **final**. This means that methods in the **final** class cannot be overridden by subclasses. If you explicitly use the **final** keyword with a method in a class (whether the class is **final** or not), it further enforces that the method cannot be overridden.
* If a method in a non-final class is marked as **final**, it cannot be overridden by subclasses.
* Declaring a class as **final** prevents it from being subclassed, and the methods in that class are implicitly **final** as well. This design choice is useful when you want to ensure that a class or certain methods in a class should not be extended or overridden, respectively.
* **HW FOR VACATION:**

1. **Conversion functions (static and non-static) take a look at that.**
2. **Abstract Class (Idea, concept is same. Syntax change) read it.**