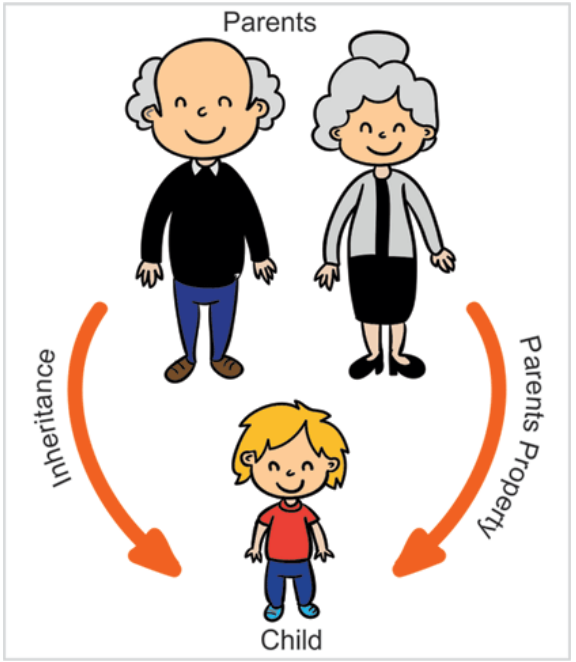
Inheritance in java



Java Inheritance Topics

- Is-a relationship
- Extents keyword
- Code reuse
- Subclass and supper class
- Inheritance with access modifiers
- No multiple inheritance
- Type casting- upcasting and down casting
- Overriding methods
- Calling superclass methods
- Instanceof instruction
- Fields and inheritance
- Constructors and inheritance
- Nested class and inheritance
- Abstract classes and inheritance

Is-a relationship

- **A relationship** in Java means different relations between two or more classes. For example, if a class Bulb inherits another class Device, then we can say that Bulb is having is-a relationship with Device, which implies Bulb is a device.
- In Java, we have two types of relationship:
    1. **Is-A relationship:** Whenever one class inherits another class, it is called an IS-A relationship.
    2. **Has-A relationship:** Whenever an instance of one class is used in another class, it is called HAS-A relationship.

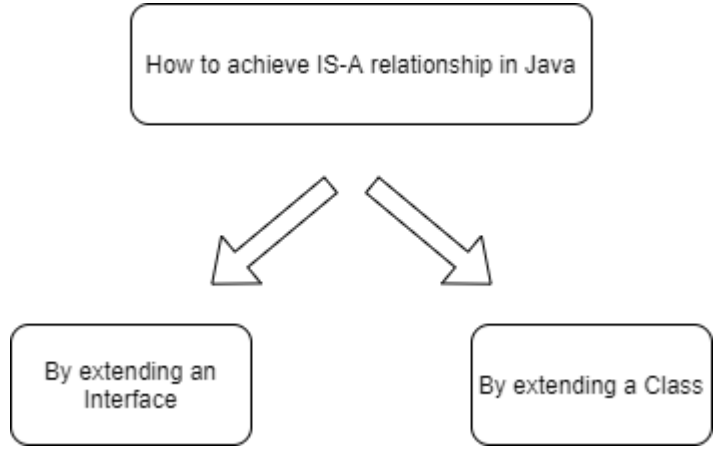
Is-A relationship

- **IS-A Relationship** is wholly related to Inheritance. For example – a kiwi is a fruit; a bulb is a device.
- IS-A relationship can simply be achieved by using extends Keyword.
- IS-A relationship is additionally used for code reusability in Java and to avoid code redundancy.
- IS-A relationship is unidirectional, which means we can say that a bulb is a device, but vice versa; a device is a bulb is not possible since all the devices are not bulbs.
- IS-A relationship is tightly coupled, which means changing one entity will affect another entity.

Advantage of IS-A relationship

•Code Reusability.

•Reduce redundancy.



Example-1

```
// Java program to demonstrate the
// working of the Is-A relationship
import java.io.*;
// parent class
class Device {
        private String deviceName;
        public void setDeviceName(String deviceName)
        {
                this.deviceName = deviceName;
        }
        public String getDeviceName()
        {
                return this.deviceName + " is a Device";
        }
}
```

```
// child class
class Bulb extends Device {
        public static void main(String gg[])
        {
                // parent class can store the reference
                // of instance of child classes
                Device device = new Bulb();

                // set the device name to bulb
                System.out.println("Device name is Bulb");
                device.setDeviceName("Bulb");

                // print the device name
                System.out.println(device.getDeviceName());
        }
}
```

Output

```
Device name is Bulb
Bulb is a Device
```

Superclass and Subclass

- By using the inheritance concept, we can acquire all the features (members) of a class and use them in another class by relating the objects of two classes.
- The class from where a subclass inherits the features is called **superclass**. It is also called a base class or parent class.
- A class that inherits all the members (fields, method, and nested classes) from the other class is called **subclass**. It is also called a derived class, child class, or extended class.

Example-2

```
public class NumberTest
{
public static void main(String[] args)
{
   Number2 n = new Number2();
    n.display();
 }
}
```

Output

X = 20
X = 50

```
package inheritance;

//Super class declaration

public class Number

{

 int x = 20;

 void display()

 {

   System.out.println("X = " +x);

 }

}


//Subclass declaration

public class Number2 extends Number

{

   int x = 50;

 void display()

 {

// Accessing superclass instance variable using super keyword.

System.out.println("X = " +super.x);

    System.out.println("X = " +x);

  }

 }
```

Inheritance with access modifiers

Superclass members can be inherited to subclass provided they are eligible by access modifiers. The behavior of access specifiers in the case of inheritance in java is as follows:

1. The private members of the superclass cannot be inherited to the subclass because the private members of superclass are not available to the subclass directly. They are only available in their own class.
2. The default members of the parent class can be inherited to the derived class within the same package.
3. The protected members of a parent class can be inherited to a derived class but the usage of protected members is limited within the package.
4. Public members can be inherited to all subclasses.

Example-3

```
public class MainClass
{
 public static void main(String[] args)
 {
// Private members cannot be accessed due to not available in subclass.
   Derivedclass d = new Derivedclass();
     d.m2();
   System.out.println("y = " +d.y);
  }
 }
```

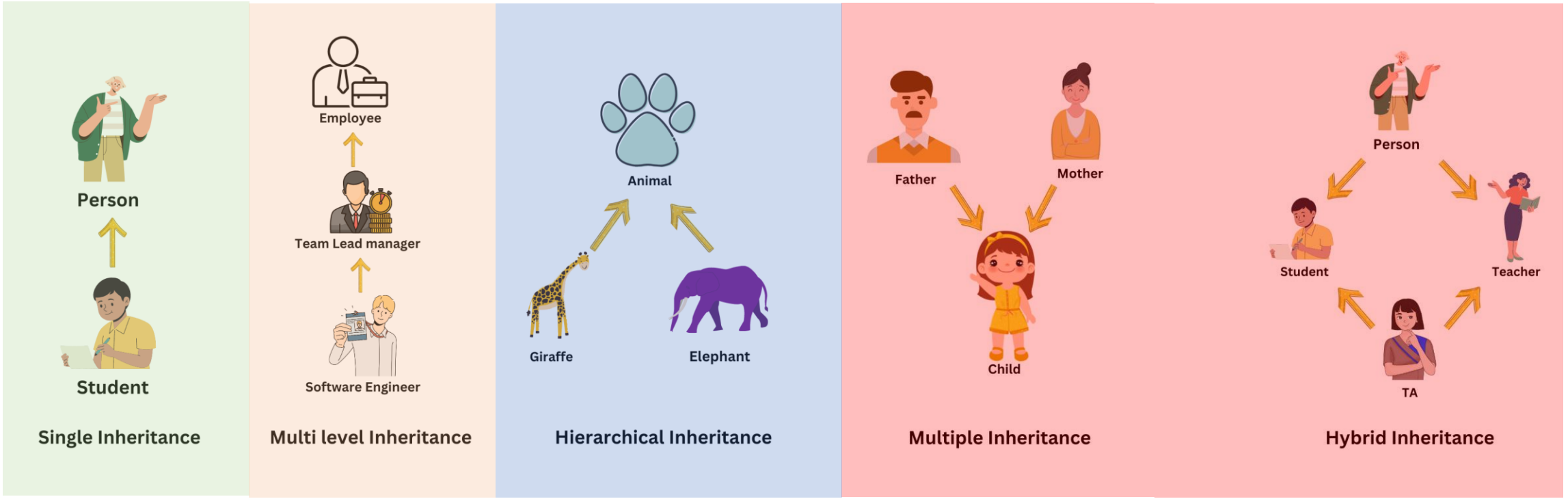
Output

Base class m2 method
y = 50

```
package inheritance;
public class Baseclass
{
 private int x = 30;
 protected int y = 50;
 private void m1()
 {
    System.out.println("Base class m1 method");
 }
protected void m2()
{
   System.out.println("Base class m2 method");
 }
}
public class Derivedclass extends Baseclass
{


 }
```

Private members can only be accessible by their getter and setter method in inheritance

Types of Inheritance



Can apply in Java

Can apply in C++

**Assignment for home: Implement all at home. And, submit your program at google classroom

No multiple inheritance in Java. Why?

Multiple inheritance in java is the capability of creating a single class with multiple super classes. Unlike some other popular object oriented programming languages like C++, **java doesn’t provide support for multiple inheritance in classes**. Java doesn’t support multiple inheritances in classes because it can lead to **diamond problem** and rather than providing some complex way to solve it, there are better ways through which we can achieve the same result as multiple inheritances.

Example-4

```java
package com.journaldev.inheritance;

// this is just an assumption to explain the diamond problem
//this code won't compile
public class ClassC extends ClassA, ClassB{

	public void test(){
		//calling super class method
		doSomething();
	}

}
```

```java
package com.journaldev.inheritance;

public abstract class SuperClass {

	public abstract void doSomething();
}
```

```java
package com.journaldev.inheritance;

public class ClassA extends SuperClass{

	@Override
	public void doSomething(){
		System.out.println("doSomething implementation of A");
	}

	//ClassA own method
	public void methodA(){

	}
}
```

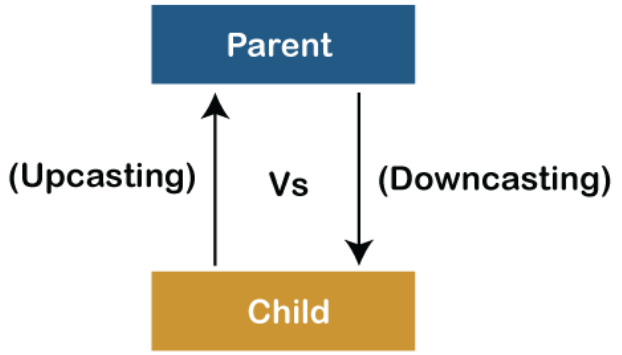
```java
package com.journaldev.inheritance;

public class ClassB extends SuperClass{

	@Override
	public void doSomething(){
		System.out.println("doSomething implementation of B");
	}

	//ClassB specific method
	public void methodB(){

	}
}
```

Notice that test() method is making a call to superclass doSomething() method.
This leads to the ambiguity as the compiler doesn't know which superclass method to execute. Because of the diamond-shaped class diagram, it's referred to as ***Diamond Problem*** in java. The diamond problem in Java is the main reason java doesn't support multiple inheritances in classes. Notice that the above problem with multiple class inheritance can also come with only three classes where all of them has at least one common method.

Multiple inheritance can be achieved in java through Interface or composition. We will learn it later.

Type casting- upcasting and down casting

- A process of converting one data type to another is known as Typecasting and Upcasting and Downcasting is the type of object typecasting.
- In Java, the object can also be typecasted like the datatypes. Parent and Child objects are two types of objects. So, there are two types of typecasting possible for an object, i.e., Parent to Child and Child to Parent or can say Upcasting and Downcasting.
- Typecasting is used to ensure whether variables are correctly processed by a function or not. In Upcasting and Downcasting, we typecast a child object to a parent object and a parent object to a child object simultaneously. We can perform Upcasting implicitly or explicitly, but downcasting cannot be implicitly possible.




```
String x = getStringFromSomewhere();
Object y = x;
// This will *always* work
```

```
Object x = getObjectFromSomewhere();
String y = (String) x;
// This might fail with an exception
```

Example-5: Upcasting

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

      Parent obj1 = (Parent) new Child();
      Parent obj2 = (Parent) new Child();
      obj1.PrintData();
      obj2.PrintData();
   }
}
```

```
class  Parent{
   void PrintData() {
      System.out.println("method of parent class");
   }
}
class Child extends Parent {
   void PrintData() {
      System.out.println("method of child class");
   }
}
```

Output

```
method of child class
method of child class
```

Example-5: Downcasting

```
public class Downcasting{

    public static void main(String[] args)
    {
        Parent p = new Child();
        p.name = “mayukhnila";
    }
}
```

```
//Parent class
class Parent {
    String name;

    // A method which prints the data of the parent class
    void showMessage()
    {
        System.out.println("Parent method is called");
    }

    // Performing overriding
    @Override
    void showMessage()
    {
        System.out.println("Child method is called");
    }
}
```

Upcasting is another type of object typecasting. In Upcasting, we assign a parent class reference object to the child class. In Java, we cannot assign a parent class reference object to the child class, but if we perform downcasting, we will not get any compile-time error. However, when we run it, it throws the "ClassCastException". Now the point is if downcasting is not possible in Java, then why is it allowed by the compiler? In Java, some scenarios allow us to perform downcasting. Here, the subclass object is referred by the parent class.

Output

Mayukhnila
4
Child method is called

Why we need Upcasting and Downcasting?

S.No	Upcasting	Downcasting
1.	A child object is typecasted to a parent object.	The reference of the parent class object is passed to the child class.
2.	We can perform Upcasting implicitly or explicitly.	Implicitly Downcasting is not possible.
3.	In the child class, we can access the methods and variables of the parent class.	The methods and variables of both the classes(parent and child) can be accessed.
4.	We can access some specified methods of the child class.	All the methods and variables of both classes can be accessed by performing downcasting.
5.	Parent p = new Parent()	Parent p = new Child() Child c = (Child)p;

Overriding methods

If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.

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

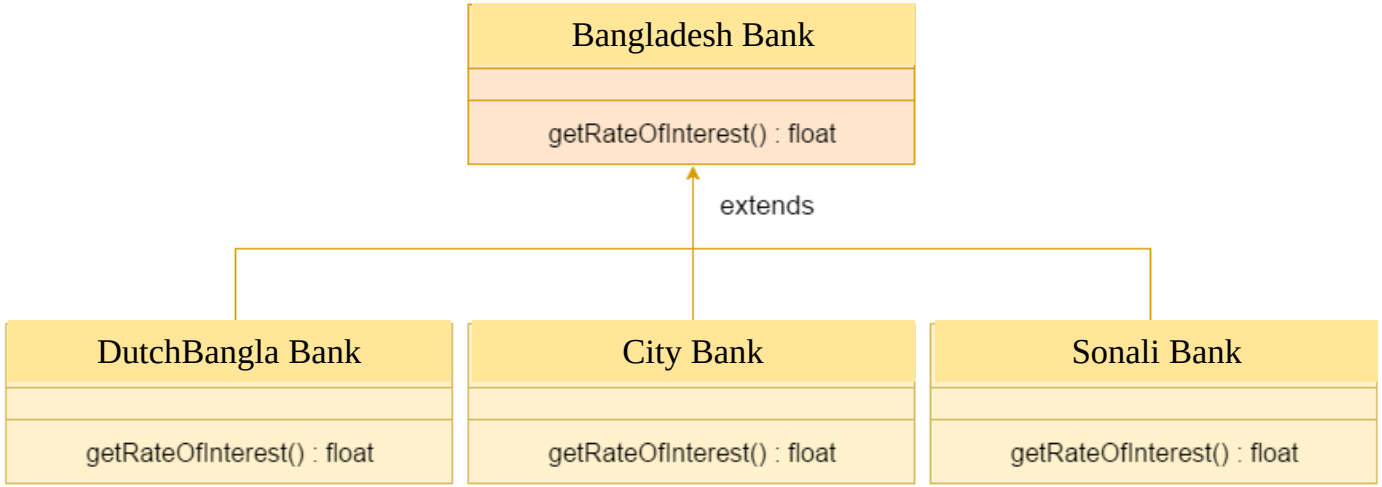
Usage of Java Method Overriding

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

Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

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



Example-6

Output

Bike is running safely

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

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

Calling superclass methods

Uses of super keyword

1. To call methods of the superclass that is overridden in the subclass.
2. To access attributes (fields) of the superclass if both superclass and subclass have attributes with the same name.
3. To explicitly call superclass no-arg (default) or parameterized constructor from the subclass constructor.

Example-7

```
class Main {
  public static void main(String[] args) {
    Dog dog1 = new Dog();
    dog1.printMessage();
  }
}
```

Output

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

```
class Animal {
  // overridden method
  public void display(){
    System.out.println("I am an animal");
  }
}
class Dog extends Animal {
  // overriding method
  @Override
  public void display(){
    System.out.println("I am a dog");
  }
  public void printMessage(){
    // this calls overriding method
    display();
    // this calls overridden method
    super.display();
  }
}
```

Instanceof instruction

The java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface).

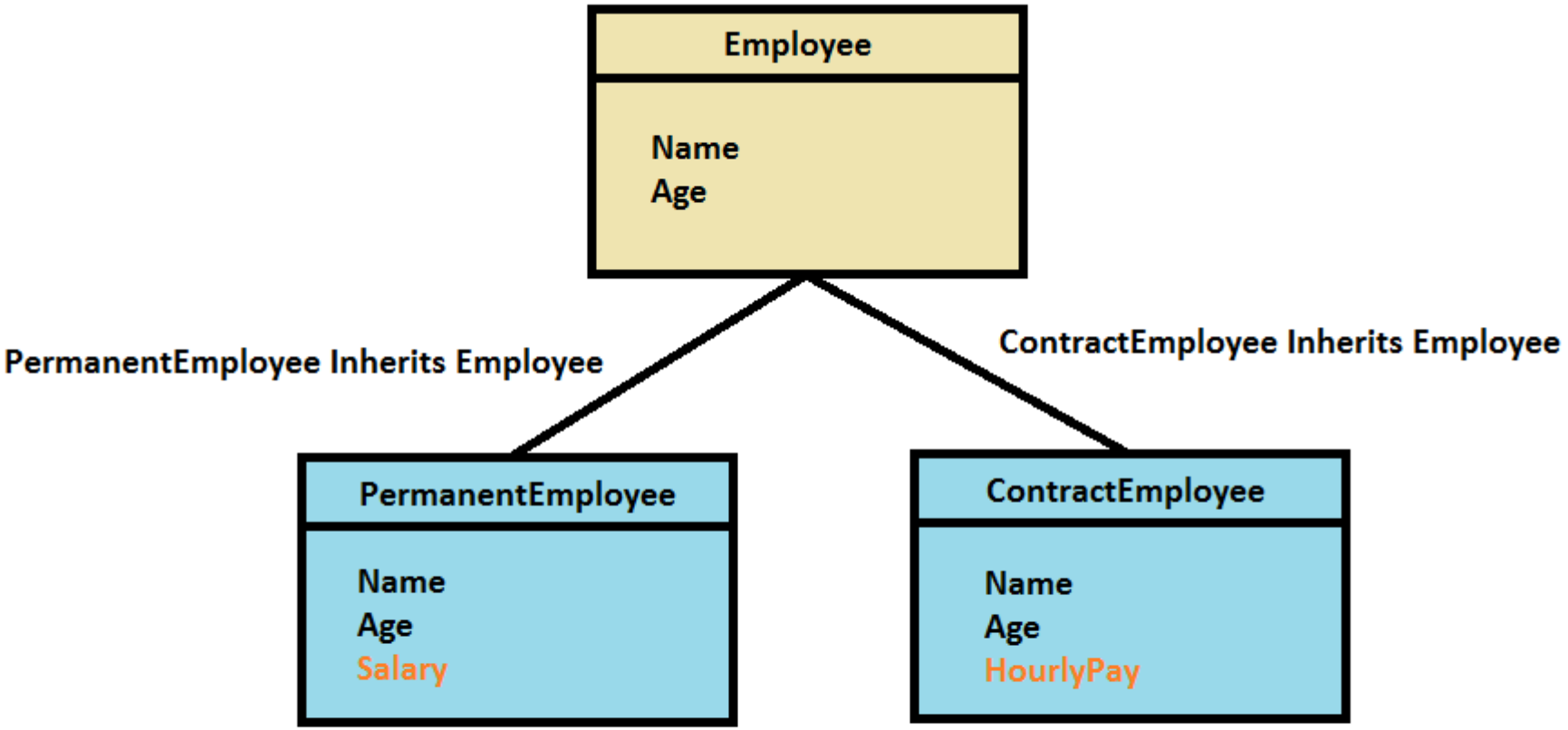
The instanceof in java is also known as type comparison operator because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

```
class Animal{}
class Dog1 extends Animal{//Dog inherits Animal

 public static void main(String args[]){
 Dog1 d=new Dog1();
 System.out.println(d instanceof Animal);//true
 }
}
```

Output

True

Field and Inheritance



Both PermanentEmployee and ContractEmployee has access to Name,Age Property of Parent Class Employee

Constructors and inheritance

Constructors in Java are used to initialize the values of the attributes of the object serving the goal to bring Java closer to the real world. We already have a default constructor that is called automatically if no constructor is found in the code. But if we make any constructor say parameterized constructor in order to initialize some attributes then it must write down the default constructor because it now will be no more automatically called.

```
// Main class
public class Test {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating object of sub class
        // inside main() method
        Derived d = new Derived(10, 20);

        // Invoking method inside main() method
        d.Display();
    }
}
```

Output

```
10
20
```

```
class Base {
    int x;
     // Constructor of super class
    Base(int _x) { x = _x; }
}

// Class 2
// Sub class
class Derived extends Base {
     int y;
     // Constructor of sub class
    Derived(int _x, int _y)
    {
         // super keyword refers to super class
        super(_x);
        y = _y;
    }
     // Method of sub class
    void Display()
    {
        // Print statement
        System.out.println("x = " + x + ", y = " + y);
    }
}
```

Nested class and inheritance

First Learn about Nested class then we will revise this topic

Abstract classes and inheritance

First Learn about Abstract class then we will revise this topic

Interface in Inheritance

First Learn about Interface then we will revise this topic

Happy Learning