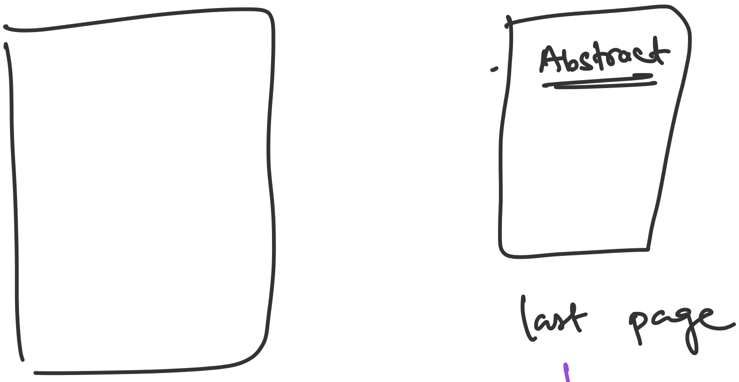
5__16-07-2022

Saturday, 16 July 2022 8:00 PM

Topics to cover -

- → Abstraction
- → Abstract class
- → Polymorphism
- → Design principle
- → SOLID principle

Abstraction



Research paper

last page

↓

Mention here important details

create solution

Create

(Abstraction)

Customer

Hiding all the unnecessary details

ways to make him happy

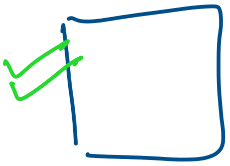
- simplicity
- Ease to use
- Security

Advantages of abstraction

→ Simplicity (Easy to use)

→ security

→ we can upgrade or improve the implementation without impacting end users.



iPad

iOS 15

↓

iOS 15.5

↓

iOS 16

ways to achieve Abstraction

- Abstract class
- Interface

Syntax for abstract class

```
public abstract Class A
{
        constructor

        public void Details ()
        {
            =
        }

        public abstract void Print();
}
```

f^n with declaration and definition

Just function declaration

```
public abstract class Dept {
   public Dept()
   {
      System.out.println("Abstract class constructor");
   }

   public void DeptDetails()
   {
      System.out.println("Department details function");
   }

   public abstract void SubDeptDetails();
}
```

```
public class CSE extends Dept {
   @Override
   public void SubDeptDetails() {
      System.out.println("CSE dept details");
   }
}
```

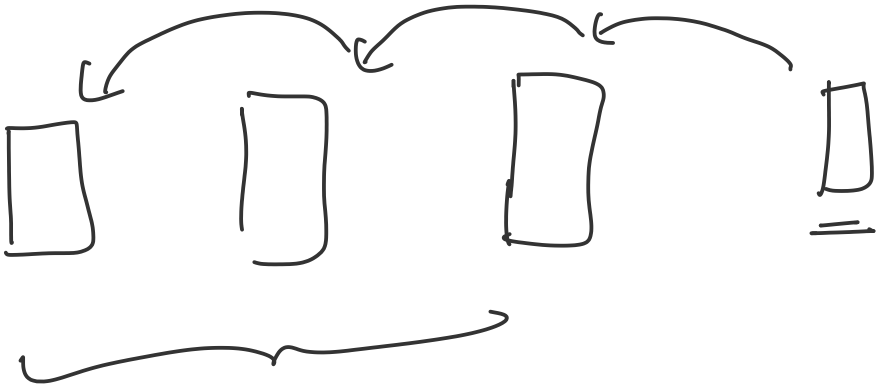
```
public class ECE extends Dept {
   @Override
   public void SubDeptDetails() {
      System.out.println("ECE dept details");
   }
}
```

Output :

```
Abstract class constructor
Department details function
CSE dept details

Abstract class constructor
Department details function
ECE dept details
```

Abstraction vs Encapsulation

Abstraction	Encapsulation
→ solves problem in the design level.	→ Solves problem in the implementation level.
→ Works on the outer layer.	→ works on the inner layer.
→ Focus on what to do.	→ Focus on how to do.
→ Hide unwanted data and give relevant information.	→ Hides the code and data in a single unit.



Abstraction at multiple layer.

Abstract class cannot be instantiated.

```
public abstract class A
{
    public abstract void func1();
    public void fun2()
    {
        ==
    }
}


class B extends A
{

    public void func1()
    {
        ==
    }
}
```

main method -

```
B objB = new B();

objB. func1();

objB . func2();


A objA = objB;

objA. func1();

objB . func2();
```

```
public class Program {
   public static void main(String[] args) {
      CSE cse = new CSE();
      cse.DeptDetails();
      cse.SubDeptDetails();
      System.out.println();

      ECE ece = new ECE();
      ece.DeptDetails();
      ece.SubDeptDetails();
      System.out.println();

      Dept dept = ece;
      dept.SubDeptDetails();
   }
}
```

Output:

```
Abstract class constructor
Department details function
CSE dept details

Abstract class constructor
Department details function
ECE dept details
```

ECE dept details

Purely abstract class

↳ When a class will have only abstract methods.

```
public abstract class A
{
        public abstract void f1();
        public abstract void f2();
}
```

f^n Declaration

A purely abstract behaves like an interface.

Q When to prefer abstract class and when to prefer interface.

```
public abstract class A
```

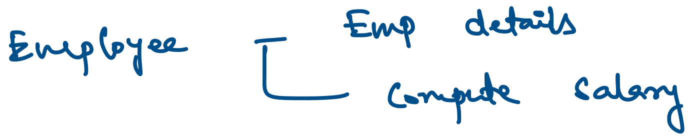
{

constructor ✓

f^n with definition

f^n declaration

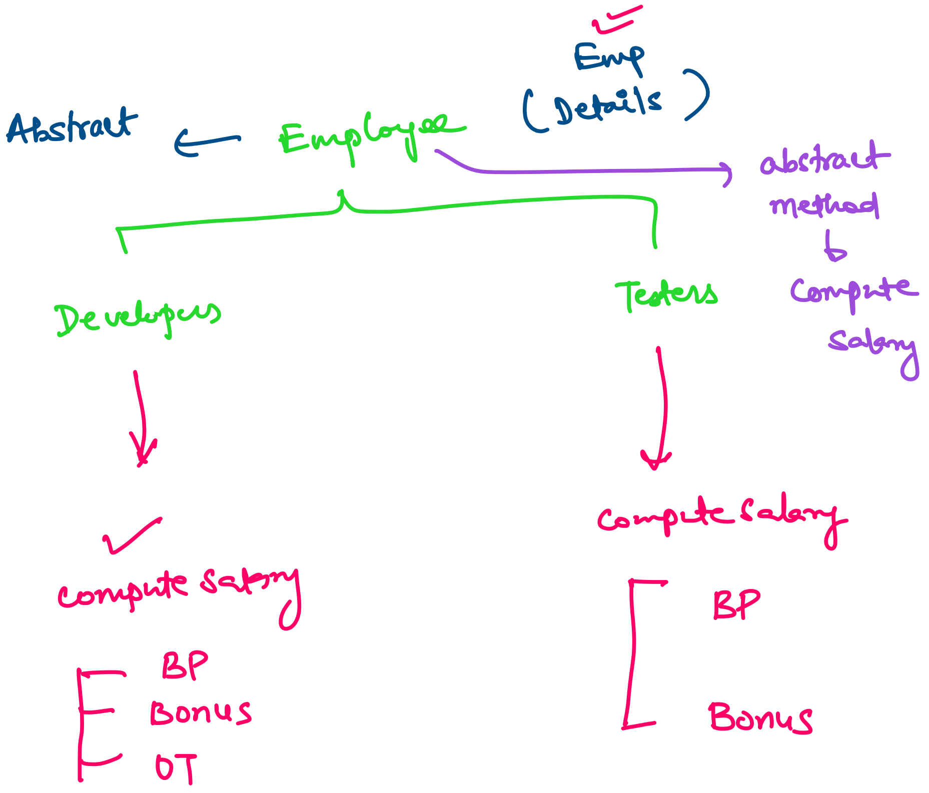
}



Employee

Emp details

Compute salary



Abstract ← Employee

✓ Emp (Details)

abstract method

compute salary

Developers

✓ compute salary

BP

Bonus

OT

Testers

compute salary

BP

Bonus

Regular class vs Abstract class

Abstract class - will force the

base class to provide implementation for the abstract methods.

Regular class – you will have the option to implement or not implement.

Note – Abstract class with no abstract methods will behave like a regular class only.

Abstract class vs interface

interface IEmployee

{

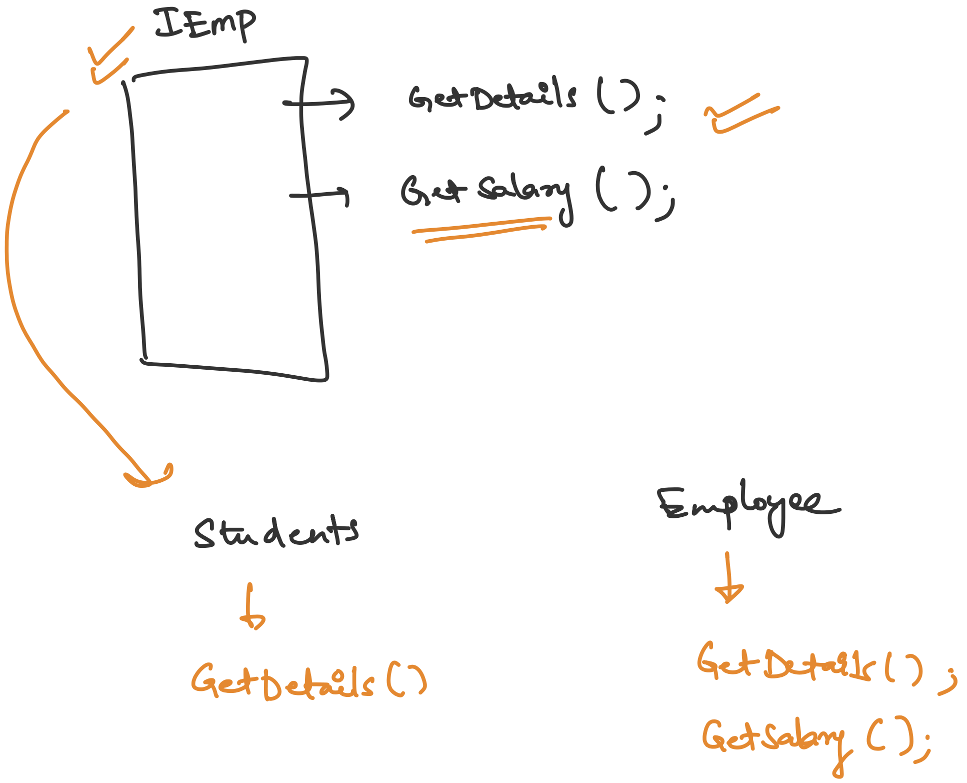
F^n 1 (); → Calculate bonus

F^n 2 (); → Is Emp of Month

}

Note –

When you want only f declaration then prefer interfaces.



IEmployee

↓

Split it into 2 parts

↓

1. IDetails

↳ GetDetails ();

Reused

2. ISalary

↳ GetSalary ();

Note -

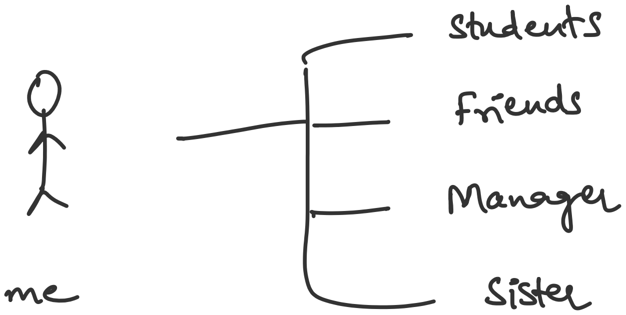
We can extend only one abstract class, even it is purely abstract class.

But we can implement one or more interfaces.

Poly morphism

Poly - Many

morp - form



Types of polymorphism

Static (

Run time |

Compile time polymorphism

Dynamic polymorphism

Static | Compile Time polymorphism

↳ At the compile time

↓

Method overloading

Method overloading

↳ 2 or more methods

↓

with same name

but different signatures

```
public int add (int num1, int num2)
{
    ______
}

int add (num1, num2, num3)
```

```
public int add (num1, num2)

{

}  =
```

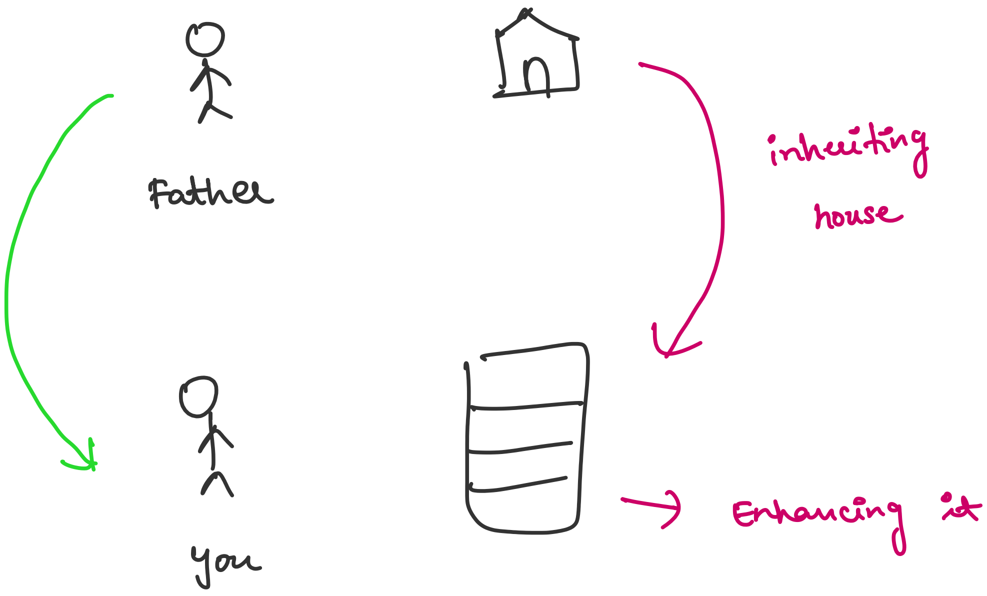
```
public class Sum {
   public int add(int num1, int num2)
   {
      return num1 + num2;
   }

   public int add(int num1, int num2, int num3)
   {
      return num1 + num2 + num3;
   }

   public int add(float num1, float num2)
   {
      return (int)(num1 + num2);
   }

   public float add(int num1, float num2)
   {
      return num1 + num2;
   }

   public float add(float num1, float num2, float num3)
   {
      return num1 + num2 + num3;
   }
}


public class Program {
   public static void main(String[] args) {
      Sum s = new Sum();
      System.out.println(s.add(1,2));
      System.out.println(s.add(10,20,30));
      System.out.println(s.add(1.1f, 2.2f));
      System.out.println(s.add(10, 2.3f));
      System.out.println(s.add(1.1f, 2.2f, 3.4f));
   }
}
```

Output:

```
3
60
3
12.3
6.7000003
```

Run time | Dynamic polymorphism

↳ Method overriding



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


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


public class Program {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();
    }
}
```

✓

Output:
Dog is eating

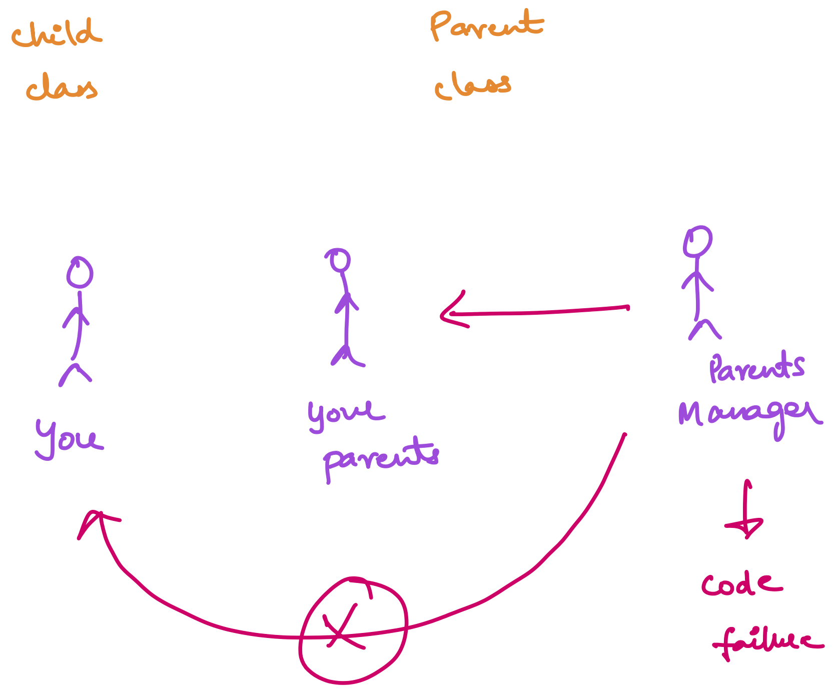
Note -

When the parent class and child class, both have exactly same method then it is called as method overriding.

Based on inheeitance

Dog d = new Animal();

↓ ↓



```
Dog d = new Animal();
```

This will be true if we have —

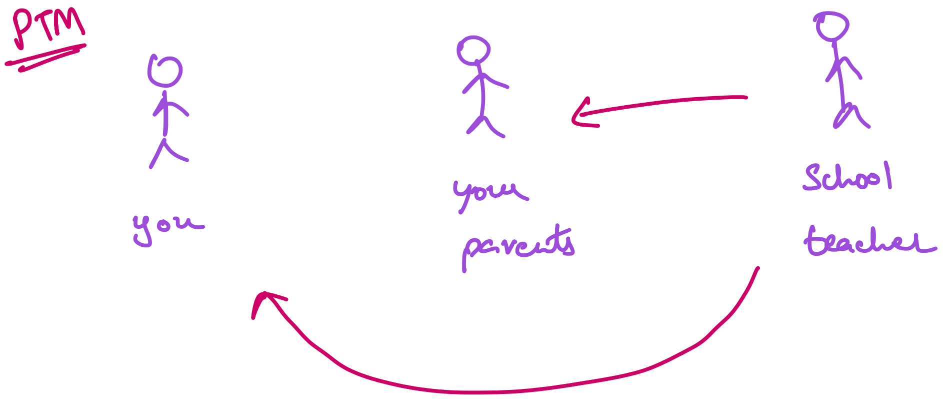
" Every animal is a dog". ✗

↓

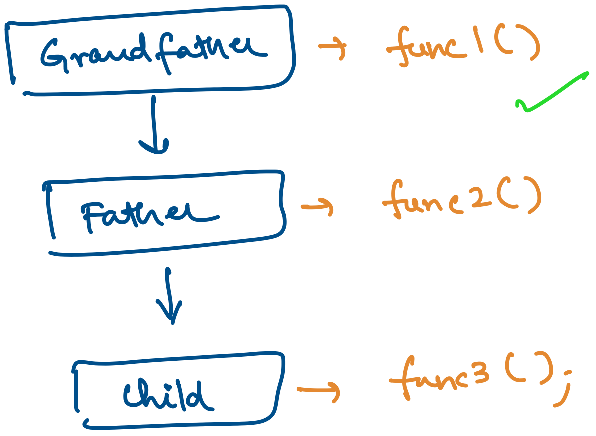
Thats why this representation is not true and it will give error.

2nd statement

```
Animal a = new Dog();
```



This is going to be correct representation.



```
Grandfather obj1 = new Father();

    obj1.func1();


Grand father obj2 = new Child();

    obj2.func1();


Parent obj3 = new Child();
```

obj3. func1();

obj3. func1();

Child obj4 = new child();

obj4. func1();

obj4. func2();

obj4. func3();

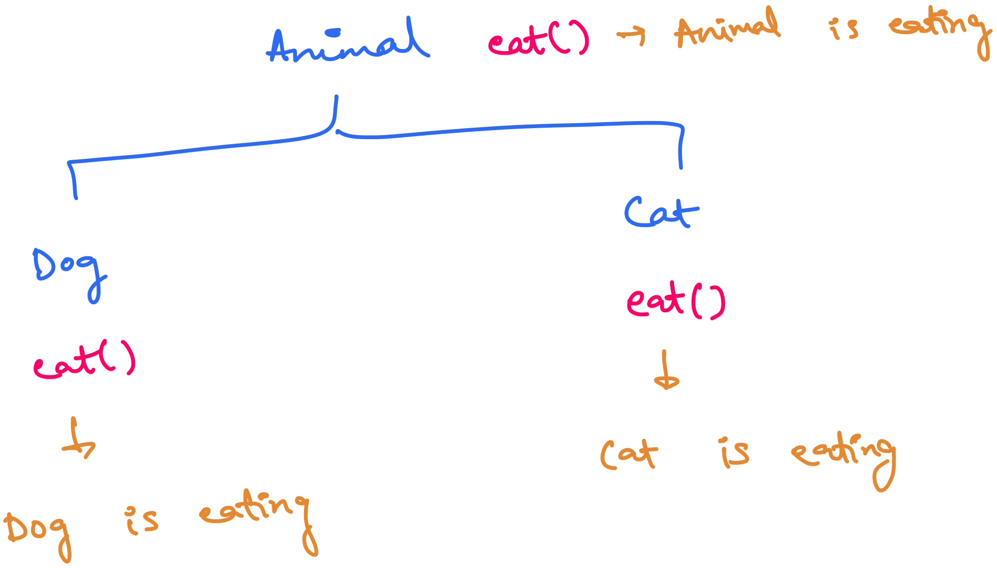
```
public class GrandFather {
   public void function1()
   {
      System.out.println("Function 1 : Grand father
class");
   }
}


public class Father extends GrandFather {
   public void function2()
   {
      System.out.println("Function 2 : Father class");
   }
}


public class Child extends Father {
   public void function3()
   {
      System.out.println("Function 3 : Child class");
   }
}


public class Program {
   public static void main(String[] args) {
     GrandFather obj1 = new GrandFather();
```

```
        obj1.function1();
        System.out.println();

        GrandFather obj2 = new Father();
        obj2.function1();
        System.out.println();

        GrandFather obj3 = new Child();
        obj3.function1();
        System.out.println();

        Father obj4 = new Child();
        obj4.function1();
        obj4.function2();
        System.out.println();

        Child obj5 = new Child();
        obj5.function1();
        obj5.function2();
        obj5.function3();
    }
}
```

Output:

```
Function 1 : Grand father class

Function 1 : Grand father class

Function 1 : Grand father class

Function 1 : Grand father class
Function 2 : Father class

Function 1 : Grand father class
Function 2 : Father class
Function 3 : Child class
```

Method overriding



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


public class Dog extends Animal {
   public void sleep()
   {
      System.out.println("Dog is sleeping");
   }

   public void eat()
   {
      System.out.println("Dog is eating");
   }
}


public class Cat extends Animal {
   public void eat()
   {
      System.out.println("Cat is eating");
   }
}

public class Program {
   public static void main(String[] args) {
     Animal a1 = new Animal();
```

```
    a1.eat();

    Animal a2 = new Dog();
    a2.eat();

    Animal a3 = new Cat();
    a3.eat();
  }
}
```

```
Output:
Animal is eating
Dog is eating
Cat is eating
```

a1 is printing – Animal is eating

a2 is printing – Dog is eating

a3 is printing – Cat is eating

If you look into all the variables, they are all of same type, i.e. Animal, the method which is getting called is also same, that is eat().

But which method is getting executed, this is depending upon which object is being assigned to this variable at the run time.

So same type of variable is behaving differently depending upon what object is attached (linked) to it, how program is executed, this is what we call as polymorphism because same behavior variable is behaving differently in multiple ways, that is what we can call it as dynamic or runtime polymorphism because it is taking values at runtime.