LOW LEVEL DESIGN

101

Paradigms in programming :-

- Procedural, Functional, Declaratie, Event Driven,
- Object Oriented Programming

OOP

- Thinking in terms of entities and objects.

- Object
    - ↳ Properties / attributes
    - ↳ Functionality / methods
    - ↳ State -> Object is a specific state for an entity.
    - ↳ Relationships with other objects.

- Object is an instance of a class

Abstraction, Encapsulati
Inheritance, Polymorphim

Consider Bird

```
Class Bird:
    color : RGB          }
    species : String     }
    weight : Float       }
    hight : float        }  Attributes
    dietType : String    }

    def fly ( )          }
    def eat ( )          }  functionality / methods
b = Bird ( ). # instantiating classes
```

Constructor:-

- It is a bound method ~~with~~
- Used to construct an object.

```
class Bird :
    def __init__ (self, name):     # initializer, not
        self.name = name                constructor
                                   # The lingo flies through
```

Consider the following function - fly (string type)

```
def fly (string type):
    if (type == "Eagle"):
        print ("Long wingspan")
    elif (type == "Penguin"):
        print ("Can't fly")
    elif (type == "Chicken"):
        print ("fast span, not good");
    elif (type == "Humming bird"):
        print ("Short span")
        ,
        ,
        ,
        ,
```

This style of coding is not good because it is:

1. Difficult to read
2. Hard to maintain → new type addition if used as libraries
3. Not extensible
4. Difficult to test

Solution to these problems → Abstraction

Abstraction :-

- Hides the details of implementation
- Not being concrete
- Not caring about the exact details

V1.0

```
Class Bird {
    ___
    ___
    ___
    void fly () {
        print ("flaps wings");
    }
}
Class Eagle extends Bird {
    // All the attribs & fncs get copied
    @override
    void fly() {
        print (" Spread Wings");
    }
}
Class HummingBird extends Bird {
    @override
    void fly(){
        print ("Vigorously fly/flap the wings");
    }
}
```

every eagle is also a bird

Note :-

- Abstraction goes hand-in-hand with generalization.
- Generalization :- Taking all the common things / attributes / behaviour & put it someplace
- You can achieve generalization by abstraction.

Benefits

① Readability improved

② Improved maintainability → Only need to change the behavian of a particula class.

③ Extensibility improved → Can easily create new birds types.

④ Testing improved

Drawbacks

- Consider a bird that can't fly - Dodo.

```
class Dodo extends Bird {
        // Since there is no flying capability for the
        // bird, we don't implement by overriding.

}
```

What happens if I call

```
        Dodo d= new Dodo( );
        d.fly( );?
```

Even though d cannot fly, and does not have any @overrides for fly, this fn call would call fly() from its parent class because of the inheritan

- This means there is no contract.
- We can ~~im~~ enforce a schema to have contracts using abstract methods.

- In Java, we have
    1. Abstract methods
    2. Abstract Interfaces.

V. 2.0

1. Abstract Methods :- These, although are defined in ther parent classes, their implementation is not.

```
class Bird :
        @ abstractMethod
        def fly ( )
class Eagle (Bird):
        def fly ( ):
            print ("Long Span")
```

- They force you to implement the behaviour of the function in the inherited / child classes.

Java

```
abstract class B {
        abstract void fly();
}
class C extends B {
        void fly() {

        }
}
```

-> One cannot create objects of an abstract class.

-> Abstract class can have concrete methods. This helps in generality

Note :-

① Bird b = new Eagle(); ✓ valid since every eagle is also a bird

② Eagle e = new Bird(); X Invalid.

Incase of ① => b.fly() will call the fly method inside Eagle.

This is runtime polymorphism.

Abstraction	Encapsulation
- Hiding details	- Hiding / controlling data ↑ ascess modifications

Encapsulation example:-

```
class Rectangle {
    int h; int b;
    public Rectangle (int h, int b) {
        this. h = h;
        this. b = b;
    }
    ~~void~~ int area() {
        return h*b;
    }
}
```

Main

```
    Rectangle    r = new Rectangle(2, 3);
                 r.h = 6; // here, we can change
                          // the value of h.
                          // If we do not wish to
                          // let the user do that,
                          // we make h & b Private
                                    access modifier
```

→ Python has no encapsulation. But we can simulate it using closures & inspectors.

Design a ~~Coffee~~ Vending Machine

Requirements

- Makes beverages based on some ingredients
- Should have a display
- Show the stock & price of items & the purchase money/cost
- User can interact with this menu & purchase beverages
- Cost of a drink is determined by the cost of the ~~items~~ ingredients.

Nouns → Entities ; adj → Attrib s/behaviours
verbs → behaviours/relationships.

Entities

Beverage
quality, type, flavor, cost
quench_thirst(), freeze(), boil(), steam(), fizz()

Vending Machie
height cost
color state
ON
OFF
Repair
Service
makeBeverage() GiveReceipt()
autoClean()

Ingredient

Display

Menu

User

Note:- At this point, I moved from Scaler Intermediate to Scaler Advanced. So the further lectures would be of Advanced level.