⇒ Object Oriented Programming

Top 3

4

10.15 pm.

OOP

I : Basics – (class, object, constructors destructors)

Problem

A PIE

II : Inheritance. → Theory (1–1.30)

III : Polymorphism – static / dynamic → practical (rem)

IV : Design a coffee machine

Why do we need $\overset{\text{Object.}}{\boxed{\text{OOP}}}$?

reading : writing.

80 : 20

- Structure the code
- relate to real world entities.

x → student.

y → mentor.

Transparent

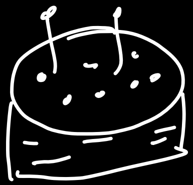
Maintenable → easy to test & fix things.

Extensible → add new features quickly.

Reusable → ability to reuse code.

⇒ **Procedural Programming**

Bake a Cake ⇓



Recipe

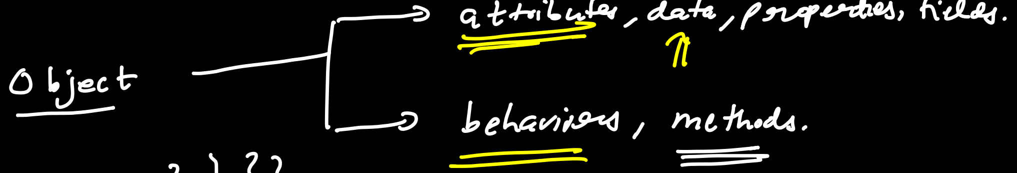
- Gather ingredients
- Follow recipe step by step.

⇒ **OOP**

→ Oven
→ Mixer
→ Pan.

} Ingredients.

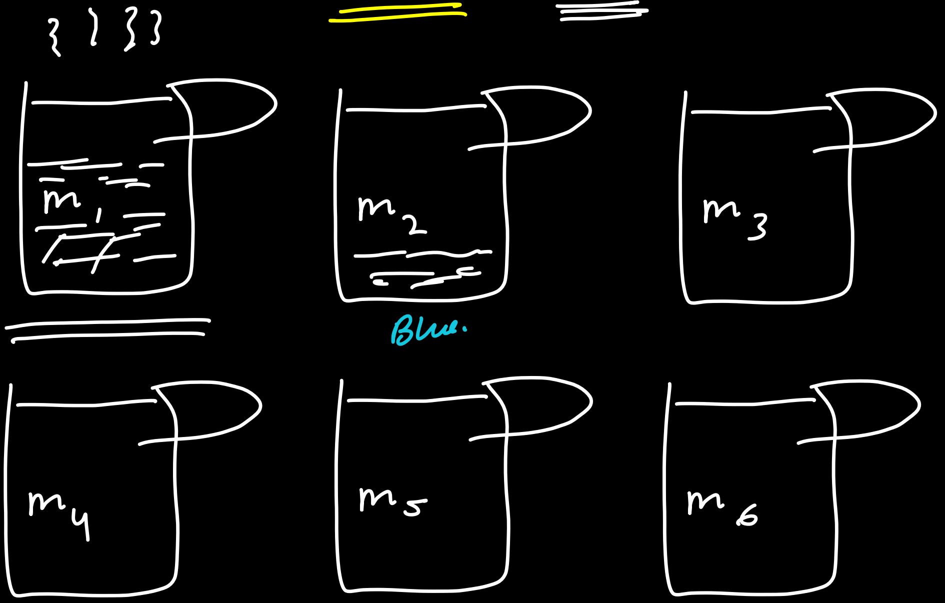
Interaction.



Different mugs.

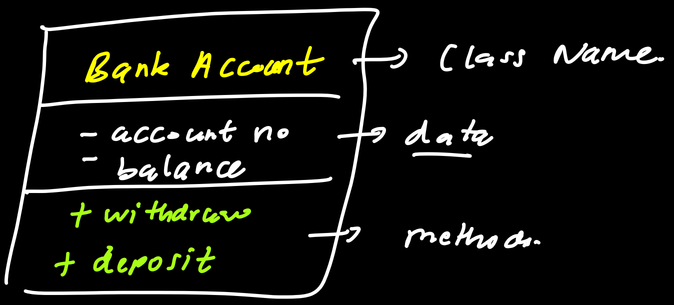
They are mugs

Type



Tangible?

Class diagram


Bank Account → Class Name.

- account no
- balance

→ data

+ withdraw
+ deposit

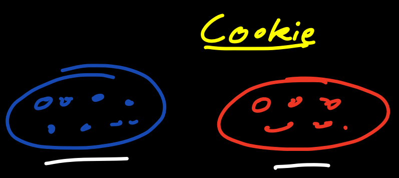
→ methods.

Classes

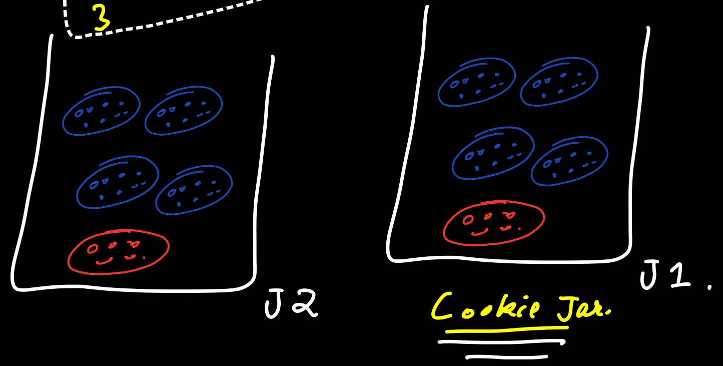
- template
- blueprint
- user defined data type.

```
int x;

Mug m1;
```

```
class Cookie{
    string type;
    int price;
}
```



```
class CookieJar {
    int numCookies;
    int capacity;
    vector<Cookie> data;

}
```



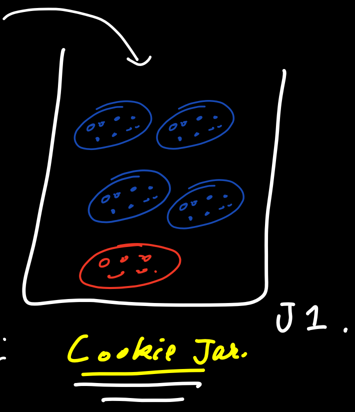
→ Method → only called using an object | s.length()

→ Function → called independently. | max(a,b)

→ Objects are created using the classes.

Defn

```
class CookieJar {
        int numCookies;
        int capacity;
        vector<Cookie> data;

        void addCookie(Cookie ck);

random ← Cookie requestCookie();

    };
```



Instantiate

```
CookieJar j1;
```

Support some default value while creating object.

Constructor

```
class CookieJar {
→ private: int numCookies ; =0.
           int capacity;
           vector<Cookie> data;

→ public:  CookieJar ( ) {
              numCookies = 0;
              capacity =100;
           }
           void addCookie ( Cookie ck),

        ← Cookie requestCookie( );

};
```

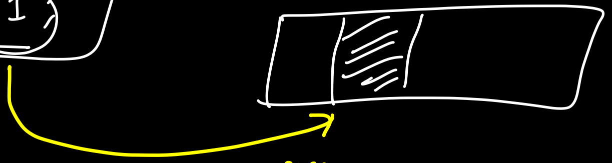
12 bytes.

30 bytes.

Constructor

- same name as class
- no return type.
- always public.
- can have args.

```
CookieJar j1;
```



Once object created → allocate memory resource.

→ Default constructor auto created if we don't create one.

```
CookieJar(int count){

    numCookies = count;
    capacity = 100;
}
```

```
CookieJar j1;

CookieJar j2(100);
```

When object is created.

How you create

↓

will determine which constructor is called.

Scenario-I

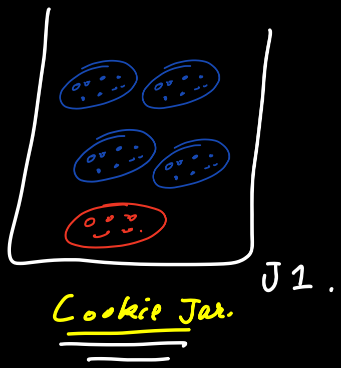
```
                CJ
      class  CookieJar {
public :
         int numCookies ;
         int capacity;
         vector<Cookie> data;

         CookieJar ( ) {
             numCookies = 0;
             capacity =100;
         }
        void  addCookie ( Cookie ck)

        Cookie  requestCookie( ) ;

   };
```

```
→ CJ   j1;

→ j1.numCookies = -100;   ⇐
```

Direct access

↑ dot access

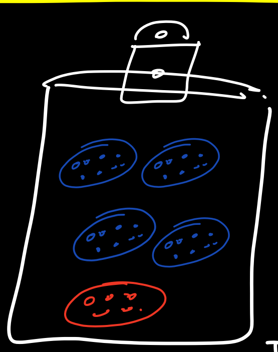
```
j1.addCookie( c);
```



Scenario II

```
class CookieJar {
private:
    int numCookie;
    int capacity;
    vector<Cookie> data;

public:
    CookieJar() {
        numCookies = 0;
        capacity=100;
    }
    void addCookie(Cookie ck)
    Cookie requestCookie();
    int getNumCookies(){
        return numCookies;
    }
}
```

can be accessed internally



J1.

Cookie Jar.

```
CJ j1;
```

j1.numCookies ✗

```
j1.addCookie( _ )
```

→ check internally whether we can add a cookie

→ update the data if possible.

ENCAPSULATION

binding the methods with data

→ public methods allow you to interact with the object.

→ Indirect Access of data using

Getter methods.

Setter methods.

```
void setNumCookies( int result){
    numCookies = result;
}
```