Start where you are. Use what you have. Do what you can

What is OOPs

Object-Oriented Programming (OOP) is a programming paradigm built around the concept of "objects."

class

object = Diff. objects : But common properties

```
class Animal
{
    sound
    legs
}
```

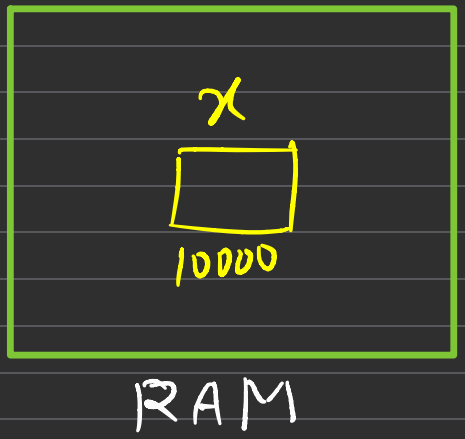
obj1 (Dog)
obj2 (Lion)
obj3 (Tiger)

int (logical)

int x (physical)

int x = 10

- int → Data Type
  - What type of operations we can perform on given data
  - ① Type of data we can store
- x → variable
  - Address
- 10 → value



~~int x = 10~~ ~~float x = 40.60~~

int y = 10.40 ~~char c = 90~~

x float z = 'a'

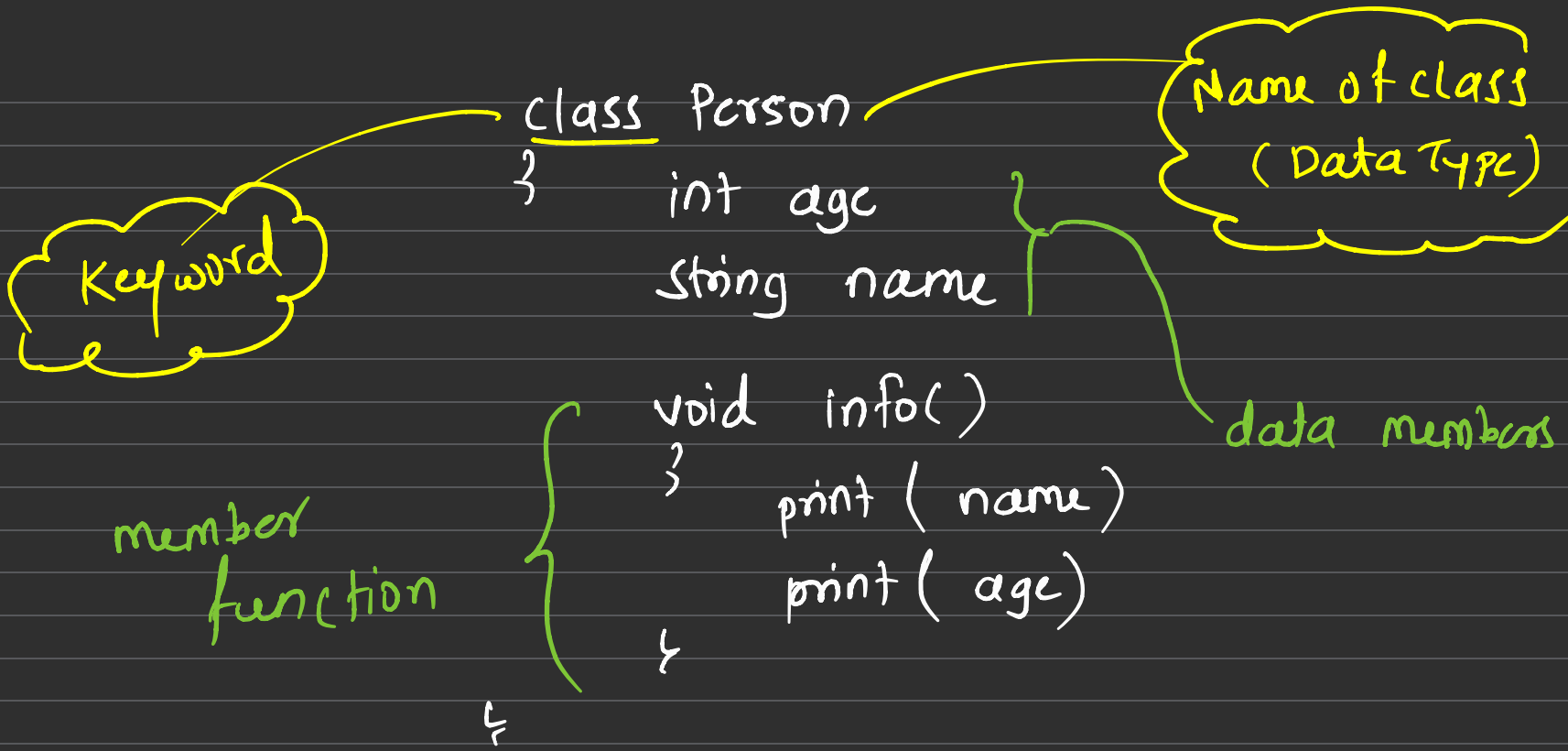
Class

- User defined data type, which contains
  1. Data members (Variable)
  2. Member functions (function)

* Syntax

```
class className
{
    // data members

    // member functions
};
```



Object

- It is Real world entity.
- Instance of class which tells what kind of operations we can perform on class

* Syntax:

```
className   objName
```

Eg: Person pramod

(ClassNm) object

Person obj

int y = 10

print (y) //10

Accessing members of the class

— We can access the members of class using '(.) dot' operator

```
class Person
{
    int age
    string name
    void info()
    {
        print(nm)
    }
};
```

```
Person sushant

sushant.age
sushant.name

sushant.info()
```

Pillars of Object Oriented Programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Identify which is Encapsulation

① ✓

```
class Volume{
    int radius;
    int diameter;
public:
    Volume(){
        radius = 20;
        diameter = 30;
    }

    void info(){
        int v = radius * diameter;
        cout << v;
    }
}
```

②

```
class Volume{
    int radius;
    int diameter;
public:
    Volume(){
        radius = 20;
        diameter = 30;
    }

    void info(){
        int height = 30;
        int volume = 40 * height;
        cout << volume;
    }
}
```

Encapsulation

Encapsulation is binding of data members and member functions which is performing the operation on data members into single unit.

```
class Square{
    int lenght;
    int breadth;

public:
    Square(int lenght, int breadth){
        this -> lenght = lenght;
        this -> breadth = breadth;
    }

    void printTheSquare(){
        int height = 49;
        int sq = lenght * height;
        cout << sq << " ";
    }

    void printTheVolume(){
        int height = 39;

        int vol = height * breadth;
        cout << vol;
    }

    void info(){
        cout << lenght << " " << breadth;
    }
};
```

Encapsulation internally implements "Data Hiding"

Access Specifier

Encapsulation: Real life examples

An organization consists of several departments like the production department, purchase department, sales department, and Accounts department. It combines all these departments together and formed the organization.

Abstraction

Abstraction means displaying only essential information and hiding the details.

Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

arr() = { 7, 6, 3, 1, 2 }

Arrays.sort(arr)

sort(arr, arr+n)

1.) Header file

2) Class

- Header file

  sort (arr, arr+n)

  --gcd (a, b)

- Class

> 💡 **NOTE**
>
> Encapsulation and Abstraction go hand in hand since in encapsulation we use **Data Hiding** which is also used to hide the detailing of the properties.

Abstraction: Real life examples

The man only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of an accelerator, brakes, etc. in the car.

Constructor

Mainly used to initialise the object of the class.

- Rules:
  - Name of constructor = Name of class
  - No return type, not even void.
  - Must be public

Because implicit Return type is class itself.

```cpp
class Student{

public:
    int age;
    string name;
    string branch;
    bool isInterestedInAI;
    bool isInterestedInSports;
    bool isInterestedInStartup;
    string ofCampusBranch;
    string clubs;

    void informationOfPerson(){
        cout << "Name of the Student: " << name << "\n";
        cout << "Age of the Student: " << age << "\n";
        cout << "Branch of the Student: " << branch << "\n";
        cout << "is Interested In AI: " << isInterestedInAI << "\n";
        cout << "is Interested In Sport: " << isInterestedInSports << "\n";
        cout << "is Interested In Startup: " << isInterestedInStartup << "\n";
        cout << "club: " << clubs << "\n";
        cout << "of Campus Branch: " << ofCampusBranch << "\n";
    }
};
```

int x (decln)

int x = 10 (initialize)

```cpp
Student rohan;
rohan.age = 20;
rohan.name = "Rohan";
rohan.branch = "CSE";
rohan.isInterestedInAI = true;
rohan.isInterestedInSports = false;
rohan.isInterestedInStartup = true;
rohan.clubs = "Coder";
rohan.ofCampusBranch = "none";

rohan.informationOfPerson();
```

Types of Constructors

Default Constructor

Paramerterised Constructor

```
void fun()
{
}
```

```
main()
{
    fun()
}
```

Default constructor

```
class Person{
    int age;
    string name;

    void info(){
        cout << name;
        cout << age;
    }
}
```

Person obj

```
class Person{
    int age;
    string name;

public:
    Person(){
        name = "Pramod";
        age = 20;
    }

    void info(){
        cout << name;
        cout << age;
    }
}
```

Person pramod

pramod.info()

* Parameterized constructor

```
class Student{
private:
    int age;
    string name;
    string branch;

public:

    Student(string name, int age, string branch){
        this -> name = name;
        this -> age = age;
        this -> branch = branch;
    }

    void informationOfPerson(){
        cout << "Name of the Student: " << name << "\n";
        cout << "Age of the Student: " << age << "\n";
        cout << "Branch of the Student: " << branch << "\n";
    }
};
```

class

? name

Student (name)

?

this → name = name

}

}