Object Oriented Programming

MAS Foundation Course WS19
Hochschule Bonn-Rhein-Sieg

Lokesh Veeramacheneni

Outline

- Object Oriented Programming
- Principles of Object Oriented Programming
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Why Object Oriented Programming?

1. Real world Modelling can be done with ease.
    a. Entities in real world has attributes and behaviors of their own.
1. Restricted Access to data can be achieved.
2. It makes the code modular and easy to maintain.

Object Oriented Programming

- Basic parts of Object Oriented Programming includes:
  1. **Class:** It defines the template of object, i.e. What type of data must it store and typically consists of what each function does. It consists of Instance members and functions.
  2. **Object:** It is the instance of class. They store the data when the program is executed.

Example:

```
class Animal
{
    int number_of_legs;
    string special ability;
    public:
        void eat(){
            cout<<"Eating..[Animal class]"<<endl;
        }
};
```

Instance Members

Functions

CLASS

Object Oriented Programming - **Lokesh**

```
int main(){
    int stomach_status;
    Animal X;
    Carnivorous Wolf;
    Wolf.eat();
    X.eat();
    stomach_status = X.eat("fish");
```

Object of Animal Class

Principles of Object Oriented Programming:

There are four major principles of Object Oriented Programming (OOP).

1. Inheritance
2. Polymorphism
3. Abstraction
4. Encapsulation

Inheritance

- Inheritance is the process where one object acquires the properties(public and protected) of all its parent class object.

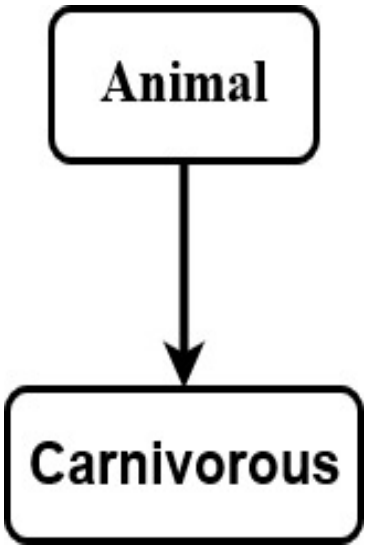
Why Inheritance:

- Inheritance helps in code reusability i.e. the parent class members can be used in child class without declaring them in child class once again.

Types of Inheritance:

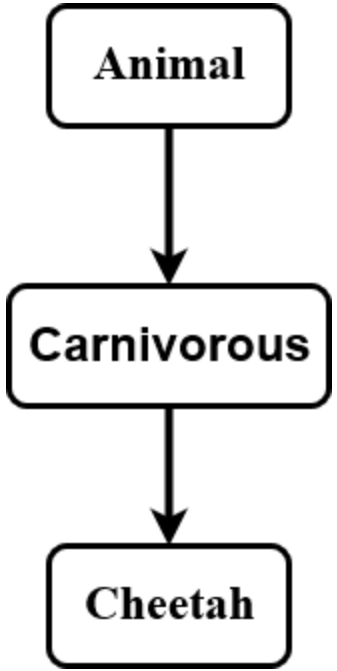
1. Single Inheritance
2. Multiple Inheritance
3. Multi-level Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

Single Inheritance:



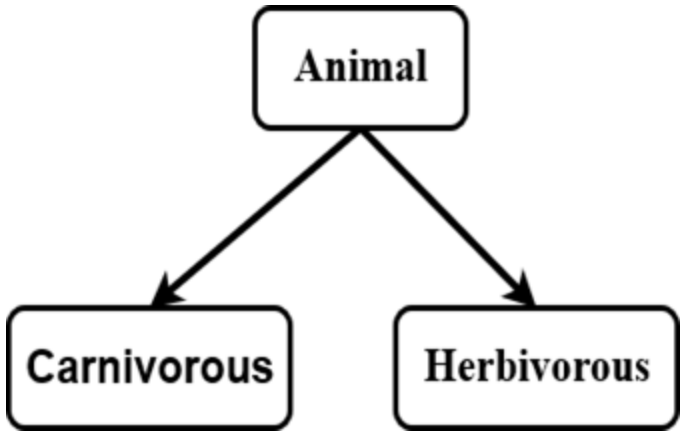

```
class Animal
{
    public:
        void eat(){
            cout<<"Eating..[Animal class]"<<endl;
        }
};
// Single Inhertiance (Animal-->Carnivorous)
class Carnivorous : public Animal
{
    public:
        void characteristic(){
            cout<<"Carnivorous have canine tooth..[Carnivorous class]"<<endl;
        }
};


int main(void) {
// Single Inheritance
    Carnivorous wolf;
    cout<<""<<endl;
    wolf.characteristic();
    wolf.eat();
    cout<<""<<endl;
```

Multi-level Inheritance:



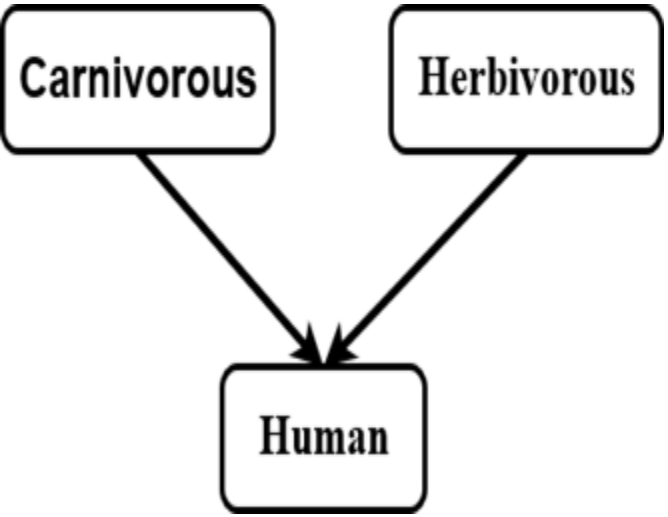

```
class Animal
{
    public:
        void eat(){
            cout<<"Eating..[Animal class]"<<endl;
        }
};
// Single Inhertiance (Animal-->Carnivorous)
class Carnivorous : public Animal
{
    public:
        void characteristic(){
            cout<<"Carnivorous have canine tooth..[Carnivorous class]"<<endl;
        }
};

// Multi level Inheritance (Animal-->Carnivorous-->Cheetah)
class Cheetah : public Carnivorous
{
    public:
        void run(){
            cout<<"Cheetah outruns Ussain Bolt..[Cheetah class]"<<endl;
        }
};

int main(void) {
// Multi-level Inheritance
    Cheetah C;
    C.run();
    C.eat();
    C.characteristic();
    cout<<""<<endl;
}
```

Hierarchical Inheritance:



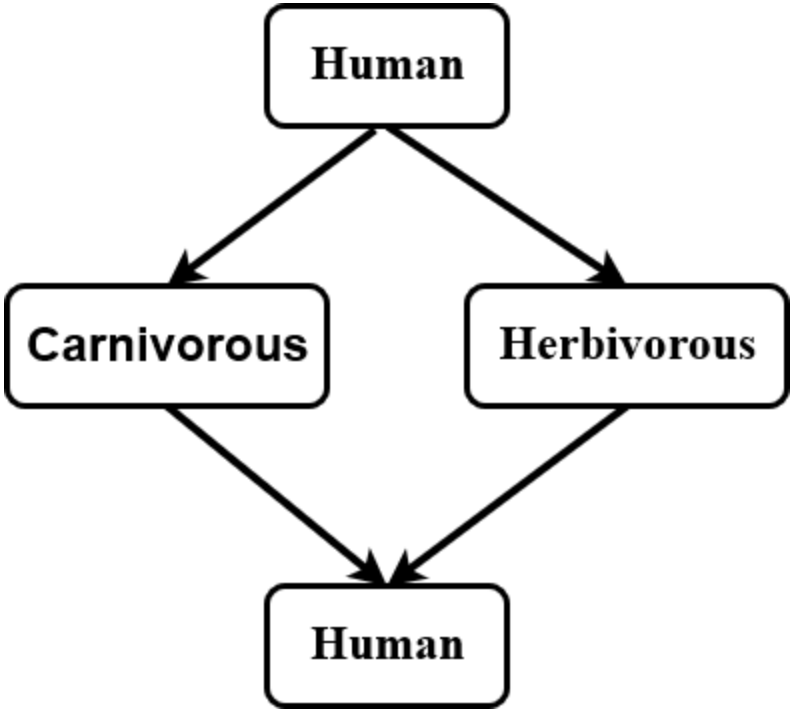

```
class Animal
{
    public:
        void eat(){
            cout<<"Eating..[Animal class]"<<endl;
        }
};
// Heirarchical Inheritance (Animal-->Carnivorous,Herbivorous)
class Carnivorous : public Animal
{
    public:
        void characteristic(){
            cout<<"Carnivorous have canine tooth..[Carnivorous class]"<<endl;
        }
};
// Heirarchical Inheritance (Animal-->Carnivorous,Herbivorous)
class Herbivorous : public Animal
{
    public:
        void quality(){
            cout<<"Herbivorous have stubby teeth..[Herbivorous class]"<<endl;
        }
};
int main(void) {
// Heirachical Inhertiance
    wolf.eat();
    Herbivorous gazelle;
    gazelle.eat();
    gazelle.quality();
    cout<<""<<endl;
}
```

Multiple Inheritance:




```
class Carnivorous : public Animal
{
    public:
        void characteristic(){
            cout<<"Carnivorous have canine tooth..[Carnivorous class]"<<endl;
        }
};
class Herbivorous : public Animal
{
    public:
        void quality(){
            cout<<"Herbivorous have stubby teeth..[Herbivorous class]"<<endl;
        }
};

// Multiple Inheritance (Carnivorous,Herbivorous-->Humans)
class Human : public Herbivorous, public Carnivorous
{
    public:
        void think(){
            cout<<"Humans are able to think at complex level..[Human class]"<<endl;
        }
};
int main(void) {
// Multiple Inheritance
    Human h;
    h.think();
    h.quality();
    h.characteristic();
}
```

Hybrid Inheritance:



Polymorphism

- Informally, polymorphism is one in many forms. For Example, A function can have different behaviors with same function name.
- There are two kinds of polymorphism:
  1. Static Polymorphism
  2. Dynamic Polymorphism
- **Static Polymorphism** is achieved by **Function Overloading**. Function overloading is having the same function in same class with different function signatures.
- **Dynamic Polymorphism** is achieved by **Function Overriding.** Function overriding is having the same function and function signature in both parent and child classes.

Static Polymorphism:

- In this example, the **Animal** class has two functions **eat** which are differed by their function signature where one function has **no arguments** and **no return type.**
- The other **eat** function accepts a **String argument** and **returns an integer value.**

```
class Animal
{
    int stomach_capacity;
    public:
    // Function Overloading
    void eat(){
        cout<<"Animal Eats for survival..[Animal Class]"<<endl;
        stomach_capacity = 1;
    }
    int eat(string food){
        if (stomach_capacity == 1)
        {
            cout<<"Animal Stomach is full..[Animal Class]"<<endl;
            return 1;
        }
        else
        {
            cout<<"Animal is Eating"<<food<<"..[Animal Class]"<<endl;
            return 0;
        }
    }
};
int main(){
    int stomach_status;
    Animal X;
    X.eat();
    stomach_status = X.eat("fish");
}
```

Dynamic Polymorphism:

- There exists two **eat** functions one in Animal and other in its child class.
- Both the methods have **same function signature** but they behave according to the object which called it.

```
class Animal
{
    public:
    void eat(){
        cout<<"Animal Eats for survival..[Animal Class]"<<endl;
        stomach_capacity = 1;
    }
};
class Carnivorous : public Animal
{
    public:
    // Function Overriding
    void eat(){
        cout<<"Carnivorous animal eats meat..[Carnivorous Class]"<<endl;
    }
};
int main(){
    Animal X;
    Carnivorous Wolf;
    Wolf.eat();
    X.eat();
}
```

Virtual Functions:

- Virtual Functions can also be used for achieving Dynamic Polymorphism.

Why Virtual Functions are needed?

- Virtual Functions are used for late binding that is implementation of the method is determined by the compiler at the runtime where as in previous implementation the method which must be called is decided at compile time (early binding).

Need for Virtual Functions:

- In this kind of implementation when the **Animal object pointer** and **Carnivorous object pointer** is sent as argument.
- When this **function** in **need_virtual class** is called the result is the execution of **eat function** in **Animal class.**

```
class Animal{
    void eat(){
        cout<<"Animal Eats for survival..[Animal Class]"<<endl;
        stomach_capacity = 1;
    }
};
class Carnivorous : public Animal
{
    public:
    // Function Overriding
    void eat(){
        cout<<"Carnivorous animal eats meat..[Carnivorous Class]"<<endl;
    }
};
class need_virtual
{
    public:
    void function(Animal *y){
        y->eat();
    }
};
int main(){

    Animal *calf;
    Carnivorous *wolf;
    need_virtual yes;
    yes.function(calf);
    yes.function(wolf);
}
```

Implementation:

```
class Animal{
    // Virtual Functions
    virtual void kill(){
        cout<<"Typically,animals kill for survival..[Animal Class]"<<endl;
    }
};
class Carnivorous : public Animal
{
    public:
    void kill(){
        cout<<"Some Carnivorous kill for pleasure..[Carnivorous Class]"<<endl;
    }
};
int main(){
    //Vitual Functions
    Animal *a;
    Carnivorous human;
    a = &human;
    cout<<""<<endl;
    // Usig a parent class pointer to call virtual and normal function
    a->kill();
    a->eat();
}
```

Abstraction

- Abstraction is hiding the details and showing necessary details to the end-user.
- Abstraction helps in code reusability and also provide some level security since implementation to end-user is unknown.
- In the side example the car number and car model are not accessible directly the user can only register a car and get details.

```
class Car{
    private:
    int car_number;
    string car_model;
    public:
    void register_car(int number, string model){
        car_number = number;
        car_model = model;
    }
    void get_details()
    {
        cout<<"Car number is: "<<car_number<<endl;
        cout<<"Car model is: "<<car_model<<endl;
    }
};
int main(){
    Car opel;
    opel.register_car(124,"Asta");
    opel.get_details();
}
```