Object Oriented Programming using C++

Prepared by:

Dr. Bakshi Rohit Prasad
(M.Tech., PhD, IIIT-Allahabad)

Topics to be covered

- Inheritance
- Friend functions
- Access specifiers – *public private and protected*
- Types of Inheritance
  - Single
  - Multiple
  - Multi-level
  - Hierarchical
  - Hybrid
- Function overriding
- Virtual functions

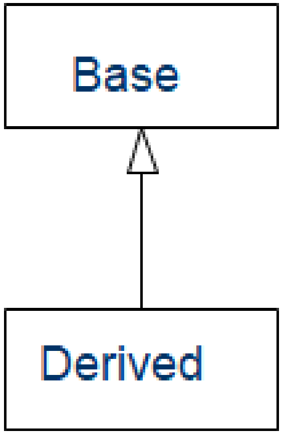
Inheritance

- It can be defined as the ability of an object oriented programming language where one class derives/inherits the features/properties of another class.

- Class which inherits the properties is known as sub-class, child class or derived class

- Class whose properties are inherited is known as super-class, parent class or base class

- Inheritance is helpful in:
  - Reusing the existing features of base class
  - Adding new features/behaviors to base class by adding them in derived class
  - Re-defining some of the behaviors of base class in derived class



Some real life example of inheritance

- Every human inherits the features of mammals

- Every mal female inherits features of human beings

- Two wheeler and Four wheeler are example of class of vehicles, thus inherit its property

- Rectangle, triangle, pentagon all belong the class of polygon, hence, inherit the properties of polygon

Inheritance

- **The derived class:**
  - will inherit all the attributes and functions of the base class
  - can have additional attributes
  - can have additional functions/methods
  - can override functions/methods of the base class

- **Syntax:**

```
class derived_class_name  :  access_specifier base_class_name
{
        list of data members of this class
        list of member functions of this class
 };
```

Example of Inheritance

- **Consider the class Rectangle:**

```
 class Rectangle {
          public:
                 int length, breadth;
          public:
                 int area(int length, int breadth) {
                             return length*breadth;
                 }
 };
```

- **Creating a class ‘Box’ which inherits Rectangle:**

```
class Box : public Rectangle {
     public:
                 int height;
      public:
                 void volume() {
                      cout<<“volume is: ”<<length*breadth*height;
                  }
};
```

Example of Inheritance

- **Consider the class Rectangle:**

```
class Rectangle {
        public:
              int length, breadth;
        public:
              int area(int length, int breadth) {
                        return length*breadth;
              }
};
```

NOTE:

1. public inheritance is most commonly used.
2. However, private or protected inheritance may also be used which is rare.
3. If the method of inheritance is omitted it **defaults to private.**

- **Creating a class ‘Box’ which inherits Rectangle:**

```
class Box : public Rectangle {
    public:
              int height;
    public:
              void volume() {
                  cout<<“volume is: ”<<length*breadth*height;
              }
};
```

Inheritance and Constructors

- **Constructor:**
  - of derived class should invoke the constructor of its base class
  - Base constructor invocation may be omitted *iff* the base class has a default constructor which will be automatically invoked

Example: Inheritance and Constructors

```
class Base {
     public:
             int data;

    public:
              Base(int x)  {          //parameterized constructor

                 data = x;

                 cout << "base constructor called recently" << data <<endl;

            }
 };

class Derived :  public Base
 {

     public:
              float moredata;
       public:
              Derived(float i, int j) : Base(j)   {        //parameterized constructor

               moredata = i;

               cout << "base constructor called recently" << data <<endl;}
 };
```

Call to base constructor

Friend Function

- Friend function of a class X:
  - is defined outside the class X
  - can access the private, protected (and public obviously) members of the class X
  - is not a member of class X

- **Declaration Syntax:** using *'friend'* keyword

```
class X {
        private:      list of private data members of class X
        protected:  list of protected data members of class X
        public:
                list of member functions of class X
                friend return_type func_name (argument_list)
    };
```

Characteristics of Friend Function

- It is not in the ‘scope’ of the class X to which it has been declared a friend.
- It cannot be invoked using the object of class X, as it is not in the scope of that class X.
- Friend functions have objects (of class X) as arguments.
- It cannot access the member names (of class X) directly and has to use ‘object_name.member_name’
- We can declare it either in the ‘public’ or the ‘private’ part.

Friend Function Definition

- A friend function of class X can be defined:
  - outside any class (globally)
  - inside some other class Y
- When defined inside other class, it is invoked on the object of that class
- When defined globally, it is invoked by its name only

Friend Function Example – global Definition

```
class Rectangle {
   private:   int len;
                 int bre;
   public:
           Rectangle() { len = 0;  bre = 0; }
           //friend function declaration
           friend void displayRectangleData(Rectangle&);
};

//global definition of friend function
void displayRectangleData(Rectangle& r) {
        // displayRectangleData() can access private
        members of Rectangle object
        cout << "rectangle length = " << r.len <<endl;
         cout << "rectangle breadth = " << r.bre;
}
```

```
int main()
{
        Rectangle obj;
        displayRectangleData(obj);
        return 0;
}
```

Friend Function Example – inside class definition

```
class Rectangle;

class rectangleFriend {
  private:
          int data;
  public:
          void displayRectangleData(Rectangle&);
};

class Rectangle {
  private:   int len;
              int bre;
  public:
     Rectangle() { len = 0;  bre = 0; }

     //friend function declaration
     friend void rectangleFriend::displayRectangleData(Rectangle&);
};
```

```
//friend function definition inside class
void rectangleFriend::displayRectangleData(Rectangle& r)
{
     cout << "rectangle length  = " << r.len <<endl;
     cout << "rectangle breadth = " << r.bre;
}

 int main()
 {
           Rectangle obj;
            rectangleFriend  rf;
            rf.displayRectangleData(obj);
            return 0;
 }
```

Pure Virtual Functions

- Defined as ‘empty’ function
- Also known as ‘do-nothing’ functions

Syntax:

```
virtual return_type func_name() = 0;
```

- In such cases derived class must:
  - either define the function
  - or re-declare it as virtual inside it

Pure virtual functions

- A class containing pure virtual functions cannot be used to create its own objects

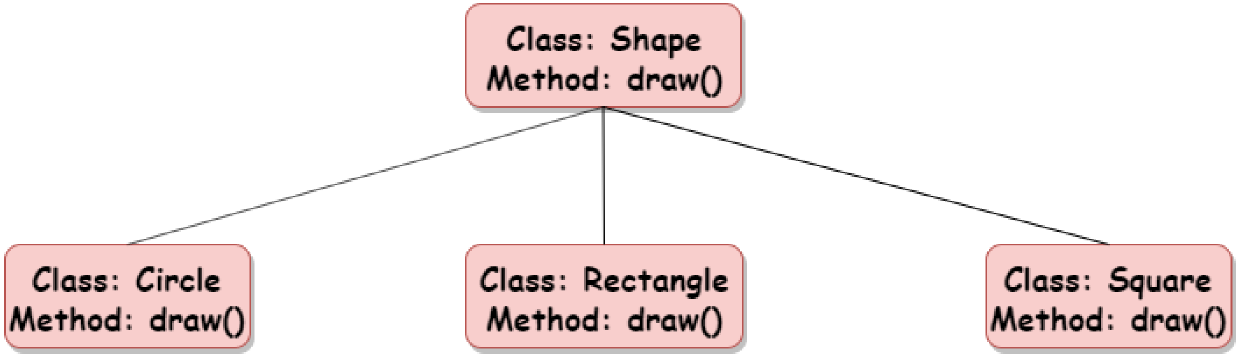
- Such classes are called abstract base classes

- Main objective is to provide some traits to the derived classes and to create a base pointer required for achieving run-time polymorphism

Polymorhism

- Made up of two Greek words:
  - Poly: it means ***‘Many’***
  - Morphism: it means ***‘Forms’***

- It is defined as ‘having the same name but different behavior’



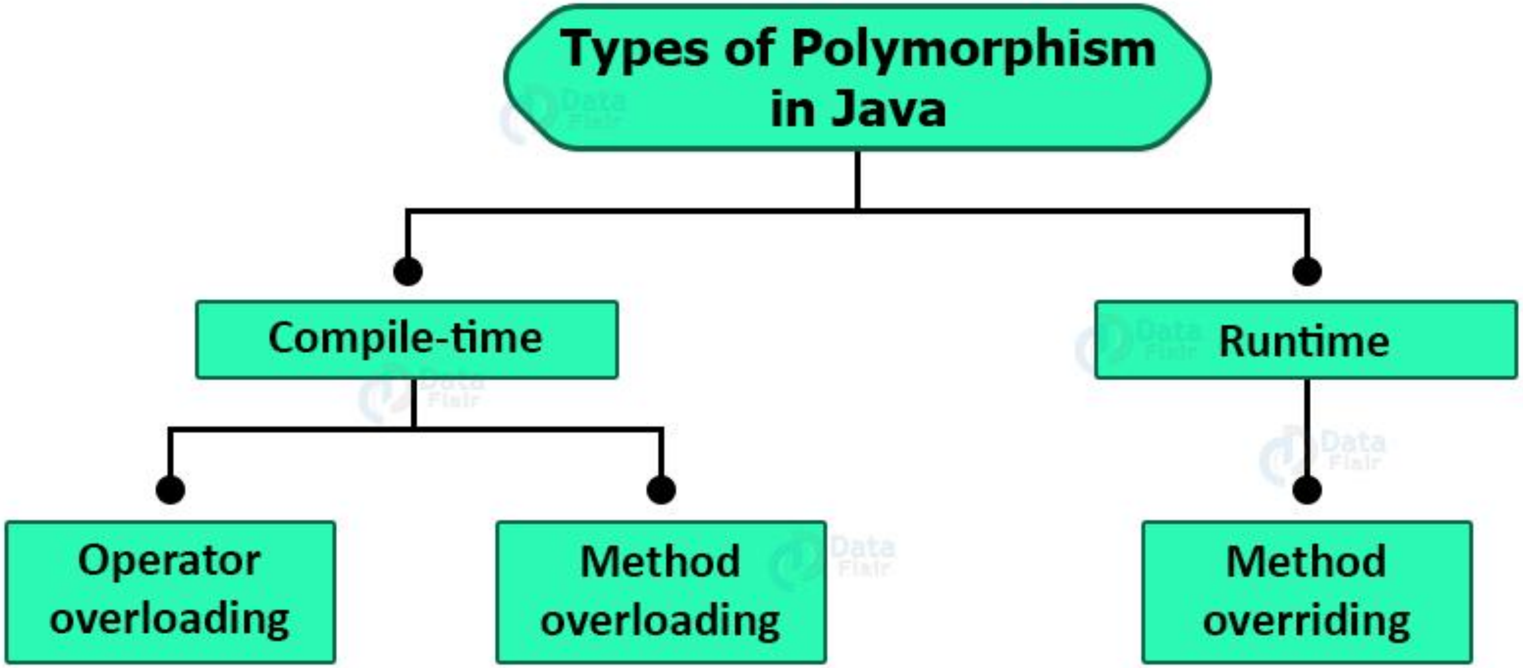
Class: Shape
Method: draw()

Class: Circle
Method: draw()

Class: Rectangle
Method: draw()

Class: Square
Method: draw()

Types of Polymorhism



Types of Polymorhism

- One categorization specifies two categories:
  - **Compile-Time (Static) Polymorphism:**
    - ***Uses concept of Early Binding:*** Version of the overloaded function to be called is decided at compile time, i.e. function-call and respective function-definition (to be called) is binded at compile-time
    - ***Example:*** **function overloading and operator overloading**
  - **Run-Time (Dynamic) Polymorphism:**
    - ***Uses concept of Late Binding:*** Version of the overloaded function to be called is decided at run time, i.e. function-call and respective function-definition (to be called) is binded at run-time
    - ***Example:*** **function overriding (use of virtual functions)**

Types of Polymorhism - Example

- Function overloading: Example already explained
- Function overriding (use of virtual functions): Example already explained
- Operator overloading – to be discussed

Operator Overloading

- It is an example of compile time polymorphism
- Operator overloading is a type of polymorphism in which an operator is overloaded to give a new meaning to the operator as per user requirement
- **Example:** + operator is overloaded to work for String class objects to concatenate two strings
- **‘operator’** keyword is used to overload an operator

Operator Overloading Example-1

```
#include<iostream>
using namespace std;

class Comp {
    private:
            int re, im;
    public:
            Comp (int rval = 0, int ival =0)  {re = rval;   im = ival;}
            // Operator + overloaded for Complex objects
            Comp operator + (Comp &obj) {
                        Comp addobj;
                        addobj.re = re + obj.re;
                        addobj.im = im + obj.im;
                        return addobj;
             }
 void print() { cout << “Comp No =” <<re << " + i" << im << endl; }
};
```

```
int main()
{
   Comp cn1(3, 2), cn2(1, 3);
   Comp cn3 = cn1 + cn2;
   cn3.print();
}
```

```
cn1:      3 + 2i
cn2:      1 + 3i
--------------------
 addobj: 4 + 5i
--------------------
```

Any Queries??