# CHAPTER 6: POLYMORPHISM

## *Introduction to Polymorphism*

Polymorphism is an object-oriented programming concept that refers to the ability of a variable, function or object to take on multiple forms. A language that features polymorphism allows developers to program in the general rather than program in the specific.

In a programming language that exhibits polymorphism, objects of classes belonging to the same hierarchical tree (i.e. inherited from a common base class) may possess functions bearing the same name, but each having different behaviors.

As an example, let us assume there is a base class named Animals from which the subclasses Horse, Fish and Bird are derived. Let us also assume that the Animals class has a function named Move, which is inherited by all subclasses mentioned. With polymorphism, each subclass may have its own way of implementing the function. So, for example, when the Move function is called in an object of the Horse class, the function might respond by displaying trotting on the screen. On the other hand, when the same function is called in an object of the Fish class, swimming might be displayed on the screen. In the case of a Bird object, it may be flying.

In effect, polymorphism trims down the work of the developer because he can now create a sort of general class with all the attributes and behaviors that he envisions for it. When the time comes for the developer to create more specific subclasses with certain unique attributes and behaviors, the developer can simply alter code in the specific portions where the behaviors will differ. All other portions of the code can be left as is.

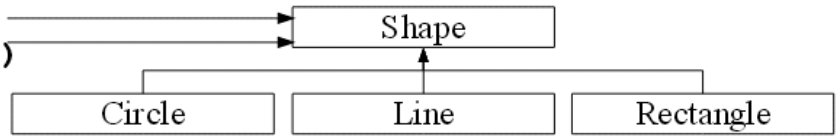


## Advantages/importance of Polymorphism

When an object has a reference to another, it can invoke methods on that object reference without knowing, or caring, what the implementation is.

**Why Polymorphism?**

```
// substitutability
Shape s;
s.draw()
s.resize()
```




**Note**

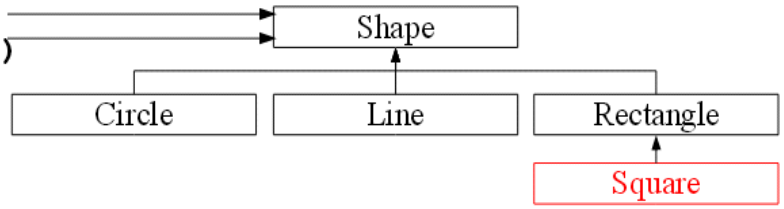
•Substitutability means, the type of the variable does not have to match with the type of the value assigned to that variable.

•Substitutability cannot be achieved in conventional languages in C, but can be achieved in Object Oriented languages like Java.

•We have already seen the concept of “Assigning a subclass object to superclass variable or reference”. This is called substitutability. Here I am substituting the superclass object with the

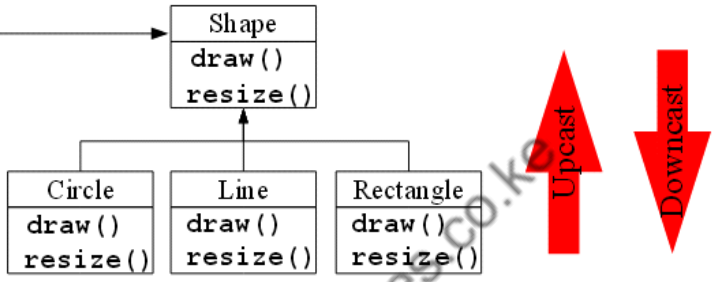
object of subclass.

```
// extensibility
Shape s;
s.draw()
s.resize()
```




Extensibility is the ability of an object-oriented system to add new behaviors to an existing system without changing the application shell.

```
// common interface
Shape s;
s.draw()
s.resize()
```






```
// upcasting
Shape s = new Line();
s.draw()
s.resize()
```

## Discussion

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes:

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

class Shape {
   protected:
```

```
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape{
   public:
      Rectangle( int a=0, int b=0):Shape(a, b) { }
      int area ()
      {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};
class Triangle: public Shape{
   public:
      Triangle( int a=0, int b=0):Shape(a, b) { }
      int area ()
      {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};
// Main function for the program
int main( )
{
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   // store the address of Rectangle
   shape = &rec;
   // call rectangle area.
   shape->area();

   // store the address of Triangle
   shape = &tri;
   // call triangle area.
   shape->area();

   return 0;
}
```



When the above code is compiled and executed, it produces the following result:

```
Parent class area
Parent class area
```

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the

program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.
But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      virtual int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result:

```
Rectangle class area
Triangle class area
```

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.



As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

## Encapsulation / Information hiding

**Information hiding** is one of the most important principles of OOP inspired from real life which says that all information should not be accessible to all persons. Private information should only be accessible to its owner.
By Information Hiding we mean "*Showing only those details to the outside world which are necessary for the outside world and hiding all other details from the outside world.*"

Real Life Examples of Information Hiding

- ✓ Your name and other personal information is stored in your brain we can't access this information directly. For getting this information we need to ask you about it and it will be up to you how much details you would like to share with us.

- ✓ An email server may have account information of millions of people but it will share only our account information with us if we request it to send anyone else accounts information our request will be refused.

- ✓ A phone SIM card may store several phone numbers but we can't read the numbers directly from the SIM card rather phone-set reads this information for us and if the owner of this phone has not allowed others to see the numbers saved in this phone we will not be able to see those phone numbers using phone.

In object oriented programming approach we have objects with their attributes and behaviors that are hidden from other classes, so we can say that object oriented programming follows the principle of information hiding.
In the perspective of Object Oriented Programming Information Hiding is,
*"Hiding the object details (state and behavior) from the users"*
Here by users we mean **"an object"** of another class that is calling functions of this class using the reference of this class object or it may be some other program in which we are using this class.
Information Hiding is achieved in Object Oriented Programming using the following principles,

- All information related to an object is stored within the object
- It is hidden from the outside world
- It can only be manipulated by the object itself



Advantages of Information Hiding
Following are two major advantages of information hiding. It simplifies our Object Oriented Model:
As we saw earlier that our object oriented model only had objects and their interactions hiding implementation details so it makes it easier for everyone to understand our object oriented model. It is a barrier against change propagation. As implementation of functions is limited to our class and we have only given the name of functions to user along with description of parameters so if we change implementation of function it doesn't affect the object oriented model.
We can achieve information hiding using Encapsulation and Abstraction, so we see these two concepts in detail now.

**Encapsulation** means *"we have enclosed all the characteristics of an object in the object itself".*
Encapsulation and information hiding are much related concepts (information hiding is achieved using Encapsulation). We have seen in previous lecture that object characteristics include data members and behavior of the object in the form of functions. So we can say that Data and Behavior are tightly coupled inside an object and both the information structure and implementation details of its operations are hidden from the outer world.

Examples of Encapsulation
Consider the same example of object Ali of previous lecture we described it as follows.

| Ali |
| --- |
| Characteristics (attributes)<br>· Name<br>· Age |
| Behavior (operations)<br>· Walks<br>· Eats |

You can see that Ali stores his personal information in itself and its behavior is also implemented in it. Now it is up to object Ali whether he wants to share that information with outside world or not. Same thing stands for its behavior if some other object in real life wants to use his behavior of walking it can not use it without the permission of Ali. So we say that attributes and behavior of Ali are encapsulated in it. Any other object don't know about these things unless Ali share this information with that object through an interface. Same concept also applies to phone which has some data and behavior of showing that data to user we can only access the information stored in the phone if phone interface allow us to do so.

**Advantages of Encapsulation**
The following are the main advantages of Encapsulation,

1. **Simplicity and clarity**

As all data and functions are stored in the objects so there is no data or function around in program that is not part of any object and is this way it becomes very easy to understand the purpose of each data member and function in an object.

2. **Low complexity**

As data members and functions are hidden in objects and each object has a specific behavior so there is less complexity in code there will be no such situations that a functions is using some other function and that functions is using some other function.

3. **Better understanding**

Everyone will be able to understand whole scenario by simple looking into object diagrams without any issue as each object has specific role and specific relation with other objects.



## Encapsulation / Information hiding subject properties

### Virtual Function:

A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function. What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

### Pure Virtual Functions:

It's possible that you'd want to include a virtual function in a base class so that it may be

redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      // pure virtual function
      virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

## Data Abstraction

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.



Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.

Now, if we talk in terms of C++ Programming, C++ classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.
For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.
In C++, we use **classes** to define our own abstract data types (ADT). You can use

the **cout** object of class **ostream** to stream data to standard output like this:

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

int main( )
.{
   cout << "Hello C++" <<endl;
   return 0;
}
```

Here, you don't need to understand how **cout** displays the text on the user's screen. You need to only know the public interface and the underlying implementation of cout is free to change.

**Access Labels Enforce Abstraction:**
In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels:

- Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
- Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.



**Benefits of Data Abstraction:**
Data abstraction provides two important advantages:

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data are public, then any function that directly accesses the data members of the old representation might be broken.

**Data Abstraction Example:**
Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example:

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

class Adder{
   public:
      // constructor
      Adder(int i = 0)
      {
```

```
        total = i;
      }
      // interface to outside world
      void addNum(int number)
      {
          total += number;
      }
      // interface to outside world
      int getTotal()
      {
          return total;
      };
   private:
      // hidden data from outside world
      int total;
};
int main( )
{
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal**are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that the user doesn't need to know about, but is needed for the class to operate properly.



**Designing Strategy:**
Abstraction separates code into interface and implementation. So while designing your component, you must keep interface independent of the implementation so that if you change underlying implementation then interface would remain intact.

In this case whatever programs are using these interfaces, they would not be impacted and would just need a recompilation with the latest implementation.

## Data Encapsulation

All C++ programs are composed of the following two fundamental elements:

- **Program statements (code):** This is the part of a program that performs actions and they are called functions.
- **Program data:** The data is the information of the program which affected by the program functions.

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data**

**hiding**.

**Data encapsulation** is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected** and **public**members. By default, all items defined in a class are private. For example:

```
class Box
{
   public:
      double getVolume(void)
      {
         return length * breadth * height;
      }
   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

The variables length, breadth, and height are **private**. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

To make parts of a class **public** (i.e., accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.

Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.



**Data Encapsulation Example:**

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example:

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

class Adder{
   public:
      // constructor
      Adder(int i = 0)
      {
        total = i;
      }
      // interface to outside world
      void addNum(int number)
      {
          total += number;
      }
      // interface to outside world
      int getTotal()
      {
          return total;
      };
```

```
   private:
      // hidden data from outside world
      int total;
};
int main( )
{
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal**are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that is hidden from the outside world, but is needed for the class to operate properly.

**Designing Strategy:**
Most of us have learned through bitter experience to make class members private by default unless we really need to expose them. That's just good **encapsulation**.

This wisdom is applied most frequently to data members, but it applies equally to all members, including virtual functions.



## Interfaces (Abstract Classes)

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.
The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.
A class is made abstract by declaring at least one of its functions as **pure virtual** function. A pure virtual function is specified by placing "= 0" in its declaration as follows:

```
class Box
{
   public:
      // pure virtual function
      virtual double getVolume() = 0;
   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

The purpose of an **abstract class** (often referred to as an ABC) is to provide an