Learning C-Sharp

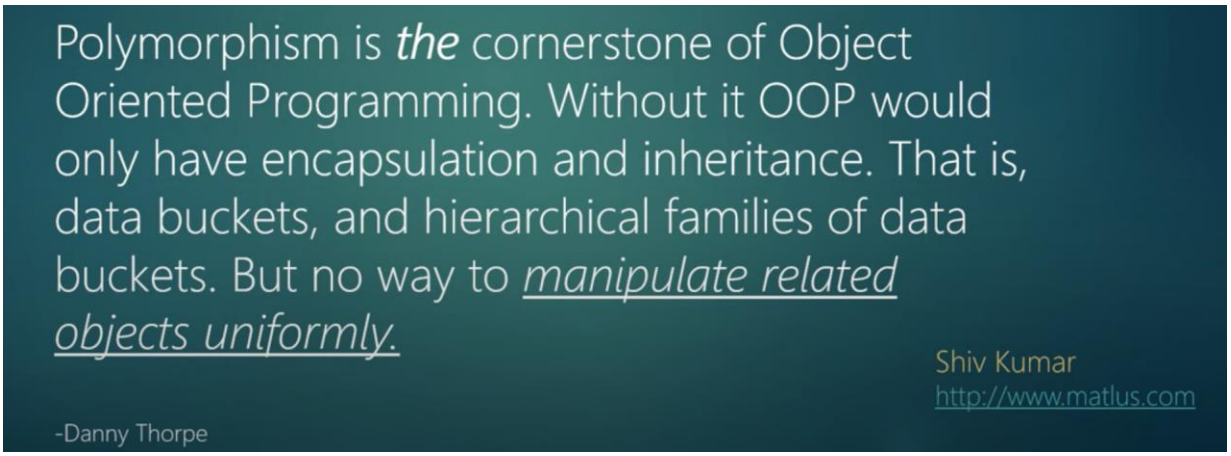
Polymorphism

WHAT IS POLYMORPHISM AND WHY IT IS IMPORTANT?

- Polymorphism is one of the most important concepts in Object Oriented Programming (OOP).
- We can implement polymorphism in two ways:
    - Static or Compile time Polymorphism implemented by function and operator overloading
    - Runtime or Dynamic Polymorphism by virtual methods

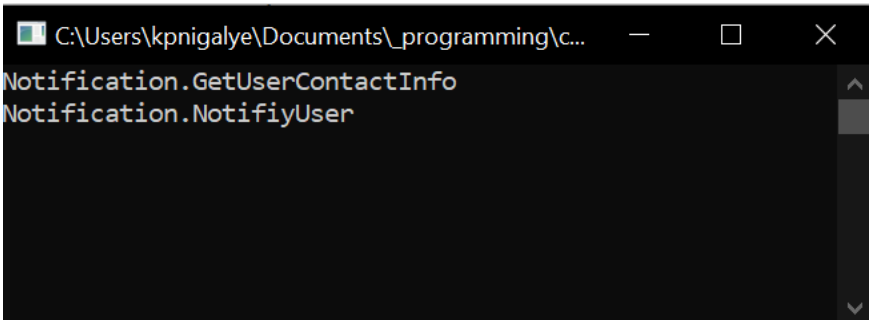
In this tutorial we are going to see Dynamic or Runtime Polymorphism.

- Let me start by saying this, it is hard to explain Polymorphism without Inheritance.



Let us consider this code. We are having a '**Notification**' class which is sending user notifications. So, the idea here is to get user information before sending notification. Therefore, GetUserInfor() is defined as protected and it is called from NotifiyUser() method as shown below.

```
public class Notification
{
    protected void GetUserContactInfo()
    {
        Console.WriteLine("Notification.GetUserContactInfo");
    }

    public void NotifiyUser()
    {
        GetUserContactInfo();
        Console.WriteLine("Notification.NotifiyUser");
    }
}

 class Program
 {
     static void Main(string[] args)
     {
         Notification notification = new Notification();
         notification.NotifiyUser();

         Console.ReadLine();
     }
 }
```



In real life situations, notifications can be in the form of SMS or Email.

Let us add a class ‘SMSNotification’ which will derive from the base class ‘Notification’.

C# lets you define a variable of base class type and assign it to the instance of derived class type.

```
Notification notification = new SMSNotification();
```

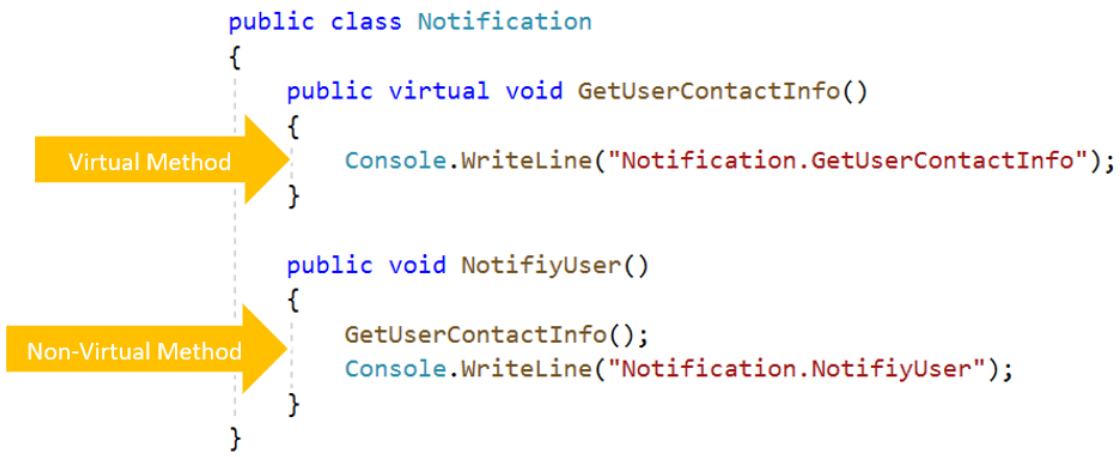
Lets look at the code below,

```
namespace c_sharp_polymorphism
{
    class Program
    {
        static void Main(string[] args)
        {
            Notification notification = new SMSNotification();
            notification.NotifiyUser();

            Console.ReadLine();
        }
    }


    public class Notification
    {
        public void GetUserContactInfo()
        {
            Console.WriteLine("Notification.GetUserContactInfo");
        }

        public void NotifiyUser()
        {
            GetUserContactInfo();
            Console.WriteLine("Notification.NotifiyUser");
        }
    }

    public class SMSNotification : Notification
    {

    }
}
```

```
C:\Users\kpnigalye\Documents\_...
Notification.GetUserContactInfo
Notification.NotifiyUser
```

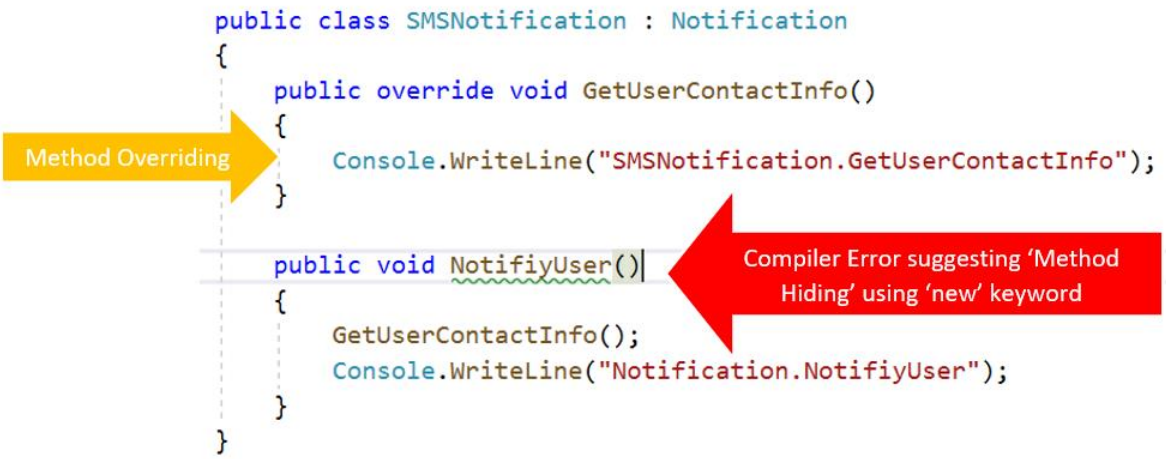
As you can see in the code, when you call the ‘NotifyUser()’ function using an instance of derived class ‘SMSNotification’, it will call the functions defined in the base class.

Now, let’s make few more changes to the code.



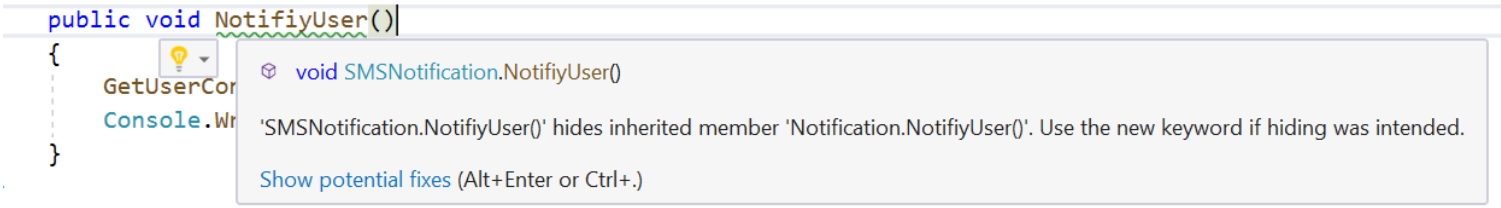

```
public class Notification
{
    public virtual void GetUserContactInfo()
    {
        Console.WriteLine("Notification.GetUserContactInfo");
    }

    public void NotifiyUser()
    {
        GetUserContactInfo();
        Console.WriteLine("Notification.NotifiyUser");
    }
}
```

When we define both the methods in the derived class ‘SMSNotification’, you can see following things.




```
public class SMSNotification : Notification
{
    public override void GetUserContactInfo()
    {
        Console.WriteLine("SMSNotification.GetUserContactInfo");
    }

    public void NotifiyUser()
    {
        GetUserContactInfo();
        Console.WriteLine("Notification.NotifiyUser");
    }
}
```

Now, when you define non-virtual methods in the derived class, compiler will give you an error as shown below.



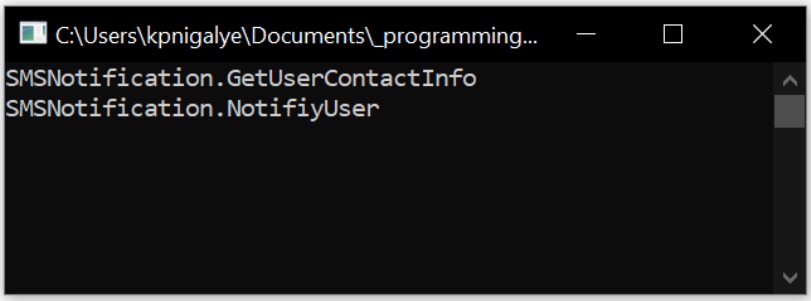
If you want, you can hide the method defined in the base class using ‘new’ keyword.

```
public new void NotifiyUser()
{
    GetUserContactInfo();
    Console.WriteLine("Notification.NotifiyUser");
}
```

Now remember, ‘new’ keyword has nothing to do with polymorphism. It just helps you in method hiding in the derived class.

The following code will work perfectly. It will call the functions of derived classes as both the variable and the object are of derived class type.

```
class Program
{
    static void Main(string[] args)
    {
        SMSNotification notification = new SMSNotification();
        notification.NotifiyUser();

        Console.ReadLine();
    }
}


public class Notification
{
    public virtual void GetUserContactInfo()
    {
        Console.WriteLine("Notification.GetUserContactInfo");
    }

    public void NotifiyUser()
    {
        GetUserContactInfo();
        Console.WriteLine("Notification.NotifiyUser");
    }
}

public class SMSNotification : Notification
{
    public override void GetUserContactInfo()
    {
        Console.WriteLine("SMSNotification.GetUserContactInfo");
    }

    public new void NotifiyUser()
    {
        GetUserContactInfo();
        Console.WriteLine("SMSNotification.NotifiyUser");
    }
}
```



But when you have a variable of base class and object of derived class like this

```
Notification notification = new SMSNotification();
```

then, the code will behave differently.

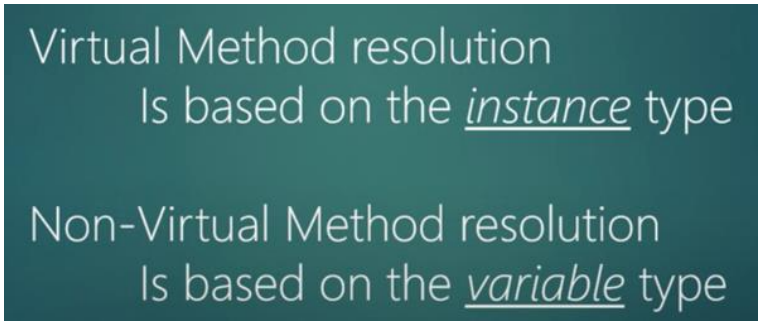
```
class Program
{
    static void Main(string[] args)
    {
        Notification notification = new SMSNotification();
        notification.NotifiyUser();

        Console.ReadLine();
    }
}


public class Notification
{
    public virtual void GetUserContactInfo()
    {
        Console.WriteLine("Notification.GetUserContactInfo");
    }

    public void NotifiyUser()
    {
        GetUserContactInfo();
        Console.WriteLine("Notification.NotifiyUser");
    }
}

public class SMSNotification : Notification
{
    public override void GetUserContactInfo()
    {
        Console.WriteLine("SMSNotification.GetUserContactInfo");
    }

    public new void NotifiyUser()
    {
        GetUserContactInfo();
        Console.WriteLine("SMSNotification.NotifiyUser");
    }
}
```

```
C:\Users\kpnigalye\Documents\_progr...
SMSNotification.GetUserContactInfo
Notification.NotifiyUser
```

Take a look at the output on console.

First, Control is calling the 'NotifyUser()' function of 'SMSNotification' but inside that function, it is not calling the 'GetUserContactInfo()' function of derived class but instead it calls a function of base class.

Remember two important points here.



Therefore,

1. Virtual method of derived class ‘SMSNotification’ is called and it is decided at runtime
2. Non-virtual method of ‘Notification’ class is called which is based on the variable type and it is decided at compile time.

Now, let’s go ahead and make some more changes to the code.

```
public abstract class Notification
{
    protected void GetUserContactInfo()
    {
        Console.WriteLine("Notification.GetUserContactInfo");
    }

    public abstract void NotifiyUser();
}
```

I have two classes ‘SMSNotification’ and ‘EmailNotification’ as shown below.

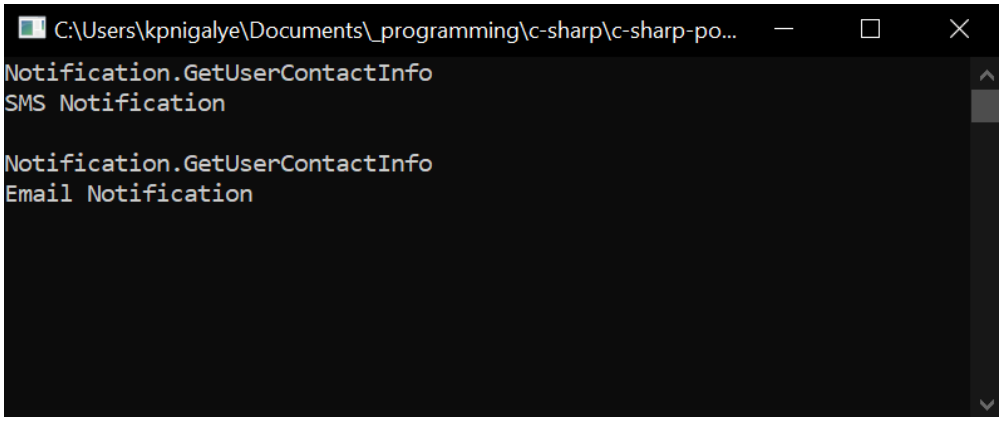
```
public class SMSNotification : Notification
{
    public override void NotifiyUser()
    {
        GetUserContactInfo();
        Console.WriteLine("SMS Notification\n");
    }
}

public class EmailNotification : Notification
{
    public override void NotifiyUser()
    {
        GetUserContactInfo();
        Console.WriteLine("Email Notification\n");
    }
}
```

As you can see, these two classes has own implementation of ‘NotifiyUser()’ method.

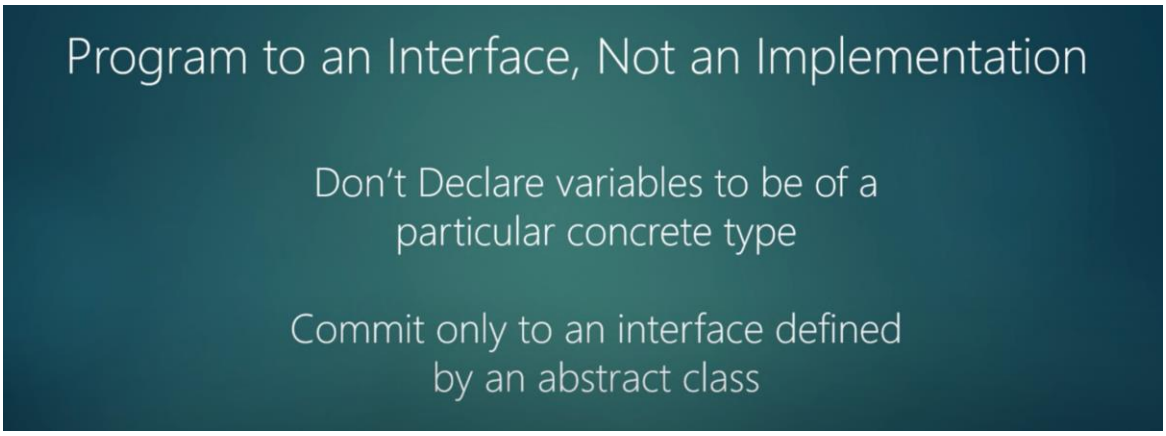
```
class Program
{
    static void Main(string[] args)
    {
        SMSNotification smsNotification = new SMSNotification();
        smsNotification.NotifiyUser();

        EmailNotification emailNotification = new EmailNotification();
        emailNotification.NotifiyUser();

        Console.ReadLine();
    }
```

You can create objects of these derived classes and call their 'NotifyUser()' function.

Here let us take a look at the type of variables and their instances. Both are of the same type so we get the expected result as below.



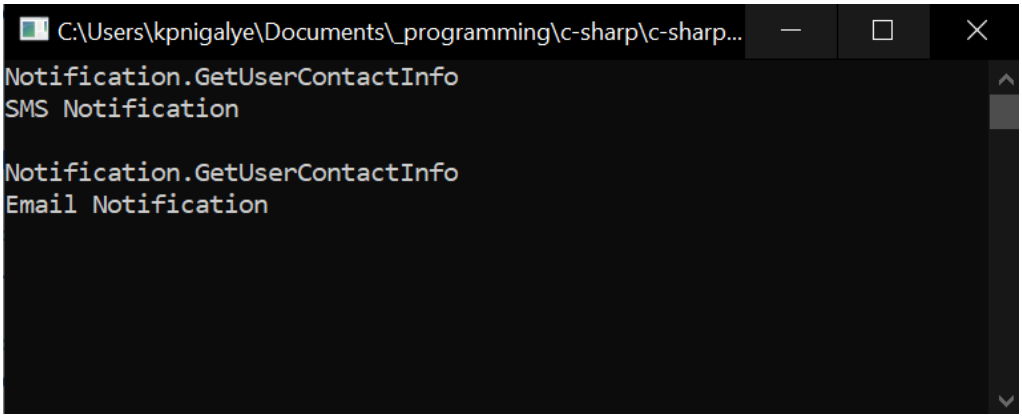
But there is a famous programming principle while using Inheritance and Polymorphism.



Here interface is not necessarily an interface in C# but something which is of abstract type. So, let us make changes to our object declarations.

```
class Program
{
    static void Main(string[] args)
    {
        Notification smsNotification = new SMSNotification();
        smsNotification.NotifiyUser();

        Notification emailNotification = new EmailNotification();
        emailNotification.NotifiyUser();

        Console.ReadLine();
    }
}
```

Let's run the code and see it the output.



Well, as you can see the output is still the same. So, Polymorphism allows you to declare a variable of base class and assign an object of derived class to it.

Let us observe the code in detail.

```
class Program
{
    static void Main(string[] args)
    {
        Notification smsNotification = new SMSNotification();
        smsNotification.NotifiyUser();

        Notification emailNotification = new EmailNotification();
        emailNotification.NotifiyUser();

        Console.ReadLine();
    }
}
```

Now the control goes inside the function call which is the method defined in the 'SMSNotification' class.

```
30      public class SMSNotification : Notification
31      {
32          public override void NotifiyUser()
33          {
34              GetUserContactInfo();  ≤ 1ms elapsed
35              Console.WriteLine("SMS Notification\n");
36          }
37      }
```

As you can see the control goes inside the GetUserContactInfo() of the abstract base class.

```
20      public abstract class Notification
21      {
22          protected void GetUserContactInfo()
23          {
24              Console.WriteLine("Notification.GetUserContactInfo");
25          }
26
27          public abstract void NotifiyUser();
28      }
```

So, the non-virtual method of the abstract base class is getting called.

Let us consider one more case. What will happen if I declare the 'NotifyUser()' method in the abstract base class as *virtual*? Rest of the code remains as it is. Only change we will make is making 'NotifyUser()' method virtual.

```
public abstract class Notification
{
    protected void GetUserContactInfo()
    {
        Console.WriteLine("Notification.GetUserContactInfo");
    }

    public virtual void NotifiyUser()
    {
        Console.WriteLine("Notification.NotifiyUser");

    }
}
```

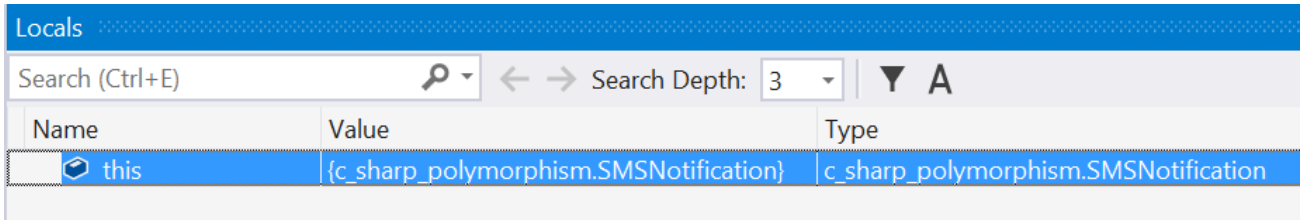
Let us run the code again.

```
30      public class SMSNotification : Notification
31      {
32          public override void NotifiyUser()
33          {
34              GetUserContactInfo();  ≤ 1ms elapsed
35              Console.WriteLine("SMS Notification\n");
36          }
37      }
```

Control still goes inside the 'NotifyUser()' method of derived class and not the one defined in base class. Why is that?

First of all, the method is marked as '*virtual*' and it is overridden in the derived class using '*override*' keyword.

But there is one more important thing. During the function call, if we observe the 'locals' of this function call, you will see this.



From the above image we can see that, the object is of type 'SMSNotification' so it is calling the method of derived class.

With this example we can also observe that

Polymorphism allows base classes to reach its decedents.

Base class doesn't know about its descendants. Still it is able to reach its descendent through polymorphism (obviously by calling its methods) using its instance type.

So, Polymorphism can be defined as follow,

Polymorphism is the use of *virtual methods* to make one method (name) produce one of the many possible outcomes (behaviours) depending upon the instance.

Further, this is the way we can write a method to create an object based on user input.

```
private static Notification CreateNotification(int choice)
{
    switch (choice)
    {
        case (int)NotificationTypeEnum.SMS:
            return new SMSNotification();
        case (int)NotificationTypeEnum.Email:
            return new EmailNotification();
        default:
            throw new Exception();
    }
```

So, as you can see, there is a change in behaviour in 'Notification' type depending on the user input. We are able to achieve a behavioural change by calling 'NotifyUser()' method based on the instance of derived class.

Also, the client code (in our case it is '*main*' method) don't have any idea about what type of object which is instantiated or how many derived classes are there.