Тема: «ООП. Наследование. Абстрактные классы и члены классов. Виртуальные члены классов. Запечатанные классы и члены классов.»

Цель: получить навыки создания абстрактных классов и членов класса, создания виртуальных членов классов. Изучить наследование и основные принципы ООП.

Время выполнения: 8 часов.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

ООП. Наследование

Наследование, вместе с инкапсуляцией и полиморфизмом, является одной из трех основных характеристик объектно-ориентированного программирования. Наследование позволяет создавать новые классы, которые повторно используют, расширяют и изменяют поведение, определенное в других классах. Класс, члены которого наследуются, называется базовым классом, а класс, который наследует эти члены, называется производным классом. Производный класс может иметь только один прямой базовый класс. Однако наследование является транзитивным. Если `ClassC` является производным от `ClassB`, а `ClassB` –от `ClassA`, `ClassC` наследует члены, объявленные в `ClassB` и `ClassA`.

Примечание

Структуры не поддерживают наследование, но могут реализовывать интерфейсы.

Концептуально производный класс является специализацией базового класса. Например, при наличии базового класса `Animal` возможно наличие одного производного класса, который называется `Mammal`, и еще одного производного класса, который называется `Reptile`. `Mammal` является `Animal` и `Reptile` является `Animal`, но каждый производный класс представляет разные специализации базового класса.

При определении класса для наследования от другого класса производный класс явно получает все члены базового класса за исключением конструкторов и методов завершения. Производный класс повторно использует код в базовом классе без необходимости его повторной реализации. В производный класс можно добавить дополнительные члены. Производный класс расширяет функциональность базового класса.

На рисунке 1 показан класс `WorkItem`, представляющий рабочий элемент в бизнес-процессе. Как и другие классы, он является производным от

System.Object и наследует все его методы. *WorkItem* добавляет шесть собственных членов. К ним относится конструктор, так как конструкторы не наследуются. Класс *ChangeRequest* наследует от *WorkItem* и представляет конкретный вид рабочего элемента. *ChangeRequest* добавляет еще два члена к членам, унаследованным от *WorkItem* и *Object*. Он должен добавить собственный конструктор, и он также добавляет *originalItemID*. Свойство *originalItemID* позволяет *ChangeRequest* связать экземпляр с исходным объектом *WorkItem*, к которому применен запрос на изменение.

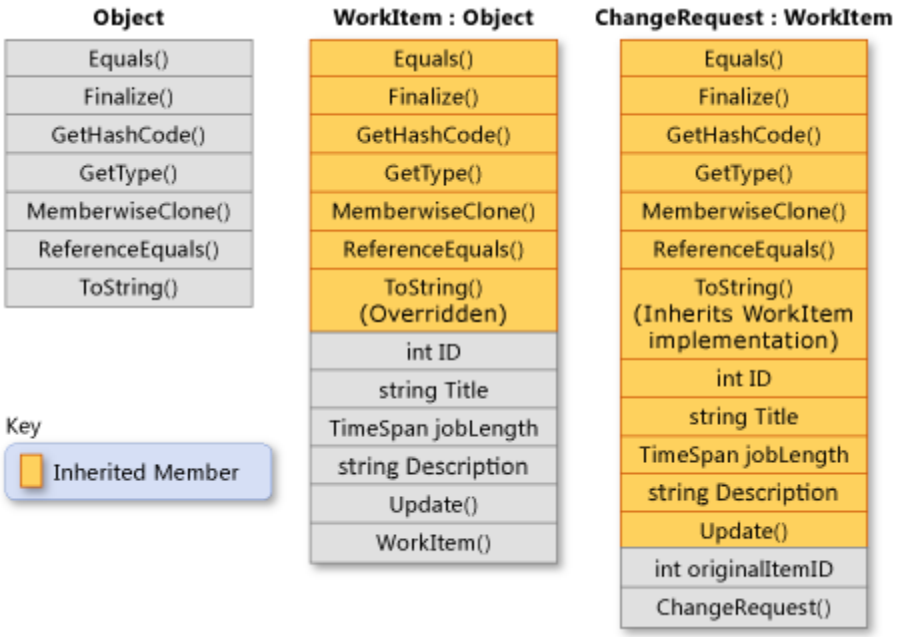

Рисунок 1 – Схема наследования классов

В следующем примере показано, как выражаются в C# отношения между классами, продемонстрированные на предыдущем рисунке. В примере также показано, как *WorkItem* переопределяет виртуальный метод *Object.ToString* и как класс *ChangeRequest* наследует *WorkItem* реализацию метода. В первом блоке определяются классы:

```
// WorkItem implicitly inherits from the Object class.
public class WorkItem
{
      // Static field currentID stores the job ID of the last
WorkItem that
    // has been created.
    private static int currentID;

    //Properties.
```

```
    protected int ID { get; set; }
    protected string Title { get; set; }
    protected string Description { get; set; }
    protected TimeSpan jobLength { get; set; }

    // Default constructor. If a derived class does not invoke a
base-
     // class constructor explicitly, the default constructor is
called
    // implicitly.
    public WorkItem()
    {
        ID = 0;
        Title = "Default title";
        Description = "Default description.";
        jobLength = new TimeSpan();
    }

    // Instance constructor that has three parameters.
    public WorkItem(string title, string desc, TimeSpan joblen)
    {
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = joblen;
    }

      // Static constructor to initialize the static member,
currentID. This
    // constructor is called one time, automatically, before any
instance
     // of WorkItem or ChangeRequest is created, or currentID is
referenced.
    static WorkItem() => currentID = 0;

     // currentID is a static field. It is incremented each time
a new
    // instance of WorkItem is created.
    protected int GetNextID() => ++currentID;

      // Method Update enables you to update the title and job
length of an
    // existing WorkItem object.
    public void Update(string title, TimeSpan joblen)
    {
        this.Title = title;
```

```
        this.jobLength = joblen;
    }

     // Virtual method override of the ToString method that is
inherited
    // from System.Object.
    public override string ToString() =>
        $"{this.ID} - {this.Title}";
}

// ChangeRequest derives from WorkItem and adds a property
(originalItemID)
// and two constructors.
public class ChangeRequest : WorkItem
{
    protected int originalItemID { get; set; }

     // Constructors. Because neither constructor calls a base-
class
     // constructor explicitly, the default constructor in the
base class
      // is called implicitly. The base class must contain a
default
    // constructor.

    // Default constructor for the derived class.
    public ChangeRequest() { }

    // Instance constructor that has four parameters.
      public ChangeRequest(string title, string desc, TimeSpan
jobLen,
                           int originalID)
    {
         // The following properties and the GetNexID method are
inherited
        // from WorkItem.
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = jobLen;

         // Property originalItemID is a member of ChangeRequest,
but not
        // of WorkItem.
        this.originalItemID = originalID;
    }
```

```
}
```

В следующем блоке показано, как использовать базовый и производный классы:

```
// Create an instance of WorkItem by using the constructor in
the
// base class that takes three arguments.
WorkItem item = new WorkItem("Fix Bugs",
                             "Fix all bugs in my code branch",
                             new TimeSpan(3, 4, 0, 0));

// Create an instance of ChangeRequest by using the constructor
in
// the derived class that takes four arguments.
ChangeRequest change =
     new ChangeRequest(
     "Change Base Class Design","Add members to the class",
     new TimeSpan(4, 0, 0),1
     );

// Use the ToString method defined in WorkItem.
Console.WriteLine(item.ToString());

// Use the inherited Update method to change the title of the
// ChangeRequest object.
change.Update("Change the Design of the Base Class",
    new TimeSpan(4, 0, 0));

// ChangeRequest inherits WorkItem's override of ToString.
Console.WriteLine(change.ToString());
/* Output:
    1 - Fix Bugs
    2  - Change the Design of the Base Class
*/
```

Ключевое слово `virtual`

Ключевое слово *`virtual`* используется для изменения объявлений методов, свойств, индексаторов и событий и разрешения их переопределения в производном классе. Например, этот метод может быть переопределен любым наследующим его классом:

```
public virtual double Area()
{
    return x * y;
}
```

Реализацию виртуального члена можно изменить путем переопределения члена в производном классе.

При вызове виртуального метода тип времени выполнения объекта проверяется на переопределение члена. Вызывается переопределение члена в самом дальнем классе. Это может быть исходный член, если никакой производный класс не выполнял переопределение этого члена.

По умолчанию методы не являются виртуальными. Такой метод переопределить невозможно.

Нельзя использовать модификатор *virtual* с модификаторами *static*, abstract, *private*, или *override*. В следующем примере показано виртуальное свойство.

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int _num;
    public virtual int Number
    {
        get { return _num; }
        set { _num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string _name;

    // Override auto-implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return _name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                _name = value;
```

```
            }
            else
            {
                _name = "Unknown";
            }
        }
    }
}
```

Действие виртуальных свойств аналогично виртуальным методам, за исключением отличий в синтаксисе объявлений и вызовов.

Использование модификатора `virtual` в статическом свойстве является недопустимым.

Виртуальное наследуемое свойство может быть переопределено в производном классе путем включения объявления свойства, которое использует модификатор `override`.

Пример

В этом примере класс `Shape` содержит две координаты x, y и виртуальный метод `Area()`. Различные классы фигур, такие как `Circle`, `Cylinder` и `Sphere`, наследуют класс `Shape`, и для каждой фигуры вычисляется площадь поверхности. Каждый производный класс обладает собственной реализацией переопределения метода `Area()`.

Обратите внимание, что наследуемые классы *`Circle`*, *`Sphere`* и *`Cylinder`* используют конструкторы, которые инициализируют базовый класс, как показано в следующем объявлении.

```
public Cylinder(double r, double h): base(r, h) {}
```

Следующая программа вычисляет и отображает соответствующую область для каждой фигуры путем вызова нужной реализации метода `Area()` в соответствии с объектом, связанным с методом.

```
class TestClass
{
    public class Shape
    {
        public const double PI = Math.PI;
        protected double _x, _y;

        public Shape()
        {
```

```
    }

    public Shape(double x, double y)
    {
        _x = x;
        _y = y;
    }

    public virtual double Area()
    {
        return _x * _y;
    }
}

public class Circle : Shape
{
    public Circle(double r) : base(r, 0)
    {
    }

    public override double Area()
    {
        return PI * _x * _x;
    }
}

public class Sphere : Shape
{
    public Sphere(double r) : base(r, 0)
    {
    }

    public override double Area()
    {
        return 4 * PI * _x * _x;
    }
}

public class Cylinder : Shape
{
    public Cylinder(double r, double h) : base(r, h)
    {
    }

    public override double Area()
    {
```

```
            return 2 * PI * _x * _x + 2 * PI * _x * _y;
        }
    }

    static void Main()
    {
        double r = 3.0, h = 5.0;
        Shape c = new Circle(r);
        Shape s = new Sphere(r);
        Shape l = new Cylinder(r, h);
        // Display results.
              Console.WriteLine("Area  of  Circle     =  {0:F2}",
c.Area());
              Console.WriteLine("Area  of  Sphere     =  {0:F2}",
s.Area());
               Console.WriteLine("Area  of  Cylinder  =  {0:F2}",
l.Area());
    }
}
/*
Output:
Area of Circle   = 28.27
Area of Sphere   = 113.10
Area of Cylinder = 150.80
*/
```

Абстрактные классы и члены классов

Классы могут быть объявлены абстрактными путем помещения ключевого слова `abstract` перед определением класса. Пример:

```
public abstract class A
{
    // Class members here.
}
```

Создавать экземпляры абстрактного класса нельзя. Назначение абстрактного класса заключается в предоставлении общего определения для базового класса, которое могут совместно использовать несколько производных классов. Например, в библиотеке классов может быть определен абстрактный класс, используемый в качестве параметра для многих из ее функций, поэтому программисты, использующие эту библиотеку, должны задать свою реализацию этого класса, создав производный класс.

Абстрактные классы могут определять абстрактные методы. Для этого перед типом возвращаемого значения метода необходимо поместить ключевое слово *abstract*. Пример:

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Абстрактные методы не имеют реализации, поэтому определение такого метода заканчивается точкой с запятой вместо обычного блока метода. Классы, производные от абстрактного класса, должны реализовывать все абстрактные методы. Если абстрактный класс наследует виртуальный метод из базового класса, абстрактный класс может переопределить виртуальный метод с помощью абстрактного метода. Пример:

```
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }
}
```

Если метод *virtual* объявляется как *abstract*, он все равно считается виртуальным по отношению к любому классу, наследующему от абстрактного класса. Класс, наследующий абстрактный метод, не может получать доступ к исходной реализации метода (так, в предыдущем примере *DoWork* для класса *F* не может вызывать *DoWork* для класса *D*). Таким образом,

абстрактный класс может принудительно задавать необходимость предоставлять новые реализации виртуальных методов в производных классах.

Запечатанные классы и члены классов

Классы могут быть объявлены как запечатанные путем помещения ключевого слова `sealed` перед определением класса. Пример:

```
public sealed class D
{
    // Class members here.
}
```

Запечатанный класс не может использоваться в качестве базового класса. Поэтому он также не может быть абстрактным классом. Запечатанные классы предотвращают наследование. Поскольку их нельзя использовать в качестве базовых классов, определенная оптимизация во время выполнения позволяет несколько ускорить вызов членов запечатанных классов.

Метод, индексатор, свойство или событие для производного класса, переопределяющего виртуальный член базового класса, может объявлять этот член как запечатанный. Это делает бесполезным виртуальный аспект члена для каждого последующего производного класса. Для этого в объявлении члена класса необходимо перед ключевым словом `override` поместить ключевое слово `sealed`. Пример:

```
public class D : C
{
    public sealed override void DoWork() { }
}
```

Индивидуальные задания для лабораторной работы

1. По полученному базовому классу (из предыдущей лабораторной работы) создать классы наследников по двум разным ветвям наследования (B←P1←P11 и B←P2←P21):

- во всех классах должны быть свои данные (характеристики объектов);

- во всех классах создать конструкторы инициализации объектов для всех классов (не забыть про передачу параметров в конструкции базовых классов);
- остальные методы создавать по необходимости.

2. Создать в базовом классе виртуальные функции расчета (например, расчет площади фигуры и т.п.) и вывода объекта на экран (всех его параметров). Выполнить реализацию этих виртуальных функций в классах наследниках.

3. В классе контейнере создать массив, состоящий из объектов базового класса. Заполнить массив динамически создаваемыми объектами производных классов (P1, P11, P2, P21). Для каждого элемента массива проверить работу виртуальных функций.

4. Отладить и выполнить полученную программу. Проверить, что будет, если вышеописанные методы не будут виртуальными.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Изучить теоретическую часть лабораторной работы.

2. Реализовать индивидуальное задание согласно варианту, сделать скриншоты работающих программ. Написать комментарии.

3. Написать отчет, содержащий:

1. Титульный лист, на котором указывается:

а) полное наименование министерства образование и название учебного заведения;

б) название дисциплины;

в) номер практического занятия;

г) фамилия преподавателя, ведущего занятие;

д) фамилия, имя и номер группы студента;

е) год выполнения лабораторной работы.

2. Индивидуальное задание (листинг кода программы, скриншоты консоли с результатами работы).

3. Вывод о проделанной работе.