ЛЕКЦИЯ 4

- Модуль 2
- 08.11.2023
- Полиморфизм

ЦЕЛИ ЛЕКЦИИ

- Обсудить понятие полиморфизма
- Разобраться с вариантами связывания
- Познакомиться с принципом подстановки Лисков

ОБРАЩЕНИЯ К ЧЛЕНАМ БАЗОВОГО КЛАССА

```
class A
{
    protected string Info() => "A";
}
class B : A
{
    new protected string Info() => base.Info() + "B";
}
class C : B
{
    new public string Info() => base.Info() + "C";
}
```

```
C obj = new C();
Console.Write(obj.Info());
```

ВИДЫ ПОЛИМОРФИЗМА

статический

Связан с этапом компиляции

на этапе компиляции программы будет выбрана наиболее подходящая версия, и она будет использоваться в программе

динамический

Связан со временем исполнения

Используются возможности виртуальных членов классов

ПРИМЕР СТАТИЧЕСКОГО ПОЛИМОРФИЗМА

```
public static class Arifmetics
{
    public static int Add(int a, int b) => a + b;
   public static int Add(int a, int b, int c) => a + b + c;
}
```

```
Console.WriteLine(Arifmetics.Add(10, 20)); // Будет выведено: 30.
Console.WriteLine(Arifmetics.Add(10, 20, 30)); // Будет выведено: 60
```

Обязательно ли статический полиморфизм связан со статическими классами и членами класса?

СВЯЗЫВАНИЕ

Связывание – процесс, запускаемый при назначении объекта переменной

Раннее связывание

Связано с этапом компиляции

Позднее связывание

Связано со временем исполнения

РАННЕЕ СВЯЗЫВАНИЕ

```
class A
{
    public void M() => Console.Write("A");
}
class B : A
{
    public new void M() => Console.Write("B");
}
```

Модификатор **new** не изменяет раннее связывание на позднее

По ссылке базового типа доступна версия метода из базового типа

```
A a = new A(); // Связали ссылку A с объектом типа A.
a = new B();   // Заменили объект по ссылке с типом A
               // на объект с типом B.
a.M(); // Вывод – A.
```

ПОЗДНЕЕ СВЯЗЫВАНИЕ

```
class A
{
    public virtual void M() => Console.Write("A");
}
class B : A
{
    public override void M() => Console.Write("B");
}
```

Для запуска позднего связывания нужна пара: virtual и override

Версия метода будет выбрана по типу объекта

```
A a = new A(); // Связали ссылку A с объектом типа A.
a = new B();   // Заменили объект по ссылке с типом A
               // на объект с типом B.
a.M(); // Вывод – A.
```

НЕОБХОДИМЫЕ УСЛОВИЯ ПОЗДНЕГО СВЯЗЫВАНИЯ

- классы должны образовывать иерархию наследования
- в классах должны присутствовать методы с одинаковой сигнатурой
- элементы (методы) класса должны быть виртуальными, то есть должны быть отмечены модификаторами словами `virtual`, `override`

если в иерархии унаследованных классов объявляется

невиртуальный - раннее связывание

виртуальный - позднее связывание

ПОЛИМОРФИЗМ

- позволяет при помощи позднего связывания выбирать поведение по фактическому типу объекта
- часто устраняет необходимость в конструкции выбора

ПОНИМАЕМ ПОЛИМОРФИЗМ

Использование экранирования (new) и переопределения (override)

ПОЛИМОРФИЗМ

- Позволяет при помощи позднего связывания выбирать поведение по фактическому типу объекта
- Часто устраняет необходимость в конструкции выбора

```
class Bar {
        private string x = "bar";

        public virtual void Do() { Console.Write(x); }
    }
    class Foo : Bar {
        private string y = "foo";

        public override void Do() { Console.Write(y); }
    }
```

ПРИМЕР: НАПОМИНАНИЕ

```
public class A
{
    int x;

    public int X { get => x; }
    public override string ToString() => this.GetType().Name.ToString();
}
```

```
public class B : A // Наследник A.
{

}
public class C : A
{

}
```

```
class Program
{
    public static void Main()
    {
        A[] objs = { new A(), new B(), new C() };
        foreach (A obj in objs)
        {
            Console.WriteLine(obj);
        }
    }
}
```

МОДИФИКАЦИЯ С ЭКРАНИРОВАНИЕМ

```
public class A
{
    int x;

    public int X { get => x; }
    public override string ToString() =>
                    this.GetType().Name.ToString();
}
```

```
A[] objs = { new A(), new B(), new C() };

foreach (A obj in objs)
{
    if (obj is C)
    {
        Console.WriteLine((C)obj);
    }
}
```

```
public class B : A // Наследник A.
{
    public new string ToString() => this.GetType().Name.ToString() + "!";
}
public class C : A
{
    public new string ToString() => this.GetType().Name.ToString() + "!!";
}
```

МОДИФИКАЦИЯ С ПЕРЕОПРЕДЕЛЕНИЕМ

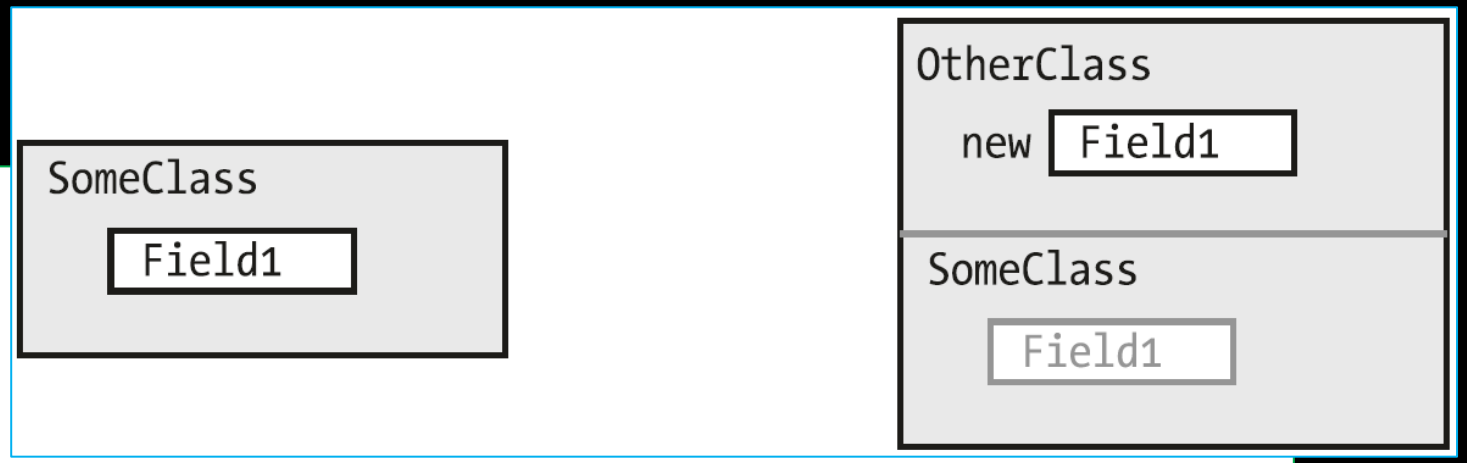
```
public class A
{
    int x;

    public int X { get => x; }
    public override string ToString() =>
                    this.GetType().Name.ToString();
}
```

```
A[] objs = { new A(), new B(), new C() };

foreach (A obj in objs)
{
    if (obj is C)
    {
        Console.WriteLine((C)obj);
    }
}
```

```
public class B : A // Наследник A.
{
    public override string ToString() => this.GetType().Name.ToString() + "!";
}
public class C : A
{
    public override string ToString() => this.GetType().Name.ToString() + "!!";
}
```

ЭКРАНИРОВАНИЕ (HIDING) ЧЛЕНОВ БАЗОВОГО КЛАССА

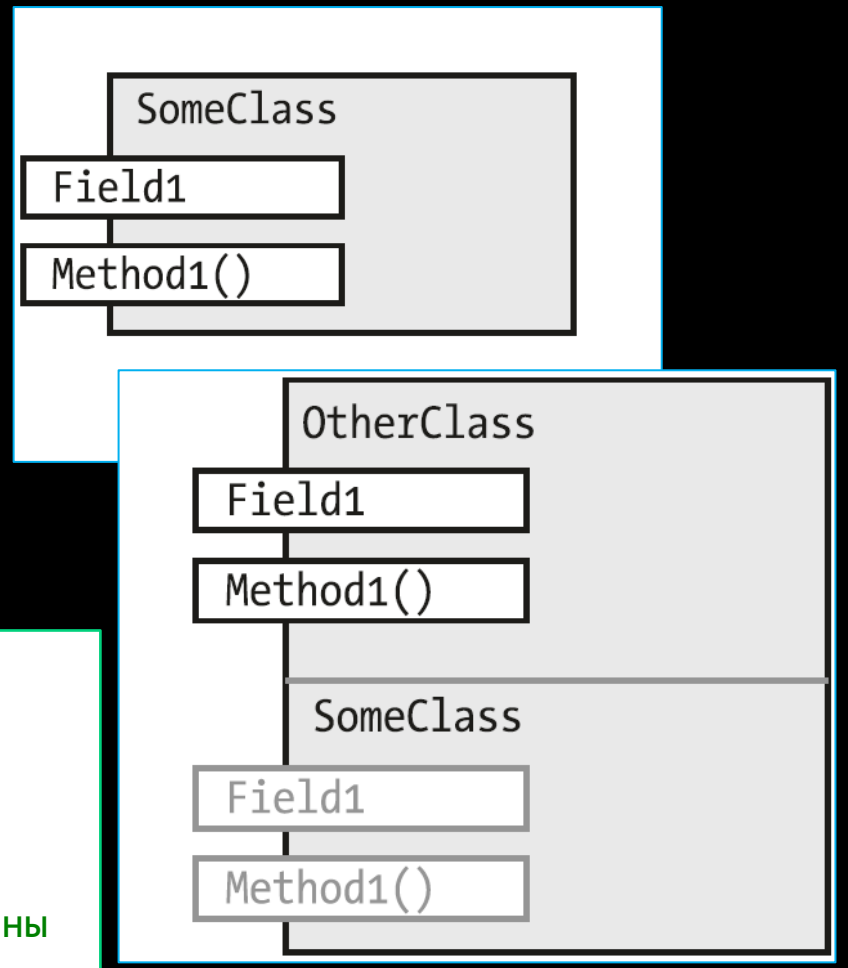
```
class SomeClass { // базовый класс
    string Field1;
    //...
}

class OtherClass : SomeClass { // производный класс
            new string Field1;  // маскируем член базового класса
             ↑
}    ключевое слово
```



ЭКРАНИРОВАНИЕ ДВУХ ЧЛЕНОВ

```
class SomeClass {    // базовый класс
    public string Field1 = "SomeClass Field1";
    public void Method1(string value)
    {
        Console.WriteLine("SomeClass.Method1: {0}", value);
    }
}
class OtherClass : SomeClass {    // производный класс
    new public string Field1 = "OtherClass Field1"; // Mask ...
    new public void Method1(string value) // Mask ...
    {
        Console.WriteLine("OtherClass.Method1: {0}", value);
    }
}
```

```
class Program
{
    static void Main()
    {
        OtherClass oc = new OtherClass();
        oc.Method1(oc.Field1); // Используем маскирующие члены
    }
}
```



ПРИМЕР С ВИРТУАЛЬНЫМ МЕТОДОМ

```
class BaseClass
{
    virtual public string Info => "This is the base class.";
}

class DerivedClass : BaseClass
{
    override public string Info => "This is the derived class.";

}
```

```
DerivedClass derived = new DerivedClass();
BaseClass mybc = (BaseClass)derived;
Console.WriteLine(derived.Info);  // вызов Print из производного класса
Console.Write(mybc.Info);     // вызов Print из производного класса
```

ПЕРЕОПРЕДЕЛЕНИЕ ПЕРЕОПРЕДЕЛЕННОГО МЕТОДА

```
class BaseClass
{
    virtual public string Info => "This is the base class.";
}
class DerivedClass : BaseClass
{
    override public string Info => "This is the derived class.";
}
class AnotherDerivedClass: DerivedClass
{
    override public string Info => "This is the second derived class.";
}
```

```
DerivedClass derived = new DerivedClass();
AnotherDerivedClass aDerived = new AnotherDerivedClass();
BaseClass mybc = (BaseClass)derived;
Console.WriteLine(derived.Info);  // вызов Info из производного класса
Console.Write(mybc.Info);     // вызов Info из производного класса
mybc = (BaseClass)aDerived;
Console.Write(mybc.Info);     // вызов Info из AnotherDerivedClass
```

ОБРАЩЕНИЕ К ПЕРЕОПРЕДЕЛЕНИЮ МЕТОДА, ПОМЕЧЕННОГО OVERRIDE

создаем объект типа AnotherDerivedClass:

- `AnotherDerivedClass aDerived = new AnotherDerivedClass();`

Создаем ссылку типа BaseClass:

- `mybc = (BaseClass)aDerived;`
- `Console.Write(mybc.Info);`

В зависимости от модификатора new / override в классе AnotherDerivedClass

- При override → Info из SecondDerived
- При new → Info из MyDerivedClass

Цепочку виртуальных членов классов иерархии разрывает использование оператора **new**

```
class A
{
    public virtual void M() => Console.Write("A");
}
class B : A
{
    public override void M() => Console.Write("B");
}
class C : B
{
    public new void M() => Console.Write("C");
}
```

```
A a = new A(); // Связали ссылку A с объектом типа A.
a = new C();   // Связали ссылку A с объектом типа C.
a.M();         // Вывод – B
```

ИСПОЛЬЗОВАНИЕ VIRTUAL И OVERRIDE В МЕТОДАХ

Виртуальным (virtual) может быть:

- Метод
- Свойство
- Индексатор
- Событие

```
class MyBaseClass   // базовый класс
{
    virtual public void Print()
    { ... }
}


class MyDerivedClass : MyBaseClass // производный класс
{
    override public void Print()
    { ... }
}
```

Виртуальный метод (его можно переопределить)

Переопределенный метод (переопределяет виртуальный метод родителя)

ПРИНЦИП ПОДСТАНОВКИ БАРБАРЫ ЛИСКОВ

Принцип подстановки Барбары Лисков (Liskov Substitution Principle, LSP)

- Функции, которые используют ссылки на базовые классы, должны иметь возможность использовать объекты производных классов, не зная об этом

Принцип подстановки Барбары Лисков (https://solidbook.vercel.app/lsp):

- помогает проектировать систему, опираясь на поведение модулей
- вводит ограничения и правила наследования объектов, чтобы их потомки не противоречили базовому поведению
- делает поведение модулей последовательным и предсказуемым
- помогает избегать дублирования, выделять общую для нескольких модулей функциональность в общий интерфейс
- позволяет выявлять при проектировании проблемные абстракции и скрытые связи между сущностями

НАРУШЕНИЕ LSP

Здесь неверный посыл, что все птицы умеют летать. Это ошибка проектирования иерархии классов

```
public class Bird
{
    public virtual string Fly() => "Each bird can fly";
}
```

```
public class Crow : Bird
{
    public override string Fly() => "Crow is flying...";
}
```

```
public class Ostrich : Bird
{
    // Здесь нарушается принцип подстановки Лисков.
    public override string Fly() => throw new Exception("Ostrich can't fly.");
}
```

```
static void Main(string[] args)
{
    Bird crow = new Crow();
    Bird ostrich = new Ostrich();

    MakeBirdFly(crow);
    MakeBirdFly(ostrich);
}

static void MakeBirdFly(Bird bird)
{
    bird.Fly();
}
```

ПЕРЕДАЧА ПОЛИМОРФНОЙ ССЫЛКИ В МЕТОД

```
public class Animal
{
    public virtual string MakeSound() => "The animal makes a sound";
}
public class Dog : Animal
{
    public override string MakeSound() => "The dog barks";
}
public class Cat : Animal
{
    public override string MakeSound() => "The cat meows";
}
```

```
static void MakeAnimalSound(Animal animal)
{
    animal.MakeSound();
}
```

```
Animal myDog = new Dog();
Animal myCat = new Cat();

MakeAnimalSound(myDog);
MakeAnimalSound(myCat);
```

ИСПОЛЬЗОВАННАЯ ЛИТЕРАТУРА

- Буч, Гради, Роберт А., Энгл, Майкл У., Янг, Бобби Дж., Коналлен, Джим, Хьюстон, Келли А. Объектно-ориентированный анализ и проектирование с примерами приложений, 3-е изд. – М.: ООО «И.Д. Вильямс», 2008.
- Meyer, B. (1997). Object-Oriented Software Construction. 2nd Edition. Prentice Hall: Upper Saddle River, N.J.
- Martin, R. C. (2002). Agile Software Development, Principles, Patterns, and Practices. Prentice Hall/Pearson Education: Upper Saddle River, N.J.