

Инкапсуляция. Наследование. Полиморфизм.

Авторы: Кравчук Анжелика Ивановна
Саркисян Гаяне Феликсовна

Что такое инкапсуляция?

Инкапсуляция

Способность типа, позволяющее объединить данные и методы, работающие с ними, скрывая при этом внутренние данные и детали реализации, делая доступными для приложений только определенные части типа

Единственный способ взаимодействия внешнего кода с объектом или классом это осуществление доступа через четко определенный набор членов типа (type contract)

Преимущества

Клиентские приложения не могут исказить состояние типа, выполняя изменения, вызывающие сбои в работе типа и приводящие к непредсказуемым результатам

Внешний код сосредоточен только на полезных свойствах объекта

Возможность легко изменить детали реализации типа без необходимости переписывать приложения, использующие тип

Преимущества сокрытия реализации

```
namespace QuickSort
{
    1 reference
    public static class Sort
    {
        3 references
        public static void quickSort(int[] a, int l, int r)
        {
            int temp;
            int x = a[l + (r - 1) / 2];
            int i = l;
            int j = r;
            while (i <= j)
            {
                while (a[i] < x) i++;
                while (a[j] > x) j--;
                if (i <= j)
                {
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                    i++; j--;
                }
            }
            if (i < r) quickSort(a, i, r);

            if (l < j) quickSort(a, l, j);
        }
    }
}
```

Solution 'Encapsulation' (2 projects)

- QuickSort
 - Properties
 - References
 - Class1.cs**
 - Class2.cs
- QuickSortConsole
 - Properties
 - References
 - App.config
 - Program.cs

```
static void Main(string[] args)
{
    int[] a = { 1, -48, 90, 234, 12, -45, 6, 7, -12,
               -56, 34, 123, 456, -894, 5, -56 };

    Sort.quickSort(a, 0, a.Length - 1);
}
```

Example

Преимущества сокрытия реализации

```
namespace QuickSort
```

```
{
```

```
    0 references
```

```
    public static class Sort2
```

```
    {
```

```
        0 references
```

```
        public static void Sort(int[] a)
```

```
        {
```

```
            quickSort(a, 0, a.Length - 1);
```

```
        }
```

```
        3 references
```

```
        private static void quickSort(int[] a, int l, int r)
```

```
        {
```

```
            int temp;
```

```
            int x = a[l + (r - 1) / 2]; int i = l; int j = r;
```

```
            while (i <= j)
```

```
            {
```

```
                while (a[i] < x) i++;
```

```
                while (a[j] > x) j--;
```

```
                if (i <= j)
```

```
                {
```

```
                    temp = a[i]; a[i] = a[j];
```

```
                    a[j] = temp; i++; j--;
```

```
                }
```

```
            }
```

```
            if (i < r) quickSort(a, i, r);
```

```
            if (l < j) quickSort(a, l, j);
```

```
        }
```

```
    }
```

```
}
```

Solution 'Encapsulation' (2 projects)

QuickSort

Properties

References

Class1.cs

Class2.cs

QuickSortConsole

Properties

References

App.config

Program.cs

```
static void Main(string[] args)
```

```
{
```

```
    int[] a = { 1, -48, 90, 234, 12, -45, 6, 7, -12,  
                -56, 34, 123, 456, -894, 5, -56 };
```

```
    Sort2.Sort(a);
```

```
}
```

Example

Модификаторы доступа

C# предоставляет ключевые слова, известные как модификаторы доступа, позволяющие задать уровень доступа для типов и их членов

public

Доступ к типу или члену возможен из любого кода в той же сборке или другой сборке, ссылающейся на него

private

Доступ к типу или члену можно получить только из кода в том же классе

protected

Доступ к типу или члену можно получить только из кода в том же классе либо в производном классе

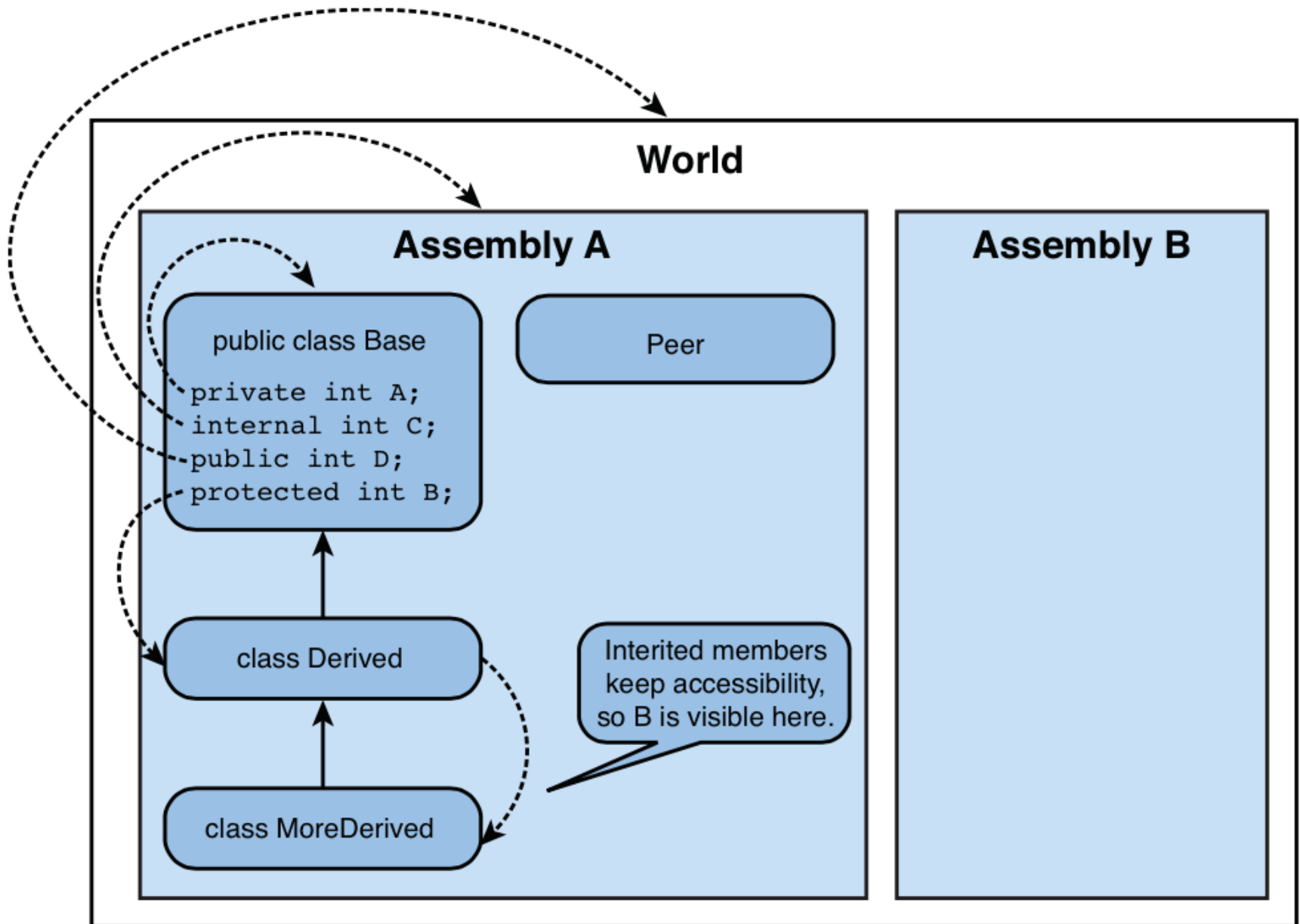
internal

Доступ к типу или члену возможен из любого кода в той же сборке, но не из другой сборки

protected internal

Доступ к типу или члену возможен из любого кода в той же сборке, либо из производного класса в другой сборке

Модификаторы доступа



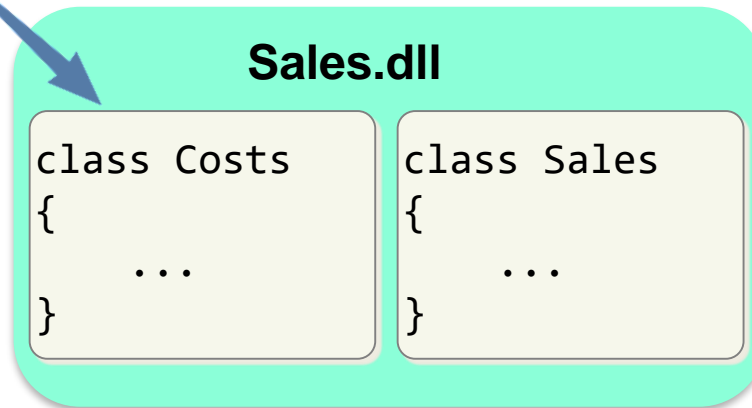
Сравнение internal и public типов

internal

Модификатор доступа по умолчанию для типа

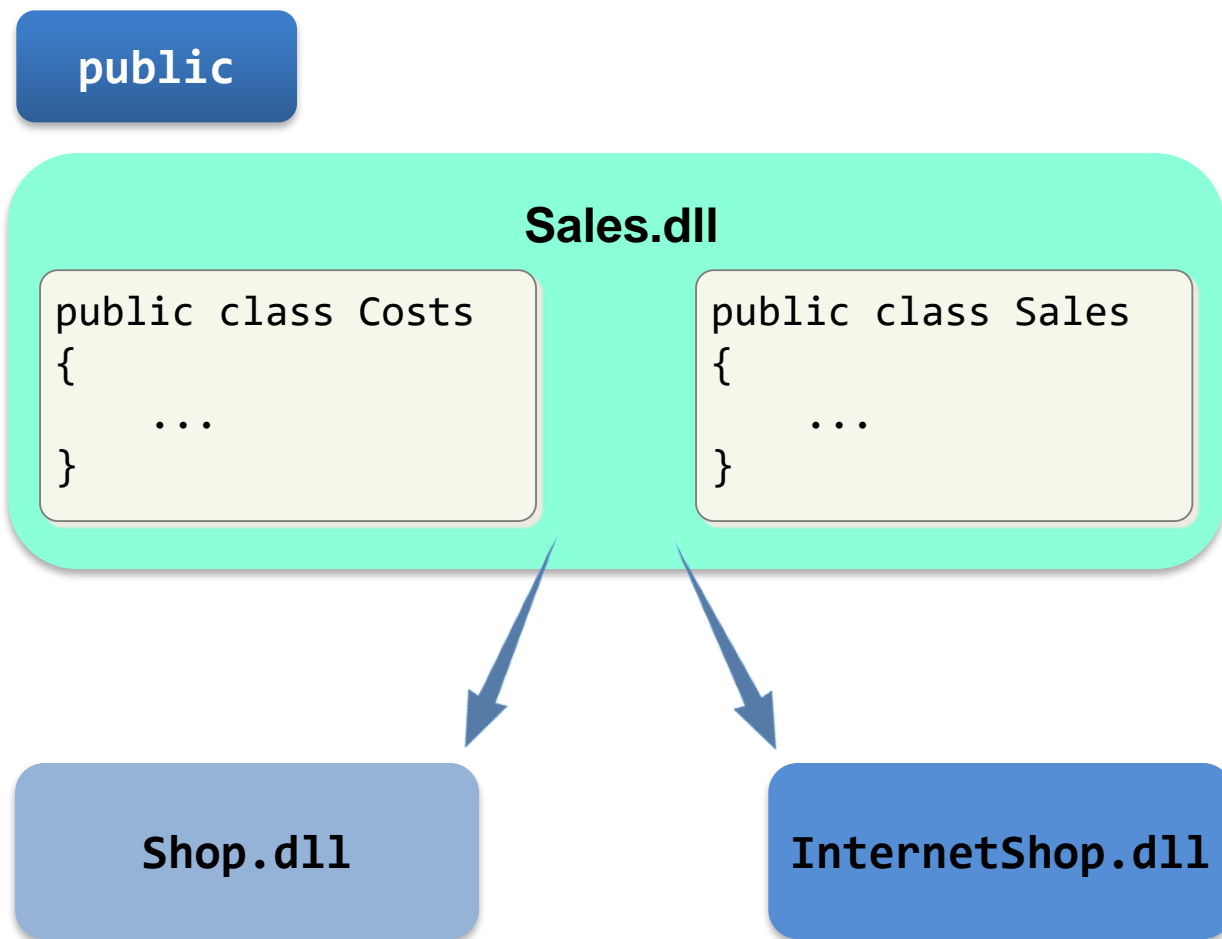
Ограничивает видимость кода типа только в пределах типов, определенных в той же сборке

Типы доступны только
типам в Sales.dll



Аналогичные правила видимости распространяются на internal поля и свойства

Сравнение internal и public типов



Сравнение private и public членов типов

private

Тип можно определить как private, только если он вложен в другой тип

```
public class Sales
{
    private Revenue salesRevenue;
    public void SetRevenue(string currency, double amount)
    {
        this.salesRevenue = new Revenue(currency, amount);
    }
    private struct Revenue
    {
        string currency;
        double amount;
        public Revenue(string currency, double amount)
        {
            this.currency = currency;
            this.amount = amount;
        }
    }
}
```

Сравнение private и public членов типа

public

Наибольший разрешительный уровень доступа

```
class Sales
{
    private double monthlyProfit;
    public void SetMonthlyProfit(double monthlyProfit)
    {
        this.monthlyProfit = monthlyProfit;
    }
    public double GetAnnualProfitForecast()
    {
        return (this.monthlyProfit * 12);
    }
}
```

Любой тип может
получить доступ к
public члену

```
class Program
{
    static void Main()
    {
        Sales companySales = new Sales();
        companySales.SetMonthlyProfit(3400);
        Console.WriteLine(companySales.GetAnnualProfitForecast());
    }
}
```

Скрытие внутренних данных

LINQPadQueries.Properties 1.

```
class Employee
{
    private int salary;
    private string name;
    private string department;

    public string GetName() { return name; }
    public void SetName(string value){ name = value; }

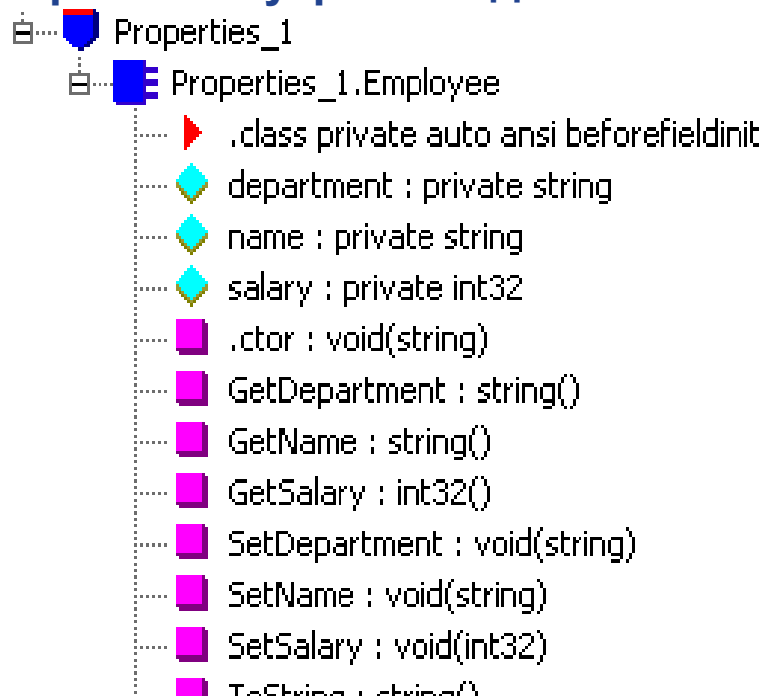
    public string GetDepartment() { return department; }
    public void SetDepartment(string value){ department = value; }

    public int GetSalary() { return salary; }
    public void SetSalary(int value)
    { salary = (value >= 0 && value <= 1000000) ? value : 0; }

    public Employee(string name)
    {
        this.name = name;
        this.salary = 10000;
        this.department = "Customer Service";
    }

    public override string ToString()
    {
        return String.Format("{0} earns ${1} and is in the {2} department.",
                               name, salary.ToString(), department.ToLower());
    }
}
```

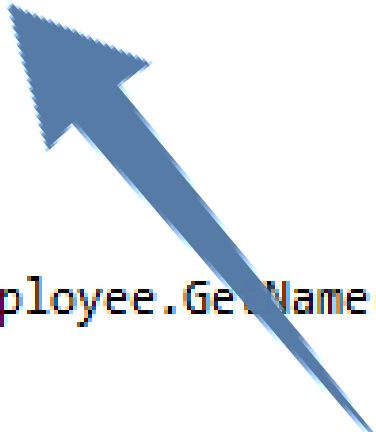
Соккрытие внутренних данных



```
.method public hidebysig instance void SetSalary(int32 'value') cil managed
{
    // Code size          25 (0x19)
    .maxstack 8
    IL_0000: nop
    IL_0001: ldarg.0
    IL_0002: ldarg.1
    IL_0003: ldc.i4.0
    IL_0004: blt.s          IL_000e
    IL_0006: ldarg.1
    IL_0007: ldc.i4          0xf4240
    IL_000c: ble.s          IL_0011
    IL_000e: ldc.i4.0
    IL_000f: br.s          IL_0012
    IL_0011: ldarg.1
    IL_0012: nop
    IL_0013: stfld          int32 Properties_1.Employee::salary
    IL_0018: ret
} // end of method Employee::SetSalary
```

Скрытие внутренних данных

```
Employee employee = new Employee("Joe");  
Console.WriteLine(employee);  
  
employee.SetSalary(12000000);  
  
Console.WriteLine("name: " + employee.GetName());  
  
Console.WriteLine(employee);
```



```
public Employee(string name)  
{  
    this.name = name;  
    this.salary = 10000;  
    this.department = "Customer Service";  
}
```

Скрытие внутренних данных

```
Employee employee = new Employee("Joe");  
Console.WriteLine(employee);
```

```
employee.SetSalary(12000);
```

```
Console.WriteLine("name: " + employee.Name);
```

```
Console.WriteLine(employee.ToString());
```

```
public override string ToString()  
{  
    return String.Format("{0} earns ${1} and is in the {2} department.",  
                           name, salary.ToString(), department.ToLower());  
}
```

```
Joe earns $100000 and is in the customer service department.  
name: Joe  
Joe earns $0 and is in the customer service department.
```

Скрытие внутренних данных

```
Employee employee = new Em  
Console.WriteLine(employee
```

```
employee.SetSalary(12000000);
```

```
Console.WriteLine("name: " + employee.GetName();)
```

```
Console.WriteLine(employee);
```

```
public void SetSalary(int value)  
{  
    salary = (value >= 0 && value <= 1000000) ? value : 0;  
}
```

```
public string GetName() { return name; }
```

```
Joe earns $10000 and is in the customer service department.  
name: Joe  
Joe earns $0 and is in the customer service department.
```

Что такое свойство?



Свойства позволяют осуществлять

контролируемый доступ к полям

проверку данных

контроль чтения/записи

Модификаторы свойства

Статический модификатор

static

Модификатор доступа

public internal private protected

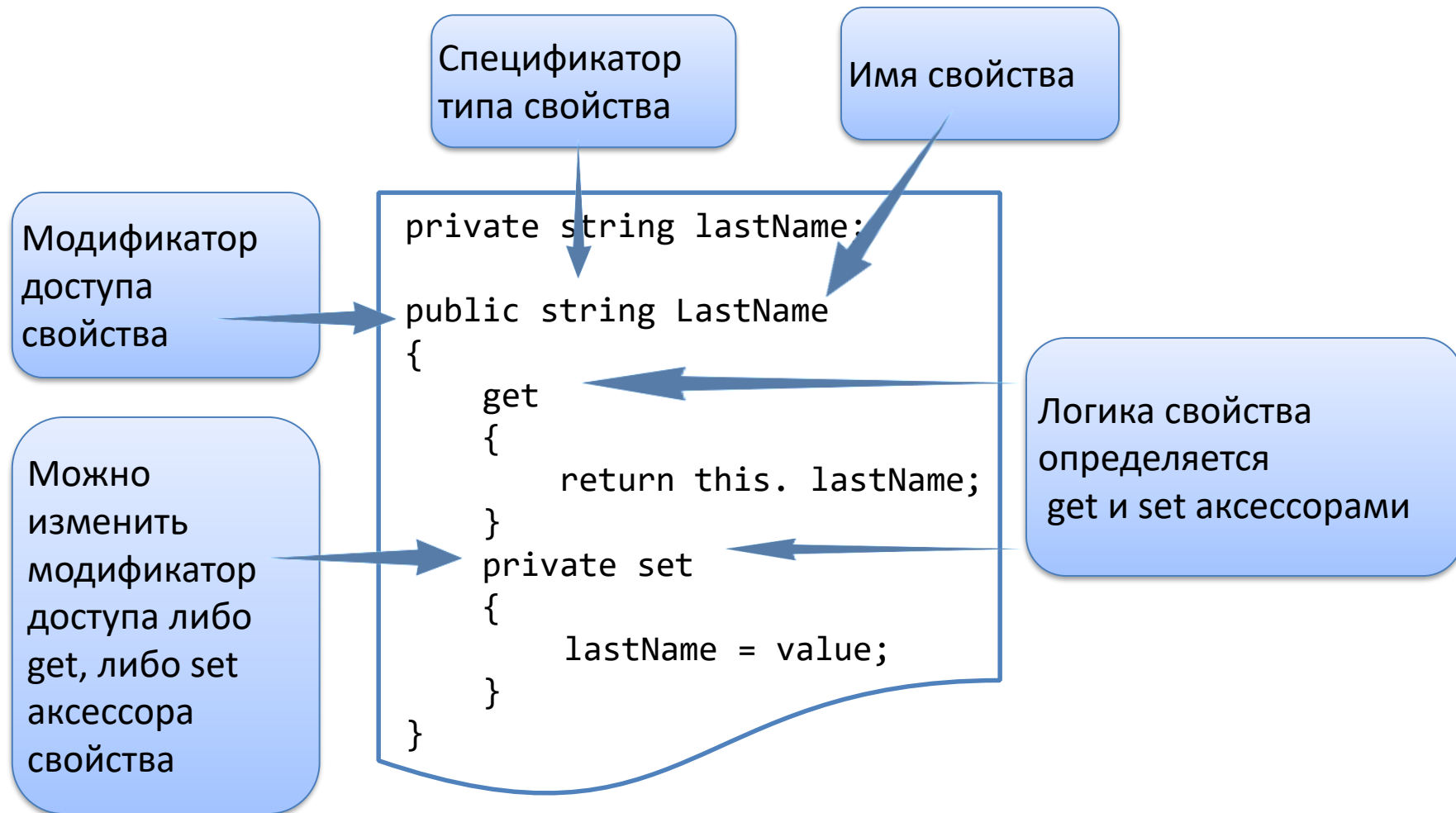
Модификатор наследования

new virtual abstract override sealed

Модификатор небезопасного кода

unsafe extern

Определение свойства



Set-аксессор всегда имеет один параметр типа, предоставляемый свойством

LINQPadQueries.Properties 2.

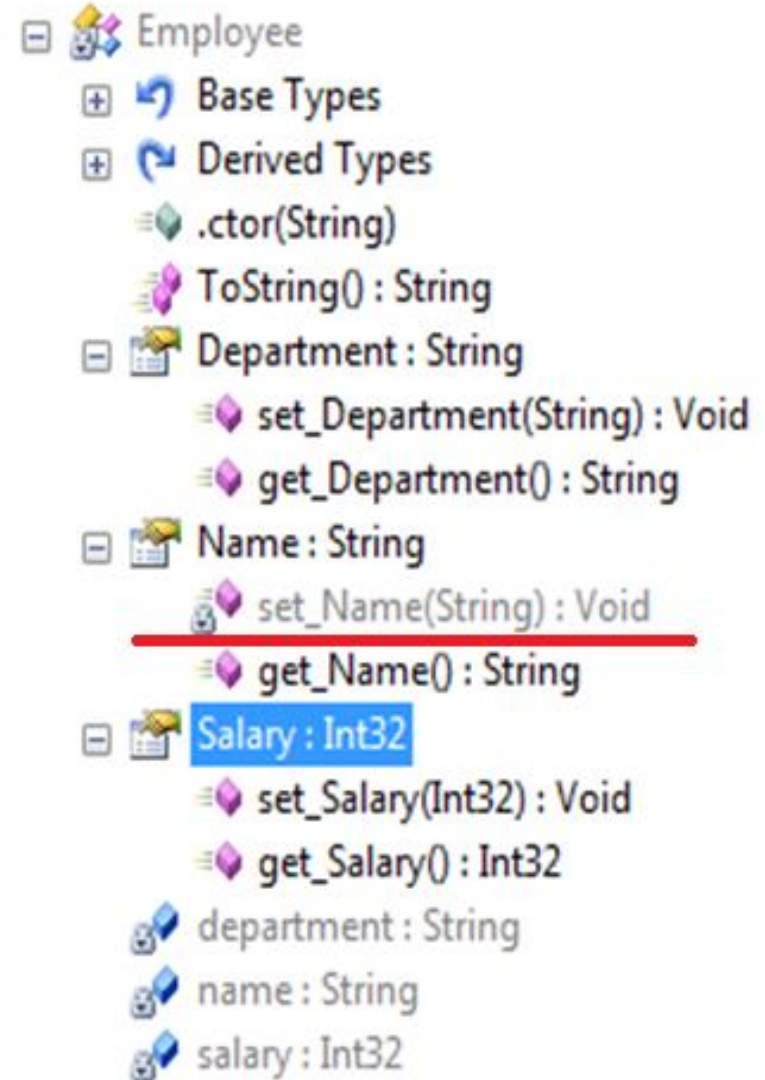
```
class Employee{
    private DateTime salary;
    private string name;
    private string department;

    public DateTime Salary
    {
        get { return salary; }
        set { salary = value; }
    }

    public string Name
    {
        get { return name; }
        private set { name = value; }
    }

    public string Department {
        get { return department; }
        set { department = value; }
    }

    public Employee(string name)
    {
        this.Name = name;
        this.Department = string.Empty;
    }
}
```



The image shows the Properties window in LINQPad for the Employee class. The window is organized into a tree view on the left and a list of properties on the right. The tree view includes 'Employee', 'Base Types', 'Derived Types', 'Department : String', 'Name : String', and 'Salary : Int32'. The 'Salary : Int32' property is highlighted with a blue background. The list of properties on the right includes: '.ctor(String)', 'ToString() : String', 'set_Department(String) : Void', 'get_Department() : String', 'set_Name(String) : Void', 'get_Name() : String', 'set_Salary(Int32) : Void', 'get_Salary() : Int32', 'department : String', 'name : String', and 'salary : Int32'. A red horizontal line is drawn under the 'set_Name(String) : Void' and 'get_Name() : String' properties.

- Employee
 - Base Types
 - Derived Types
 - .ctor(String)
 - ToString() : String
 - Department : String
 - set_Department(String) : Void
 - get_Department() : String
 - Name : String
 - set_Name(String) : Void
 - get_Name() : String
 - Salary : Int32
 - set_Salary(Int32) : Void
 - get_Salary() : Int32
 - department : String
 - name : String
 - salary : Int32

```


















class Employee{
    private DateTime salary;
    private string name;
    private string department;

    public DateTime Salary
    {
        get { return salary; }
        set { salary = value; }
    }

    public string Name
    {
        get { return name; }
        private set { name = value; }
    }
    public string Department {
        get { return department; }
        set { department = value; }
    }

    public Employee(string name)
    {
        this.Name = name;
        this.Department = string.Empty;
    }
}

```

 .class private auto ansi beforefieldinit
 Employee
 .class nested private auto ansi beforefieldinit
 field department : private string
 field name : private string
 field salary : private int32
 method .ctor : void(string)
 method ToString : string()
 method get_Department : string()
 method get_Name : string()
 method get_Salary : int32()
 method set_Department : void(string)
 method set_Name : void(string)
 method set_Salary : void(int32)
 prop Department : instance string()
 prop Name : instance string()
 prop Salary : instance int32()

Employee

- .class nested private auto ansi beforefieldinit
- field department : private string
- field name : private string
- field salary : private int32
- method .ctor : void(string)
- method ToString : string()
- method get_Department : string()
- method get_Name : string()
- method get_Salary : int32()
- method set_Department : void(string)
- method set_Name : void(string)
- method set_Salary : void(int32)
- prop Department : instance string()
- prop Name : instance string()**
- prop Salary : instance int32()

Employee::prop Name : instance string()

Find Find Next

```
.property instance string Name()
{
    .get instance string ConsoleApplication1.Program/Employee::get_Name()
    .set instance void ConsoleApplication1.Program/Employee::set_Name(string)
} // end of property Employee::Name
```

Employee

- ▶ .class nested private auto ansi beforefieldinit
- ◆ field department : private string
- ◆ field name : private string
- ◆ field salary : private int32
- method .ctor : void(string)
- method ToString : string()
- method get_Department : string()
- method get_Name : string()
- method get_Salary : int32()
- method set_Department : void(string)
- method set_Name : void(string)
- method set_Salary : void(int32)
- ▲ prop Department : instance string()
- ▲ prop Name : instance string()
- ▲ prop Salary : instance int32()

Employee::method set_Name : void(string)

```

Find Find Next
.method private hidebysig specialname instance void
    set_Name(string 'value') cil managed
{
    // Code size          9 (0x9)
    .maxstack 8
    IL_0000: nop
    IL_0001: ldarg.0
    IL_0002: ldarg.1
    IL_0003: stfld      string ConsoleApplication1.Program/Employee::name
    IL_0008: ret
} // end of method Employee::set_Name
  
```

Рекомендации по определению и использованию свойств

Свойства следует использовать надлежащим образом

BankAccount

✗ `double Balance (get, set)`
✓ `WithdrawMoney(double Amount)`
✓ `DepositMoney (double Amount)`

Не следует представлять каждое поле как свойство, если для этого нет веских оснований

Не следует реализовывать get аксессоры с побочными эффектами

Следует использовать соглашения об именовании

```
int data;  
public int Data  
{  
    get  
    {  
        return Data;  
    }  
    ...  
}
```

StackOverflowException

Создание экземпляра объекта с помощью свойства

```
class Employee
{
    public Employee ()
    {
        ...
    }
    public Employee (int grade)
    {
        ...
    }
    public string Name { get; set; }
    public string Department { get; set; }
    ...
}
```

Нужно стараться определять только конструкторы, устанавливающие все необходимые значения свойств по умолчанию

```
Employee louisa = new Employee() { Department = "Technical" };
Employee john = new Employee { Name = "John" };
Employee mike = new Employee
{
    Name = "Mike",
    Department = "Technical"
};
```

Инициализация объекта

Автоматические свойства (Automatically Implemented Properties, AIP)

```
public string Name { get; set; }
```

При использовании автоматического свойства, компилятор создает private поля и автоматически генерирует код для чтения и записи этого поля

```
private string <Name>k__BackingField;  
public string Name  
{  
    get  
    {  
        return <Name>k__BackingField;  
    }  
    set  
    {  
        this._name = value;  
    }  
}
```

Везде, где необходимо добавить поле и можно его сделать public, а не писать свойство для получения и установки его значения, можно использовать автоматические свойства

Полезны, когда не требуется дополнительной обработки или проверки значений полей

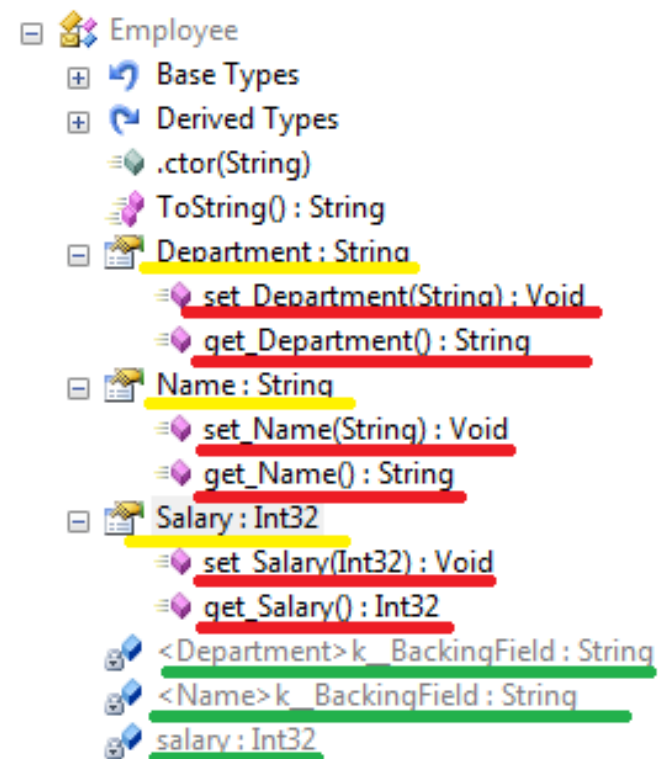
Не возникает никаких последствий при переходе от автоматических свойств на определенные

Автоматические свойства

```
class Employee
{
    public int salary;

    public string Name { get; set; }
    public string Department { get; set; }
    public int Salary {...}

    public Employee(string Name) {...}
    public override string ToString() {...}
}
```



Создание экземпляра объекта с помощью свойства

```
class Employee
{
    private string name;
    private string department;
    // Initialize both fields
    public Employee(string empName, string empDepartment)
    {
        this.name = empName;
        this.department = empDepartment;
    }
    // Initialize name only
    public Employee(string empName)
    {
        this.name = empName;
    }
    // Initialize department only
    public Employee(string empDepartment)
    {
        this.department = empDepartment
    }
    ...
}
```

Компилятор не может различить два конструктора, принимающих один параметр

СТЕ

```
// Is "Fred" the name of an employee or a department?
Employee myEmployee = new Employee("Fred");
```

Недостатки автоматического свойства:

1. Синтаксис объявления поля может включать инициализацию, таким образом, вы объявляете и инициализируете поле в одной строке кода. Однако нет подходящего синтаксиса для установки при помощи AIP начального значения. **Следовательно, необходимо неявно инициализировать все автоматически реализуемые свойства во всех конструкторах.**

2. Механизм сериализации на этапе выполнения сохраняет имя поля в сериализованном потоке. Имя резервного поля для AIP определяется компилятором, и он может менять это имя каждый раз, когда компилирует код, сводя на нет возможность десериализации экземпляров всех типов, содержащих автоматически реализуемые свойства. **Не используйте этот механизм для типов, подлежащих сериализации и десериализации.**

LINQPadQueries.Properties 2.

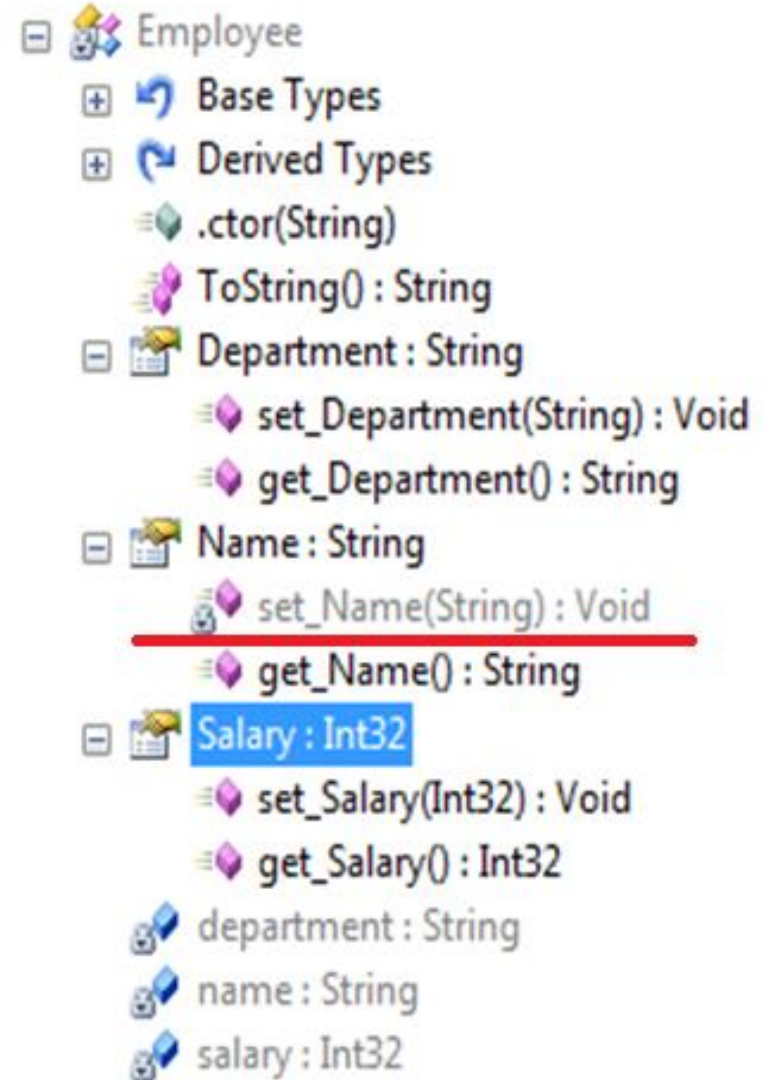
```
class Employee{
    private DateTime salary;
    private string name;
    private string department;

    public DateTime Salary
    {
        get { return salary; }
        set { salary = value; }
    }

    public string Name
    {
        get { return name; }
        private set { name = value; }
    }

    public string Department {
        get { return department; }
        set { department = value; }
    }

    public Employee(string name)
    {
        this.Name = name;
        this.Department = string.Empty;
    }
}
```



The image shows the LINQPad Properties window for the `Employee` class. The window is organized into a tree view on the left and a list of properties on the right. The `Salary : Int32` property is highlighted with a blue background and a red underline. The `Department : String` and `Name : String` properties are also visible. The `Salary` property has a `set_Salary(Int32) : Void` method and a `get_Salary() : Int32` method. The `Department` property has a `set_Department(String) : Void` method and a `get_Department() : String` method. The `Name` property has a `set_Name(String) : Void` method and a `get_Name() : String` method. The `Salary` property is also listed as a field in the bottom section of the window.

- Employee
 - Base Types
 - Derived Types
 - .ctor(String)
 - Tostring() : String
 - Department : String
 - set_Department(String) : Void
 - get_Department() : String
 - Name : String
 - set_Name(String) : Void
 - get_Name() : String
 - Salary : Int32
 - set_Salary(Int32) : Void
 - get_Salary() : Int32
 - department : String
 - name : String
 - salary : Int32

Определение свойств в интерфейсе

Интерфейс определяет контракт, специфицирующий методы, которые класс должен реализовывать

Поля не могут быть определены в интерфейсе

Свойства могут быть определены в интерфейсе

Детали реализации свойств являются ответственностью класса

Используется тот же синтаксис
автоматических свойств

```
interface IPerson
{
    string Name { get; set; }
    int Age { get; }
    DateTime DateOfBirth { set; }
}
```

Нельзя указать модификатор доступа

Не обязательно указывать оба аксессуора

Рекомендации по определению и использованию свойств

Свойства могут быть «только для чтения» или «только для записи», а поля всегда доступны и для чтения, и для записи.

Метод свойства может привести к исключению, а при доступе к полям исключений не бывает

Свойства нельзя передавать в метод как параметры с ключевым словом `out` или `ref`

Свойство-метод может выполняться довольно долго, а доступ к полям выполняется моментально

При вызове несколько раз подряд метод свойства может возвращать разные значения (`System.DateTime.Now`), а поле возвращает одно и то же значение

Метод свойства может создавать наблюдаемые сторонние эффекты, а при доступе к полю это невозможно

Что такое индекатор?

Индексатор обеспечивает механизм инкапсуляции множества значений, так же, как свойство инкапсулирует одно значение

get и set аксессоры используются для управления тем, как значения извлекаются или устанавливаются на основе индекса передаваемого в качестве параметра для индексации

get и set аксессоры используют свойство-подобный синтаксис

Индексатор использует массив-подобный при доступе к элементам множества

При индексации можно использовать нецелый тип индекса

```
CustomerAddressBook addressBook = ...;  
Address customerAddress = addressBook["a2332"];  
...  
Address customerAddress = addressBook[99];
```

Можно определить перегруженные индексаторы

Создание индексатора

Модификатор
доступа

Тип возвращаемого
значения

Имя индексатора
всегда this

Типы и имена
параметров

```
public Address this[string CustomerID]
{
    get
    {
        return database.FindCustomer(CustomerID);
    }
    set
    {
        database.UpdateCustomer(CustomerID, value);
    }
}
```

Параметры индексатора
могут быть описаны как
параметры-значения
или как параметр-
список

При написании индексатора следует убедиться, что он содержит логику обработки ошибки в случае, когда код принимает недопустимое значение индекса

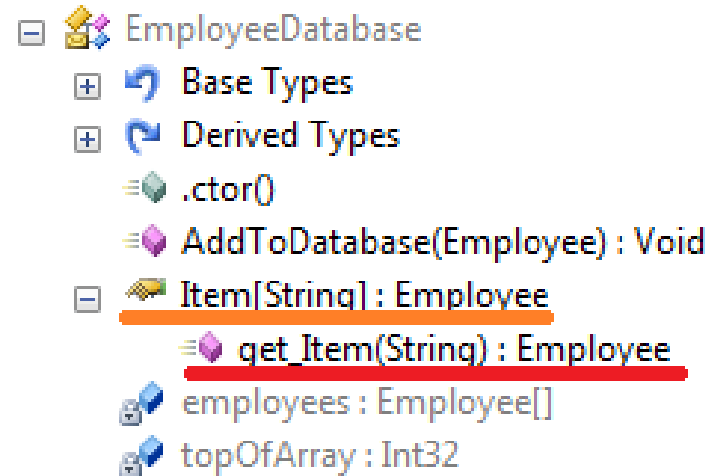
Нельзя определить статические индексаторы

Создание индексатора

```
class EmployeeDatabase
{
    employees [] Employee;
    int topOfArray;

    public EmployeeDatabase(){...}
    public void AddToDatabase(Employee employee){...}

    public Employee this[string name]{...}
}
```



EmployeeDatabase

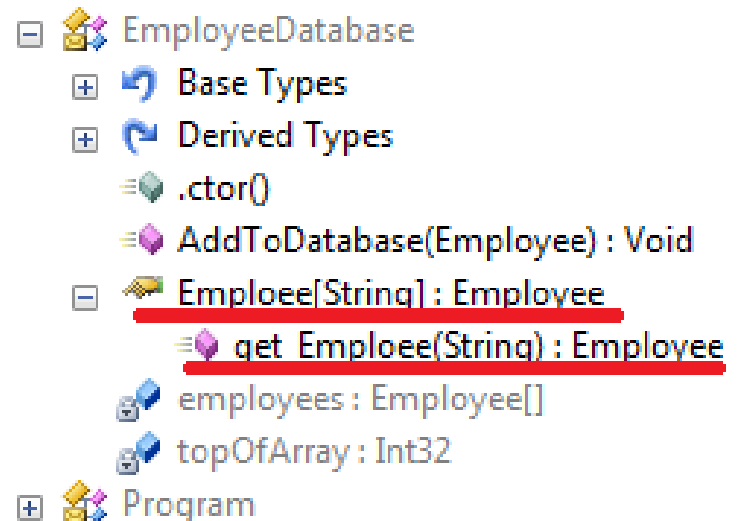
- Base Types
- Derived Types
- .ctor()
- AddToDatabase(Employee) : Void
- Item[String] : Employee
 - get_Item(String) : Employee
- employees : Employee[]
- topOfArray : Int32

Создание индексатора

```
using System.CompilerServices.Runtime;

class EmployeeDatabase
{
    employees [] Employee;
    int topOfArray;


    public EmployeeDatabase(){...}
    public void AddToDatabase(Employee employee){...}
    [IndexerName("Emploee")]
    public Employee this[string name]{...}
}
```



Создание индексатора

В интерфейсе можно указать индексатор, тогда любой реализующий интерфейс класс должен реализовать и этот индексатор

Нельзя указать
модификатор
доступа



```
interface IEmployeeDatabase
{
    Employee this[string Name] { get; set; }
}
```

Не обязательно указывать оба аксессора

Реализовать индексатор в реализующем интерфейс классе можно явно или неявно

```
class EmployeeDatabase : IEmployeeDatabase
{
    public Employee this[string Name]
    {
        get { ... return employee; }
        set { ... }
    }
}
```

Сравнение индексаторов и массивов

При использовании индексатора используется массиво-подобный синтаксис, однако между индексаторами и массивами существует несколько важных различий

Индексы

При индексировании элементов в массиве можно использовать только числовые индексы

Индексаторы предоставляют возможность использования не числовых индексов

Перегрузка

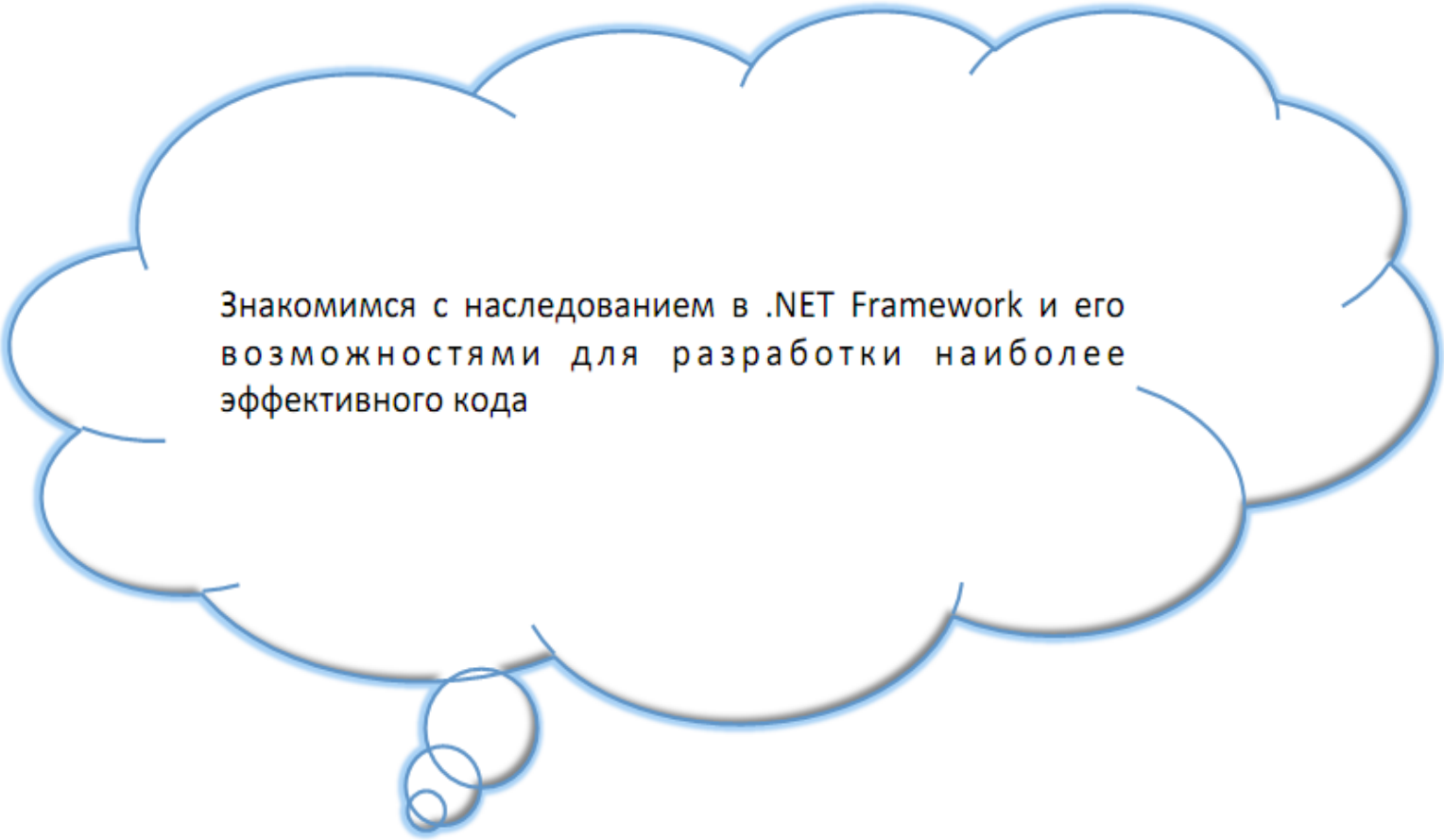
Операцию индексирования нельзя перегрузить

Возможно переопределять и перегружать индексаторы

Использование в качестве параметра

Индексированные элементы массива могут использоваться как при передаче параметров по значению, так и как ref и out параметры

Индексаторы нельзя использовать в качестве ref или out параметров, но можно при передаче параметров по значению



Знакомимся с наследованием в .NET Framework и его возможностями для разработки наиболее эффективного кода

Что такое наследование?

Наследование в C# бывает 2-х видов:

1. Наследование **реализации**.
2. Наследование **интерфейса** (типа, а не интерфейса класса)

В C# поддерживается только единичное наследование реализации, и множественное наследование интерфейсов.

Наследование **реализации**, в свою очередь используются наследование ***is a*** и ***has a***.

Has a – это наследование реализации, когда класс является частью другого класса, т.е. объект одного класса может быть полем другого класса, при этом и здесь рассматривается два вида это ***композиция*** и ***агрегация***.

Is a - это классическое открытое наследование, которое ***необходимо использовать только когда производный класс и есть базовый***, т.к. при наследовании реализации типа ***is a***, ссылка базового класса может быть проинициализирована ссылкой производного класса.

Что такое наследование?

Наследование реализации - это свойство системы, позволяющее описать новый класс на основе существующего с целью повторного использования, расширения и изменения функциональности базового класса

Базовый класс

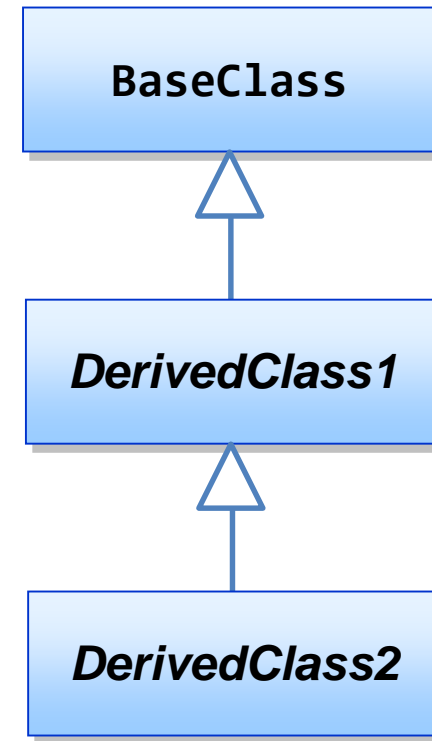
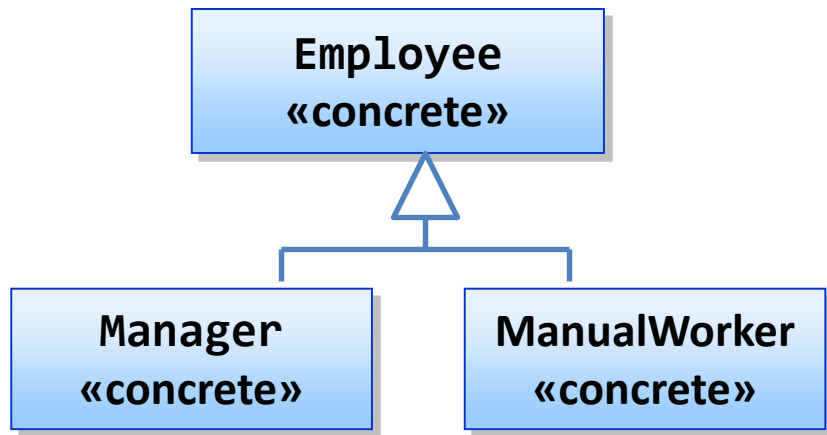
```
class Employee
{
    protected string empNum;
    protected string empName;
    protected void DoWork()
    { ... }
}
```

Производные классы

```
// Inheriting classes
class Manager : Employee
{
    public void DoManagementWork()
    { ... }
}

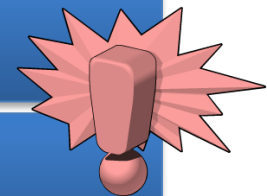
class ManualWorker : Employee
{
    public void DoManualWork()
    { ... }
}
```


Что такое наследование?



С# поддерживает только единичное наследование реализации

Наследование является транзитивным



Классическое открытое наследование (is a)

Наследование позволяет создавать новые классы, которые

- 1. повторно используют код (reusable code)*
- 2. имеют возможность расширения* - добавить свою функциональность
- 3. изменять поведение* базового класса, при этом это возможно только при использовании механизма виртуальных функций.

Наследуется все кроме конструкторов и финализатора, даже статические данные.

Данные с модификатором `private` напрямую не доступны в производном классе.

Следовательно, если необходим доступ к закрытой части в производном классе, то используется модификатор `protected`.

Вызов конструкторов базового класса

```
public class BaseClass
{
    private int numBase;

    public BaseClass(int number){
        numBase = number;
        Console.WriteLine("BaseClass(int)");
    }
    ...
}
```

Конструктор базового класса

```
public class DerivedClass : BaseClass
{
    private string name;
    private int numDerived;

    public DerivedClass (int a, int b, string name): base(a) {
        numDerived = b;
        this.name = name;
        Console.WriteLine("DerivedClass(int , int , string)");
    }
    ...
}
```

Вызывается BaseClass (int number)

Хорошей практикой для конструктора производного класса является вызов конструктора базового класса как части инициализации

Вызов конструкторов базового класса

```
void Main()
{
    BaseClass b1 = new BaseClass(7);
    b1.Dump("b1");

    ("\n").Dump();

    DerivedClass d1 = new DerivedClass(8, -67, "Tom");
    d1.Dump("d1");
}
```

BaseClass(int)

b1

BaseClass	
BaseClass has number 7	
NumBase	7

BaseClass..ctor:

```
IL_0000: ldarg.0
IL_0001: call      System.Object..ctor
IL_0006: nop
IL_0007: nop
IL_0008: ldarg.0
IL_0009: ldarg.1
IL_000A: stfld      UserQuery+BaseClass.numBase
IL_000F: ldstr      "BaseClass(int)"
IL_0014: call      System.Console.WriteLine
IL_0019: nop
IL_001A: nop
IL_001B: ret
```

Вызов конструкторов базового класса

```
void Main()
{
    BaseClass b1 = new BaseClass(7);
    b1.Dump("b1");

    ("\n").Dump();

    DerivedClass d1 = new DerivedClass(8, -67, "Tom");
    d1.Dump("d1");
}
```

DerivedClass..ctor:

IL_0000: ldarg.0

IL_0001: ldarg.1

IL_0002: call UserQuery+BaseClass..ctor

IL_0007: nop

IL_0008: nop

IL_0009: ldarg.0

IL_000A: ldarg.2

IL_000B: stfld UserQuery+DerivedClass.numDerived

IL_0010: ldarg.0

IL_0011: ldarg.3

IL_0012: stfld UserQuery+DerivedClass.name

IL_0017: ldstr "DerivedClass(int , int , string)"

IL_001C: call System.Console.WriteLine

IL_0021: nop

IL_0022: nop

IL_0023: ret

BaseClass(int)
DerivedClass(int , int , string)

d1

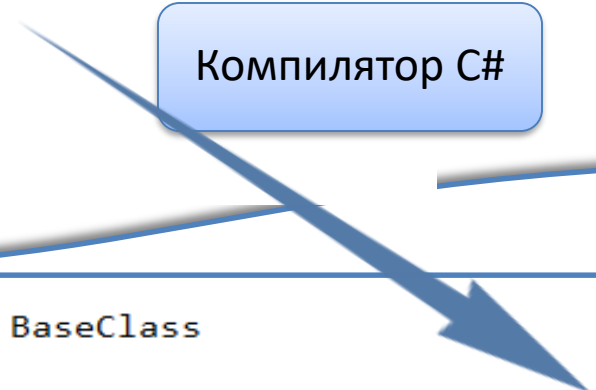
DerivedClass	
BaseClass has number 8 Tom -67	
NumBase	8
NumDerived	-67
Name	Tom

Вызов конструкторов базового класса

```
public class DerivedClass : BaseClass
{
    private string name;
    private int numDerived;

    public DerivedClass (int a, int b, string name) {
        numDerived = b;
        this.name = name;
    }
}
```

Компилятор C#



```
public class DerivedClass : BaseClass
{
    public DerivedClass (int a, int b, string name):base() {
        numDerived = b;
        this.name = name;
    }
}
```

Если в конструкторе производного класса нет явного вызова конструктора базового класса, перед выполнением кода компилятор пытается вставить в конструктор производного класса вызов конструктора по умолчанию базового класса

Вызов конструкторов базового класса

Рекомендуется придерживаться следующих правил, при наследовании:

1. Всегда в производном конструкторе явно вызывайте конструктор с параметрами;
2. Всегда в базовом классе прописывайте конструктор без параметров или конструктор по умолчанию, если прописали хотя бы один конструктор с параметром.

В каких классах компилятор не создает конструктор по умолчанию?

LINQPadQueries.Inheritance-> FIELDS

```

class Base
{
    public int x = F1();

    public Base()
    {
        F2();
    }

    public static int F1()
    {
        "Base.F1.static".Dump();
        return 0;
    }

    public void F2()
    {
        "Base.F2".Dump();
    }
}

void Main()
{
    Derived d = new Derived();
}

```

```

class Derived : Base
{
    public int x = F3();

    public Derived()
    {
        F4();
    }

    public static int F3()
    {
        "Derived.F3.static".Dump();
        return 0;
    }

    public void F4()
    {
        "Derived.F4".Dump();
    }
}

```



```

class Base
{
    public int x = F1();

    public Base()
    {
        F2();
    }

    public static int F1()
    {
        "Base.F1.static".Dump();
        return 0;
    }

    public void F2()
    {
        "Base.F2".Dump();
    }
}

```

```

class Derived : Base
{
    public int x = F3();

    public Derived()
    {
        F4();
    }

    public static int F3()
    {
        "Derived.F3.static".Dump();
        return 0;
    }

    public void F4()
    {
        "Derived.F4".Dump();
    }
}

```

▼ Results λ SQL IL

```

Derived.F3.static
Base.F1.static
Base.F2
Derived.F4


```

```

class Derived : Base
{
    public int x = F3();

    public Derived()
    {
        F4();
    }
}

```


Derived::.ctor : void()

Find Find Next

```

.method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
{
    // Code size          28 (0x1c)
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: call        int32 ConsoleApplication1.Program/Derived::F3()
    IL_0006: stfld        int32 ConsoleApplication1.Program/Derived::x
    IL_000b: ldarg.0
    IL_000c: call        instance void ConsoleApplication1.Program/Base::.ctor()
    IL_0011: nop
    IL_0012: nop
    IL_0013: ldarg.0
    IL_0014: call        instance void ConsoleApplication1.Program/Derived::F4()
    IL_0019: nop
    IL_001a: nop
    IL_001b: ret
} // end of method Derived::.ctor

```

```

class Base
{
    public int x = F1();

    public Base()
    {
        F2();
    }
}

```

Base::.ctor : void()

Find Find Next

```

.method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
{
    // Code size          28 (0x1c)
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: call        int32 ConsoleApplication1.Program/Base::F1()
    IL_0006: stfld      int32 ConsoleApplication1.Program/Base::x
    IL_000b: ldarg.0
    IL_000c: call        instance void [mscorlib]System.Object::.ctor()
    IL_0011: nop
    IL_0012: nop
    IL_0013: ldarg.0
    IL_0014: call        instance void ConsoleApplication1.Program/Base::F2()
    IL_0019: nop
    IL_001a: nop
    IL_001b: ret
} // end of method Base::.ctor

```

Присваивание и ссылка на классы в иерархии наследования

```
class Employee  
{  
  ...  
}
```

```
class Manager : Employee  
{  
  ...  
}
```

```
class ManualWorker : Employee  
{  
  ...  
}
```

```
...  
// Manager constructor expects a name and a grade  
Manager myManager = new Manager("Fred", "VP");  
ManualWorker myWorker = myManager;
```



CTE

```
Manager myManager = new Manager("Fred", "VP");  
Employee myEmployee = myManager;  
// legal, Employee is the base class of Manager
```

Можно ссылаться на объект другого типа до тех пор, пока этот тип является классом, находящимся выше в иерархии наследования

Иерархии наследования

```
Manager manager = new Manager("Fred", "VP");  
Employee employee = manager; // employee refers to a Manager  
...  
Manager managerAgain = employee as Manager;  
// OK - employee is a Manager  
...  
ManualWorker worker = new ManualWorker("Bert");  
employee = worker; // employee now refers to a ManualWorker  
...  
bool ok = employee is Manager;  
// returns false - employee is a ManualWorker
```

Операция **as** проверяет, является ли объект ссылкой на указанный тип и, если это так, возвращает новую ссылку, используя этот тип, в противном случае возвращает null

Операция **is** проверяет, является ли объект ссылкой на указанный тип и возвращает true, если это так и false в противном случае

Соккрытие методов базового класса

```
class Employee
{
    protected void DoWork()
    {
        ...
    }
}
```

```
class Manager : Employee
{
    public new void DoWork()
    {
        // Hide the DoWork method in the base class
        ...
    }
    ...
}
```

Соккрытие :
замена функциональности базового класса новым поведением

Для указания намеренного действия используется ключевое слово **new**

```
void Main()
{
    A a = new A();
    (a.ToString()).Dump();

    object o = a;
    (o.ToString()).Dump();
}
class A{
    public String ToString(){
        return "class A";
    }
}
```

```

void Main()
{
    A a = new A();
    (a.ToString()).Dump();

    object o = a;
    (o.ToString()).Dump();
}
class A{
    public String ToString(){
        return "class A";
    }
}

```

Results A SQL

```

class A
UserQuery+A

```


Переопределение виртуальных методов базового класса

```
class Object
{
    public virtual string ToString()
    {
        ...
    }
}

class Employee
{
    protected string empName;
    ...
    public override string ToString()
    {
        return string.Format("Employee: {0}", empName);
    }
}
```

Переопределение:

намеренное изменение или расширение абстрактной или виртуальной реализации унаследованного метода, свойства, индексатора или события базового класса

Для переопределения в наследуемом классе используется ключевое слово **override**

Переопределить можно только члены класса, которые помечены в базовом классе как **virtual**, **override** или **abstract**

Полиморфизм

```
class Employee
{
    public virtual string GetTypeName()
    {
        return "This is an Employee";
    }
}
```

```
class Manager : Employee
{
    public override string GetTypeName()
    {
        return "This is a Manager";
    }
}
```

```
Employee employee;
Manager manager = new Manager();
ManualWorker worker = new ManualWorker();
employee = manager;
Console.WriteLine(employee.GetTypeName());
employee = worker;
Console.WriteLine(employee.GetTypeName());
```

Виртуальные методы, определенные в классах, разделяющих иерархию наследования, позволяют вызывать различные версии одного и того же метода в зависимости от типа объекта, который определяется динамически во время выполнения

```
class ManualWorker : Employee
{
    // Does not override GetTypeName
}
```



LINQPadQueries.Inheritance -> polymorphism 1.

```
class A
{
    public virtual void M() { Console.WriteLine("метод M() класса A"); }
}
class B: A
{
    public override void M() { Console.WriteLine("метод M() класса B"); }
}
class C: B
{
    public override void M() { Console.WriteLine("метод M() класса C"); }
}
class D: C
{
    public override void M() { Console.WriteLine("метод M() класса D"); }
}
```

LINQPadQueries.Inheritance -> polymorphism 1.

TypeDefName: A (02000002)

Flags : [Public] [AutoLayout] [Class] [AnsiClass] [BeforeFieldInit] (00100001)

Extends : 01000001 [TypeRef] System.Object

Method #1 (06000001)

MethodName: M (06000001)

Flags : [Public] [Virtual] [HideBySig] [NewSlot] (000001c6)

RVA : 0x00002050

ImplFlags : [IL] [Managed] (00000000)

TypeDefName: B (02000003)

Flags : [Public] [AutoLayout] [Class] [AnsiClass] [BeforeFieldInit] (00100001)

Extends : 02000002 [TypeDef] ClassLibrary1.A

Method #1 (06000003)

MethodName: M (06000003)

Flags : [Public] [Virtual] [HideBySig] [ReuseSlot] (000000c6)

RVA : 0x00002060

LINQPadQueries.Inheritance -> polymorphism 1.

TypeDefName: C (02000004)

Flags : [Public] [AutoLayout] [Class] [AnsiClass] [BeforeFieldInit] (00100001)

Extends : 02000003 [TypeDef] ClassLibrary1.B

Method #1 (06000005)

MethodName: M (06000005)

Flags : [Public] [Virtual] [HideBySig] [ReuseSlot] (000000c6)

RVA : 0x0000207c

ImplFlags : [IL] [Managed] (00000000)

TypeDefName: D (02000005)

Flags : [Public] [AutoLayout] [Class] [AnsiClass] [BeforeFieldInit] (00100001)

Extends : 02000004 [TypeDef] ClassLibrary1.C

Method #1 (06000007)

MethodName: M (06000007)

Flags : [Public] [Virtual] [HideBySig] [ReuseSlot] (000000c6)

RVA : 0x00002092

ImplFlags : [IL] [Managed] (00000000)

LINQPadQueries.Inheritance -> polymorphism 1.

A		
v <u>ToString</u>	Object	A
v Equals		
v <u>HashCode</u>		
v. Finalize		
v M (A)		

B		
v <u>ToString</u>	Object	A, B
v Equals		
v <u>HashCode</u>		
v. Finalize		
o M (B)		

C		
v <u>ToString</u>	Object	A, B, C
v Equals		
v <u>HashCode</u>		
v. Finalize		
o M (C)		

D		
v <u>ToString</u>	Object	A, B, C, D
v Equals		
v <u>HashCode</u>		
v. Finalize		
o M(D)		

LINQPadQueries.Inheritance -> polymorphism 1.

```
void Main()
{
    A a = new A();
    a.M();
}
```

A		
v ToString	Object	A
v Equals		
v GetHashCode		
v.Finalize		
v M (A)		



Results

λ

SQL

IL

метод M() класса A

```

void Main()
{
    B b = new B();
    b. M();

    a = b;
    a. M();
}

```

B		
v <u>ToString</u>	Object	A B
v Equals		
v <u>GetHashCode</u>		
v. Finalize		
o M (B)		



Results

λ

SQL

IL

метод M() класса B
метод M() класса B


```

void Main()
{
    C c = new C();
    c. M();

    a = c;
    a. M();

    b = c;
    b. M();
}

```

C		
v <u>ToString</u>	Object	A, B, C
v Equals		
v <u>GetHashCode</u>		
v. Finalize		
o M (C)		



Results

λ

SQL

IL

метод M() класса C

метод M() класса C

метод M() класса C

```
public class A
{
    public new virtual void Function()
    {
        Console.WriteLine("Метод класса A");
    }
    public virtual void C()
    {
        Console.WriteLine("Метод C класса A");
    }
}
```

TypeDef #1 (02000002)

TypeDefName: Nasledovanie.A (02000002)

Flags : [Public] [AutoLayout] [Class] [AnsiClass] [BeforeFieldInit] (00100001)

Extends : 01000001 [TypeRef] System.Object

Method #1 (06000001)

MethodName: Function (06000001)

Flags : [Public] [Virtual] [HideBySig] [NewSlot] (000001c6)

RVA : 0x00002050

ImplFlags : [IL] [Managed] (00000000)

CallConvtn: [DEFAULT]

hasThis

ReturnType: Void

No arguments.

Method #2 (06000002)

MethodName: C (06000002)

Flags : [Public] [Virtual] [HideBySig] [NewSlot] (000001c6)

RVA : 0x0000205e

ImplFlags : [IL] [Managed] (00000000)

CallConvtn: [DEFAULT]

hasThis

ReturnType: Void

No arguments.

```
public class B : A
{
    public void Function()
    {
        Console.WriteLine("Метод класса B");
    }
    public override void C()
    {
        Console.WriteLine("Метод C класса B");
    }
}
```

TypeDef #2 (02000003)

TypeDefName: Nasledovanie.B (02000003)

Flags : [Public] [AutoLayout] [Class] [AnsiClass] [BeforeFieldInit] (00100001)

Extends : 02000002 [TypeDef] Nasledovanie.A

Method #1 (06000004)

MethodName: Function (06000004)

Flags : [Public] [HideBySig] [ReuseSlot] (00000086)

RVA : 0x00002074

ImplFlags : [IL] [Managed] (00000000)

CallConvntn: [DEFAULT]

hasThis

ReturnType: Void

No arguments.

Method #2 (06000005)

MethodName: C (06000005)

Flags : [Public] [Virtual] [HideBySig] [ReuseSlot] (000000c6)

RVA : 0x00002082

ImplFlags : [IL] [Managed] (00000000)

CallConvntn: [DEFAULT]

hasThis

ReturnType: Void

No arguments.

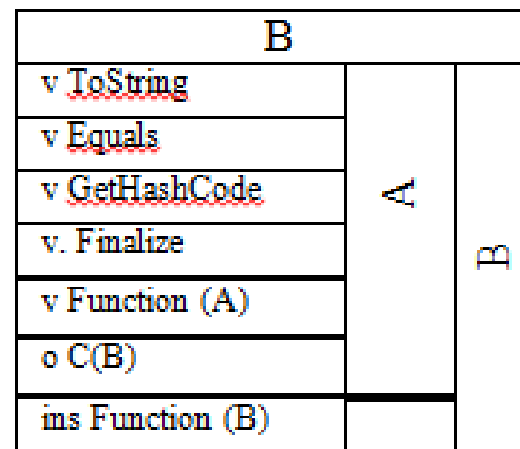
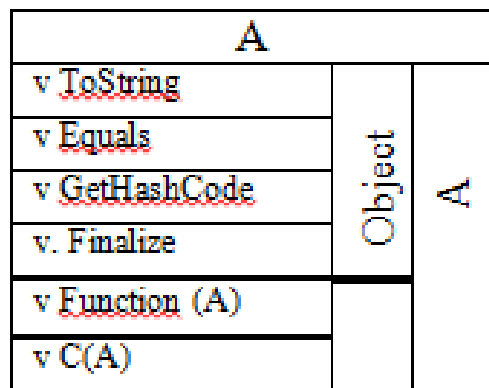
```

void Main(){
    A objectA = new B();
    objectA.Function();
    objectA.C();
}

public class A
{
    public new virtual void
    {
        Console.WriteLine("Метод класса A");
    }
    public virtual void C()
    {
        Console.WriteLine("Метод C класса A");
    }
}

public class B : A
{
    public void Function()
    {
        Console.WriteLine("Метод класса B");
    }
    public override void C()
    {
        Console.WriteLine("Метод C класса B");
    }
}

```



```

void Main(){

    A objectA = new B();
    objectA.Function();
    objectA.C();

}

public class A
{
    public new virtual void Function()
    {
        Console.WriteLine("Метод класса A");
    }
    public virtual void C()
    {
        Console.WriteLine("Метод C класса A");
    }
}

public class B : A
{
    public void Function()
    {
        Console.WriteLine("Метод класса B");
    }
    public override void C()
    {
        Console.WriteLine("Метод C класса B");
    }
}

```

Results

λ

SQL

IL

Метод класса A

Метод C класса B

```
public class A
{
    public virtual void Function()
    {
        Console.WriteLine("Метод класса A");
    }
}
```

```
public class B : A
{
    public virtual void Function()
    {
        Console.WriteLine("Метод класса B");
    }
}
```

```
public class C : B
{
    public override void Function()
    {
        Console.WriteLine("Метод класса C");
    }
}
```

```
void Main(){
    A objectA = new B();
    objectA.Function();

    objectA = new C();
    objectA.Function();

    B objectB = (B)objectA;
    objectB.Function();
}
```



```

public class A
{
    public virtual void Function()
    {
        Console.WriteLine("Метод класса A");
    }
}

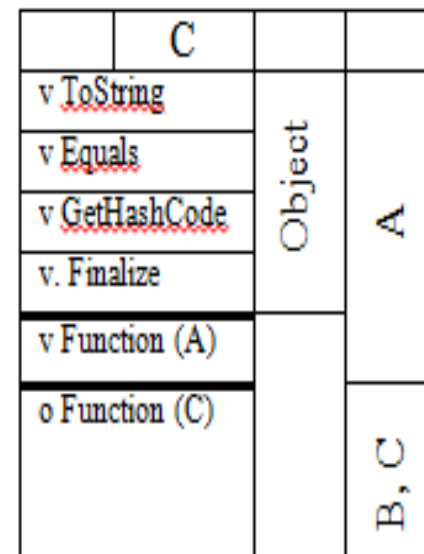
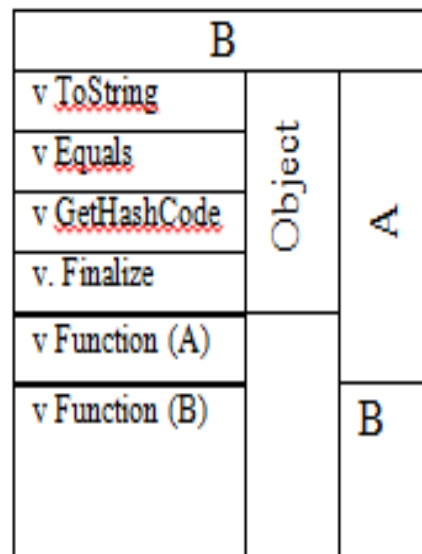
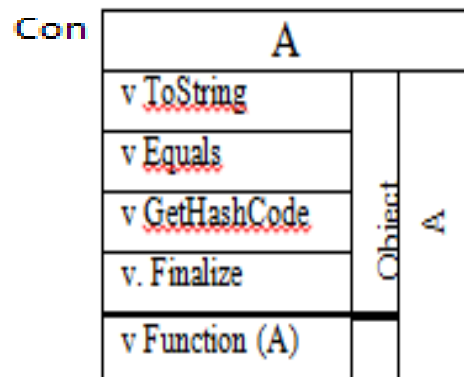
public class B : A
{
    public virtual void Function()
    {
        Console.WriteLine("Метод класса B");
    }
}

```

```

public class C : B
{
    public override void Function()
    {
    }
}

```



LINQPadQueries.Inheritance->polymorphism3.

```

void Main(){
    A objectA = new B();
    objectA.Function();

    objectA = new C();
    objectA.Function();

    B objectB = (B)objectA;
    objectB.Function();
}

```

Results λ SQL IL

Метод класса A
 Метод класса A
 Метод класса C

A		
v ToString	Object	A
v Equals		
v GetHashCode		
v. Finalize		
v Function (A)		

B		
v ToString	Object	A
v Equals		
v GetHashCode		
v. Finalize		
v Function (A)		B
v Function (B)		

	C		
v ToString	Object	A	
v Equals			
v GetHashCode			
v. Finalize			
v Function (A)			B, C
o Function (C)			

Переопределение и сокрытие методов

```
class A
{
    public virtual void M() { Console.Write("A"); }
}
class B: A
{
    public override void M() { Console.Write("B"); }
}
class C: B
{
    new public virtual void M() { Console.Write("C"); }
}
class D: C
{
    public override void M() { Console.Write("D"); }
}
static void Main()
{
    D d = new D(); C c = d; B b = c; A a = b;
    d.M(); c.M(); b.M(); a.M();
}
```



DDBB

```
class A
{
    public virtual void M() {
        Console.WriteLine("метод M() класса A");
    }
}
class B: A
{
    public override void M() {
        Console.WriteLine("метод M() класса B");
    }
}
class C: B
{
    new public virtual void M() {
        Console.WriteLine("метод M() класса C");
    }
}
class D: C
{
    public override void M() {
        Console.WriteLine("метод M() класса D");
    }
}
```

LINQPadQueries.Inheritance->polymorphism 4.

A	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
v M(A)	A

B	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B

C	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B
v M(C)	C

D	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B
v M(D)	C, D

LINQPadQueries.Inheritance->polymorphism 4.

```
void Main()
{
    A a ;

    B b = new B();
    a = b;      a. M();
}
```

A	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
v M(A)	A

B	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B

C	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B
v M(C)	C

D	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B
v M(D)	C, D

```
void Main()
{
    A a ;

    B b = new B();
    a = b;      a. M();
}
```

▼ Results λ SQL IL

метод M() класса B

A	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
v M(A)	A

B	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B

C	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B
v M(C)	C

D	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B
v M(D)	C, D

LINQPadQueries.Inheritance->polymorphism 4.

```
void Main()
{
    A a;
    B b;
    C c = new C();
    c.M();
    a = c;      a.M();
    b = c;      b.M();
}
```

A	
<u>v ToString</u>	Object
<u>v Equals</u>	
<u>v GetHashCode</u>	
<u>GetType</u>	
<u>v ToString</u>	
v M(A)	A

B	
<u>v ToString</u>	Object
<u>v Equals</u>	
<u>v GetHashCode</u>	
<u>GetType</u>	
<u>v ToString</u>	
o M(B)	A, B

C	
<u>v ToString</u>	Object
<u>v Equals</u>	
<u>v GetHashCode</u>	
<u>GetType</u>	
<u>v ToString</u>	
o M(B)	A, B
v M(C)	C

D	
<u>v ToString</u>	Object
<u>v Equals</u>	
<u>v GetHashCode</u>	
<u>GetType</u>	
<u>v ToString</u>	
o M(B)	A, B
v M(D)	C, D


```

void Main()
{
    A a;
    B b;
    C c = new C();
    c.M();
    a = c;      a.M();
    b = c;      b.M();
}

```

▼ Results λ SQL IL

метод M() класса C
метод M() класса B
метод M() класса B

A	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
v M(A)	A

B	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B

C	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B
v M(C)	C

D	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B
v M(D)	C, D

LINQPadQueries.Inheritance->polymorphism 4.

```
void Main()
{
    A a;
    B b;
    C c;
    D d = new D();
    d.M();
    a = d;      a. M();
    b = d;      b. M();
    c = d;      c. M();
}
```

A	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
v M(A)	A

B	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B

C	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B
v M(C)	C

D	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B
v M(D)	C, D

LINQPadQueries.Inheritance->polymorphism 4.

```
void Main()
{
    A a;
    B b;
    C c;
    D d = new D();
    d.M();
    a = d;      a. M();
    b = d;      b. M();
    c = d;      c. M();
}
```



Results

λ

SQL

IL

метод M() класса D
 метод M() класса B
 метод M() класса B
 метод M() класса D

A	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
v M(A)	A

B	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B

C	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B
v M(C)	C

D	
v ToString	Object
v Equals	
v GetHashCode	
GetType	
v ToString	
o M(B)	A, B
v M(D)	C, D

```
class Base
{
    public virtual void Foo(int i)
    {
        Console.WriteLine("Base.Foo(int)");
    }
}

class Derived : Base
{
    public override void Foo(int i)
    {
        Console.WriteLine("Derived.Foo(int)");
    }

    public void Foo(object o)
    {
        Console.WriteLine("Derived.Foo(object)");
    }
}

void Main()
{
    int i = 42;
    Derived d = new Derived();
    d.Foo(i);
}
```

```

class Base
{
    public virtual void Foo(int i)
    {
        Console.WriteLine("Base.Foo(int)");
    }
}

class Derived : Base
{
    public override void Foo(int i)
    {
        Console.WriteLine("Derived.Foo(int)");
    }

    public void Foo(object o)
    {
        Console.WriteLine("Derived.Foo(object)");
    }
}

void Main()
{
    int i = 42;
    Derived d = new Derived();
    d.Foo(i);
}

```

Results

λ

SQL

IL

Derived.Foo(object)

```

class Base
{
    public virtual void Foo(int i)
    {
        Console.WriteLine("Base.Foo(int)");
    }
}

```

```

class Derived : Base

```

```

{
    public override void Foo(int i)
    {
        Console.WriteLine("Derived.Foo(int)");
    }
}

```

```

    public void Foo(char o)
    {
        Console.WriteLine("Derived.Foo(object)");
    }
}

```

```

void Main()

```

```

{
    int i = 42;
    Derived d = new Derived();
    d.Foo(i);
}

```

Results

λ

SQL

IL

Derived.Foo(int)

```

class Base
{
    public Base()
    {
        F1();
    }

    public virtual void F1()
    {
        "Base.F1.virtual".Dump();
    }
}

```

```

class Derived : Base
{
    public Derived()
    {
        F1();
    }
    public override void F1()
    {
        "Derived.F1.virtual".Dump();
    }
}

```

```

void Main()
{
    Derived d = new Derived();
}

```

▼ Results λ SQL IL

Derived.F1.virtual
Derived.F1.virtual

Вызов методов базового класса

```
class Employee
{
    protected virtual void DoWork()
    {
        ...
    }
}
```

```
class Manager : Employee
{
    protected override void DoWork()
    {
        // Do processing specific to Managers
        ...
        // Call the DoWork method in the base class
        base.DoWork();
    }
    ...
}
```

Позволяет создавать собственную функциональность в дополнение к существующей, определяемой базовым классом

Производный класс с замещенным или переопределенным методом или свойством сохраняет доступ к методу или свойству базового класса с помощью ключевого слова **base**

Определение запечатанных классов и методов



```
sealed class Manager : Employee { . . . }
```

В запечатанном классе не могут объявляться виртуальные методы

```
class Manager : Employee  
{  
    ...  
    protected sealed override void DoWork()  
    {  
        ...  
    }  
}
```

Производный класс не может переопределить запечатанный метод

Можно запечатать только **override** методы, и следует объявлять их как **sealed override**

В .NET Framework все значимые типы (структуры и перечисления) неявно запечатаны

Полиморфизм

Ссылка на объект одного типа может быть инициализирована ссылкой на объект другого типа только пока он является типом, находящимся выше в иерархии наследования



Доступ к конкретным членам класса определяется типом переменной ссылки на объект, а не типом объекта, на который она ссылается



При вызове виртуальных методов по ссылке на базовый класс определяется именно тот вариант виртуального метода, который следует вызывать, исходя из типа объекта, к которому происходит обращение по ссылке, причем тип объекта определяется во время выполнения





Понятие полиморфизма имеет два аспекта (msdn)

- ✓ Во время выполнения объекты производного класса могут рассматриваться как объекты базового класса в коллекциях и в качестве параметров методов, при этом объявленный тип объекта больше не идентичен его типу времени выполнения
- ✓ Базовые классы могут определять и реализовывать виртуальные методы, а производные классы могут переопределять их, предоставляя свои собственные определение и реализацию. Во время выполнения метода среда CLR ищет тип времени выполнения объекта и вызывает это переопределение виртуального метода. Таким образом, в коде можно вызвать метод базового класса и вызвать выполнение метода с версией производного класса

Спасибо за внимание

БГУ, ММФ, кафедра веб-технологий и компьютерного моделирования

Автор: к. ф.-м. н., доцент, Кравчук Анжелика Ивановна

e-mail: anzhelika.kravchuk@gmail.com