# ЛЕКЦИЯ 8

- Модуль 2
- 22.11.2023
- Интерфейсы

# ЦЕЛИ ЛЕКЦИИ

- Познакомиться с типом интерфейса в языке C#
- Разобраться с особенностями реализации членов интерфейсов
- Изучить некоторые системные типы интерфейсов
- Получить представление об особенностях участия интерфейсов в иерархиях наследования

# АБСТРАКЦИЯ В C#

Одним из способов реализации концепции абстракции в языке C# являются абстрактные классы

- Как правило, абстрактные классы позволяют объявлять некоторую иерархию типов (формы, мебель и т. д.) с частично реализованным функционалом и некоторыми данными, а конкретные классы доопределяют необходимые абстрактные члены и расширяют функциональные возможности

Какие недостатки/проблемы использования абстрактных классов можно выделить?

# ОСОБЕННОСТИ ИСПОЛЬЗОВАНИЯ АБСТРАКТНЫХ КЛАССОВ

При использовании абстрактных классов возникает ряд особенностей, о которых приходится помнить:

- Привязка к определённой иерархии наследования и структуре типов
- Наследование допустимо только от одного класса
- Возможность предоставления общей реализации функционала наследникам путём добавления неабстрактных функциональных членов
- Допускается добавление некоторого состояния (данные), которое будет доступным для всех наследников

Возникает жёсткая привязка типов к определённой иерархии.

Это неудобно, если в программе есть идейно различные сущности, имеющие лишь небольшое сходство в поведении (например, возможность копирования или сортировки)

# ИНТЕРФЕЙСЫ В C#

- **Интерфейс** – это ссылочный тип, предоставляющий объявление функциональных членов, как правило, не имеющих реализации (C# 8)
- Синтаксис объявления интерфса:
  - `[Модификаторы] interface <Идентификатор> { [Объявление членов] }`
  - Для классов, которые определяют поведение, объявленное в интерфейсах принято говорить: «*Класс А реализует (не наследует!) интерфейс IImplementable*»

Хорошей практикой именования интерфейсов является добавление заглавной «I» в начало имени + использование прилагательного, описывающего возможность, предоставляемую интерфейсом (примеры: `IComparable`, `IBreakable`)

# ИНТЕРФЕЙС C#

- **Интерфейс представляет собой именованный набор сигнатур методов**
  - Рихтер Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#. 4-е изд. — СПб.: Питер, 2013. — 896 с

```
1.  interface IMyInterface {
2.   // список открытых методов и свойств,
3.   // событий или индексаторов
4. }
```

# ПОЛНАЯ СПЕЦИФИКАЦИЯ ОБЪЯВЛЕНИЯ ИНТЕРФЕЙСА

*attributes*$_{opt}$
*interface-modifiers*$_{opt}$
partial$_{opt}$
interface *identifier*
*type-parameter-list*$_{opt}$
*interface-base*$_{opt}$
*type-parameter-constraints-clauses*$_{opt}$
*interface-body*
;$_{opt}$

# ЧЛЕНЫ ИНТЕРФЕЙСА

- Все функциональные члены (поведение) могут быть членами интерфейса:
  - Методы
  - Свойства
  - Индексаторы
  - События

- Дополнительно (начиная с C# 8) можно добавлять:
  - Статические поля
  - Статические конструкторы

# ОСОБЕННОСТИ ИНТЕРФЕЙСОВ

- По умолчанию члены интерфейсов открытые и абстрактные
- Идейно предполагается, что они не имеют состояния (*stateless*) и определяют некоторый контракт, который выполняют реализующие интерфейс типы
- Один класс может реализовывать несколько интерфейсов
- Не могут объявлять нестатические данные (поля, автоматически реализуемые свойства, события)
- Не могут объявлять нестатические конструкторы и финализаторы (деструкторы)

Интерфейсы являются полезным инструментом для обеспечения гибкости кода в будущем при работе над проектами в долгосрочной перспективе

# ПРАВИЛА РЕАЛИЗАЦИИ ИНТЕРФЕЙСОВ

В случае комбинации наследования и реализации интерфейсов задаётся ряд правил:

- Тип-родитель должен быть указан первым после двоеточия
- Интерфейсы должны идти через запятую после родительского типа
- Абстрактные классы должны явно повторно объявлять метод интерфейса как abstract

```
using System;
record class Base(int value);

abstract record class Derived : Base, IComparable, IFormattable
{
    public Derived(int value) : base(value) { }
    // Реализация метода ToString IFormatable объявлена как абстрактная:
    public abstract string ToString(string format, IFormatProvider formatProvider);

    public int CompareTo(object obj) => value.CompareTo(((Derived)obj).value);
}
```

Базовый тип

реализуемые интерфейсы

# ЧТО НУЖНО ПОМНИТЬ ПРО ИНТЕРФЕЙСЫ?

- Интерфейс не содержит реализаций методов
- Класс, реализующий интерфейс, должен реализовывать все его члены
- Невозможно создать экземпляр интерфейса
- Класс или структура может реализовывать несколько интерфейсов (ограниченный вариант множественного наследования)

# АБСТРАКТНЫЕ КЛАССЫ VS. ИНТЕРФЕЙСЫ

Интерфейсы не являются полной заменой абстрактным классам. Эти типы данных реализуются в различных рабочих сценариях

- Использование интерфейсов может быть лучшим решением, когда необходимо только выполнение некоторого контракта по функционалу без привязки к типам

- Абстрактные классы могут быть более подходящим вариантом в сценариях, когда у сущностей уже должно быть определено некоторое общее базовое поведение и состояние

# ИНТЕРФЕЙСЫ И АБСТРАКТНЫЕ КЛАССЫ

## Абстрактные классы могут реализовывать интерфейсы

- Абстрактный класс должен представлять программный код реализации всех методов, объявленных в интерфейсе
- Абстрактный класс может также «реализовывать методы интерфейса как абстрактные»
  - Тогда реализации методов интерфейса предоставляет не абстрактный класс, а его производные классы

# ИНТЕРФЕЙСЫ VS. АБСТРАКТНЫЕ КЛАССЫ

| Интерфейс | Абстрактный класс |
|---|---|
| Поддерживает (ограниченно) множественное наследование | Не поддерживает множественное наследование |
| Не содержит членов с данными (полей) | Содержит члены с данными (поля) |
| Не содержит конструкторов | Содержит (может содержать) конструкторы |
| Содержит только сигнатуры членов (не полные реализации) | Содержит как сигнатуры членов (неполные реализации), так и полные реализации |
| По умолчанию интерфейс и его члены имеют открытый (public) уровень доступа | Поддерживает модификаторы доступа разного уровня |
| Члены интерфейса не могут быть статическими | Может содержать статические члены, но только полностью реализованные |

Интерфейсы vs. Классы (https://habr.com/ru/post/30444/)
Difference between Abstract Class and Interface in C# (http://www.differencebetween.net/technology/difference-between-abstract-class-and-interface-in-c/)

# ПРИМЕРЫ ИНТЕРФЕЙСОВ

## Системный тип IComparable

# ПРИМЕРЫ ИНТЕРФЕЙСОВ ИЗ SYSTEM

Метод `CompareTo()` интерфейса `IComparable` возвращает `int`

Возвращаемое значение используется для определения порядка сортировки по следующим правилам:

- Если результат больше нуля, данный объект условно «больше» другого
- Если результат равен нулю, данный объект условно «равен» другому
- Если результат меньше нуля, данный объект условно «меньше» другого

```
1.  public interface IComparable {
2.      int CompareTo(object obj);
3.  }
```

Иными словами, `CompareTo` позволяет задать отношение порядка для множества значений типа, реализующего `IComparable`

```
1.  public interface IEnumerable<out T> : IEnumerable {
2.      IEnumerator<T> GetEnumerator();
3.  }
```

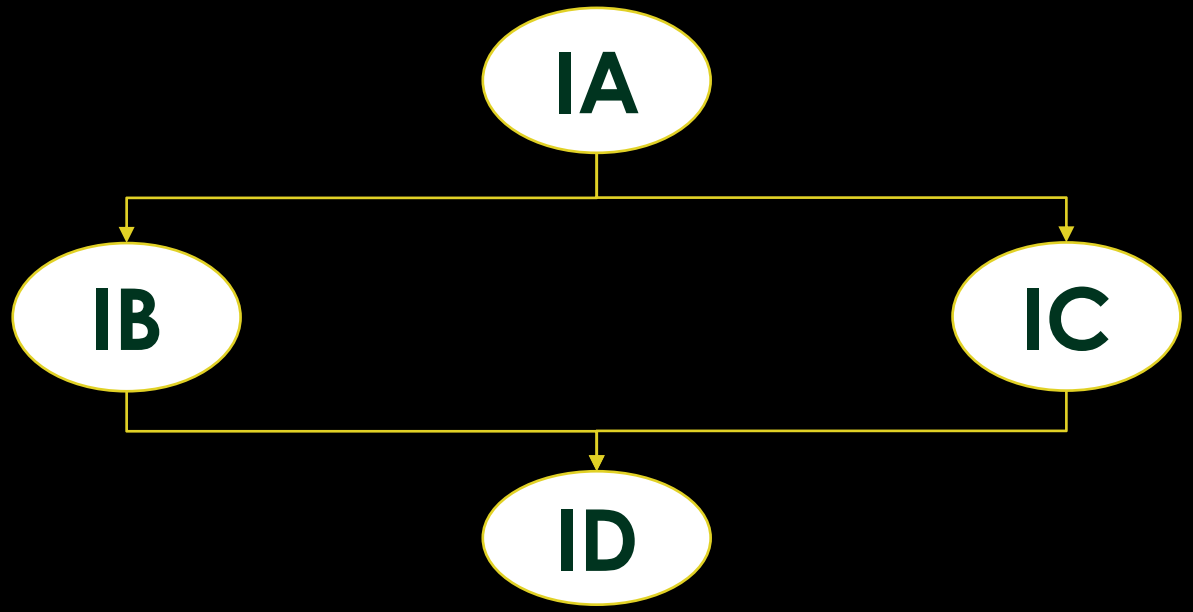
```
public class Person : IComparable<Person>
{
    public string Name { get; set; }
    public int Age { get; set; }

    public int CompareTo(Person other) => this.Age.CompareTo(other.Age);
}
```

```
Person person1 = new Person { Name = "John", Age = 25 };
Person person2 = new Person { Name = "Jane", Age = 30 };

int result = person1.CompareTo(person2);
if (result < 0)
{
    Console.WriteLine($"{person1.Name} is younger than {person2.Name}");
}
else if (result > 0)
{
    Console.WriteLine($"{person1.Name} is older than {person2.Name}");
}
else
{
    Console.WriteLine($"{person1.Name} and {person2.Name} are of the same age");
}
```

# ИНТЕРФЕЙСЫ И НАСЛЕДОВАНИЕ

# НАСЛЕДОВАНИЕ ИНТЕРФЕЙСОВ

- Интерфейсы в C# могут наследоваться только от других интерфейсов
- При этом возможна ситуация, когда иерархия интерфейсов в результате образует ациклический направленный граф, а не дерево




Подумайте, как может обрабатываться ситуация, когда интерфейсы **IB** и **IC** содержат объявления методов с полностью или частично совпадающими заголовками?

# ПРИМЕР 1: РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА

```
using System;
interface IPrintable { void PrintOut(string s); }

class PrintableElement : IPrintable {
    public void PrintOut(string s) => Console.WriteLine($"Called through {s}.");
}
class Program {
    static void Main() {
        PrintableElement element = new();
        // Вызов метода по ссылке типа объекта:
        element.PrintOut("PrintableElement");
        // Вызов метода по ссылке типа интерфейса:
        IPrintable printable = element;
        printable.PrintOut("IPrintable");
    }
}
```

**Вывод:**
Called through PrintableElement.
Called through IPrintable.

Реализация интерфейса

- Как и в случае для других ссылочных типов, ссылки типов интерфейсов хранят адрес объекта в куче
- Помните, что в связи с этим ссылка на типы значений будет приводить к упаковке

# ПРИМЕР 2: РЕАЛИЗАЦИЯ НЕСКОЛЬКИХ ИНТЕРФЕЙСОВ

```
Storage storage = new(10);
storage.Data = 5;
Console.WriteLine($"Value stored = {storage.Data}");

interface IDataProvider { int Data { get; } }
interface IDataStorage { int Data { set; } }

class Storage : IDataProvider, IDataStorage
{
    public int Data { get; set; }
    public Storage(int value) => Data = value;
}
```

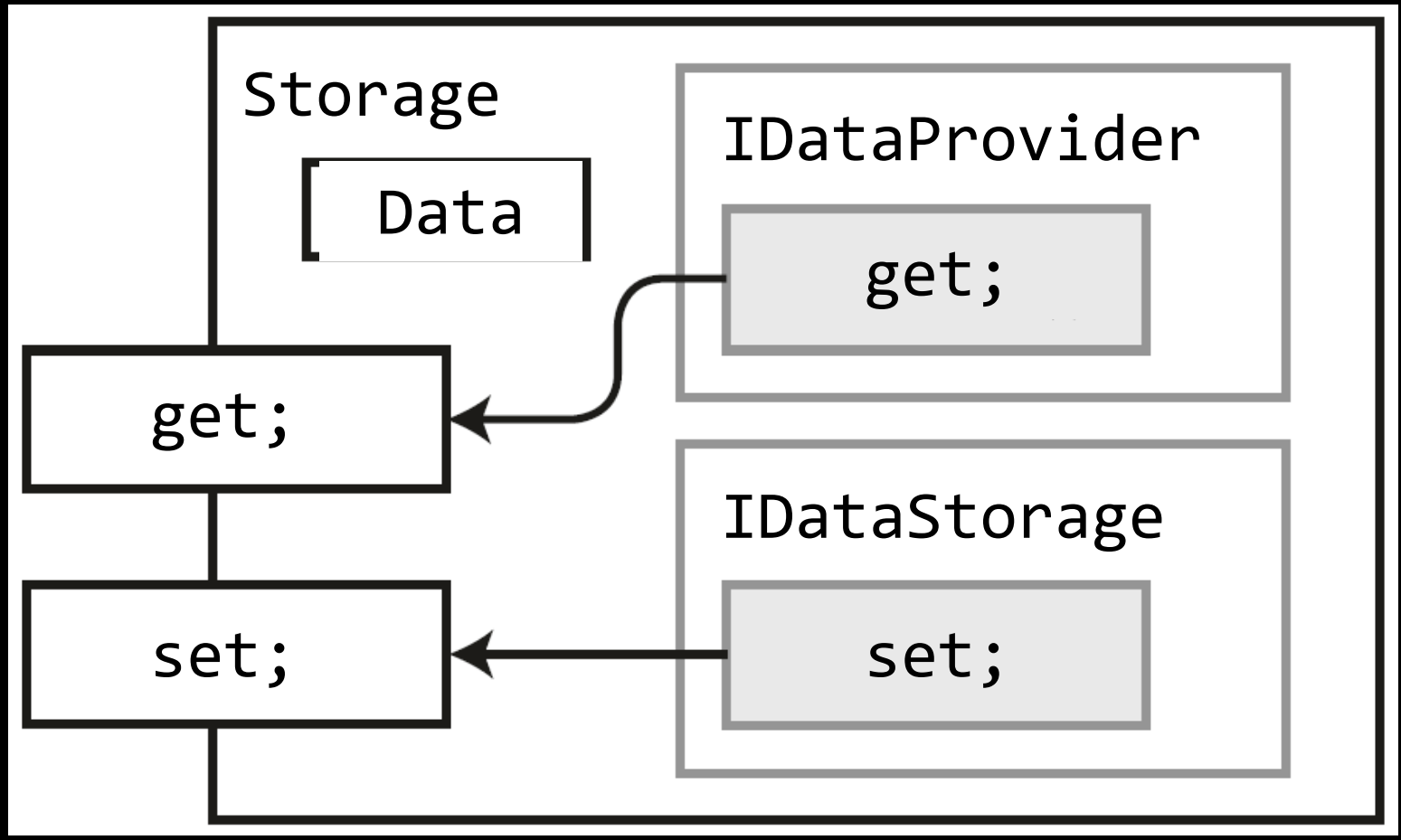
**Важно:** в данном случае компилятор не определяет автоматически реализуемое свойство! Предоставляются только методы доступа get/set без реализации

Автоматически реализуемое свойство выполняет одновременно требования обоих интерфейсов

**Вывод:**
Value stored = 5

# СХЕМА К ПРИМЕРУ 2

- Обратите внимание, что интерфейсы допускают наличие одноимённых членов.
- При этом достаточно, чтобы в реализующем их типе было соответствие определению



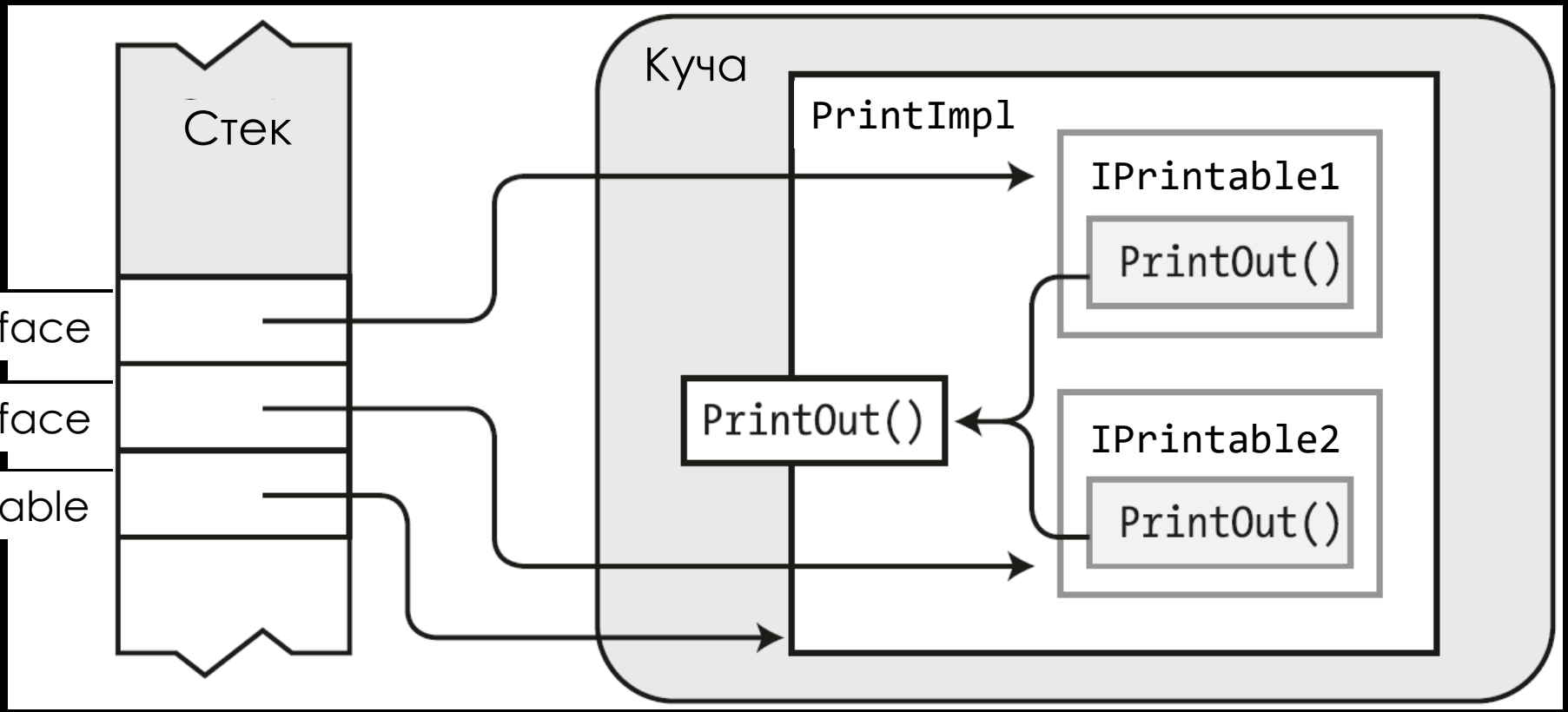
# ПРИМЕР 3: ИНТЕРФЕЙСЫ С СОВПАДАЮЩИМИ ЧЛЕНАМИ

```
PrintImpl printable = new();
printable.PrintOut("PrintableLine");
IPrintable2 interface2 = printable;
interface2.PrintOut("IPrintable2");
IPrintable1 interface1 = printable;
interface1.PrintOut("IPrintable1");

interface IPrintable1 { void PrintOut(string s); }
interface IPrintable2 { void PrintOut(string s); }

class PrintImpl : IPrintable1, IPrintable2
{
    public void PrintOut(string s) =>
                        System.Console.WriteLine($"Called through {s}.");
}
```

**Вывод:**
Called through: PrintableLine.
Called through: IPrintable2.
Called through: IPrintable1.

При реализации нескольких интерфейсов допускается наличие методов с полным совпадением заголовков. При этом, по умолчанию *реализация будет общей для обоих типов интерфейсов*.

# СХЕМА К ПРИМЕРУ 3



# ПРИМЕР 4: НАСЛЕДОВАНИЕ ИНТЕРФЕЙСОВ (1)

```
using System.Collections.Generic;

// Интерфейс для типов, являющихся целями взрывов.
public interface IExplosionTarget
{
    void ReceiveExplosion(IExplosive explosionSource);
}
// Интерфейс для типов, представляющих взрывающиеся снаряды.
public interface IExplosive
{
    void Explode(IEnumerable<IExplosionTarget> targets);
}

// Интерфейс для типов, представляющих мощно взрывающиеся снаряды.
public interface IPowerfulExplosive : IExplosive
{
    void ScalableExplode(IEnumerable<IExplosionTarget> targets, double strength);
}
```

Интерфейс `IEnumerable` в данном случае используется для поддержки обхода элементов коллекции в цикле `foreach`.

Интерфейс `IPowerfulExplosive` – частный случай `IExplosive`.

# ПРИМЕР 4: НАСЛЕДОВАНИЕ ИНТЕРФЕЙСОВ (2)

```
public class Torpedo : IPowerfulExplosive
{
    public void Explode(IEnumerable<IExplosionTarget> targets)
    {
        foreach (var target in targets) {
            target.ReceiveExplosion(this);
        }
    }
    public void ScalableExplode(IEnumerable<IExplosionTarget> targets, double strength)
    {
        foreach (var target in targets) {
            System.Console.WriteLine($"Locked on target: {target},\n"
                $"sending a torpedo with power: {strength}.");
            target.ReceiveExplosion(this);
        }
    }

    public override string ToString() => "Massive Torpedo";
}
```

Реализация метода интерфейса IExplosive.

Реализация метода интерфейса IPowerfulExplosive.

# ПРИМЕР 4: НАСЛЕДОВАНИЕ ИНТЕРФЕЙСОВ (3)

```
using System;
using System.Collections.Generic;

public class Tank : IExplosionTarget {
    public void ReceiveExplosion(IExplosive explosionSource)
        => Console.WriteLine($"{ToString()}: hit by: {explosionSource}");

    public override string ToString() => "Tank";
}
class Program {
    static void Main() {
        IExplosionTarget[] targets = new IExplosionTarget[5];
        for (int i = 0; i < targets.Length; ++i) {
            targets[i] = new Tank();
        }
        IPowerfulExplosive powerfulExplosive = new Torpedo();
        powerfulExplosive.ScalableExplode(targets, 42.0);
    }
}
```

Реализация метода интерфейса IExplosionTarget

**Вывод:**

```
Locked on target: Tank, sending a torpedo with power: 42.
Tank: hit by: Massive Torpedo
Locked on target: Tank, sending a torpedo with power: 42.
Tank: hit by: Massive Torpedo
Locked on target: Tank, sending a torpedo with power: 42.
Tank: hit by: Massive Torpedo
Locked on target: Tank, sending a torpedo with power: 42.
Tank: hit by: Massive Torpedo
Locked on target: Tank, sending a torpedo with power: 42.
Tank: hit by: Massive Torpedo
```

# ПРИМЕР 5: ПОЛУЧЕНИЕ РЕАЛИЗАЦИИ ОТ РОДИТЕЛЯ

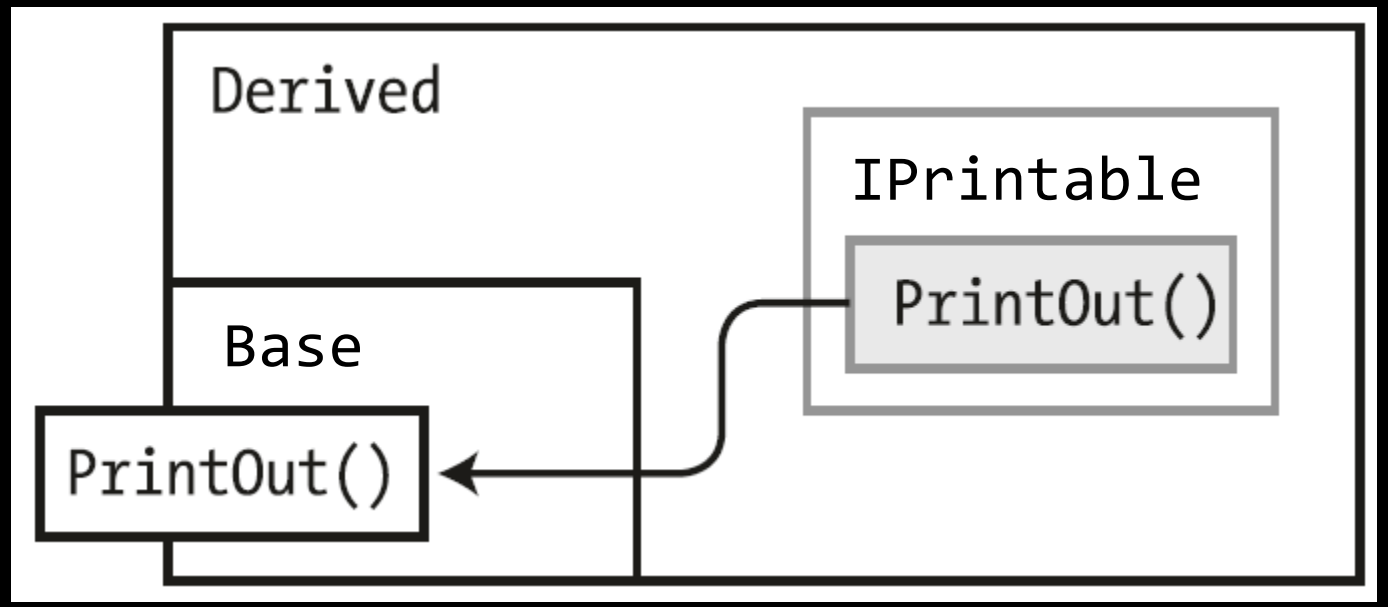
```
using System;
Derived derived = new();
derived.PrintOut("Derived");
(derived as IPrintable).PrintOut("IPrintable");

interface IPrintable { void PrintOut(string s); }
class Base
{
    public void PrintOut(string s) => Console.WriteLine($"Called through {s}.");
}
// IPrintable получает реализацию от Base.
class Derived : Base, IPrintable { }
```

Хотя наследник явно не определяет реализацию, метод с подходящим заголовком присутствует в родителе.

**Вывод:**

Called through: Derived.
Called through: IPrintable.

# СХЕМА К ПРИМЕРУ 5




Для реализации интерфейса не обязательно определять нужный метод именно в самом типе. Подойдёт так же доступная реализация в одном из типов-родителей

# ЯВНЫЕ РЕАЛИЗАЦИИ

# ЯВНАЯ РЕАЛИЗАЦИЯ ИНТЕРФЕЙСОВ

- Иногда при реализации интерфейсов может возникнуть сценарий, когда необходимо реализовать:
  - Два или более интерфейсов, содержащих методы с одинаковой сигнатурой, однако необходимо в зависимости от типа ссылки интерфейса выполнять различные действия
  - Интерфейсы, члены которых имеют одинаковые имена, но разные сигнатуры/назначение
- Для таких случаев C# поддерживает возможность явной реализации интерфейсов, доступной только по ссылке интерфейса соответствующего типа
  При этом не допускается указание модификатора доступа
- Для этого используется синтаксис:
  - `<Тип Возв. Знач.> <Тип Интерфейса>.<Член> <Идентификатор>() { … }`

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/explicit-interface-implementation

# ЯВНАЯ РЕАЛИЗАЦИЯ ЧЛЕНОВ ИНТЕРФЕЙСА

- Требуется при реализации двух и более интерфейсов, содержащих члены с одинаковой сигнатурой
- При явной реализации члена интерфейса должно присутствовать имя интерфейса, в котором был определен член, полностью соответствующий явной реализации интерфейса (совпадают полное имя члена, тип и список формальных параметров)

# ПРИМЕР: ЯВНАЯ РЕАЛИЗАЦИЯ ЧЛЕНОВ ИНТЕРФЕЙСА (1)

```
1. interface IControl { void Paint(); }
2. interface ISurface { void Paint(); }
3. class SampleClass : IControl, ISurface {
4.     public void Paint() {
5.         Console.WriteLine("Paint method in SampleClass");
6.     }
7. }
```

```
1. class Test {
2.     static void Main() {
3.         SampleClass sc = new SampleClass();
4.         IControl ctrl = sc;
5.         ISurface srfc = sc;

6.         sc.Paint();
7.         ctrl.Paint();
8.         srfc.Paint();
9.     }
10.}
```

https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/interfaces/explicit-interface-implementation

# ПРИМЕР: ЯВНАЯ РЕАЛИЗАЦИЯ ЧЛЕНОВ ИНТЕРФЕЙСА (2)

```
1.  interface IControl { void Paint(); }
2.  interface ISurface { void Paint(); }
3.  public class SampleClass : IControl, ISurface {
4.      void IControl.Paint() { System.Console.WriteLine("IControl.Paint"); }
5.      void ISurface.Paint() { System.Console.WriteLine("ISurface.Paint"); }
6.  }
```

```
1.      static void Main() {
2.          // Call the Paint methods from Main.
3.          SampleClass obj = new SampleClass();
4.          //obj.Paint();  // Compiler error.
5.          IControl c = obj;
6.          c.Paint();  // Calls IControl.Paint on SampleClass.
7.          ISurface s = obj;
8.          s.Paint(); // Calls ISurface.Paint on SampleClass.
9.      }
```

# СХЕМАТИЧЕСКИЙ ПРИМЕР ЯВНОЙ РЕАЛИЗАЦИИ (1)

```
1.  interface IA {
2.      void A();
3.  }
4.  interface IB {
5.      void B();
6.  }
7.  class C : IA, IB {
8.  // реализация метода a интерфейса IA
9.      void IA.A() {
10.         Console.Write("a");
11.     }
12. // реализация метода b интерфейса IB
13.     void IB.B() {
14.         Console.Write("b");
15.     }
16. }
```

```
1.  class Program {
2.      static void Main(string[] args) {
3.          C c = new C();
4.          c.A();
5.          c.B();
6.      }
7.  }
```

явные реализации интерфейсов доступны только по ссылкам интерфейсов соответствующих типов. Для доступа по ссылке типа можно дополнительно добавить неявную реализацию.

# СХЕМАТИЧЕСКИЙ ПРИМЕР ЯВНОЙ РЕАЛИЗАЦИИ (2)

```
1.  interface IA { void A(); }
2.  interface IB { void B(); }
3.  class Program : IA, IB {
4.      // реализация метода a интерфейса IA
5.      void IA.A() { Console.Write("a"); }
6.      // реализация метода b интерфейса IB
7.      void IB.B() { Console.Write("b"); }
8.      static void Main(string[] args) {
9.          Program obj = new Program();
10.         IA i1 = obj; // экземпляр интерфейса i1 используется для
11.                      // доступа к явно реализованному члену интерфейса IA
12.         i1.A();
13.         IB i2 = obj; // экземпляр интерфейса i2 используется для
14.                      // доступа к явно реализованному члену интерфейса IB
15.         i2.B();
16.     }
17. }
```

# ЯВНАЯ РЕАЛИЗАЦИЯ ИНТЕРФЕЙСОВ ПО УМОЛЧАНИЮ (C# 8)

- Начиная с C# 8.0, Вы можете объявлять реализации членов прямо внутри интерфейсов. Однако, такие реализации будут являться явными:

```
interface IAutoImplemented {
    void Method() => Console.WriteLine("IAutoImplemented.Method");
}

class C : IAutoImplemented { } // OK.

public static void Main() {
    IAutoImplemented i = new C();
    i.Method();
    // Попытка раскомментировать строчку ниже приведёт к ошибке:
    // new C().Method();
}
```

# ЗАЧЕМ НУЖНА РЕАЛИЗАЦИЯ ПО УМОЛЧАНИЮ

Одна из проблем, которую решили реализации интерфейсов по умолчанию – проблема расширения API интерфейса путём добавления новых членов

- До C# 8.0 это приводило к проблеме – добавление метода в интерфейс нарушало обратную совместимость пользователям библиотек, т. к. приходилось в обязательном порядке предоставлять реализацию новому(ым) члену(ам)

# ДОСТУП К ЯВНЫМ РЕАЛИЗАЦИЯМ ЧЛЕНОВ ИНТЕРФЕЙСОВ

```
interface IPrintData { void PrintOut(string s); }

class MyClass : IPrintData
{
    // Явная реализация интерфейса.
    void IPrintData.PrintOut(string s) => Console.WriteLine("IPrintData");
    public void Method1()
    {
        PrintOut("...");              //  1
        this.PrintOut("...");         //  2
        ((IPrintData)this).PrintOut("..."); //  3
        (IPrintData)this.PrintOut("..."); //  4
    }
}
```

Найдём ошибки

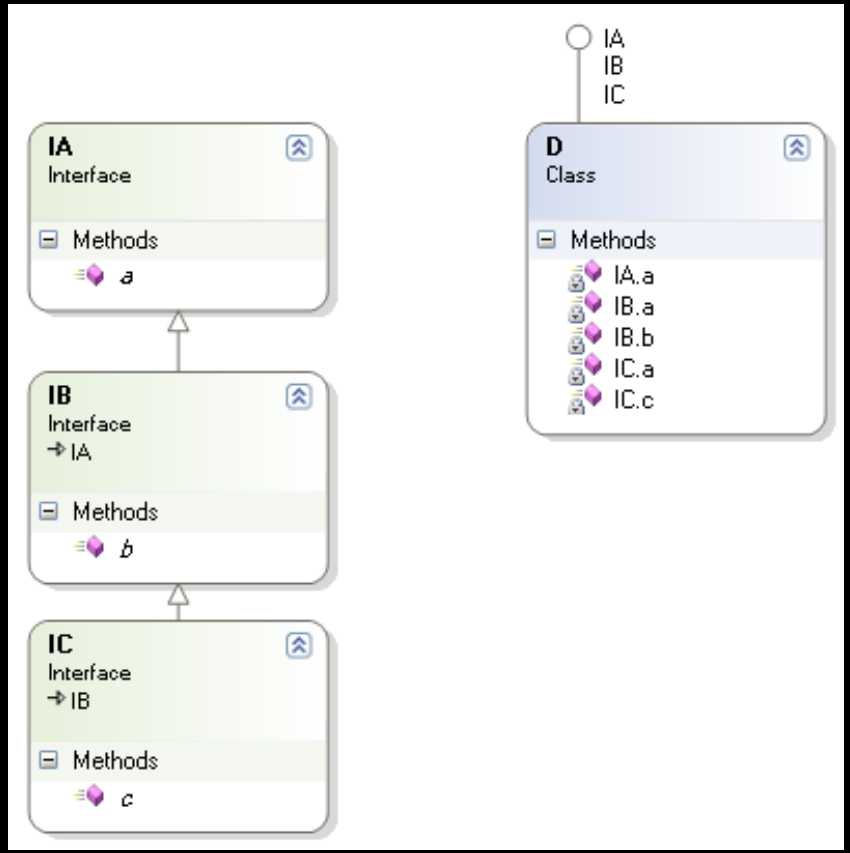
# ПОИСК РЕАЛИЗАЦИИ

- **Поиском реализации** интерфейса называют процесс определения места реализации членов интерфейса в классе или структуре
- При поиске реализации члена интерфейса явная реализация члена интерфейса имеет приоритет над другими членами класса или структуры

```
interface IA
{
    void A();
}
class B
{
    public void A() { /* реализация класса */  }
}
class C : B, IA
{
    public void A() { /* реализация метода */ }
}
```

# ПРИМЕР: ПОИСК РЕАЛИЗАЦИИ. РЕАЛИЗАЦИЯ БАЗОВОГО ИНТЕРФЕЙСА (1)

```
interface IA
{
    void A();
}
interface IB : IA
{
    void B();
}
interface IC : IB
{
    void C();
}
class D : IA, IB, IC
{
    void IA.A() { /* код реализации */ }
    void IB.A() { /* код реализации */ } // Ошибка!
    void IC.A() { /* код реализации */ } // Ошибка!

    void IB.B() { /* код реализации */ }
    void IC.C() { /* код реализации */ }
}
```



# ПРОЧИЕ ИЗМЕНЕНИЯ В C# 8

- Кроме реализаций функциональных членов по умолчанию:
  - Явное объявление модификаторов доступа функциональных членов
  - Переопределение явных реализаций по умолчанию при наследовании интерфейсов
  - Поддержка статических функциональных членов и данных, возможность добавления статического конструктора
  - Объявление вложенных типов
  - Объявление констант (модификатор const)
  - Объявление перегрузок операций

# ИНТЕРФЕЙСЫ И ПОЛИМОРФИЗМ

# ИНТЕРФЕЙСЫ И ПОЛИМОРФИЗМ

- Интерфейсы обеспечивают единообразный набор методов  и свойств объектам разных классов
  - Эти методы и свойства позволяют программам **полиморфно** обрабатывать объекты указанных разрозненных классов.

# ПРИМЕР: ИНТЕРФЕЙС И ПОЛИМОРФИЗМ (1)



```
/// <summary>
/// Интерфейс объявляет свойство для получения данных о возрасте.
/// </summary>
public interface IAge
{
    int Age { get; } // Получить значение возраста.
    string Name { get; } // Получить имя.
}
```

```
public class Person : IAge
{
    string _firstName;
    string _lastName;
    int _yearBorn;

    public Person(string firstNameValue, string lastNameValue, int yearBornValue)
    {
        (_firstName, _lastName) = (firstNameValue, lastNameValue);

        if (yearBornValue > 0 && yearBornValue <= DateTime.Now.Year)
            _yearBorn = yearBornValue;
        else
            _yearBorn = DateTime.Now.Year;
    }
    // Реализация свойством Age интерфейса IAge.
    public int Age { get => DateTime.Now.Year - _yearBorn;  }
    // Реализация свойством Name интерфейса IAge.
    public string Name { get => _firstName + " " + _lastName; }
}
```

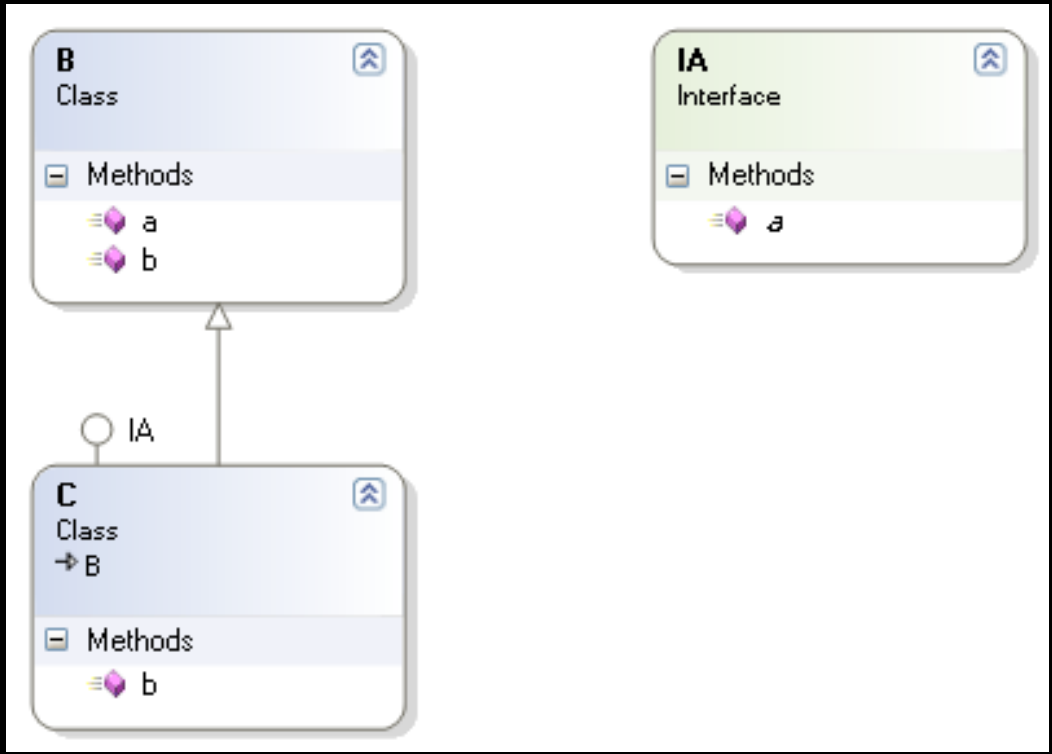
# ПРИМЕР: ИНТЕРФЕЙС И ПОЛИМОРФИЗМ (2)

```
public class Tree : IAge
{
    private int rings; // Число колец на стволе дерева.

    public Tree(int yearPlanted) =>  rings = DateTime.Now.Year - yearPlanted;

    // Приращение числа колец.
    public void AddRing() => rings++;

    // Реализация свойством Age интерфейса IAge.
    public int Age { get => rings; }
    // Реализация свойством Name интерфейса IAge.
    public string Name { get => "Tree"; }
} // конец класса Tree
```

# ПРИМЕР: ИНТЕРФЕЙС И ПОЛИМОРФИЗМ (4)

```
Tree tree = new Tree(1978);
Person person = new Person("Степан", "Степанов", 1971);
IAge[] iAgeArray = new IAge[2]; // создание массива ссылок IAge

iAgeArray[0] = tree; // iAgeArray[0] полиморфно ссылается на объект Tree
iAgeArray[1] = person; // iAgeArray[1] полиморфно ссылается на объект Person

// отображение информации о дереве
string output = $"{tree}: {tree.Name} {Environment.NewLine} возраст равен: " +
    $"{tree.Age}{Environment.NewLine}{Environment.NewLine}";
// отображение информации о человеке
output += $"{person}: {person.Name} {Environment.NewLine} возраст равен: " +
    $"{person.Age}{Environment.NewLine}{Environment.NewLine}";
// отображение имени и возраста для каждого объекта  в массиве iAgeArray
foreach (IAge ageReference in iAgeArray)
{
    output += $"{ageReference.Name}: возраст равен: " +
        $"{ageReference.Age}{Environment.NewLine}{Environment.NewLine}";
}
Console.WriteLine(output);
```

# ПРИМЕР: ПОИСК РЕАЛИЗАЦИИ В БАЗОВОМ КЛАССЕ (1)

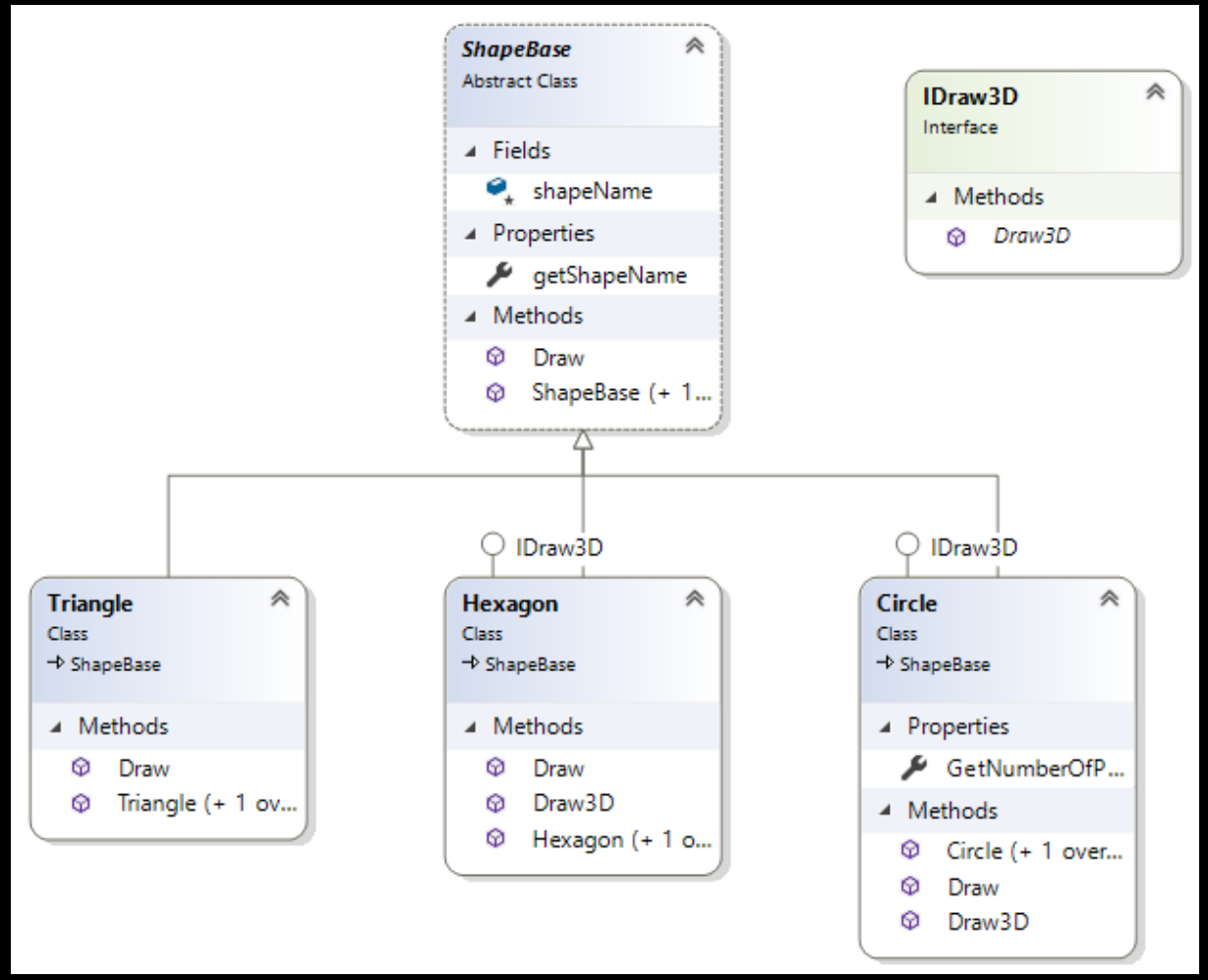



```
1.  interface IA {
2.      void a();
3.  }
4.  class B {
5.      public void a() { }
6.      // метод будет считаться реализацией
7.      // метода a() интерфейса IA
8.      public void b() { }
9.  }
10. class C : B, IA {
11.     new public void b() { }
12. }
```

При поиске реализации  интерфейса в текущем классе принимаются во внимание и члены базового класса

# ССЫЛКИ С ТИПОМ ИНТЕРФЕЙСА КАК ПАРАМЕТРЫ МЕТОДОВ

- Параметры с типом интерфейса могут передаваться в методы
- Тип интерфейса может быть возвращён из метода



# ПРИМЕР: ССЫЛКИ С ТИПОМ ИНТЕРФЕЙСА КАК ПАРАМЕТРЫ МЕТОДОВ (1)

```
public interface IDraw3D { string Draw3D(); }
```

```
public abstract class ShapeBase
{
    // Каждый объект "геометрическая фигура" будет иметь имя.
    protected string _shapeName;

    public ShapeBase(string s = "noname") { _shapeName = s; }

    public virtual string Draw() => "Shape.Draw()";
    public string ShapeName
    {
        get => _shapeName;
        set => _shapeName = value;
    }
}
```

# ПРИМЕР: ССЫЛКИ С ТИПОМ ИНТЕРФЕЙСА КАК ПАРАМЕТРЫ МЕТОДОВ (2)

```
public class Circle : ShapeBase, IDraw3D
{
    public Circle(string name = "") : base(name) { }

    public override string Draw() => $"Рисуем окружность {ShapeName}";

    public byte NumberOfPoints => 0;

    public string Draw3D() => $"Рисуем окружность {ShapeName} в 3D!";
}
```

```
public class Triangle : ShapeBase
{
    public Triangle(string name = "") : base(name) { }

    // метод рисует треугольник
    public override string Draw() => $"Рисуем окружность {ShapeName}";
}
```

# ПРИМЕР: ССЫЛКИ С ТИПОМ ИНТЕРФЕЙСА КАК ПАРАМЕТРЫ МЕТОДОВ (3)

```
public class Hexagon : ShapeBase, IDraw3D
{
    public Hexagon(string name = "") : base(name) { }

    // метод рисует шестиугольник
    public override string Draw() => $"Рисуем шестиугольник {ShapeName}";
    public string Draw3D() => $"Рисуем шестиугольник {ShapeName} в 3D!";
 }
```

# ПРИМЕР: ССЫЛКИ С ТИПОМ ИНТЕРФЕЙСА КАК ПАРАМЕТРЫ МЕТОДОВ (4)



```
class ShapeAppearence
{
    public static void DrawThisShapeIn3D(IDraw3D it3d) { it3d.Draw3D(); }
    static void Main(string[] args)
    {
        ShapeBase[] s = { new Hexagon(), new Circle(),
                new Triangle("Треугольник Паскаля"),
                new Circle("Кольцо Всевластья")};
        // Рисуем фигуры методом базового класса.
        Console.WriteLine($"{Environment.NewLine}-----\tРисование методом
базового класса\t-----{Environment.NewLine}");
```

```
for (int i = 0; i < s.Length; i++) { s[i].Draw(); }
        // Рисуем фигуры методом интерфейса.
        Console.WriteLine($"{Environment.NewLine}{Environment.NewLine}" +
            $"-----\tРисование методом интерфейса\t-----{Environment.NewLine}");
        for (int i = 0; i < s.Length; i++)
        {
            //Могу ли я нарисовать этот объект в трех измерениях?
            if (s[i] is IDraw3D) DrawThisShapeIn3D(s[i] as IDraw3D);
        }
        Console.WriteLine("");
    }
}
```

# ИНТЕРФЕЙСЫ В ИЕРАРХИЯХ НАСЛЕДОВАНИЯ

# НАСЛЕДОВАНИЕ РЕАЛИЗАЦИИ ИНТЕРФЕЙСА

- Производный класс наследует реализацию интерфейса из базового
- В производном классе нельзя изменить установленное соответствие между определением интерфейса и его реализацией из базового класса без явной повторной реализации этого же интерфейса

# ПРИМЕР. ИНТЕРФЕЙС И ЕГО РЕАЛИЗАЦИЯ В БАЗОВОМ КЛАССЕ

```
interface IA { void A(); }
class B : IA
{
    public void A()
    { // Реализация A() из IA.
        Console.Write("1");
    }
}
class C : B
{
    // Введение нового  A().
    new public void A()
    {
        Console.Write("2");
    }
}
```

```
 B objB = new B();
    C objC = new C();
    IA i1 = objB;
    IA i2 = objC;
    objB.A(); // Вызов реализации метода a из IA
    objC.A(); // Вызов нового метода a из C
    i1.A(); // Вызов реализации метода a из IA
    i2.A(); // Вызов реализации метода a из IA
}
```

# ЗАМЕЩЕНИЕ ВИРТУАЛЬНОГО МЕТОДА В БАЗОВОМ КЛАССЕ, РЕАЛИЗУЮЩИМ ИНТЕРФЕЙС

- Если метод интерфейса реализован как виртуальный в базовом классе, то в производном классе возможно замещение этого виртуального метода
- Явные реализации членов интерфейса не могут быть объявлены как виртуальные
  - Замещение явно реализованных членов интерфейса невозможно

```
interface IA { void A(); }
class B : IA {
   public virtual void A() => Console.Write("1");
}
class C : B {
   public override void A() => Console.Write("2");
}
```

```
B objB = new B();
C objC = new C();
IA i1 = objB;
IA i2 = objC;
objB.A(); // вызов реализации A() из IA
objC.A(); // вызов нового A() из C
i1.A(); // вызов реализации A() из IA
i2.A(); // вызов нового A() из C
```

# ПОВТОРНАЯ РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА

- **Допустима повторная реализация интерфейса в классе, наследующем его реализацию из базовых классов**
  - Для этого в список реализуемых интерфейсов явно добавляется имя интерфейса
- **Как выполняется поиск реализации?**

# ПРИМЕР: ПРОВЕРКА РЕАЛИЗОВАННОСТИ ИНТЕРФЕЙСА, ИСПОЛЬЗОВАНИЕ **IS** (1)

```
public class Book
{
    public Book() { }
    protected string _bookName = "";
    public override string ToString() => _bookName;
}

interface IBilling { bool CalculateDiscount(); }

public class MyBook : Book, IBilling
{
    public MyBook() { }
    public bool CalculateDiscount()
    {
        // Проверка наличия скидки.
        return true;
    }
}
```

Интерфейс

Реализация

# ПРИМЕР: ПРОВЕРКА РЕАЛИЗОВАННОСТИ ИНТЕРФЕЙСА, ИСПОЛЬЗОВАНИЕ **IS** (2)

```
public class Program
{
    public static void Main()
    {
        MyBook mb = new MyBook();

        if (mb is IBilling)
        {
            // Реализуется ли интерфейс MyBook?
            // Интерфейс реализуется - создаем объект
            IBilling bill = mb;
            bool status = mb.CalculateDiscount();
        }
        else { Console.WriteLine("Интерфейс не поддерживается"); }
    }
}
```

Осуществляем проверку по типу интерфейса

# ПОЛУЧЕНИЕ ССЫЛКИ НА ИНТЕРФЕЙС, ИСПОЛЬЗОВАНИЕ **AS**

```
public interface IPointy { byte NumberOfPoints { get; } }

public class Hexagon : IPointy
{
    string _name;
    public Hexagon(string name = "") { this._name = name; }
    public byte NumberOfPoints { get => 6; }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Hexagon hex = new Hexagon("Шестиугольник");
        IPointy iPointyEx;
        iPointyEx = hex as IPointy; // Это приведение тут лишнее.
        Console.WriteLine(iPointyEx is null? "Интерфейс IPointy не реализуется!" :
                iPointyEx.NumberOfPoints);
    }
}
```

# КОНТРОЛИРУЕМОЕ УПРАВЛЕНИЕ РЕСУРСОВ

## IDisposable

# ИНТЕРФЕЙС IDISPOSABLE

- Интерфейс `IDisposable` и подход к его реализации предназначены для обеспечения возможности контролируемого освобождения ресурсов без необходимости ожидания момента сборки мусора
- Это бывает полезно, когда некоторый тип захватывает какие-либо внешние ресурсы

```
// Для освобождения неуправляемых ресурсов.
[ComVisible(true)]
public interface IDisposable
{
// Выполняем задачи по освобождению ресурсов.
    void Dispose();
}
```

# РЕАЛИЗАЦИЯ IDISPOSABLE – ПРОСТОЙ СЛУЧАЙ

- Данная реализация предназначена для опечатанных классов без неуправляемых ресурсов:

```
public sealed class SealedClass : IDisposable
{
    public void Dispose()
    {
        // Освободить управляемые ресурсы, т.е.
        // вызвать Dispose() на всех членах.
    }
}
```

# РЕАЛИЗАЦИЯ IDISPOSABLE – ОБЩИЙ СЛУЧАЙ (1)

```
class BaseClass : IDisposable
{
    // Флаг для проверки: был ли уже вызван Dispose()?
    bool disposed = false;

    // Общедоступная реализация Dispose(),
    // вызываемая пользовательским кодом:
    public void Dispose()
    {
        // См. реализацию на следующем слайде.
        Dispose(true);
        // не вызывать финализатор сборщику мусора:
        GC.SuppressFinalize(this);
    }

    // Продолжение на следующем слайде…
}
```

# РЕАЛИЗАЦИЯ IDISPOSABLE – ОБЩИЙ СЛУЧАЙ (2)

```
// Защищённая реализация Dispose(bool), доступная для переопределения:
protected virtual void Dispose(bool disposing)
{
    if (disposed)
        return;
    if (disposing) {
        // Освободить управляемые ресурсы…
    }
    // Освободить неуправляемые ресурсы…
    disposed = true; // Установить флаг, что очистка ресурсов выполнена.
}

// Финализатор имеет смысл только при использовании
// неуправляемых ресурсов непосредственно в BaseClass:
~BaseClass() {
    Dispose(false); // По сути ~BaseClass – и есть Finalize().
}
```

# ИСПОЛЬЗОВАННАЯ ЛИТЕРАТУРА


- Обзорная информация по интерфейсам:
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface
https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/interfaces
- Сравнение абстрактных классов и интерфейсов:
https://stackoverflow.com/questions/761194/interface-vs-abstract-class-general-oo
- Явная реализация интерфейсов:
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/interfaces/explicit-interface-implementation
- Реализация интерфейсов по умолчанию в C# 8.0, обсуждение мотивации:
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-8.0/default-interface-methods
- https://stackoverflow.com/questions/62832992/when-should-we-use-default-interface-method-in-c
- О traits:
https://en.wikipedia.org/wiki/Trait_(computer_programming)
- http://scg.unibe.ch/archive/papers/Scha03aTraits.pdf
- Интерфейс IComparable, принцип его реализации: https://docs.microsoft.com/en-us/dotnet/api/system.icomparable
- Интерфейс IDisposable, принцип его реализации:
- https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/implementing-dispose
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-8.0/default-interface-methods
- Обсуждение на StackOverflow:
https://stackoverflow.com/questions/62832992/when-should-we-use-default-interface-method-in-c
- https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface
- Обсуждение вопроса в деталях: https://stackoverflow.com/questions/761194/interface-vs-abstract-class-general-oo
- https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/implementing-dispose