В.В. Подбельский

Использованы иллюстрации пособия Daniel Solis, Illustrated C#

Иллюстрации к курсу лекций по дисциплине «Программирование на C#»

Модуль 3. Лекция 3а

Интерфейсы

Благодарность Д. Д. Сагалову за помощь в переработке презентаций

Абстракция в C#

На предыдущих лекциях уже обсуждалось такое понятие, как **абстракция** – подход к описанию кода через некоторый контракт, который должны выполнять используемые в определённом контексте типы.

Одним из способов реализации абстракции в языке являются изученные ранее **абстрактные классы**.

Как правило, абстрактные классы позволяли объявлять некоторую категорию типов (формы, мебель и т. д.) *с частично реализованным функционалом* и *некоторыми данными*, а конкретные классы уже доопределяли необходимые абстрактные члены и расширяли функциональные возможности при необходимости.

Подумайте, 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.

На заметку: в итоге, возникает жёсткая привязка типов к определённой иерархии. Такой подход может оказаться неудобным, особенно когда 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).

Для объявления интерфейсов используется синтаксис:

```
[Модификаторы] interface <Идентификатор> { [Объявление членов] }
```

Фактически, члены интерфейса являются неявно *открытыми* и *абстрактными*.

Для классов, которые определяют поведение, объявленное в интерфейсах принято говорить: «Класс A **реализует** (не наследует!) интерфейс IImplementable.»

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

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface

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:

- Методы;
- Свойства;
- Индексаторы;
- События.

Дополнительно (начиная с C# 8) можно добавлять:

- Статические поля;
- Статические конструкторы.

Особенности Интерфейсов

В отличие от классов, для интерфейсов имеется ряд особенностей:

- 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 типы 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);
}
```

```
родитель      реализуемые интерфейсы
↓             ↓              ↓
```

Абстрактные Классы vs. Интерфейсы

Важно понимать, что интерфейсы не являются полной заменой абстрактным классам. Обе конструкции имеют право на существование в различных 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.

Обсуждение вопроса в деталях: https://stackoverflow.com/questions/761194/interface-vs-abstract-class-general-oo

Пример Объявления Интерфейса: IComparable

ключевое слово ↓ идентификатор ↓

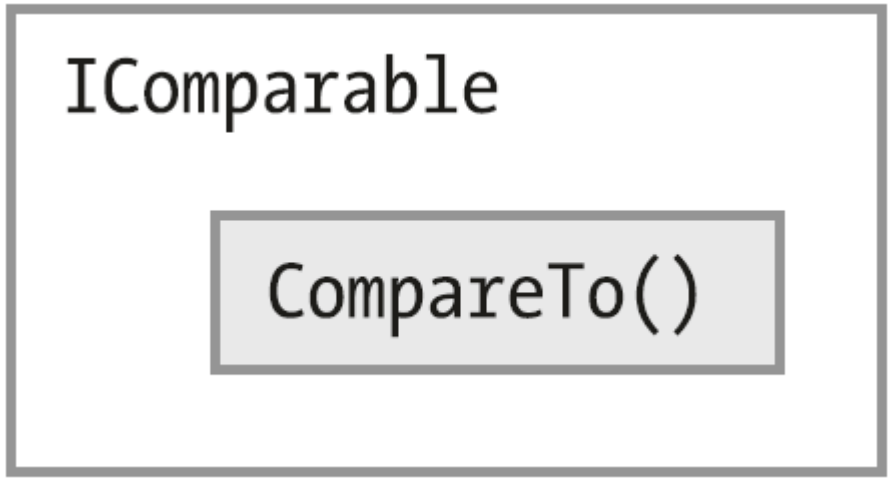
```
public interface IComparable
{
```

опционально ↓

```
    public int CompareTo(object obj);
}
```

Реализация отсутствует, аналогично определению абстрактных методов.

Представление интерфейса:



Документация по библиотечному IComparable: https://docs.microsoft.com/en-us/dotnet/api/system.icomparable

Принцип Реализации Интерфейса IComparable

Заметьте, что метод **CompareTo** интерфейса IComparable возвращает int.

Фактически, это 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.

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

Обсуждение вопроса в деталях: https://stackoverflow.com/questions/761194/interface-vs-abstract-class-general-oo

Пример: Интерфейс IComparable. Часть 1

Интерфейс IComparable используется типами BCL в сценариях, *где нужна сортировка*. Так, попытка сортировки массива объектов, не реализующих IComparable будет приводить к **исключениям** в методах сортировки:

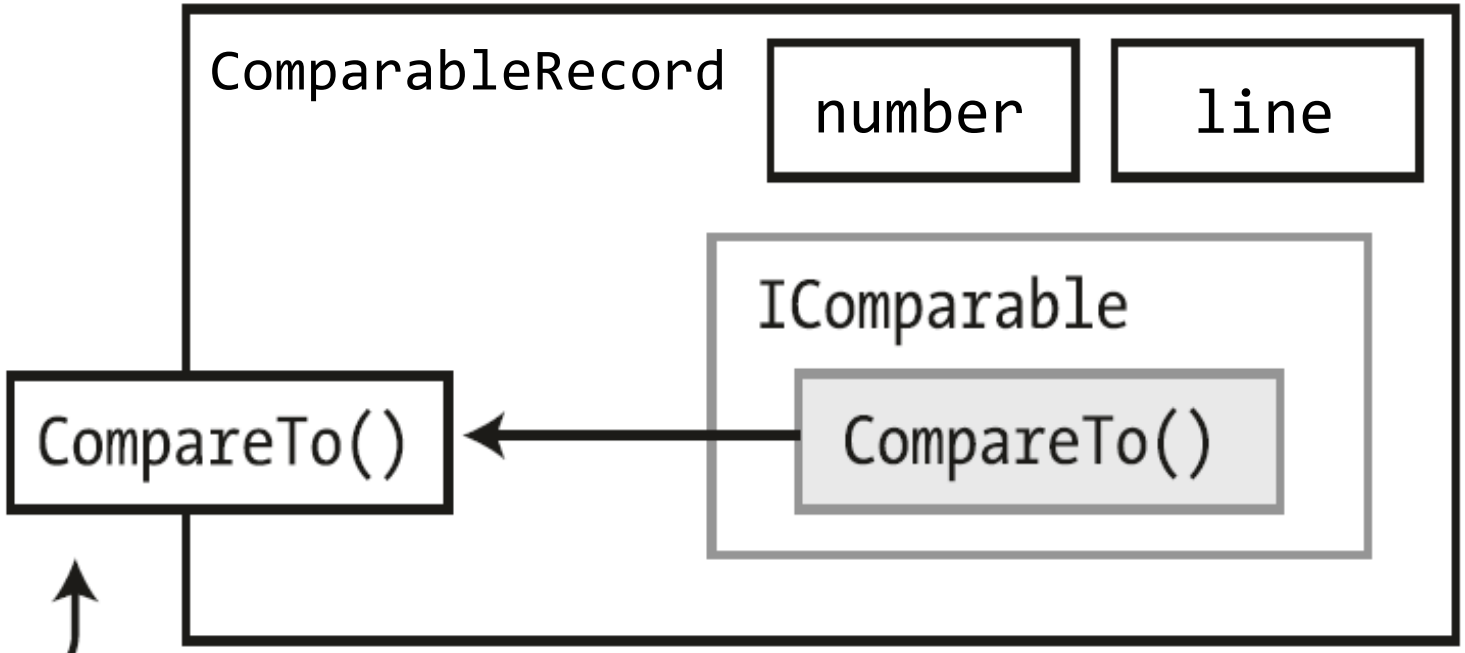
```
using System.Collections.Generic;

// Простая ссылочная запись без реализации IComparable:
public record class NonComparableRecord(int number, string line);


List<NonComparableRecord> nonComparables = new();
nonComparables.Add(new(1, "line1"));
nonComparables.Add(new(2, "line2"));
// При сортировке возникнет InvalidOperationException.
nonComparables.Sort();
```

Пример: Интерфейс IComparable. Часть 2

Реализация IComparable исправляет ситуацию – используется CompareTo():

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

List<ComparableRecord> nonComparables = new();
nonComparables.Add(new(1, "line1"));
nonComparables.Add(new(2, "line2"));
nonComparables.Sort();

public record class ComparableRecord(int number, string line) : IComparable {
    public int CompareTo(object obj) {
        if (obj != null && obj is ComparableRecord cmp) {
            return number.CompareTo(cmp.number);
        }
        // null и объекты других типов всегда "меньше".
        return -1;
    }
}
```

Схема Реализации IComparable в Примере

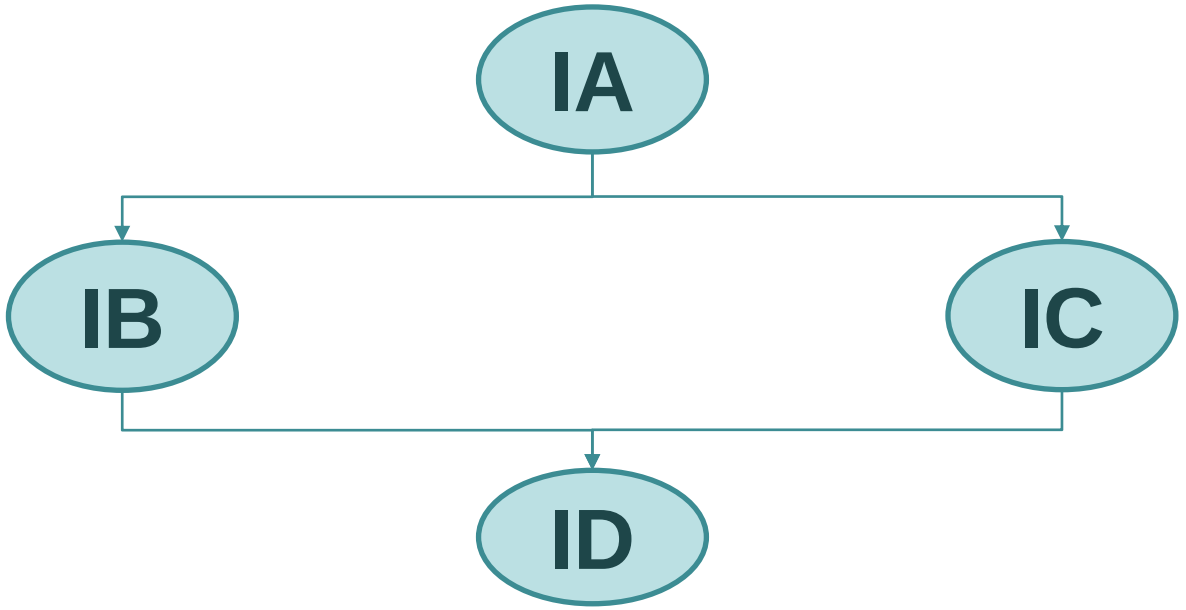


Метод реализуется записью.

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?*

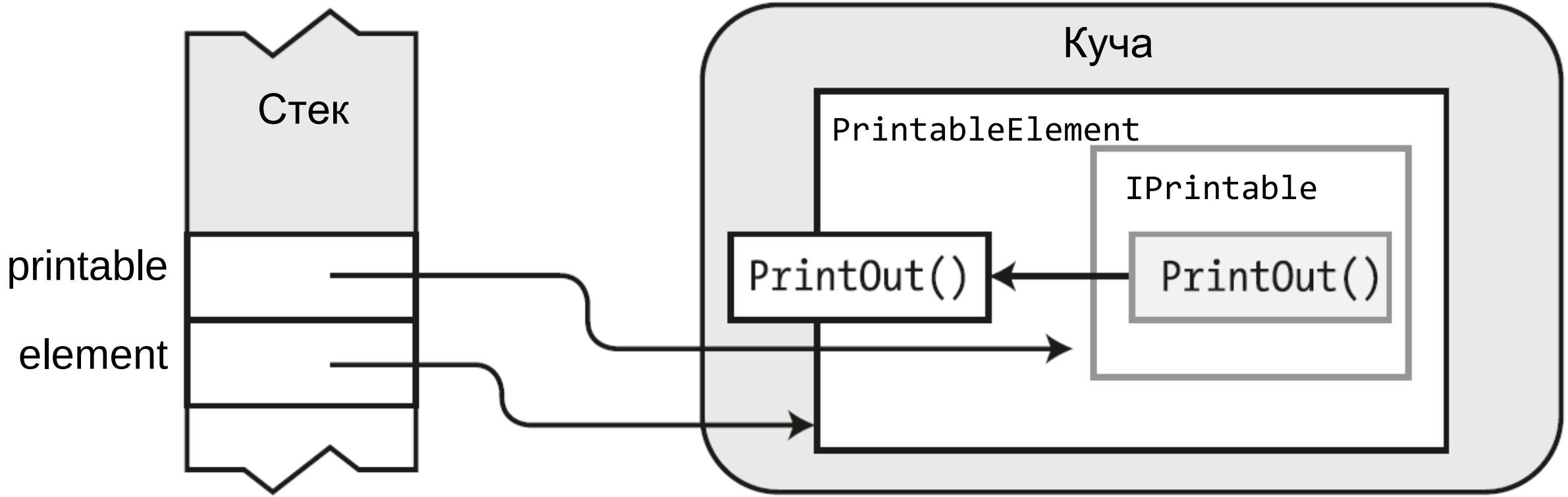
Пример 1: Реализация Интерфейса

```
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");
    }
}
```

Реализация интерфейса.

Вывод:
Called through PrintableElement.
Called through IPrintable.

Схема к Примеру 1



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 2.1: Реализация Нескольких Интерфейсов

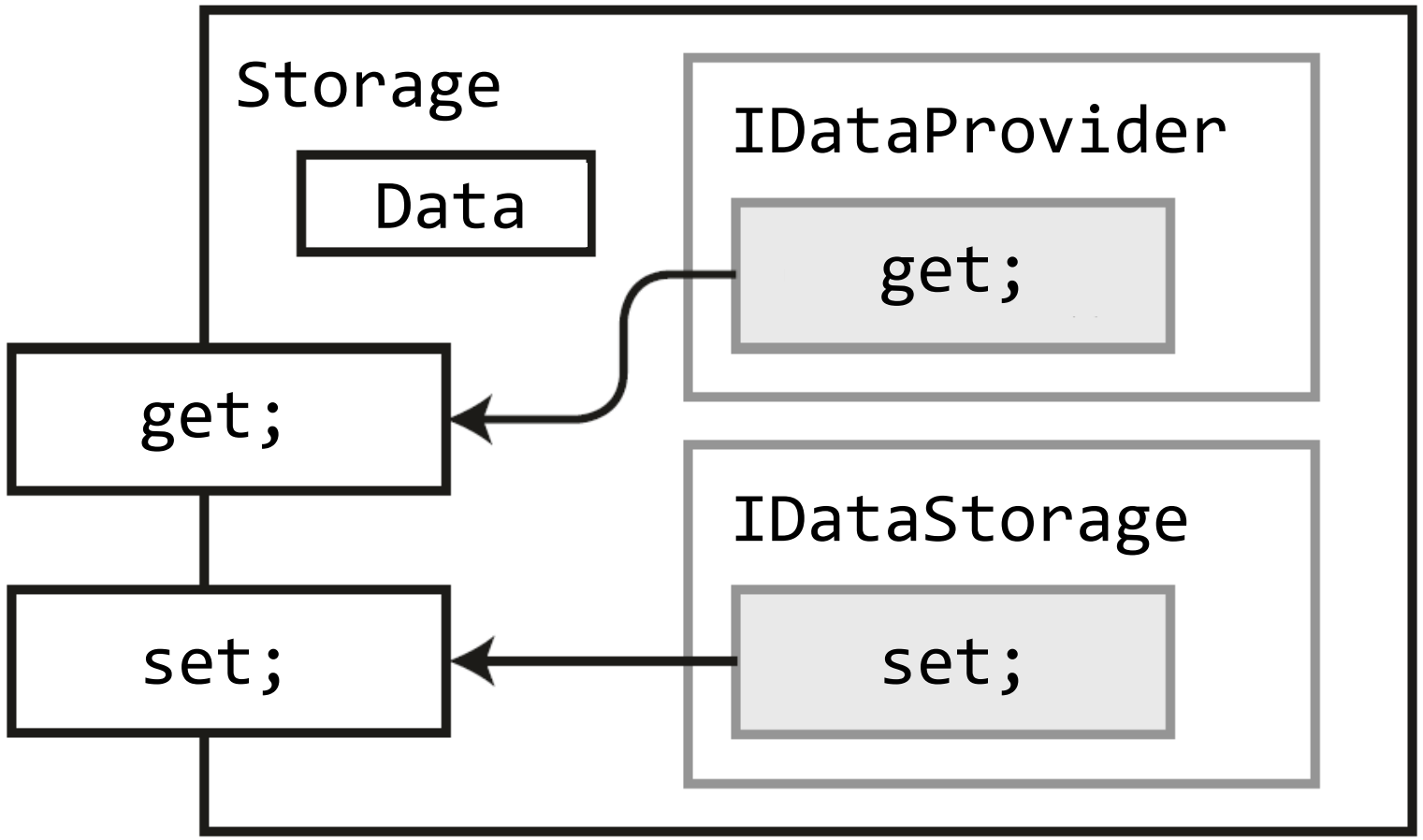
```
using System;

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 2.1



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

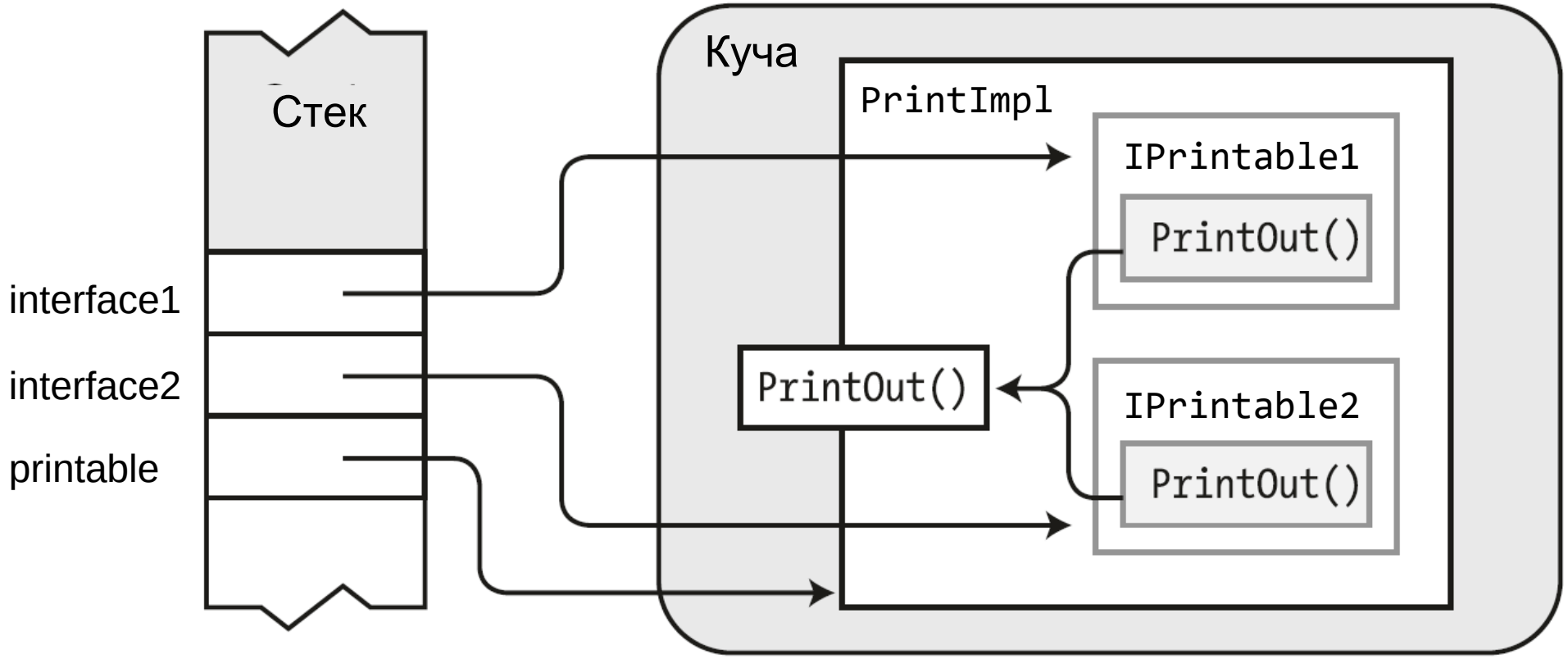
Пример 2.2: Интерфейсы с Совпадающими Членами

```
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.

Важно: 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 2.2



При наличии функциональных членов с полностью идентичными заголовками по умолчанию будет использоваться общая реализация для обоих интерфейсов.

Пример 3: Наследование Интерфейсов. Часть 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 3: Наследование Интерфейсов. Часть 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 3: Наследование Интерфейсов. Часть 3

Реализация метода интерфейса IExplosionTarget.

```
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);
    }
}
```

Пример 3: Вывод Программы

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
```

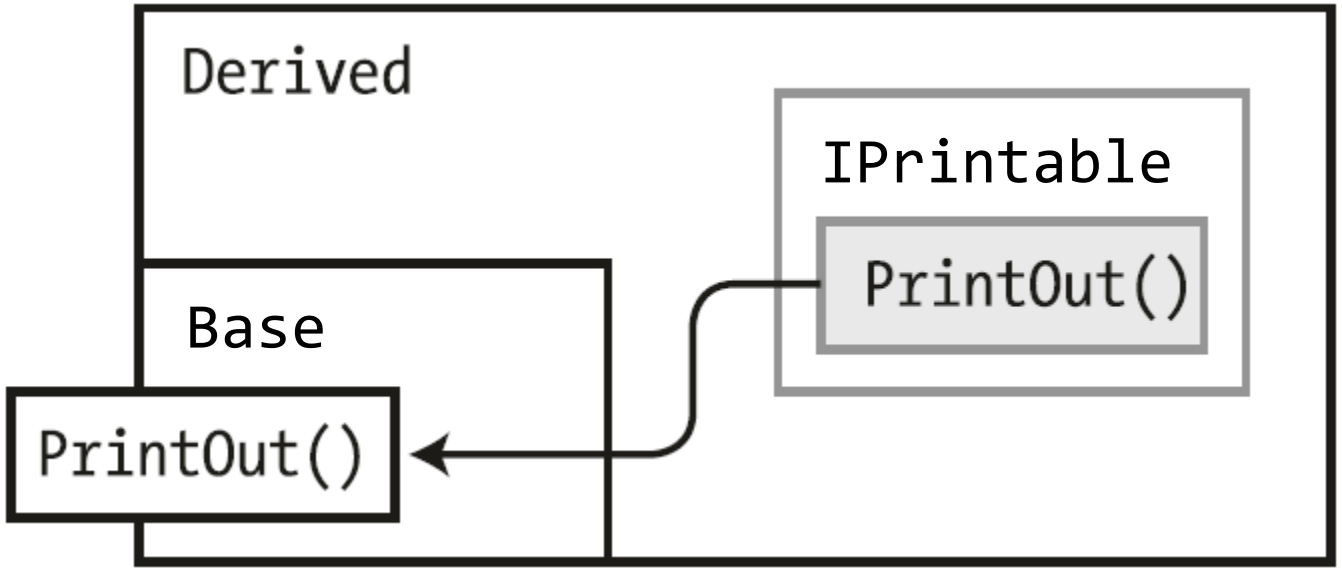
Пример 4: Получение Реализации от Родителя

```
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 4



Обратите внимание, что для реализации интерфейса не обязательно определять нужный метод именно в самом типе. Подойдёт так же доступная реализация в одном из типов-родителей.

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*.

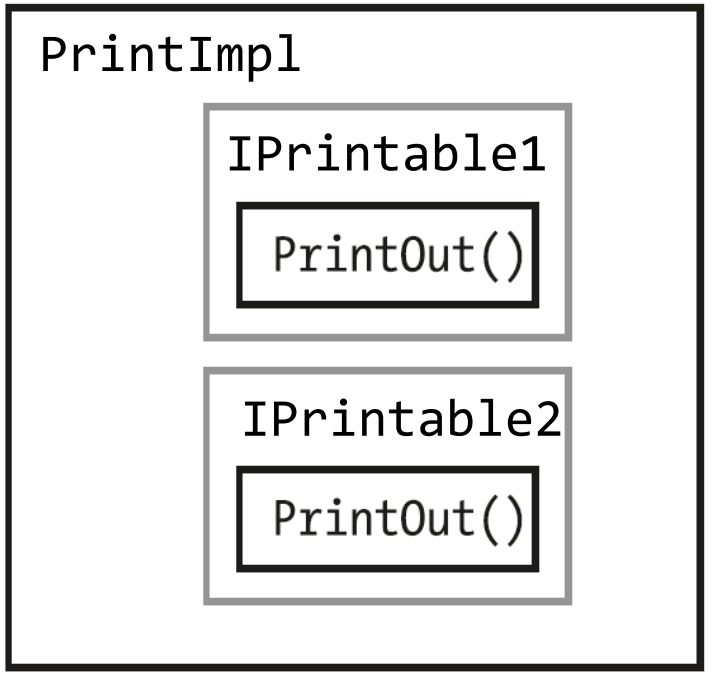
Для этого используется синтаксис:

```
<Тип Возв. Знач.> <Тип Интерфейса>.<Член> <Идентификатор>() { … }
```

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

Пример Явной Реализации Интерфейсов

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


// IPrintable получает реализацию от Base.
class PrintImpl : IPrintable1, IPrintable2
{
    void IPrintable1.PrintOut(string s)
        => Console.WriteLine($"IPrintable1 explicit: {s}");

    void IPrintable2.PrintOut(string s)
        => Console.WriteLine($"IPrintable2 explicit: {s}");
}
```



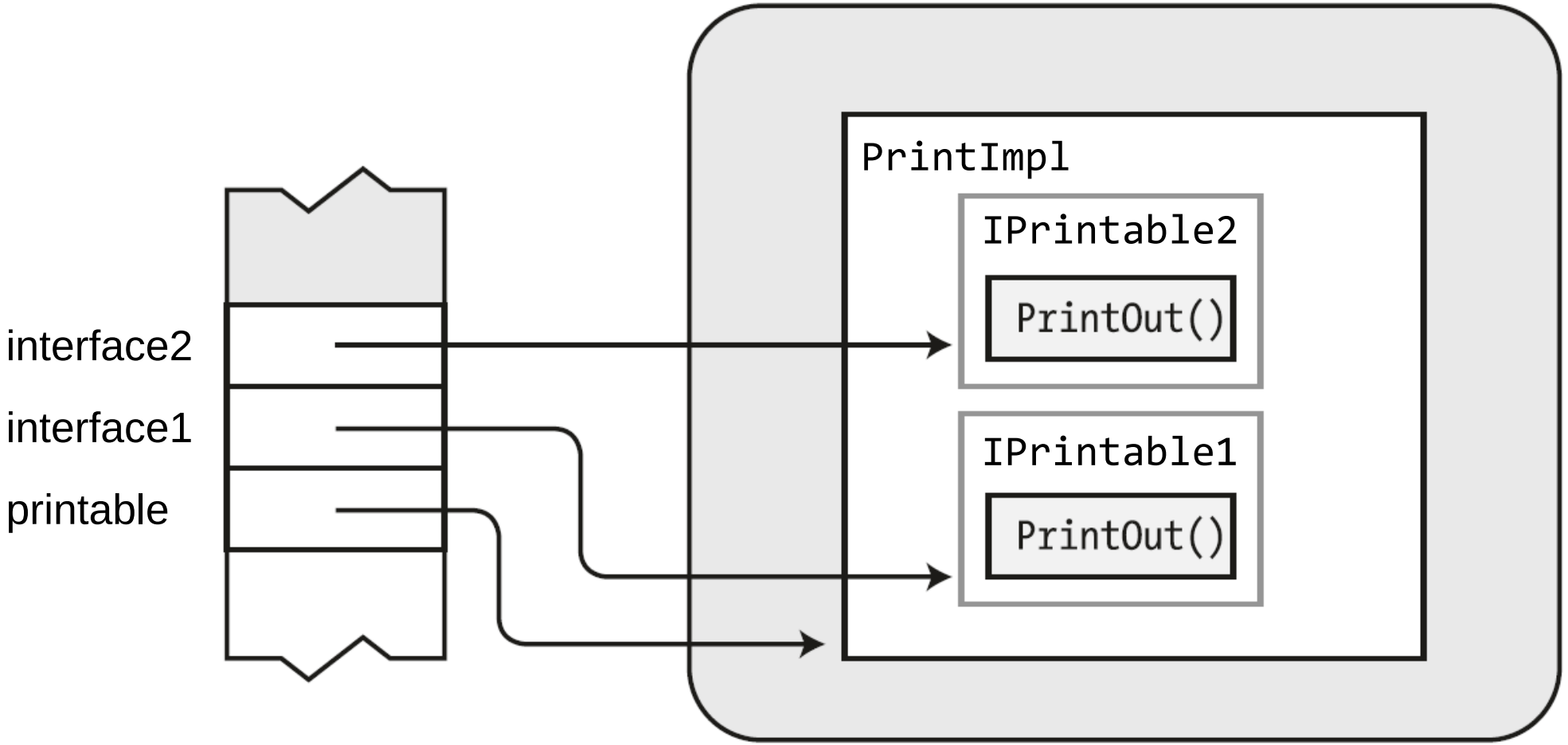
Доступность Явных Реализаций Интерфейсов

```
PrintImpl printable = new();
// Явная реализация IPrintable1.PrintOut:
IPrintable1 interface1 = printable;
interface1.PrintOut("interface1");
// Явная реализация IPrintable2.PrintOut:
IPrintable2 interface2 = printable;
interface2.PrintOut("interface2");
```

Такой Код НЕ Скомпилируется:

```
// Тип PrintImpl не содержит неявной реализации интерфейса:
printable.PrintOut("error");
```

Схема Явных Реализаций Интерфейсов



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

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 *ломало обратную совместимость* пользователям библиотек, т. к. приходилось в обязательном порядке предоставлять реализацию новому(ым) члену(ам).

Кроме того, данная возможность позволяет организовать поддержку traits в языке, о чём можно прочитать в источниках.

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

О traits: https://en.wikipedia.org/wiki/Trait_(computer_programming)
http://scg.unibe.ch/archive/papers/Scha03aTraits.pdf

Доступ к явным реализациям членов интерфейсов

```
interface IIfc1 { void PrintOut(string s); }

class MyClass : IIfc1          {
    void IIfc1.PrintOut(string s) // явная реализация интерфейса
    {      Console.WriteLine("IIfc1");  }
    public void Method1()     {
          PrintOut("...");                    //  1
          this.PrintOut("...");               //  2
          ((IIfc1)this).PrintOut("...");      //  3
          (IIfc1)this.PrintOut("...");        //  4
    }
}
```

Укажите номера строк с ошибками!

Доступ к явным реализациям членов интерфейсов

```
interface IIfc1 { void PrintOut(string s); }

class MyClass : IIfc1          {
    void IIfc1.PrintOut(string s) // явная реализация интерфейса
    {      Console.WriteLine("IIfc1");  }
    public void Method1()     {
          PrintOut("...");                    //  1 ошибка
          this.PrintOut("...");               //  2 ошибка
          ((IIfc1)this).PrintOut("...");      //  3 OK
          (IIfc1)this.PrintOut("...");        //  4 ошибка
    }
}
```

Прочие Изменения в C# 8

Возможности реализации интерфейсов были заметно расширены в C# 8.0.

Кроме реализаций функциональных членов по умолчанию, появились такие возможности, как:

- 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.

Интерфейс IDisposable

Warning: *данный слайд выходит за рамки темы лекции и в первую очередь предназначен для более продвинутой аудитории.*

Интерфейс IDisposable и подход к его реализации предназначены для того, чтобы обеспечить возможность 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();
}
```

https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/implementing-dispose

Реализация IDisposable – Простой Случай

Warning: данный слайд выходит за рамки темы лекции и в первую очередь предназначен для более продвинутой аудитории.

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. Часть 1

Warning: *данный слайд выходит за рамки темы лекции и в первую очередь предназначен для более продвинутой аудитории.*

```
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. Часть 2

Warning: данный слайд выходит за рамки темы лекции и в первую очередь предназначен для более продвинутой аудитории.

```
// 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().
}
```

Ссылки с Источниками по Интерфейсам

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

O 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