



# C# Developer.

## Интерфейсы



**Проверить, идет ли запись**

# **Меня хорошо видно && слышно?**



Ставим "+", если все хорошо  
"-", если есть проблемы

Тема вебинара

# Интерфейсы



**Елена Сычева**

**Team Lead Full-Stack Developer**

**Об опыте:**

Более 15 лет опыта работы разработчиком (C#, Angular, .Net, React, NodeJs)

**Телефон / эл. почта / соц. сети:**

<https://t.me/lentsych>



# Правила вебинара



Активно  
участвуем



Задаем вопрос  
в чат или голосом



Вопросы вижу в чате,  
могу ответить не сразу

## Условные обозначения



Индивидуально



Время, необходимое  
на активность



Пишем в чат



Говорим голосом



Документ



Ответьте себе или  
задайте вопрос

# Маршрут вебинара

Введение

Композиция

Назначение

Реализация интерфейсов

Наследование

Переопределение

Дефолтные методы

# Цели вебинара

К концу занятия вы сможете

- 
1. Объяснить что такое интерфейс
  2. Применять их на практике.
-

# Смысл

## Зачем вам это уметь

1. Это одно из фундаментальных концепций с# и ООП
2. Необходимы в разработке надежных и масштабируемых приложений

# Введение




# Понятие интерфейса

## interface

*noun* [ C ]

UK  /'ɪn.təˌfeɪs/ US  /'ɪn.təˌfeɪs/

Add to word list 

**a connection between two pieces of electronic equipment, or between a person and a computer:**

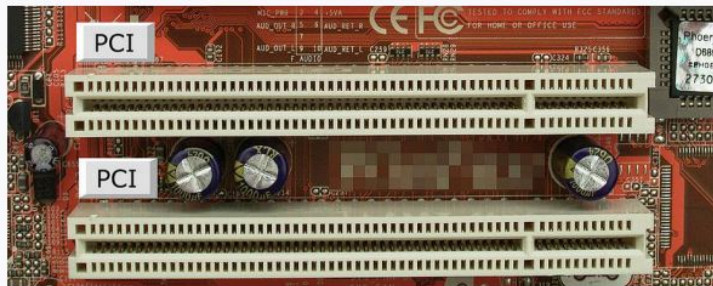
- *My computer has a network interface, which allows me to get to other computers.*
- *The new version of the program comes with a much better **user** interface (= way of showing information to a user) than the original.*

+ 

**a situation, way, or place where two things come together and affect each other:**

- *the interface **between** technology and tradition*
- *We need a clearer interface **between** management and the workforce.*

# Понятие интерфейса



Модульность

Раздельное изготовление

Вариативность

Интерфейс как требование



# Определение

Интерфейс в C# — это контракт, определяющий набор методов, свойств, событий или индексов, которые должен реализовать класс или структура.

В отличие от классов, интерфейсы не предоставляют никакой реализации для определяемых ими членов; они только определяют члены, которые должны быть реализованы.

```
public interface IAnimal
{
    void MakeSound();
    void Eat();
}
```

# Соглашения по использованию интерфейса

Соглашения:

- Находится в отдельном файле
- Название начинается на "I"
- Если в названии нужно обозначить признак по действию, добавляется "-able" (IDisposable, ICloneable)

# Члены интерфейса

```
public interface IExample
{
    // Method declaration
    void Method1();

    // Property declaration
    int Property1 { get; set; }

    // Event declaration
    event EventHandler Event1;

    // Indexer declaration
    string this[int index] { get; set; }

    // Constant declaration
    const int ConstantValue = 42;

    // Default method implementation
    void DefaultMethod()
    {
        Console.WriteLine("This is a default method implementation.");
    }
}
```

Метод: `void Method1()` должен быть реализован любым классом, реализующим `IExample`.

Свойство: `int Property1 { get; set; }` должно быть реализовано, включая как геттер, так и сеттер.

Событие: `event EventHandler Event1` должно быть реализовано для обработки событий.

Индексатор: `string this[int index] { get; set; }` должно быть реализовано для индексации.

Константа: `const int ConstantValue = 42` предоставляет постоянное значение, которое нельзя изменить.

Метод по умолчанию: `void DefaultMethod()` предоставляет реализацию по умолчанию, которая может быть опционально переопределена реализующим классом

# Различия между классом и интерфейсом

	Класс	Интерфейс
<b>Реализация.</b>	Предоставляет реализации для своих членов.	Только объявляет методы, свойства, события или индексаторы.
<b>Создание экземпляра</b>	может быть создан для создания объектов	не может быть создан напрямую
<b>Множественное наследование</b>	класс может наследовать только от одного базового класса.	класс может реализовывать несколько интерфейсов, что позволяет использовать форму множественного наследования.

# Различия между классом и интерфейсом

	Класс	Интерфейс
<b>Члены</b>	члены могут иметь разные модификаторы доступа (публичный, приватный, protected, internal). Может содержать поля, конструкторы, деструкторы, статические члены и т. д.	члены по умолчанию являются public и не могут иметь модификаторов доступа. Не могут содержать поля, конструкторы, деструкторы или статические члены.
<b>Варианты использования</b>	используется для определения фактической реализации и поведения объектов.	используется для определения контракта, который могут реализовать несколько классов, гарантируя, что все они предоставляют определенное поведение

# Сценарии использования

- **Определение общего поведения:** Интерфейсы могут определять набор методов, которые должны реализовывать несколько классов. Это обеспечивает согласованность между различными типами.  
Пример: интерфейс `Comparable` позволяет сравнивать объекты разных классов.
- **Разделение кода:** Интерфейсы помогают отделить реализацию класса от кода, который использует этот класс. Это делает код более модульным и простым в обслуживании.  
Пример: интерфейс `ILogger` может быть реализован различными классами ведения журнала, такими как `FileLogger` и `DatabaseLogger`.
- **Полиморфизм:** Интерфейсы обеспечивают полиморфное поведение, позволяя одному методу работать с объектами разных классов, реализующих один и тот же интерфейс.  
Пример: метод, обрабатывающий список объектов `IAntimal`, может работать с любым классом, реализующим интерфейс `IAntimal`.



# Сценарии использования

- **Модульное тестирование и имитация:** Интерфейсы обычно используются в модульном тестировании для создания имитационных реализаций зависимостей.

Пример: интерфейс `IMailService` может быть имитацией для проверки функциональности класса, который отправляет электронные письма без фактической отправки электронных писем.

- **Расширение функциональности:** Интерфейсы можно использовать для расширения функциональности класса без изменения его исходного кода.

Пример: интерфейс `IDisposable` можно реализовать для предоставления логики очистки неуправляемых ресурсов.

```
public interface IPaymentProcessor
{
    void ProcessPayment(decimal amount);
}

public class PayPalProcessor : IPaymentProcessor
{
    public void ProcessPayment(decimal amount)
    {
        Console.WriteLine($"Processing PayPal payment of {amount:C}.");
    }
}

public class CreditCardProcessor : IPaymentProcessor
{
    public void ProcessPayment(decimal amount)
    {
        Console.WriteLine($"Processing credit card payment of {amount:C}.");
    }
}
```



# Композиция интерфейсов

# Определение композиция

- Композиция интерфейса предполагает объединение нескольких интерфейсов для создания более сложных и универсальных типов. Это позволяет классу реализовать несколько интерфейсов, тем самым наследуя контракт нескольких источников и способствуя лучшей организации и гибкости кода.

```
public interface IPrintable
{
    void Print();
}

public interface IScannable
{
    void Scan();
}
```

# Преимущества

Разделение и гибкость:

Композиция интерфейсов помогает разделить функциональность в системе. Разбивая сложные функции на более мелкие, специализированные интерфейсы, вы можете создать более гибкий и поддерживаемый код.

Повторное использование:

Когда функциональность определяется в небольших, одноцелевых интерфейсах, ее можно повторно использовать в нескольких классах. Это предотвращает дублирование кода и способствует согласованности.

Полиморфизм:

Композиция интерфейсов позволяет рассматривать класс как несколько типов. Этот полиморфизм особенно полезен при проектировании систем, которым необходимо работать с различными объектами через общий интерфейс.

Поддерживаемость:

Меньшие интерфейсы легче понимать и поддерживать. Изменения одного интерфейса не влияют на другие, что снижает риск ошибок и упрощает расширение системы.

# Пример

```
public interface IPrintable
{
    void Print();
}

public interface IScannable
{
    void Scan();
}

public class MultiFunctionDevice : IPrintable, IScannable
{
    public void Print()
    {
        Console.WriteLine("Printing document...");
    }

    public void Scan()
    {
        Console.WriteLine("Scanning document...");
    }
}
```

# Назначение интерфейсов

# Проблема жесткой связности

```
public class UserService
{
    private UserRepository _userRepository;

    1 usage  1 <1>
    public UserService()
    {
        _userRepository = new UserRepository();
    }
}
```

Разделение кода относится к практике сокращения зависимостей между различными частями приложения. Это делает систему более модульной и простой в обслуживании.

Как помогают интерфейсы: Интерфейсы определяют контракт без указания реализации. Программируя интерфейс, вы можете изменить базовую реализацию, не затрагивая код, который зависит от интерфейса.



## Проблема разработки «сразу всего функционала»



Повторное использование означает возможность использования компонентов в разных контекстах без внесения изменений. Гибкость позволяет расширять или заменять компоненты без существенных изменений.

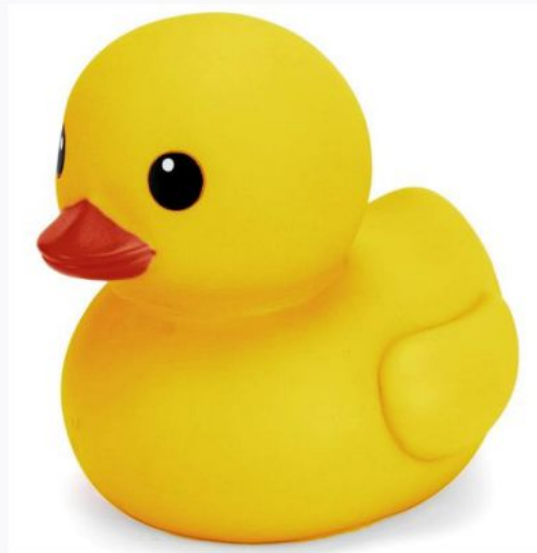
Как помогают интерфейсы:

Интерфейсы позволяют использовать различные реализации взаимозаменяемо. Это способствует повторному использованию, позволяя реализовывать один и тот же интерфейс несколькими способами, и гибкости, позволяя легко заменять компоненты.

Пример:

Рассмотрим систему обработки платежей. Определив интерфейс `IPaymentProcessor`, вы можете создавать различные платежные процессоры, такие как `CreditCardProcessor` и `PayPalProcessor`.

# Проблема отсутствия множественного наследования



# Содействие тестированию и мокингу

```
public interface IEmailService
{
    void SendEmail(string recipient, string subject, string body);
}

public class NotificationService
{
    private readonly IEmailService _emailService;

    public NotificationService(IEmailService emailService)
    {
        _emailService = emailService;
    }

    public void Notify(string recipient, string message)
    {
        _emailService.SendEmail(recipient, "Notification", message);
    }
}
```

Определение:

Тестирование и имитация подразумевают создание упрощенных версий компонентов для проверки их взаимодействия и поведения в изоляции.

Как помогают интерфейсы:

Интерфейсы упрощают создание имитационных реализаций для целей тестирования. Это позволяет вам тестировать компоненты в изоляции, предоставляя контролируемое поведение с помощью имитаций.

```
public class MockEmailService : IEmailService
{
    public List<string> SentEmails = new List<string>();

    public void SendEmail(string recipient, string subject, string body)
    {
        SentEmails.Add($"{recipient}: {subject} - {body}");
    }
}
```

# Реализация интерфейсов

# Реализация методов интерфейса в классах

## Определение:

Когда класс реализует интерфейс, он должен предоставлять конкретные реализации для всех членов, определенных в этом интерфейсе. Это означает, что класс должен содержать фактический код, который выполняет операции, указанные методами, свойствами, событиями или индексаторами интерфейса.

## Шаги по реализации методов интерфейса:

- Определите интерфейс: создайте интерфейс с методами, свойствами, событиями или индексаторами, которые вы хотите реализовать.
- Реализуйте интерфейс в классе: класс должен предоставлять конкретные реализации для всех членов, объявленных в интерфейсе.

```
// Define the interface
public interface IVehicle
{
    void Start();
    void Stop();
}

// Implement the interface in a class
public class Car : IVehicle
{
    public void Start()
    {
        Console.WriteLine("Car started.");
    }

    public void Stop()
    {
        Console.WriteLine("Car stopped.");
    }
}
```

# Использование

```
public class Program
{
    public static void Main()
    {
        IVehicle myCar = new Car();
        myCar.Start(); // Output: Car started.
        myCar.Stop();  // Output: Car stopped.
    }
}
```



# Наследование интерфейсов

# Наследование

- Наследование интерфейсов в C# позволяет одному интерфейсу наследовать от другого интерфейса. Это означает, что производный интерфейс будет включать в себя все элементы базового интерфейса, а также любые дополнительные элементы, которые он определяет.

Назначение:

- Расширение функциональности: позволяет расширить существующий интерфейс, добавляя больше членов без изменения исходного интерфейса.
- Повторное использование: способствует повторному использованию, позволяя интерфейсам надстраиваться над другими интерфейсами.



# Пример

```
// Define the base interface
public interface IAnimal
{
    void Eat();
}

// Define the derived interface that inherits from the base interface
public interface IDog : IAnimal
{
    void Bark();
}
```

# Реализация унаследованных интерфейсов в классах

```
// Define the base interface
public interface IAnimal
{
    void Eat();
}

// Define the derived interface that inherits from the base interface
public interface IDog : IAnimal
{
    void Bark();
}

// Implement the derived interface in a class
public class GermanShepherd : IDog
{
    public void Eat()
    {
        Console.WriteLine("The dog is eating.");
    }

    public void Bark()
    {
        Console.WriteLine("The dog is barking.");
    }
}
```

# Множественное наследование интерфейсов

```
// Define the base interfaces
public interface IAnimal
{
    void Eat();
}

public interface IPet
{
    void Play();
}

// Define the derived interface that inherits from multiple interfaces
public interface ICat : IAnimal, IPet
{
    void Meow();
}
```

# Реализация

```
// Implement the derived interface in a class
public class PersianCat : ICat
{
    public void Eat()
    {
        Console.WriteLine("The cat is eating.");
    }

    public void Play()
    {
        Console.WriteLine("The cat is playing.");
    }

    public void Meow()
    {
        Console.WriteLine("The cat is meowing.");
    }
}
```

# Реализация интерфейса с идентичными методами

# Явная реализация интерфейса

```
interface IFirstInterface
{
    void Method();
}

interface ISecondInterface
{
    void Method();
}

public class ImplementationClass : IFirstInterface, ISecondInterface
{
    void IFirstInterface.Method()
    {
        Console.WriteLine("IFirstInterface Method implementation.");
    }

    void ISecondInterface.Method()
    {
        Console.WriteLine("ISecondInterface Method implementation.");
    }
}
```

Явная реализация интерфейса — это метод, используемый в C# для разрешения конфликтов, возникающих, когда класс реализует несколько интерфейсов, содержащих методы с одинаковой сигнатурой. Используя явную реализацию интерфейса, вы можете предоставить отдельные реализации для каждого метода интерфейса, избегая конфликтов имен и гарантируя, что контракт каждого интерфейса выполняется правильно.

# Явная реализация интерфейса

```
public class Program
{
    public static void Main()
    {
        ImplementationClass obj = new ImplementationClass();

        IFirstInterface first = obj;
        first.Method(); // Output: IFirstInterface Method implementation.

        ISecondInterface second = obj;
        second.Method(); // Output: ISecondInterface Method implementation.
    }
}
```

Ключевые моменты:

- Явная реализация интерфейса гарантирует, что имена методов не будут конфликтовать.
- Методы можно вызывать только через экземпляры интерфейса, а не напрямую через экземпляр класса.

# Разрешение конфликтов имен методов

```
interface IDrawable
{
    void Draw();
}

interface IRenderable
{
    void Draw();
}

public class Shape : IDrawable, IRenderable
{
    void IDrawable.Draw()
    {
        Console.WriteLine("Drawing as IDrawable.");
    }

    void IRenderable.Draw()
    {
        Console.WriteLine("Drawing as IRenderable.");
    }
}
```

При реализации нескольких интерфейсов, имеющих методы с одинаковыми сигнатурами, могут возникнуть конфликты. Явная реализация интерфейса — основной способ разрешения этих конфликтов в C#.



# Использование

```
public class Program
{
    public static void Main()
    {
        Shape shape = new Shape();

        IDrawable drawable = shape;
        drawable.Draw(); // Output: Drawing as IDrawable.

        IRenderable renderable = shape;
        renderable.Draw(); // Output: Drawing as IRenderable.
    }
}
```

# Константы и Дефолтная реализация

# Константы

В C# вы можете определять константы в интерфейсах. Эти константы неявно статичны и не могут быть изменены. Константы в интерфейсах можно использовать для предоставления фиксированных значений, общих для разных реализаций.

Ключевые моменты:

- Статическая природа: константы в интерфейсах статичны и используются во всех реализациях.
- Только для чтения: значения констант не могут быть изменены после компиляции.
- Нет доступа к экземпляру: доступ к константам невозможен через экземпляр класса, реализующего интерфейс; доступ к ним должен быть возможен через сам интерфейс.

```
public interface ILogger
{
    const string DefaultLogLevel = "INFO";

    void Log(string message);

    void LogError(string message)
    {
        Log($"ERROR: {message}");
    }
}

public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
```

# Реализация по умолчанию

Начиная с C# 8.0, интерфейсы могут предоставлять реализации методов по умолчанию. Это позволяет интерфейсам включать тела методов, позволяя определять поведение по умолчанию, которое реализующие классы могут использовать как есть или переопределить.

Ключевые моменты:

- Необязательное переопределение: реализующие классы могут использовать реализацию по умолчанию или предоставить свою собственную.
- Поведение по умолчанию: предоставляет способ определить общее поведение в интерфейсах, которое может использоваться в нескольких реализациях.

```
public interface ILogger
{
    const string DefaultLogLevel = "INFO";

    void Log(string message);

    void LogError(string message)
    {
        Log($"ERROR: {message}");
    }
}

public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
```

# Реализация по умолчанию

Ограничения:

- Нет полей экземпляра: интерфейсы не могут содержать поля экземпляра или нестатические члены, кроме методов, свойств, событий и индексаторов.
- Нет конструкторов: интерфейсы не могут иметь конструкторов. Поэтому методы по умолчанию не могут полагаться на поля экземпляра для инициализации.
- Перегрузка методов: методы по умолчанию не могут быть перегружены так же, как методы в классах. Каждый метод в интерфейсе должен иметь уникальную сигнатуру.
- Статические члены: интерфейсы не могут содержать статические методы, а методы по умолчанию не являются по-настоящему статическими, а предоставляют тело метода, которое совместно используют все реализующие классы.

# Интерфейс и абстрактный базовый класс

		Абстрактный класс	Интерфейс
Назначение		Предоставить классам-наследникам базовый функционал	Предоставить контракт, который должны выполнить классы, реализующие этот интерфейс
Множественное наследование/Реализация		Множественного наследования нет	Множественная реализация есть
В каком виде содержит методы		Декларации, Декларации с реализациями, которые доступны потомкам	Декларации, Декларации с реализациями, которые недоступны потомкам (дефолтные члены)
Возможные члены	Поля	Поля и делегаты есть, Константы есть	<b>Полей и делегатов нет,</b> Константы есть
	Декларации	1. Свойства 2. События 3. Методы 4. Индексаторы	1. Свойства 2. События 3. Методы 4. Индексаторы
	Декларации с реализациями	1. Свойства 2. События 3. Методы 4. Индексаторы	1. Свойства 2. Методы 3. Индексаторы <b>Недоступны из реализующих классов</b>
	Статические члены	1. Поля 2. Делегаты 3. Свойства 4. Методы 5. События 6. Операторы	1. Поля 2. Делегаты 3. Свойства 4. Методы 5. События 6. Операторы
	Конструктор	Да (но экземпляр создать нельзя) Статический конструктор есть	Нет Статический конструктор есть

# Стандартные интерфейсы



# Стандартные интерфейсы

IEnumerator - определяет функционал для перебора внутренних объектов в контейнере

IEnumerable – определяет перечислитель для перебора коллекции

ICollection: IEnumerable – добавляется метод определения количества элементов, методы копирования и синхронизации

ICollection: IEnumerable – добавляется индексатор и др. методы

IQueryable: IEnumerable – интерфейс для Linq-To-Sql

IDisposable

IComparer

# Вопросы?



Ставим “+”,  
если вопросы есть



Ставим “-”,  
если вопросов нет

# Рефлексия

# Вопросы для проверки

[https://docs.google.com/forms/d/e/1FAIpQLScxSLyTe9\\_ILyZ\\_3livodwO206D7PZIoRUS8L6mKeRmlwl8TA/viewform](https://docs.google.com/forms/d/e/1FAIpQLScxSLyTe9_ILyZ_3livodwO206D7PZIoRUS8L6mKeRmlwl8TA/viewform)

# Рефлексия



С какими впечатлениями уходите с вебинара?



Как будете применять на практике то, что узнали на вебинаре?