# Объектно-Ориентированное Программирование в C#: Подробная лекция с примерами

В этой лекции мы подробно рассмотрим ключевые концепции ООП, такие как инкапсуляция, наследование, полиморфизм и абстракция, а также углубимся в их практическую реализацию в C# с множеством конкретных примеров кода.

## 1. Введение в Объектно-Ориентированное Программирование в C#

### 1.1. Основы Объектно-Ориентированного Программирования (ООП)

Фундамент ООП базируется на четырех ключевых принципах, которые обеспечивают его эффективность и широкое применение:

* **Инкапсуляция:** Этот принцип подразумевает связывание данных (полей) и методов, которые манипулируют этими данными, в единый компонент, известный как класс. Ключевым аспектом инкапсуляции является сокрытие внутренних деталей реализации от внешнего мира, предоставляя контролируемый доступ через публичные методы и свойства. Такой подход добавляет дополнительный уровень защиты, оберегая конфиденциальную информацию от несанкционированного доступа или случайных изменений.1
* **Абстракция:** Абстракция направлена на упрощение сложности кода путем представления его высокоуровневого обзора, при этом скрывая несущественные детали реализации. Сложные аспекты кода группируются в простые классы, что позволяет разработчикам взаимодействовать только с необходимыми секциями кода, не углубляясь в их внутреннее устройство.2
* **Полиморфизм:** Термин "полиморфизм" происходит от греческого слова, означающего "много форм", и описывает способность объектов принимать множество обличий. Он позволяет методам, свойствам или операторам вести себя по-разному в зависимости от контекста, в котором они вызываются. В C# полиморфизм достигается двумя основными способами: через перегрузку методов (полиморфизм времени компиляции) и переопределение методов (полиморфизм времени выполнения). Это позволяет обрабатывать объекты как экземпляры их базового класса, обеспечивая гибкость и повторное использование кода.2
* **Наследование:** Наследование — это механизм, который позволяет новому классу, называемому производным или дочерним классом, повторно использовать функции и атрибуты существующего класса, известного как базовый или родительский класс. Производный класс может модифицировать или расширять свойства родительского класса, сохраняя при этом его основные качества. Это способствует повторному использованию кода и созданию иерархий классов, отражающих реальные отношения между сущностями.2

### 1.2. Классы и Объекты в C#

**Класс** выступает в роли чертежа для объекта, определяя его структуру и поведение. Он содержит **поля** — переменные, используемые для хранения данных, и **методы** — функции, предназначенные для выполнения определенных задач. Поля и методы, объявленные внутри класса, называются **членами** этого класса.7

**Пример класса Dog:**

using System;  
  
public class Dog  
{  
 // Поле (атрибут)  
 public string Breed;   
  
 // Метод (поведение)  
 public void Bark()  
 {  
 Console.WriteLine("Гав-Гав!!");  
 }  
}

В этом примере Dog — это имя класса, Breed — поле, а Bark() — метод.

**Объект** является конкретным экземпляром класса, созданным на основе этого чертежа с использованием ключевого слова new.7 Если класс

Dog — это чертеж, то бульдог или немецкая овчарка — это объекты этого класса. После создания объект может получать доступ к полям и методам, определенным в его классе, используя оператор точки (.).

**Пример создания и использования объекта Dog:**

public class Program  
{  
 public static void Main(string args)  
 {  
 // Создаем объект Dog  
 Dog bullDog = new Dog();  
  
 // Доступ к полю breed и его установка  
 bullDog.Breed = "Бульдог";  
 Console.WriteLine($"Порода собаки: {bullDog.Breed}");  
  
 // Вызов метода bark  
 bullDog.Bark();   
 }  
}  
Вывод:

Порода собаки: Бульдог  
Гав-Гав!!

Из одного класса можно создать **множество объектов**, и каждый из них будет иметь свою собственную версию полей с различными значениями.7

**Пример создания нескольких объектов Employee:**

using System;  
  
public class Employee  
{  
 public string Name;  
 public string Department;  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 // Создаем первый объект Employee  
 Employee sheeran = new Employee();  
 sheeran.Name = "Ширан";  
 sheeran.Department = "Разработка";  
 Console.WriteLine($"Сотрудник: {sheeran.Name}, Отдел: {sheeran.Department}");  
  
 // Создаем второй объект Employee  
 Employee taylor = new Employee();  
 taylor.Name = "Тейлор";  
 taylor.Department = "Контент";  
 Console.WriteLine($"Сотрудник: {taylor.Name}, Отдел: {taylor.Department}");  
 }  
}

Вывод:

Сотрудник: Ширан, Отдел: Разработка  
Сотрудник: Тейлор, Отдел: Контент

Каждый объект sheeran и taylor имеет свою собственную версию поля Department с разными значениями.

### 1.3. Пространства Имен (Namespaces) в C#

**Пространства имен** в C# используются для организации множества классов в приложениях, что способствует снижению избыточности кода и предотвращению конфликтов имен между классами с одинаковыми названиями.9

**Синтаксис определения пространства имен:**

namespace Namespace\_Name  
{  
 // Тело пространства имен: классы, структуры, другие пространства имен  
}

**Пример определения пространства имен MyCompany.Utilities:**

namespace MyCompany.Utilities  
{  
 public class DateHelper  
 {  
 public string FormatDate(DateTime date)  
 {  
 return date.ToString("yyyy-MM-dd");  
 }  
 }  
}

Для доступа к членам пространства имен без использования директивы using требуется полное квалифицированное имя.

**Пример использования полного квалифицированного имени:**

using System; // Для Console.WriteLine  
  
namespace MyApp  
{  
 public class Program  
 {  
 public static void Main(string args)  
 {  
 // Полное квалифицированное имя: Namespace.Class.Method  
 System.Console.WriteLine("Hello, World!");   
 }  
 }  
}

**Директива using** импортирует пространства имен, позволяя напрямую обращаться к их типам и методам, что значительно сокращает объем кода.9

**Пример использования директивы using:**

using System; // Импортируем пространство имен System  
  
namespace MyApp  
{  
 public class Program  
 {  
 public static void Main(string args)  
 {  
 // Теперь Console.WriteLine() можно вызывать напрямую  
 Console.WriteLine("Hello, World!");   
 }  
 }  
}

Пространства имен также могут быть **вложенными**, то есть одно пространство имен может содержать другое.9

**Пример вложенных пространств имен:**

using System;  
  
namespace OuterNamespace  
{  
 namespace InnerNamespace // Вложенное пространство имен  
 {  
 public class MyClass  
 {  
 public void DisplayMessage()  
 {  
 Console.WriteLine("Сообщение из вложенного пространства имен.");  
 }  
 }  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 // Доступ к классу через полное квалифицированное имя  
 OuterNamespace.InnerNamespace.MyClass obj1 = new OuterNamespace.InnerNamespace.MyClass();  
 obj1.DisplayMessage();  
  
 // Или с использованием директивы using для вложенного пространства имен  
 using OuterNamespace.InnerNamespace;  
 MyClass obj2 = new MyClass();  
 obj2.DisplayMessage();  
 }  
}

Вывод:

Сообщение из вложенного пространства имен.  
Сообщение из вложенного пространства имен.

#### Глобальные пространства имен и неявные using

**"Глобальное пространство имен"** представляет собой единое пространство объявлений, куда все единицы компиляции вносят свои члены.12

**Директивы extern alias** позволяют ссылаться на различные иерархии пространств имен, что особенно полезно для разрешения конфликтов, когда разные сборки содержат одноименные типы.12

Пример extern alias:

Представьте, что у вас есть две сторонние библиотеки, LibA.dll и LibB.dll, и обе содержат класс MyClass в пространстве имен Common.

Для использования extern alias необходимо сначала добавить ссылки на сборки с псевдонимами в файле проекта (.csproj):

<ItemGroup>  
 <Reference Include="LibA" Aliases="LibAAlias" />  
 <Reference Include="LibB" Aliases="LibBAlias" />  
</ItemGroup>

Затем в коде C#:

extern alias LibAAlias; // Объявляем внешний псевдоним для LibA  
extern alias LibBAlias; // Объявляем внешний псевдоним для LibB  
  
using System;  
  
namespace MyProject  
{  
 public class Program  
 {  
 public static void Main(string args)  
 {  
 // Доступ к MyClass из LibA  
 LibAAlias::Common.MyClass objA = new LibAAlias::Common.MyClass();  
 Console.WriteLine($"Объект из LibA: {objA.GetType().Assembly.FullName}");  
  
 // Доступ к MyClass из LibB  
 LibBAlias::Common.MyClass objB = new LibBAlias::Common.MyClass();  
 Console.WriteLine($"Объект из LibB: {objB.GetType().Assembly.FullName}");  
 }  
 }  
}  
  
// Предполагаемые классы в библиотеках (для демонстрации)  
namespace Common  
{  
 public class MyClass { }  
}

*(Примечание: Для запуска этого примера вам потребуется создать две отдельные библиотеки LibA.dll и LibB.dll, каждая из которых содержит класс Common.MyClass, и добавить их в проект с указанными псевдонимами.)*

**Директива global using**, введенная в C# 10, позволяет объявлять пространство имен глобально в одном файле, делая его доступным для всех файлов в проекте и тем самым уменьшая дублирование директив using.

Пример global using:

Создайте файл GlobalUsings.cs (или любой другой файл) в вашем проекте и добавьте:

// GlobalUsings.cs  
global using System;  
global using System.Collections.Generic;  
global using System.Linq;  
global using System.Text;  
global using MyCompany.Utilities; // Ваше пользовательское пространство имен

Теперь в любом другом файле .cs в этом проекте вам не нужно будет явно добавлять эти using директивы.

**Неявные директивы using**, появившиеся в.NET 6, означают, что компилятор C# автоматически добавляет предопределенный набор директив using в зависимости от типа проекта. Например, для консольных приложений по умолчанию неявно включаются System, System.IO, System.Collections.Generic и другие.

**Пример неявных using (для консольного приложения.NET 6+):**

// Program.cs (без явных using директив)  
// See https://aka.ms/new-console-template for more information  
Console.WriteLine("Hello, World!"); // System.Console доступен неявно  
List<string> names = new List<string>(); // System.Collections.Generic доступен неявно

Неявные using можно отключить или удалить определенные из них через конфигурацию файла проекта (.csproj).

<PropertyGroup>  
 <ImplicitUsings>disable</ImplicitUsings>  
</PropertyGroup>  
  
<ItemGroup>  
 <Using Remove="System.Collections" />  
</ItemGroup>

#### Директивы using alias и using static

**Директива using alias** создает псевдоним для пространства имен или типа, упрощая длинные квалифицированные имена.14

**Пример using alias:**

using System;  
using MyVeryLongNamespaceAlias = MyCompany.Utilities.DateAndTimeHelpers; // Псевдоним для пространства имен  
using MyDateTime = System.DateTime; // Псевдоним для типа  
  
namespace MyProject  
{  
 public class Program  
 {  
 public static void Main(string args)  
 {  
 // Использование псевдонима пространства имен  
 MyVeryLongNamespaceAlias.TimeConverter converter = new MyVeryLongNamespaceAlias.TimeConverter();  
 Console.WriteLine($"Используем конвертер из: {converter.GetType().Namespace}");  
   
 // Использование псевдонима типа  
 MyDateTime now = MyDateTime.Now;  
 Console.WriteLine($"Текущее время: {now}");  
 }  
 }  
}  
  
namespace MyCompany.Utilities.DateAndTimeHelpers  
{  
 public class TimeConverter { /\*... \*/ }  
}

Вывод:

Используем конвертер из: MyCompany.Utilities.DateAndTimeHelpers  
Текущее время: 17.07.2025 15:30:00 (зависит от текущего времени)

**Директива using static** позволяет напрямую обращаться к статическим членам (методам, свойствам, полям) и вложенным типам указанного типа без необходимости квалифицировать их именем типа.16

**Пример using static:**

using static System.Console; // Импортируем статические члены класса Console  
using static System.Math; // Импортируем статические члены класса Math  
  
namespace MyProject  
{  
 public class Program  
 {  
 public static void Main(string args)  
 {  
 // Вместо Console.WriteLine(Math.Sqrt(25));  
 WriteLine(Sqrt(25)); // Вызываем WriteLine и Sqrt напрямую  
 WriteLine($"Значение PI: {PI}"); // Доступ к статическому полю PI  
 }  
 }  
}

Вывод:  
Значение PI: 3.141592653589793

using static также может быть применена к типам enum.16

using static System.DayOfWeek; // Импортируем члены перечисления DayOfWeek  
using System;  
  
namespace MyProject  
{  
 public class Program  
 {  
 public static void Main(string args)  
 {  
 DayOfWeek today = DateTime.Today.DayOfWeek;  
 if (today == Saturday |  
  
| today == Sunday) // Вместо DayOfWeek.Saturday |  
| DayOfWeek.Sunday  
 {  
 WriteLine("Это выходной!");  
 }  
 else  
 {  
 WriteLine("Это рабочий день.");  
 }  
 }  
 }  
}

**Вывод (если сегодня четверг):**

Это рабочий день.

### 1.4. Точки входа в программу: Класс Program, метод Main и операторы верхнего уровня

Традиционно, до C# 9, программа на C# требовала метод static void Main(string args) внутри класса Program в пространстве имен в качестве точки входа.19 Это часто приводило к значительному объему шаблонного кода.

**Традиционная структура программы (до C# 9):**

using System;  
  
namespace HelloWorldApp  
{  
 class Program  
 {  
 static void Main(string args) // Точка входа  
 {  
 Console.WriteLine("Hello, World!");  
 }  
 }  
}  
Эта структура обычно занимает 11 строк кода для простой программы "Hello World!".19

**Операторы верхнего уровня (Top-level statements)**, введенные в C# 9, позволяют писать код непосредственно в корне файла, устраняя необходимость в методе Main, классе Program или явном пространстве имен.

**Пример программы с операторами верхнего уровня (C# 9+,.NET 5+):**

// Program.cs  
// See https://aka.ms/new-console-template for more information  
Console.WriteLine("Hello, World!");

Эта программа сокращается до двух строк.19

**Преимущества операторов верхнего уровня:**

* **Краткость:** Значительно уменьшают объем шаблонного кода, особенно для простых программ и скриптов.
* **Упрощенное обучение:** Делают C# более доступным для новичков.
* **Быстрое прототипирование:** Идеально подходят для экспериментов и скриптовых сценариев.
* **Читаемость:** Устраняют "шум" в коде, позволяя сосредоточиться на логике.

**Ограничения операторов верхнего уровня:**

* **Один файл:** Операторы верхнего уровня могут использоваться только в одном исходном файле в приложении.
* **Порядок:** Директивы using должны предшествовать любым операторам верхнего уровня.
* **args:** Переменная args (аргументы командной строки) неявно доступна.
* **await:** Выражения await в операторах верхнего уровня приводят к тому, что компилятор генерирует асинхронную точку входа.

**Пример использования args и await с операторами верхнего уровня:**

// Program.cs  
using System;  
using System.Threading.Tasks; // Необходимо для Task.Delay  
  
Console.WriteLine("Ваш вопрос:");  
foreach (var arg in args) // Доступ к аргументам командной строки  
{  
 Console.Write($"{arg} ");  
}  
Console.WriteLine();  
  
Console.Write("Думаю...");  
await Task.Delay(2000); // Асинхронная операция  
  
string answers = { "Да", "Нет", "Возможно", "Спросите позже" };  
Random rand = new Random();  
Console.WriteLine($"Ответ: {answers[rand.Next(answers.Length)]}");  
  
// Запуск из командной строки: dotnet run -- Могу ли я использовать операторы верхнего уровня?

Вывод (пример):

Ваш вопрос:  
Могу ли я использовать операторы верхнего уровня?   
Думаю...Ответ: Возможно

## II. Фундаментальные строительные блоки: Классы и объекты

### 2.1. Определение классов: Чертежи для объектов

**Класс** — это чертеж или прототип, который определяет атрибуты (поля/свойства) и поведение (методы) конкретного объекта.7

**Синтаксис определения класса:**

[модификатор\_доступа] class ClassName  
{  
 // Члены класса: поля, свойства, методы, конструкторы и т.д.  
}

**Пример класса Student:**

public class Student  
{  
 // Поле  
 public int Id;   
 // Свойство  
 public string FirstName { get; set; }  
 public string LastName { get; set; }  
  
 // Метод  
 public string GetFullName()  
 {  
 return $"{FirstName} {LastName}";  
 }  
}

### 2.2. Поля, свойства и методы как члены класса

**Поля (Fields)** — это переменные уровня класса, которые хранят данные.8 Обычно они

private и доступны через свойства.8

**Пример поля:**

public class Student  
{  
 public int id; // Публичное поле  
 private string \_name; // Приватное поле (обычно с префиксом \_)  
}

**Свойства (Properties)** инкапсулируют приватные поля, предоставляя контролируемый доступ для чтения (get) и записи (set).8 Они могут включать дополнительную логику в своих аксессорах.8

**Пример свойства:**

public class Student  
{  
 private int \_id; // Приватное поле  
  
 public int StudentId // Публичное свойство  
 {  
 get { return \_id; } // Аксессор для чтения  
 set   
 {   
 if (value > 0) // Дополнительная логика валидации  
 \_id = value;   
 } // Аксессор для записи  
 }  
}

**Автоматически реализуемые свойства (Auto-implemented properties)** (с C# 3.0) упрощают объявление свойств, при котором компилятор автоматически создает скрытое приватное поле.8

**Пример автоматически реализуемого свойства:**

public class Student  
{  
 public string FirstName { get; set; } // Компилятор создаст скрытое приватное поле  
 public string LastName { get; set; }  
}

**Свойства с телом выражения (Expression-bodied properties)** (с C# 6) предоставляют лаконичный синтаксис для аксессоров свойств с использованием лямбда-подобных выражений.20

**Пример свойства только для чтения с телом выражения:**

public class Location  
{  
 private string locationName;  
 public Location(string name) { locationName = name; }  
 public string Name => locationName; // Свойство только для чтения с телом выражения  
}

**Пример свойства чтения-записи с телом выражения:**

public class Location  
{  
 private string locationName;  
 public Location(string name) => Name = name; // Выраженно-теловой конструктор  
 public string Name  
 {  
 get => locationName; // Выраженно-теловой аксессор get  
 set => locationName = value; // Выраженно-теловой аксессор set  
 }  
}

**Методы (Methods)** содержат одно или несколько операторов для выполнения в качестве единого целого; они могут возвращать или не возвращать значение и могут иметь входные параметры.8

**Пример метода:**

public class Calculator  
{  
 // Метод, возвращающий значение и принимающий параметры  
 public int Sum(int num1, int num2)  
 {  
 var total = num1 + num2;  
 return total;  
 }  
  
 // Метод, не возвращающий значение (void) и не принимающий параметры  
 public void Greet()  
 {  
 Console.WriteLine("Привет, мир!");  
 }  
}

### 2.3. Создание экземпляров объектов и их использование

**Объект** является экземпляром класса.7 Объекты создаются с помощью ключевого слова

new.

**Синтаксис создания объекта:**

ClassName obj = new ClassName();

**Пример создания объекта Dog:**

using System;  
  
public class Dog  
{  
 public string Breed;   
 public void Bark()  
 {  
 Console.WriteLine("Гав-Гав!!");  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Dog bullDog = new Dog(); // Создаем объект myDog класса Dog  
 bullDog.Breed = "Бульдог"; // Доступ к полю Breed  
 Console.WriteLine($"Порода собаки: {bullDog.Breed}");  
 bullDog.Bark(); // Вызов метода Bark()  
 }  
}

Вывод:

Порода собаки: Бульдог  
Гав-Гав!!

Из одного и того же класса можно создавать несколько объектов, каждый из которых будет иметь свое собственное состояние.12

**Пример создания нескольких объектов Employee:**

using System;  
  
public class Employee  
{  
 public string Name;  
 public string Department;  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Employee sheeran = new Employee(); // Первый объект  
 sheeran.Name = "Ширан";  
 sheeran.Department = "Разработка";  
 Console.WriteLine($"Сотрудник: {sheeran.Name}, Отдел: {sheeran.Department}");  
  
 Employee taylor = new Employee(); // Второй объект  
 taylor.Name = "Тейлор";  
 taylor.Department = "Контент";  
 Console.WriteLine($"Сотрудник: {taylor.Name}, Отдел: {taylor.Department}");  
 }  
}

Вывод:

Сотрудник: Ширан, Отдел: Разработка  
Сотрудник: Тейлор, Отдел: Контент

### 2.4. Конструкторы: Инициализация объектов

**Конструкторы** — это специальные методы, которые автоматически вызываются при создании экземпляра класса.23 Их основное назначение — инициализировать члены данных объекта.23

**Синтаксис конструктора:**

[модификатор\_доступа] ClassName(параметры)  
{  
 // Тело конструктора  
}

Конструктор по умолчанию (без аргументов):

Если вы не предоставляете ни одного конструктора для класса, C# автоматически предоставляет конструктор по умолчанию с пустым телом.23

**Пример неявного конструктора по умолчанию:**

using System;  
  
public class MyClass  
{  
 public int Value { get; set; }  
 // Неявный конструктор MyClass() { } будет создан компилятором  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 MyClass obj = new MyClass(); // Вызывается неявный конструктор по умолчанию  
 Console.WriteLine($"Значение по умолчанию: {obj.Value}"); // Вывод: 0  
 }  
}

**Параметризованный конструктор:** Принимает аргументы для инициализации полей конкретными значениями.23

**Пример параметризованного конструктора:**

using System;  
  
public class Complex  
{  
 private int x;  
 private int y;  
  
 public Complex(int i, int j) // Конструктор с 2 аргументами  
 {  
 x = i;  
 y = j;  
 }  
  
 public void ShowXY()  
 {  
 Console.WriteLine($"{x}i + {y}");  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Complex c1 = new Complex(20, 25); // Вызывается параметризованный конструктор  
 c1.ShowXY();  
 }  
}

Вывод: 20i + 25

**Перегрузка конструкторов:** Класс может иметь несколько конструкторов с одинаковым именем, но с разным количеством, типами или порядком аргументов.24

**Пример перегрузки конструкторов:**

using System;  
  
public class Complex  
{  
 public Complex(int i, int j)  
 {  
 Console.WriteLine("Конструктор с 2 целочисленными аргументами");  
 }  
  
 public Complex(double i, double j)  
 {  
 Console.WriteLine("Конструктор с 2 аргументами типа double");  
 }  
  
 public Complex() // Конструктор без аргументов  
 {  
 Console.WriteLine("Конструктор без аргументов");  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Complex c1 = new Complex(20, 25); // Вывод: Конструктор с 2 целочисленными аргументами  
 Complex c2 = new Complex(2.5, 5.9); // Вывод: Конструктор с 2 аргументами типа double  
 Complex c3 = new Complex(); // Вывод: Конструктор без аргументов  
 }  
}

Вывод:

Конструктор с 2 целочисленными аргументами  
Конструктор с 2 аргументами типа double  
Конструктор без аргументов

**Приватные конструкторы:** Ограничивают создание экземпляров класса. Часто используются в служебных классах, содержащих только статические члены, или для реализации паттернов Singleton.23

**Пример приватного конструктора (для служебного класса):**

using System;  
  
public class MathUtils  
{  
 private MathUtils() { } // Приватный конструктор, предотвращает создание экземпляров  
  
 public static int Add(int a, int b)  
 {  
 return a + b;  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 // MathUtils util = new MathUtils(); // Ошибка компиляции: 'MathUtils.MathUtils()' недоступен из-за его уровня защиты  
 Console.WriteLine($"Сумма: {MathUtils.Add(5, 3)}"); // Доступ только к статическим членам  
 }  
}

Вывод: Сумма: 8

**Статические конструкторы:** Используются для инициализации статических членов данных класса. Они вызываются один раз, до создания любого экземпляра или доступа к любому статическому члену.23 Они не могут принимать аргументы, быть перегруженными или иметь модификаторы доступа.23

**Пример статического конструктора:**

using System;  
  
public class AppSettings  
{  
 public static string ConnectionString { get; private set; }  
 public static int MaxUsers { get; private set; }  
  
 static AppSettings() // Статический конструктор  
 {  
 Console.WriteLine("Статический конструктор AppSettings вызван.");  
 // Инициализация статических членов  
 ConnectionString = "Data Source=my\_database.db";  
 MaxUsers = 100;  
 }  
  
 public static void DisplaySettings()  
 {  
 Console.WriteLine($"Строка подключения: {ConnectionString}");  
 Console.WriteLine($"Максимальное количество пользователей: {MaxUsers}");  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Console.WriteLine("Начало программы.");  
 AppSettings.DisplaySettings(); // Первый доступ к статическому члену, вызывает статический конструктор  
 Console.WriteLine("Продолжение программы.");  
 AppSettings.DisplaySettings(); // Повторный доступ, статический конструктор не вызывается  
 }  
}

Вывод:

Начало программы.  
Статический конструктор AppSettings вызван.  
Строка подключения: Data Source=my\_database.db  
Максимальное количество пользователей: 100  
Продолжение программы.  
Строка подключения: Data Source=my\_database.db  
Максимальное количество пользователей: 100

**Цепочка конструкторов:** Один конструктор может вызывать другой конструктор в том же классе (:this()) или в его базовом классе (:base()).23 Это способствует повторному использованию кода в логике инициализации.

**Пример цепочки конструкторов (:this()):**

using System;  
  
public class Product  
{  
 public string Name { get; set; }  
 public decimal Price { get; set; }  
 public int Quantity { get; set; }  
  
 public Product(string name, decimal price, int quantity)  
 {  
 Name = name;  
 Price = price;  
 Quantity = quantity;  
 Console.WriteLine($"Создан продукт: {Name}, Цена: {Price}, Количество: {Quantity}");  
 }  
  
 public Product(string name, decimal price) : this(name, price, 1) // Вызывает другой конструктор  
 {  
 Console.WriteLine($"Конструктор с именем и ценой вызван.");  
 }  
  
 public Product(string name) : this(name, 0m) // Вызывает конструктор с именем и ценой  
 {  
 Console.WriteLine($"Конструктор только с именем вызван.");  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Console.WriteLine("Создание продукта 'Молоко':");  
 Product p1 = new Product("Молоко");  
 // Вывод:  
 // Создан продукт: Молоко, Цена: 0, Количество: 1  
 // Конструктор с именем и ценой вызван.  
 // Конструктор только с именем вызван.  
  
 Console.WriteLine("\nСоздание продукта 'Хлеб':");  
 Product p2 = new Product("Хлеб", 2.50m);  
 // Вывод:  
 // Создан продукт: Хлеб, Цена: 2.50, Количество: 1  
 // Конструктор с именем и ценой вызван.  
 }  
}

Вывод:

Создание продукта 'Молоко':  
Создан продукт: Молоко, Цена: 0, Количество: 1  
Конструктор с именем и ценой вызван.  
Конструктор только с именем вызван.  
  
Создание продукта 'Хлеб':  
Создан продукт: Хлеб, Цена: 2.50, Количество: 1  
Конструктор с именем и ценой вызван.

**Конструкторы и наследование:** Конструкторы производного класса неявно вызывают конструктор по умолчанию базового класса, если нет явного вызова base().23

**Пример явного вызова конструктора базового класса (:base()):**

using System;  
  
public class BaseClass  
{  
 public BaseClass()  
 {  
 Console.WriteLine("Конструктор BaseClass без параметров вызван.");  
 }  
  
 public BaseClass(string message)  
 {  
 Console.WriteLine($"Конструктор BaseClass с параметром: {message} вызван.");  
 }  
}  
  
public class DerivedClass : BaseClass  
{  
 public DerivedClass() : base("Сообщение из DerivedClass") // Явно вызываем параметризованный конструктор BaseClass  
 {  
 Console.WriteLine("Конструктор DerivedClass без параметров вызван.");  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 DerivedClass d = new DerivedClass();  
 }  
}

Вывод:

Конструктор BaseClass с параметром: Сообщение из DerivedClass вызван.  
Конструктор DerivedClass без параметров вызван.

### 2.5. Инициализаторы: Инициализация объектов и коллекций

**Инициализаторы объектов** позволяют присваивать значения любым доступным полям или свойствам объекта во время его создания, упрощая создание экземпляров.22

**Пример инициализатора объекта:**

using System;  
  
public class Cat  
{  
 public int Age { get; set; }  
 public string Name { get; set; }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 // Использование инициализатора объекта  
 Cat cat = new Cat { Age = 10, Name = "Пушистик" };  
 Console.WriteLine($"Имя кота: {cat.Name}, Возраст: {cat.Age}");  
 }  
}

Вывод:

Имя кота: Пушистик, Возраст: 10

Инициализаторы объектов можно комбинировать с вызовами конструкторов.22

using System;  
  
public class Cat  
{  
 public int Age { get; set; }  
 public string Name { get; set; }  
  
 public Cat(string name)  
 {  
 Name = name;  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 // Комбинирование конструктора и инициализатора объекта  
 Cat sameCat = new Cat("Пушистик") { Age = 10 };  
 Console.WriteLine($"Имя кота: {sameCat.Name}, Возраст: {sameCat.Age}");  
 }  
}

Вывод:

Имя кота: Пушистик, Возраст: 10

**Модификатор required** (C# 11+) обязывает вызывающие стороны устанавливать значение свойства/поля с помощью инициализатора объекта, обеспечивая необходимую инициализацию.22

**Пример модификатора required:**

using System;  
  
public class Pet  
{  
 public required int Age { get; set; } // Это поле должно быть инициализировано  
 public string Name { get; set; } // Это поле необязательно для инициализации  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Pet pet1 = new Pet() { Age = 5 }; // OK: Age инициализирован  
 Console.WriteLine($"Возраст питомца: {pet1.Age}");  
 // Pet pet2 = new Pet(); // Ошибка компиляции: Required member 'Pet.Age' must be set in the object initializer  
 }  
}

Вывод:

Возраст питомца: 5

**Аксессор init** (C# 9+) ограничивает установку значения свойства только инициализацией, делая свойство неизменяемым после создания.22

**Пример аксессора init:**

using System;  
  
public class Person  
{  
 public string FirstName { get; set; } // Можно изменять после инициализации  
 public string LastName { get; init; } // Можно установить только при инициализации  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Person p = new Person() { FirstName = "Иван", LastName = "Иванов" };  
 Console.WriteLine($"Имя: {p.FirstName}, Фамилия: {p.LastName}");  
 p.FirstName = "Петр"; // OK  
 Console.WriteLine($"Новое имя: {p.FirstName}");  
 // p.LastName = "Петров"; // Ошибка компиляции: Init-only property or indexer 'Person.LastName' can only be assigned in an object initializer...  
 }  
}

Вывод:

Имя: Иван, Фамилия: Иванов  
Новое имя: Петр

**Инициализация вложенных объектов:** Позволяет инициализировать свойства вложенных объектов либо путем изменения существующего экземпляра (без new), либо путем создания нового (с new).22

**Пример инициализации вложенных объектов:**

using System;  
  
public class Settings  
{  
 public string Theme { get; set; } = "Light";  
 public int FontSize { get; set; } = 12;  
}  
  
public class Application  
{  
 public string Name { get; set; } = "";  
 public Settings AppSettings { get; } = new Settings(); // AppSettings инициализируется в конструкторе Application  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Console.WriteLine("--- Пример 1: Изменение существующего вложенного объекта ---");  
 // Инициализация вложенного объекта без 'new' (модифицирует существующий экземпляр)  
 var app = new Application  
 {  
 Name = "МоеПриложение",  
 AppSettings = { Theme = "Dark", FontSize = 14 } // Модифицируем существующий AppSettings  
 };  
 Console.WriteLine($"Тема: {app.AppSettings.Theme}, Размер шрифта: {app.AppSettings.FontSize}");  
  
 Console.WriteLine("\n--- Пример 2: Создание нового вложенного объекта ---");  
 // Инициализация вложенного объекта с 'new' (создает новый экземпляр)  
 var app2 = new Application  
 {  
 Name = "МоеПриложение2",  
 AppSettings = new Settings { Theme = "Blue", FontSize = 16 } // Создаем НОВЫЙ экземпляр Settings  
 };  
 Console.WriteLine($"Тема: {app2.AppSettings.Theme}, Размер шрифта: {app2.AppSettings.FontSize}");  
 }  
}

Вывод:

--- Пример 1: Изменение существующего вложенного объекта ---  
Тема: Dark, Размер шрифта: 14  
  
--- Пример 2: Создание нового вложенного объекта ---  
Тема: Blue, Размер шрифта: 16

### 2.6. Деконструкторы: Деконструкция объектов

В C# "деконструкция" относится к извлечению нескольких значений из кортежей или пользовательских типов в отдельные переменные за одну операцию.

**Деконструкция кортежей:** Встроенная поддержка распаковки элементов кортежа.

**Пример деконструкции кортежей:**

using System;  
  
public class Program  
{  
 // Метод, возвращающий кортеж  
 static (string City, int Population, double Area) QueryCityData(string name)  
 {  
 if (name == "Нью-Йорк")  
 return ("Нью-Йорк", 8175133, 468.48);  
 return ("", 0, 0);  
 }  
  
 // Метод, возвращающий кортеж с данными за несколько лет  
 static (string Name, double Area, int Year1, int Population1, int Year2, int Population2) QueryCityDataForYears(string name, int year1, int year2)  
 {  
 int population1 = 0, population2 = 0;  
 double area = 0;  
 if (name == "Нью-Йорк")  
 {  
 area = 468.48;  
 if (year1 == 1960) population1 = 7781984;  
 if (year2 == 2010) population2 = 8175133;  
 return (name, area, year1, population1, year2, population2);  
 }  
 return ("", 0, 0, 0, 0, 0);  
 }  
  
 public static void Main(string args)  
 {  
 // Деконструкция с явным объявлением типов  
 (string city, int population, double area) = QueryCityData("Нью-Йорк");  
 Console.WriteLine($"Город: {city}, Население: {population}");  
  
 // Деконструкция с использованием 'var'  
 var (city2, population2, area2) = QueryCityData("Лондон"); // Вернет ("", 0, 0)  
 Console.WriteLine($"Город: {city2}, Население: {population2}");  
  
 // Деконструкция в уже объявленные переменные  
 string currentCity = "Париж";  
 int currentPopulation = 2141000;  
 (currentCity, currentPopulation, \_) = QueryCityData("Токио"); // Игнорируем Area с помощью '\_'  
 Console.WriteLine($"Текущий город: {currentCity}, Текущее население: {currentPopulation}");  
  
 // Деконструкция с отбрасываемыми значениями  
 var (\_, \_, \_, pop1, \_, pop2) = QueryCityDataForYears("Нью-Йорк", 1960, 2010);  
 Console.WriteLine($"Изменение населения, 1960 по 2010: {pop2 - pop1:N0}");  
 }  
}

**Вывод:**

Город: Нью-Йорк, Население: 8175133  
Город: , Население: 0  
Текущий город: Токио, Текущее население: 0  
Изменение населения, 1960 по 2010: 393,149

**Пользовательские типы:** Классы, структуры или интерфейсы могут поддерживать деконструкцию путем реализации одного или нескольких методов Deconstruct. Эти методы должны возвращать void и использовать параметры out для извлекаемых значений.

**Пример метода Deconstruct для пользовательского типа:**

using System;  
  
public class Person  
{  
 public string FirstName { get; set; }  
 public string LastName { get; set; }  
 public string City { get; set; }  
  
 public Person(string firstName, string lastName, string city)  
 {  
 FirstName = firstName;  
 LastName = lastName;  
 City = city;  
 }  
  
 // Метод Deconstruct (деконструирует в имя и фамилию)  
 public void Deconstruct(out string firstName, out string lastName)  
 {  
 firstName = FirstName;  
 lastName = LastName;  
 }  
  
 // Перегруженный метод Deconstruct (деконструирует в имя, фамилию и город)  
 public void Deconstruct(out string firstName, out string lastName, out string city)  
 {  
 firstName = FirstName;  
 lastName = LastName;  
 city = City;  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Person p = new Person("Анна", "Смирнова", "Москва");  
  
 // Деконструкция с использованием первого метода Deconstruct  
 var (fName, lName) = p;  
 Console.WriteLine($"Имя: {fName}, Фамилия: {lName}");  
  
 // Деконструкция с использованием перегруженного метода Deconstruct  
 var (fName2, lName2, city) = p;  
 Console.WriteLine($"Имя: {fName2}, Фамилия: {lName2}, Город: {city}");  
  
 // Деконструкция с отбрасываемыми значениями  
 var (fName3, \_, city2) = p; // Игнорируем фамилию  
 Console.WriteLine($"Имя: {fName3}, Город: {city2}");  
 }  
}

**Вывод:**

Имя: Анна, Фамилия: Смирнова  
Имя: Анна, Фамилия: Смирнова, Город: Москва  
Имя: Анна, Город: Москва

**Методы расширения для деконструкции:** Позволяют деконструировать объекты типов, исходным кодом которых вы не управляете, путем определения Deconstruct как метода расширения.

**Пример метода расширения Deconstruct:**

using System;  
using System.Reflection; // Для PropertyInfo  
  
public static class ReflectionExtensions  
{  
 // Метод расширения для деконструкции PropertyInfo  
 public static void Deconstruct(this PropertyInfo p, out bool isStatic, out Type propertyType)  
 {  
 isStatic = p.GetMethod?.IsStatic?? false;  
 propertyType = p.PropertyType;  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 PropertyInfo prop = typeof(DateTime).GetProperty("Now");  
 var (isStatic, propType) = prop; // Деконструкция PropertyInfo с помощью метода расширения  
 Console.WriteLine($"Свойство DateTime.Now: Статическое: {isStatic}, Тип: {propType.Name}");  
 }  
}

**Вывод:**

Свойство DateTime.Now: Статическое: True, Тип: DateTime

**Типы record:** Компилятор автоматически генерирует метод Deconstruct для типов record, объявленных с позиционными параметрами.

**Пример деконструкции record типа:**

using System;  
  
public record Book(string Title, string Author, int Year);  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Book book = new Book("Война и мир", "Лев Толстой", 1869);  
  
 // Деконструкция record типа  
 var (title, author, year) = book;  
 Console.WriteLine($"Название: {title}, Автор: {author}, Год: {year}");  
 }  
}

**Вывод:**

Название: Война и мир, Автор: Лев Толстой, Год: 1869

## III. Структуры данных: Структуры против классов

### 3.1. Структуры: Типы значений с возможностями классов

struct (структура) похожа на класс в C#, но является **типом значения**.27 Она определяется с помощью ключевого слова

struct.27 Структуры могут содержать поля, методы, индексаторы и свойства, как и классы.27

**Синтаксис определения структуры:**

[модификатор\_доступа] struct StructName  
{  
 // Члены структуры: поля, свойства, методы, конструкторы  
}

**Пример базовой структуры Employee:**

using System;  
  
public struct Employee  
{  
 public int Id; // Поле  
 public string Name; // Поле  
  
 // Метод  
 public void DisplayInfo()  
 {  
 Console.WriteLine($"ID сотрудника: {Id}, Имя: {Name}");  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Employee emp; // Объявляем переменную структуры  
 emp.Id = 101; // Доступ к членам структуры  
 emp.Name = "Алексей";  
 emp.DisplayInfo(); // Вызов метода  
  
 Employee emp2 = new Employee(); // Создание экземпляра с 'new' (вызывает неявный конструктор без параметров)  
 Console.WriteLine($"ID по умолчанию: {emp2.Id}, Имя по умолчанию: {emp2.Name?? "null"}");  
 }  
}

Вывод:

ID сотрудника: 101, Имя: Алексей  
ID по умолчанию: 0, Имя по умолчанию: null

**Конструкторы в структурах:** До C# 10 структуры не могли объявлять конструктор экземпляра без параметров.29 C# 10 ввел явные конструкторы без параметров для структур. Параметризованные конструкторы разрешены, но

*все* поля должны быть инициализированы в конструкторе.27

**Пример параметризованного конструктора структуры:**

using System;  
  
public struct Point  
{  
 public int X;  
 public int Y;  
  
 public Point(int x, int y) // Параметризованный конструктор  
 {  
 X = x; // Все поля должны быть инициализированы  
 Y = y;  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Point p1 = new Point(10, 20);  
 Console.WriteLine($"Точка: ({p1.X}, {p1.Y})");  
 }  
}

Вывод:

Точка: (10, 20)

**Свойства в структурах:** Структуры могут использовать свойства для инкапсуляции полей, аналогично классам.27

**Пример свойства в структуре:**

using System;  
  
public struct Employee  
{  
 private int \_id;  
   
 public int Id // Свойство  
 {  
 get { return \_id; }  
 set { \_id = value; }  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Employee emp = new Employee();  
 emp.Id = 1;  
 Console.WriteLine($"ID сотрудника: {emp.Id}");  
 }  
}

Вывод:

ID сотрудника: 1

### 3.2. Ключевые различия: Типы значений против ссылочных типов

Выбор между struct и class является фундаментальным проектным решением с глубокими последствиями для производительности, использования памяти и корректности. Непонимание их семантики копирования (по значению против по ссылке) является распространенным источником незаметных ошибок в приложениях C#.

|  |  |  |
| --- | --- | --- |
| Характеристика | Класс (Ссылочный тип) | Структура (Тип значения) |
| **Категория типа** | Ссылочный тип | Тип значения |
| **Размещение в памяти** | Куча (Heap) | Стек (Stack) для локальных переменных и параметров; Встроенно для полей |
| **Поведение при копировании** | Копируется ссылка; обе переменные указывают на один и тот же объект. Изменения через одну переменную влияют на объект, на который ссылается другая. | Копируется значение данных; операции с копией не влияют на оригинал. |
| **Изменяемость** | Изменяемый (Mutable) | Обычно неизменяемый (Immutable), изменения создают новую копию. |
| **Допускает null** | Может быть null. | Не может быть null по умолчанию (только если это тип значения, допускающий null, T?). 29 |
| **Наследование** | Поддерживает наследование от других классов и реализацию нескольких интерфейсов. 31 | Неявно наследует от System.ValueType и не может указывать базовый класс. Неявно запечатан (sealed). 29 |
| **Конструктор по умолчанию** | Неявный конструктор по умолчанию (если не определен). 23 | Неявный конструктор без параметров (C# 9- и C# 10+). 29 |
| **Инициализаторы полей** | Разрешены для полей экземпляра. | Не разрешены для полей экземпляра (до C# 10). 29 |
| **Упаковка/Распаковка** | Упаковка только при приведении к object или интерфейсу. 32 | Всегда упаковывается при приведении к object или интерфейсу (копирование значения в кучу). 29 |

**Пример поведения при копировании:**

using System;  
  
// Класс (ссылочный тип)  
public class MyClass  
{  
 public int Value { get; set; }  
}  
  
// Структура (значимый тип)  
public struct MyStruct  
{  
 public int Value { get; set; }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Console.WriteLine("--- Поведение класса (ссылочный тип) ---");  
 MyClass obj1 = new MyClass { Value = 10 };  
 MyClass obj2 = obj1; // Копируется ссылка  
 obj2.Value = 20; // Изменяем через obj2  
 Console.WriteLine($"Класс: obj1.Value = {obj1.Value}, obj2.Value = {obj2.Value}");  
  
 Console.WriteLine("\n--- Поведение структуры (значимый тип) ---");  
 MyStruct struct1 = new MyStruct { Value = 10 };  
 MyStruct struct2 = struct1; // Копируется значение  
 struct2.Value = 20; // Изменяем копию  
 Console.WriteLine($"Структура: struct1.Value = {struct1.Value}, struct2.Value = {struct2.Value}");  
 }  
}

**Вывод:**

--- Поведение класса (ссылочный тип) ---  
Класс: obj1.Value = 20, obj2.Value = 20  
  
--- Поведение структуры (значимый тип) ---  
Структура: struct1.Value = 10, struct2.Value = 20

### 3.3. Структуры readonly и члены экземпляра readonly

**Структуры readonly:** Введенный в C# 7.2 модификатор readonly в объявлении struct указывает, что тип структуры является **неизменяемым**. Все поля экземпляра внутри структуры readonly также должны быть readonly.29 Ни один из ее членов не может иметь сеттеров.

**Пример readonly struct:**

using System;  
  
public readonly struct Point3D // Неизменяемая структура  
{  
 public double X { get; } // Свойства только для чтения  
 public double Y { get; }  
 public double Z { get; }  
  
 public Point3D(double x, double y, double z)  
 {  
 X = x;  
 Y = y;  
 Z = z;  
 }  
  
 // Этот метод неявно является readonly, потому что структура является readonly  
 public double CalculateDistanceToOrigin()  
 {  
 // Попытка изменить X, Y или Z здесь приведет к ошибке компиляции  
 // this.X = 10; // Ошибка компиляции  
 return Math.Sqrt(X \* X + Y \* Y + Z \* Z);  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Point3D p = new Point3D(3, 4, 0);  
 Console.WriteLine($"Расстояние до начала координат: {p.CalculateDistanceToOrigin()}");  
 }  
}

**Вывод:**

Расстояние до начала координат: 5

**Члены экземпляра readonly:** Внутри struct модификатор readonly может быть применен к членам экземпляра (методам, свойствам, индексаторам, событиям), чтобы указать, что они **не изменяют состояние структуры**. Это позволяет компилятору избегать защитных копий при вызове членов на экземплярах структур readonly, что улучшает производительность.

**Пример readonly члена экземпляра в структуре:**

using System;  
  
public struct MutablePoint  
{  
 public int X;  
 public int Y;  
  
 public readonly double GetDistance() // Метод помечен как readonly  
 {  
 // X = 10; // Ошибка компиляции: нельзя изменять состояние в readonly методе  
 return Math.Sqrt(X \* X + Y \* Y);  
 }  
  
 public void Move(int dx, int dy) // Изменяющий метод  
 {  
 X += dx;  
 Y += dy;  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 MutablePoint p = new MutablePoint { X = 3, Y = 4 };  
 Console.WriteLine($"Расстояние: {p.GetDistance()}"); // Вызов readonly метода  
 p.Move(1, 1);  
 Console.WriteLine($"Новая позиция: ({p.X}, {p.Y})");  
 }  
}  
**Вывод:**

Расстояние: 5  
Новая позиция: (4, 5)

## IV. Организация и контроль кода

### 4.1. Пространства имен: Предотвращение конфликтов имен и организация кода

**Пространства имен** (namespace) в C# используются для организации классов и предотвращения конфликтов имен, выступая в качестве контейнера для связанных типов.9

Основной синтаксис определения пространства имен:

namespace MyCompany.Utilities  
{  
 public class MathHelper { /\*... \*/ }  
}

Для дальнейшей организации кода можно использовать **вложенные пространства имен**.9

using System;  
  
namespace OuterNamespace  
{  
 namespace InnerNamespace  
 {  
 public class InnerClass  
 {  
 public void Display()  
 {  
 Console.WriteLine("Я из InnerNamespace.");  
 }  
 }  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 // Доступ через полное квалифицированное имя  
 OuterNamespace.InnerNamespace.InnerClass obj1 = new OuterNamespace.InnerNamespace.InnerClass();  
 obj1.Display();  
  
 // Доступ с использованием директивы using  
 using OuterNamespace.InnerNamespace;  
 InnerClass obj2 = new InnerClass();  
 obj2.Display();  
 }  
}

**Вывод:**

Я из InnerNamespace.  
Я из InnerNamespace.

### 4.2. Директивы using: Оптимизация читаемости кода

**Директива using** позволяет импортировать пространства имен, избегая необходимости указывать полное квалифицированное имя для каждого типа.9

using System; // Импортирует пространство имен System  
using MyCompany.Utilities; // Импортирует ваше пользовательское пространство имен  
  
// Теперь можно использовать Console.WriteLine вместо System.Console.WriteLine  
// И MathHelper вместо MyCompany.Utilities.MathHelper

#### using static: Прямой доступ к статическим членам

Начиная с C# 6, директива **using static** позволяет напрямую обращаться к статическим членам (методам, свойствам, полям) указанного типа без необходимости квалификации их именем типа.16

using static System.Console; // Импортируем статические члены класса Console  
using static System.Math; // Импортируем статические члены класса Math  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 // Вместо Console.WriteLine(Math.Sqrt(3\*3 + 4\*4));  
 WriteLine(Sqrt(3\*3 + 4\*4));   
 }  
}

Вывод:5

#### global using и неявные директивы using (C# 10+)

**global using** — это функция C# 10, которая позволяет объявить пространство имен глобально для всех файлов в проекте, уменьшая количество повторяющихся директив using.

// В одном файле (например, GlobalUsings.cs)  
global using System;  
global using System.Collections.Generic;  
global using System.Linq;  
global using System.Text;  
//... и так далее

**Неявные директивы using** (введенные в.NET 6+) означают, что SDK.NET (например, Microsoft.NET.Sdk.Web) автоматически включает предопределенный набор global using директив, основанных на типе проекта.

### 4.3. Модификаторы доступа: Определение видимости и инкапсуляции

**Модификаторы доступа** — это ключевые слова в C#, которые контролируют доступность (видимость) классов, членов класса и других типов.1 Они играют центральную роль в концепции

**инкапсуляции**, защищая конфиденциальные данные и гарантируя, что код используется так, как задумано.1

**Таблица: Сводка модификаторов доступа C# с примерами**

|  |  |  |  |
| --- | --- | --- | --- |
| Модификатор | Доступность | Типичный сценарий использования | Пример |
| Public | Из любого места 1 | Для методов или классов, предназначенных для широкого использования в программе или библиотеках 1 | csharp<br>public class DateHelper<br>{<br> public static string FormatDate(DateTime date)<br> {<br> return date.ToString("yyyy-MM-dd");<br> }<br>}<br>// Использование: DateHelper.FormatDate(DateTime.Now); 1 |
| Private | Только внутри того же класса 1 | Для инкапсуляции конфиденциальных данных или вспомогательных методов 1 | csharp<br>public class BankAccount<br>{<br> private decimal balance; // Доступно только внутри BankAccount<br> public void Deposit(decimal amount)<br> {<br> balance += amount;<br> }<br> public decimal GetBalance() { return balance; }<br>}<br>// Использование: new BankAccount().balance = 100; // Ошибка компиляции 1 |
| protected | Внутри того же класса и производных классов 1 | Чтобы подклассы могли повторно использовать функциональность, скрывая ее от других 1 | csharp<br>public class GameCharacter<br>{<br> protected int Health = 100; // Доступно подклассам<br>}<br>public class Warrior : GameCharacter<br>{<br> public void DisplayHealth()<br> {<br> Console.WriteLine($"Здоровье воина: {Health}"); // Доступно<br> }<br>} 1 |
| Internal | Только внутри той же сборки (проекта) 1 | Чтобы компоненты были многократно используемыми внутри проекта без внешнего раскрытия 1 | csharp<br>// В проекте MyProject.Core<br>internal class Logger<br>{<br> internal static void Log(string message)<br> {<br> Console.WriteLine($"[LOG]: {message}");<br> }<br>}<br>// В проекте MyProject.Web (другая сборка)<br>// Logger.Log("Сообщение"); // Ошибка компиляции 1 |
| protected internal | Внутри той же сборки И из производных классов в других сборках 1 | Для общих базовых классов, которые должны наследоваться в других проектах 1 | csharp<br>// В сборке CoreLib<br>public class Shape<br>{<br> protected internal int Sides = 0; // Доступно в CoreLib и подклассам везде<br>}<br>// В сборке WebApp (другая сборка)<br>public class Circle : Shape<br>{<br> public Circle()<br> {<br> Sides = 1; // Доступ разрешен, т.к. Circle - подкласс<br> }<br>} 1 |
| private protected | Внутри того же класса И производных классов в той же сборке 1 | Для строгих сценариев наследования внутри того же проекта 1 | csharp<br>// В сборке MyProject<br>public class Device<br>{<br> private protected string SerialNumber = "SN12345"; // Доступно в Device и его подклассах в MyProject<br>}<br>public class SmartDevice : Device<br>{<br> public void DisplaySerialNumber()<br> {<br> Console.WriteLine($"Серийный номер: {SerialNumber}"); // Доступ разрешен, т.к. SmartDevice - подкласс в той же сборке<br> }<br>} 1 |

**Уровни доступа по умолчанию:**

* Члены enum всегда public.18
* Доступ по умолчанию для class — private.18
* Члены interface всегда public.18
* Доступ по умолчанию для struct — private.18

### 4.4. Область видимости переменных и контекст: Где живут ваши переменные

**Область видимости (Scope)** переменной определяет ту часть приложения, где она доступна.16

* **Область видимости на уровне класса (поля):** Переменные, объявленные внутри класса, но за пределами любого метода, называются полями или членами класса. Они доступны в любом месте этого класса.16  
  **Пример области видимости уровня класса:**

using System;  
  
public class ClassLevel  
{  
 int classLevelVariable = 10; // Переменная уровня класса (поле)  
  
 public void DisplayValue()  
 {  
 Console.WriteLine($"Значение переменной уровня класса: {classLevelVariable}"); // Доступно  
 }  
  
 public static void Main(string args)  
 {  
 ClassLevel obj = new ClassLevel();  
 Console.WriteLine($"Доступ через объект: {obj.classLevelVariable}"); // Доступно  
 obj.DisplayValue();  
 }  
}  
  
Вывод:  
Доступ через объект: 10  
Значение переменной уровня класса: 10

* **Область видимости на уровне метода (локальные переменные):** Переменные, определенные внутри метода, имеют область видимости на уровне метода, что означает, что они доступны только в пределах этого конкретного метода.16 Они прекращают свое существование после завершения выполнения метода.  
  **Пример области видимости уровня метода:**

C#  
using System;  
  
public class MethodLevel  
{  
 public void MyMethod()  
 {  
 string methodLevelVariable = "Привет из метода"; // Переменная уровня метода  
 Console.WriteLine(methodLevelVariable); // Доступно  
 }  
  
 public static void Main(string args)  
 {  
 MethodLevel obj = new MethodLevel();  
 obj.MyMethod();  
 // Console.WriteLine(methodLevelVariable); // Ошибка компиляции: 'methodLevelVariable' не существует в текущем контексте  
 }  
}  
  
16  
  
Вывод:  
Привет из метода

* **Блочная область видимости (вложенная область видимости):** Переменные, объявленные внутри блока кода (например, оператор if, цикл for, цикл while), доступны только внутри этого блока.16  
  **Пример области видимости уровня блока:**

using System;  
  
public class BlockLevel  
{  
 public static void Main(string args)  
 {  
 int x = 10; // Переменная уровня метода Main  
  
 if (x > 5)  
 {  
 string blockLevelVariable = "Внутри блока if"; // Переменная уровня блока  
 Console.WriteLine(blockLevelVariable); // Доступно  
 }  
 // Console.WriteLine(blockLevelVariable); // Ошибка компиляции: 'blockLevelVariable' не существует в текущем контексте  
 }  
}  
  
Вывод:  
Внутри блока if

## V. Принципы объектно-ориентированного программирования в действии

### 5.1. Наследование: Повторное использование и расширение функциональности

**Наследование** — это фундаментальная концепция объектно-ориентированного программирования, которая позволяет производному (дочернему) классу наследовать свойства и методы от базового (родительского) класса, устанавливая отношение "является-чем-то" (is-a).5 Для наследования в C# используется символ двоеточия (

:).5

#### Базовые и производные классы: Отношение "является-чем-то"

* **Базовый класс:** Класс, чьи функции наследуются.6
* **Производный класс:** Класс, который наследует функции другого класса.6

using System;  
  
public class Person  
{  
 public string FirstName { get; set; }  
 public string LastName { get; set; }  
 public string GetFullName() { return $"{FirstName} {LastName}"; }  
}  
  
public class Employee : Person // Employee "является" Person  
{  
 public int EmployeeId { get; set; }  
 public string CompanyName { get; set; }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Employee emp = new Employee();  
 emp.FirstName = "Стив";  
 emp.LastName = "Джобс";  
 emp.EmployeeId = 1;  
 emp.CompanyName = "Apple";  
  
 Console.WriteLine($"Полное имя сотрудника: {emp.GetFullName()}");  
 Console.WriteLine($"ID сотрудника: {emp.EmployeeId}");  
 Console.WriteLine($"Компания: {emp.CompanyName}");  
 }  
}

Вывод:

Полное имя сотрудника: Стив Джобс  
ID сотрудника: 1  
Компания: Apple

Общедоступные (public) и защищенные (protected) члены базового класса доступны в производном классе.42 Приватные (

private) члены базового класса не могут быть доступны напрямую из производного класса.42

#### Типы наследования

* **Единичное наследование:** Производный класс наследует только от одного базового класса. Классы C# поддерживают только единичное наследование.5
* **Многоуровневое наследование:** Производный класс наследует от базового класса, который, в свою очередь, сам является производным от другого класса (например, A -> B -> C).6

using System;  
  
public class Animal { public void Eat() { Console.WriteLine("Животное ест."); } }  
public class Mammal : Animal { public void Run() { Console.WriteLine("Млекопитающее бежит."); } }  
public class Horse : Mammal { public void Gallop() { Console.WriteLine("Лошадь скачет галопом."); } }  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Horse horse = new Horse();  
 horse.Eat(); // Унаследовано от Animal  
 horse.Run(); // Унаследовано от Mammal  
 horse.Gallop(); // Собственный метод Horse  
 }  
}  
  
Вывод:  
Животное ест.  
Млекопитающее бежит.  
Лошадь скачет галопом.

**Иерархическое наследование:** Несколько классов наследуют от одного базового класса.6  
C#  
using System;  
  
public class Animal { public void Eat() { Console.WriteLine("Животное ест."); } }  
public class Dog : Animal { public void Bark() { Console.WriteLine("Собака лает."); } }  
public class Cat : Animal { public void Meow() { Console.WriteLine("Кошка мяукает."); } }  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Dog dog = new Dog();  
 dog.Eat();  
 dog.Bark();  
  
 Cat cat = new Cat();  
 cat.Eat();  
 cat.Meow();  
 }  
}  
  
**Вывод:**  
Животное ест.  
Собака лает.  
Животное ест.  
Кошка мяукает.

* **Множественное наследование (через интерфейсы):** Классы C# не поддерживают множественное наследование от других классов. Однако множественное наследование может быть достигнуто путем реализации нескольких интерфейсов.5

using System;  
  
interface IWalkable { void Walk(); }  
interface ISwimmable { void Swim(); }  
  
public class Duck : IWalkable, ISwimmable // Утка может ходить и плавать  
{  
 public void Walk() { Console.WriteLine("Утка ходит."); }  
 public void Swim() { Console.WriteLine("Утка плавает."); }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Duck duck = new Duck();  
 duck.Walk();  
 duck.Swim();  
 }  
}  
  
Вывод:  
Утка ходит.  
Утка плавает.

#### Поведение конструкторов при наследовании

При создании объекта производного класса сначала вызывается конструктор базового класса, а затем конструктор производного класса.23 Можно явно вызвать конструктор базового класса из конструктора производного класса с помощью ключевого слова

base().23

using System;  
  
public class BaseClass  
{  
 public BaseClass() { Console.WriteLine("Вызван конструктор базового класса"); }  
 public BaseClass(string message) { Console.WriteLine($"Вызван конструктор базового класса с: {message}"); }  
}  
  
public class DerivedClass : BaseClass  
{  
 public DerivedClass() : base() { Console.WriteLine("Вызван конструктор производного класса"); }  
 public DerivedClass(string message) : base(message) { Console.WriteLine("Вызван конструктор производного класса с сообщением"); }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Console.WriteLine("Создание DerivedClass без параметров:");  
 DerivedClass d1 = new DerivedClass();  
 // Вывод:  
 // Вызван конструктор базового класса  
 // Вызван конструктор производного класса  
  
 Console.WriteLine("\nСоздание DerivedClass с параметром:");  
 DerivedClass d2 = new DerivedClass("Привет");  
 // Вывод:  
 // Вызван конструктор базового класса с: Привет  
 // Вызван конструктор производного класса с сообщением  
 }  
}

Вывод:

Создание DerivedClass без параметров:  
Вызван конструктор базового класса  
Вызван конструктор производного класса  
  
Создание DerivedClass с параметром:  
Вызван конструктор базового класса с: Привет  
Вызван конструктор производного класса с сообщением

### 5.2. Полиморфизм: Сила "многих форм"

**Полиморфизм**, что означает "многообразие форм" на греческом языке, является третьим столпом объектно-ориентированного программирования, после инкапсуляции и наследования.3 Он позволяет объектам принимать множество форм, улучшая гибкость и повторное использование кода.2 Полиморфизм достигается через перегрузку методов и переопределение методов.4

#### Перегрузка методов: Полиморфизм времени компиляции

**Перегрузка методов** позволяет определять несколько методов в одном классе, которые имеют одно и то же имя, но разные списки параметров (сигнатуры). Компилятор C# автоматически определяет, какой из перегруженных методов вызвать во время компиляции, основываясь на количестве, типах и порядке аргументов, предоставленных при вызове метода.

**Правила перегрузки методов:**

* Все перегруженные методы должны иметь **одно и то же имя**.
* Они должны отличаться по **количеству параметров**, **типам параметров** или **порядку параметров**.
* **Тип возвращаемого значения не имеет значения** для перегрузки; он не используется компилятором для различения перегруженных методов.

using System;  
  
public class Calculator  
{  
 public int Add(int a, int b) { return a + b; } // Перегрузка 1  
 public double Add(double a, double b) { return a + b; } // Перегрузка 2 (разные типы параметров)  
 public int Add(int a, int b, int c) { return a + b + c; } // Перегрузка 3 (разное количество параметров)  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Calculator calc = new Calculator();  
 int result1 = calc.Add(5, 3); // Вызывает Add(int, int)  
 double result2 = calc.Add(5.5, 3.2); // Вызывает Add(double, double)  
 int result3 = calc.Add(5, 3, 2); // Вызывает Add(int, int, int)  
  
 Console.WriteLine($"Результат 1: {result1}");  
 Console.WriteLine($"Результат 2: {result2}");  
 Console.WriteLine($"Результат 3: {result3}");  
 }  
}

**Вывод:**

Результат 1: 8  
Результат 2: 8.7  
Результат 3: 10

#### Переопределение методов: Полиморфизм времени выполнения

**Переопределение методов** позволяет производному классу предоставить новую реализацию для метода, унаследованного от его базового класса.3

* **Ключевые слова virtual и override:**
  + Метод базового класса должен быть помечен ключевым словом virtual, чтобы его можно было переопределить в производных классах.3
  + Метод в производном классе, который переопределяет метод базового класса, должен использовать ключевое слово override.3

using System;  
using System.Collections.Generic;  
  
public class Shape  
{  
 public virtual void Draw() // Может быть переопределен  
 {  
 Console.WriteLine("Выполнение базовых задач рисования.");  
 }  
}  
  
public class Circle : Shape  
{  
 public override void Draw() // Переопределяет базовый метод  
 {  
 Console.WriteLine("Рисуем круг.");  
 base.Draw(); // Может вызвать базовую реализацию  
 }  
}  
  
public class Rectangle : Shape  
{  
 public override void Draw()  
 {  
 Console.WriteLine("Рисуем прямоугольник.");  
 base.Draw();  
 }  
}  
  
public class Triangle : Shape  
{  
 public override void Draw()  
 {  
 Console.WriteLine("Рисуем треугольник.");  
 base.Draw();  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 // Полиморфный список  
 var shapes = new List<Shape>  
 {  
 new Rectangle(),  
 new Triangle(),  
 new Circle()  
 };  
  
 foreach (var shape in shapes)  
 {  
 shape.Draw(); // Вызывается переопределенный метод в зависимости от типа объекта  
 }  
 }  
}

Вывод:

Рисуем прямоугольник.  
Выполнение базовых задач рисования.  
Рисуем треугольник.  
Выполнение базовых задач рисования.  
Рисуем круг.  
Выполнение базовых задач рисования.

Решение о том, какой метод вызвать, принимается во время выполнения программы, основываясь на *фактическом типе объекта*, а не на объявленном типе переменной.3

#### Ключевое слово new (скрытие метода) против override

Ключевое слово new используется для **скрытия** (shadowing) члена базового класса в производном классе.49 Это означает, что производный класс объявляет член с тем же именем, что и член в базовом классе, эффективно скрывая его.

Ключевое отличие заключается в том, что если используется new, вызываемый метод зависит от *объявленного типа переменной* (решение во время компиляции), а не от типа объекта во время выполнения.51 Это может привести к путанице и ошибкам, если

new используется там, где предполагалось override.

using System;  
  
public class BasePrinter  
{  
 public void PrintMessage() { Console.WriteLine("Сообщение базового принтера"); }  
 public virtual void PrintVirtualMessage() { Console.WriteLine("Виртуальное сообщение базового принтера"); }  
}  
  
public class DerivedPrinter : BasePrinter  
{  
 public new void PrintMessage() { Console.WriteLine("Новое сообщение производного принтера"); } // Скрывает BasePrinter.PrintMessage  
 public override void PrintVirtualMessage() { Console.WriteLine("Переопределенное сообщение производного принтера"); } // Переопределяет BasePrinter.PrintVirtualMessage  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 Console.WriteLine("--- Доступ через ссылку на производный класс ---");  
 DerivedPrinter dp = new DerivedPrinter();  
 dp.PrintMessage(); // Вывод: Новое сообщение производного принтера  
 dp.PrintVirtualMessage(); // Вывод: Переопределенное сообщение производного принтера  
  
 Console.WriteLine("\n--- Доступ через ссылку на базовый класс ---");  
 BasePrinter bp = new DerivedPrinter(); // Ссылка на базовый класс  
 bp.PrintMessage(); // Вывод: Сообщение базового принтера (!!!) - Вызван скрытый метод  
 bp.PrintVirtualMessage(); // Вывод: Переопределенное сообщение производного принтера - Вызван переопределенный метод  
 }  
}

Вывод:

--- Доступ через ссылку на производный класс ---  
Новое сообщение производного принтера  
Переопределенное сообщение производного принтера  
  
--- Доступ через ссылку на базовый класс ---  
Сообщение базового принтера  
Переопределенное сообщение производного принтера

**Таблица: Сравнение new и override**

|  |  |  |
| --- | --- | --- |
| Критерий | override (Переопределение) | new (Скрытие) |
| **Ключевое слово** | override 47 | new 47 |
| **Назначение** | Предоставляет новую реализацию метода, уже определенного в базовом классе.47 | Указывает компилятору использовать реализацию дочернего класса, эффективно скрывая реализацию родительского класса.47 |
| **Требование к базовому методу** | Базовый метод должен быть помечен как virtual или abstract.52 | Базовый метод может быть любым (не обязательно virtual). Если new опущено, компилятор выдаст предупреждение, но будет вести себя как при использовании new.52 |
| **Разрешение вызова** | Во время выполнения (runtime polymorphism). CLR определяет фактический тип объекта.46 | Во время компиляции (compile-time). Компилятор определяет метод на основе типа ссылки.47 |
| **Поведение при полиморфном доступе** | Вызывается реализация производного класса, даже если доступ осуществляется через ссылку на базовый тип.3 | Вызывается реализация базового класса, если доступ осуществляется через ссылку на базовый тип. Вызывается реализация производного класса, если доступ осуществляется через ссылку на производный тип.51 |
| **Поддержка полиморфизма** | Да, полностью поддерживает полиморфизм.3 | Нет, не поддерживает полиморфизм.51 |
| **Потенциальные риски** | Меньше рисков, если используется правильно; обеспечивает предсказуемое полиморфное поведение. | Высокие риски из-за нарушения ожиданий полиморфизма; может привести к трудноуловимым ошибкам. |

### 5.3. Абстрактные Классы

**Абстрактные классы** являются фундаментальным элементом ООП в C#, позволяя определять общие контракты и частично реализованную функциональность в иерархиях классов.

* **Абстрактный класс:** Это класс, который предназначен исключительно для использования в качестве базового класса для других классов и не может быть инстанцирован напрямую.9 Он служит в качестве неполного чертежа, который должен быть завершен производными классами. Абстрактный класс может содержать как абстрактные (без реализации), так и неабстрактные (с полной реализацией) члены.54 Важное ограничение: абстрактный класс не может быть модифицирован ключевым словом  
  sealed, поскольку sealed предотвращает наследование, что противоречит основному назначению abstract класса, которое требует наследования.54
* **Абстрактный метод/свойство:** Это член класса, который не содержит реализации (не имеет тела) и должен быть реализован в любом неабстрактном классе, производном от абстрактного класса.54 Абстрактные методы неявно являются  
  virtual, что означает, что они участвуют в механизме полиморфизма времени выполнения.54 Объявления абстрактных методов разрешены только в абстрактных классах.54

**Пример абстрактного класса и метода:**

using System;  
  
// Абстрактный класс Shape  
public abstract class Shape  
{  
 public string Color { get; set; }  
  
 // Конструктор абстрактного класса  
 protected Shape(string color)  
 {  
 Color = color;  
 Console.WriteLine($"Создана фигура цвета {color}.");  
 }  
  
 // Абстрактный метод (без реализации), должен быть реализован в производных классах  
 public abstract double CalculateArea();  
  
 // Неабстрактный метод (с реализацией)  
 public void DisplayColor()  
 {  
 Console.WriteLine($"Цвет фигуры: {Color}");  
 }  
}  
  
// Конкретный класс Square, производный от Shape  
public class Square : Shape  
{  
 private double \_side;  
  
 public Square(double side, string color) : base(color) // Вызов конструктора базового класса  
 {  
 \_side = side;  
 }  
  
 // Обязательная реализация абстрактного метода CalculateArea()  
 public override double CalculateArea()  
 {  
 return \_side \* \_side;  
 }  
}  
  
// Конкретный класс Circle, производный от Shape  
public class Circle : Shape  
{  
 private double \_radius;  
  
 public Circle(double radius, string color) : base(color)  
 {  
 \_radius = radius;  
 }  
  
 public override double CalculateArea()  
 {  
 return Math.PI \* \_radius \* \_radius;  
 }  
}  
  
public class Program  
{  
 public static void Main(string args)  
 {  
 // Shape myShape = new Shape("Красный"); // Ошибка компиляции: нельзя создать экземпляр абстрактного класса  
  
 Square square = new Square(5, "Синий");  
 square.DisplayColor();  
 Console.WriteLine($"Площадь квадрата: {square.CalculateArea()}");  
  
 Circle circle = new Circle(3, "Зеленый");  
 circle.DisplayColor();  
 Console.WriteLine($"Площадь круга: {circle.CalculateArea():F2}");  
 }  
}

Вывод:

Создана фигура цвета Синий.  
Цвет фигуры: Синий  
Площадь квадрата: 25  
Создана фигура цвета Зеленый.  
Цвет фигуры: Зеленый  
Площадь круга: 28.27

### 5.4. Класс System.Object и его Методы

Класс System.Object является фундаментальным типом в.NET Framework и в C#. Он служит базовым классом для всех других типов, что означает, что каждый тип в C#, включая пользовательские классы и структуры, неявно наследует от System.Object.3

System.Object определяет общий набор членов, которые поддерживаются каждым типом в.NET Framework.57 Среди этих унаследованных членов есть четыре ключевых метода:

* Equals, GetHashCode, GetType и ToString. Некоторые из этих членов являются виртуальными, что означает, что их реализация по умолчанию может быть переопределена в производных классах для обеспечения специфического поведения, соответствующего логике пользовательского типа.57

#### Переопределение ToString(), Equals(), GetHashCode()

Переопределение этих

#### Источники

1. Beginner's Guide to To C# Access Modifiers (With Code Examples ..., дата последнего обращения: июля 17, 2025, <https://zerotomastery.io/blog/c-sharp-access-modifiers/>
2. What is Object-Oriented Programming? Definition, Pros, Cons, and ..., дата последнего обращения: июля 17, 2025, <https://pg-p.ctme.caltech.edu/blog/coding/what-is-object-oriented-programming>
3. Polymorphism - C# | Microsoft Learn, дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/polymorphism>
4. C# Polymorphism - GeeksforGeeks, дата последнего обращения: июля 17, 2025, <https://www.geeksforgeeks.org/c-sharp/c-polymorphism/>
5. Inheritance in C# - TutorialsTeacher, дата последнего обращения: июля 17, 2025, <https://www.tutorialsteacher.com/csharp/inheritance>
6. C# Inheritance - GeeksforGeeks, дата последнего обращения: июля 17, 2025, <https://www.geeksforgeeks.org/c-sharp/c-sharp-inheritance/>
7. C# Class and Object (With Examples) - Programiz, дата последнего обращения: июля 17, 2025, <https://www.programiz.com/csharp-programming/class-objects>
8. C# Class and Objects - Tutorials Teacher, дата последнего обращения: июля 17, 2025, <https://www.tutorialsteacher.com/csharp/csharp-class>
9. C# Namespaces with Examples - Tutlane, дата последнего обращения: июля 17, 2025, <https://www.tutlane.com/tutorial/csharp/csharp-namespaces-with-examples>
10. Namespaces in C# | C# Tutorial - upGrad KnowledgeHut, дата последнего обращения: июля 17, 2025, <https://www.knowledgehut.com/tutorials/csharp/csharp-namespaces>
11. Creating Class Library in Visual C# - C# Corner, дата последнего обращения: июля 17, 2025, <https://www.c-sharpcorner.com/UploadFile/61b832/creating-class-library-in-visual-C-Sharp/>
12. Namespaces - C# language specification - Learn Microsoft, дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/namespaces>
13. abstract keyword - C# reference | Microsoft Learn, дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract>
14. The using directive: Import types from a namespace - C# reference ..., дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-directive#using-alias-directive>
15. System.Object.Equals method - .NET | Microsoft Learn, дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/dotnet/fundamentals/runtime-libraries/system-object-equals>
16. The using directive: Import types from a namespace - C# reference ..., дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/using-directive#using-static-directive>
17. Expression-bodied members - C# | Microsoft Learn, дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/expression-bodied-members>
18. What Are Access Modifiers In C#, дата последнего обращения: июля 17, 2025, <https://www.c-sharpcorner.com/UploadFile/puranindia/what-are-access-modifiers-in-C-Sharp/>
19. Top-level statements tutorial - C# | Microsoft Learn, дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/tutorials/top-level-statements>
20. Exploring Top Level Statements - Programs Without Main Methods in C#, дата последнего обращения: июля 17, 2025, <https://www.c-sharpcorner.com/article/exploring-top-level-statements-programs-without-main-methods-in-c-sharp/>
21. Abstract Class in .NET C#: Syntax, Usage and Example - C# Corner, дата последнего обращения: июля 17, 2025, <https://www.c-sharpcorner.com/article/abstract-class-in-net-c-sharp-syntax-usage-and-example/>
22. Object and Collection Initializers - C# | Microsoft Learn, дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/object-and-collection-initializers>
23. Constructor And Destructors In C# - C# Corner, дата последнего обращения: июля 17, 2025, <https://www.c-sharpcorner.com/article/constructor-and-destructors-in-C-Sharp/>
24. What are destructors in C#?, дата последнего обращения: июля 17, 2025, <https://www.c-sharpcorner.com/interview-question/what-are-destructors-in-c-sharp2>
25. static modifier - C# reference | Microsoft Learn, дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static>
26. What are readonly structs in C#? - Educative.io, дата последнего обращения: июля 17, 2025, <https://www.educative.io/answers/what-are-readonly-structs-in-c-sharp>
27. Deconstructing tuples and other types - C# | Microsoft Learn, дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/functional/deconstruct>
28. C# struct (With Examples) - Programiz, дата последнего обращения: июля 17, 2025, <https://www.programiz.com/csharp-programming/struct>
29. Structs - C# language specification | Microsoft Learn, дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/structs>
30. readonly keyword - C# reference | Microsoft Learn, дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/readonly>
31. Overriding Static Members in C# | Coding for Smarties - WordPress.com, дата последнего обращения: июля 17, 2025, <https://codingforsmarties.wordpress.com/2018/02/19/overriding-static-members-in-c/>
32. Nullable value types - C# reference | Microsoft Learn, дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/nullable-value-types>
33. Inheriting from Generics: Set the Datatype ... Or Not - Visual Studio Magazine, дата последнего обращения: июля 17, 2025, <https://visualstudiomagazine.com/blogs/tool-tracker/2018/03/inheriting-from-generics.aspx>
34. C# | Generics - Introduction - GeeksforGeeks, дата последнего обращения: июля 17, 2025, <https://www.geeksforgeeks.org/c-sharp/c-sharp-generics-introduction/>
35. Type Conversions And The Implicit and Explicit Operators In C# - C# ..., дата последнего обращения: июля 17, 2025, <https://giannisakritidis.com/blog/Type-Conversion-Implicit-Explicit-Operators/>
36. Using Generics In C# - C# Corner, дата последнего обращения: июля 17, 2025, <https://www.c-sharpcorner.com/UploadFile/84c85b/using-generics-with-C-Sharp/>
37. C# Interface - GeeksforGeeks, дата последнего обращения: июля 17, 2025, <https://www.geeksforgeeks.org/c-sharp/c-sharp-interface/>
38. value types and reference types in C# - Microsoft Q&A - Learn Microsoft, дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/answers/questions/1407356/value-types-and-reference-types-in-c>
39. ICloneable.Clone Method (System) - Learn Microsoft, дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/dotnet/api/system.icloneable.clone?view=net-9.0>
40. C# Interface: Define, Implement and Use (With Examples) - Tutorials Teacher, дата последнего обращения: июля 17, 2025, <https://www.tutorialsteacher.com/csharp/csharp-interface>
41. Exploring Default Interface Methods in C#: Advanced Approaches and Best Practices | by Anderson Godoy | Medium, дата последнего обращения: июля 17, 2025, <https://medium.com/@anderson.buenogod/exploring-default-interface-methods-in-c-advanced-approaches-and-best-practices-49f9a34f086c>
42. Create a .NET class library using Visual Studio Code - .NET ..., дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/dotnet/core/tutorials/library-with-visual-studio-code>
43. Add or remove imported namespaces (Visual Basic) - Visual Studio (Windows), дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/visualstudio/ide/how-to-add-or-remove-imported-namespaces-visual-basic?view=vs-2022>
44. Should we inherit from base class and implement interface in this scenarios?, дата последнего обращения: июля 17, 2025, <https://softwareengineering.stackexchange.com/questions/386718/should-we-inherit-from-base-class-and-implement-interface-in-this-scenarios>
45. Generic Methods and Inheritance in C# | by Røckŷ Käñís - Medium, дата последнего обращения: июля 17, 2025, <https://medium.com/@rckks/generic-methods-and-inheritance-in-c-444b6134d07c>
46. Runtime Polymorphism: Method Overriding - Tutorials Teacher, дата последнего обращения: июля 17, 2025, <https://www.tutorialsteacher.com/csharp/method-overriding>
47. c# - Difference between new and override - Stack Overflow, дата последнего обращения: июля 17, 2025, <https://stackoverflow.com/questions/1399127/difference-between-new-and-override>
48. virtual keyword - C# reference | Microsoft Learn, дата последнего обращения: июля 17, 2025, <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/virtual>
49. Value Types vs Reference Types in C# - C# Corner, дата последнего обращения: июля 17, 2025, <https://www.c-sharpcorner.com/article/value-types-vs-reference-types-in-c-sharp/>
50. Method Hiding in C# - GeeksforGeeks, дата последнего обращения: июля 17, 2025, <https://www.geeksforgeeks.org/c-sharp/method-hiding-in-c-sharp/>
51. Hide Base Class Members in C# .NET with new Keyword - C# Corner, дата последнего обращения: июля 17, 2025, <https://www.c-sharpcorner.com/article/hide-base-class-members-in-c-sharp-net-with-new-keyword/>
52. overriding - When to use new instead of override C# - Stack Overflow, дата последнего обращения: июля 17, 2025, <https://stackoverflow.com/questions/9590243/when-to-use-new-instead-of-override-c-sharp>
53. Designing Sortable Collections using IComparable - C# Corner, дата последнего обращения: июля 17, 2025, <https://www.c-sharpcorner.com/article/designing-sortable-collections-using-icomparable/>
54. Global using Directive in C# 10 - C# Corner, дата последнего обращения: июля 17, 2025, <https://www.c-sharpcorner.com/article/global-using-directive-in-c-sharp-102/>
55. Implicit and Explicit Interface Examples - C# Corner, дата последнего обращения: июля 17, 2025, <https://www.c-sharpcorner.com/UploadFile/8911c4/implicit-and-explicit-interface-examples/>
56. C# Method Overloading Examples - Beginner Guide - Codejack, дата последнего обращения: июля 17, 2025, <https://codejack.com/2024/09/c-method-overloading-examples-beginner-guide/>
57. Overriding The System.Object Class Methods - C# Corner, дата последнего обращения: июля 17, 2025, <https://www.c-sharpcorner.com/UploadFile/Ashush/overriding-the-system-object-class-methods/>