

# Struct и Record

## 1. Введение

На прошлых лекциях мы уже познакомились с **ссылочными и значимыми** типами.

- **Ссылочные типы** хранятся в куче и работают через ссылку на объект. К ним относятся все классы, например строки (`string`) и массивы (`int[]`).
- **Значимые типы** хранятся в стеке и передаются по значению (копируются). К ним относятся, например, числа (`int`, `double`, `bool`).

Для создания собственных значимых типов в C# используется ключевое слово `struct`. На самом деле все встроенные числовые типы — это именно структуры.

Кроме того, в новых версиях языка появился еще один удобный инструмент — **record**, который может быть как ссылочным, так и значимым типом (в зависимости от того, `record class` или `record struct`). В этой лекции мы подробно разберем, как работают структуры, чем они отличаются от классов, а затем познакомимся с рекордами.

## 2. Структуры (struct)

**Структура** — это пользовательский значимый тип данных.

Создается с помощью ключевого слова `struct`.

Простейший пример:

```
struct Point
{
    public int X;
    public int Y;

    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    public void Print()
    {
        Console.WriteLine($"({X}, {Y})");
    }
}
```

Использование:

```
var p1 = new Point(1, 2);
p1.Print(); // (1, 2)

var p2 = p1; // создается копия!
p2.X = 5;

p1.Print(); // (1, 2)
p2.Print(); // (5, 2)
```

Здесь видно ключевое отличие от классов — при присваивании **структура копируется**, а не передается по ссылке.

## Когда использовать

Структуры стоит применять, если:

- нужно хранить небольшой набор данных, например точку, цвет, размеры (`Point`, `Rectangle`, `Color`);
- объект **неизменяемый** (`immutable`) и копирование не приведет к проблемам;
- важна **эффективность** (структурь быстрее создаются и не требуют сборки мусора).

Классы же лучше использовать, если:

- объект большой и часто передается между методами (копирование структур может стать накладным);
- требуется наследование;
- нужна работа с полиморфизмом.

# Отличия от классов

Основные различия между `struct` и `class`:

## 1. Передача

- Классы — ссылочные типы, передаются по ссылке.
- Структуры — значимые типы, передаются по значению (копируются).

## 2. Наследование

- Классы могут наследоваться.
- Структуры **не поддерживают наследование**, но могут реализовывать интерфейсы.

## 3. Конструкторы

- В структурах нельзя явно объявлять конструктор без параметров (он создается всегда по умолчанию).
- В классах можно определять любые конструкторы.

## 4. Хранение

- Объекты классов хранятся в куче.
- Структуры, как правило, хранятся в стеке.

## 5. Null

- Ссылочные типы могут быть `null`.
- Структуры — нет, если только не использовать `Nullable<T>` или сокращение `T?`.

### 3. Проблемы при работе со структурами

Хотя структуры в C# полезны и во многих случаях эффективны, при работе с ними есть несколько важных нюансов и проблем, которые нужно учитывать.

#### Боксинг и анбоксинг

**Боксинг** — это процесс преобразования значимого типа в ссылочный. Когда мы присваиваем значение типа `struct` переменной типа `object` или интерфейсу, структуру "упаковывается" в объект в куче.

Пример:

```
int x = 42;           // структура int
object obj = x;      // боксинг: int упакован в object
```

**Анбоксинг** — это обратный процесс: извлечение значения из объекта.

```
object obj = 42;
int y = (int)obj;  // анбоксинг
```

Почему это проблема?

- Боксинг и анбоксинг создают **лишние расходы памяти и процессорного времени**.
- Каждое упакованное значение хранится отдельно в куче, что увеличивает нагрузку на сборщик мусора.

Например, в цикле:

```
for (int i = 0; i < 1000; i++)
    object obj = i; // каждый раз происходит боксинг! Это очень неэффективно
```

## Пример

Предположим, у нас есть структура, которая реализует интерфейс:

```
interface IShape
{
    double Area();
}

struct Circle : IShape
{
    public double Radius { get; set; }
    public Circle(double r) => Radius = r;

    public double Area() => Math.PI * Radius * Radius;
}
```

Теперь создадим коллекцию фигур:

```
var shapes = new List<IShape>();
for (int i = 0; i < 1_000_000; i++)
{
    shapes.Add(new Circle(i)); // ❌ здесь происходит боксинг!
}
```

## Что произошло?

- `Circle` — это `struct`.
- Но коллекция `List<IShape>` хранит элементы как ссылки (`IShape` — интерфейс).
- Чтобы положить `struct` в `List<IShape>`, оно будет "упаковано" в `object`.
- То есть каждый круг в коллекции будет скопирован в кучу.

Это приведёт к:

- лишнему выделению памяти в куче;
- дополнительной нагрузке на **сборщик мусора**;
- замедлению программы при большом количестве элементов.

# Анбоксинг

Когда мы достаём элемент:

```
foreach (var shape in shapes)
{
    var area = shape.Area(); // анбоксинг при вызове метода
}
```

Каждый вызов `shape.Area()` сопровождается **анбоксингом**, потому что упакованный `Circle` достается из `object`.

## Как исправить?

Использовать **дженерики** и хранить `struct` напрямую, без упаковки:

```
var circles = new List<Circle>(); // ✅ нет боксинга
for (int i = 0; i < 1_000_000; i++)
{
    circles.Add(new Circle(i));
}

foreach (var circle in circles)
{
    var area = circle.Area(); // вызов без анбоксинга
}
```

## Сравнение

```
const int N = 100_000_000;

// Вариант 1: List<IShape> (происходит боксинг)
var sw1 = Stopwatch.StartNew();
var shapes = new List<IShape>();
for (int i = 0; i < N; i++)
    shapes.Add(new Circle(i)); // здесь боксинг
double sum1 = 0;
foreach (var shape in shapes)
    sum1 += shape.Area(); // здесь анбоксинг
sw1.Stop();
Console.WriteLine($"List<IShape>: время = {sw1.ElapsedMilliseconds} ms");

// Вариант 2: List<Circle> (без боксинга)
var sw2 = Stopwatch.StartNew();
var circles = new List<Circle>();
for (int i = 0; i < N; i++)
    circles.Add(new Circle(i)); // без упаковки
double sum2 = 0;
foreach (var circle in circles)
    sum2 += circle.Area(); // без анбоксинга
sw2.Stop();
Console.WriteLine($"List<Circle>: время = {sw2.ElapsedMilliseconds} ms");
```

```
List<IShape>: время = 7505 ms
List<Circle>: время = 1468 ms
```

## Размер структур

Ещё одна проблема связана с размером структур:

- Структуры маленького размера (несколько полей: `int`, `double`) очень эффективны.
- Но если структура содержит много полей (например, десятки чисел или вложенные структуры), то при каждом копировании создаётся **большой объём работы**.

Пример:

```
struct BigStruct
{
    public int A, B, C, D, E, F, G, H, I, J;
}
```

При передаче такой структуры в метод создаётся **копия всех полей**. Для больших структур это может быть даже медленнее, чем работа с классами.

Поэтому общее правило:

- маленькие простые данные → `struct` ;
- большие сложные объекты → `class` .

# Проблемы с копированием

Когда мы работаем со структурами, нужно помнить, что это **значимые типы**. А значит, при копировании переменной структуры создаётся её **полная копия**, а не ссылка, как у классов.

Из-за этого иногда возникают неожиданные ситуации при работе со **свойствами структур**:

## Пример 1

```
struct Point
{
    public int X { get; set; }
    public int Y { get; set; }
}

var list = new List<Point>();
list.Add(new Point { X = 1, Y = 2 });
// Берём элемент списка
var p = list[0];
p.X = 10; // меняем свойство
Console.WriteLine(list[0].X); // ❌ всё равно выведет 1, а не 10
```

Почему так произошло?

Потому что `list[0]` вернул **копию структуры**, а мы изменили её свойство у копии, а не у оригинала.

Чтобы реально изменить элемент внутри списка, нужно записать его обратно:

```
p.X = 10;
list[0] = p; // теперь X изменится
```

## Пример 2

Если структура используется как **свойство другого объекта**, то при доступе к нему через `get` возвращается тоже **копия**.

```
class Container
{
    public Point P { get; set; } = new Point { X = 1, Y = 2 };
}
```

```
var c = new Container();
c.P.X = 5; // ❌ Ошибка: нельзя присвоить, потому что P возвращает копию
```

Чтобы изменить координату, придётся переприсвоить всё свойство целиком:

```
var tmp = c.P;
tmp.X = 5;
c.P = tmp; // ✅ теперь значение изменится
```

## 4. Записи (record)

**Record** — это специальный тип данных в C#.

Записи (records) — это **ссыпочные типы**, как и классы, но их главная особенность заключается в том, что они ориентированы на **работу с данными**.

Основные особенности:

- По умолчанию считаются **иммутабельными** (неизменяемыми).
- Для них автоматически создаются методы `Equals` , `GetHashCode` И `ToString` .
- Сравнение записей идёт **по значению**, а не по ссылке (в отличие от классов).

Таким образом, запись можно рассматривать как удобный способ хранить и сравнивать данные.

## Синтаксис

Запись можно объявить так же, как класс:

```
public record Person(string Name, int Age);
```

Это короткая форма, которая автоматически:

- Создаёт свойства `Name` и `Age` с `init`-сеттером (их можно задавать только при создании объекта).
- Реализует методы `Equals` и `GetHashCode` для сравнения по значениям.
- Переопределяет `ToString`, чтобы выводить данные.

Пример использования:

```
var p1 = new Person("Alice", 25);
var p2 = new Person("Alice", 25);

Console.WriteLine(p1 == p2); // true (сравнение по значениям)
Console.WriteLine(p1);      // Person { Name = Alice, Age = 25 }
```

# Виды записей

В C# есть несколько способов объявления записей:

## 1. Позиционные записи

Удобны, если нужно быстро описать тип данных:

```
public record Point(int X, int Y);
var p = new Point(10, 20);
```

## 2. Записи с явными свойствами

Если нужны дополнительные методы или более гибкая логика:

```
public record Car
{
    public string Model { get; init; }
    public int Year { get; init; }
}
```

## 3. record struct

Можно создавать **значимые типы** (struct) с теми же возможностями:

```
public record struct Point(int X, int Y);
```

## 4. readonly record struct

Полностью неизменяемый значимый тип:

```
public readonly record struct Point(int X, int Y);
```

## Оператор `with`

Одной из ключевых возможностей **record** является оператор `with`, который позволяет создавать копии объектов с изменением отдельных свойств.

Это особенно удобно при работе с **иммутабельными объектами**, где нельзя просто изменить свойство.

Пример:

```
public record Person(string Name, int Age);

var p1 = new Person("Alice", 25);

// создаём копию с изменённым возрастом
var p2 = p1 with { Age = 26 };

Console.WriteLine(p1); // Person { Name = Alice, Age = 25 }
Console.WriteLine(p2); // Person { Name = Alice, Age = 26 }
```

Особенности:

- Создаётся **новый объект**, а не изменяется старый.
- Все остальные свойства сохраняются без изменений.
- Работает только с типами, у которых есть `init`-свойства (а у `record` они такие по умолчанию).

Пример с вложенными записями:

```
public record Address(string City, string Street);
public record Person(string Name, int Age, Address Address);

var p1 = new Person("Alice", 25, new Address("New York", "5th Avenue"));

// копия с изменением только города
var p2 = p1 with { Address = p1.Address with { City = "Boston" } };

Console.WriteLine(p1);
// Person { Name = Alice, Age = 25, Address = Address { City = New York, Street = 5th Avenue } }

Console.WriteLine(p2);
// Person { Name = Alice, Age = 25, Address = Address { City = Boston, Street = 5th Avenue } }
```

## Сравнение: Class vs Struct vs Record

Характеристика	Class	Struct	Record
Тип	Ссылочный	Значимый	Ссылочный / Значимый
Передача	По ссылке	По значению	По ссылке / значению
Сравнение	По ссылке (если не переопределён Equals )	По значению (если не переопределён Equals )	По значению (по умолчанию)
Наследование	Да	Нет	Да ( record class )
Использование	Сложные объекты	Лёгкие данные	модели данных

# Практическое задание

## Что нужно сделать:

1. Продолжайте работу над проектом **TodoList**.

Теперь программа должна поддерживать **несколько профилей пользователей**.

2. **Модифицируйте класс Profile :**

- Добавьте поля:
  - Login ;
  - Password ;
  - Id (уникальный идентификатор, например Guid ).
- Свойства FirstName , LastName , BirthYear оставить без изменений.

3. **Хранение пользователей:**

- Все профили должны сохраняться в файле profile.csv .
- Формат:

```
Id;Login;Password;FirstName;LastName;BirthYear
```

- При запуске программы:
  - Если папка и файл не существуют — создать.
  - Если файл существует — загрузить все профили в память.

4. **Хранение текущего профиля:**

- В AppInfo должен храниться список профилей List<Profile> .
- Добавьте в AppInfo переменную CurrentProfileId .
- При входе в профиль в неё сохраняется Id текущего пользователя.
- Все действия (просмотр/создание/изменение заметок) должны выполняться только для профиля с этим Id .

## 5. Работа с профилями:

- При запуске программы выводить сообщение:

```
Войти в существующий профиль? [y/n]
```

- Если выбран у :

- Пользователь вводит логин и пароль.
- Если они корректны, устанавливается соответствующий CurrentProfileId , загружается список заметок этого пользователя.

- Если выбран n :

- Создать новый профиль:
  - Ввести логин, пароль, имя, фамилию и год рождения.
  - Сгенерировать уникальный Id ( Guid.NewGuid() ).
  - Сохранить профиль в profile.csv .
  - Установить CurrentProfileId на этот новый Id .

## 6. Хранение заметок:

- Все заметки должны храниться в словаре:
  - Ключ: Id пользователя;
  - Значение: список заметок ( List<TodoItem> ).
- Для каждого пользователя заметки сохраняются в отдельный файл:

```
todos_<Id>.csv
```

- Формат записи:

```
Index;Text;Status;LastUpdate
```

- При входе в профиль загрузить заметки из его файла.
- Если файла ещё нет — создать.

**7. Измените команды `add`, `update`, `delete`, `status`:**

- Теперь они работают только с задачами профиля, у которого `Id == CurrentProfileId`.
- Все изменения должны сразу сохраняться в файл соответствующего пользователя.

**8. Добавьте новый флаг к команде `profile`:**

- Флаг `--out` или `-o`.
- При его использовании выполняется **выход из текущего профиля**:
  - `CurrentProfileId` обнуляется (или становится `null`),
  - программа возвращается в состояние выбора профиля

**9. Учтите работу с `Undo/Redo`:**

- При каждом входе в новый профиль необходимо **обнулять стеки `undoStack` и `redoStack`**, чтобы команды отката/возврата не переносились между пользователями.

10. Обновите **README.md** — добавьте описание новых возможностей программы.

11. Сделайте `push` изменений в GitHub.