

# объектно ориентированное программирование

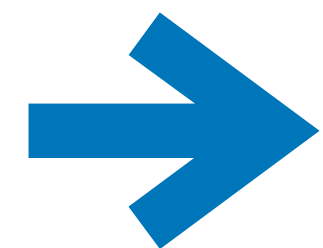
ОСНОВЫ

**ВВЕДЕНИЕ**

# структурное программирование

## парадигмы

```
0  VAR i
1  SET i 1
2  PRINT i
3  INC i
4  JIFLS i 10 2
```



```
for (var i = 1; i < 10; i++)
{
    Console.WriteLine(i);
}
```

# процедурное программирование

## парадигмы

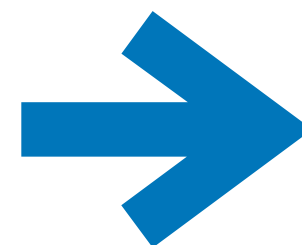
```
var probability1 = Random.Shared.NextDouble();
var coefficient1 = probability1 > 0.5 ? 10d : 0;

var probability2 = Random.Shared.NextDouble();
var coefficient2 = probability2 > 0.5 ? 10d : 0;

var input = Console.ReadLine();
var value = input is null ? 0 : int.Parse(input);

var result = value * coefficient1 / coefficient2;

var message = $"result: {result}";
Console.WriteLine(message);
```



```
var coefficient1 = CalculateCoefficient();
var coefficient2 = CalculateCoefficient();

var value = ReadValue();

var result = value * coefficient1 / coefficient2;
OutputResult(result);

static double CalculateCoefficient()
{
    var probability = Random.Shared.NextDouble();
    return probability > 0.5 ? 10d : 0;
}

static int ReadValue()
{
    var input = Console.ReadLine();
    return input is null ? 0 : int.Parse(input);
}

static void OutputResult(double result)
{
    var message = $"result: {result}";
    Console.WriteLine(message);
}
```

# **мы научимся**

## **в рамках курса**

- объектно-ориентированному проектированию
- C#
- писать объектно-ориентированный код на объектно-ориентированном языке

# ОСНОВНЫЕ КОНЦЕПЦИИ ООП

# инкапсуляция

## основные концепции ООП

- **тип – шаблон**, описывающий какие данные и методы имеет объект
- **объект – экземпляр** типа, существующий во время выполнения кода, занимающий память

# инкапсуляция

## основные концепции ООП

```
public class SampleClass
{
    public int _firstField;
    public int _secondField;
}
```

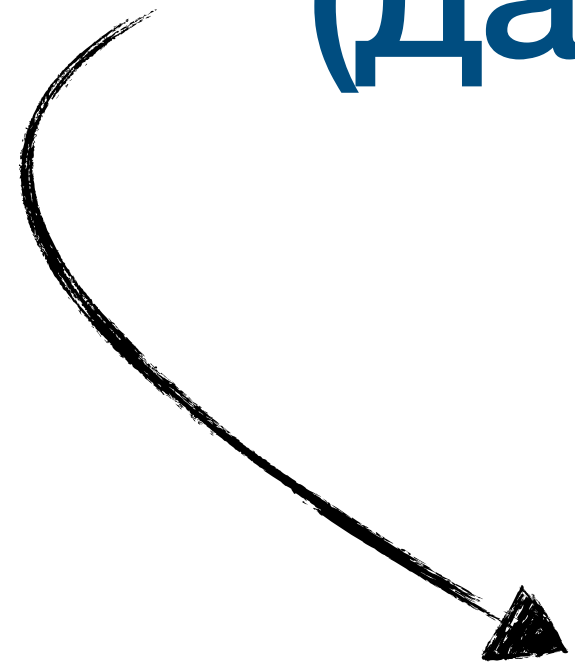
- **ссылочные** типы
- данные объекта хранятся **на куче**
- на стеке хранится только ссылка

```
public struct SampleStruct
{
    public int _firstField;
    public int _secondField;
}
```

- **значимые** типы
- данные хранятся там, где находится объект структуры



**принцип объединения атрибутов и поведений  
(данных и методов) в рамках одного типа**



**инкапсуляция**

# инкапсуляция

## основные концепции ООП

- улучшает структурированность кода
- локализует логику относительно данных которые она обрабатывает
- упрощает процесс изменения кода
- уменьшает возможность внести изменения ломающие инвариант типа

# ИНКАПСУЛЯЦИЯ

## ОСНОВНЫЕ КОНЦЕПЦИИ ООП

```
public class BankAccount
{
    public decimal _value;

    public BankAccount(decimal value)
    {
        _value = value;
    }

    public bool TryAccrue(decimal amount)
    {
        if (amount > _value)
            return false;

        _value -= amount;
        return true;
    }
}
```

**набор правил, определяющих  
корректное состояние данных**



**инвариант данных**

**набор данных, их инварианта и поведений,  
позволяющих изменять эти данные согласно их  
инварианту**



**инвариант типа**

# СОКРЫТИЕ

## ОСНОВНЫЕ КОНЦЕПЦИИ ООП

```
var acc = new BankAccount(0);  
acc._value = 1000;  
  
if (acc.TryAccrue(100))  
{  
    Console.WriteLine("Вы совершили финансовую махинацию!");  
}
```

# сокрытие

## основные концепции ООП

- ограничить доступ к данным и методам позволяют модификаторы доступа
  - `public` – доступ есть **отовсюду**
  - `private` – доступ есть только **внутри типа**
  - `protected` – как `private`, но доступ также есть **у дочерних типов**
  - `internal` – доступ есть только в рамках **текущей сборки** (модуля)

# сокрытие

## основные концепции ООП

- ограничить доступ к данным и методам позиции доступа
- `public` – доступ есть отовсюду
- `private` – доступ есть только внутри типа
- `protected` – как `private`, но доступ также есть у дочерних типов
- `internal` – доступ есть только в рамках текущей сборки (модуля)

использование `internal`  
для обеспечение  
сокрытия в бизнес  
логике свидетельствует  
о не правильно  
выстроенной  
абстракции



# СОКРЫТИЕ

## ОСНОВНЫЕ КОНЦЕПЦИИ ООП

```
public class BankAccount
{
    private decimal _value;

    public BankAccount(decimal value)
    {
        _value = value;
    }

    public bool TryAccrue(decimal amount)
    {
        if (amount > _value)
            return false;

        _value -= amount;
        return true;
    }
}
```

код не скомпилируется

```
var acc = new BankAccount(0);
acc._value = 1000;
```

# инкапсуляция + сокрытие

## основные концепции ООП

- объединяя инкапсуляцию и сокрытие мы можем гарантировать соблюдение инварианта типа
- объединяя инкапсуляцию и сокрытие мы получаем **абстракцию**
  - представляет собой “**черный ящик**” с точки зрения реализации
  - позволяет писать **более простой код** на более высоких уровнях абстракции

# СВОЙСТВА

```
public class BankAccount
{
    private decimal _value;

    public decimal GetValue() => _value;

    public bool TryAccrue(decimal amount)
    {
        if (amount > _value)
            return false;

        _value -= amount;
        return true;
    }
}
```

# СВОЙСТВА

- Без свойств
  - нужно писать отдельный метод получения данных
  - нужно содержать отдельное поле
- Со свойствами
  - Не нужно **лишних** методов
  - Описывают **хранимые данные** и **способы работы** с ними
  - Сохраняют “**семантику поля**”

# АВТО-СВОЙСТВА

## СВОЙСТВА

```
public class BankAccount
{
    public decimal Value { get; private set; }

    public bool TryAccrue(decimal amount)
    {
        if (amount > Value)
            return false;

        Value -= amount;
        return true;
    }
}
```

# ВЫЧИСЛЯЕМЫЕ СВОЙСТВА

## свойства

```
public class ShoppingCart
{
    public decimal TotalCost ⇒ Items.Sum(item ⇒ item.Cost);
}
```

# КОМПОЗИЦИЯ

## основные концепции ООП

```
public class SampleClass
{
    public int _firstField;
    public int _secondField;
}
```

**способ создания одних типов на основе других,  
при помощи хранения объектов одних типов в  
объектах других типов**



**КОМПОЗИЦИЯ**



# агрегация

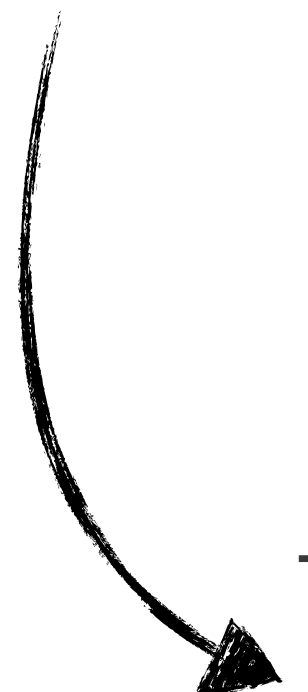
## ВИДЫ КОМПОЗИЦИИ

```
public struct Point2D
{
    public Point2D(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }

    public double Y { get; }
}
```

```
var point = new Point2D(1, 2);
```



```
{
    "X": 1,
    "Y": 2
}
```

**ВИД КОМПОЗИЦИИ, ПОДРАЗУМЕВАЮЩИЙ, ЧТО  
ХРАНИМЫЕ ЗНАЧЕНИЯ ПОЛУЧАЮТСЯ ИЗВНЕ**



**агрегация**

# ассоциация

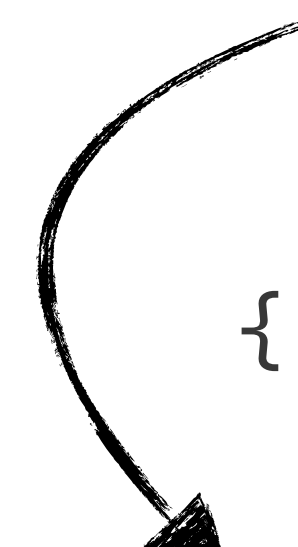
## ВИДЫ КОМПОЗИЦИИ

```
public class Car
{
    private readonly Engine _engine;
    private readonly Wheel[] _wheels;

    public Car()
    {
        _engine = new Engine("V8");

        _wheels = Enumerable
            .Range(0, 4)
            .Select(index => new Wheel(index))
            .ToArray();
    }
}
```

```
var car = new Car();
```



```
{
    "_engine": {
        "Name": "V8"
    },
    "_wheels": [
        {"Index": 0},
        {"Index": 1},
        {"Index": 2},
        {"Index": 3}
    ]
}
```

**вид композиции, подразумевающий, что  
хранимые значения создаются самим объектом**



**ассоциация**

# агрегация vs ассоциация

## композиция

- агрегация
  - более низкая связанность между вложенными и содержащими типами
  - мы не определяем как создаются вложенные объекты
- ассоциация
  - более высокая связанность между вложенными и содержащими типами
  - мы сами создаём вложенные объекты

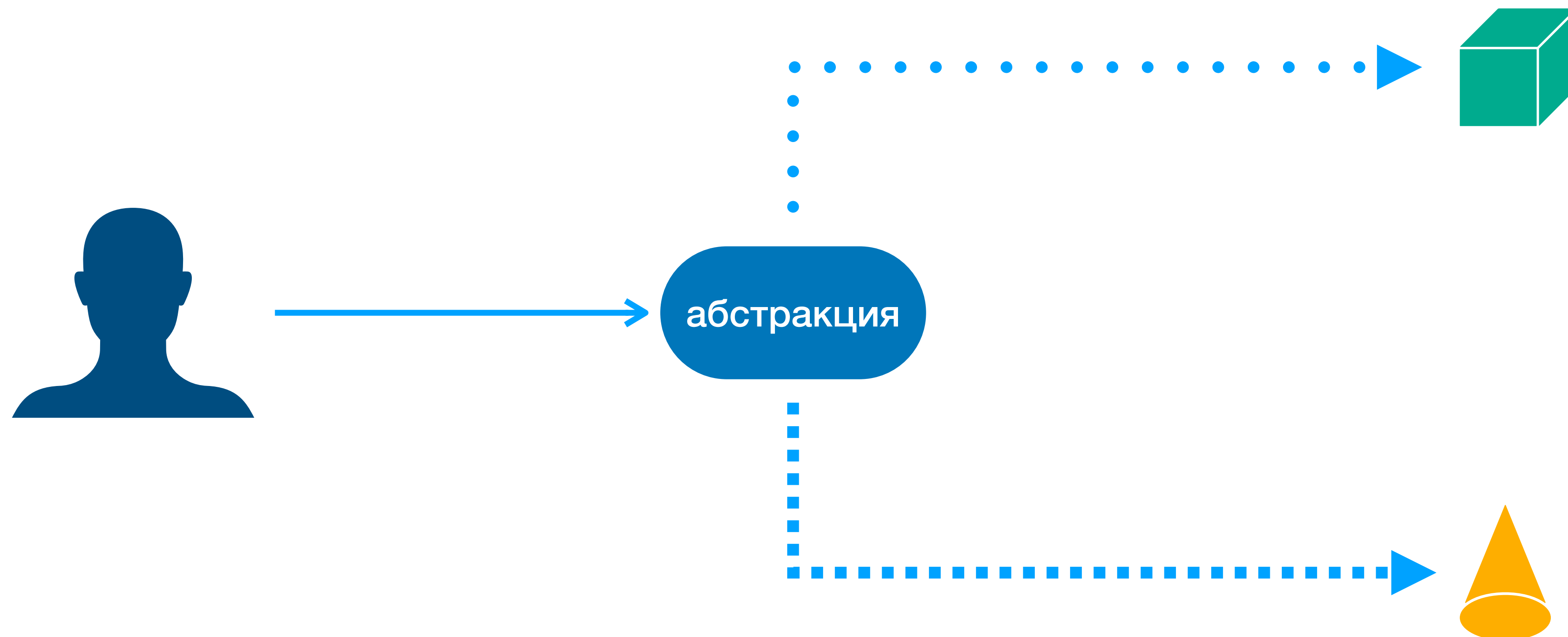
# полиморфизм

## основные концепции ООП

- подразумевает что логика, написанная один раз, может выполняться по разному
- в контексте ООП – полиморфизм подтипов
- больше отделяем абстракцию от реализации

# полиморфизм

## основные концепции ООП



# интерфейсы

## полиморфизм

- определяют только сигнатуры поведений
- не определяют реализаций
- являются контрактом к объекту типа который его реализует
- зачастую называются абстракцией

```
public interface IMovable
{
    Point Location { get; }

    void MoveTo(Point point);
}
```



# интерфейсы

## полиморфизм

```
public class Car : IMovable
{
    public Point Location { get; private set; }

    public void MoveTo(Point point)
    {
        Console.WriteLine("Wroom-wroom!");
        Location = point;
    }
}
```

```
public struct Stone : IMovable
{
    public Point Location { get; private set; }

    public void MoveTo(Point point)
    {
        Console.WriteLine("Flop-flop");
        Location = point;
    }
}
```

# полиморфизм

## основные концепции ООП

- интерфейсы позволяют лучше структурировать вариативность логики
- полиморфизм позволяет избежать излишней условной логики
- полиморфизм позволяет моделировать вариативность логики представлением субъектов предметной области в виде типов, реализующих соответствующие им поведения

# наследование

## полиморфизм

- наследование в C# поддерживают только классы
- в отличие от интерфейсов получаем и реализации базового класса
- дочерние классы могут переопределять реализации родительских классов

# Виртуальные методы

## наследование

```
public class Car
{
    public Point Location { get; protected set; }

    public virtual void MoveTo(Point point)
    {
        Console.WriteLine("Wroom-wroom!");
        Location = point;
    }
}
```

```
public class FastCar : Car
{
    public override void MoveTo(Point point)
    {
        Console.WriteLine("Wroom-wroom (fast)");
        Location = point;
    }
}
```

# абстрактные методы

## наследование

```
public abstract class CarBase
{
    public Point Location { get; protected set; }

    public abstract void MoveTo(Point point);
}
```

```
public class FastCar : CarBase
{
    public override void MoveTo(Point point)
    {
        Console.WriteLine("Wroom-wroom (fast)");
        Location = point;
    }
}
```

отделение абстракции от реализации,  
позволяющее прозрачно использовать различные  
реализации, посредством единого контракта



**полиморфизм подтипов**

используются интерфейсы, в С# реализовывать интерфейсы  
могут как классы, так и структуры  
говорят, что тип реализует интерфейс  
(класс Car реализует интерфейс IMovable)



**реализация (наследование поведения)**

используются базовые классы,  
в С# одна структура не может быть унаследована от другой, либо от класса  
говорят, что класс является наследником другого класса, либо же его  
подклассом  
(класс FastCar является наследником класса Car)



**наследование (наследование реализаций)**



# наследование

## полиморфизм

- использовать для реализации полиморфизма
- не использовать для переиспользования бизнес логики
- наследование приводит к сильной связанности между типами
  - со временем становится сложнее разорвать эту связь
  - код становится сложнее рефакторить

# наследование

## полиморфизм

применение наследования  
для переиспользования  
данных

ПЛОХО



```
public enum Suit
{
    Hearts,
    Diamonds,
    Clubs,
    Spades,
}

public enum CardValue
{
    Six,
    Seven,
    Eight,
    Nine,
    Ten,
    Jack,
    Queen,
    King,
    Ace,
}

public class Card
{
    public Card(Suit suit, CardValue value)
    {
        Suit = suit;
        Value = value;
    }

    public Suit Suit { get; }

    public CardValue Value { get; }
}
```

```
public class Card
{
    public Card(Suit suit, CardValue value) { ... }

    public Suit Suit { get; }

    public CardValue Value { get; }
}

public class Deck
{
    public Deck(ICollection<Card> cards) { ... }

    public ICollection<Card> Cards { get; }

    public void Shuffle()
    {
        ...
    }
}

public class Dealer : Deck
{
    public Dealer(ICollection<Card> cards) : base(cards) { }

    public void StartGame()
    {
        ...

        Shuffle();

        ...
    }
}
```

# наследование

## полиморфизм

для переиспользования  
логики – используем  
композицию

**ХОРОШО**



```
public class Dealer
{
    private readonly Deck _deck;

    public Dealer(Deck deck)
    {
        _deck = deck;
    }

    public void StartGame()
    {
        ...

        _deck.Shuffle();

        ...
    }
}
```

**совокупность атрибутов и поведений, реализация  
и данные которого скрыты от его конечного  
пользователя**



**объект (с точки зрения теории ООП)**

# проектирование объектной модели

# различия парадигм

## имутабельность

object oriented

```
class Model
{
    public readonly int Value;

    public Model(int value)
    {
        Value = value;
    }
}
```

≠

functional

```
type Model(value: int) =
    let mutable Value = value
```

свойство данных, не подразумевающее изменения

в ООП, используется в виде сокрытия мутабельных данных и имутабельность значений не требующих изменения



**имутабельность**

# ИЗЛИШНЯЯ ИМУТАБЕЛЬНОСТЬ

## имутабельность

```
public class StudentGroup
{
    public long Id { get; set; }

    public string Name { get; set; }

    public List<long> StudentIds { get; set; }

    public void AddStudent(long studentId)
    {
        if (StudentIds.Contains(studentId) is false)
            StudentIds.Add(studentId);
    }
}
```



# МИНИМИЗАЦИЯ МУТАБЕЛЬНОСТИ

## имутабельность

```
public class StudentGroup
{
    private readonly HashSet<long> _studentsIds;

    public StudentGroup(long id, string name)
    {
        Id = id;
        Name = name;

        _studentsIds = new HashSet<long>();
    }

    public long Id { get; }

    public string Name { get; set; }

    public IReadOnlyCollection<long> StudentIds => _studentsIds;

    public void AddStudent(long studentId)
    {
        _studentsIds.Add(studentId);
    }
}
```

# Find/Get конвенции

## проектирование объектной модели

```
public record Post(long Id, string Title, string Content);

public class User
{
    private readonly List<Post> _posts;

    public User(IEnumerable<Post> posts)
    {
        _posts = posts.ToList();
    }
}
```

# корректная семантика

## Find/Get конвенции

```
public record Post(long Id, string Title, string Content);

public class User
{
    private readonly List<Post> _posts;

    public User(IEnumerable<Post> posts)
    {
        _posts = posts.ToList();
    }

    public Post GetPostById(long postId)
    {
        return _posts.Single(x => x.Id.Equals(postId));
    }

    public Post? FindPostByTitle(string title)
    {
        return _posts.SingleOrDefault(x => x.Title.Equals(title));
    }
}
```

# некорректная семантика

## Find/Get конвенции

```
public Post FindPost(long postId)
{
    return _posts.Single(x => x.Id.Equals(postId));
}
```

```
public Post? FindPost(string title)
{
    return _posts.SingleOrDefault(x => x.Title.Equals(title));
}
```

# ИСКЛЮЧЕНИЯ

## обработка ошибок

- исключения не отражены в сигнатуре
- поиск конкретного типа исключения и ситуации когда оно кидается приводит к протёкшей абстракции
- неудачное выполнение операции  $\neq$  исключительная ситуация

**абстракция, для работы с которой, необходимо  
иметь знание о деталях её реализации**



**протёкшая абстракция**

# result types

## обработка ошибок

```
public abstract record AddStudentResult
{
    private AddStudentResult() { }

    public sealed record Success : AddStudentResult;

    public sealed record AlreadyMember : AddStudentResult;

    public sealed record StudentLimitReached(int Limit) : AddStudentResult;
}
```

# result types

## обработка ошибок

```
public AddStudentResult AddStudent(long studentId)
{
    if (_studentsIds.Count.Equals(MaxStudentCount))
        return new AddStudentResult.StudentLimitReached(MaxStudentCount);

    if (_studentsIds.Add(studentId) is false)
        return new AddStudentResult.AlreadyMember();

    return new AddStudentResult.Success();
}
```



# result types

## обработка ошибок

```
if (result is AddStudentResult.AlreadyMember)
{
    Console.WriteLine("Student is already member of specified group");
    return;
}

if (result is AddStudentResult.StudentLimitReached err)
{
    var message = $"Cannot add student to specified group, maximum student count of {err.Limit} already reached";
    Console.WriteLine(message);

    return;
}

if (result is not AddStudentResult.Success)
{
    Console.WriteLine("Operation finished unexpectedly");
    return;
}

Console.WriteLine("Student successfully added");
```

# value objects

## проектирование объектной модели

```
public class Account
{
    public decimal Balance { get; private set; }

    public void Withdraw(decimal value)
    {
        if (value < 0)
            throw new ArgumentException("Value cannot be negative", nameof(value));

        Balance -= value;
    }
}
```

# value objects

## проектирование объектной модели

```
public struct Money
{
    public Money(decimal value)
    {
        if (value < 0)
        {
            throw new ArgumentException(
                "Value cannot be negative",
                nameof(value));
        }

        Value = value;
    }

    public decimal Value { get; }

    public static Money operator -(Money left, Money right)
    {
        var value = left.Value - right.Value;
        return new Money(value);
    }
}
```

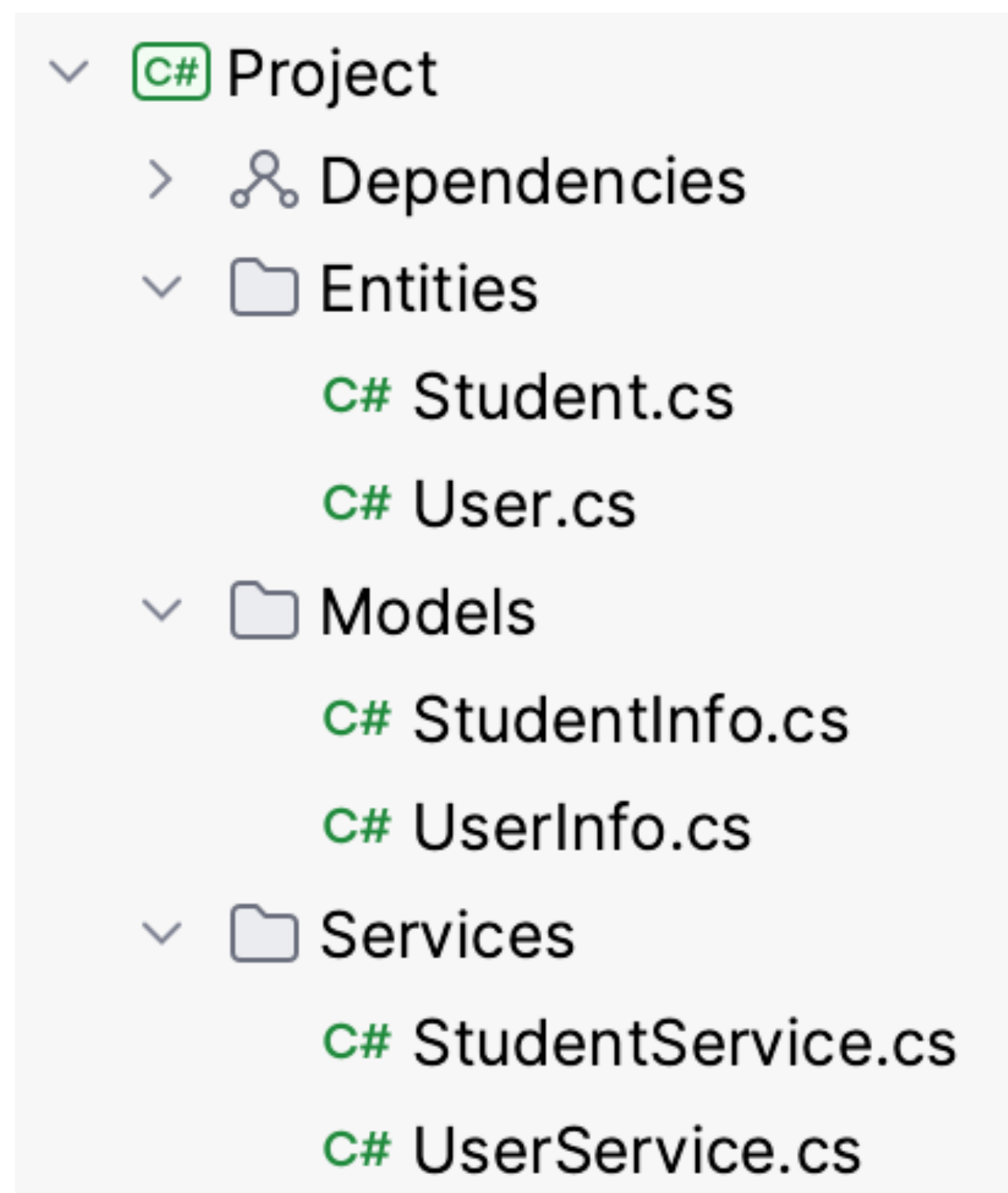
```
public class Account
{
    public Money Balance { get; private set; }

    public void Withdraw(Money value)
    {
        Balance -= value;
    }
}
```

# vertical slices

## проектирование объектной модели

инфраструктурная



семантическая

