

Объектно-ориентированное программирование

Юрий Литвинов
y.litvinov@spbu.ru

18.02.2025

Объектно-ориентированное программирование

- ▶ Программа представляется в виде набора взаимодействующих объектов
- ▶ **Объект** — это набор данных и методов (состояние и поведение), представляющий некую независимую сущность
- ▶ Объекты общаются друг с другом через интерфейсы, объект вправе сам решать, как обработать вызов
- ▶ **Интерфейс** — собственно то, что может делать объект

Абстракция

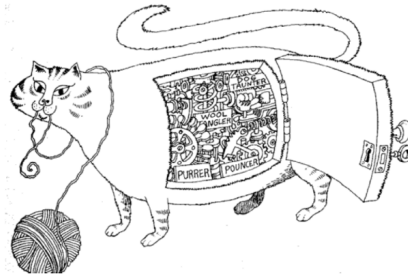
Абстракция выделяет существенные характеристики объекта, отличающие его от остальных объектов, с точки зрения наблюдателя



© G. Booch, «Object-oriented analysis and design»

Инкапсуляция

Инкапсуляция разделяет интерфейс (**контракты**) абстракции и её реализацию



© G. Booch, «Object-oriented analysis and design»

Инвариант объекта

Инкапсуляция защищает **инварианты** абстракции

Инвариант — набор условий на состояние объекта, которые выполняются всё время его жизни

Например:

- ▶ Поле size списка должно содержать число, равное количеству элементов списка
- ▶ Поле next элемента списка, на который указывает поле prev текущего элемента, должно быть текущим элементом
- ▶ Баланс на счету мобильного больше нуля или нельзя звонить

Инвариант может нарушаться внутри метода, но восстанавливаться при его окончании

Объект сам отвечает за поддержание своих инвариантов

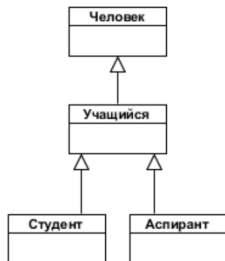
Класс

- ▶ **Класс** — тип объекта (определяет поля и методы, которые могут быть у объекта)
- ▶ Бывают не во всех объектно-ориентированных языках
- ▶ **Поля** — собственно, состояние объекта
- ▶ **Поля класса** — общее состояние ВСЕХ объектов данного класса
- ▶ **Методы** — поведение объекта
- ▶ **Методы класса** — поведение, не зависящее от состояния конкретного объекта (одинаковое для всех объектов данного класса)

Наследование

Генерализация

- ▶ **Генерализация** — отношение между классами, связывающее более общее понятие и более конкретное
 - ▶ Например, всякий студент — человек (надеюсь)
 - ▶ Поэтому у каждого студента есть имя и фамилия, поскольку они есть у человека
 - ▶ То же с поведением, более частное понятие «наследует» поведение более общего



Наследование

Сабтайпинг

- ▶ Объект класса-потомка **является (is-a)** одновременно объектом класса предка
- ▶ И может быть использован везде, где может быть использован предок
- ▶ Отношение «является» и называется сабтайпингом (subtype polymorphism)
- ▶ Бывает наследование без сабтайпинга (например, в C++), бывает сабтайпинг без наследования (например, в Паскале)

Типы времени компиляции и времени выполнения

- ▶ Каждый объект на самом деле имеет много типов — свой и всех своих предков
- ▶ Его «настоящий» тип — **тип времени выполнения**
- ▶ Тип, по которому мы с ним работаем в конкретном месте программы — **тип времени компиляции**

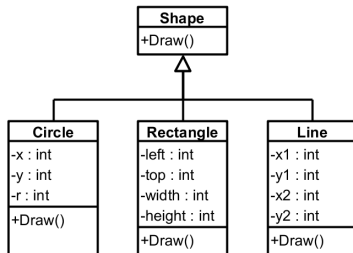
Пример:

```
Shape a = new Circle();
```

Тип времени компиляции — *Shape*, следовательно у *a* можно вызывать только методы *Shape*.

Тип времени выполнения — *Circle*.

Полиморфизм



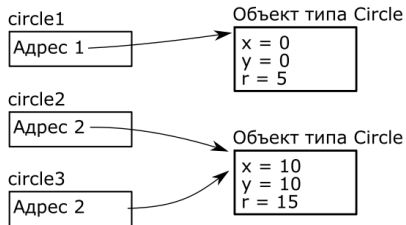
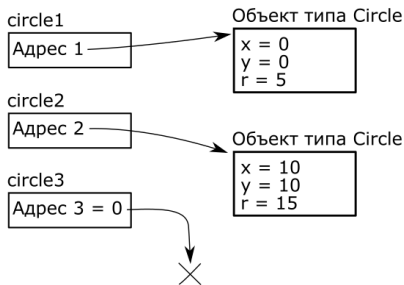
```
List<Shape> shapes = new List {  
    new Circle(0, 0, 5), new Line(0, 0, 10, 10) };
```

```
foreach (var shape in shapes)  
{  
    shape.Draw();  
}
```

Абстрактные классы

- ▶ Иногда у предка нет разумного поведения по умолчанию
 - ▶ Например, что такое *Draw* для просто *Shape*?
- ▶ Такие методы не реализуются, а остаются **абстрактными**
- ▶ Если у класса есть хоть один абстрактный метод, он не может порождать объекты
 - ▶ То есть сам класс является абстрактным
- ▶ **Интерфейс** — класс, у которого все методы абстрактные
 - ▶ Например, *Shape* мог бы быть интерфейсом
- ▶ На самом деле, интерфейс — это контракт класса

Ссылочные типы и типы-значения



```
var circle1 = new Circle(0, 0, 5);  
var circle2 = new Circle(10, 10, 15);  
var circle3 = null;
```

```
circle3 = circle2;
```

Ссылочные типы и типы-значения в C#

Ссылочные типы:

- ▶ Пользовательские классы
- ▶ Строки
- ▶ Массивы
- ▶ Исключения
- ▶ Делегаты

Типы-значения:

- ▶ Примитивные типы
- ▶ Перечисления
- ▶ Структуры

Пример

```
static void Add(string s1, string s2, string s3)
```

```
{
```

```
    s3 = s1 + s2;
```

```
}
```

```
private static void Main(string[] args)
```

```
{
```

```
    string s1 = "a";
```

```
    string s2 = "b";
```

```
    string s3 = "c";
```

```
    Add(s1, s2, s3);
```

```
}
```

Передача параметров по ссылке

```
static void Add(string s1, string s2, ref string s3)
```

```
{
```

```
    s3 = s1 + s2;
```

```
}
```

```
private static void Main(string[] args)
```

```
{
```

```
    string s1 = "a";
```

```
    string s2 = "b";
```

```
    string s3 = "c";
```

```
    Add(s1, s2, ref s3);
```

```
}
```

Out-параметры

```
static void Add(string s1, string s2, out string s3)
```

```
{  
    s3 = s1 + s2;  
}
```

```
private static void Main(string[] args)
```

```
{  
    string s1 = "a";  
    string s2 = "b";  
    Add(s1, s2, out string s3);  
    Console.WriteLine(s3);  
}
```


На самом деле, это не нужно

```
static string Add(string s1, string s2)
{
    return s1 + s2;
}
```

```
private static void Main(string[] args)
{
    string s1 = "a";
    string s2 = "b";
    string s3 = Add(s1, s2);
    Console.WriteLine(s3);
}
```

C# умеет возвращать пары, тройки и т.д.

Конструкторы

```
class Circle
```

```
{  
    public Circle(int x, int y, int r)  
    {  
        this.x = x;  
        this.y = y;  
        this.r = r;  
    }  
  
    private int x;  
    private int y;  
    private int r;  
}
```

Перегрузка конструкторов, chaining

```
class Circle
```

```
{  
    public Circle(int x, int y, int r)  
    {  
        this.x = x;  
        this.y = y;  
        this.r = r;  
    }  
  
    public Circle(int r)  
        : this(0, 0, r)  
    {  
    }  
  
    private int x;  
    private int y;  
    private int r;  
}
```

Наследование (1)

```
class Shape
```

```
{
```

```
    public Shape()
```

```
    {
```

```
    }
```

```
    public Shape(int x, int y)
```

```
    {
```

```
        this.x = x;
```

```
        this.y = y;
```

```
    }
```

```
    protected int x;
```

```
    protected int y;
```

```
}
```

Наследование (2)

```
class Circle : Shape
{
    Circle(int x, int y, int r)
    {
        this.x = x;
        this.y = y;
        this.r = r;
    }

    Circle(int r)
        : base(0, 0)
    {
    }

    private int r;
}
```

Интерфейсы

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

Реализация интерфейса

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

    protected int x;
    protected int y;
}
```

Явная реализация интерфейса

```
class Shape : IDrawable
{
    void IDrawable.Draw()
    {
        Console.WriteLine("Drawing Shape");
    }

    protected int x;
    protected int y;
}
```


Зачем

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

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

```
class Shape : IDrawable, IMySpecialDrawable  
{  
    void IDrawable.Draw()  
        => Console.WriteLine("Drawing Shape");  
  
    void IMySpecialDrawable.Draw()  
        => Console.WriteLine("Drawing Shape, but completely unrelated to IDrawable");  
  
    protected int x;  
    protected int y;  
}
```

И вот что будет

```
var shape = new Shape();  
shape.Draw(); // Ошибка компиляции  
((IDrawable)shape).Draw(); // Drawing Shape  
((IMySpecialDrawable)shape).Draw(); // Drawing Shape, but...
```

Абстрактные классы

```
abstract class Shape
```

```
{
```

```
    public Shape()
```

```
    {
```

```
    }
```

```
    public abstract void Draw();
```

```
    protected int x;
```

```
    protected int y;
```

```
}
```

Виртуальные методы (1)

```
class Shape
```

```
{
```

```
    public virtual void Draw()
```

```
    {
```

```
        Console.WriteLine(
```

```
            $"Drawing Shape with coords ({x}, {y})");
```

```
    }
```

```
    protected int x;
```

```
    protected int y;
```

```
}
```

Виртуальные методы (2)

```
class Circle : Shape
{
    public Circle(int x, int y, int r)
    {
        this.x = x;
        this.y = y;
        this.r = r;
    }

    public override void Draw()
    {
        Console.WriteLine($"Drawing Circle with radius {r}");
    }

    private int r;
}
```

Виртуальные методы (3)

```
class Rectangle : Shape
{
    public Rectangle(int x, int y, int width, int height)
    {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    public override void Draw()
    {
        base.Draw();
        Console.WriteLine(
            $"Drawing Rectangle with width={width} and height={height}");
    }
    protected int width;
    protected int height;
}
```

Виртуальные методы (4)

```
private static void Main(string[] args)
{
    var circle = new Circle(0, 0, 10);
    var rectangle = new Rectangle(0, 0, 10, 10);
    var list = new System.Collections.Generic.List<Shape>();
    list.Add(circle);
    list.Add(rectangle);
    foreach (var shape in list)
    {
        shape.Draw();
    }
}
```

Абстрактные методы

```
abstract class Shape
{
    public abstract void Draw();

    protected int x;
    protected int y;
}
```


Переведение методов, new (1)

```
class Shape
```

```
{
```

```
    public virtual void Draw()
```

```
    {
```

```
        Console.WriteLine(
```

```
            $"Drawing Shape with coords ({x}, {y})");
```

```
    }
```

```
    protected int x;
```

```
    protected int y;
```

```
}
```

Переведение методов, new (2)

```
class Circle : Shape
{
    public Circle(int x, int y, int r)
    {
        this.x = x;
        this.y = y;
        this.r = r;
    }

    public new void Draw()
    {
        Console.WriteLine($"Drawing Circle with radius {r}");
    }

    private int r;
}
```

Переведение методов, new (3)

```
Circle circle1 = new Circle(10, 10, 3);  
Shape circle2 = new Circle(10, 10, 3);  
var circle3 = new Circle(10, 10, 3);
```

```
circle1.Draw();  
circle2.Draw();  
circle3.Draw();
```

Лучше про это забыть и никогда не пользоваться, хотя на собеседованиях спрашивают

Модификаторы видимости

- ▶ **public** — применяется к типам и членам, доступ без ограничений
- ▶ **protected** — применяется только к членам, доступ в типе и потомках
- ▶ **internal** — применяется к типам и членам, доступ внутри сборки
- ▶ **protected internal** — применяется к типам и членам, доступ внутри сборки **или** в потомках
- ▶ **private** — применяется к типам и членам, доступ только внутри типа
- ▶ По умолчанию для типов **internal**, для членов — **private**

Другие модификаторы

- ▶ **partial** — «частичный» класс, декларирует, что определение класса разбито на несколько файлов
 - ▶ Для интеграции сгенерированного и рукописного кода, не используйте без нужды
- ▶ **sealed** — запрещение наследования от класса
- ▶ **static** — не может быть инстанцирован, может содержать только **static**-методы

Вложенные классы

```
class Circle
{
    private readonly Point pos;
    private readonly int r;

    private class Point
    {
        public int x;
        public int y;
    }

    public Circle(int x, int y, int r)
    {
        pos = new Point {x = 10, y = 10};
        this.r = r;
    }

    public void Draw() =>
        Console.WriteLine($"({pos.x}, {pos.y}), radius {r}");
}
```

Преобразования типов

```
Shape shape = new Circle();
```

```
Circle circle = (Circle)shape;
```

```
Circle circle = shape as Circle;
```

```
if (shape is Circle)
```

```
{  
    Circle circle = (Circle)shape;  
}
```

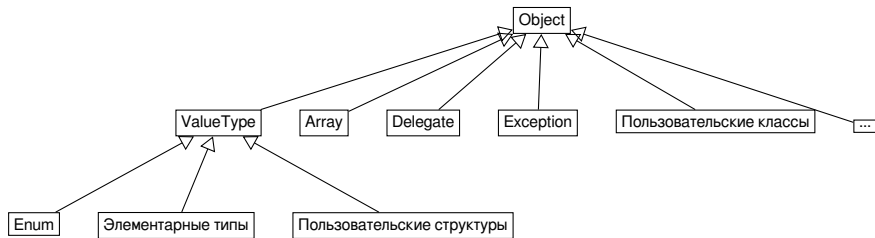
```
if (shape is Circle circle)
```

```
{  
  
}
```

Сопоставление шаблонов

```
switch (shape)
{
    case Circle c:
        WriteLine($"circle with radius {c.Radius}");
        break;
    case Rectangle s when (s.Length == s.Height):
        WriteLine($"{s.Length} x {s.Height} square");
        break;
    case Rectangle r:
        WriteLine($"{r.Length} x {r.Height} rectangle");
        break;
    default:
        WriteLine("<unknown shape>");
        break;
    case null:
        throw new ArgumentNullException(nameof(shape));
}
```


Иерархия основных классов



Методы System.Object

- ▶ Equals — виртуальный
- ▶ GetHashCode — виртуальный
- ▶ ToString — виртуальный
- ▶ GetType — не виртуальный
- ▶ MemberwiseClone — не виртуальный защищённый
 - ▶ Создаёт объект, не вызывая конструктор
- ▶ Finalize — виртуальный защищённый

Объекты в памяти

```
void Example()
{
    Employee e =
        new Manager();
    e.GenProgressReport();
}
```

