ЛЕКЦИЯ 1

- Модуль 2
- 01.11.2023
- Основные концепции объектно-ориентированного программирования

ЦЕЛИ ЛЕКЦИИ

- Познакомится с общеархитектурными принципами проектирования
- Познакомится с понятием объекта и класса
- Разобраться с созданием объектов

ГРАФИК КОНТРОЛЕЙ, МОДУЛЬ 2

Контроль	Начало	Окончание
КД3_2_1 (Обработка csv-файла)	**1 нед**	**3 нед**
СР2_1 (простое ООП, перегрузка операций)	**3 нед**	
КД3_2_2	**3 нед**	**5 нед**
СР2_2 (наследование, полиморфизм, virtual, new, is)	**5 нед**	
КД3_2_3	**5 нед**	**7 нед**
КР_2	**7 нед**	

Темы контролей предварительные и могут варьироваться в зависимости от скорости подачи и освоения материала

ОБЩЕАРХИТЕКТУРНЫЕ ПРИНЦИПЫ

KISS, YAGNI, DRY

и

Бритва Оккама

KEEP IT SHORT AND SIMPLE

Архитектура системы должна быть максимально простой

YOU AIN'T GONNA NEED IT

Избегаем реализации ненужной функциональности

WRITE EVERYTHING TWICE OR WE ENJOY TYPING

- Stay **dry**, don't be **wet**

Не дублируем представление данных и реализацию

БРИТВА ОККАМА

Не следует множить сущее без необходимости

Самое простое решение почти всегда самое правильное

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

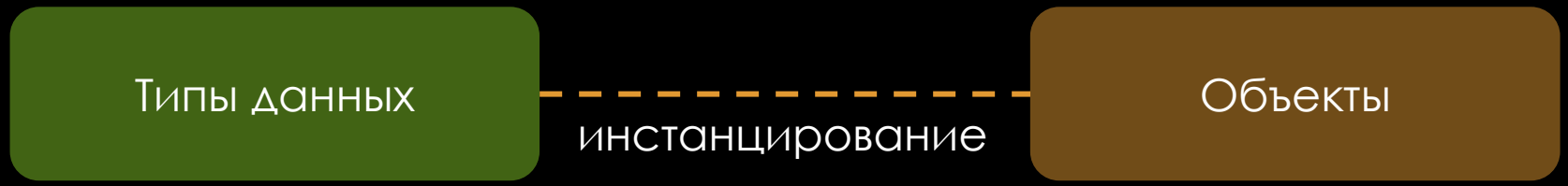
Основные определения

Базовые принципы

ОО-ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ

- **Объектно-ориентированный подход** [object-oriented approach] – подход к решению проблем, предполагающий объектовую декомпозицию проблемы
- **Объектно-ориентированное программирование** [object-oriented programming] – это парадигма программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром некоторого класса, входящего в иерархию наследования

ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ ПАРАДИГМА В C#




```
string str = new string('a', 15);
```

```
int[] ar = new int[2];
```

ПРЕИМУЩЕСТВА ИСПОЛЬЗОВАНИЯ ОО ПАРАДИГМЫ

Объекты достаточно естественны для восприятия человеком

Объекты могут содержать внутренне устройство, закрытое от внешнего мира

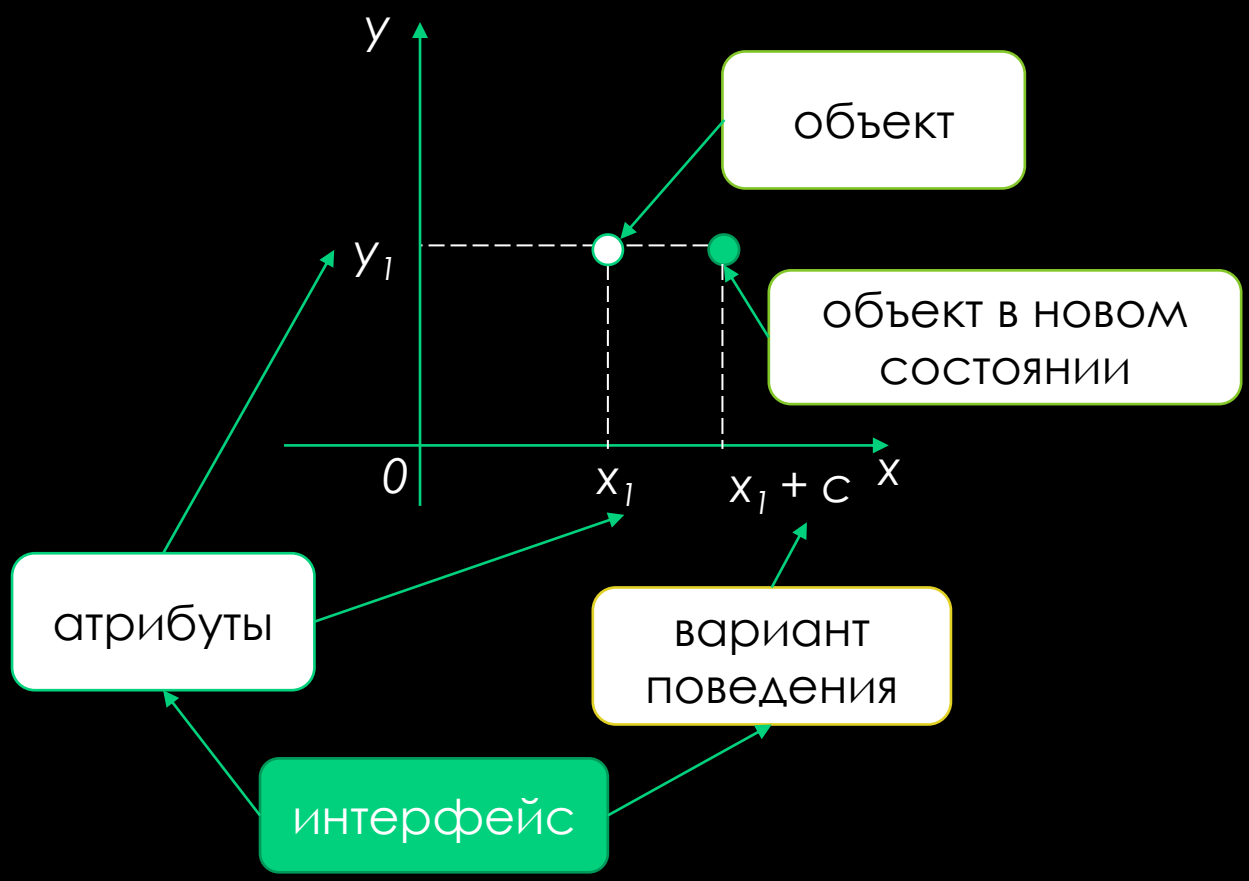
Можно представлять одни типы как подтипы других

Подтип всегда можно использовать там, где ожидается базовый тип

ОПРЕДЕЛЕНИЯ, СВЯЗАННЫЕ С ОБЪЕКТОМ

- **Объект** [object] – произвольная сущность, для которой достаточно легко определить границы, состояние и поведение
  - **Состояние объекта** определяется совокупностью значений атрибутов
    Атрибут [attribute] – свойство объекта
  - **Поведение объекта** определяется совокупностью методов
    - **Метод** [method] – вариант поведения объекта
  - **Интерфейс** [interface] **объекта** – внешнее представление объекта, то есть всего его атрибуты и методы
    - Описание интерфейса задаёт **границы объекта** самим фактом включения или не включения тех или иных атрибутов или методов

РАЗБИРАЕМСЯ С ТЕРМИНАМИ



СОЗДАНИЕ ОБЪЕКТОВ

Синтаксис создания объектов в C#:

```
<Тип> <Идентификатор ссылки> = new <Тип>([Параметры]);
```

```
string str = new string('a', 15);
```

```
string str = new ('a',15);
```

```
<Тип> <Идентификатор ссылки> = new ([Параметры]); // (начиная с C# 9.0)
```

ОПРЕДЕЛЕНИЯ, СВЯЗАННЫЕ С КЛАССОМ

- **Класс** [class] – описание однотипных объектов, то есть их интерфейса
- Объект, удовлетворяющий этому шаблону, называется **экземпляром** [instance] **класса**

Класс:
- Точка на вещественной плоскости

Состояние:
- координата по оси X
- координата по оси Y

Поведение:
- получить координату по оси X
- получить координату по оси Y
- изменить координату по оси X на вещественную константу C

КЛАСС, КАК ТИП ДАННЫХ

Класс – это тип данных

```
class Point
{
    double _x;
    double _y;

    public override string ToString() =>
                          $"({_x:f3};{_y:f3})";
}
```

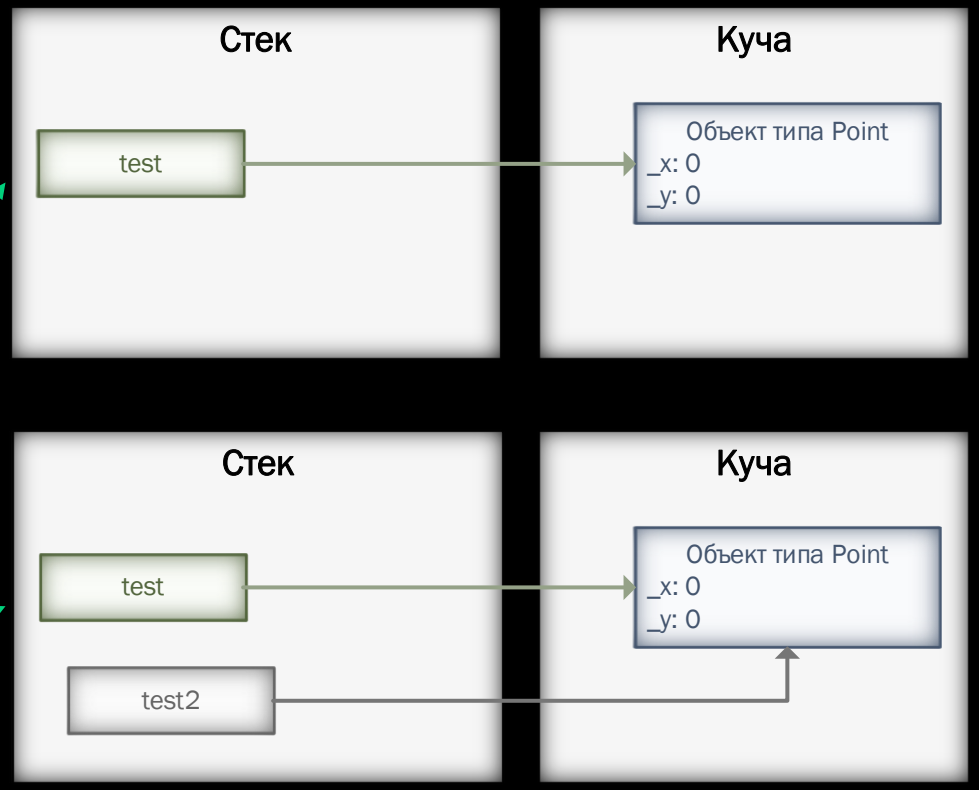
Состояние

Поведение

```
Point test = new Point();
Console.WriteLine(test);
```

```
Point test2 = test;
```

Класс – это ссылочный тип данных



По умолчанию **internal**

```
class Point
{
    double _x;
    double _y;

    public double X
    {
        get { return _x; }
        set { _x = value; }
    }
    public double Y
    {
        get { return _y; }
        set { _y = value; }
    }
    public void MoveX(double c) => _x += c;

    public override string ToString() => $"({_x:f3};{_y:f3})";
}
```

По умолчанию **private**

Свойства доступа к полям

В интерфейс входит только отмеченное **public**

Класс:
- Точка на вещественной плоскости

Состояние:
- координата по оси X
- координата по оси Y

Поведение:
- получить координату по оси X
- получить координату по оси Y
- изменить координату по оси X на вещественную константу C

МОДИФИКАЦИЯ НА ОСНОВЕ АВТОРЕАЛИЗУЕМЫХ СВОЙСТВ

```
class Point
{
    public double X { get;set; }
    public double Y { get;set; }

    public void MoveX(double c) => X += c;

    public override string ToString() => $"({X:f3};{Y:f3})";
}
```

Поля теперь неявно спрятаны за автореализуемые свойства

Теперь обращаемся к свойствам, а не к полям

СОЗДАНИЕ ОБЪЕКТОВ КЛАССА

Конструктор

Операция new

ОПЕРАЦИЯ NEW

- **вычисляет количество байтов**, необходимых для хранения всех экземплярных полей типа и всех его базовых типов, включая System.Object
- **выделяет память для объекта с резервированием необходимого для данного типа количества байтов в управляемой куче**
  - Выделенные байты инициализируются нулями
- **инициализирует указатель на объект-тип и индекс блока синхронизации**
- **вызывает конструктор экземпляра типа с параметрами, указанными при вызове new**
- **возвращает ссылку на вновь созданный объект**

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ ОПЕРАЦИИ NEW

```
Point p1 = new Point();
```

```
Point p2 = new();
```

Имя типа можно опустить, т.к. известен целевой тип конструктора

```
Point p3 = new() { X = 1.2, Y = 4.3 };
```

Имя типа можно опустить, т.к. известен целевой тип конструктора и использовать список инициализации

```
Point p2 = new(X: 1.2, Y: 4.3);
```

Такой вариант в нашем случае не сработает, т.к. требуется конструктор с двумя параметрами

КОНСТРУКТОРЫ КЛАССОВ

Конструктор – специальный функциональный член, позволяющий создавать объекты данного типа. Конструкторы позволяют описать обязательные параметры, необходимые для создания экземпляров

Синтаксис описания конструктора:

```
[Модификаторы] <Идентификатор Типа> ([Параметры]) {[Тело]}
```

При наличии хотя бы одного явно определённого конструктора, конструктор по умолчанию не генерируется

ОСОБЕННОСТИ КОНСТРУКТОРОВ

- Имя обязано совпадать с именем типа
- **Нет явно указанного типа возвращаемого значения**, они всегда (неявно) возвращают ссылку на созданный объект
- Могут быть перегружены, аналогично методам
- При отсутствии явных определений конструктора, к**омпилятор генерирует конструктор без параметров**, инициализирующий поля значениями их типов по умолчанию

```
class Point
{
    public double X { get;set; }
    public double Y { get;set; }

    public void MoveX(double c) => X += c;

    public override string ToString() => $"({X:f3};{Y:f3})";
}
```

```
Point p1 = new Point();
```

ЭКЗЕМПЛЯРЫ КЛАССА

```
class Point
{
    public double X { get;set; }
    public double Y { get;set; }

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

    public void MoveX(double c) => X += c;

    public override string ToString() => $"({X:f3};{Y:f3})";
}
```

Добавили конструкторы

```
Point p0= new Point();
Point p1 = new Point(1, 0);
Point p2 = new (x: 1.2, y: 4.3);
Point p3 = new() { X = 1.2, Y = 4.3 };
```

Теперь это допустимо

ССЫЛКА THIS

- Используется объектом для ссылки на себя
- Позволяют избегать путаницы при совпадающих именах параметров и членов классов
- В описании индексатора (об этом немного позже)

```
public Point(double x, double y) => (this.X, this.Y) = (x, y);
```

```
public Point(double x, double y) : this() => (this.X, this.Y) = (x, y);
```

КОНСТРУКТОРЫ И МОДИФИКАТОР READONLY

```
class Point
{
    readonly char _separator = ';';
    public double X { get; set; }
    public double Y { get; set; }

    public Point() => _separator = ':';

    public Point(double x, double y) (X, Y) = (x, y);
    public void MoveX(double c) => X += c;

    public override string ToString() => $"({X:f3}{_separator}{Y:f3})";
}
```

readonly-поле получает значение только при объявлении или в конструкторе

Теперь для точек, созданных конструтором по умолчанию сепаратор будет :

ДЕСТРУКТОР

- **Деструктор** (метод завершения) – специальный метод класса, который вызывается автоматически при уничтожении объекта сборщиком мусора

```
class Point
{
    readonly char _separator = ';';
    public double X { get; set; }
    public double Y { get; set; }

    public Point() => _separator = ':';

    public Point(double x, double y) => (this.X, this.Y) = (x, y);
    public void MoveX(double c) => X += c;

    public override string ToString() => $"({X:f3}{_separator}{Y:f3})";
    ~Point() { }
}
```

Код для освобождения ресурсов

ОСОБЕННОСТИ ДЕСТРУКТОРА

- Деструкторы применяются только в классах, **в структурах их определить невозможно**
- Имя деструктора совпадает с именем класса
- При описании деструктора его имя предваряется знаком **~**
- Параметры у деструктора отсутствуют
- **Деструкторы не перегружаются, то есть в классе может быть только один деструктор**
- Деструктор в классе может быть только один
- Напрямую деструктор вызвать нельзя, он запускается только автоматически при наличии

СРАВНЕНИЕ КОНСТРУКТОРОВ И ДЕСТРУКТОРОВ

Для чего	**Кто**	**Когда и как вызывается**
Экземпляр	Конструктор	Однократно вызывается для создания каждого объекта класса (одно создание объекта – один вызов конструктора)
	Деструктор	Вызывается для каждого объекта класса, в некоторый момент времени после того, как в программе с объектом потеряна связь (ссылки отсутствуют)

КЛАССЫ – НАСЛЕДНИКИ OBJECT

Переопределение методов базового типа (немного о полиморфизме)

Ссылки с типом Object

КЛАССЫ C# НАСЛЕДНИКИ SYSTEM.OBJECT

Имя	Описание
Открытые (public)	
Equals(Object)	Возвращает true, если два объект-параметр совпадает с текущим.
GetHashCode()	Возвращает хеш-код для значения текущего объекта.
GetType()	Возвращает объект Type для текущего экземпляра.
ToString()	По умолчанию возвращает полное имя типа.
Защищённые (protected)	
MemberwiseClone()	Создаёт новый экземпляр и присваивает его полям значения соответствующих полей объекта this.
Finalize()	Позволяет объекту выполнить набор действий или освободить ресурсы до того, как он будет удалён сборщиком мусора.

ПЕРЕОПРЕДЕЛЕНИЕ TOSTRING()

Переопределение ToString() из класса Point

```
public override string ToString() => $"({X:f3}{_separator}{Y:f3})";
```

Вызов ToString() для объектов класса Point

```
Point p0 = new Point();
Console.WriteLine(p0.ToString());

Point p1 = new Point(1, 0);
Console.WriteLine(p1.ToString());
```

```
(0,000:0,000)
(1,000;0,000)
```

```
Point p2 = new(x: 1.2, y: 4.3);
Console.WriteLine(p2.ToString());

Point p3 = new() { X = 1.2, Y = 4.3 };
Console.WriteLine(p3.ToString());
```

```
(1,200;4,300)
(1,200:4,300)
```

ПРИМЕР ССЫЛКИ С ТИПОМ OBJECT

```
public class Cat
{
    string _name;

    public Cat(string n) => _name = n;

    public override string ToString() => $"Cat:: {_name}";
}

public class Dog
{
    string _name;

    public Dog(string n) => _name = n;

    public override string ToString() => $"Dog:: {_name}";
}
```

Классы **Cat** и **Dog** не объединены общим предком, кроме **Object**

```
Cat c = new Cat("Мурка");
Dog d = new Dog("Шарик");

Object[] pets = { c, d };
foreach (Object cur in pets)
{
    Console.WriteLine(cur);
}
```

Объекты типов **Cat** и **Dog** собраны в массиве ссылок с типом **Object[]**

ИСПОЛЬЗОВАННАЯ ЛИТЕРАТУРА

- Модификаторы доступа (Справочник по C#) (https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/access-modifiers)
- Оператор is (справочник по C#)
- The Unix Philosophy in One Lesson [http://www.catb.org/~esr/writings/taoup/html/ch01s07.html]
- Occam's Razor and the Art of Software Design [http://michaellant.com/2010/08/10/occams-razor-and-the-art-of-software-design/]