ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

C# – объектно-ориентированный язык. В этой лекции изучим терминологию и концепцию ООП.

Классы и объекты

Ключевым словом в ООП является класс. Все языки программирования могут обращаться с общими данными и методами. Эта возможность помогает избежать дублирования. Главная концепция программирования – не писать один и тот же код дважды. Программы без дублирования лучше и понятнее, так как содержат меньше кода. ООП переводит эту концепцию на новый уровень, он позволяет описывать классы (множества объектов), которые делают общими и структуру, и поведение. Классы не ограничиваются описанием конкретных объектов, они также могут описывать и абстрактные вещи.

Объект – это конкретный представитель класса. Его определяет три характеристики: уникальность, поведение и состояние. *Уникальность* – это характеристика, определяющая отличие одного объекта от другого. *Поведение* определяет то, чем объект может быть полезен, что он может делать. Поведение объекта классифицирует его. Объекты разных классов отличаются своим поведением. *Состояние* описывает внутреннюю работу объекта то, что обеспечивает его поведение. Хорошо спроектированный объект оставляет своё состояние недоступным. Нас не интересует, как он это делает, нам важно то, что он умеет это делать.

Сравним структуры и классы. В C# структуры могут иметь методы, но желательно избегать этого. Однако в некоторых структурах необходимы операторы. Операторы – стилизованные методы, но они не добавляют поведение, а обеспечивают более краткий синтаксис.

Структурные типы – нижний уровень программы, это элементы, из которых строятся более сложные элементы. Переменные структурных типов свободно копируются и используются как поля и атрибуты объектов.

Ссылочные типы – верхний уровень программы, они состоят из мелких элементов. Ссылочные типы в основном не могут быть скопированы.

Абстракция – это тактика очистки объекта от всех несущественных деталей, оставляя только существенную минимальную форму. Абстракция – важный принцип программирования. Хорошо спроектированный класс содержит минимальный набор методов, полностью описывающий его поведение.

Инкапсуляция данных

Традиционное процедурное программирование содержит много данных и много процедур. Любая функция имеет доступ к любым данным. Когда программы становятся большими, это создаёт много проблем – при небольших изменениях в коде необходимо следить за всей программой. Другая проблема – это хранение данных отдельно от функций. В ООП эта проблема решается за счет объединения данных и методов, которые работают с этими данными как одно целое.

Данные и функции объединены в одну сущность, эта сущность ограничивает капсулу. Получаем две области: снаружи этой капсулы и внутри неё. Элементы, которые доступны извне, называются public, те, которые доступны только внутри класса – private. C# не ограничивает области видимости, любой элемент может быть как public, так и private.

Две причины использования инкапсуляции – это контроль использования и минимизация воздействий при изменениях. Можно использовать инкапсуляцию данных и определить поведение для того, чтобы с объектом работали по заданным правилам. Вторая причина вытекает из первой, если данные закрыты от внешнего использования, то их изменение не влияет на использование объекта извне.

Большинство данных внутри объектов описывают информацию об индивидуальности объекта. Данные внутри объекта обычно private и доступны только из методов класса.

Иногда необязательно хранить информацию внутри каждого объекта. Т.е. может быть информация, одинаковая для всех объектов данного класса. Для этого используются статичные поля, которые принадлежат не конкретному объекту, а всему классу.

Статичные методы инкапсулируют статичные данные. Статичные методы существуют на уровне класса, в них нельзя использовать оператор this, но можно обращаться к полям класса, если получить объект класса как параметр.

```
class Time
{
public static void Reset(Time t)
{
t.hour = 0; // ОК
t.minute = 0; // ОК
hour = 0; // ошибка при компиляции
minute = 0; // ошибка при компиляции
}
private int hour, minute;
}
```

Теперь вернемся к программе Hello world, рассмотрим её со стороны объектно-ориентированного программирования. Ответим на два вопроса: как при выполнении вызывается класс и почему метод Main статичный?
Если в файле два класса с методом Main, то точка входа определяется при компиляции.

```
// TwoEntries.cs
using System;
class EntranceOne
{
public static void Main( )
{
Console.Write("EntranceOne.Main( )");
}
}
class EntranceTwo
{
public static void Main( )
{
Console.Write("EntranceTwo.Main( )");
}
}
// Конец файла
```

```
c:\> csc /main:EntranceOne TwoEntries.cs
c:\> twoentries.exe
EntranceOne.Main( )
c:\> csc /main:EntranceTwo TwoEntries.cs
c:\> twoentries.exe
EntranceTwo.Main( )
c:\>
```

Если в файле нет классов с методом Main, то из него нельзя скомпилировать запускаемый файл, только библиотеку dll.

```
// NoEntrance.cs
using System;
class NoEntrance
{
public static void NotMain( )
{
Console.Write("NoEntrance.NotMain( )");
}
}
// Конец файла
c:\> csc /target:library NoEntrance.cs
c:\> dir
...
NoEntrance.dll
...
```

Почему метод Main должен быть статичным? Так как для вызова нестатичного метода необходимо создать объект, а при вызове Main программа только начинает работу, и никаких объектов ещё нет.
Для определения простых классов необходимо выполнить следующую последовательность действий: обозначить ключевым словом class начало класса, определить поля как в структурах, определить методы внутри класса, установить модификаторы доступа для всех полей и методов класса. Модификатор public означает, что “доступ неограничен”, private – “доступ ограничен типом, которому принадлежит”. Если пропустить модификатор, то по умолчанию поле или метод будут private.

```
class BankAccount
{
public void Deposit(decimal amount)
{
balance += amount;
}
private decimal balance;
}
```

При объявлении объекта класса, объект не создаётся, необходимо использовать оператор new, при этом все поля инициализируются нулями. При использовании объекта без создания будет ошибка при компиляции.
Time now = new Time(); // пример объявления объекта класса
Ключевое слово this неявно указывает на объект вызвавший метод. Например, в следующем примере полю name нельзя присвоить значение, компилятор будет считать его параметром

```
class BankAccount
{
public void SetName(string name)
{
name = name;
}
private string name;
}
```

Здесь необходимо было использовать this.name = name;
В C# можно выделить пять различных видов типов:
class struct interface enum delegate
Любой из них можно использовать в классе. Т.е. в классе могут содержаться другие классы. Вложенные классы должны помечаться модификатором доступа, использование вложенных классов переводит имена из глобальной области видимости и пространства имен.
Public вложенные классы не имеют ограничений по использованию, полное имя класса можно использовать в любом месте программы. Private вложенный класс виден только из класса, содержащего его. Класс без модификатора по умолчанию является private.

Наследование и полиморфизм

Наследование – это связи на уровне классов. Новый класс может наследовать существующий класс. Наследование – это мощная связь, так как наследуемый класс наследует все (не private) элементы базового класса. От базового класса может наследоваться любое количество классов. Изменение базового класса, автоматически изменяет все классы-потомки.

Классы потомки могут быть одновременно и базовыми классами для других классов. Группа классов, связанных наследованием, формирует структуру, называемую иерархией классов. При движении по иерархии вверх переходим к более общим классам. При движении вниз – к более специализированным классам.

Простое наследование – это случай, когда у класса есть только один прямой базовый класс.

Множественное наследование, когда у класса есть несколько прямых базовых классов. Множественное наследование создаёт предпосылки к ошибочному использованию наследования. Поэтому C#, как и большинство современных языков программирования, запрещает множественное наследование. Напомним, что наследование, особенно множественное, позволяет рассматривать один объект с разных точек зрения.

Полиморфизм с литературной точки зрения означает много форм или много обликов. Это концепция, по которой один и тот же метод, определенный в базовом классе, может быть по-разному определен в разных классах потомках. Появляется новая проблема, как работать методу у объекта базового класса? Есть возможность не определять метод в базовом классе, т.е. оставить тело метода пустым. Такие методы называют *операциями.*

В типичной иерархии классов операции объявляются в базовом классе, а определяются различными путями в различных классах потомках. Базовый класс представляет имя метода в иерархии. В случае, когда метод не определен в базовом классе, то нельзя создавать объекты этого класса, так для этого объекта будет не определен этот метод. Такие классы называются *абстрактными.*

Абстрактные классы и интерфейсы похожи, так как не могут иметь объектов. Отличие между ними в том, что абстрактный класс может содержать

определения методов, *интерфейсы* содержат только операции (имена методов). То есть интерфейсы абстрактнее абстрактных классов.
Когда вызываем метод напрямую из объекта, а не из операции базового класса, то метод связывается с вызовом при компиляции – *раннее связывание*. При вызове метода не напрямую через объект, а через операцию базового типа, он вызывается при выполнении программы – *позднее связывание.* Гибкость позднего связывания обеспечивается физической и логической ценой. Позднее связывание выполняется дольше, чем раннее. При позднем связывании классы-потомки могут заменять базовые классы. Операции могут быть вызваны из интерфейса, а классы-потомки обеспечат правильное выполнение.