Классы

Класс является наиболее распространенной разновидностью ссылочного типа. Вот как выглядит объявление простейшего класса из возможных:

```
class Foo
{
}
```

Перед ключевым словом `class`	*Атрибуты и модификаторы класса*. Модификаторами невложенных массов являются `public`, `internal`, `abstract`, `sealed`, `static`, `unsafe` и `partial`
После имени класса	*Параметры обобщенных типов, базовый класс* и *интерфейсы*
Внутри фигурных скобок	*Члены класса* (к ним относятся методы, свойства, индексаторы, события, поля, конструкторы, перегруженные операции, вложенные типы и финализатор)

Поля

Поле — это переменная, которая является членом класса или структуры. Например:

```
class Octopus
{
  string  name;
  public int Age = 10;
}
```

Поле может иметь модификатор `readonly`, который предотвращает его изменение после конструирования. Присваивать значение полю, допускающему только чтение, можно лишь в его объявлении или внутри конструктора типа, в котором оно определено.

Инициализация полей является необязательной. Неинициализированное поле получает свое стандартное значение (`0`, `\0`, `null`, `false`). Инициализаторы полей выполняются перед конструкторами в порядке, в котором они указаны.

Для удобства множество полей одного типа можно объявлять в списке, разделяя запятыми. Это подходящий способ обеспечить совместное использование всеми полями одних и тех же атрибутов и модификаторов полей. Например:

```
static readonly int legs = 8, eyes = 2;
```

Методы

Метод выполняет действие в виде последовательности операторов. Метод может получать *входные* данные из вызывающего кода посредством указания *параметров* и возвращать *выходные* данные обратно вызывающему коду за счет указания *возвращаемого типа*. Для метода может быть определен возвращаемый тип `void`, который указывает на то, что метод никакого значения не возвращает. Метод также может возвращать выходные данные вызывающему коду через параметры `ref` и `out`.

Сигнатура метода должна быть уникальной в рамках типа. Сигнатура метода

включает в себя имя метода и типы параметров (но не содержит *имена* параметров и возвращаемый тип).

Методы, сжатые до выражений

Метод, который состоит из единственного выражения, следующего вида:

```
int Foo (int x) { return x * 2; }
```

можно записать более кратко как *метод, сжатый до выражения* (expression-bodied method). Фигурные скобки и ключевое слово `return` заменяются комбинацией `=>`:

```
int Foo (int x) => x*2;
```

Функции, сжатые до выражений, могут также иметь возвращаемый тип `void`:

```
void Foo (int x) =>Console.WriteLine (x);
```

Перегрузка методов

Тип может перегружать методы (иметь несколько методов с одним и тем же именем) при условии, что типы параметров отличаются. Например, все перечисленные ниже методы могут сосуществовать внутри одного типа:

```
void Foo (int x);
void Foo (double x);
void Foo (int x, float y);
void Foo (float x, int y);
```

Локальные методы (C# 7)

В версии C# 7 можно определять метод внутри другого метода:

```
void WriteCubes()
{
  Console.WriteLine (Cube (3));
  int Cube (int value) => value * value * value;
}
```

Локальный метод (`Cube()` в данном случае) будет видимым только для охватывающего метода (`WriteCubes()`). Это упрощает содержащий тип и немедленно подает сигнал любому просматривающему код, что `Cube()` больше нигде не применяется. Локальные методы могут обращаться к локальным переменным и параметрам охватывающего метода, что имеет несколько последствий, которые описаны в разделе “Захватывание внешних переменных” на стр. 102.

Локальные методы могут появляться внутри функций других видов, таких как средства доступа к свойствам, конструкторы и т.д., и даже внутри других локальных методов. Локальные методы могут быть итераторными или асинхронными. Модификатор `static` недействителен для локальных методов; они будут неявно статическими, если охватывающий метод статический.

Конструкторы экземпляров

Конструкторы выполняют код инициализации класса или структуры. Конструктор определяется подобно методу за исключением того, что вместо имени метода и

возвращаемого типа указывается имя типа, к которому относится этот конструктор:

```
public class Panda
{
  string name;                    // Определение поля
  public Panda (string n)         //  Определение конструктора
  {
    name = n;                     //  Код инициализации
  }
}
...
Panda p = new Panda ("Petey");    // Вызов конструктора
```

В версии C# 7 конструкторы, содержащие единственный оператор, могут записываться как члены, сжатые до выражений:

```
public Panda (string n) => name = n;
```

Класс или структура может перегружать конструкторы. Один перегруженный конструктор может вызывать другой, используя ключевое слово `this`:

```
public class Wine
{
  public Wine (decimal price) {...}
  public Wine (decimal price, int year):this (price) {...}
}
```

Когда один конструктор вызывает другой, то первым выполняется *вызванный конструктор*. Другому конструктору можно передавать *выражение*:

```
public Wine (decimal price, DateTime year)
            : this (price, year.Year) {...}
```

В самом выражении не допускается применять ссылку `this`, скажем, для вызова метода экземпляра. Тем не менее, вызывать статические методы разрешено.

Неявные конструкторы без параметров

Компилятор C# автоматически генерирует для класса открытый конструктор без параметров тогда и только тогда, когда в нем не было определено ни одного конструктора. Однако после определения хотя бы одного конструктора конструктор без параметров больше автоматически не генерируется.

Неоткрытые конструкторы

Конструкторы не обязательно должны быть открытыми. Распространенной причиной наличия неоткрытого конструктора является управление созданием экземпляров через вызов статического метода. Статический метод может использоваться для возвращения объекта из пула вместо созда- ния нового объекта или для возвращения специализированного подкласса, выбираемого на основе входных аргументов.

Деконструкторы (C# 7)

В то время как конструктор обычно принимает набор значений (в виде параметров) и присваивает их полям, деконструктор делает противоположное и присваивает поля набору переменных. Метод деконструирования должен называться `Deconstruct()` и иметь один или большее число параметров `out`:

```
class Rectangle
{
  public readonly float Width, Height;
  public Rectangle (float width, float height)
  {
    Width = width;
    Height = height;
  }
  public void Deconstruct (out float width, out float height)
  {
    width = Width;
    height =  Height;
  }
}
```

Для вызова деконструктора применяется следующий специальный синтаксис:

```
var rect = new Rectangle (3, 4);
(float width, float height) = rect;
Console.WriteLine (width + " " + height);     // 3 4
```

Деконструирующий вызов содержится во второй строке. Он создает две локальных переменных и затем обращается к методу `Deconstruct()`. Вот чему эквивалентен этот деконструирующий вызов:

```
rect.Deconstruct (out var width, out var height);
```

Деконструирующие вызовы допускают неявную типизацию, так что наш вызов можно было бы сократить следующим образом:

```
(var width, var height) = rect;
```

Или просто:

```
var (width, height) = rect;
```

Если переменные, в которые производится деконструирование, уже определены, то типы вообще не указываются; это называется *деконструирующим присваиванием*:

```
(width, height) = rect;
```

Перегружая метод `Deconstruct()`, вызывающему коду можно предложить целый диапазон вариантов деконструирования.

НА ЗАМЕТКУ!

Метод `Deconstruct()` может быть расширяющим методом. Это удобный прием, если вы хотите деконструировать типы, автором которых не являетесь.

Инициализаторы объектов

Для упрощения инициализации объекта любые его доступные поля и свойства могут быть установлены с помощью *инициализатора объекта* непосредственно после создания. Например, рассмотрим следующий класс:

```
public class Bunny
{
  public string Name;
  public bool LikesCarrots, LikesHumans;
  public Bunny () {}
  public Bunny (string n) { Name = n; }
}
```

Используя инициализаторы объектов, создавать объекты `Bunny` можно так:

```
Bunny b1 = new Bunny {
                        Name="Bo",
                        LikesCarrots = true,
                        LikesHumans = false
                       };
Bunny b2 = new Bunny ("Bo") {
                                LikesCarrots = true,
                                LikesHumans = false
                            };
```

Ссылка this

Ссылка `this` указывает на сам экземпляр. В следующем примере метод `Marry()` использует ссылку `this` для установки поля `Mate` экземпляра `partner`:

```
public class Panda
{
  public Panda Mate;
  public void Marry (Panda partner)
  {
    Mate = partner;
    partner.Mate = this;
  }
}
```

Ссылка `this` также устраняет неоднозначность между локальной переменной или параметром и полем. Например:

```
public class Test
{
  string name;
  public Test (string name) { this.name = name; }
}
```

Ссылка `this` допустима только внутри нестатических членов класса или структуры.

Свойства

Снаружи свойства выглядят похожими на поля, но внутренне, как и методы, они содержат логику. Например, взглянув на следующий код, невозможно сказать, чем является `CurrentPrice` — полем или свойством:

```
Stock msft = new Stock();
msft.CurrentPrice = 30;
msft.CurrentPrice -= 3;
Console.WriteLine (msft.CurrentPrice);
```

Свойство объявляется подобно полю, но с добавлением блока `get/set`. Ниже показано, как реализовать `CurrentPrice` в виде свойства:

```
public class Stock
{
  decimal CurrentPrice;             // Закрытое "поддерживающее" поле
  public decimal CurrentPrice       // Открытое свойство
  {
    get { return CurrentPrice; }
    set { CurrentPrice = value; }
  }
}
```

С помощью `get` и `set` обозначаются *средства доступа* к свойству. Средство доступа `get` запускается при чтении свойства. Оно должно возвращать значение, имеющее тип самого свойства. Средство доступа `set` выполняется во время присваивания свойству значения. Оно принимает неявный параметр по имени `value` с типом свойства, который обычно присваивается закрытому полю (в данном случае `currentPrice`).

Хотя доступ к свойствам осуществляется таким же способом, как к полям, свойства отличаются тем, что предоставляют программисту полный контроль над получением и установкой их значений. Такой контроль позволяет программисту выбрать любое необходимое внутреннее представление, не демонстрируя детали пользователю свойства. В приведенном примере метод `set` мог бы генерировать исключение, если значение `value` выходит за пределы допустимого диапазона.

НА ЗАМЕТКУ!

В учебных целях повсеместно применяются открытые поля, чтобы излишне не усложнять примеры и не отвлекать от сути. В реальном приложении для содействия инкапсуляции предпочтение обычно отдается открытым свойствам, а не открытым полям.

Свойство будет предназначено только для чтения, если для него указано одно лишь средство доступа `get`, и только для записи, если определено одно лишь средство доступа `set`. Свойства только для записи используются редко.

Свойство обычно имеет отдельное поддерживающее поле, предназначенное для хранения лежащих в основе данных. Тем не менее, это не обязательно – свойство может возвращать значение, вычисленное на базе других данных. Например:

```
decimal currentPrice, sharesOwned;
public decimal Worth
{
  get { return currentPrice * sharesOwned; }
}
```

Свойства, сжатые до выражений

В версии C# 6 свойство только для чтения вроде показанного в предыдущем разделе можно объявлять более кратко как *свойство, сжатое до выражения* (expression-bodied property). Фигурные скобки, а также ключевые слова `get` и `return` заменяются комбинацией =>:

```
public decimal Worth => currentPrice * sharesOwned;
```

В версии C# 7 дополнительно допускается объявлять сжатыми до выражения также и средства доступа `set`:

```
public decimal Worth
{
  get => currentPrice * sharesOwned;
  set => sharesOwned = value /  currentPrice;
}
```

Автоматические свойства

Наиболее распространенная реализация свойства предусматривает наличие средств доступа `get` и/или `set`, которые просто читают и записывают в закрытое поле того же типа, что и свойство. Объявление *автоматического свойства* указывает компилятору на необходимость предоставления такой реализации. Первый пример в этом разделе можно усовершенствовать, объявив `CurrentPrice` как автоматическое свойство:

```
public class Stock
{
  public decimal CurrentPrice { get; set; }
}
```

Компилятор автоматически создает закрытое поддерживающее поле со специальным сгенерированным именем, ссылаться на которое невозможно. Средство доступа `set` может быть помечено как `private` или `protected`, если свойство должно быть доступно другим типам только для чтения.

Инициализаторы свойств

Начиная с версии C# 6, к автоматическим свойствам можно добавлять инициализаторы свойств в точности как к полям:

```
public decimal CurrentPrice { get; set; } = 123;
```

В результате свойство `CurrentPrice` получает начальное значение `123`. Свойства с инициализаторами могут допускать только чтение:

```
public int Maximum { get; } = 999;
```

Как и поля, предназначенные только для чтения, автоматические свойства, допускающие только чтение, могут устанавливаться также в конструкторе типа. Это удобно при создании неизменяемых (только для чтения) типов.

Доступность get и set

Средства доступа `get` и `set` могут иметь разные уровни доступа. В типичном сценарии применения есть свойство `public` с модификатором доступа `internal` или `private`, указанным для средства доступа `set`:

```
private decimal x; public decimal X
{
  get { return x; }
  private set { x = Math.Round (value, 2); }
}
```

Обратите внимание, что само свойство объявлено с более либеральным уровнем доступа (`public` в данном случае), а к средству доступа, которое должно быть *менее* доступным, добавлен модификатор.

Индексаторы

Индексаторы предлагают естественный синтаксис для доступа к элементам в классе или структуре, которая инкапсулирует список либо словарь значений. Индексаторы подобны свойствам, но предусматривают доступ через аргумент индекса, а не имя свойства. Класс `string` имеет индексатор, который позволяет получать доступ к каждому его значению `char` посредством индекса `int`:

```
string s = "строка";
Console.WriteLine (s[0]);       // 'с'
Console.WriteLine (s[3]);       // 'о'
```

Синтаксис использования индексов похож на синтаксис работы с массивами за исключением того, что аргумент (аргументы) индекса может быть любого типа (типов). Индексаторы могут вызываться `null`-условным образом за счет помещения вопросительного знака перед открывающей квадратной скобкой (см. лекцию «Операции для работы со значениями `null`»):

```
string s = null;
Console.WriteLine (s?[0]);       // Ничего не выводится;
                                 // ошибка не возникает.
```

Реализация индексатора

Для реализации индексатора понадобится определить свойство по имени `this`, указав аргументы в квадратных скобках. Например:

```
class Sentence
{
  string[] words = "Большой хитрый рыжий лис".Split();
  public string this [int wordNum]   //  индексатор
  {
    get { return words  [wordNum];}
    set { words [wordNum] = value;}
  }
}
```

Ниже показано, как можно было бы применять индексатор:

```
Sentence s = new Sentence();
Console.WriteLine (s[3]);       // лис
s[3] = "пес";
Console.WriteLine (s[3]);       // пес
```

Для типа можно объявлять несколько индексаторов, каждый с параметрами разных типов. Индексатор также может принимать более одного параметра:

```
public string this [int arg1, string arg2]
{
  get { ... } set { ... }
}
```

Если опустить средство доступа `set`, то индексатор станет предназначенным только для чтения, и его определение можно сократить с использованием синтаксиса, сжатого до выражения (введенного в C# 6):

```
public string this [int wordNum] => words [wordNum];
```

Константы

Константа — это статическое поле, значение которого никогда не может изменяться. Константа оценивается статически на этапе компиляции и компилятор литеральным образом подставляет ее значение всегда, когда она встречается (довольно похоже на макрос в C++). Константа может относиться к любому из встроенных числовых типов, `bool`, `char`, `string` или `enum`.

Константа объявляется с применением ключевого слова `const` и должна быть инициализирована каким-нибудь значением. Например:

```
public class Test
{
  public const string Message = "Добро пожаловать";
}
```

Константа гораздо более ограничена, чем поле `static readonly` — как в типах, которые можно использовать, так и в семантике инициализации поля. Константа также отличается от поля `static readonly` тем, что ее оценка производится на этапе компиляции. Константы можно также объявлять локально внутри метода:

```
static void Main()
{
  const double twoPI= 2 * System.Math.PI;
  ...
}
```

Статические конструкторы

Статический конструктор выполняется однократно для *типа*, а не однократно для *экземпляра*. В типе может быть определен только один статический конструктор, он должен не принимать параметры и иметь то же имя, что и тип:

```
class Test
{
  static Test() { Console.Write ("Тип инициализирован"); }
}
```

Исполняющая среда автоматически вызывает статический конструктор прямо перед тем, как тип начинает применяться. Этот вызов инициируется двумя действиями: создание экземпляра типа и доступ к статическому члену типа.

ВНИМАНИЕ!

Если статический конструктор генерирует необработанное исключение, тогда тип, к которому он относится, становится непригодным в жизненном цикле приложения.

Инициализаторы статических полей запускаются непосредственно *перед* вызовом статического конструктора. Если тип не имеет статического конструктора, то инициализаторы полей будут выполняться перед тем, как тип начнет использоваться — или *в любой момент раньше* по прихоти исполняющей среды.

Статические классы

Класс может быть помечен как `static`, указывая на то, что он должен состоять исключительно из статических членов и не допускать создание подклассов на своей основе. Хорошими примерами статических классов могут служить `System.Console` и `System.Math`.

Финализаторы

Финализаторы — это методы, предназначенные только для классов, которые выполняются до того, как сборщик мусора освободит память, занятую объектом с отсутствующими ссылками на него. Синтаксически финализатор записывается как имя класса, предваренное символом ~:

```
class Class1
{
  ~Class1() { ... }
}
```

Финализатор транслируется в метод, который переопределяет метод `Finalize()` класса `object`. ***Сборка мусора и финализаторы подробно обсуждаются в главе 12 книги C# 7.0. Справочник. Полное описание языка. Албахари***

В версии C# 7 финализаторы, состоящие из единственного оператора, могут быть записаны с помощью синтаксиса сжатия до выражения.

Частичные типы и методы

Частичные типы позволяют расщеплять определение типа, обычно разнося его по нескольким файлам. Распространенный сценарий предполагает автоматическую генерацию частичного класса из какого-то другого источника (например, шаблона Visual Studio) и последующее его дополнение вручную написанными методами. Например:

```
// PaymentFormGen.cs - сгенерирован автоматически
partial class PaymentForm { ... }

// PaymentForm.cs - написан вручную
partial class PaymentForm { ... }
```

Каждый участник должен иметь объявление `partial`.

Участники не могут содержать конфликтующие члены. Скажем, конструктор с теми же самыми параметрами повторять нельзя. Частичные типы распознаются полностью компилятором, а это значит, что каждый участник должен быть доступным на этапе компиляции и располагаться в той же самой сборке.

Базовый класс может быть указан для единственного участника или для множества участников (при условии, что для каждого из них базовый класс будет тем же). Кроме того, для каждого участника можно независимо указывать интерфейсы, подлежащие реализации. Базовые классы и интерфейсы рассматриваются в лекциях «Наследование» и «Интерфейсы».

Частичные методы

Частичный тип может содержать *частичные методы*. Они позволяют автоматически сгенерированному частичному типу предоставлять настраиваемые точки привязки для ручного написания кода. Например:

```
partial class PaymentForm // В файле автоматически сгенерированного кода
{
  partial void ValidatePayment (decimal amount);
```

```
}
partial class PaymentForm       // В файле написанного вручную кода
{
  partial void ValidatePayment (decimal amount)
  {
    if (amount > 100) Console.Write ("Дорого!");
  }
}
```

Частичный метод состоит из двух частей: *определение* и *реализация*. Определение обычно записывается генератором кода, а реализация — вручную. Если реализация не предоставлена, то определение частичного метода при компиляции валяется (вместе с кодом, в котором он вызывается). Это дает автоматически сгенерированному коду большую свободу в предоставлении точек привязки, не заставляя беспокоиться по поводу эффекта разбухания кода. Частичные методы должны быть `void`, и они неявно являются `private`.

Операция nameof

Операция `nameof` (введенная в C# 6) возвращает имя любого символа (типа, члена, переменной и т.д.) в виде строки:

```
int count = 123;
string name = nameof  (count);       // name получает значение "count"
```

Преимущество применения этой операции по сравнению с простым указанием строки связано со статической проверкой типов. Инструменты, подобные Visual Studio, способны воспринимать символические ссылки, поэтому переименование любого символа приводит к переименованию также его ссылок.

Для указания имени члена типа, такого как поле или свойство, необходимо включать тип члена.

Это работает со статическими членами и членами экземпляра:

```
string name = nameof (StringBuilder.Length);
```

Результатом будет `"Length"`. Чтобы возвратить `"String Builder.Length"`, понадобится следующее выражение:

```
nameof(StringBuilder)+"."+nameof(StringBuilder.Length);
```