Содержание

НА ЗАМЕТКУ!

Метод `Deconstruct()` может быть расширяющим методом (см. раздел "Расширяющие методы" на стр. 165). Прием удобен, 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 подобно методам внутренне они содержат логику. Например, взглянув на следующий код, невозможно выяснить, 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 выбирать 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 книге 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

Начиная с 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;
```

Начиная с 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. раздел "Операции для работы со значениями null" на стр. 60):

```
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 (начиная с версии C# 6):

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

Использование индексов и диапазонов с помощью индексаторов (C# 8)

Поддерживать в своих классах индексы и диапазоны (см. раздел "Индексы и диапазоны (C# 8)" на стр. 41) можно за счет определения индексатора с типом параметра `Index` или `Range`. Мы можем расширить предыдущий пример, добавив в класс `Sentence` следующие индексаторы:

```
public string this [Index index] => words [index];
public string[] this [Range range] => words [range];
```

Затем можно будет поступать так:

```
Sentence s = new Sentence();
Console.WriteLine (s [^1]);        // лис
string[] firstTwoWords = s [..2];  // (Большой, хитрый)
```

Статические конструкторы

Статический конструктор выполняется однократно для *типа*, а не однократно для *экземпляра*. В типе может быть определен только один статический конструктор, он не должен принимать параметры, и обязан 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 статических 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() { ... }
}
```

Компилятор C# транслирует финализатор в метод, который переопределяет метод `Finalize()` класса `object`. Сборка мусора и финализаторы подробно обсуждаются в главе 12 книги *C# 8.0. Справочник. Полное описание языка.*

Начиная с 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 разделах “Наследование” на стр. 91 и “Интерфейсы” на стр. 109.

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: *определения* и *реализации*. Определение обычно записывается генератором кода, а реализация — вручную. Если реализация не предоставлена, то оп-

ределение частичного метода при компиляции удаляется (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` (появившаяся в версии 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 данной 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, что работает со статическими членами и членами экземпляра:

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

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

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

Наследование

Класс может быть *унаследован* от другого класса с целью расширения или настройки исходного класса. Наследование от класса позволяет повторно использовать функциональность данного класса вместо ее построения с нуля. Класс может наследоваться только от одного класса, но сам может быть унаследован множеством классов, формируя иерархию классов. В этом примере мы начнем с определения класса по имени `Asset`:

```
public class Asset { public string Name; }
```

Далее мы определим классы `Stock` и `House`, которые будут унаследованы от `Asset`. Классы `Stock` и `House` получат все, что имеет `Asset`, плюс любые дополнительные члены, которые в них будут определены:

```
public class Stock : Asset     // унаследован от Asset
{
  public long SharesOwned;
}
public class House : Asset     // унаследован от Asset
{
  public decimal Mortgage;
}
```

Вот как можно работать с данными классами:

```
Stock msft = new Stock { Name="MSFT",
                         SharesOwned=1000 };
Console.WriteLine (msft.Name);          // MSFT
Console.WriteLine (msft.SharesOwned);   // 1000
House mansion = new House { Name="Mansion",
                            Mortgage=250000 };
Console.WriteLine (mansion.Name);       // Mansion
Console.WriteLine (mansion.Mortgage);   // 250000
```

Подклассы `Stock` и `House` наследуют свойство `Name` от *базового класса* `Asset`.

Подклассы также называются *производными классами*.

Полиморфизм

Ссылки являются полиморфными. Это значит, что переменная типа *X* может ссылаться на объект, относящийся к подклассу *X*. Например, рассмотрим следующий метод:

```
public static void Display (Asset asset)
{
  System.Console.WriteLine (asset.Name);
}
```

Метод `Display()` способен отображать значение свойства `Name` объектов `Stock` и `House`, т.к. они оба являются `Asset`. В основе работы полиморфизма лежит тот факт, что подклассы (`Stock` и `House`) обладают всеми характеристиками своего базового класса (`Asset`). Однако обратное утверждение не будет верным. Если метод `Display()` переписать так, чтобы он принимал `House`, то передача ему `Asset` станет невозможной.