Исключения и обработка ошибок

Юрий Литвинов
y.litvinov@spbu.ru

1. Data-driven testing

Начнём мы внезапно с продолжения про модульное тестирование, а именно с подхода «Data-driven testing», то есть тестирования, управляемого данными. Часто бывает так, что один и тот же тест бывает полезно проверить на нескольких разных наборах данных. А если принять во внимание тот факт, что объекты — это тоже данные, можно несколько разных реализаций одной и той же абстракции проверять одним тестом. Например, в домашнем задании было про реализацию стека двумя разными способами — правильное решение включало в себя один интерфейс IStack и два разных класса (например, ArrayStack и ListStack), которые этот интерфейс реализуют. Это две реализации одной и той же абстракции, так что должны внешне вести себя абсолютно одинаково. А тогда можно написать один набор тестов, которые бы проверяли сразу и одну реализацию, и другую.

Мудрые, но необразованные люди написали бы тест с циклом for, который бы просто последовательно вызывал проверку для двух вариантов стека. Однако современные библиотеки модульного тестирования готовы сделать это за вас, в частности, в NUnit есть атрибут TestCaseSource специально для этого:

```
public class StackTest
{
    [TestCaseSource(nameof(Stacks))]
    public void StackShouldNotEmptyAfterPush(IStack stack)
    {
        stack.Push(1);
        Assert.IsFalse(stack.IsEmpty());
    }

    private static IEnumerable<TestCaseData> Stacks
        => new TestCaseData[]
        {
            new TestCaseData(new ArrayStack()),
            new TestCaseData(new ListStack()),
        };
}
```

Класс TestCaseData представляет собой данные, на которых надо запустить тест. TestCaseSource в качестве параметра принимает имя свойства, откуда надо брать эти дан-

ные. Ну а свойство Stacks возвращает массив TestCaseData, содержащий как раз две реализации стека. Содержимое TestCaseData передастся в качестве параметра тесту, в нашем случае StackShouldNotEmptyAfterPush — он запустится дважды, для ArrayStack и для ListStack.

TestCaseData может содержать в себе произвольное количество данных, и тестовый метод, соответственно, может иметь больше одного параметра. Это можно использовать, чтобы ещё больше параметризовать тест, а тот факт, что массив TestCaseData можно формировать динамически и этим занимается обычный код на C#, позволяет параметризовать тесты очень гибко. Например, часто бывает ситуация, когда у нас есть два варианта реализации какого-то интерфейса, и некоторое количество данных, на которых надо потестить каждую, так что тесты по идее должны запускаться на декартовом произведении этих множеств. Легко:

```
[TestCaseSource(nameof(Stacks))]
public void StackShouldNotEmptyAfterPush(int data, IStack stack)
{
    stack.Push(data);
    Assert.IsFalse(stack.IsEmpty());
}

private static IEnumerable<TestCaseData> Stacks
{
    get
    {
        var stacks = new List<IStack>() { new ArrayStack(), new ListStack() };
        var data = new List<int>() { 1, 2, 3 };
        var result = new List<TestCaseData>();
        foreach (var stack in stacks)
        {
            foreach (var item in data)
            {
                result.Add(new TestCaseData(item, stack));
            }
        }
        return result;
    }
}
```

На самом деле, с использованием методов LINQ[1] это можно записать очень коротко, буквально в пару строк, но про это чуть попозже, когда пройдём лямбды.

Для более простых случаев, не требующих хитрой генерации входных данных, есть более удобные механизмы, например, атрибут TestCase:

```
[TestCase(12, 3, 4)]
[TestCase(12, 2, 6)]
```

[1] Language Integrated Query, часть стандартной библиотеки .NET.

```
public void DivideTest(int n, int d, int q)
{
    Assert.AreEqual(q, n / d);
}
```

Или даже с явным указанием ожидаемого значения:

```
[TestCase(12, 3, ExpectedResult=4)]
[TestCase(12, 2, ExpectedResult=6)]
public int DivideTest(int n, int d)
{
    return n / d;
}
```

2. Исключения, бросание исключений

Часть ошибок ловится на этапе компиляции, однако с остальными так или иначе приходится иметь дело во время выполнения. Каждый язык имеет свои механизмы и средства для обнаружения и обработки ошибочных ситуаций. В C, например, это было соглашение о возвращаемых функциями значениях + переменная errno, по которой можно было узнать о том, что же конкретно произошло. Недостатки данного подхода:

- это не особенность языка или среды, а лишь соглашение между разработчиками, на которое при желании вполне можно забить;
- можно забыть или не знать о каких-то особенностях возвращаемых значений;
- если проверять значение, возвращаемое каждым вызовом, читаемость кода резко падает.

Создавать большие, надёжные и в то же время простые системы с таким подходом очень сложно.

Концепция обработки исключений начала зарождаться ещё в 60-х годах с развитием операционных систем или даже с появлением в языках программирования операции goto. Исключения в C# работают идеологически так же, как и исключения в Java, которые основаны на исключениях C++, которые в свою очередь берут начало от исключений языка Ada.

В том месте, где случилось что-то нехорошее, мы можем и не знать, что же с этой ситуацией делать. Но продолжать работу дальше нельзя — кто-то где-то (если не в этом контексте, то где-то выше, например, в вызывающем методе внешнего объекта) обязан сделать что-то с имеющейся ситуацией. Но не каждая ошибка должна генерировать исключение. Общая рекомендация такова, что исключительными являются ситуации, которые мы не можем обработать в данном контексте. В таком случае необходимо передать управление и ответственность за разрешение этой проблемы обработчику исключений. Например, перед операцией деления можно проверить, не является ли делитель нулём. Но что значит нулевой делитель в данном контексте? Нарушено ли какое-то важное правило бизнес-логики, или это можно как-то локально исправить и пойти дальше?

Когда генерируется (бросается) исключение, на куче создаётся специального вида объект и управление передаётся обработчику исключений, назначение которого заключается в обработке подобных ситуаций и восстановлении корректности внутреннего состояния программы. Например, у нас есть ссылка на некий объект t, и мы хотим убедиться, что она была проинициализирована.

```
if (t == null)
{
    throw new NullReferenceException();
}
```

На самом деле дотнет-машина сама умеет бросать это исключение (как и многие другие), и каждую ссылку проверять не нужно. Если исключение не поймать, программа аварийно завершится («упадёт»), и пользователю покажут окошко с предложением отправить отчёт об ошибке разработчикам, или что-то вроде, в зависимости от ОС.

У объектов исключений обычно есть три конструктора — конструктор по умолчанию, конструктор с одним строковым аргументом, в котором можно передать сообщение, конкретизирующее произошедшее, и конструктор, принимающий помимо сообщения ещё внутреннее исключение. Про внутренние исключения будет немного позже, пока вот пример использования второго конструктора:

```
throw new NullReferenceException("Something is very wrong");
```

Это сообщение, если исключение останется необработанным, потом может быть показано пользователю в каком-нибудь окошке, или, если исключение обработать, записано в лог, или ещё куда-нибудь.

Ключевое слово throw инициирует следующие события:

- создаётся объект-исключение;
- этот объект неким образом «возвращается» из метода, хотя тип возвращаемого значения метода, естественно, другой; выполнение метода прерывается;
- на стеке ищется метод, содержащий обработчик для данного типа исключений, управление передаётся обработчику; все методы между методом, бросившим исключение, и методом, «поймавшим» исключение, также прерываются;
- если обработчика так и не нашлось и исключение распространилось за Main, вся программа прерывается.

Так что механизм бросания исключений в некотором смысле можно рассматривать как аналогию механизма возврата значений (однако, на этом сходство заканчивается, т.к. при генерации исключения мы попадаем совсем в другое место, которое может быть очень далеко от места ошибки). throw в каком-то смысле можно понимать как return, вот только использовать его для обычного возврата значений — плохая идея.

В общем-то, бросить можно объект любого класса, унаследованного от класса Exception (корневого класса иерархии исключений .NET), или даже прямо его

(`throw new Exception("Something wrong");` тоже вполне ок), но хорошим стилем является создание отдельного класса ошибки для каждого типа исключительных ситуаций. Так обработчик исключения по типу и информации, хранящейся в объекте, сможет понять, что же произошло. Собственно, почему так, будет понятно, когда мы рассмотрим, как ловить исключения.

3. Обработка исключений

Для того, чтобы исключения ловить, разберём понятие «охраняемой области» (guarded region). Если бросить исключение внутри метода, то произойдёт выход из этого метода. Если же мы хотим остаться внутри этого метода, нужно поместить «опасный код» внутрь блока try. В языках, не поддерживающих механизмы исключений, по сути вы обязаны помещать каждый вызов функции в аналог такого блока (проверка и действия при некорректном значении). Здесь же нужен только один такой блок, это разграничивает основной, «функциональный» код и действия, которые должны осуществляться, если что-то пошло не так.

Блок обработки исключений должен идти сразу же за блоком try:

```
try
{
    // Код, который может бросать исключения
}
catch (Type1 id1)
{
    // Обработка исключения типа Type1
}
catch (Type2 id2)
{
    // Обработка исключения типа Type2
}
catch (Type3 id3)
{
    // Обработка исключения типа Type3
}
finally
{
    // Код, который исполняется всегда, было брошено исключение или нет
}
```

Каждый catch — это как отдельный метод с единственным аргументом, типа ожидаемого исключения. При возникновении исключения обработчики просматриваются по порядку и управление передаётся первому, тип аргумента которого совпадает с типом объекта исключения (с учётом наследования классов, то есть сам тип и все его наследники подойдут). Из этого, кстати, следует, что типы исключений надо располагать от более частного к более общему, иначе поймается общим обработчиком, а у частного даже не будет шансов (в C# это, кстати, ошибка компиляции). Из этого же следует, что чтобы поймать все

исключения вообще, надо ловить по типу Exception, поскольку от него наследуются все остальные (и, разумеется, располагать его в самом низу последовательности catch-ей).

Каждый блок catch можно понимать как функцию, которая принимает параметром объект-исключение указанного типа и ничего не возвращает. Нахождение нужного обработчика путём сопоставления реального объекта-исключения с типами-исключениями, принимаемыми обработчиками, сродни такому понятию функционального программирования, как сопоставление шаблонов (pattern matching). Это, кстати, приводит к тому, что исключения, определяемые в программе, как правило, никакой содержательной информации, кроме собственно своего типа, не имеют. Тем более, не бывает исключений, имеющих содержательные методы, поэтому пугаться класса, у которого есть один конструктор, и всё, не следует — он нужен лишь как объект для сопоставления с шаблоном в обработчиках.

Блок finally содержит в себе код, который будет исполняться вне зависимости от того, какой обработчик события сработал (и было ли исключение вообще). В языках без сборки мусора блок finally предназначен для того, чтобы закрыть все ресурсы и освободить выделенную память. Память в C# освобождается сборщиком мусора, однако все остальные ресурсы, такие как файлы и сетевые соединения, на вашей совести.

Вот небольшой пример работы с исключениями при чтении из файла:

```
private void ReadData(String pathName)
{
    FileStream fs = null;
    try
    {
        fs = new FileStream(pathName, FileMode.Open);
        // Делать что-то полезное
    }
    catch (IOException)
    {
        // Код восстановления после ошибки
    }
    finally
    {
        // Закрыть файл надо в любом случае
        if (fs != null)
        {
            fs.Close();
        }
    }
}
```

Тут мы внутри охраняемой области создаём поток, позволяющий читать из файла или писать в файл, что-то с ним делаем, и если произошло какое-то исключение, связанное с файлом, мы что-то делаем (например, пишем сведения об исключении в лог и заканчиваем работу). В любом случае в блоке finally мы закрываем файл. На самом деле, так делать не надо, потому что есть блок using, который закроет файл сам даже в случае исключения, но про это чуть попозже.

Ещё в catch может быть фраза when, которая позволяет фильтровать исключения в зависимости от их свойств или вообще от состояния системы (после when может быть любое булевое выражение). Вот пример:

```
public static string MakeRequest()
{
    try
    {
        var request = WebRequest.Create("http://se.math.spbu.ru");
        var response = request.GetResponse();
        var stream = response.GetResponseStream();
        var reader = new StreamReader(stream);
        var data = reader.ReadToEnd();
        return data;
    }
    catch (WebException e)
        when (e.Status == WebExceptionStatus.ConnectFailure)
    {
        return "Connection failed";
    }
}
```

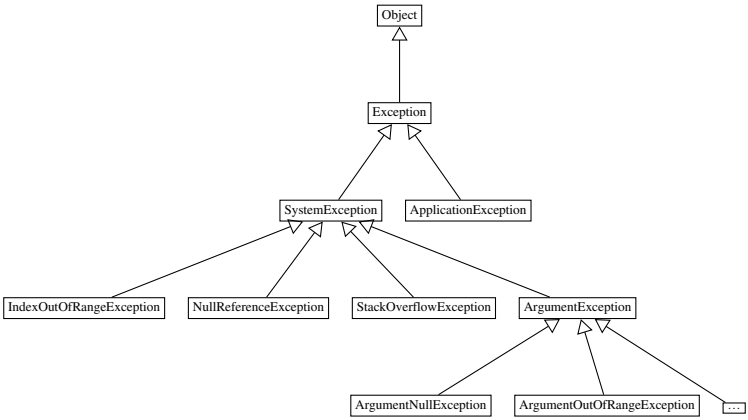
Этот кусок кода скачивает HTML-страницу с сайта `http://se.math.spbu.ru`, и, если произошло исключение работы с сетью, которое в свойстве Status имеет WebExceptionStatus.ConnectFailure, мы возвращаем строку, означающую, что соединение не установлено. Этот пример на самом деле тоже неправильный, потому что соединение не закрыто, но правильно писать через using, а про это немного потом.

4. Иерархия классов-исключений

Все исключения в .NET объекты (как и всё в .NET), поэтому они все наследуются от Object. Исключения — ссылочные типы, поэтому выделяются на куче (поэтому бросать новое исключение в обработчике OutOfMemory может быть плохой идеей). Все исключения наследуются от класса System.Exception, от которого в библиотеке .NET наследуются два класса: SystemException и ApplicationException. Изначально идея была такой, что от SystemException наследуются только исключения, бросаемые стандартной библиотекой, а от ApplicationException наследуются все пользовательские исключения. Это давало бы возможность ловить ApplicationException и знать, что это исключение, которое точно бросили мы, и ловить SystemException зная, что это бросили не мы, что означает аварийную ситуацию или ошибку кодирования.

Однако такое разделение оказалось не очень полезно, потому как в пользовательском коде рекомендовалось бросать библиотечные исключения (наследников SystemException), такие как IndexOutOfRangeException, а библиотека иногда сама кидала наследников ApplicationException. Поэтому сейчас рекомендуется не заморачиваться и наследовать свои исключения прямо от Exception, ApplicationException использовать не рекомендуется вовсе.

Вот картинка с иерархией некоторых исключений в .NET:



5. Свойства исключений

Объекты, унаследованные от Exception, имеют следующие полезные свойства:

- Data — коллекция пар ключ/значение, в которой может лежать любая дополнительная информация об исключении. На самом деле, не нужна, потому что бросать надо по возможности свои исключения, а в своих исключениях можно определить нормальные поля и свойства с этой самой дополнительной информацией;
- HelpLink — ссылка на страницу с информацией об исключении. Чтобы тот, кто с этим исключением столкнулся, знал, куда смотреть.
- InnerException — информация о предыдущем исключении, которое привело к возникновению текущего. Чуть дальше будет про перебрасывание исключений, будет понятно, что это и зачем.
- Message — текстовое описание исключения.
- Source — информация о месте, где возникло исключение (название сборки или класса). В общем-то не очень полезно, потому что есть более интересное свойство StackTrace.
- StackTrace — распечатка стека вызовов до места возникновения исключения. По нему можно понять, кто кого вызывал и где именно всё упало, очень полезно при отладке.
- HResult — свойство, выставляемое некоторым исключениям, которые были кинуты из нативного кода «под» библиотекой .NET, это что-то вроде кода ошибки. Нужно это прежде всего для взаимодействия с COM-объектами, а так как COM-объекты — это что-то из конца 90-х, то уже особо и не нужно.

Небольшой пример:

```
try
{
    throw new Exception("Something is very wrong");
}
catch (Exception e)
{
    Console.WriteLine("Caught Exception");
    Console.WriteLine("e.Message: " + e.Message);
    Console.WriteLine("e.ToString(): " + e.ToString());
    Console.WriteLine("e.StackTrace:\n" + e.StackTrace);
}
```

Вывод в данном случае будет:

```
Caught Exception
e.Message: Something is very wrong
e.ToString(): System.Exception: Something is very wrong
   в CSharpConsoleApplication.Program.Main(String[] args) в c:\Users\yurii_000\D
ocuments\Visual Studio 2012\Projects\CSharpConsoleApplication\CSharpConsoleAppli
cation\Program.cs:строка 15
e.StackTrace:
   в CSharpConsoleApplication.Program.Main(String[] args) в c:\Users\yurii_000\D
ocuments\Visual Studio 2012\Projects\CSharpConsoleApplication\CSharpConsoleAppli
cation\Program.cs:строка 15
Для продолжения нажмите любую клавишу . . .
```

Только не делайте так никогда в реальном коде, throw внутри блока try — это бессмыслица (потому что если мы можем обработать исключение прямо тут, зачем его вообще бросать).

6. Перебрасывание исключений

Допустим, мы в обработчике исключения посмотрели на текущую ситуацию и поняли, что сами мы разобраться с ошибкой не можем и её надо отправить вверх по стеку. Или мы что-то сделали, но всё равно хотим, чтобы исключение попало в обработчик, который выше по стеку. Тогда мы можем, во-первых, бросить то же исключение, во-вторых, бросить новое исключение, при желании передав ему как внутреннее исключение (свойство InnerException) наше исключение, которое мы поймали. Эти способы принципиально отличаются тем, что новый объект-исключение не будет помнить истории старого объекта (StackTrace, например), а если бросить старый объект, то его StackTrace останется неизменным (словно его никто не ловил). Перебрасывать старое исключение, например, так:

```
try
{
    throw new Exception("Something is very wrong");
}
```

```
catch (Exception e)
{
    Console.WriteLine("Caught Exception");
    throw;
}
```

Новый объект-исключение можно бросить так:

```
try
{
    throw new Exception("Something is very wrong");
}
catch (Exception e)
{
    Console.WriteLine("Caught Exception");
    throw new Exception("Outer exception", e);
}
```

Ситуация с броском нового объекта кажется довольно экзотичной, но весьма часто используется в дотнетовских библиотеках, так что прежде чем говорить «оно просто упало, и я не могу понять почему», имеет смысл посмотреть поле InnerException, там может быть написано, что именно пошло не так. Делается это, чтобы не ловить, например, тысячу разных исключений из базы данных, а одно исключение «в базе данных что-то не так», и потом уже с ним разбираться, смотря на его поле InnerException.

7. Создание своих классов исключений

Свои классы-исключения имеет смысл создавать всегда, когда вы хотите сообщить об ошибке, кроме каких-то распространённых случаев типа неправильного аргумента или некорректного состояния объекта (хотя даже в этих случаях может иметь смысл сделать свой класс-исключение). Наследовать своё исключение надо от наиболее подходящего библиотечного класса-исключения, либо прямо от Exception, если подходящего класса не нашлось. Выглядеть всё это будет примерно так:

```
public class MyException : Exception
{
    public MyException()
    {
    }

    public MyException(string message)
        : base(message)
    {
    }
}
```

Обратите внимание, что исключения довольно часто оказываются вне сборки, в которой объявлены, так что весьма желательно их делать public. Более того, исключения довольно часто оказываются даже не на той машине, на которой были брошены (сейчас клиент-серверных распределённых приложений больше, чем обычных, так что исключение, брошенное сервером, вполне может быть вынужден обрабатывать клиент) так что исключения желательно помечать атрибутом [Serializable], но пока я не рассказывал, что это такое (и, видимо, пока не буду), так что можно не заморачиваться. На самом деле, совсем идеологически правильное объявление своего исключения выглядит так:

```
[Serializable]
public class MyException : Exception
{
    public MyException() { }
    public MyException(string message) : base(message) { }
    public MyException(string message, Exception inner)
            : base(message, inner) { }
    protected MyException(
        System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context)
        : base(info, context) { }
}
```

Такую штуку Visual Studio сгенерит автоматически, если вы наберёте exc и дважды нажмёте Tab.

8. Интересные библиотечные исключения

В стандартной библиотеке есть несколько исключений, которые обрабатываются по-особенному и, если про них не знать, могут удивить.

Самые интересные исключения называются Corrupted state exceptions (CSE) и интересны они тем, что не ловятся catch-ем. Они бросаются тогда, когда .NET-машина обнаружила, что находится в некорректном состоянии, поэтому, собственно, их особо нельзя ловить — мы не можем доверять не только своему коду, но даже среде выполнения. Типичный пример такого исключения — StackOverflowException, переполнение стека. Очень популярно среди прикладых программистов, и, пожалуй, самое неприятное исключение в реальной жизни, поскольку переполнение стека не позволяет даже толком отследить, кто кого вызвал, чтобы привести систему в ошибочную ситуацию. Ещё к CSE относится AccessViolationException — то, что в C++ встречалось на каждом шагу, попытка доступа к не выделенной программе памяти. Поскольку .NET-машина гарантирует корректность обращений к памяти, то AccessViolationException из кода чисто на .NET не добиться, надо что-то делать с указателями или вызывать нативный код. И понятно, почему его бесполезно ловить — у нас может быть испорчена память программы. System.Runtime.InteropServices.SEHException — это CSE, которое представляет в коде на .NET сразу несколько исключений, бросаемых операционной системой и нативным кодом, которые тоже свидетельствуют о том, что всё очень плохо. В принципе, все эти исключе-

ния можно поймать и обработать, если хорошо попросить .NET-машину, но это делать, как правило, не нужно, так что подробности смотрите в документации.

Исключения FileLoadException, BadImageFormatException, InvalidProgramException и т.д. интересны тем, что их .NET-машина бросает сама при попытке (неудачной) загрузить сборку в процессе работы программы. Например, вы вызываете метод класса из какой-то библиотеки, только в этот момент .NET пытается загрузить библиотеку в память (в .NET нет статической линковки[2], как в C++, всё разрешение внешних зависимостей проходит во время выполнения), и если загрузка по тем или иным причинам не удалась, бросается одно из этих исключений. Например, когда файла со сборкой вообще нет или там некорректный байт-код. Такие исключения могут быть брошены, вообще говоря, при каждом обращении к какому-то типу, так что ловить их и обрабатывать обычно нет особого смысла (если только в Main-е, чтобы сообщить об ошибке и попрощаться).

Ещё исключение, которое может произойти внезапно и без всяких видимых причин — это ThreadAbortException. Кидается, когда другой поток говорит нашему потоку прерваться, при этом наш поток может заниматься своими делами и быть совершенно неготовым к тому, что его прервут. К счастью, мы пока не пишем многопоточные программы.

TypeInitializationException и TargetInvocationException кидаются, когда что-то пошло не так при инициализации класса или при вызове метода через механизм рефлексии. Они в свойстве InnerException содержат «настоящее» исключение, которое привело к фэйлу. Тоже могут появляться внезапно, но пока мы ничего такого с рефлексией не делаем, встретить их в домашках маловероятно.

OutOfMemoryException тоже может кидаться внезапно, при любом выделении памяти на куче (например, при конкатенации строк). Оно ловится, но, как правило, ловить его нет особого смысла.

Можно вообще кинуть не new КакойТоТамException, а null. Тогда .NET-машина сама бросит NullReferenceException.

Есть ещё метод Environment.FailFast — это просьба .NET-машине аварийно завершить работу программы (что-то вроде функции terminate() в C). Очень полезная штука если вы поняли, что не можете корректно восстановиться после ошибки — чем быстрее в этом случае программа будет остановлена, тем лучше. Например,

```
try
{
     // Делать что-то полезное
}
catch (OutOfMemoryException e)
{
    Console.WriteLine("Закончилась память :(");
    Environment.FailFast(
        String.Format($"Out of Memory: {e.Message}"));
}
```

[2] Однако есть single-file assembly, что довольно близко.

9. Исключения и тесты

Любая нормальная библиотека юнит-тестирования умеет проверять в тесте, что исключение было брошено. Например, в MsTest это делается с помощью атрибута ExpectedException, например:

```
[TestMethod]
[ExpectedException(typeof(System.DivideByZeroException))]
public void DivideTest()
{
    DivisionClass target = new DivisionClass();
    int numerator = 4;
    int denominator = 0;
    int actual = target.Divide(numerator, denominator);
}
```

Этот тест будет считаться успешно пройденным, если в ходе работы было брошено исключение типа System.DivideByZeroException (или любой его наследник), и, что важно, будет считаться не пройденным, если это исключение не было брошено (либо было брошено, но другое). Обратите внимание, что после кода, который должен по вашему мнению бросить исключение, никакого кода можно уже не писать, до него всё равно не дойдёт управление, если система работает корректно. И если в вашем тесте есть блоки try/catch, то, скорее всего, вы делаете что-то не так.

Более правильно проверять, что исключение брошено, прямо в месте, где оно должно быть брошено, с помощью Assert.Throws. Например,

```
[TestMethod]
public void DivideTest()
{
    var target = new DivisionClass();
    int numerator = 4;
    int denominator = 0;
    Assert.Throws<DivideByZeroException>(() => target.Divide(numerator, denominator));
}
```

Лучше это тем, что в варианте с ExpectedException тест будет считаться пройденным, если ожидаемое исключение внезапно было брошено *до* нужного места. В случае с Assert.Throws мы проверяем, что исключение брошено именно там, где должно быть. В некоторых библиотеках (например, NUnit) ExpectedException в последних версиях вообще убрали.

10. Исключения, best practices

Есть некоторые общие правила по поводу работы с исключениями, которые компилятор не проверяет, но любой нормальный программист должен про них знать и им следовать.

Самое важное из этих правил — не бросать исключения без нужды. Исключения — не альтернатива return и не средство манипуляции потоком управления программы, просто

потому, что бросание исключения — вычислительно дорогая операция. Если где-то в глубине рекурсивных вызовов может произойти что-то не ошибочное, но требующее возврата из рекурсии, тут лучше использовать коды возврата, как бы ни был велик соблазн использовать исключения, чтобы сразу свернуть весь стек до того места, которое вам нужно. Общее правило таково, что при нормальном исполнении, когда всё хорошо, исключения не должны бросаться вообще за всё время жизни программы.

Свои исключения следует наследовать от класса System.Exception, если нет более подходящего по смыслу библиотечного класса. Имеет смысл продумать заранее иерархию исключений, чтобы их удобно было ловить и обрабатывать.

Все бросаемые методом исключения стоит документировать. Это старательно делается в библиотеке .NET, и то же рекомендуется делать прикладным программистам, по крайней мере для методов, которыми будут пользоваться другие люди (например, если вы сами пишете библиотеку). Для чего — чтобы они хотя бы знали, что им надо ловить.

Нельзя ловить исключения как класс Exception, потому что это поймает их все, в том числе и NullReferenceException и прочие штуки, указывающие на ошибку кодирования. Единственное разумное исключение из этого правила — ловить их в методе Main, логировать и завершать работу программы. Продолжать работу, если произошло что-то реально неожиданное, нельзя — система находится в некорректном состоянии и там может происходить что угодно, в том числе порча пользовательских данных. По этой же причине нельзя ловить SystemException и тех его наследников, которые указывают на ошибку в коде, потому что как можно восстановиться после ошибки, если мы собственному коду не можем доверять? Лучше дать программе упасть. Например, catch (NullReferenceException) всегда неправильно, потому что всегда, где null — возможное значение ссылочной переменной, на это можно и нужно явно проверить, а не надеяться на исключение и его обработку. Просто потому, что вы думаете, что исключение произошло по одной причине, а оно произошло совершенно по другой.

По возможности выбирайте библиотечное исключение, которое вам подходит. Например, хорошими кандидатами будут InvalidOperationException для ситуации, когда класс находится в таком состоянии, когда не может выполнить запрос (например, попытка достать значение из пустого стека). Или ArgumentException, когда переданный аргумент некорректен (например, факториал от отрицательного числа). Чем больше таких проверок в программе, тем лучше, а если ваш метод будут вызывать из чужого кода (например, он часть публичного API вашей библиотеки), такие проверки вообще обязательны.

По стайлгайду имя любого класса-исключения должно заканчиваться на Exception.

При написании кода надо помнить, что бывают исключения, и заботиться об освобождении ресурсов, даже если исключение было брошено. Если вспомним про «интересные» исключения, то окажется, что это требование почти невыполнимо, ведь исключение может броситься всегда и возникает соблазн вообще весь код писать внутри try/finally. К счастью, есть языковые конструкции, которые сами просят компилятор сгенерировать finally, поэтому гарантируют освобождение ресурса.

Самая известная такая конструкция — это блок using. Класс может задекларировать, что управляет ресурсами, которые надо освободить, реализовав библиотечный интерфейс IDisposable. В этом интерфейсе есть всего один метод Dispose(), который и должен освободить ресурсы. Это самое близкое в C#, что есть к понятию «деструктор» из C++ (на самом деле, в C# тоже есть деструкторы, но они тесно связаны с Dispose() и главное, пользоваться ими надо с большой осторожностью). using позволяет определить одну или несколько

переменных, для которых гарантированно будет вызван Dispose(), как только они выйдут из области видимости:

```
public static class Program
{
    public static void Main()
    {
        // Create the bytes to write to the temporary file.
        var bytesToWrite = new Byte[] { 1, 2, 3, 4, 5 };
        // Create the temporary file.
        using (var fs = new FileStream("Temp.dat", FileMode.Create))
        {
            // Write the bytes to the temporary file.
            fs.Write(bytesToWrite, 0, bytesToWrite.Length);
        }
        // Delete the temporary file.
        File.Delete("Temp.dat");
    }
}
```

Как видим, это особенно полезно при работе с файлами — мы внутри заголовка using открываем поток ввода-вывода в файл, работаем с ним, а using позаботится о вызове метода Dispose(), который закроет файл. Вся работа с подобного рода ресурсами должна быть через using.

11. Особенности современного C#

11.1. C# 7 и старее

Дальше будет не относящаяся к теме исключений демонстрация разных возможностей языка, про которые неплохо бы знать любому C#-программисту. Первая такая возможность — кортежи и работа с кортежами, которая есть в любом нормальном функциональном языке (в том числе F#), и которой, видимо, уже никогда не будет в Java. Кортеж — это пара, тройка и т.д. значений, потенциально разного типа, на уровне библиотеки они были в языке очень давно (класс Tuple). С .NET Framework 4.7 и C# 7.0 появилась их удобная синтаксическая поддержка и, в отличие от класса Tuple, кортежи стали типами-значениями, что существенно улучшило их в плане скорости работы и нагрузки на кучу. Вот так это примерно выглядит:

```
static (int prev, int cur) Fibonacci(int n)
{
    var (prevPrev, prev) = n <= 2 ? (0, 1) : Fibonacci(n - 1);
    return (prev, prevPrev + prev);
}

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

```
{
    var (_, result) = Fibonacci(7);
    Console.WriteLine(result);
}
```

В принципе, писать имена полей не обязательно, тогда к ним всё равно будет доступ через деконструкцию (то, что в примере происходит в Main, разбор кортежа по элементам), или через свойства Item1, Item2 и т.д. кортежа. Но если имена полей написать, то можно работать с кортежем как со структурой, не Item1 и Item2, а prev и cur, что гораздо более читаемо. Можно указывать имена явно или доверить компилятору их назвать по используемым для их определения переменным:

```
int count = 5;
string label = "Colors used in the map";
var pair = (count: count, label: label);

int count = 5;
string label = "Colors used in the map";
var pair = (count, label); // Имена элементов тут будут "count" и "label"
```

Ещё, про свойства тут уже рассказывалось, но что можно было пропустить — то, что автоматические свойства можно инициализировать прямо в месте объявления, как поля:

```
public class Person
{
    public int Age { get; set; } = 0;
    public string Name { get; set; } = "Anonymous";
}
```

Ещё автоматические свойства, у которых есть только геттер, на самом деле могут быть инициализированы в месте объявления или в конструкторе, прямо как readonly-поля.

Следующая интересная фича — это операторы для работы с null-значениями. Во-первых, условный null-оператор:

```
var first = person?.FirstName;

int? age = person?.Age;
if (age.HasValue)
{
    int realAge = age.Value;
}
```

Он работает так: если слева от него не null, он выполняет разыменование и возвращает значение свойства, если слева null, то возвращает null. Интересно, что эти операторы работают и в цепочке: person?.FirstName?.Item(0) или что-нибудь в таком духе вполне будет работать. Это хорошо и понятно для ссылочных типов, но для типов-значений null не является корректным значением, поэтому просто написать int age = person?.Age; нельзя,

а вдруг null будет. Поэтому есть ещё nullable-типы int?, double? и т.д. На самом деле это структуры (то есть типы-значения), которые хранят в себе значение, если оно есть, и знают, если его нет. Их можно спросить свойствами HasValue и Value.

Есть ещё оператор ??, он возвращает значение слева, если оно не null, а иначе значение справа. Почти как тернарный оператор ... ? ... : ..., но для null-ов. Этакий хороший способ указать значение по умолчанию:

```
var otherFirst = person?.FirstName ?? "Unspecified";
```

Ещё одна относительно недавно добавленная в язык возможность — объявление локальных функций внутри метода:

```
public int Fibonacci(int x)
{
    if (x < 0)
        throw new ArgumentException("Less negativity please!", nameof(x));
    return Fib(x).current;

    (int current, int previous) Fib(int i)
    {
        if (i == 0) return (1, 0);
        var (p, pp) = Fib(i - 1);
        return (p + pp, p);
    }
}
```

Это чтобы кучу private-методов не разводить, все необходимые вспомогательные штуки упакованы внутрь метода и снаружи не видны.

В .NET бывают свойства, принимающие параметры, самый известный пример таких свойств — это свойство Item или индексовое свойство. Объявляется оно, например, так:

```
class SampleCollection {
    private int[] arr = new int[100];
    public int this[int i]
    {
        get
        {
            return arr[i];
        }
        set
        {
            arr[i] = value;
        }
    }
}
```

Потом можно коллекцию, в котором есть такое свойство, использовать как массив,

```
var collection = new SampleCollection();
collection[1] = 10;
```

Естественно, у класса может быть только один индексер.

Есть и вот такой довольно странный способ инициализации коллекции, у которой есть индексер:

```
private Dictionary<int, string> webErrors = new Dictionary<int, string>
{
    [404] = "Page not Found",
    [302] = "Page moved, but left a forwarding address.",
    [500] = "The web server can't come out to play today."
};
```

Ещё пара особенностей, касающихся передачи аргументов в методы: необязательные и именованные аргументы:

```
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");

public void ExampleMethod(int required,
    string optionalstr = "default string",
    int optionalint = 10)

anExample.ExampleMethod(3, optionalint: 4);
```

Для чего необязательные аргументы, я думаю, понятно, хотя использовать их особо не рекомендуется, значение по умолчанию может быть неожиданно для вызывающего, и вообще, плохо, когда что-то происходит само по себе. Именованные аргументы же полезны, если есть опасность перепутать порядок параметров и хочется повысить читабельность кода в месте вызова. Хотя используются они нынче довольно редко.

А вот методы с переменным числом параметров используются довольно часто. Выглядит это так:

```
public static void UseParams(params int[] list)
{
    for (int i = 0; i < list.Length; i++)
    {
        Console.Write(list[i] + " ");
    }
    Console.WriteLine();
}
...
UseParams(1, 2, 3, 4);
```

Ключевое слово params говорит, что метод может принимать параметры через запятую, и они должны складываться в массив, объявленный с params. Аргументов у метода может быть много, но параметр с params должен быть один и он должен быть объявлен в самом конце. Так, например, работает Console.WriteLine, которым все пользовались.

11.2. C# 8

В C# 8 появилось много приятного синтаксического сахара (что по моему мнению неоправданно усложнило язык, зато ещё приблизило синтаксис C# к современным функционально-объектно-ориентированным языкам, в частности, F#). Первый — это switch-выражение.

Сравните

```
public static RgbColor FromRainbow(Rainbow colorBand)
{
    switch (colorBand)
    {
        case Rainbow.Red:
            return new RGBColor(0xFF, 0x00, 0x00);
        case Rainbow.Orange =>
            return new RGBColor(0xFF, 0x7F, 0x00);
        case Rainbow.Yellow =>
            return new RGBColor(0xFF, 0xFF, 0x00);
        case Rainbow.Blue =>
            return new RGBColor(0x00, 0x00, 0xFF);
        case Rainbow.Indigo =>
            return new RGBColor(0x4B, 0x00, 0x82);
        case Rainbow.Violet =>
            return new RGBColor(0x94, 0x00, 0xD3);
        default:
            throw new ArgumentException(
                message: "invalid enum value",
                paramName: nameof(colorBand));
    };
}
```

И

```
public static RGBColor FromRainbow(Rainbow colorBand) =>
    colorBand switch
    {
        Rainbow.Red    => new RGBColor(0xFF, 0x00, 0x00),
        Rainbow.Orange => new RGBColor(0xFF, 0x7F, 0x00),
        Rainbow.Yellow => new RGBColor(0xFF, 0xFF, 0x00),
        Rainbow.Blue   => new RGBColor(0x00, 0x00, 0xFF),
        Rainbow.Indigo => new RGBColor(0x4B, 0x00, 0x82),
        Rainbow.Violet => new RGBColor(0x94, 0x00, 0xD3),
        _ => throw new ArgumentException(
                message: "invalid enum value",
                paramName: nameof(colorBand)),
    };
```

Второй вариант в полтора раза короче сам по себе, но его главное достоинство в том, что обычный switch — это оператор, так что внутри тела case-ов надо писать тоже операторы (например, return) и его нельзя использовать в выражениях. Switch expression — это именно выражение, так что может использоваться в других выражениях (этакое обобщение тернарного оператора). Все ветки должны возвращать выражение одного типа (или кидать исключение), никаких case и return писать не надо. На самом деле, это давно известный и любимый в функциональных языках (в частности, в F#) оператор match, просто его переименовали в switch, чтобы хомячкам было привычней.

Как и в функциональных языках, слева от => может стоять не просто какое-то значение, а *шаблон*, что позволяет делать довольно интересные вещи. Например, switch по значению какого-нибудь свойства:

```
public static decimal ComputeSalesTax(Address location, decimal salePrice) =>
    location switch
    {
        { State: "WA" } => salePrice * 0.06M,
        { State: "MN" } => salePrice * 0.75M,
        { State: "MI" } => salePrice * 0.05M,
        // ...
        _ => 0M
    };
```

Или даже по кортежу:

```
public static string RockPaperScissors(string first, string second)
    => (first, second) switch
    {
        ("rock", "paper") => "rock is covered by paper. Paper wins.",
        ("rock", "scissors") => "rock breaks scissors. Rock wins.",
        ("paper", "rock") => "paper covers rock. Paper wins.",
        ("paper", "scissors") => "paper is cut by scissors. Scissors wins.",
        ("scissors", "rock") => "scissors is broken by rock. Rock wins.",
        ("scissors", "paper") => "scissors cuts paper. Scissors wins.",
        (_, _) => "tie"
    };
```

Как видим, кортеж мы собираем прямо тут, но могли бы и принимать сразу пару. А потом в каждом из случаев switch-а мы разбираем кортеж по составным частям и сравниваем с константами. «_», как обычно, означает «не важно», матчится со всем и никуда не присваивает поматченное значение.

На самом деле, шаблоны можно использовать и при объявлении переменных:

```
var (x, y) = (1, 2);
```

И даже управлять для произвольных типов, как значение этого типа разбирается шаблонами:

```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);

    public void Deconstruct(out int x, out int y) =>
        (x, y) = (X, Y);
}
```

Теперь можно писать

```
var point = new Point(10, 20);
var (a, b) = point;
```

Метод Deconstruct с out-параметрами ищется компилятором по имени, и если у типа он есть (и с подходящим числом параметров), используется при сопоставлении шаблонов (out-параметры, видимо, чтобы избежать коллизий с «нормальными» методами, которые просто называются Deconstruct — нормальные методы возвращали бы кортеж).

Деконструкция, естественно, работает и в expression switch, и можно даже давать имена частям шаблона, чтобы потом их использовать после стрелки:

```
static string Quadrant(Point p) => p switch
{
    (0, 0) => "origin",
    (var x, var y) when x > 0 && y > 0 => "Quadrant 1",
    (var x, var y) when x < 0 && y > 0 => "Quadrant 2",
    (var x, var y) when x < 0 && y < 0 => "Quadrant 3",
    (var x, var y) when x > 0 && y < 0 => "Quadrant 4",
    (var x, var y) => "on a border",
    _ => "unknown"
};
```

Опять-таки, прямо как в F#, но если вы никогда не программировали на F# (не удивлюсь), просто будьте в курсе, что это не какие-то новые крутые фичи, это просто постепенное перетаскивание в C# синтаксиса и языковых особенностей более продвинутых с теоретической точки зрения «языков-родственников».

Ещё одна штука, которая была в F# сколько себя помню, и появилась в C# 8 — using var, синтаксический сахар, чтобы не писать using (var ...) и фигурные скобки. Вот так это выглядит:

```
static void WriteLinesToFile(IEnumerable<string> lines)
{
    using var file = new System.IO.StreamWriter("WriteLines2.txt");
    foreach (string line in lines)
    {
```

```
        // If the line doesn't contain the word 'Second',
        // write the line to the file.
        if (!line.Contains("Second"))
        {
            file.WriteLine(line);
        }
    }
// file is disposed here
}
```

Это то же самое, что и

```
static void WriteLinesToFile(IEnumerable<string> lines)
{
    using (var file = new System.IO.StreamWriter("WriteLines2.txt"))
    {
        foreach (string line in lines)
        {
            // If the line doesn't contain the word 'Second',
            // write the line to the file.
            if (!line.Contains("Second"))
            {
                file.WriteLine(line);
            }
        }
    }
// file is disposed here
}
```

Особенно хорошо, когда у вас несколько файлов, которые надо открыть одновременно (в старом синтаксисе код бы сполз вправо). Но опасайтесь: если вам надо закрыть файл в какой-то определённый момент (например, вы закончили в него писать, закрыли и хотите начать читать), using var может не подойти (он закроет файл только в конце функции).

А вот Nullable reference-типы, поддержка которых появилась в C# 8, это не синтаксический сахар, а реально крутая новая фича языка (ну как, в F# оно встроено в язык с самого начала...). Можно сказать компилятору, что ни одна ссылочная переменная не может иметь значение null. Зачем? Как говорил Tony Hoare (автор кусорта и значительного куска теории параллельных вычислений), изобретение null был его ошибкой, стоившей миллиарды долларов. Без nullability-анализа **любая** ссылочная переменная могла иметь значение null, поэтому формально при **каждом** разыменовании (будь то обращение к полю, вызов метода и т.п.) мы должны быть уверены, что переменная не null. Однако так, конечно, никто не делал, поэтому в любой сколько-нибудь сложной программе всегда есть куча багов, приводящих к Null Reference/Pointer Exception.

Nullability-анализ во время компиляции пытается доказать про каждую ссылочную переменную, что она не null. Например, запрещает явное присвоение null, запрещает использование неинициализированной ссылочной переменной, запрещает присвоение ссылочной переменной значения, которое каким-то образом может быть null. Как с этим жить, если

даже чтобы реализовать список, обязательно нужен null, которому было бы равно свойство Next последнего элемента? Некоторые переменные помечаются как nullable — им можно иметь значение null, в таком случае они ведут себя как обычно, но в не-nullable-переменную их так просто не присвоить. И выполнять разыменование таких переменных можно только явно сказав, что вы знаете, что делаете.

Например, `string name;` — это не nullable-переменная, использование её без инициализации или `name = null;` вызовет ошибку компиляции.

`string? name;` — это nullable-переменная, она может быть null, так что `name = null;` писать можно, но теперь `name.Length` вызовет ошибку компиляции (мы не можем доказать, что вправе обратиться к свойству). Если мы знаем, что делаем, мы можем использовать «null-forgiving operator», «!.»: `name!.Length;` работать будет. На самом деле, если все переменные сделать nullable и всегда использовать !., всё будет работать по-старому, но зачем.

По умолчанию начиная с C# 10 nullability-анализ включён, но в старых проектах есть куча кода, которая с ним просто не скомпилируется, поэтому есть опция проектного файла, которая этим управляет:

```
<Nullable>enable</Nullable>
```

Например,

```
<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net5.0</TargetFramework>
  <Nullable>enable</Nullable>
</PropertyGroup>

</Project>
```

Анализом можно управлять и пофайлово, и включать-выключать его по месту, директивами `#nullable disable` и `#nullable enable`. Однако учтите, что не null-овость ссылочной переменной во всех случаях доказать во время компиляции невозможно теоретически, так что исключения, связанные с разыменованием нулевого указателя, всё-таки возможны, но вероятность их существенно меньше.

Вообще, можно утверждать, что любой современный код должен использовать nullability-анализ, чтобы Хоар в конце концов мог быть прощён за свою ошибку. Поэтому вот подробности про nullability в C#: https://docs.microsoft.com/en-us/dotnet/csharp/nullable-references, и начиная с этой пары она должна быть обязательно включена во всех домашках.

11.3. C# 9

Самую интересную фичу C# 9 мы уже видели аж на первой паре — это top-level statement-ы, возможность писать программу без всякого boilerplate-кода типа объявления классов. Это сделано для того, чтобы на C# можно было легко писать небольшие программы и скрипты, и проще было учиться (и был хоть какой-то шанс против засилья Питона

для решения небольших задач). Для сколько-нибудь больших проектов это ничего не даёт, впрочем.

Остальные изменения направлены на то, чтобы улучшить уже существующие фичи и сделать язык более консистентным (как внутри, так и более согласованным с F#, как обычно). Во-первых, несколько улучшен синтаксис сопоставления шаблонов, теперь это целый полноценный подъязык:

```
public static bool IsLetterOrSeparator(char c) =>
    c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or '.' or ',';
```

Чаще всего, однако, это используется в простых шаблонах, например, `if (e is not null) ...`. Так более человечно, чем всякие !=.

Что ещё стало приятнее, это продвинутый локальный вывод типов. Теперь можно использовать не только var, но и не писать явно тип справа от присваивания, если понятно, о чём идёт речь. Сравните:

```
    private List<WeatherObservation> observations = new List<WeatherObservation>();
```

Тут var вам ничем не поможет, потому что var нельзя использовать при объявлении полей. Поможет C# 9, где можно писать вот так:

```
    private List<WeatherObservation> observations = new();
```

Тип может быть понятен не только при объявлении поля, но и при вызове метода, так что можно писать так:

```
public WeatherForecast ForecastFor(
        DateTime forecastDate, WeatherForecastOptions options)
...
var forecast = station.ForecastFor(DateTime.Now.AddDays(2), new());
```

Ещё это хорошо работает с init-only-свойствами, ещё одной новой фичей C# 9. init-only свойство — это свойство, из которого можно только читать, кроме момента создания объекта, объявляется так:

```
public struct WeatherObservation
{
    public DateTime RecordedAt { get; init; }
    public decimal TemperatureInCelsius { get; init; }
    public decimal PressureInMillibars { get; init; }

    public override string ToString() =>
        $"At {RecordedAt:h:mm tt} on {RecordedAt:M/d/yyyy}: " +
        $"Temp = {TemperatureInCelsius}, with {PressureInMillibars} pressure";
}
```

Обратите внимание на init; вместо set;. Такие свойства можно использовать в синтаксисе инициализации так:

```
WeatherStation station = new() { Location = "Seattle, WA" };
```

Дальше поменять значение такого свойства будет нельзя (что хорошо и правильно, всё, что может не меняться, должно быть немутабельным, const ваш лучший друг).

Ну и последняя штука, которую обычно ставят на первое место по значимости, это record-ы. Record — это по сути ссылочная структура, либо же класс, который ведёт себя как структура с точки зрения оператора сравнения и хеш-кода. Кроме того, record-ы неизменяемы. Объявляются так:

```
public record Person
{
    public string LastName { get; }
    public string FirstName { get; }

    public Person(string first, string last)
        => (FirstName, LastName) = (first, last);
}
```

Это буквально record-ы из F#, причём там они были такими изначально и F# долгое время вообще не поддерживал C#-овые структуры, что создавало массу неудобств при написании «кроссязыковых» приложений и интеграции (да-да, писать половину кода программы на C# и половину на F# вполне нормально). Сначала в F# добавили struct record-ы (то есть record-ы, но размещаемые на стеке, как в C# struct-ы), а затем, через три года, и в C# добавили F#-овые record-ы.

Зачем они нужны в обычном коде — как обычно, чтобы компилятор гарантировал немутабельность, что всегда приятно. Ну и для оптимизации — если вы хотите семантику передачи по значению, но структура просто физически большая по размеру (не знаю, 150 полей типа Int64), передавать её повсюду как struct заставит постоянно копировать данные на стеке вызовов, что медленно. При передаче record-а по ссылке копируется только указатель, всего 8 байт.

11.4. C# 10

C# 10 вводит меньше новых языковых особенностей, он нацелен прежде всего на улучшение уже существующих (чтобы не вызывать изумление, которое иногда вызывал C# 9 — иногда что-то можно, иногда то же самое, но немного в другом контексте нельзя). Вот, однако, краткое перечисление наиболее значимых новых возможностей:

- File-scoped namespaces — это, пожалуй, самая заметная и полезная фича языка. Можно писать `namespace NamespaceName;`, и дальше без лишних фигурных скобок и отступа. Беда в том, что шаблоны Visual Studio всё ещё генерят неймспейсы в старом формате, но в новом код выглядит гораздо опрятнее. Объявление неймспейса желательно делать в самом начале файла.

- global using — можно задекларировать часто используемые пространства имён ключевым словом global using в одном файле в программе (обычно там, где точка входа), и компилятор будет вести себя так, будто их using-и прописаны в каждом файле.

Чтобы using System везде не писать, удобно (но не следует увлекаться, есть риск конфликтов имён).

- Улучшенные шаблоны: например, для свойств шаблоны теперь поддерживают точку — `{ Prop1.Prop2: pattern }`.

11.5. C# 11

C# 11 ещё менее впечатляющ в плане синтаксиса, тем не менее кое-какие изменения там были:

- Шаблоны для списков: `sequence is [1, 2, 3]`. Это деятельность, направленная на унификацию разбора шаблонов с тем, что принято в функциональных языках (том же F#), но удивительно, что выражения для создания списков в такой форме появились только в C# 12.

- Сырые строки — просто удобная форма записи строкового литерала со всякими специальными символами внутри, некое обобщение записи через `@`. Вот пример:

```
string longMessage = """
This is a long message.
It has several lines.
Some are indented
        more than others.
Some should start at the first column.
Some have "quoted text" in them.
""";
```

Работает и интерполяция, правда в более страшном синтаксисе:

```
var location = $$"""
You are at {{{Longitude}}, {{Latitude}}}
""";
```

Тут количество долларов перед строкой задаёт количество фигурных скобок, которые обрамляют интерполированное значение. Если фигурных скобок меньше, они считаются частью литерала. Почему оно так, можно почитать в документации (https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-11.0/raw-string-literal (дата обращения: 03.03.2024)).

Ещё одна относительно важная вещь — это модификатор видимости file для типов. Он ограничивет область видимости файлом, в котором тип объявлен (и пришло прямиком из Kotlin, где использовалось для создания закрытых иерархий классов, которые никто извне не мог расширить своими подклассами). Формально существует, чтобы более аккуратно управлять областями видимости типов, но подозреваю, что будет использоваться более умно, например, для симуляции размеченных объединений из F# (кто не знает, что это такое, можно проигнорировать это предложение или погуглить).

```
// File1.cs
namespace NS;

file class Widget
{
}

// File2.cs
namespace NS;

file class Widget // Совсем другой Widget
{
}

// File3.cs
using NS;

var widget = new Widget(); // Ошибка
```

11.6. C# 12

C# 12 сделал ещё один шаг в сторону хорошего языка (F#) и добавил основные конструкторы (которые ждали со времён C# 7, и вот, наконец, дождались). Основной конструктор бывает у классов или структур, параметры основного конструктора становятся как будто полями класса или структуры (но есть нюанс — к ним нельзя обращаться через this), могут использоваться везде в теле класса (или структуры), инициализируются при вызове конструктора, и поэтому (чтобы не оставлять их без значений) любое создание объекта класса (или структуры) должно так или иначе вызвать основной конструктор (то есть все остальные конструкторы обязаны вызывать основной). На самом деле основные конструкторы мы уже видели у record-ов, это прямо они и есть (разве что там объявляют свойства, а тут — как бы поля). Почему «как бы поля» — на самом деле, конструктор есть функция, создающая класс, это её параметры, которые попадают в замыкание объекту, который возвращает эта функция. Когда мы подробнее поговорим про лямбда-функции, будет более понятно, пока что можно относиться к параметрам конструктора, как к полям.

Вот пример:

```
public struct Distance(double dx, double dy)
{
    public readonly double Magnitude => Math.Sqrt(dx * dx + dy * dy);
    public readonly double Direction => Math.Atan2(dy, dx);

    public void Translate(double deltaX, double deltaY)
    {
        dx += deltaX;
        dy += deltaY;
    }
```

```
    public Distance() : this(0, 0) { }
}
```

Кстати, основные конструкторы в языке F# были всегда, и там, наоборот, без них классов не бывает.

Ещё одно новшество, которое ещё немного усилий экономит — выражения-коллекции. Это быстрый способ создать список, массив или что-то похожее:

```
int[] row0 = [1, 2, 3];
int[] row1 = [4, 5, 6];
int[] row2 = [7, 8, 9];
int[] single = [.. row0, .. row1, .. row2];

foreach (var element in single)
{
    Console.Write($"{element}, ");
}
// output:
// 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Тут ещё приведён пример spread operator-а (`..`) — он берёт элементы из коллекции, к которой применяется, и вставляет их в инициализируемую коллекцию. Опять-таки, в F# такое было всегда (оно там, правда, писалось как `yield!`), в C# только появилось.