atd сегодня в 12:42

Популярные заблуждения о C#

.NET, C#, Карьера в IT-индустрии

Эта статья является развёрнутым комментарием к другой статье. Обычно я прохожу мимо, но сейчас меня почему-то задело.

Та статья представляла из себя практически «идеальную подборку заблуждений в вакууме». Причём они (заблуждения) являются довольно популярными и постоянно встречаются в различных блогах и подборках «99 вопросов для собеседования», «как пройти собеседование на джуниора» или в данном случае «шпаргалка по C#».

Почему они такие популярные? Я считаю, что потому, что они дают простые и короткие ответы, которые очень удобны для формата теста или квиза. Думать не надо, можно просто запомнить. Практикующие же разработчики повторяют их потому что эти заблуждения часто описывают наблюдаемое положение дел, но это не значит, что они истинны. Если посмотреть в окно в центре города, то можно сделать вывод что все машины — легковые, и это в большинстве случаев будет верно, но к правде имеет довольно слабое отношение.

Мне хочется, чтобы данная статья была не чисто моим мнением и не очередной компиляцией блогов. Поэтому я проведу аналогию (весьма сомнительную, но мне так захотелось) с юриспруденцией. У нас будет так:

Закон — ECMA 334 (на данный момент ревизия 5).

Законопроекты — информация о версиях языка новее, чем в ECMA 334, а также ещё не выпущенных версиях. Эту информацию можно найти на гитхабе dotnet, а так же в статьях-анонсах на MSDN.

Подзаконные акты — документация (не статьи) MSDN.

Опыт законоприменения — статьи MSDN, Wikipedia и на других сайтах, информация не абсолютная, требует проверки.

Прямой опыт — то, что мы можем просто взять и проверить. Тут нам повезло больше, чем юристам, ведь нам не придётся что-то красть, чтобы проверить на сколько лет за это посадят))

Не долго думая, возьмём план статьи отсюда и будем рассматривать мифы по списку. Если что-то забуду, добавляйте в комментах.

Фух... ПОЕХАЛИ!

Ссылочные и значимые типы (value vs reference types)

Заблуждение: ссылочные типы (reference type) хранятся в куче (heap), а значимые (value types) — на стэке.

Почему распространено? Это — простое объяснение, и оно часто тиражируется. Более того, если написать простой метод с простыми переменными одного и другого типа, чем обычно его и иллюстрируют, то всё именно так и будет.

Закон: во всём стандарте есть только 2 упоминания слова «куча», и это вполне объяснимо, поскольку стэк и куча являются деталями реализации, а не самого языка. Второе упоминание — про то, что зафиксированные (fixed) объекты могут приводить к фрагментации кучи, это нам пока не интересно. А первое упоминание — в разделе **16.1 Structs/General**, то есть общем описании, а не определении:

> However, unlike classes, structs are value types and do not require heap allocation

Это означает только то, что структуры не требуют выделения на куче (но не означает что они располагаются на стэке).

Опыт: действительно, значимые типы, являющиеся локальными переменными в *текущей реализации* скорее всего окажутся на стэке. Но если мы создадим класс, членом которого является структура, то она, как и все остальные данные этого класса окажется в куче.

Для классов (ссылочных типов) на текущий момент действительно верно, что во всех простых случаях они окажутся размещёнными в куче. Но ссылочный тип чисто теоретически можно разместить на стэке. Более того, скоро это изменится вполне официально, так же спасибо @VladD-exrabbit за вот эту ссылку: https://github.com/dotnet/runtime/issues/11192

Так чем же отличаются value и reference типы?

Читаем стандарт (в нём всё про значимые типы лежит в разделе Structs и слово struct используется для их описания):

> **16.4.2 Value semantics**
>
> **-** A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to an object that contains the data…

Тут текст про то, что структуры содержат в себе сами данные, а переменная со ссылочным типом — только ссылку. А значит структура не может в себе содержать поля, размер которых ещё не известен (в том числе своего же типа):

```
struct Node
{
    int data;
    Node next; // error, Node directly depends on itself
}
// is an error because Node contains an instance field of its own type. Another example
struct A { B b; }
struct B { C c; }
struct C { A a; }
```

> With classes, it is possible for two variables to reference the same object…

Переменные ссылочного типа могут (но не обязаны) указывать на один и тот же объект, а каждая значимая переменная содержит свою копию данных.

Это всё, что описано в разделе про *семантику*, а для языка это самое главное. Дальше идёт описание конкретно структур (16.4.3 Inheritance) а также свойства, вытекающие из семантики (16.4.4 Assignment — копирование данных при присваивании), 16.4.5 Default values — значение по умолчанию, 16.4.6 Boxing and unboxing — если нам надо передать ссылку, то требуется боксинг. А так же конструкции языка 16.4.7 Meaning of this, 16.4.8 Field initializers, 16.4.9 Constructors, 16.4.10 Static constructors, 16.4.11 Automatically implemented properties.

На собеседовании на вопрос о различии этих типов главное ответить, что у них разная семантика. Можно, конечно, погрузиться в дальнейшие различия (по списку из стандарта), но нигде среди них нет упоминания, что одно — это то, что идёт на стэк, а другое — в кучу.

stack vs heap

Заблуждение (1): стэк быстрый, а куча большая.

Почему? Потому что мелкие локальные переменные, такие как числа, обычно располагаются на стэке, а жирные объекты размещают в куче. Очевидно, что с мелкими объектами, которые известно где лежат, работать проще и быстрее.

Закон: слово *heap* мы уже искали в стандарте и ничего серьёзного не нашли. Слово *stack* в основном встречается в параграфах про unsafe-блоки и stackalloc, но мы сейчас не про это.

Факт: размещать данные на куче действительно сложнее (нужно пройти через аллокатор памяти), а на стэке — быстрее (делать ничего не надо, уже известно где и сколько памяти мы используем). Но дальнейший доступ будет аналогичен (если и то и другое мы будем трогать одинаковым способом, например по ссылке) и скорость доступа будет зависеть только от эффектов локальности данных в кэше, но это уже не вопрос языка, а физической машины.

Про размер:

Размер стэка можно поменять для всего бинарника с помощью EDITBIN.EXE /STACK:<stacksize> file.exe

А для каждого отдельного потока — через второй аргумент конструктора new Thread().

По умолчанию, куча действительно больше стэка, но стэк можно увеличить, а куча, хоть и растёт по мере надобности, но не безгранична. Есть куча правил, по которым определяется её размер, так же её можно уменьшить в настройках. Иногда доступная куча оказывается меньше физически доступной памяти Тогда приходится расчехлять unsafe и делать offheap-аллокации.

Вывод: размер и того и другого определяются машиной и рантаймом, но не является определяющим признаком.

Передача по значению / указателю

Заблуждение: значимые типы (структуры) передаются по значению а ссылочные (классы) — по ссылке.

Почему оно популярно? Честно — не знаю. Возможно из курсов Си для начинающих.

Закон:

> **10.2.5 Value parameters**
> A parameter declared without a ref or out modifier is a ***value parameter***.
>
> **10.2.6 Reference parameters**
> A parameter declared with a ref modifier is a ***reference parameter.***
>
> **10.2.7 Output parameters**
> A parameter declared with an out modifier is an ***output parameter***.

Думаю, тут всё понятно. То, как передаётся объект, определяется не его типом (ссылочный/значимый) а тем, как объявлен и передан аргумент функции. Добавляется ещё странная вещь под названием Output parameter со своей семантикой, но в реализации это такой же ref-параметр, только требующий инициализации в вызываемом методе. На этом дискуссию можно было бы закончить, но давайте немного займёмся сравнительным языкознанием.

Для сравнения я возьму Java 8-летней давности (в последний раз что-то значимое на джаве я писал примерно тогда, а с тех пор могло что-то поменяться), C++ (а не C, потому что иначе я сам запутаюсь) и C#. Я хотел тут повторить анализ целиком, но просто приведу ссылки: Java, C++, C#.

Краткий пересказ: в Java передача только по значению, но есть ссылочные типы и примитивы (это не совсем значимые типы как в C#, но для сравнения сойдёт) В C++ все типы — значимые, но можно передавать как по значению так и по ссылке (и ещё по указателю/адресу). А в C# сочетаются обе эти семантики: можно взять значимый или ссылочный тип и передать любой из них по ссылке или по значению. Это ортогональные понятия и не надо их смешивать.

P.S.: в новых версиях языка появились in-параметры. С семантической точки зрения они не определяют способ передачи (ведь при запрете изменения объекта нет никакой разницы, как он был передан), но с точки зрения реализации они работают как неизменяемые ref-параметры (readonly ref) и соответственно тоже передаются по ссылке.

string — особенный тип

Заблуждение: ведёт себя как значимый тип, а лежит в куче.

Закон:

> **9.2.5 The string type**
> The string type is a sealed class type that inherits directly from object. Instances of the string class represent Unicode character strings.
> Values of the string type can be written as string literals (§7.4.5.6).
> The keyword string is simply an alias for the predefined class System.String

Как видим, не такой уж он и особенный.

Опыт: ну да, как и все другие классы (ссылочные типы) строки *обычно* размещаются в куче. Почему говорят, что он ведёт себя как значимый тип? Я много раз такое слышал, но так и не получил чёткого ответа, почему.

Про какие особенности речь?

1. Это неизменяемый (immutable) и запечатанный (sealed) класс. Это значит, что обычными способами нельзя изменить внутри него данные и нельзя от него унаследоваться. Ну и что? Вы можете создавать классы с такими же ограничениями, ничего особенного.

2. Можно сравнивать с помощью оператора==, а обычные структуры нельзя, и для голых классов сравнивается инстанс, но не данные. Ну и что? Для любого своего класса или структуры вы можете написать такой же оператор и они будут вести себя абсолютно так же.

Более того, иммутабельность строк можно нарушить с помощью небольшой щепотки магии. И да, это уже было на хабре.

const vs readonly

Заблуждение:

· const - значение подставляется при компиляции => установить можно только до компиляции

· readonly - установить значение можно только до компиляции или в конструкторе

Закон:

> **12.20 Constant expressions**
>
> A constant expression is an expression that shall be fully evaluated at compile-time

Не «установить значение», а «значение вычисляется» (это тонкое, но важное различие).

> **15.5.3 Readonly fields**
>
> 15.5.3.1 General When a field-declaration includes a readonly modifier, the fields introduced by the declaration are ***readonly fields***. Direct assignments to readonly fields can only occur as part of that declaration or in an instance constructor or static constructor in the same class.

Да, вроде бы похоже. Но почему они в разных разделах (12 и 15)? Давайте посмотрим на название: одно — это **выражение**, а другое — модификатор **поля**. И это главное их семантическое отличие.

Отсюда вытекает важная деталь реализации:

> 15.5.3.3 Versioning of constants and static readonly fields
> Constants and readonly fields have different binary versioning semantics. When an expression references a constant, the value of the constant is obtained at compile-time, but when an expression references a readonly field, the value of the field is not obtained until run-time.

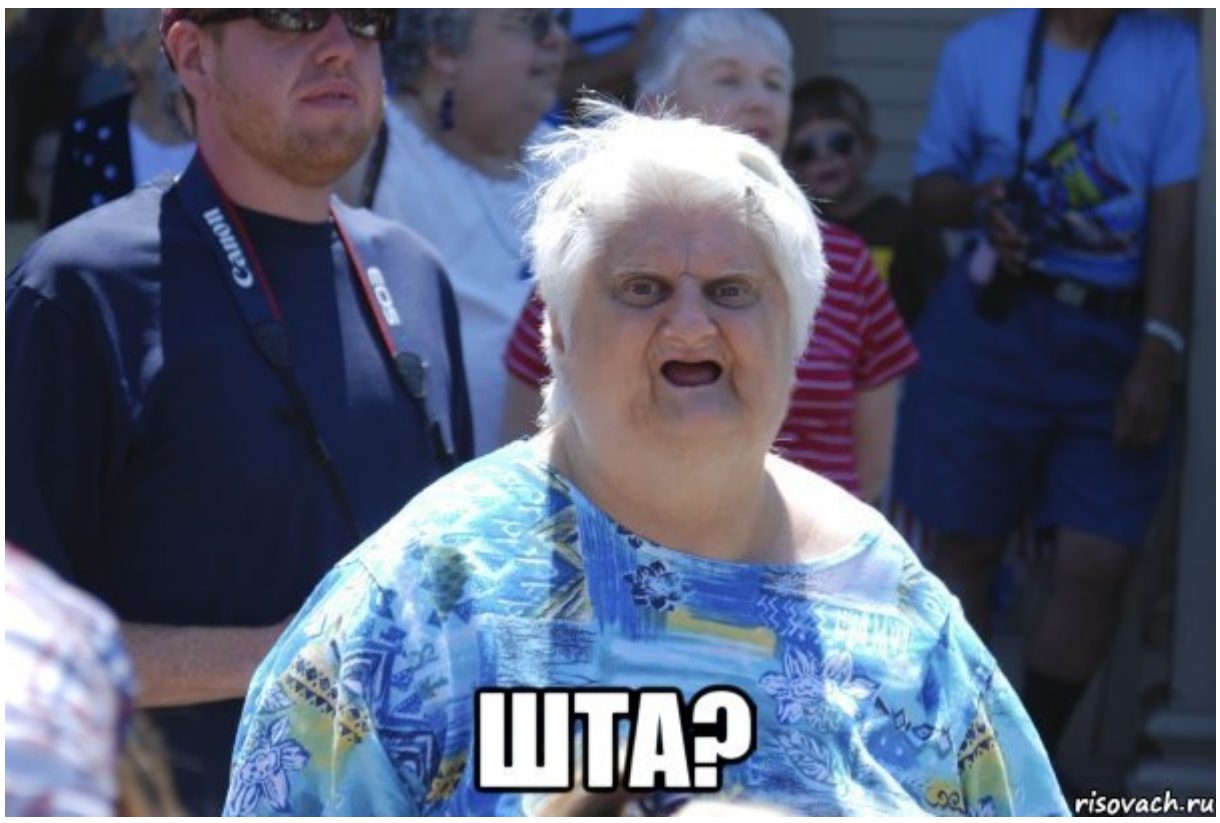
Значение константы фиксируется на момент компиляции. А статических readonly-полей (которые часто используют как замену) — на этапе выполнения. Если одна сборка зависит от другой, и берёт из неё константы и ридонли-поля, то при их изменении в первой сборке, константы во второй останутся старыми, а readonly-поля подцепятся свежие.

Факт: соберите тестовый проект, и откройте его в декомпиляторе. Вы увидите, что вместо констант встанут их значения (поэтому они и являются *выражениями*), а для readonly-полей останутся на них ссылки.

Хозяйке на заметку: до .NET Core 3 можно было поменять значение readonly-полей через рефлекшен, начиная же с этой версии такой простой способ больше не работает, но остались другие. Поменять же значения констант, не замаравшись в декомпиляции и рекомпиляции методов, у вас не выйдет.

ref и out

Из статьи-«шпаргалки»: ref и out позволяют внутри метода использовать new и для class и для struct



Факт: см выше (Передача по указателю)

> out тоже что ref, только говорит о том что, метод обязательно пересоздаст переменную

Тут, наверное, имелось в виду «переназначит», а не пересоздаст, но сильно придираться не будем.

События, делегаты

Заблуждение:

```
if (Evt != null)
    Evt("hello");
```

Закон: 15.8.2 Field-like events

```
EventHandler handler = Click;
if (handler != null)
    handler(this, e);
```

Надеюсь, разницу, объяснять не надо.

P.S.: в новом C# можно не задумываться и писать Evt?.Invoke("hello");

Finalizer (destructor) ~

Заблуждение (1): ~Foo() это «деструктор» класса Foo

Почему? Потому что по той или иной причине сами авторы языка так это называли, хотя вовремя одумались.

Закон:

> **15.13 Finalizers**
> [*Note*: In an earlier version of this standard, what is now referred to as a "finalizer" was called a "destructor". Experience has shown that the term "destructor" caused confusion and often resulted to incorrect expectations, especially to programmers knowing C++. In C++, a destructor is called in a determinate manner, whereas, in C#, a finalizer is not. To get determinate behavior from C#, one should use Dispose. *end note*]

Причина: деструкторы и финализаторы отличаются семантикой, главное отличие — детерминированность.

P.S.: в комментах появилась хорошая историческая справка, спасибо @rstm-sf.

Заблуждение (2): вызывается, когда garbage collector доберется до объекта

Почему популярно? В большинстве простых случаев это действительно именно так и происходит.

Закон:

> An instance becomes eligible for finalization when it is no longer possible for any code to use that instance. Execution of the finalizer for the instance may occur at any time after the instance becomes eligible for finalization (§8.9).

Обращаем внимание, что никаких гарантий нам тут не предоставляют.

Факт: финализатор будет вызван у объекта, если он у него есть и объект не побывал в методе SuppressFinalize до начала маркировки объектов на финализацию. Где-то между моментом определения что объект недоступен до момента фактического освобождения памяти. Но это не точно. Отменить финализацию можно с помощью метода GC.SuppressFinalize, хотя это может и не сработать. Более того, рекомендованный шаблон реализации IDisposable именно так и поступает.

Заблуждение (3): вызывается только автоматически средой .Net, нельзя вызвать самостоятельно

Факт: действительно, специального синтаксиса для этого нет. Но можно, как обычно, залезть туда через рефлекшен:

```
myObj.GetType().GetMethod("Finalize",
    BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.DeclaredOnly)
    .Invoke(myObj, null);
```

Конечно, для симуляции корректного поведения следует пройтись по всей цепочке наследования (ведь так написано в стандарте).

Singletone

Заблуждение: не забудьте про потокобезопасность и lock

Почему? Потому что часто синглтон смешивают с ленивой инициализацией, хотя это два разных паттерна, которые могут встретиться независимо друг от друга.

В стандарте очевидно ничего такого нет, так что будем опираться на здравый смысл.

Раньше собеседующий ждал тут что-то типа такого кода:

```
static Singleton singletonInstance;
static readonly Object syncRoot = new Object();
public static Singleton GetInstance()
{
    if(singletonInstance == null)
        lock(syncRoot)
            if(singletonInstance == null)
                singletonInstance = new Singleton();
    return singletonInstance;
}
```

Это известный паттерн double checked locking, он нужен для ленивой инициализации. А вопрос, напомню, стоял про синглтон.

Более того, этот код тоже имеет проблемы. Дело в том, что модель памяти работает не совсем так, как кажется на первый взгляд (это отдельная большая тема, явно не для собеседований уровня джун/миддл). Чтобы её решить, требуется или вставить volatile в нужном месте или аккуратно использовать MemoryBarrier().

Опыт: ничего из этого на самом деле не требуется для *синглтона*. Ведь рантайм нам даёт чёткие гарантии о потокобезопасности статических инициализаторов и мы просто можем написать:

```
public static Singleton Instance { get; } = new Singleton();
```

Рантайм гарантирует, что это свойство будет *потокобезопасно* проинициализировано в какой-то момент начиная от запуска и до первого использования. Причём в текущей реализации это делается с достаточной степенью ленивости, так что для 99% случаев такой простой код будет самым безопасным и надёжным. Для любителей чуть большей ленивости есть решение со вложенным классом, но всё же не полной гарантией ленивости.

Для другого паттерна, *ленивой инициализации*, в дотнете есть готовое решение — Lazy<T>, которое использует правильные для данного рантайма примитивы синхронизации.

И, значит, для оставшегося 1% случаев, когда синглтону нужна *гарантированная* ленивость, мы можем написать:

```
static readonly Lazy<Singleton> lazy = new Lazy<Singleton>(() => new Singleton());
public static Singleton Instance => lazy.Value;
```

И да, это тоже уже было на хабре, с кучей комментов.

P.S.: и так совпало, что сегодня же на хабре выложили вот такую подробную статью (18+)!

Вроде, по статье всё. Я явно что-то забыл, или написал неправильно, но для этого есть комменты, да будет срач!

Теги: ответ на статью, собеседование, буллшит

Хабы: .NET, C#, Карьера в IT-индустрии