Объектно-ориентированное программирование на C#, введение

Юрий Литвинов
yurii.litvinov@gmail.com

12.02.2021г

1. Про что этот курс

Этот семестр — про объектно-ориентированное программирование и связанные вещи, а также технологические аспекты разработки ПО. То есть, в курсе не будет (или почти не будет) новых алгоритмов, зато будет много новых концепций и конкретных технологий, которые могут пригодиться на практике. Начнём мы с ликвидации безграмотности по языку C#. Именно на примере этого языка будут объясняться основные концепции объектно-ориентированного программирования и вообще вещи, свойственные современным объектно-ориентированным языкам, но курс ни в коем случае не про C#, знания могут быть легко распространены на C++, Java, Kotlin (который, кстати, похож на C#). Дальше будет про ООП (основные понятия: объекты, классы, абстракция-инкапсуляция-наследование-полиморфизм и т.д.), про генерики, про события, лямбды, про реализацию пользовательских интерфейсов. Также будут затронуты и технологические аспекты: модульное тестирование, сборка и непрерывная интеграция, визуальное моделирование на языке UML, работа со сторонними библиотеками.

2. Отчётность

Домашек будет немного меньше, чем в первом семестре, но они будут объёмнее. Кроме того, домашек будет много в начале семестра (пока вас по другим предметам не задавило долгами), зато постепенно они сойдут на нет, а в мае домашек вообще не будет (отчасти потому, что накопятся доделки всякие, отчасти потому, что в мае всё внимание будет дипломникам и на курсовые, так что на вас будет просто не хватать времени). При этом вам в каком-то смысле не повезло, потому что мы договорились с преподом по дискретке, что раз у них в этом семестре нет практики, домашки будут больше по дискретной математике. На самом деле, где-то к середине курса наши с дискреткой пути разойдутся, к сожалению — там как раз будет много теории графов, которую нелишне попрактиковать, но к тому времени вас надо будет учить более важным штукам.

Пар будет только одна в неделю, так что особо углубляться в технические детали будет некогда, придётся больше гуглить и разбираться самостоятельно. В середине семестра

будет контрольная, в конце семестра будет зачётная работа. Как обычно, курс размещается на Blackboard, пары, по крайней мере до марта, будут дистанционными в Microsoft Teams (код команды тот же, **31ls5bh**), есть чат в Telegram (спросите у Знающих Людей, если не подписаны). Поскольку пара одна, да ещё и в удалённом формате, предполагается более активное общение вне пары, пишите по любому вопросу. Снова будут дедлайны, но раз дедлайны в 2 недели, как выяснилось после опроса, не понравились, дедлайны устанавливаете вы сами — читаете условие и пишете мне, за сколько берётесь сделать задачу. Дедлайн не может быть больше полутора месяцев, и я имею право не утвердить дедлайн (если вы обязуетесь сортировку пузырьком месяц писать, то, увы, нет). Кроме того, к зачёту надо всё уже сдать, включая исправления, так что планируйте своё время соответственно. Суммарно будет 10 домашек на примерно 70 баллов в сумме. Если не написали мне про дедлайн (например, постеснялись), действует стандартный двухнедельный. Также действует ограничение на попытки сдачи, три бесплатные, дальше -0.5 балла за каждую. При этом больше +0.5 за исправления не будет, так что если к третьей попытке есть замечания и задача оценена на 0.5 балла меньше максимума, то править её уже нет смысла. Также не забывайте про дедлайн на исправления (по умолчанию одна неделя, но тоже можно переносить). Домашки, как обычно, надо будет сдавать пуллреквестом в свой репозиторий на GitHub и ссылку выкладывать на Blackboard (если вы не записаны на курс, то слать мне на почту).

Ещё важно, что зачёт и контрольная теперь учитываются отдельно. То есть, если все домашки сдали и написали контрольную, но слили зачёт — то увы (это для поддержания интриги до конца семестра и мотивации писать контрольные хорошо).

Ориентировочно баллы надо будет набрать такие. Всего будет около 67 обязательных баллов за домашки (и где-то 7 необязательных) и по 10 за контрольную и зачёт. Так что, чтобы получить такую-то оценку, надо набрать минимум:

- A — 65 баллов за домашки, 9 за контрольную и зачёт.
- B — 62 баллов за домашки, 8 за контрольную и зачёт.
- C — 59 баллов за домашки, 7 за контрольную и зачёт.
- D — 57 баллов за домашки, 6 за контрольную и зачёт.
- E — 54 баллов за домашки, 5 за контрольную и зачёт.

Вполне возможно, что в семестре появится какая-то ещё домашняя работа (например, всё-таки по дискретке на графы), разбалловка при этом изменится.

Как в прошлом семестре, будут доклады — про разные технологии вокруг .NET. Успешный доклад даёт аж +10 баллов, что много, и даёт шанс не вылететь (но доклады даются прежде всего тем, у кого мало баллов).

В принципе, в этом курсе всё будет рассказываться с нуля, но если то, что рассказывалось на лекциях, заинтересовало и хочется почитать поподробнее про то, как всё работает, рекомендую книгу Jeffrey Richter. CLR via C#. Это фактически must read для любого профессионального .NET-программиста, хотя может быть сложновата на первом курсе. Может, к середине-концу семестра имеет смысл начать её читать. Есть ещё много разных книжек по C#, так что если что-то на парах непонятно, можно посмотреть там, например,

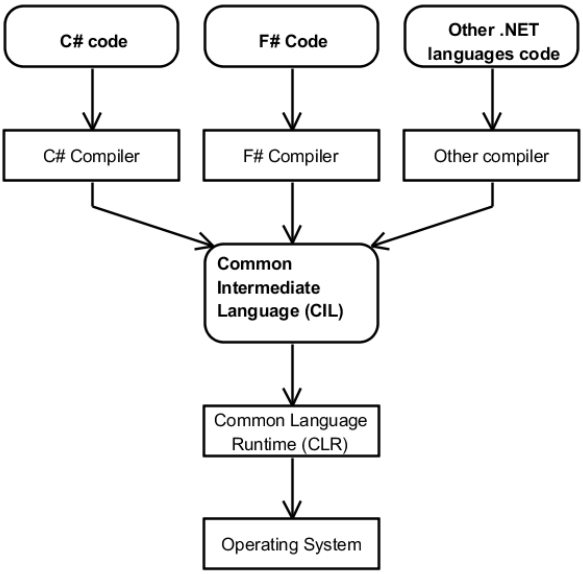
Джозеф Албахари, Бен Албахари «C# 7.0. Справочник. Полное описание языка» или Джепикс Троелсен «Язык программирования C# 7 и платформы .NET и .NET Core». Ещё есть несколько годных блогов по .NET, но рекомендовать могу, пожалуй, только официальный блог Microsoft. А вот онлайн-курсы годные есть: https://ulearn.me/ — набор открытых онлайн-курсов от СКБ «Контур» и УрГУ, они неторопливо и с большим количеством примеров и подробностей покрывают почти всю нашу программу (и даже рассказывают немного сверх, типа паттернов). Очень рекомендую, если на парах будет сложновато (а скорее всего будет, у нас очень мало времени и очень много чему надо научиться до летних школ и летних стажировок).

3. Язык C#, введение

Язык программирования C# был разработан компанией Microsoft как основной язык программирования для платформы .NET, первая версия была представлена публике в 2002 году. Впоследствии он был стандартизован ECMA (ECMA-334) и ISO (ISO/IEC 23270:2006), то есть в данный момент является международным стандартом. Последняя на данный момент версия языка, C# 9.0, опубликована 10 ноября 2020 года. Язык создавался как простой в использовании объектно-ориентированный язык общего назначения, с сильной типизацией. Используется в основном для написания прикладного ПО (включая настольные приложения, веб-приложения, он же используется для программирования под Windows Phone мобильных приложений). Всякие драйвера или системное ПО типа СУБД лучше писать на C++ (как правило, работать будет побыстрее, хотя это вопрос спорный в силу наличия в .NET развитых средств Just-in-time-компиляции). Программы, с которыми будут работать обычные пользователи — типа банковских систем, программ складского учёта, информационных систем предприятия и т.д. вполне удобно писать на C#, что очень многие и делают (язык занимает 5-е место в индексе TIOBE на февраль 2021, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html). Ближайший конкурент C# — это, скорее, Java, языки довольно похожи (поначалу были очень похожи, но теперь их пути более-менее разошлись). Принципиальная разница в том, что Java ориентирована на кроссплатформенность, C# создавался Microsoft для работы с их стеком технологий, однако же сейчас программы на C# прекрасно работают практически на чём угодно, на не-Windows есть открытая реализация почти всего .NET до версии 5 (вот только без WPF и только с частичной поддержкой WCF). В общем, весь код, что мы будем писать в этом семестре должен прекрасно работать под линуксом и под маком. В основном разработка на C# ведётся в Microsoft Visual Studio (Community Edition бесплатна и её вполне достаточно для всех практических целей), набирает популярность Rider IDE от JetBrains (https://www.jetbrains.com/rider/, он кроссплатформенный и работает на чём угодно, к тому же бесплатный для студентов. Есть ещё Visual Studio Code, которая работает под линуксом и маком, но это скорее текстовый редактор, чем полноценная IDE. Есть ещё куча текстовых редакторов, таких как Atom, Sublime, они тоже подойдут, но очень многое они не умеют.

C# называется “си шарп” примерно по тем же причинам, почему C++ называется C++ — диез в музыке указывает на то, что ноту следует играть на полтона выше, так же как ++ — оператор инкремента. C# можно рассматривать как развитие C++, при этом C# имеет ряд особенностей, которыми он радикально отличается от C++. Самая главная осо-

бенность, пожалуй, это использование виртуальной машины. Как происходит исполнение программы, написанной на C++ — исходный код компилируется сразу в исполнимый код, который линкуется в исполняемый файл, который может быть загружен в память и исполнен той операционной системой, для которой он предназначается. Другая операционная система скомпилированную программу исполнить не сможет. В C# принцип работы совсем другой, C# — один из языков, совместимых с так называемой Common Language Infrastructure. Его исходный код компилируется в последовательность команд некоей абстрактной машины (в т.н. байт-код .NET), которая потом интерпретируется специальным приложением — дотнет-машиной. Дотнет-машина должна быть реализована для каждой операционной системы, на которой нужно запускать приложения на C#, зато, поскольку система команд машины стандартизована (это тоже международный стандарт, предложенный Майкрософт и реализованный в .NET и Mono), это теоретически позволяет исполнять C#-приложение под любой операционной системой, для которой есть дотнет-машина (такой принцип называется “скомпилированное однажды, запускается везде” (compile once, run anywhere), в отличие от переносимости на уровне исходных кодов “write once, run anywhere”). Схематично такой подход можно изобразить так:



В реальности, конечно, не всё так радостно, поскольку оригинальная реализация дотнет-машины и библиотек времени выполнения от Microsoft по большей части проприетарна и ни на чём, кроме ОС Windows, не работает. Относительно недавно Microsoft выложил исходники некоторых частей дотнета (в том числе, исходники компилятора C# и основных библиотек), на это всё можно посмотреть на гитхабе (`https://github.com/dotnet`). Однако CLI (собственно, стандартизация байт-кода, дотнет-машины и библиотек поддержки) создавалась не для обеспечения переносимости, (которая была не очень-то выгодна Microsoft), а для того, чтобы иметь несколько языков программирования, работающих с одними библиотеками в одинаковой среде. Кроме C#, в байт-коды дотнет-машины компилируется Visual Basic, J# (ныне покойный), F#, ещё куча всяких языков, созданных уже не

Microsoft (например, Delphi с версии 8 умел компилироваться в байт-код .NET) — наличие единого байт-кода и виртуальной машины, его исполняющей, фактически избавляет от необходимости писать часть компилятора, относящуюся к низкоуровневой оптимизации и генерации машинного кода, уменьшая работу по созданию компилятора минимум на треть, и заодно делая все языки бинарно совместимыми совершенно даром. Бинарная совместимость означает, что код, написанный на одном языке (функция, процедура, класс, и т.д.), может быть безболезненно использован в программах, написанных на другом языке. Это очень удобно, поскольку "отвязывает" библиотеки от языка реализации, можно писать библиотеки на чём угодно, и легко ими пользоваться, и даже в одном проекте можно писать код на разных языках.

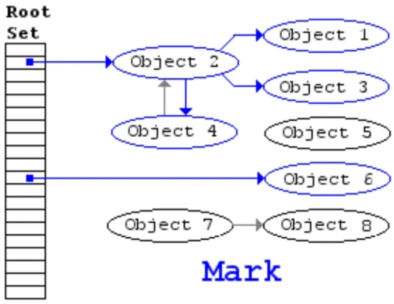
Кроме того, дотнет-машина может следить за выполнением программы более строго, чем операционная система — за выполнением "native"-программы, например, сразу же обнаружить попытку чтения или записи в чужую область памяти, выход за границы массива и т.д. Кроме того, переменные сами инициализируются, исполняемая программа имеет в себе кучу отладочной информации, и вообще, жизнь программиста делается более простой и приятной.

За такое удобство приходится платить падением скорости работы программы — ведь интерпретировать инструкцию гораздо дороже, чем просто исполнить её на процессоре. Однако с этим научились бороться, с помощью техники, называемой Just-In-Time-компиляцией. JIT-компиляция — это когда последовательность инструкций байт-кода переводится в машинный код прямо в процессе выполнения программы, после чего запоминается для последующего использования. При этом дотнет-машина может использовать для оптимизации информацию, недоступную "статическим" компиляторам — например, в процессе работы смотреть, какая функция вызывается чаще, и делать для неё инлайн-подстановку. Так что в целом, хоть хорошо написанные программы на C++, скорее всего, будут работать быстрее, дотнет-машина может выигрывать в производительности у плохо написанных программ на C++, и, поскольку время работы программиста сейчас гораздо дороже, чем время работы программы (как правило), приложения всё больше пишут на C# (ну или Яве).

Следует отметить, что виртуальную машину в качестве среды выполнения использует далеко не только C# (ну, в смысле, языки CLI, их много) — программы на Яве или Котлине тоже исполняются джава-машиной, идея виртуальной машины была предложена ещё Виртом для языка Паскаль (в 70-х годах 20-го века, так что это не суперновая технология). Причём в код Java-машины тоже можно компилировать не одну только Java — есть ещё языки, типа Скалы, и Котлина, ориентирующиеся на Java-машину как среду выполнения, или Ада — которая имеет компилятор в байт-код Java помимо нативного кода.

Следующим важным отличием C# от C++ является сборка мусора. В C++ надо было следить за выделением памяти — память, выделенную в куче, надо было освобождать вручную. В C# выделять память надо, а освобождать — нет, это сделает сама дотнет-машина, точнее, сборщик мусора (garbage collector). Дотнет-машина следит за тем, на какие области памяти ссылается программа, и если есть выделенная область памяти, на которую не ссылается больше никто, она её удалит. Опять же, это не особенность дотнет, джава-машина поступает так же, а сама идея сборки мусора предлагалась ещё в лиспе, в 1959. Концептуально сборка мусора работает так: все области памяти помечаются как "ненужные", потом все переменные на стеке вызовов помечаются как "нужные", потом все области памяти, на которые указывают переменные на стеке вызовов или их поля помечаются как "нужные",

потом все области памяти, на которые указывают “нужные” области памяти, помечаются как нужные и т.д., пока не останется указателей, на которые мы ещё не смотрели. Те области памяти, которые остались помечены как “ненужные”, очищаются (это алгоритм mark-and-sweep). Картинка из википедии, поясняющая суть:



В реальности используются более сложные алгоритмы, поскольку, во-первых, сборка мусора — это вычислительно сложная операция (средняя игра занимает в памяти до 8Гб, ходить по ней всей фактически поиском в ширину сложновато), во-вторых, требуется остановка приложения (иначе если память будет выделена или освобождена в процессе работы сборщика мусора, всё умрёт). Так что используются поколения, параллельная сборка мусора, оптимизации типа Large Object Heap и т.д., про что потом будет доклад, по идее. Сейчас важно понимать, что про delete в C# можно забыть, как про страшный сон. Естественно, организовать себе утечку памяти можно даже в C#, но это гораздо сложнее сделать случайно, для этого даже потребуется некоторое продвинутое знание языка. Что интересно, что благодаря тому, что выделение памяти выполняется виртуальной машиной из заранее выделенного пула памяти, причём это делается за константное время, программа, активно использующая кучу, на C# может работать быстрее, чем на C++ (впрочем, современные компиляторы C++ используют оптимизации, похожие на то, что делается в C#, так что кто победит трудно сказать заранее).

4. Технические детали

Собственно, традиционная программа “Hello, world”:

```
System.Console.WriteLine("Goodbye, cruel world!");
```

И всё, да. Это так называемый синтаксис операторов верхнего уровня, появившийся только в C# 9. Так что «из коробки» у вас не заработает, надо включить в настройках проекта целевую платформу .NET 5, да и в интернетах такой код вы увидите редко. Более традиционный вариант той же самой программы (как и надо было писать до C# 9, но и в новом C# это, конечно, работает):

```
using System;

namespace HelloWorld
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Goodbye, cruel world!");
        }
    }
}
```

Что мы тут видим: во-первых, работающая программа на C#, как и на любом другом объектно-ориентированном языке, представляет собой набор объектов, взаимодействующих друг с другом через чётко определённые интерфейсы посылкой и приёмом сообщений. Тип объекта — это класс, т.е. класс описывает структуру и поведение объектов. Код программы на C# представляет собой набор классов, классы организованы в нэймспейсы, свободных функций (не методов класса) в C#, в отличие от C++, на самом деле не бывает вообще. Код из C# 9, показанный выше, тоже на самом деле компилируется в класс (если интересны технические подробности, с именем `$Program`). Нэймспейс — это такое средство группировки: в одном нэймспейсе лежат связанные логически классы (ну, и другие типы), неймспейсы могут быть вложенными, позволяя создавать такую иерархическую структуру. Кстати, разделения на хедеры и с-шники в C# нет, всё, что относится к одному классу, пишется в одном файле, с расширением .cs. Есть, правда, возможность определять класс в нескольких разных файлах, через partial-классы, но это нужно для некоторых специальных целей, про которые потом. Считается хорошим тоном поддерживать соответствие «один файл — один класс». Так вот, каждый класс относится к некоторому неймспейсу, неймспейс можно и не писать, тогда класс будет считаться принадлежащим нэймспейсу по умолчанию, но это дурной тон. Одноимённые классы из разных неймспейсов можно различить, указывая *квалифицированное имя* класса: <имя нэймспейса>.<имя класса>. Кроме нэймспейса, есть ещё понятие «сборка» (assembly) — это отдельный проект, кусок кода, который собирается в отдельный .exe или .dll-файл. Один неймспейс может определяться в разных сборках, и одна сборка может иметь много неймспейсов, но тем не менее, сборки тоже работают как некое средство группировки, потому что позволяют управлять видимостью классов друг другу — можно сделать класс видимым только внутри своей сборки, а можно сделать его видимым и в других сборках тоже (типа интерфейса сборки). Вообще, создавать отдельные библиотеки и использовать их в проекте очень легко, в отличие от C, так что приложение на C# (ну или вообще любом дотнет-языке), как правило, состоит из большего числа отдельно собираемых модулей, чем C. Кстати, подключение неймспейса делается с помощью ключевого слова using, так что using System — это подключение неймспейса System, почти как #include в C. На самом деле нет, компилятор C# и так видит все объявленные классы во всех исходниках и всех подключённых библиотеках, так что #include не нужен вовсе. Видны все public-классы всех сборок, подключённых к данному проекту, во всех файлах проекта, а классы проекта видны в других файлах проекта вообще всегда, что по сравнению с C сказочно удобно.

Дальше мы видим объявление класса Program. Внутри фигурных скобочек находятся объявления полей, методов и свойств класса, вложенных классов, перечислений и т.д. В нашем случае в классе есть только метод Main, он static — то есть он является методом клас-

са, в противоположность обычным методам, которые относятся к конкретному объекту. Методы класса не используют данные, хранящиеся в объекте, так что наличие объекта им не нужно для работы. Это самое близкое, что есть в C# к обычным сишным функциям, которыми вы пользовались в прошлом семестре. Он имеет тип возвращаемого значения void, то есть, как и в C, это означает, что он не возвращает ничего. Заметим, что в отличие от C C#-овая «функция» Main код возврата сама не должна возвращать. Эта функция принимает один параметр — массив строк (то есть, объектов класса String, представляющих строки, про сишные char*, развлечения с указателями и всякие соглашения типа того, что строка кончается на 0, можно забыть, как страшный сон. То есть внутри оно всё так и есть, но хорошо скрыто, что, в частности, является преимуществом объектно-ориентированного программирования). Аналог C#-ового String – C++-ный std::string. Массивы в C# тоже не так просты, как в C, а представляют собой объекты, с полями и методами, например, у них есть свойство Length, которое содержит длину массива.

Дальше стоит вызов метода WriteLine класса Console. Он, как и следовало ожидать, печатает на консоль строку и переводит строку. У Console есть ещё много всяких полезных методов, например, Beep, и ещё больше всяких свойств, типа ForegroundColor (и BackgroundColor), и Title. Есть ещё метод Write, который делает то же, что и WriteLine, но не переводит строку. Заметьте, WriteLine может выводить всё без всякой форматной строки, но если нужна форматная строка, тоже ок: `Console.WriteLine("Hello, {0}!", "world");` или даже так: `var name = "world"; Console.WriteLine($"Hello, {name}!");`. Заметьте, никаких форматных спецификаторов не надо, C# сам догадывается о типе выводимых параметров.

Синтаксис тел методов весьма похож на C, так что скорее всего получится просто сесть и начать писать код так, будто бы это был C. Например, цикл for:

```
for (int i = 0; i < 300; ++i)
{
    Console.WriteLine("Мат-мех лучше всех!");
}
```

или так:

```
for (var i = 0; i < 300; ++i)
{
    Console.WriteLine("Мат-мех не для всех!");
}
```

Заметьте, для работы с русскоязычными строками не надо никаких setlocale или управления кодировками — все строки в C# всегда в Unicode, а все символы двухбайтовые.

Тут показана интересная фича C# — частичный вывод типов. i инициализируется интовой константой 0, так что компилятор знает, что тип переменной i должен быть интом (иначе типы не сойдутся), так что это можно не указывать и писать вместо имени типа var. Тут, правда, есть идеологический спор по поводу того, как часто надо использовать var. Некоторые говорят, что var использовать нельзя, потому что явные аннотации типов накладывают дополнительные ограничения на программу, и тем самым уменьшают вероятность ошибок и делают программу гораздо более читабельной. Некоторые говорят, что var

надо использовать всегда, поскольку информация о типе избыточна и до тех пор, пока компилятор может вывести тип, не несёт семантической нагрузки, к тому же короткое var в коде выглядит гораздо лучше, чем какое-нибудь System.Collections.Generic.Dictionary<string, int>, так что использование var делает программу гораздо более читабельной. Некоторые занимают промежуточную позицию и предпочитают использовать var только чтобы не писать длинных имён типов. Лично я люблю var, и предлагаю вам им активно пользоваться, хотя бы в тех случаях, когда тип переменной очевиден из правой части присваивания (например, `var str = new String("s");`). Кстати, в среде даже опытных программистов распространена ересь, что var ломает строгую типизацию, делая тип переменной «неопределённым» — это неправда, тип переменной в любом случае точно известен во время компиляции, ломает типизацию другая штука — dynamic, но про неё лучше забудьте прямо сейчас.

Функции задаются, например, так:

```
private static int Factorial(int n)
{
    if (n == 1)
    {
        return 1;
    }

    return n * Factorial(n - 1);
}
```

— это нужно писать в классе Program на том же уровне, что и функция Main. Использовать из функции Main, например, так:

```
Console.WriteLine(Factorial(4));
```

Кстати, компилятор многопроходный, поэтому ему плевать, выше или ниже функции Main описана наша функция.

В современном C# функции, тело которых состоит из одной строчки, можно писать вообще в одну строку, без фигурных скобок, вот так:

```
private static int Factorial(int n)
    => n <= 1 ? 1 : n * Factorial(n - 1);
```

Это называется Expression-bodied method, и когда мы дойдём до рассказа про лямбда-функции, будет понятно, почему оно так выглядит. Это на самом деле синтаксический сахар для того, чтобы писать программы кратко и красиво (хотя такой синтаксис может быть непривычен сишникам).

Обратите внимание на некоторые особенности местного стайлгайда. Вообще, стайлгайд тут не как в C, где пишут кто как умеет, а определяется исходниками от Майкрософт, так что довольно строгий (ну, вариации есть, но некоторые правила вообще общеприняты). Имена методов и свойств пишутся всегда с заглавной, имена полей всегда со строчной. Тела составных операторов, типа if-ов, считается хорошим тоном всегда оборачивать в фигурные скобки, но этому следуют не всегда. То же касается отделения закрывающей фигурной скобки пустой строкой от кода ниже. Некоторые правила стайлгайда Visual Studio применяет автоматически и без спроса, так что не удивляйтесь, если вы пишете как обычно, а

получается всё равно хорошо. Есть даже плагин к Visual Studio (и отдельный анализатор, подключаемый к проекту), который выдаёт предупреждения при нарушении общепринятого стайлгайда, https://github.com/DotNetAnalyzers/StyleCopAnalyzers. Очень рекомендую, хотя часть предупреждений придётся отключить, особенно поначалу (например, подписывать сборки мы не будем).

5. Типы данных

С типами данных в C# ситуация несколько отличается от C. Во-первых, размеры элементарных типов данных стандартизованы, так что если число байт, выделяемых под int в C зависело от реализации, то в C# это ровно 32 байта, даже если мы запускаем программу на микроволновке с 16-битным процессором. Во-вторых, что даже более важно, каждый тип данных в C# — это класс, даже int наследуется от класса System.Object и имеет методы. Хорошая новость в том, что это на прикладных программистов особо не влияет (даже делает более удобной работу с элементарными типами) — int-ы всё так же располагаются на стеке вызовов и копируются при присваивании. Подробности про это на следующей паре.

Каждый элементарный тип имеет своё ключевое слово, как в C, и ещё имя типа, объявленное в библиотеке Base Class Library (неймспейс System), которое его синоним. Так что string и System.String — это одно и то же.

У каждого типа есть значение по умолчанию, которым его переменные инициализируются сразу при объявлении. Неинициализированных переменных, с которыми приходилось бороться в C, в C# просто не бывает (хотя компилятор будет ругаться, если, например, пытаться передать как параметр локальную переменную, которой не было присвоено значение). Вообще, компилятор внимательно анализирует поток данных и выдаёт ошибку, если ему кажется, что что-то не так.

У типов-синонимов есть полезные статические методы, которые есть и у, мм, ключевых слов, им соответствующих. Например, как преобразовать строку в число:

```
var inputString = Console.ReadLine();
int number = Int32.Parse(inputString);
```

Это то же самое, что и

```
var inputString = Console.ReadLine();
int number = int.Parse(inputString);
```

В общем-то, с ООП все должны быть хотя бы примерно знакомы, но для программистов на чистом C это выглядит диковато.

Массивы:

```
int[] a = new int[10];
```

или

```
var a = new int[10];
```

```
for (var i = 0; i < a.Length; ++i)
{
    a[i] = i;
}
```

Тут всё как обычно, единственное, что массивы всегда размещаются на куче (то есть без ключевого слова new создать массив нельзя), квадратные скобочки пишутся после типа, а не после переменной, как в C, и главное, массив — это объект (имеющий, например, свойство Length, в котором хранится его длина, поэтому передавать длину вместе с массивом больше не надо). Для тех, кто в курсе, очевидна аналогия с std::vector.

Многомерные массивы:

```
int[,] numbers = new int[3, 3];
numbers[1,2] = 2;

int[,] numbers2 = new int[3, 3] { {2, 3, 2}, {1, 2, 6}, {2, 4, 5} };
```

Обратите внимание, что C# поддерживает именно многомерные массивы, а не массивы массивов, как в C. Массивы массивов тоже никто не запрещает:

```
int[][] a = new int[3][];
a[1][2] = 0;
```

— так программа упадёт (угадайте почему), а вот так будет работать:

```
int[][] a = new int[3][];
for (int i = 0; i < 3; ++i)
{
    a[i] = new int[3];
}

a[1][2] = 0;
```

Структуры:

```
struct Point
{
    public int x;
    public int y;
}
```

Обратите внимание, что видимость по умолчанию у полей структуры такая же, как и у полей класса — private, так что приходится писать public, чтобы они были доступны извне. А ещё в отличие от C++ писать модификатор видимости надо перед каждым полем и методом (ну, необязательно, но тогда видимость будет по умолчанию, и хорошим тоном считается модификатор видимости писать, даже если умолчание нас устраивает). Про всё это будет подробнее на следующей паре, структуры не так просты, как кажутся, но использовать их можно так же, как в C++.

Есть типы-перечисления:

```
enum SomeEnum
{
    red,
    green,
    blue
}
```

Использование:

```
SomeEnum a = SomeEnum.blue;
```

или так:

```
var a = SomeEnum.blue;
```

Обратите внимание, что имя енума должно предшествовать имени значения всегда, енум образует что-то вроде неймспейса для своих значений (как enum class в C++, если кто в курсе).

Константы задаются с помощью ключевого слова const, как в C++ и частично в C (но не как в Яве, где для этого есть ключевое слово final).

Всякие математические штуки находятся в классе Math неймспейса System. Например, Math.Cos(1); — раз свободных функций в языке нет, функции на самом деле не функции, а статические методы. Хорошая новость в том, что есть using static, так что префикс Math можно не писать (особенно, если погуглить, что такое using static, и начать его использовать).

6. Инструменты разработки

Есть бесплатная Visual Studio Community 2019, которую можно скачать с https://www.visualstudio.com/, есть бесплатный для студентов Rider, который работает и под виндой, и под линуксом, и под маком (правда, пока не умеет дизайнить формочки в приложениях с графическим интерфейсом, но формочки можно описывать и напрямую в коде, так что это не страшно). Оправдания типа «Я пытался сделать домашку на GtkSharp, оказалось, что он не умеет <подставить нужное>, а разобраться в WinForms я уже не успею» на зачёте приниматься не будут. Создать проект в Visual Studio можно через New Project -> C# -> Console App (.NET Core), линуксоиды сами, я надеюсь, разберутся. Ещё на Visual Studio можно поставить ReSharper (это плагин от JetBrains, бесплатный для студентов), программировать станет в разы приятнее, но только если ваше железо позволяет (он довольно ресурсоёмок и может заставить студию подтормаживать). Rider умеет то же, что и ReSharper, но, как говорят, не тормозит.