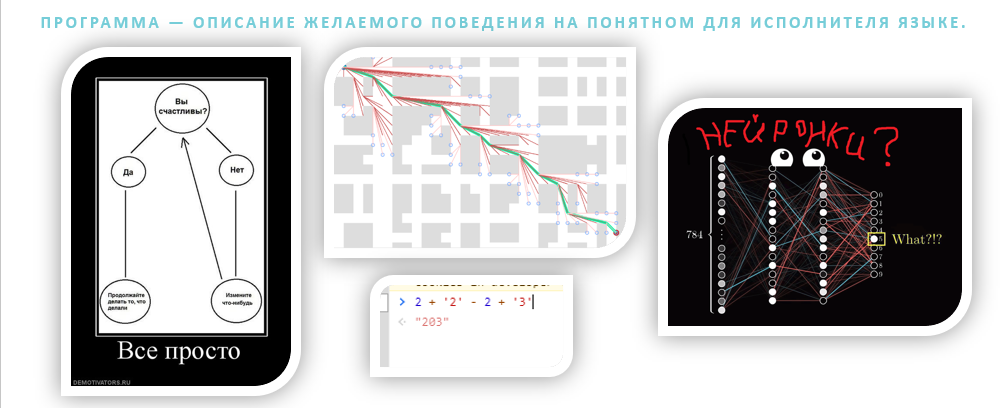
ОБЩАЯ ТЕОРИЯ

Прежде чем я расскажу вам о языке C# Давайте уточним вот такой момент: Любой язык программирования нужен для того, что бы писать на нем программы. Но что такое программы? Вот определение взятое с википедии

**Компью́терная програ́мма** — 1) комбинация компьютерных [инструкций](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4" \o "Машинный код) и [данных](https://ru.wikipedia.org/wiki/%D0%94%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5_(%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%82%D0%B5%D1%85%D0%BD%D0%B8%D0%BA%D0%B0)), позволяющая [аппаратному обеспечению вычислительной системы](https://ru.wikipedia.org/wiki/%D0%90%D0%BF%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5" \o "Аппаратное обеспечение) выполнять [вычисления](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5) или функции управления (стандарт [ISO](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D0%B4%D1%83%D0%BD%D0%B0%D1%80%D0%BE%D0%B4%D0%BD%D0%B0%D1%8F_%D0%BE%D1%80%D0%B3%D0%B0%D0%BD%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_%D0%BF%D0%BE_%D1%81%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D0%B8" \o "Международная организация по стандартизации)/[IEC](https://ru.wikipedia.org/wiki/IEC)/[IEEE](https://ru.wikipedia.org/wiki/IEEE) 24765:2010)[[1]](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0#cite_note-ISO24765-1); 2) [синтаксическая](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)" \o "Синтаксис (программирование)) единица, которая соответствует правилам определённого [языка программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F" \o "Язык программирования), состоящая из определений и [операторов или инструкций](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)" \o "Оператор (программирование)), необходимых для определённой функции, задачи или решения проблемы (стандарт ISO/IEC 2382-1:1993)[[2]](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0" \l "cite_note-ISO2382-2).

Все что происходит когда мы включаем компьютер, телефон, да хоть бы и холодильник (не уверен насчет тостеров и утюгов, но и там, наверняка есть куда копнуть) является результатом действия програм. Они окружют нас постоянно, любое, хоть сколько-нибудь сложное поведение электронного устройства – результат выполнения программы.



Хорошо, с определением мы разобрались, а из чего программы состоят? В основе любой программы лежит всего три действия:

* последовательное выполнение команд,
* ветвление,
* частный случай ветвления: цикл.

И все? Все так просто? Если бы. В реальности, на пути написания универсальных програм на все случаи жизни лежит множество препятствий. Здесь я перечислил только некоторые из них.

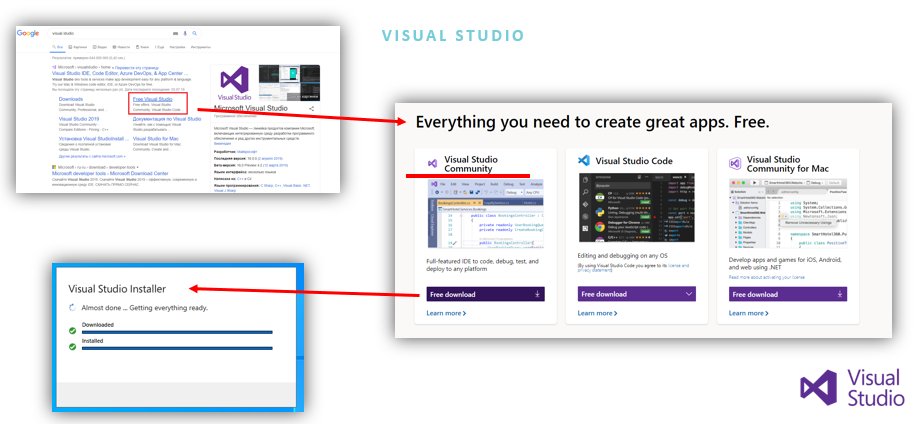


Но выход есть! Решение сложных проблем требует инструментов для борьбы со сложностью. Давате посмотрим, что должен уметь язык программирования для снижения сложности написания программ:

* Выделение абстракций делает код понятным для человека, а не только для компьютера.
* Продуманная архитектура позволяет перестраивать поведение программы в соответствии с меняющимся миром вокруг нас.
* Следование одной из парадигм прогаммирования облегчает нам построение систем на уровне концепций.
* Кросплатформенность позволяет запускать одни и те же программы под разными операционными системами.
* А специализированные среды разработки, существенно облегчают работу с кодом.

И вот здесь я представляю вам C# высокоуровневый язык программирования разработанный компанией Microsoft. У этого языка С подобный синаксис и широкое комерческое применение. Спецификация C# включает в себя великое множество инструментов, которые делают программирование комфортным, а программы понятными не только компьютеру, но и людям. C# является частью платформы .Net не будем пока останавливаться на том, что она из себя представляет и поговорим о среде разработки и основых языковых конструкциях C#.

Начнем с самого начала, нам нужна среда разработки, да можно писать код в notepad, а затем компилировать из командной строки, но мы же хотим удобства, современные программы и так достаточно сложны, давайте сразу использовать подходящие инструменты. Заходим в google и набираем visual studio. Кликаем скачать, и наслаждаемся опросником и туториалом, запускаем скачанный файл и он устанавливает нам установщик, дада, мы скачали пакет который распаковывается в установщик:



Установщик загружен, проставляем галочки на нужных нам компонентах:

* .NET desktop development
* ASP.NET and web development

Обратите внимание есть

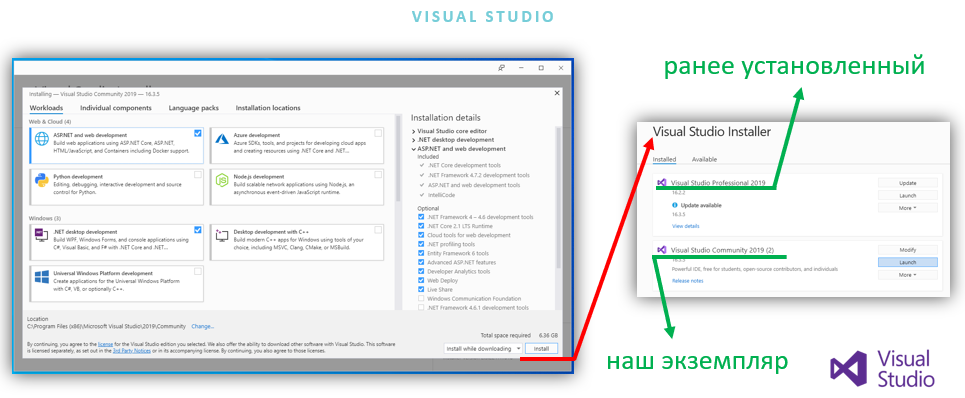
* Node.JS для работы JavaScript на сервере,
* WPF для десктопной разработки,
* Azure для облачных сервисов.

Другие языки программировния, такие как

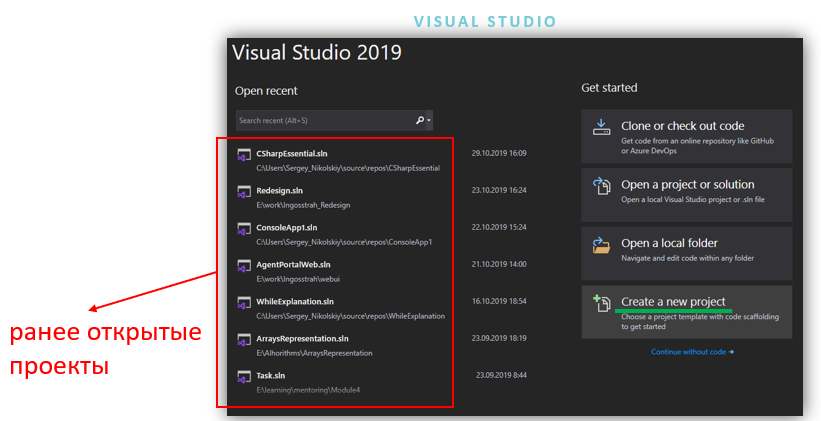
* Python,
* C++.

И всё это в бесплатной для личного пользования Community версии.

Нажимаем Install, по завершении установки запускаем visual studio. Обратите внимание, launcher знает о всех установленных на компьютере экземплярах visual studio.



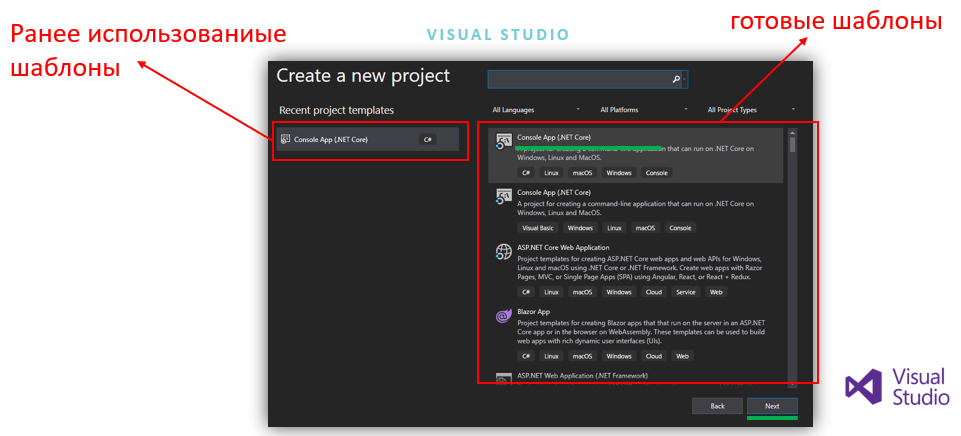
В начальном окное, мы можем выбрать ранее запущенные проекты, сейчас же нам интересна вот эта кнопка: create a new project.



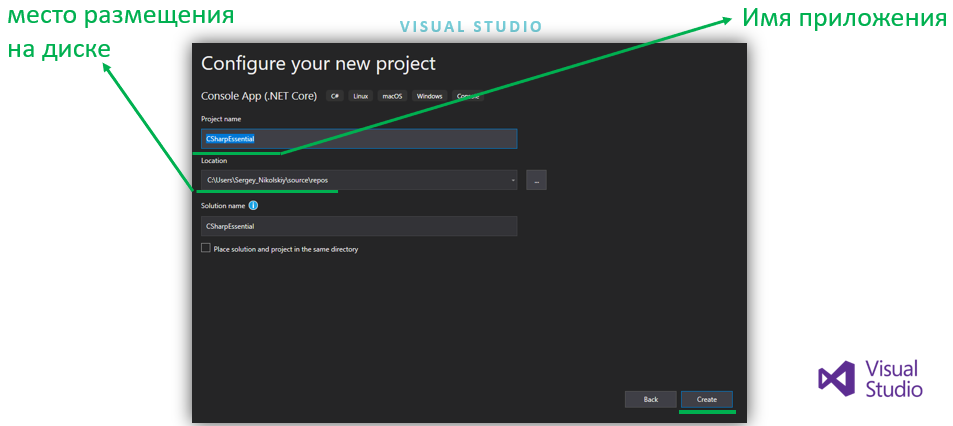
Выбираем интересующий нас шаблон, обратите внимание, каждый шаблон содержит краткое описание которое включает в себя язык, целевые платформы и тип проекта. В нашем случае это:

* язык C#
* платформы Linux macOs Windows
* тип приложения Console

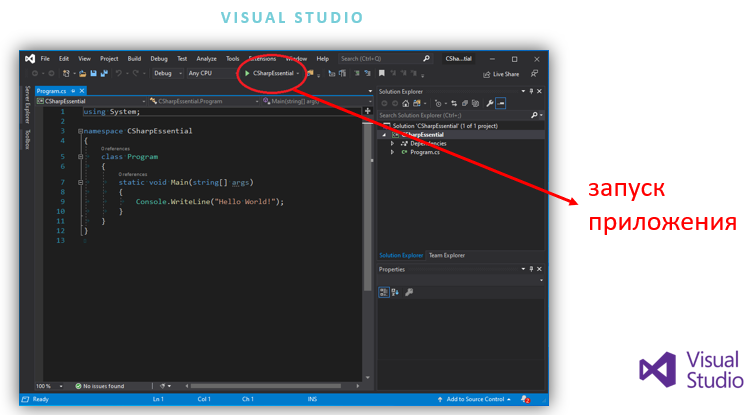
Ниже можно увидеть такой же шаблон для Visual Basic и много других готовых шаблонов. Жмем next.



Нам предлагают выбрать имя для нашего приложения и место его размещения на диске. И нажимаем create.



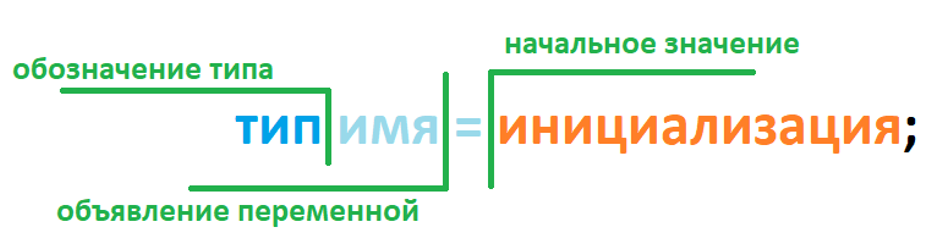
И перед нами появилось готовое консольное приложение. Жмем на зеленый треугольничек, я выделил его на скриншоте и наслаждаемся своим первым Hello World приложением на написанным с помощью языка C# на платформе .Net Core.



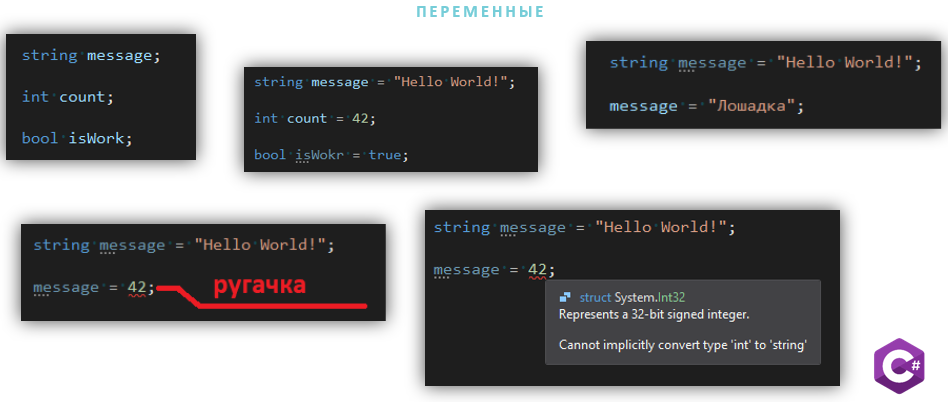
Вот вы и научились программировать на языке C# можем расходиться. Нет, все далеко не так просто. Несмотря на наличие такого прекрасного инструмента как Visual Studio. Нам всё ещё нужно самим разрабатывать програмное обеспечение и писать код. Вот об этом мы с вами и поговорм. Начнем с самго начала. С азов языка C#

БАЗОВЫЕ КОНЦЕПЦИИ

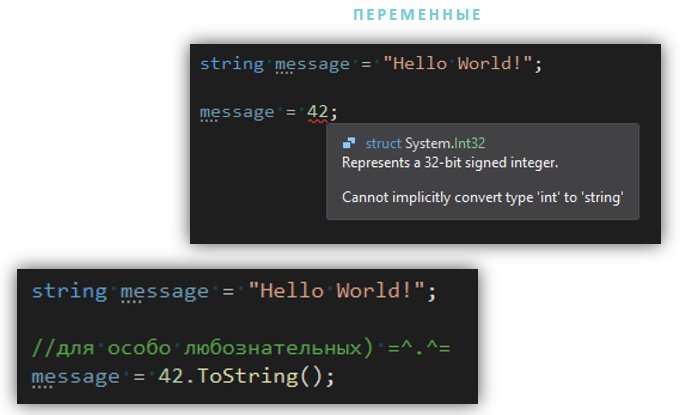
Одной из базовых конструкция языка C# является переменная – с этой концепцией вы, наверняка знакомы по С++. В C# переменные ведут себя похожим образом, но, в отличии от C++ вам не придеться вручную управлять памятью! За вас это делает сама платформа .Net, как это часто бывает в программировании – реальность несколько сложнее, но в рамках данной лекции это не важно, пока примем, что управление памятью автоматическое. Вот общая схема объявления и инициализации перменных в рамках C#



При объявлении перменных, не обязательно сразу их инициализировать, это можно сделать во время выполнения программы. Главное, что бы новое значение имело тот же тип, что и предыдущее. Так как C# - язык со статической типизацией: однажды назначенный тип переменной не меняется в ходе выполнения программы. Как видите Visual Studio предупреждает нас об ошибке типизации ещё до того, как мы бы попытались запустить программу. Что бы понять, что пошло не так достаточно навести мышку подчеркнутую красным область.



И тут вопрос: «а как превратить один тип в другой?» Для этого есть специальный механизм конвертации типов, о нем речь пойдет дальше в лекции, пока просто запомним, что значения одних типов можно правращить в значения других.

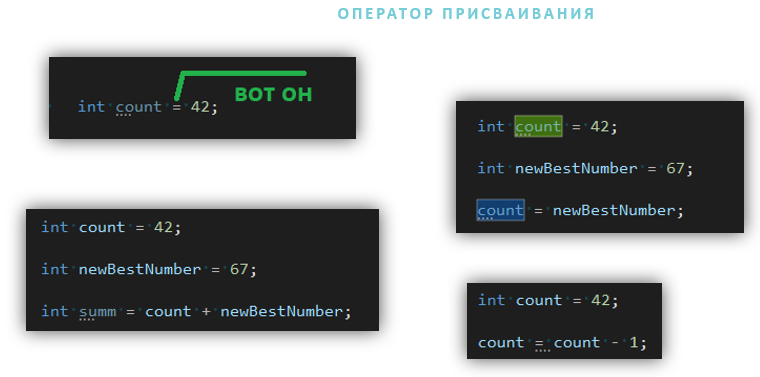




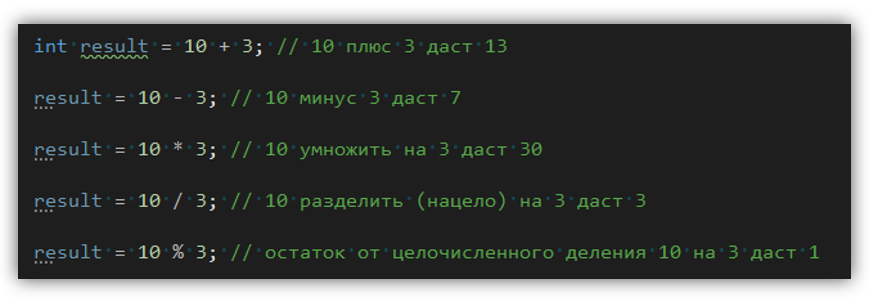
Так же ругачка появиться при попытке объявить в рамках зоны видимости (о зонах видимости я расскажу дальше в лекции, пока считаем что зона видимости – это вся наша программа) две переменные с одинаковым именем. То есть компилятор отличает переменные по имени. Но переменные только хранят данные, а мы хотим добиться каких-то действий с данными верно? Тут нам помогут операторы:

ОПЕРАТОРЫ

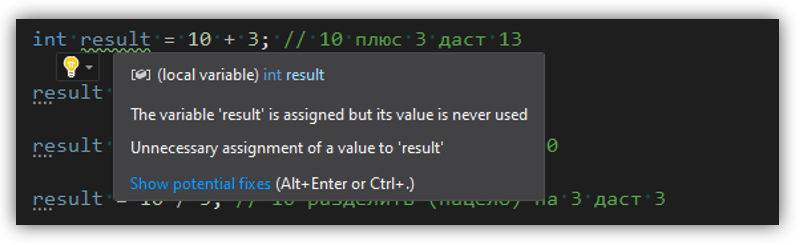
Для начала рассмотрим ОПЕРАТОР ПРИСВАИВАНИЯ он используется для передачи значения в перменную как и в C++. Дословно написанно: Переменной с именем count типа int ПРИСВОИТЬ значение 42. Так же мы можем присвоить одной переменной значение другой. Или мы можем присвоить сумму значений двух перемнных третьей. Или даже свое собственное значение которое мы предварительно модифицировали. Ничего нового относительно C++. А что насчет модификации данных?



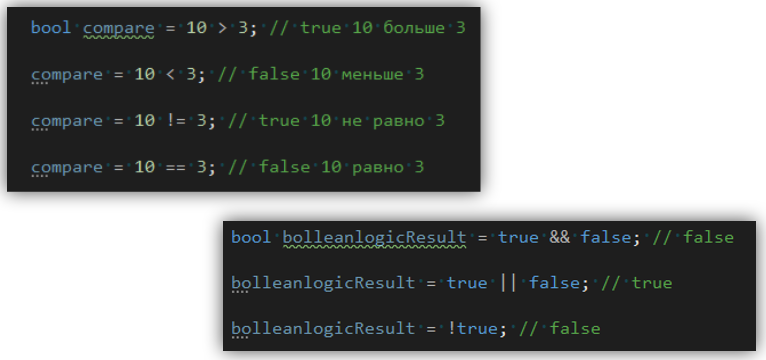
Начнем с математики, те самые плюс, минус, умножить, разделить (в нашем случае делние породило два оператора: целочисленное деление, которое просто отбрасывает остаток и, наоборот, оператор который возвращает остаток от деления). Кстати, Visual Studio нам подчеркнула переменную result зеленым, с чего бы это? Наведем и посмотрим:



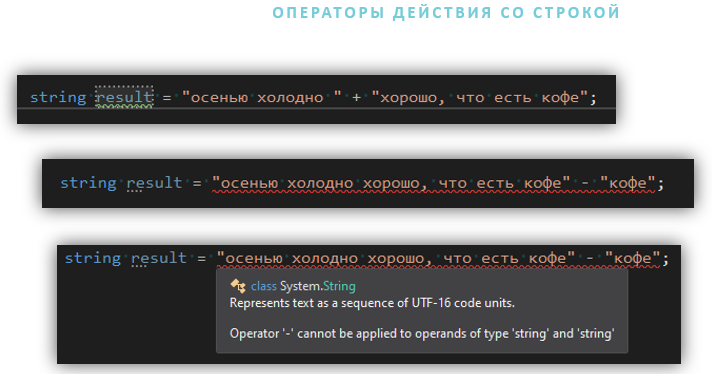
Ага, переменной result присвоенно значение, но оно никак нами не используется в дальнейшем. Это не ошибка, по этому и подчеркивание зеленое, это просто совет. Компилятор заботится о памяти нашего компьютера. Вот он и спрашивает: А вам точно нужен result? А то вы к нему совсем не обращаетесь на чтение, только на запись. Управление памятью – в рамках платформы .Net – это тема для отдельной большой лекции, не будем на этом останавливаться, на данный момент давайте просто запомним что под каждую переменную выделяется место в памяти в момент объявления, размер этой ячейки динамически меняется в процессе выполнения программы и память освобождается в тот момент, когда переменная в программе больше не используется.



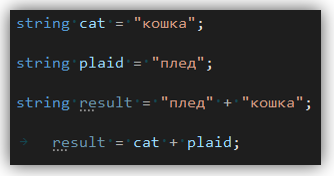
Помимо математических, у нас есть и логические операторы. Они позволяют конструировать условия. Операторы сравнения: больше, меньше равно, неравно. Обратите внимание, результатом выполнения у операторов сравнения будет значение булевого типа, которое мы присваивам переменной compare того же типа. А так же операторы булевой алгебры: И(&&) ИЛИ(||) НЕТ(!). Именно они отвечают за формирование комплексных условий, без которых невозможно представить современную программу.

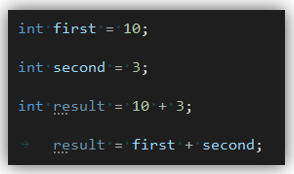


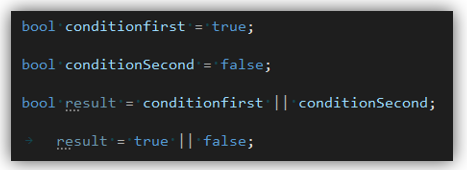
Теперь давайте рассмотрим необычный оператор, конкатинатор строк, казалось бы, он выглядит как обычный математический пльс, но, на самом деле, он представляет собой совсем другое действие: он ничего не складывает! Он создает третью строку, которая содержит в себе две предыдущих. А вот оператора минус для строк не существует. По этому компилятор думает, что мы имели ввиду математический минус и любезно нам говорит, что в математический минус он не может передать строки – не получается.



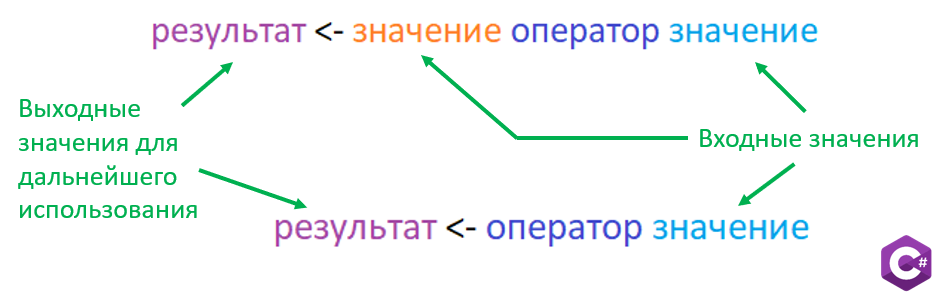
Наверняка вы и сами догадались, что операторы можно использовать не только с значениями, но и с переменными. При этом, не важно, строки это, числа или булевые значения. Обратите внимание, что результат выполнения математических операторов будет иметь числовой тип, лочических – булевый, строковых – строковый. Иными словами оператор всегда возвращает значения своего типа. И пусть вас не смущает тот факт, что операторы могут одинаково выглядить, внутри они совершенно разные и обладают разной типизацией.





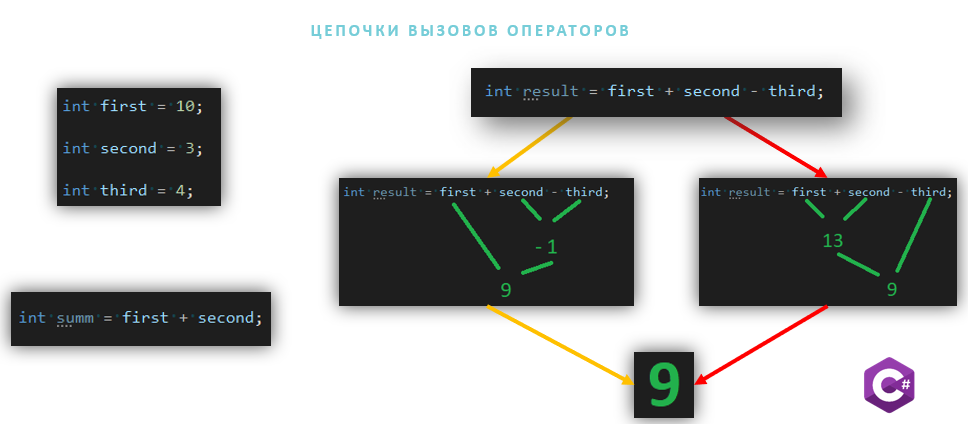


Досих пор мы рассматривали операторы, которые укладывались в две простых схемы: Бинарные и Унарные. А раз результат выполнения любого оператора – это значение, и на вход операторы принимают значения, значит мы можем соединить операторы в цепочку! На самом деле мы уже это делали, когда с помощью оператора присваивания инициализировали переменную результатом выполнения других операторов.



При инициализации переменных мы передаем 10 в оператор присваивания с помощью которого инициализируем переменную с именем first, затем мы точно так же инициализируем переменные second и third. А здесь мы передаем переменные first и second в математический оператор сложения и, полученным результатом, инициализируем переменную summ. Давайте разберем цепочку подлиннее:

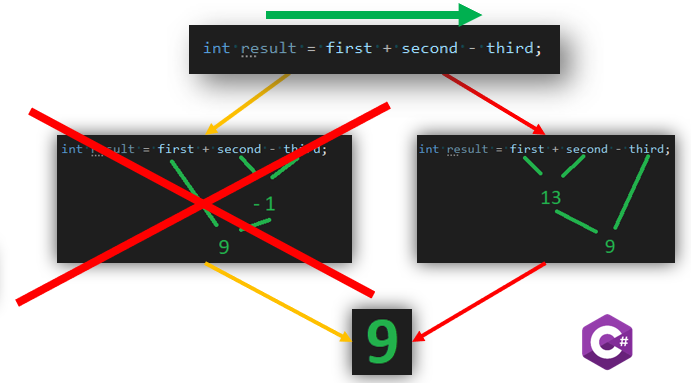
Возьмем три проинициализиарованных перменных, и построим из них вот такую цепочку вызовов.



Что же случиться вначале:

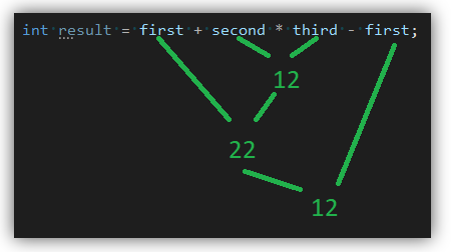
* Из 3 вычтут 4 а затем прибавят к 10ти или наоборот?
* К 10ти прибавят 3, а потом вычтут из суммы 4ре?

Да, результат в обоих случаях будет одинаковым, но вычисления-то шли разными путями. В ходе выполнения программы разница может быть очень существенной. Выполнение пойдет по второму сценарию:

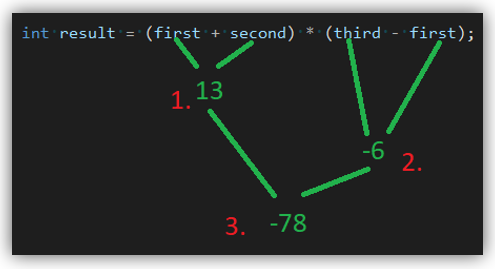


**При равном приоритете операторов, вычисления пойдут слева на право.**

Это не всегда так, и пержде чем делать такие выводы стоит свериться со справочной информацией, но большинство бинарных операторов ведут себя так. А оператор присваивания, в любом случае выполнится последним. Из этого правила мы делаем вывод о том что у оператров есть приоритезация выполнения. Подробнее за приоритетами опраторов стоит заглянуть в справочник на MSDN ссылки будут в презентации. А пока запомним, что деление и умножении, как и в обычной арифметике имеет приоритет над вычитанием и сложеним.

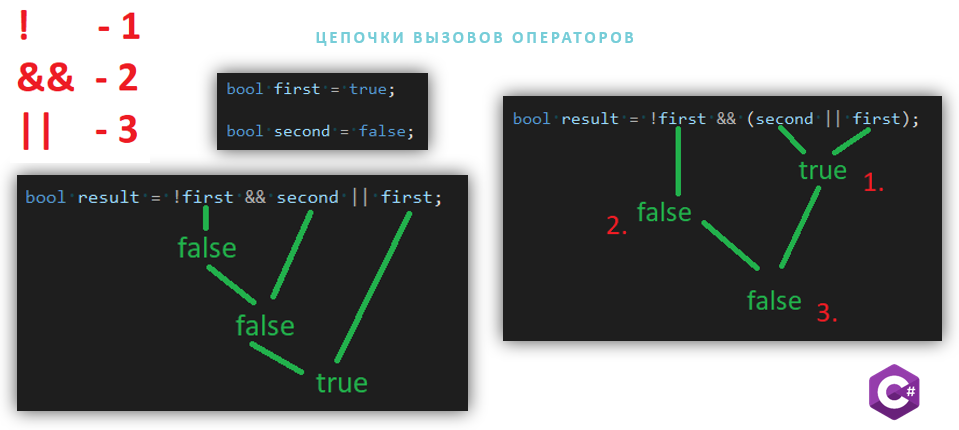


То есть: сначала 3 будет умножено на 4 затем полученые 12ть буду сложены с 10кой а потом из полученных 22х будет вычтена 10тка. И только так и не иначе. Если же мы хотим изменить приоритет выполнения операторов, то, как и в классической арифметике, мы можем воспользоваться скобками:



Мы добавили скобки, повысив тем самым приоритет операций сложения и вычитания по сравнению с операцией умножения. Слева направо: 10тка складывается с 3йкой, получаем 13ть. Затем и 4рки вычитается 10тка, получаем -6ть, и только потом перемножаем -6 на 13 ть и получаем -78

Логические операторы так же имеют свой порядок выполнения. Он представлен в левом верхнем углу. Сначала выполнится оператор отрицания, затем И, а в конце оператор ИЛИ. Скобоки меняют порядок выполнения логических операторов так же, как и математических. Сначала выполнится оператор ИЛИ затем отрицание, а в конце оператор И.

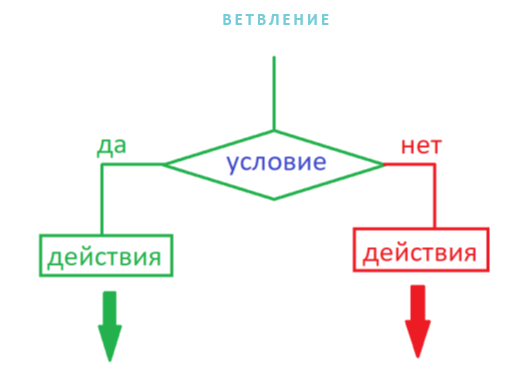


Вот ссылки для подробного ознакомления с великим многообразием операторов языка C#

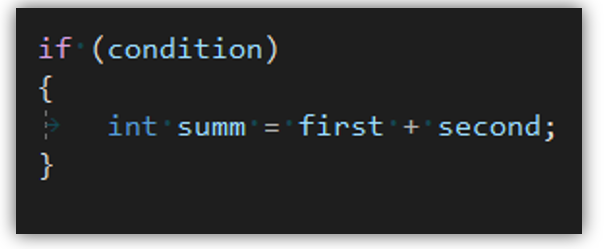
* [Оператор присваивания](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/assignment-operator)
* [Арифметические операторы](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/arithmetic-operators)
* [Операторы сравнения](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/comparison-operators) [и равенства](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/equality-operators)
* [Логические операторы](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/boolean-logical-operators)
* [Общие сведения](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/)

Теперь перейдем к более комплексным конструкциям, как мы помним из вступления, любая программа состоит из трех действий: пошаговое следование инструкциям, с этим мы разобрались выше когда знакомились с операторами, ветвлений и циклов.

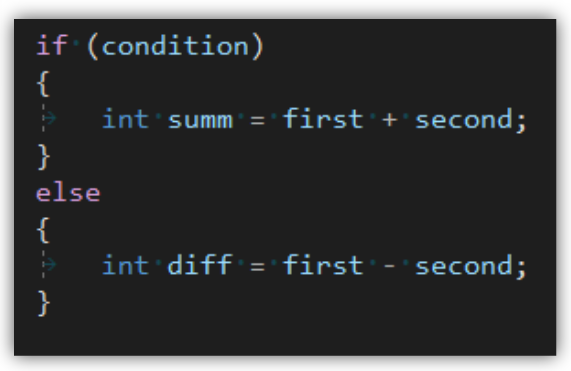
Начнем с ветвлений, В самом примитивном случае ветвление можно представить вот такой схемой, в которой условие – это некоторое выражение булевой алгебры (штука которая вернет true или false), в соответствии с которым изменится ход выполнения программы, на ДА (true) выполнится зеленая веточка, на НЕТ (false) красненькая.

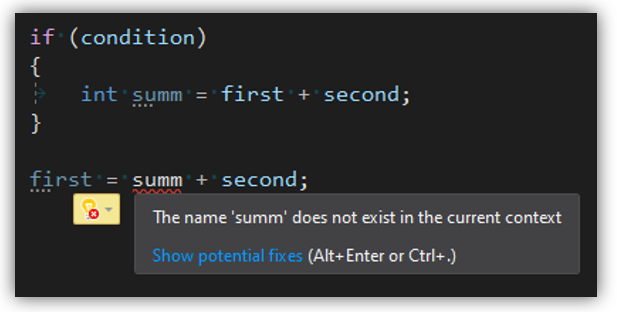


Вот простейший вариант ветвления в C# в случае если переменная condition возвращает true мы выполняем действие в скобках, иначе этот код будет проигнорирован.

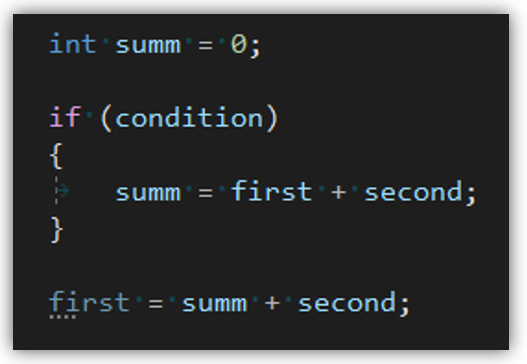


If конструкция – так же является оператором, как и математическое сложение. Можно в явном виде указать и вторую ветку условия, которая выполняется в случае если condition у нас вернет false.

Вы можете спросить: а зачем? И так же понятно, что если не выполнилось условие – то код в скобках будет просто напросто проигнорирован, и выполненние пойдет дальше. Для прояснения ситуации нам необходимо ввести новое понятие: контекст выполнения – это последовательность команд выполняемых одна за другой. Посмотрим что будет при попытке обратиться к summ за пределами фигурных скобок, компилятор говорит нам: я не знаю такой переменной!

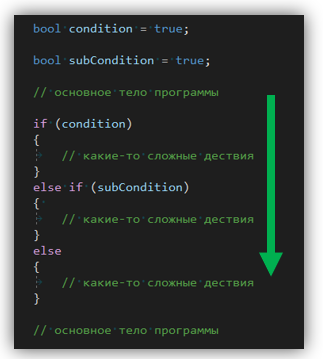


То есть фигурные скобки создали независимый дочерний контекст - отдельный, новый список команд для выполнения в рамках нашей программы, в котором доступны переменные из родительского контекста, а вот объявленные внутри него перменные доступны только в нем! Как же нам получить данные о сложении вне фигурных скобок? Мы видим, что перменная из родительского контекста доступна в дочернем. Так давайте же объявим перменную summ до фигурных скобок.



Как видим проблема исчезла теперь значение summ будет по разному влиять на дальнейшее исполнение программы в зависимости от того. Вполнился блок кода в фигурных скобках или нет.

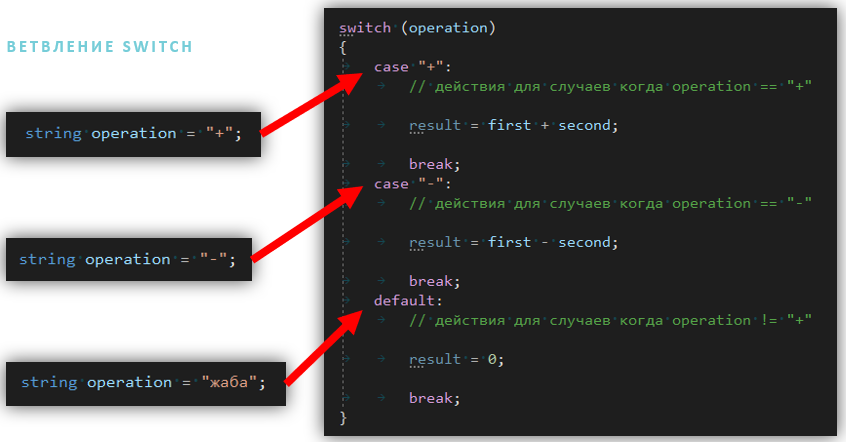
И вот тут мы подходим к пониманию того, зачем нам else – он позволяет изолировать код в рамках дочернего контекста вызова. В примере с явным указанием else код относящийся к выполнению условия condition изолирован от кода который буде выполнен в случае если condition == false. Как мы видим объявленные в рамках дочерних контекстов переменные остануться там и никак не повлияют на основное тело программы, тем самым мы создаем логические блоки не зависящие друг от друга.



Так же мы вправе состыковывать if else операторы для проверки дополнительных условий, при этом приоритет проверки будет идти сверху вниз. То есть условие subCondition будет проверенно только если condition вернет false. Естественно ветвление может иметь сколь угодно вложенную структуру.

**Не забываем, что компьютер может понять куда более запутанную логику чем программист. Программа должна быть понятна в первую очередь программисту, а уже потом компьютеру.**

Но что если мы хотим оперировать в ветвлении непосредственно значеним переменной? И выполнять действия в зависимости от её значений? Можно конечно свести это к простому булевому условию и сделать конструкцию из состыкованных операторов if но в C# есть специальный оператор для такого рода ветвлений:



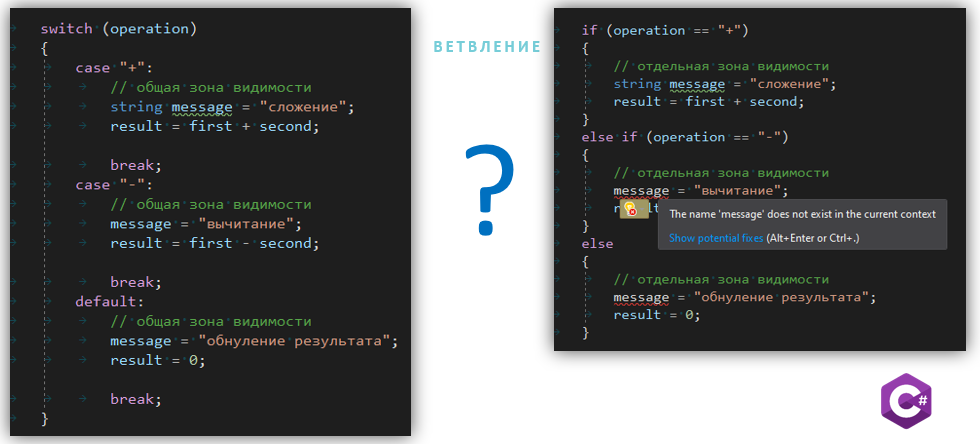
switch – ветвление в зависимости от значения перменной. Внутри себя он имеет множество кейсов, каждый из которых привязан к тому или иному значению переменной указанной в скобках. Работает это так:

* Для случаев когда operation возвращает строку плюс будет выполнеено вот это действие,
* Для случаев, когда operation возвращает строку минус – вот это действие.
* А если в operation вернет что-то непонятное, то вот это действие,

После чего выполнение ветвления завершиться, и прогамма продолжит выполняться в основном контексте.

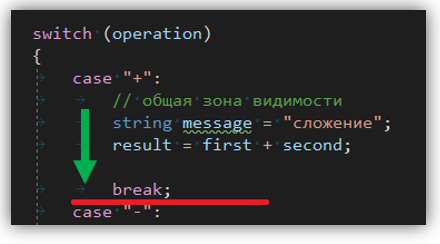
Чем оператор switch отличается от конструкции из состыкованных операторов if else? Тем, что switch создаст только один дочений контекст! В отличии от пристыкованных if ов! То есть, Мы объявляем переменную message в первом кейсе, и без проблем используем ей на протяжении всего оператора switch! В то время как в состыкованных if мы так сделать не сможем. А так же тем, что в switch мы можем использовать только одно условие, в то время как пристыкованные if могут оперировать совершенно разными условиями никак не зависящими друг от друга.

Пример на следующей странице:

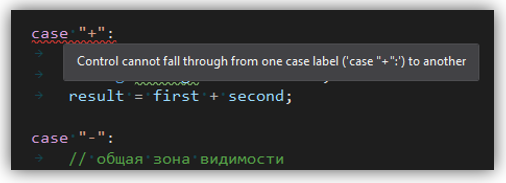


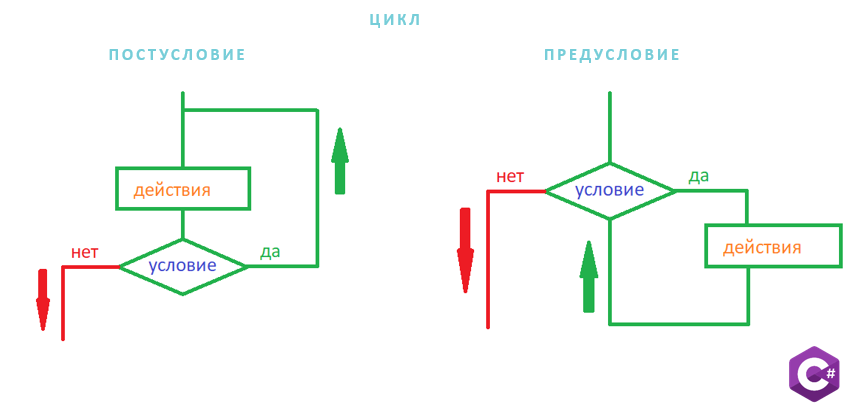
Хорошо, раз зона видимости одна на весь оператор, как же switch отделяет один кейс от другого?

Дело в том, что каждый кейс выполняется до ключевого слова break после чего выполненние кода внутри оператора switch прекращается.



Для исключения всякеских недоразумений Visual Studio предупреждает нас о том, что не может отличить один кейс от другого. Без ключег break

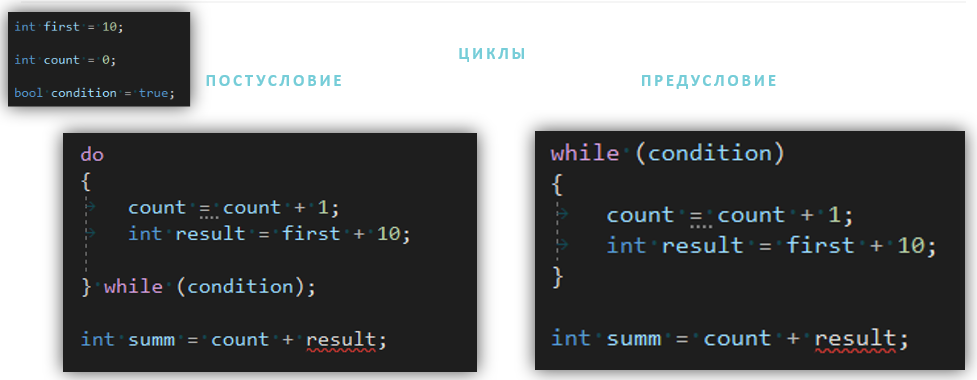




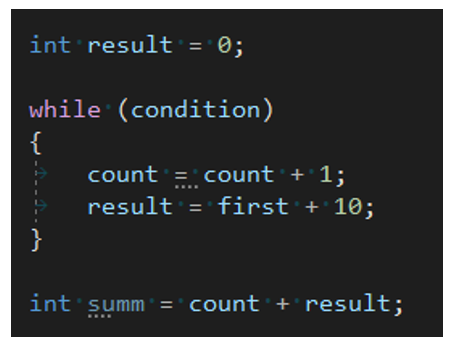
Цикл – это частный случай ветвления: **ветвление назад**, программа при выполнении какого либо условия возвращается к уже выполненному участку кода. Здесь представленны две схемы организации циклов:

* цикл с постусловием когда действие выполняется в любом случае, а затем условие определяет будет-ли оно повторяться
* цикл с предусловием, когда действие может и не выполниться ниразу, если условие вернуло нам false

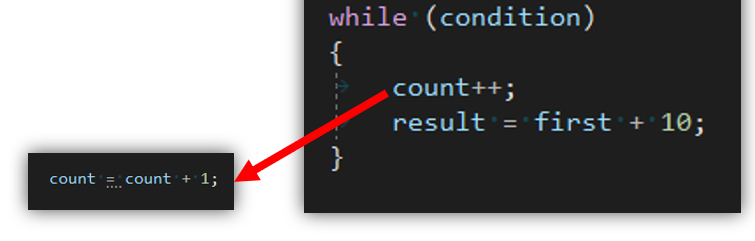
Как вы понимаете в C# есть конструкции для обоих случаев.



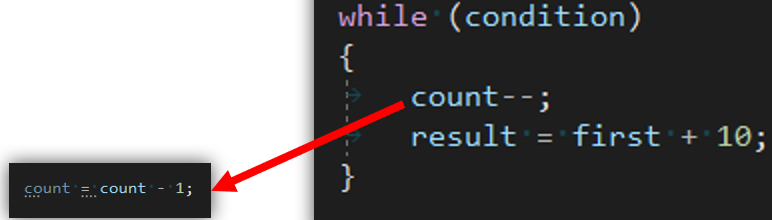
Вот так эти циклы реализованны в рамках C#. Обратите внимание, циклы, как и ветвления образуют дочерний контекст, при этом объявленная в рамках дочернего контекста переменная result не видна из основного контекста приложения в **обоих** случаях, хотя в случае цикла с постусловием действия внутри фигурных скобок будут выполненны вне зависимости от значения переменной condition. Циклические операторы (а циклы – это тоже операторы как и ветвление) создают дочерний контекст в не зависимости от того, будут они выполняться или нет. Объявим result и починим код.



Обратите внимание на переменную **count**. Внути циклов мы прибавляем 1 к её предудущему значению и результатом этой суммы снова инициализируем count. В программировании это называется инкрементирование, в C# для этого есть специальный оператор:

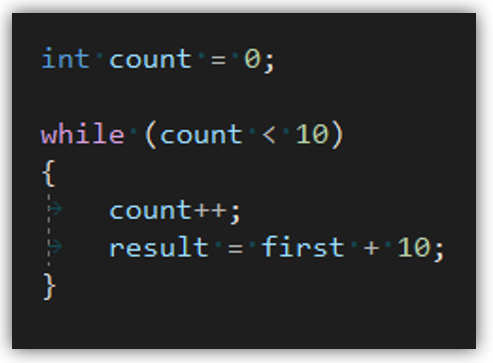


**Инкремент** – это унарный оператор, то есть принимает одно значение на вход. Он увеличивает переданое в него значение на единицу



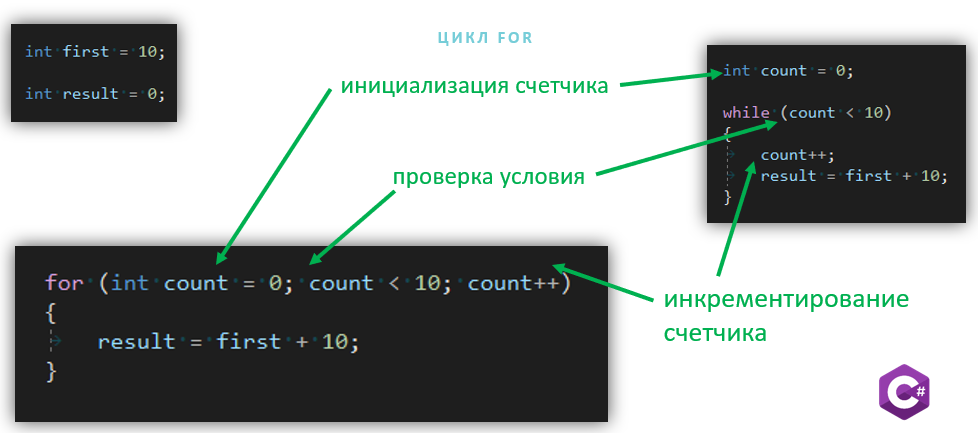
Для удобства есть и обратный опреатор: **Декремент**, который уменьшает переданое в него значение на единицу.

Давайте обратим внимание на условие повторения цикла, мы передавали в него пременную condition, заменим её на условие зависящие от значения **count.**

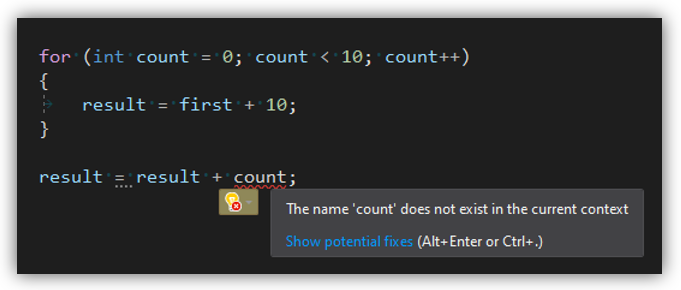


Цикл будет выполняться до тех пор, пока count меньше 10ти при этом каждую итерацию count будет прирастать на 1 изначально count проинициализирован 0 значит цикл будет выполняться 10-ть раз.

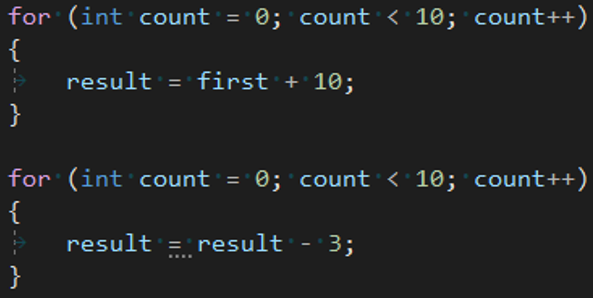
Циклы, которые надо выполнить определенное колличество раз – очень часто встречаются в программировании, по этому в C# для них есть специальная конструкция



Он во всем аналогичен вот такому циклу while кроме одного маааленького но: В конструкции while переменная count находится во внешнем контесте относительно цикла и может применяться за его рамками. В конструкции for переменная count объявляется во внутреннем контексте.



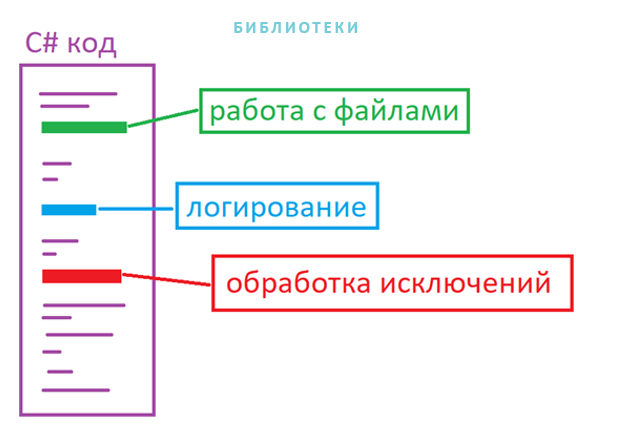
Что позволяет нам писать вот так:



Без каких либо побочных эффектов. Цикл For объявляет и инициализирует счетчик во внутреннем контексте. Тем самым обеспечивая изоляцию счетчика от основного контекста выполнения программы.

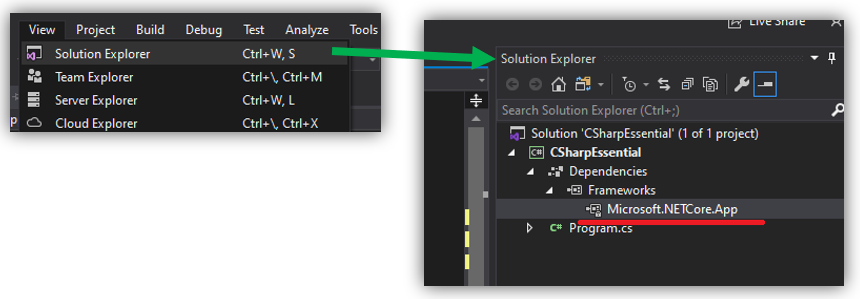


Вот такие конструкции применяются в C# для реализации циклов. Мы почти готовы писать нашу первую программу, мы уже понимаем порядок выполнения команд, умеем создавать ветвления и реализовывать циклы. Нам знакомы булевые и математические операции, мы даже умеем соединять строки! Остался последний штрих: как бы нам интегрироваться с вводом выводом? Что бы наша программа умела принимать пользовательские данные и отдавать результат? И тут на помощь нам приходят библиотеки.

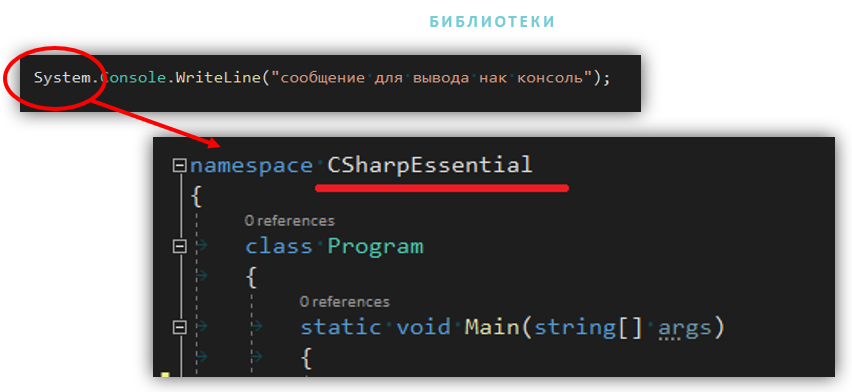


C# умеет взавимодействовать со множеством готовых библиотек концептуально они мало отличаются от библиотек C++. Они так же предоставляют нам набор компонентов, которые мы можем вызвать из написанного нами кода, передать туда необходимые параметры и получить ожидаемый результат. Некоторые библиотеки поставляются непоредственно в комплекте платформы .Net и могут использоваться сразу, другие мы можем скачать в виде nuget пакетов, так же мы можем написать библиотеку самостоятельно.

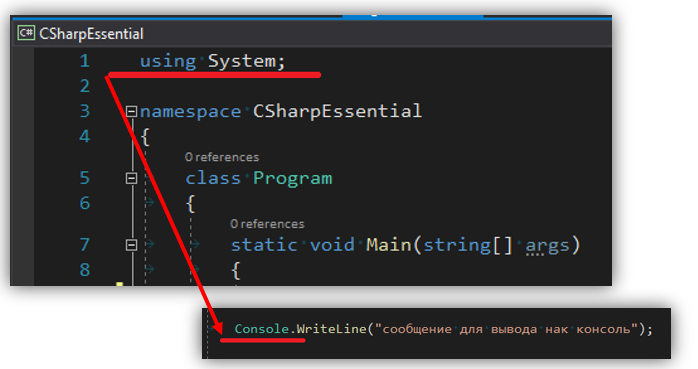
Давайте откроем ранее созданное приложение и добавитм в него вывод на экран сообщения. Для этого мы будем работать со стандартной библиотекой обеспечивающей нам доступ к консоле ввода\вывода. Она входит в стандартный пакет который подключается к нашему проекту в момент созданния, давайте посмотрим где это можно увидеть. В меню View верхней панели visual studio выбираем окно solution explorer. Именно там надо искать его, если случайно закрыли. Он открывается в с левой стороны: находим там наше приложение и раскрываем вкладку зависимости.



В этой вкладке живут все ссылки на подключаемые библиотеки, не важно, стандартные они, скаченные из интернета или написанные нами самими. Внутри мы видим только один пакет SDK содержащий стандартную для .Net Core библиотеку. Которая, в свою очередь, включает в себя огромный пакет компонетнов. Нужный нам компонент находится в пространстве имен System в классе Console и называется WriteLine так и пишем: System.Console.WriteLine(“сообщение”) здесь System – это пространство имен в котором находится статический класс Console у которого мы вызываем метод WriteLine.

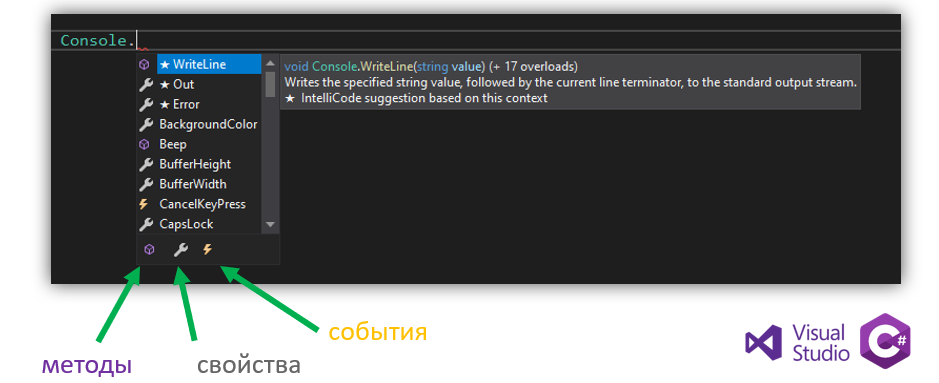


В нашем приложении тоже есть пространство имен, вот оно. Пространства имен нужны для того, что бы при построении дерева зависимостей которые мы видели на слайде ранее компилятор мог однозначно классифицировать любой объект. В рамках нашего приложения мы можем создать ссылку на пространство имен System, что бы не писать его каждый раз при обращении к классу Console. И теперь мы можем написать вот так:



Хорошо, мы научились обращаться к компонентам встроенных библиотек, но остается вопрос, откуда я узнал про метод WriteLine? И о том, что ему нужно передавать?

Я просто написал Console и поставил точку, а затем, Visual Studio выдала мне вот эту прекрасную подсказку в которой содержался полный список всех методов, свойств и событий доступных для обращения у класса консоль. Более того, выбрав соответствующий компонент я вижу его описание, список параметров и число перегрузок!



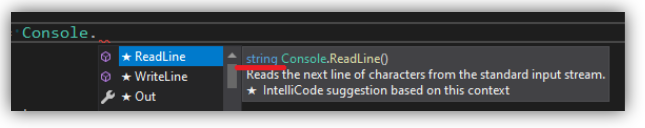
Справочная информация за которой не надо лезть в глубины MSDN. С помощью WriteLine мы можем передать значение переменной в стандартный выходной поток (это видно в описании), который будет выведен на экран в консоле.

Механизм устроен несколько сложнее, но нам это не важно, мы избавленны от необходимости собственноручно связыватся с консолью windows или linux для обработки выходного потока нашей программы. На примере работы данного механизма мы можем наблюдать АБСТРАКЦИЮ в действии: нам нет нужды знать как именно работает Console.WriteLine нам достаточно знать как им оперирвать в рамках написанной нами программы.

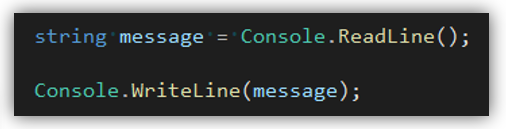
С выводом данных мы разобрались, а как бы нам получить данные из консоли? С помощью другого метода у объекта класса Console естественно!



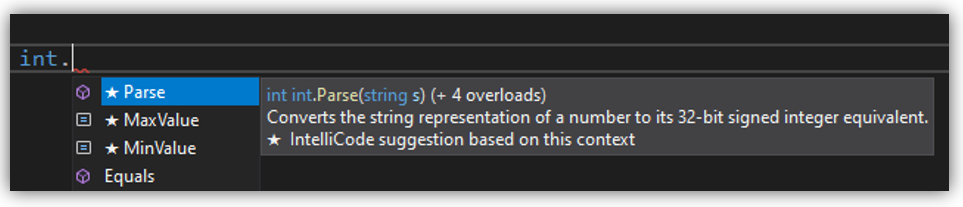
Console.ReadLine() Судя по описанию он имеет доступ к стандатному входному потоку, а значит на пару с Console.WriteLine они обеспечат нам связь с внешним миром. Обратите внимание вот сюда, у метода Console.ReadLine() перед именем указан тип string как мы помним, это тип строковых переменных. А у метода Console.WriteLine тип void – это означает, что данный метод не возвращает значений. Так же, обратите внимание, что метод Console.WriteLine принимает значение, в отличии от метода Console.ReadLine



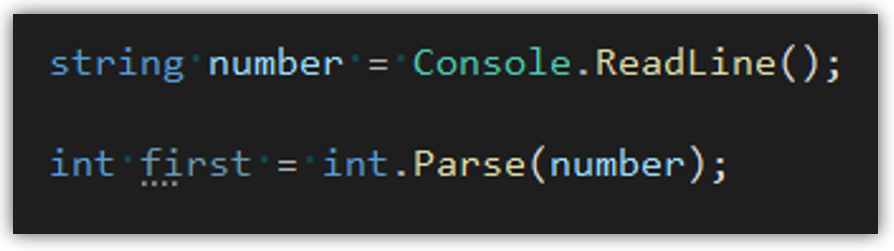
Таким образом мы можем проинициализировать переменную message значением которое вернет Console.ReadLine() и передать её в метод Console.WriteLine



Теперь мы знаем как передать строку для вывода на консоль и как получить строку из консоли, а как нам получить число введенное пользователем? C# язык строго типизированный, по этому строка 2 не равна числу 2 у них разные типы, а считать из входного потока мы можем только строку. Как мы помоним по опыту обращения с объектом класса консоль, у объектов C# есть множество различных методов двайте посмотрим. Что нам может предложить числовой тип, для решения нашей задачи.



Вот он, нужный нам метод int.Parse служит для конвертации строк в числовые объекты, по его описанию мы видим, что он принимает строку, а возвращает число. Мы можем собрать следующую конструкцию: получили строку из консоли, и конвертировали её в число.

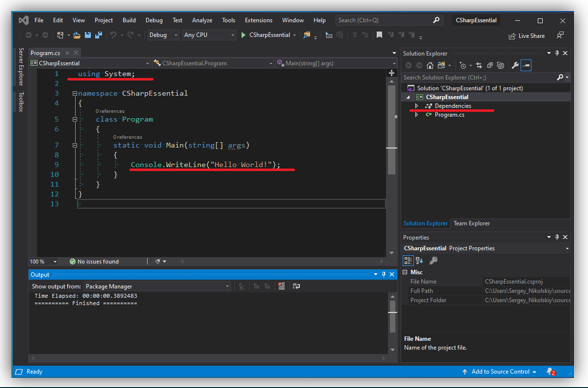


И все будет в порядке для строк 2, 35 или 678, но что же случится если строка не будет конвертироваться в число? Об этом дальше, пока же просто запомним данный метод конвертации.

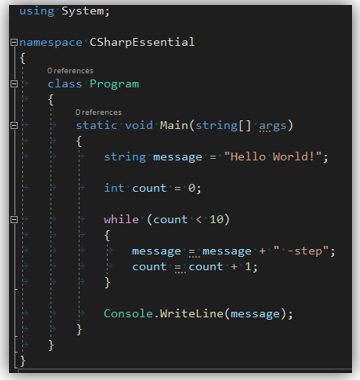
Мы с вами прошли непростой путь, от самых общих понятий о программировании мы перешли к конкретному языку и его возможностям. Вот ссылки на MSDN где вы сможете подробнее познакомиться со спецификацией C#.

* [Введение в C# и платформу .NET](https://docs.microsoft.com/ru-ru/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework)
* [Обзор C#](https://docs.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/)
* [Руководство по программированию на C#](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/)
* [Операторы с ключевыми словами](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/statement-keywords)

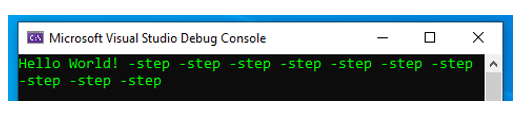
Давайте посмотрим как выглядит реальная программа на этом языке. Помните нашу первую программу? Тот самый Hello World который за нас сгенерировала Visual Studio? Давайте присмотримся к ней повнимательнее. Вот список зависимостей, вот объявленно использование пространства имен System, а вот обращение к объекту класса Console и вызов метода WriteLine в который переданна строка привет мир.



Давайте вынесем эту строку в отдельную переменную. Добавим счетик и цикл.



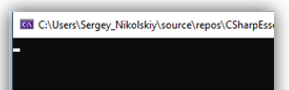
Запустим на выполненние:



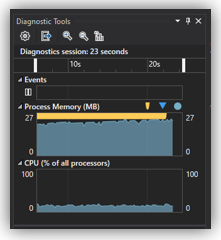
Как мы видим, всё отлично работает. Условие count меньше 10 успешно выводит программу из цикла, а что если присвоить это условие перменной? Давайте посмотри.



Мы объявляем перменную condition, инициализируем её результатом выполнения оператора сравнения и передаем в цикл. Запускаем.



И происходит вот что, на экране диагностики видно, что программа работает: память выделяется, процессор исправно выполняет иструкции, но консоль пустая. Наверное что-то пошло нетак.

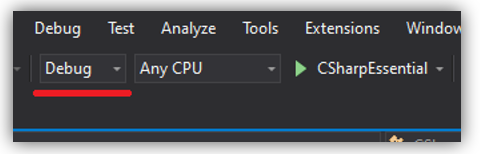


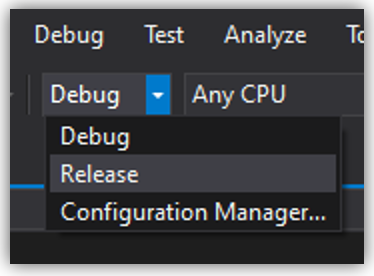
Как бы нам узнать что именно? Судя по тому, что компилятор не ругается на нашу программу – ошибка не в синтаксисе, все конструкции мы использовали верно. Значит дело в самой логике, как бы нам посмотреть что там на самом деле происходит во время выполнения программы? И тут нам на помощь приходят точки остановки. Обратите внимание вот сюда, левее нумерации строк, это панель для точек остановки. Достаточно кликнуть мышкой на этой панели напротив той строчки кода, в которой мы можем разместить точку остановки и она появится. Поставим их на линии 13 где мы объявили и проинициализировали переменную condition, вдруг с ней что нетак? И внутри цикла на строке 17-ть.



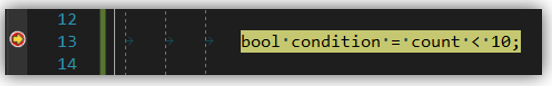
Обратите внимание, помимо красной точки, Visual Studio выделила ту область кода, в которую мы попадем при остановке выполнения программы. А непосредственно в момент простановки, мы видим сообщение о том, где именно в программе находится проставляемая точка и получаем доступ к её настройке. О настройке точек речь пойдет в следующих лекциях. Пержде чем запустить программу, пару слов о режимах запуска.

На верхней панели. Рядом с зеленым треугольником через который мы запускаем приложение, есть перечисление возможных режимов запуска, давайте откроим его и посмотри что там внутри. Дебаг, релиз и менеджер конфигураций. Здесь мы можем выбрать тип сборки, который будет создан и запущен в момент нажатия зеленого треугольника.

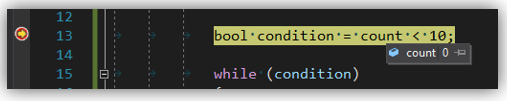




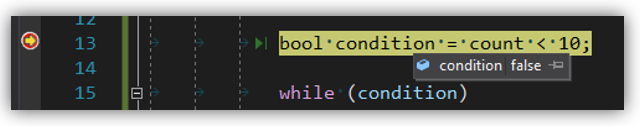
Подробнее о сборках поговорим в далее. Пока просто запомним, что в дебаг сборке точки останова сработают, а в релизной нет. Выбираем дебаг и запускаем программу в отладочной сборке. Срабатывает первая точка останова.



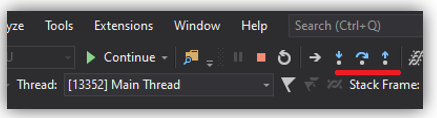
Наводим мышку на переменную count и видим её значение: 0.



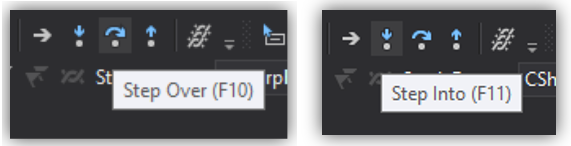
0 меньше 10ти значит переменная condition должна быть проинициализированна значением true. Наведем на неё мышь и узнаем значение.



А вот и нет! condition выдает false по крайней мере на данный момент, в чем же дело? Ноль же явно меньше 10ти? Дело в том, что точка останова срабатывает ДО того как отработал оператор присваивания, а поумолчанию значение для всех перменных булевого типа false. Мы остановились до того, как condition получит свое значение. Что ж, давайте двинимся на шаг вперед, и посмотрим. Для этого снова глянем на верхнюю панель Visual Studio:

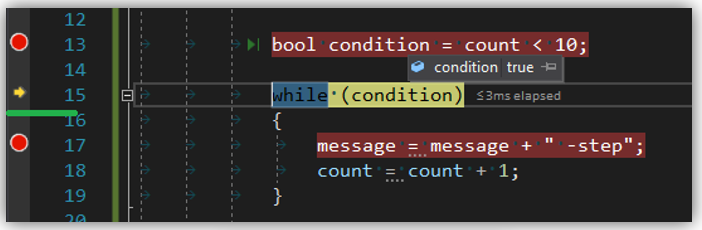


Зеленый треугольник остался на месте, но вместо имени приложения там теперь слово продолжить, если мы нажмем на треугольник теперь, в запущенном приложении, то мы продолжим выполнение программы до следующей точки останова. Нам это пока не нужно. Давайте обратим внимание на значки рядом:

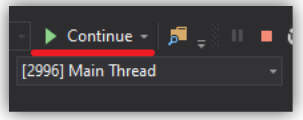


* F11 и мы попадаем внутрь исполняемго метода, пока нам это не интересно.
* F10 и мы переходим дальше по коду - наш случай.

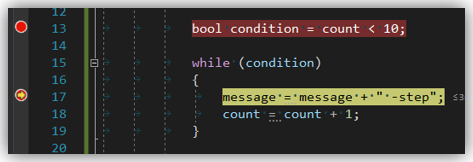
Там есть и другие опции но в рамках нашей задачи они не нужны. Жмем F10 и переходим к следующей строчке. Наводим мышь на condition и видим, что она приняла нужное нам значение.

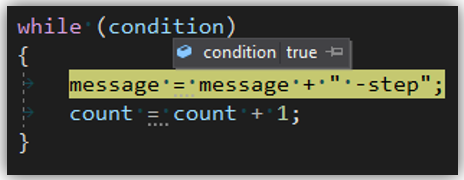


Обратите внимание на желтую стрелочку сбоку. Она показывает ту строчку ПЕРЕД выполнением которой мы остановились. Нажмем продолжить, для перехода к следующей точке остановки, той самой, которая нахдится в цикле.



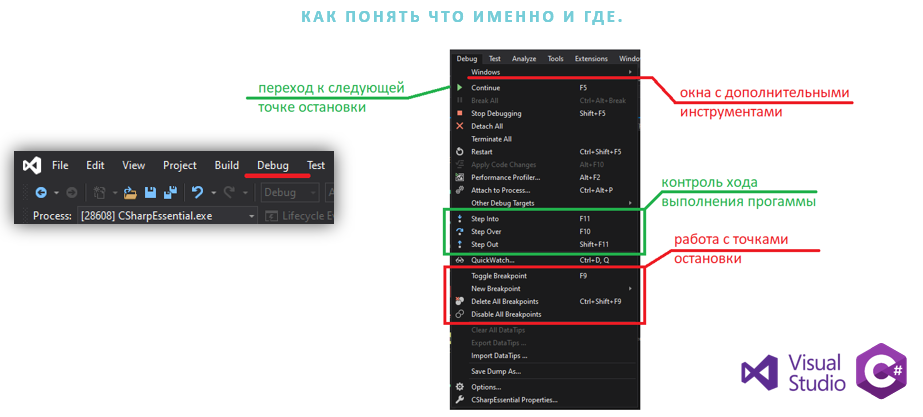
Вот мы и вошли в цикл, нажав на продолжить ещё несколько раз мы убеждаемся в том, что переменная condition не меняет свое значение.



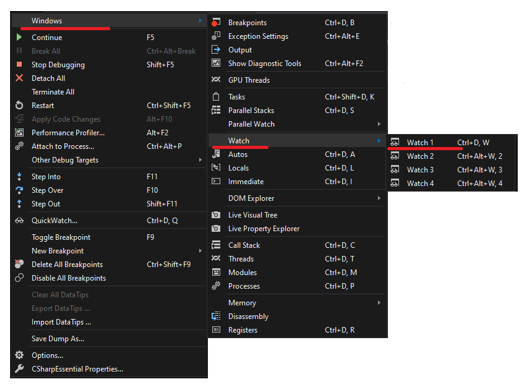


И тут возникает вопрос, а где бы нам взять инструмент для отслеживания значений всех интересующих нас переменных? И он есть в Visual Studio!

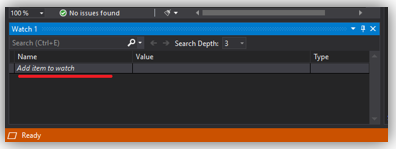
В верхней панели находим вкладку Debug, разорачиваем, и мы видим уже знакомые нам иконки: зеленый треугольник продолжить, стоп для остановки выполнения программы. F11 F10 для контроля хода выполнения программы, выключение\удаление всех точек останова и, в самом верху, окошки с дополнительными инструментами отладки.



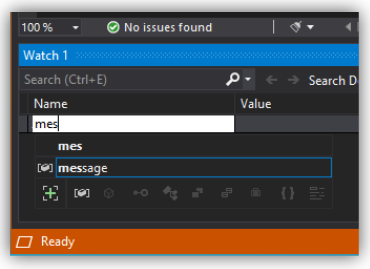
Во всем перечисленном многообразии нас интересует вот это окошко, набдлюдатель. Как видно их можно открыть 4ре штку, откроем первый.



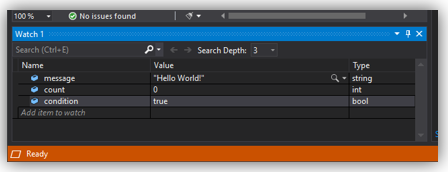
Внизу экрана появляется вот такое окошко.



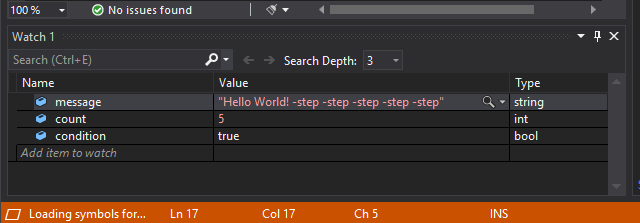
для добавления переменной к просмотру просто кликаем в поле для ввода имени. Как только мы начинаем вводить имя, Visual Studio изо всех сил начинает нам помогать, для того, что бы выбрать понравившуюся вам перменную, воспользуйтесь клавишами курсора.



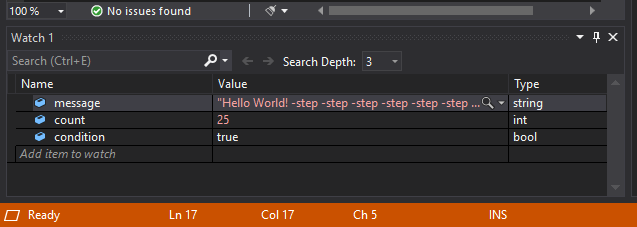
Добавим все три перменных в окно наблюдателя. Вот что мы имеем на начальном этапе.



А вот какие значения будут после 5той итерации цикла

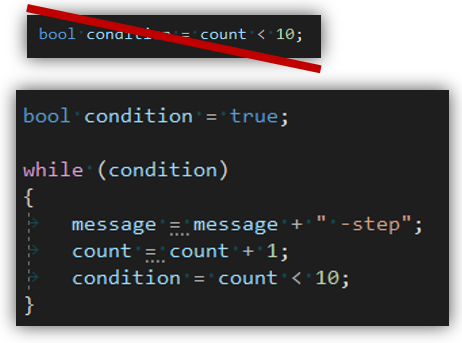


На 25-той.

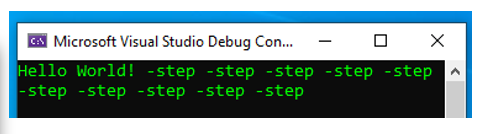


Как вы видите condition всё ещё равен true, программа бегает в бесконченом цикле. Почему так получилось? Вспоминаем как мы инициализировали переменную condition:

Мы присвоили ему результат выполнения оператора сравнения в который передали значение перменной count и сравнили его с 10тью. Проблема тут кроется в слове ЗНАЧЕНИЕ, при инициализации программа считала значение count, оно БЫЛО меньше 10ти в момент инициализации, и, соответственно, condition был проинициализирован значением true А то, что потом значение count менялось – для перменной condition осталось за кадром.

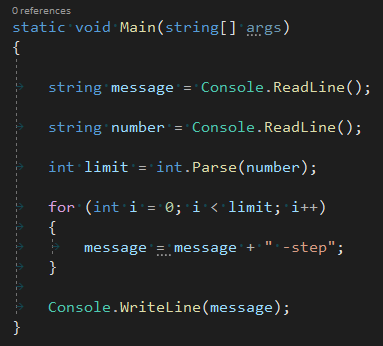


Вот мы и обнаружили проблему: вместо того, что бы динамически менять значение condition каждую итерацию цикла, мы задали его один раз. Вносим изменения в код и запускаем программу. Проблема решена!

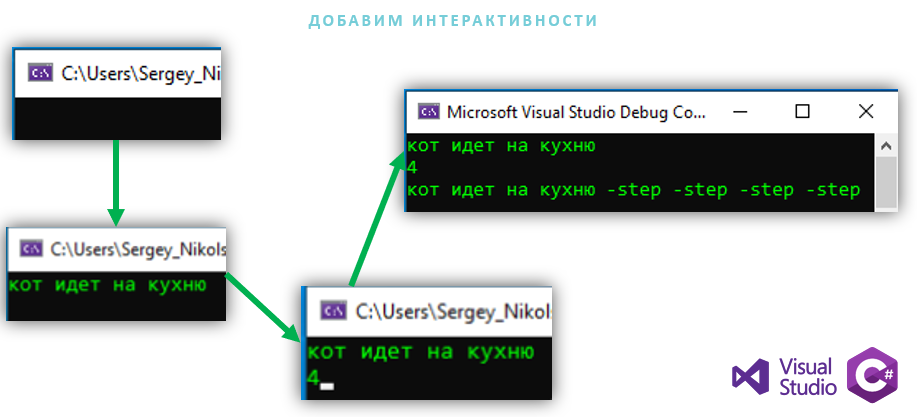


Таким образом, мы познакомились со способами отладки. Как видите в коде демонстрационного приложения мы использовали очень простые конструкции.

Но с тем инструментарием который вы получили из лекции, мы можем реализовать и что-то посложнее. Помните замечательные методы объетков Console и int? Пришло время встроить их в нашу программу, заменим так же while на for так как, в нашем случае ему тут самое место, мы же прерываем цикл по достижении счетчиком определенного значения, а for буквально создан для таких случаев. Результат на следующей странице:

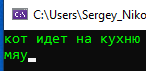


Вот что у нас получилось.

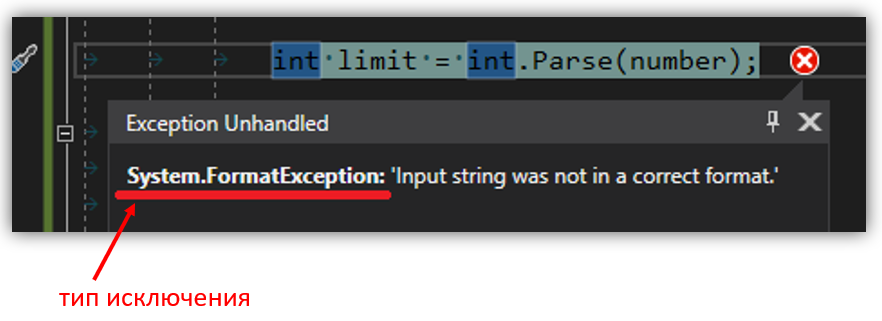


Запускаем, консоль пуста и ждет ввода сообщения, ведем его, нажмем enter, в этот момент метод Console.ReadLine считал входной поток из консоли windows и проинициализировал полученным сообщением переменную message, запустилось выполнение второго вызова метода Console.ReadLine который снова ждет от нас информацию: передаем 4рку и нажимаем enter программа получила все необходимые данные и выполняется до конца. Но это в том случае если мы вводим правильные данные.

А что случится если мы сделам так? Мяу enter!



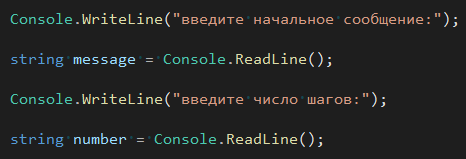
Все будет хорошо ровно де тех пор, пока мы не попытаемся распарсить (задействовать метод Parse – это стнадартное название для подобных конвертаций. По этому программисты говорят «распарсить») мяу в число, у нас ничего не получилось и метод int.Parse выдает нам ошибку:



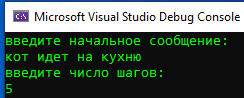
Обратите внимание, что ошибка ввода данных, в нашем случае – это вполне конкретный объект, который имеет свой тип и назвается исключением, это не ошибка в логике программы, как в предыдущем случае когда мы попали в бесконечный цикл, это ошибка в обработке данных. Обратите внимание: исключение, которое привело к остановке выполнения программы является специальным объектом, который, в свою очередь содержит информацию об ошибке.

Если мы введем те данные которые от нас ожидает программа – всё будет впорядке, то есть наша программа стала ХРУПКОЙ, она неявно для пользователя зависит от того, какие данные в неё передают.

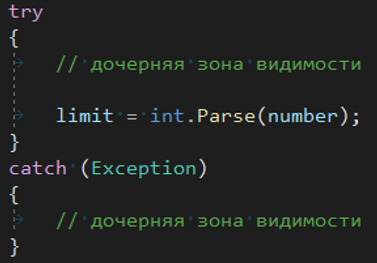
Давайте же снизим хрупкость программы путем добавления пояснений.



Шансов запутаться стало меньше, но ввод неправильных данных всё ещё возможен.



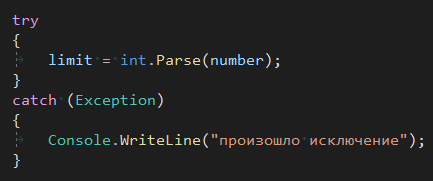
Для дальнейшего снижения хрупкости программы мы попробуем обработать объект исключения, что бы он не ломал логику. А органично в неё вписывался. Этой цели служит конструкция try catch давайте я её вам покажу.



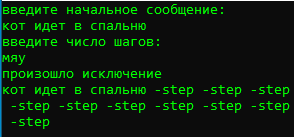
Как и в случае операторов ветвления и циклов образует дочерний контекст вызова, вы уже догадались что каждый раз, когда мы видим фигурные скобки в операторе – у нас образуется дочерний контекст вызова. По этому переменную limit котрую мы собираемся использовать в дальнейшем мы проинициализируем значением поумолчанию,



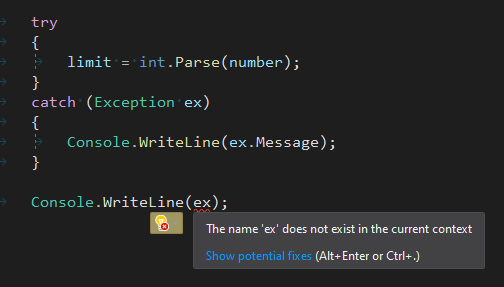
которое будет ей присвоенно в случае, если Parse не будет выполнен корректно. Вот итоговая реализация:



Запускаем нашу программу. И снова мяу вместо числа шагов



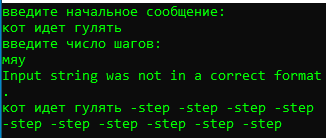
Как видим получаем те самые 10-ть шагов. Что же именно делает try catch, внутрь оператора try помещается код при выполнении которого возможно появление исключения, и если таковое случилось – срабатывает код внутри оператора catch. В случае если всё хорошо и переданная строка парсится в число – программа выполняется как обычно. В случае же мяю выполняется код обработки исключения. . Хорошо, с порядком выполнения кода оператором try catch мы разобрались, но выше фигурировал объект исключения, а в catch явно что-то есть в этих круглых скобках.



Да это то самое место, где мы можем получить доступ к самому объекту исключения, для этого нам надо к объявления типа Exception добавить название переменной, которая будет проинициализированна в блоке catch и может быть обработана в его дочернем контексте. Обратите внимание, зона видимости переменной ex не выходить за границы catch

Откуда я узнал, что у ex есть message внутри и что его можно вывести в консоле? Да все оттуда же. Поставил точку после объекта ex и прочитал описание для свойства Message.

Запустим программу, введем некорректные данные, поймаем объект исключения и обработаем его. Как видите внутри исключения есть человекочитаемая информация, которая понятна пользователю.

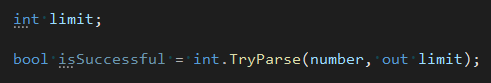


Программа стала куда устойчивее, когда мы добавили обработку исключения, но что если мы хотим дать пользователю возможность ошибаться и вводить данные снова? Нам бы метод парсинга который не приводит к исключению когда в него попадает неправильное значение...

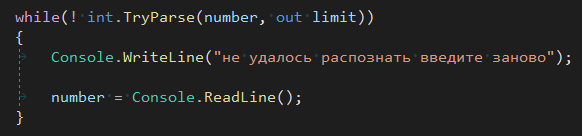


И такой метод есть! Int.tryParse Тут мы видим много всего нового, во первых в этот метод передается два параметра, первым из которых является исходная строка, а вот вторым инициализируемая методом переменная! Слово out в данном случае гаратнирует нам, что в ходе выполнения данного метода переменная будет проинициализированна. А возвращает метод значение типа bool.

По этому можем проинициализировать переменную isSuccessful значением, которое вернет метод int.tryParse и убрать инициализацию у переменной limit.

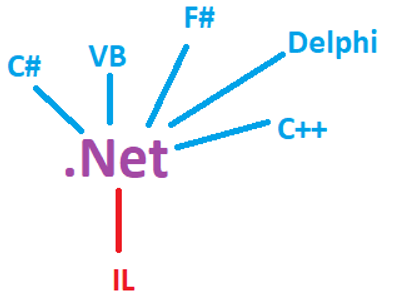


Теперь в случае, если пользователь ввел что-то что нельзя превратить в число на этапе парсинга, перменной limit будет присвоен 0. Но постойте, в чем же отличие, от работы с исключениями? Мы могли задать начальное значение limit 0 и было бы то же самое. Отличие, как раз в том, что метод возвращает bool то есть может служить условием в цикле, мы можем бесконечно долго просить пользователя ввести правильное значение и не приступать к выполнению программы до тех пор, пока оно не будет полученно.

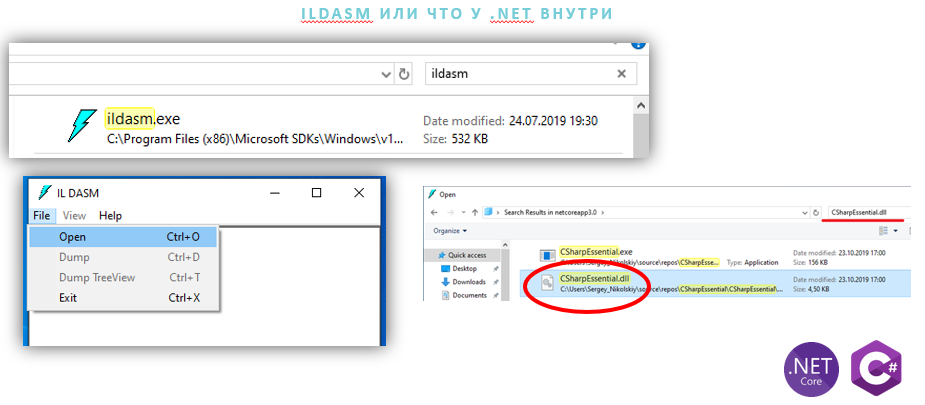


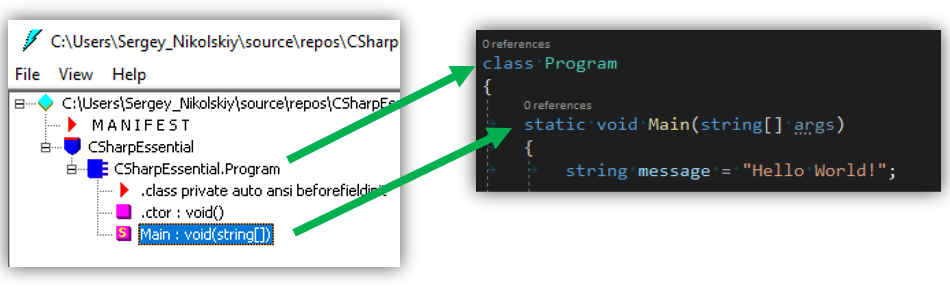
В результате программа будет просить ввести корректное число до тех пор. Пока пользователь этого не сделает. Конечно мы могли бы завязать эту логику на объект исключения – но это не очень хорошая практика. В будущем мы подробно поговорим про исключения и почему не стоит сыпать ими на каждом шагу, пока же давайте запомним: Исключения нужны для исключительных случаев.

Практическая часть оконченна, теперь снова немного теории. Вы наверняка слышали о том, что язык C# является частью платформы .Net. Пришло время рассказать что это такое, и как они связаны между собой. .Net представляет собой програмную платформу обеспечивающую возможность написания кода на целом кластере языков программирования. Первоначальная версия .Net Framework вышла в далеком 2002 и представляла собой (по неоффициальной информации) ответ микрософта на платформу Java только под Windows, но развитие продолжалось и вот в 2016 выхоит первая версия .Net Core с этого момента можно считать .Net полностью кросплатформенным. А учитывая, что в .Net обладает интеграцией с облачным сервисом Azure от той же Microsoft. Можно сказать что платформа .Net позволяет не только реализовать в рамках себя практически любое приложение, но и развернуть его в дружественной среде, которая разработана и поддреживается тем же вендором. Таким образом C# является частью обширной экосистемы, написанные вами программы не будут брошенны на произвол судьбы. Один и тот же код написанный на C# можно запустить и под Windows и под Linux и на мобильном устройстве. Более того! Приложение написаное на C# может использовать библиотеку написаную на Visual Basic или Delphi , главное. Что бы все эти библиотеки были скомпилированы в рамках платформы .Net. Как же это получается?



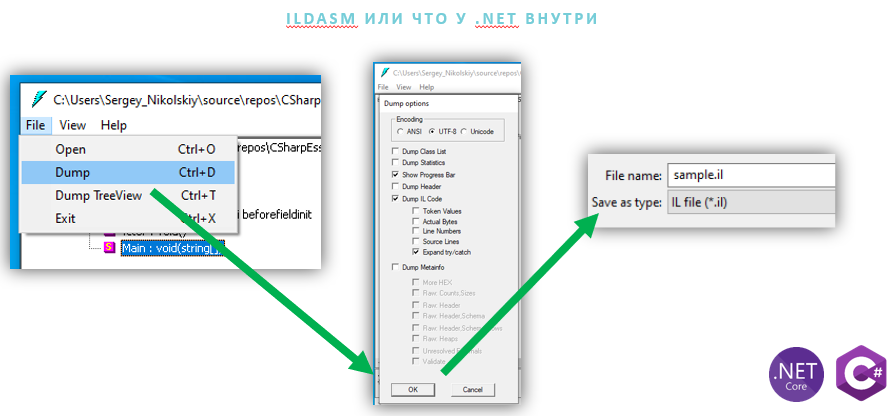
Дело в том, что код на любом из этих языков компилируется платформой в особый промежуточный язык Intermedia Language наша программа, кстати, тоже в него скомпилировалась, давайте помотрим как она выгдлядит. Для этого нам понадобиться специальная утилита: Ildasm, которая входит в комплект Visual Studio. Набираем в поисковом окне ildasm и запускаем её. Открываем меню File -> Open, и находим своё приложение в проводнике. Я искал по имени CSharpEssential.dll Открываем:

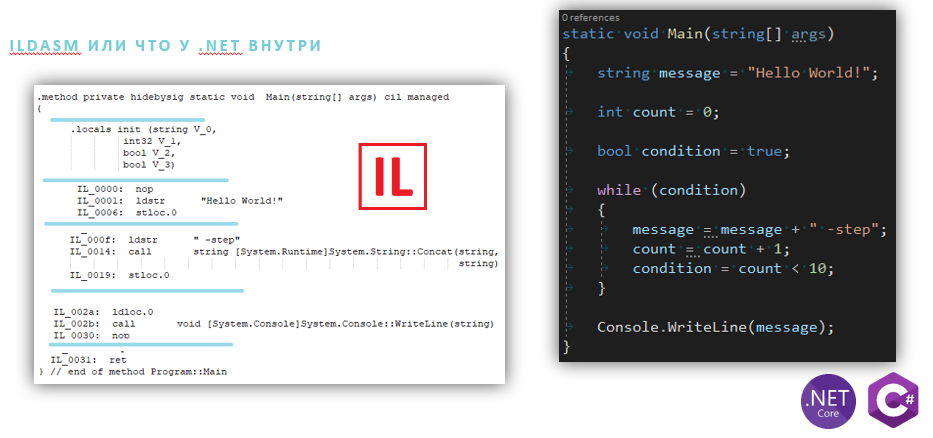




Разворачиваем плюсики, и что же мы видим? Main, Program, то же самое что и здесь. Хорошо, а где тут код-то? Что бы увидеть не только описание методов и классов, но и непосредственно код на Intermedia Language нам потребуется сделать ещё несколько действий.

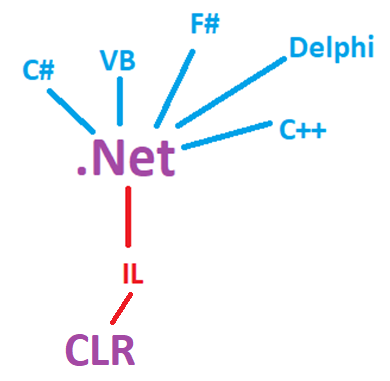
Снова идем в File -> Dump, оcтавляем настройки поумолчанию, ок, и сохраняем куда нам удобно файл с расширением \*.il, открыть файл можно в notepad.





И вот мы видим тот самы IL в который .Net скомпилировал код на писанный на C#, Не очень человекочитаемо, но это не для людей написанно. А для кого? Кто будет выполнять этот код? Промежуточный язык выполняется внутри CLR (Common Language Runtime) которая, в свою очередь является реализацией стандарта CIL (Common Language Infrastructure) и прекрасно понимает весь тот не очень человекочитаемый код, который мы видели на слайде выше.

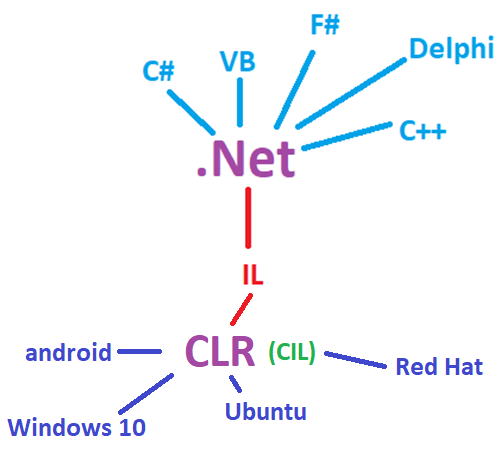
Другие языки программирования будучи скомпилированными на платформе .Net так же породят код на IL который ничем не будет отличаться! Таким образом, библиотеки написанные на разных языках будут приведены к единому, понятному CLR стандарту, тому самому CIL (Common Language Infrastructure).



На каких языках можно разрабатывать для платформы .Net – на любых которые можно скомпилировать в IL Причем возможности языка могут выходить за пределы стандарта CIL как, например C++. Правда писать на таком языке под платформу .Net придеться с некоторыми ограничениями. По приведеным здесь ссылкам можно ознакомиться с подробностями.

* [C++ для .Net](https://docs.microsoft.com/ru-ru/cpp/dotnet/dotnet-programming-with-cpp-cli-visual-cpp?view=vs-2019)
* [Visual Basic .Net](https://docs.microsoft.com/ru-ru/dotnet/visual-basic/)
* [F# .Net](https://docs.microsoft.com/ru-ru/dotnet/fsharp/)

Именно CLR отвечает за исполнение IL кода в рамках конкретной конкретной платформе. Таким образом, IL является платформонезависимым промежуточным языком, исполняемым внутри CLR Вот так вот она и получается, кросплатформенность.



Как видите C# является вершинкой обширного и интересного мира .Net разработки, мира в котором вы можете реализовать любые проекты, запуститься на любой платформе и всегда расчитывать на поддержку надежного вендора.

Подведем итоги:

* Мы познакомились со средой разработки C# (Visual Studio)
* Помотрели реализации на C# базовых операций.
* Разобрались в устройстве платформы .Net
* Написали первую, очень простую, программу на C#

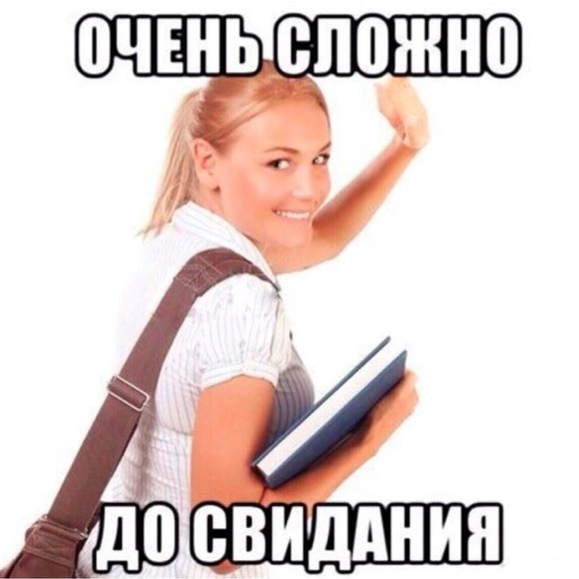
Теперь давай рассмотрим ситуацию, когда нам нужно написать приложение, которе гораздо болеше по объему чем то, которое мы написали в первой части методички. И гораздо сложнее. Что если наше приложение будет оперировать каким-то сложным бизнеспроцессом? Например что если мы будем писать систему отвечающую за выдачу денег через банкомат. Или робота который будет расслыть оповещения. Одним словом. Делать нечто такое, что нельза описать в рамках одной инструкции? Или для чего нужно будет написать такую бльшую и сложную инструкцию, что при малейших исправлениях мы получим трудно уловимую ошибку.

Что же делать? Пользоваться правильными инструментами.

Одним из таких инструментов является следование объектно ориентированной парадигме программирования (ООП), она позволяет нам строить достаточно гибкие и легко масшатбируемые приложения содержащие при этом сложную логику. Но для начала давайте разберем слово парадигма, что это вообще такое и будет-ли она писать код за нас?

* **Парадигма** - набор концепций или шаблонов мышления, включая теории, методы исследования, постулаты и стандарты, в соответствии с которыми осуществляются последующие построения, обобщения и эксперименты в области
* **Парадигма -** совокупность достижений, признаваемых всем научным сообществом в тот или иной период времни и служащая основой дальнейших исслодований
* **Парадигма** – общепризнанный образец или пример того, как наданном этапе развития стоит подходить к решению проблем в данной области.

Из вышеизложенного становиться понятно что у нас появилось ещё одно сложное непонятное понятие:



На самом же деле всё куда проще:

**Парадигма** - подход к решению задачи.

Иными словами ООП за нас код писать не будет, и приложение за нас не разработает, но ООП объяснит нам какая структура должна быть у нашей программы и из каких элементов она должна состоять. ООП – это набор принципов следование которым позволит нам избежать проблем при построении больших и сложных приложений.

**Создатель ООП – Алан Кей формулировал их так:**

1. Все является объектом
2. Каждый объект является экземпляром класса
3. Класс определяет поведение объекта
4. Классы организованы в иерархию наследования
5. Каждый объект обладает независимой памятью
6. Вычисления производятся путем взаимодействия между объектами

Как мы видим ключевыми понятиями в ООП является объект и его класс, начнем с определений:

**Объект** – логически связанная совокупность данных и действий от них зависящих.

А так как объекты определяеются их классами (следует из вышенаписанного), то для понимания ООП нам необходимо рассмотреть понятие класс.

**Класс можно понимать абстрактно как тип данных.**

О классах объектов и их реализации в C# мы поговорим позже, сейчас же давайте поговорим о концепции ООП как таковой. Она направлена на управление сложностью – чем легче вам понять код, тем проще вам изменять его поведение или добавлять новое. Все данные структурированы и их поведение единоообразно в пределах одного класса. Какие же сейчас выделяют основные концепции в ООП?

A picture containing text, book

Description automatically generated

**Инкапсуляция** - это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали  
реализации от пользователя. Проще говоря, мы сокрываем детали реализации какого-то поведения от пользователя. Пользователю не обязательно знать как работает микроволновка, он знает какие кнопки нажимать (вызывать методы) чтобы она работала.

**Наследование** - свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Мы можем рассмотреть две микроволновки разных фирм, их функции могут отличаться довольно сильно, однако, каждая из них умеет делать то, что умеет и абстрактная «базовая» микроволновка – подогревать еду.

**Полиморфизм** - это свойство системы использовать объекты с одинаковым ожидаемым поведением (интерфейсом) без информации о классе и внутренней структуре объекта. Мы знаем что у каждого автомобиля есть руль и что с помощью него можно менять направление движения. Детали реализации тут вторичны.

**Абстракцию** часто выделяют как четвертую концепцию ООП:  Абстракция - это способ выделить набор значимых характеристик объекта, исключая из рассмотрения незначимые. В математике этот процесс часто называют построением математической модели – выделить лишь значимые для задачи свойства объекта.

Хорошо, так почему же все-таки именно ООП так популярен? Какие именно задачи он может помочь нам решить?

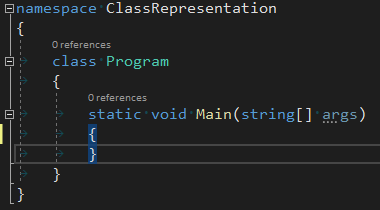
A picture containing game, light

Description automatically generated

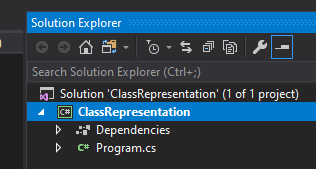
ООП достаточно прост в освоении, поскольку сознание человека в общем-то объектно-ориентированно. Основные концепции всех языков на земле при построении предложений – подлежащее и сказумое, то есть существительное (данные) и глагол (действие с этими данными).

1. При решении задачи методами ООП мы выделяем какие-то важные данные, задаем им желаемое поведение и объединяем в классы. Для которых мы выстраиваем иерархию. Например мы создаем класс студента содержащий имя, возраст, группу, факультет, массив оценок - данные и методы ходить на занятия, делать задания и сдавать экзамены - действия. У нас получится универсальный чертеж для cоздания любого студета в нашей бизнесмодели.
2. Мы контролируем использование наших данных. Доступ к модификации данных объекта полностью определяется внутри класса.
3. Из чего вытекает следующее преимущество: если мы захотим изменить программу или добавить в нее что-то, это будет сравнительно просто сделать. В хорошо спроектированном приложении изменения нужно делать лишь в одном месте – там где это действие (или эти данные) определено в классе и больше негде.

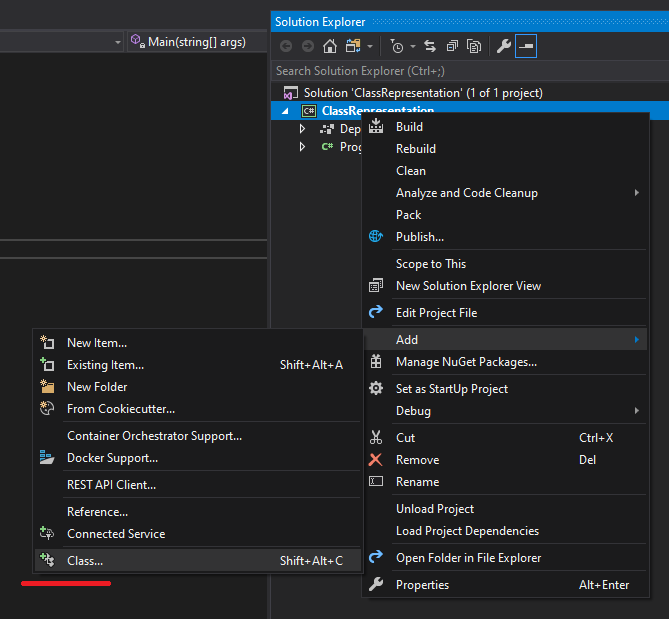
Теперь давайте глянем на классы и объекты с точки зрения их реализации в C#. Создадим новое консольное приложение, назавем его ClassRepresentation, и сотрем все внутри метода Main вот что мы увидим:



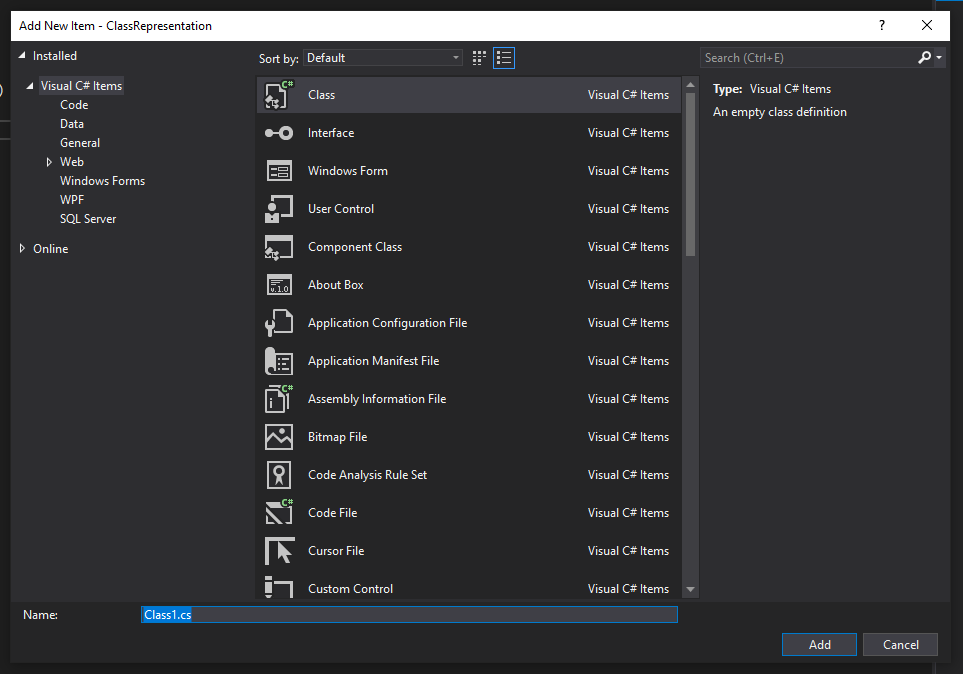
Давайте создадим наш первый класс, откроем solution explorer



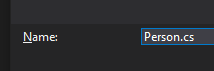
Кликнем правой кнопкой мыши на имени приложения и выберем add -> class



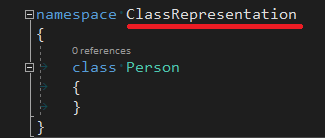
По клику открывается окно для добавления новых элементов в приложение:



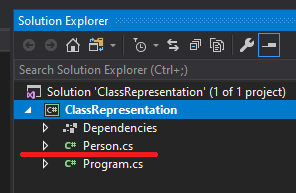
В нем сразу выбран элемент класс и visual studio пердлагает нам его имя, меняем имя на Person обратите внимание, что вновь созданный класс имеет расширение .cs которое однозначно нам сообщает, что это элемент написанный на C#, при переименовании следим за тем. Что бы не стереть расширение.



Нажимаем Add и visual studio открывает файл с новым пустым классом:



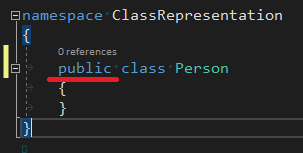
Обратите внимание, созданный нами класс имеет то же самое пространство имен, что и основная программа, более того, в нем точно так же фигурирует ключевое слово class, а в solution explorer от отображается вот так:



Имя файла полностью совпадает с именем созданного нами класса, данное поведение является общепринятым, никто не мешает нам внутри одного файла насоздавать множество классов, поменять имя файла на отличное от миени класса итд, но это считается не очень читаемым и делать подобные действия лучше только по очень веской причине.

**Наполним получившийся класс смыслом.**

Для начала добавим модификатор доступа:



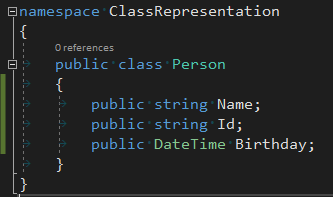
Слово public означает, что данный класс доступен повсюду внутри нашего приложения внутри пространства имен ClassRepresentation и за его пределами. Модификаторы доступа есть у всех програмных элементов: у классов, полей, констант, методов. Благодаря модификаторам доступа мы можем инкапсулировать данные – объединять их в значимые сущности и настраивать доступ к ним.

Теперь, вспомним что класс нужен нам для создания объектов, в соответствии с определением объекта нам нужно будет оперировать с данными и действиями от них зависящими, таким образом в нашем классе надо определить эти самые даннные и действия, начнем с данных.

Объект Person должен обладать:

* Идентификационным номером.
* Именем
* Датой рождения

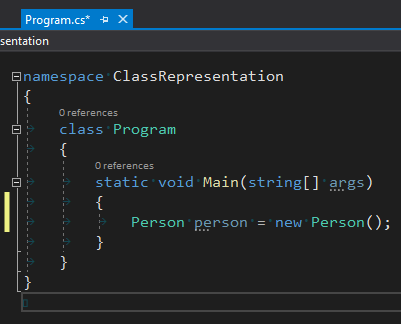
Добавим в класс соответствующие поля:



Разберем структуру добавленной конструкции. Сначала мы указываем модификатор доступа, для простоты мы воспользуемся тем же public, что и для класса. Затем мы пишем тип данных, которые собираемся хранить в поле и затем имя, после чего следует ; которая заврешает декларацию поля.

Обратите внимание, в рамках класса все поля должны иметь уникальные имена, как до этого переменные в зоне видимости.

Хорошо, класс Person наполнился смыслом, давайте сооздадим его объект и попробуем с ним что-либо сделать. Для этого в solution explorer внутри нашего проекта найдем файл Program.cs который содержит точку в хода в наше приложение и нпишем вот такую конструкцию:

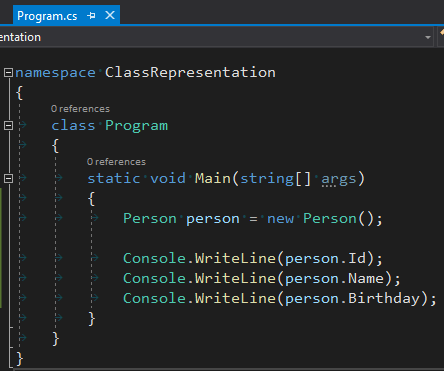


Разберем что именно мы написали: мы объявили переменную с именем person класса Person и проинициализировали её путём вызова конструктора поумолчанию для класса Person.

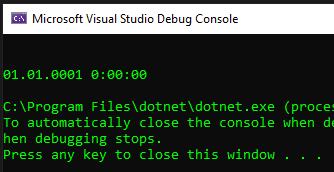
Постойте! Но мы не писали в классе Person никаких конструкторов! У нас там были только три поля для хранения данных и все.

Дело в том, что каждый класс в C# по умолчанию содержит специальное действие которое создаёт объект этого класса и называется это действие конструктор. Если мы не написали в классе конструктор явно, то он будет создан без нашего усчастия, автоматически. Для создания нового объекта через конструктор мы используем ключевое слово new как в вышеуказанном примере. Имя конструктора должно полностью совпадать с именем класса в котором он определен, а поскольку создание объекта – это действие (метод), мы ставим круглые скобки после написания имени конструктора.

Хорошо, объект создан, давайте посмотрим какие данные будут в нем изначально. Для этого напишем вот такой код:

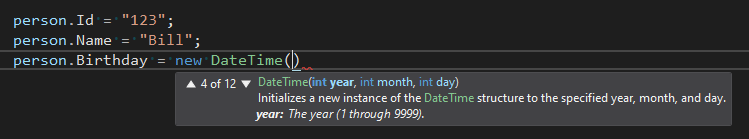


И запустим приложение:

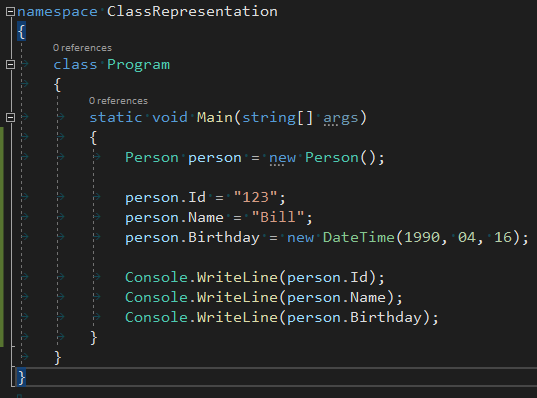


Приложение отработало без ошибок и вернуло нам две пустых строки (значения string поумолчанию) для Id и Name соответственно, и значение поумолчанию типа DateTime для поля Birthday. Отсюда мы делаем вывод о том, что конструктор поумолчанию для класса Person создал объект типа Person и пронициализировал его поля значениями поумолчанию.

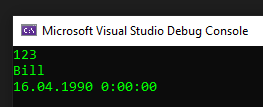
Давайте попробуем модифицировать данные которые храняться в нашем объекте, для этого обратимся к полям объекта person и запишем в них значения. Обратите внимание, при инициализации переменной Birthday я обращаюсь к конструктору типа DateTime выбираю его четвертую перегрузку (о перегрузках конструктора и как их писать поговорим ниже) которая принимает на вход три занчения целочисленного типа: год, месяц и дату.



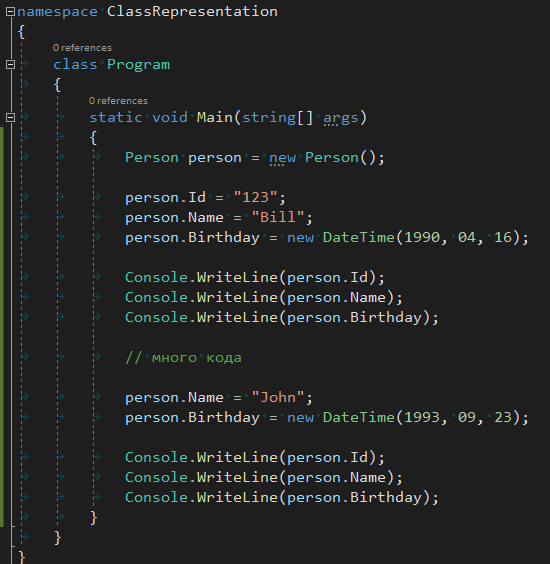
А в целом у меня получился вот такой код:



Который при выполнении выводит переданные в объект person данные



Так наш объект получил имя, личный номер и дату рождения. Как вы понимаете, по бытовой логике личный номер у человека может меняться, а вот дата рождения и имя – вряд-ли, а вот у нашего объекта person в его текущем исполнении – без проблем допишем нашу программу:



Запустим, и посмотим что будет:



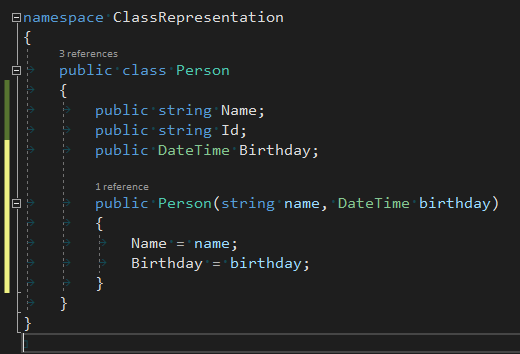
Получилось, что объект person на старте приложения содержал данные о человеке по имени Bill а потом, в какой-то момент времени, он начал содержать данные о человеке по имени John! При этом это тот же самый person, который у нас был изначально.

Давайте признаем такое поведение нежелательным и, попробуем, каким-то образом защитить данные Birthday и Name в person от модификации после создания объекта.

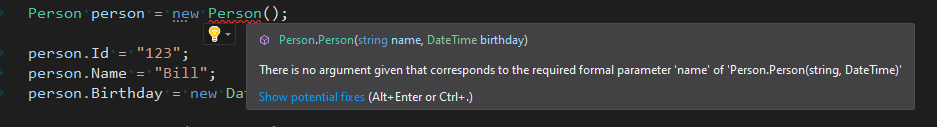
Было бы идеально сделать так, что бы имя и дату рождения можно было задать прямо в момент создания объекта. И мы это можем. Давайте напишем такой конструктор в классе Person (мы помним. Что всё поведение объекта person задается в его классе, а значит и меры надо предпринимать там) который бы ещё на моменте создания определял имя и дату рождения. Снова открываем solution explorer и переходим на файл Person.cs в котором мы прописывали класс Person.

Как мы уже отмечали выше, класс Person уже содержит конструктор поумолчанию, иначе мы бы не смогли создать объект этого класса, так что нам нужно написать ещё один контруктор для Person. Ничего страшного, **в C# класс может содержать сколько угодно конструкторов, пока они отличаются параметрами.** А мы, как раз, такой и пишем, сначала объявим модификатор доступа, так как мы собираемся вызывать кнструктор извне, то это снова будет public. Затем напишем имя класса, в нашем случае это Person откроем круглые скобочки и перечислим через запятую необходимые нам параметры с их типами string name, DateTime birthday. Обратите внимание, параметры у конструктора начинаются с маленькой буквы – это не обязательно, мы можем написать их как захочется, но есть соглашение о том, что все параметры у методов (действий), а конструктор – это вполне себе метод пишутся с маленькой буквы. Затем на новой строчке в фигурных скобках пишем те дейсвтия, которые надо выполнить в конструкторе, а именно проинициализировать поля Name и Birthday. Обратите внимание, внутри класса Person мы можем свободно оперировать его полями: считвывать их них значения, присваивать новые, всё что угодно.

Вот что у нас получилось:

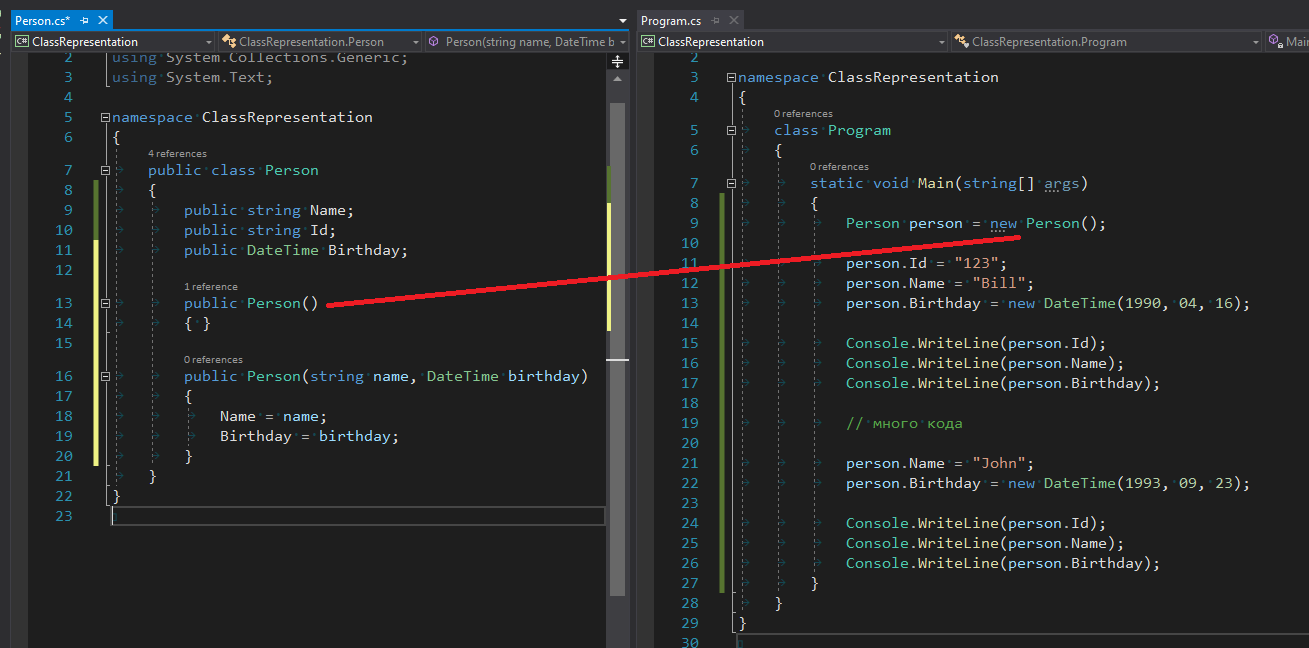


Вернемся в Program.cs, а там мы видим сообщение об ошибке:

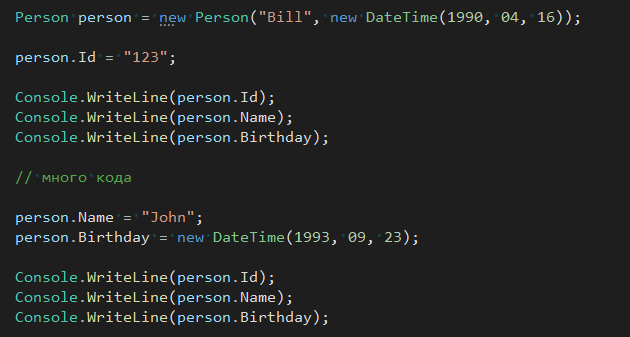


Дело в том, что как только мы написали руками в классе конструктор, C# решил, что конструктор поумолчанию для нашего класса больше не нужен (мало-ли мы не хотим иметь возможность создавать пустые объекты класса Person) и убрал к нем у доступ. Соответственно, компилятор думает что мы хотим обратиться к только что написанному конструктору, а он принимает два параметра, а мы их не передаем! Вот он и ругается.

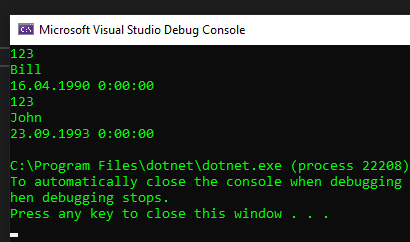
Давайте вернемся в класс Person и просто напишем конструктор без параметров, раз уж C# убрал его у нас. Снова начинаем с модификатора доступа и снова он унас public, затем снова пишем имя класса и круглые скобки, на этот раз в фигурных скобках не будет никаких действий, значения поумолчанию характерные для их типов полями присвоятся сами, как видим ошибка в Program.cs исчезла! Компилятор снова использует конструктор без параметров, но на этот раз это не конструктор поумолчанию который C# создал за нас, а нами написанный конструктор, в котором мы можем выполнять какие-то действия, дома попробуйте присвоить полям объекта созданного в таком конструкторе каие-нибудь значения и убедитесь, что данные сохраняются:

****

Но вернемся к нашей проблеме: поля Name и Birthday позволяют менять свои значения во время жизни объекта, давайте в Program использовать созданный нами корнструктор с параметрами, остальной код оставим без изменений:



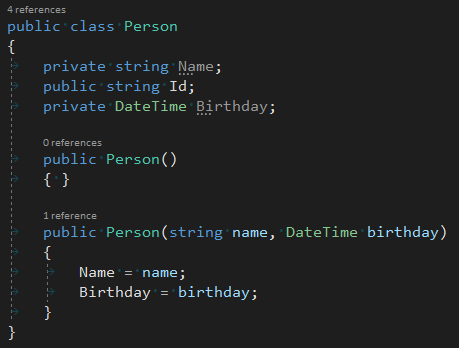
Запускаем:



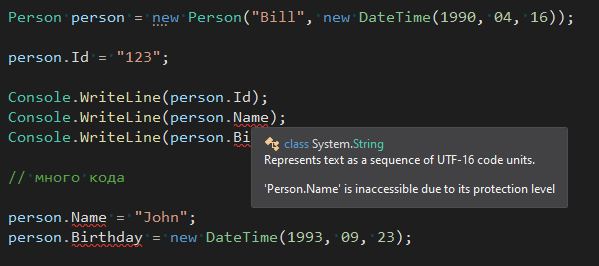
И что же мы видим? Да нам больше не нужно явно присваивать значения полям Name и Birthday за пределами класса Person, но у объекта этого класса мы по прежнему можем модифицировать данные в этих полях как угодно!

Как бы нам сделать так, что бы поля Name и Birthday можно было изменять изнутри класса Person, но при этом они были бы недоступны для изменения снаружи? У самого объекта данного класса?

Давайте для начала попробуем изменить модификатор даоступа у этих поле, вернемся в Person.cs и поменяем его с public на private. Вот так:



Вроде всё хорошо, данный модификатор доступа позволяет оперировать полями внутри класса, компилятор ни на что не ругается. Вот только посмотрим что делается при этом в Program.cs, а там происходит вот что:

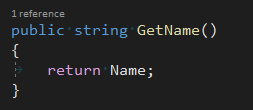


При попытке прочитать значение из нужных нам полей выскакивает ошибка компиляции, которая говорит нам о том, что данные поля недоступны в связи с их уровнем защиты. И если тот факт, что мы теперь в них ничего не можем записать (на скриншоте ниже это видно) нас полностью устраивает, тот отказ в доступе на чтение совершенно неприемлем!

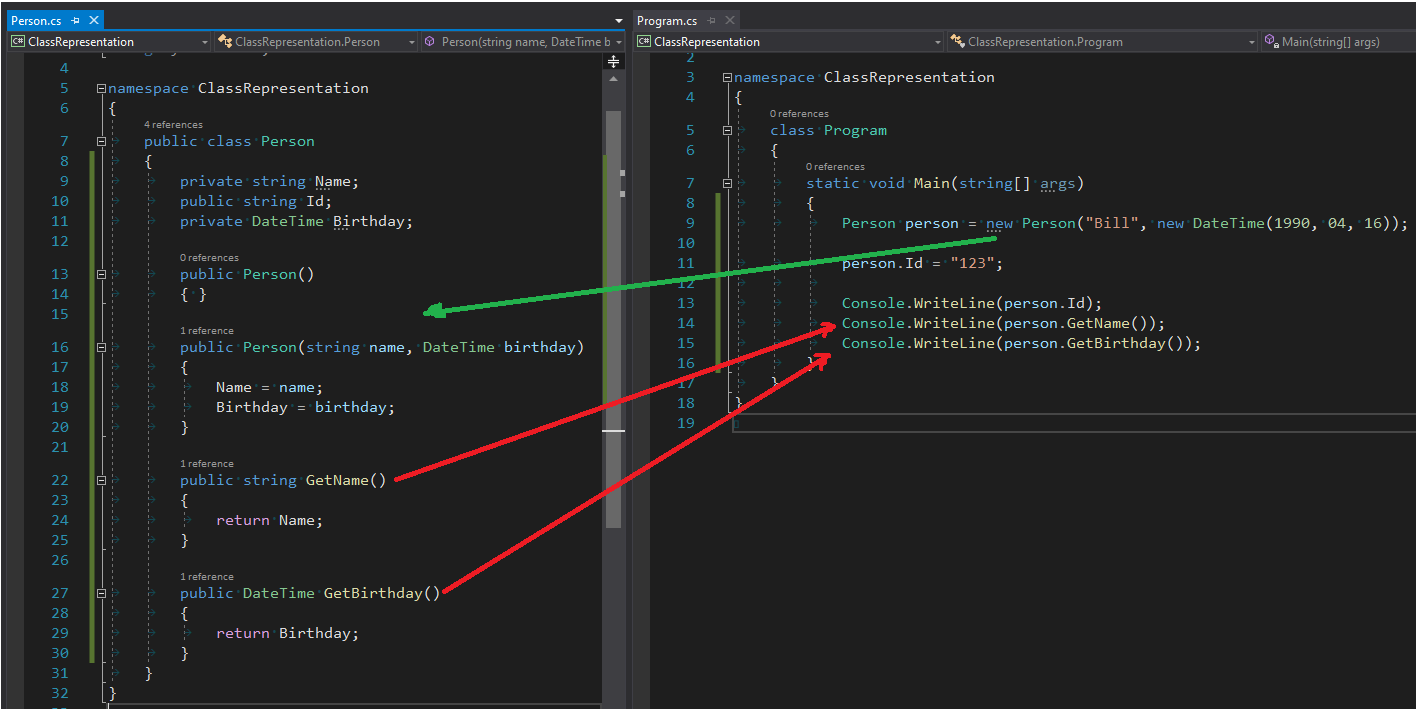
Что же делать, давайте подумает что мы имеем, с одной стороны мы не хотим давать доступ извне класса на запись данных в поля Name и Birthday, с другой стороны мы хотим как-то извлекать содержащуюуся в них информацию.

Но ведь чтение данных и запись данных – это действия с данными, то есть, теоретически, мы можем снабдить класс Person специальными методами (действиями) которые позволят нам получать данные из полей Name и Birthday и не позволят туда ничего записывать! И вот уже эти самые действия мы снабдим модификатрами доступа public и будем обращаться к ним снаружи класса Person! В то время как сами действия на чтения смогут обращаться к данным содержащимся в полях Name и Birthday, так как они (действия) будут это делать внутри класса Person.

Перейдем в Person.cs и напишем придуманные нами методы, начнем с Name. В первую очередь определимся с модификатором доступа, мы уже знаем private и public нам нужен public, данный метод будет возвращать значение свойства Name которое имеет тип string, так что и сам метод будет иметь тип string, имя метода должно однозначно рассказвать о том, что он делает, по этому мы назовем его GetString а параметров нам никаких ненадо, поэтому после имени мы поставим пустые круглые скобочки. Затем, в теле метода (в фигурных скобочках) мы напишем return – это ключевое слово позволяет нам вернуть из метода значение того типа, который мы написали перед именем метода во время его объявления, а после return напишем Name и поставим точку с запятой для завершения команды. Вот что у нас получилось:



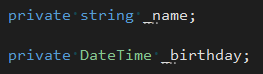
Метод для получения значения поля Birthday будет написан аналогично, вы разберетесь с ним самостоятельно. Теперь в Program.cs для получения значений Name и Birthday мы будем обращаться не к полям, которые у нас стали private, а к методам которые эти значения возвращают. Вот такая унас получилась картина:



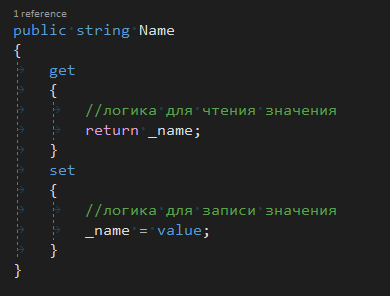
Мы принимаем данные в конструкторе, на этапе создания объекта, а получаем их значения через специальные публичные методы. Таким образом данными можно свободно оперировать внути класса Person но не снаружи! Снаружи они доступны только для чтения.

Мы добились того поведения, которое было желаемо с самого начала: данные можно задать только при создании объекта, а получить в любое время. Мы разделили запись и получение данных в классе Person.

Такого рода разделение является типичной задачей, поэтому для её решения есть типичная конструкция, она называется **свойство** и в отличие от **поля** позволяет написать раздельную логику на получение и запись данных, давайте напишем свойство для поля Name. Для начала переименуем Name, в соответсвтии с общепринятыми практиками поля с модификатором доступа private начинаются с нижнего подчеркивания и пишутся с маленькой буквы. Перепишем поля Name и Birthday в соответсвии с данным соглашением.



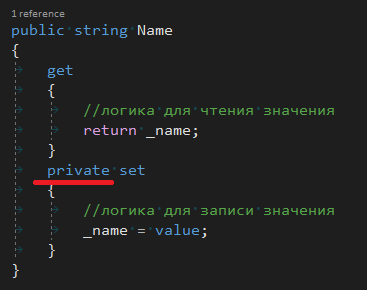
Конструкция любого свойства строиться следующим образом:



При этом get и set это обычные методы! Преднозначение их строго предопределено, как и имена, а вот логика в них ведет себя точно так же как и в любом другом методе! Мы можем помещать в них любые логические конструкции которые допускает C#.

Но важно помнить: set не может возвращать значение, а get всегда должет возварящать значение того же типа, что и тип свойства в кором он объявлен. Так же обратите внимание на особеннцю переменную value которую студия выделила синим что бы подчеркнуть её статус. Переменная value существует только в контектсе метода set и содержит в себе то значение, которое мы передали в свойство, но ещё не присвоили полю. Таким образом мы видим. Что для функционирования такого свойства нам необходимо поле, значение которого оно будет возварщать в get методе и устанавливать в set методе.

А так как get и set методы, то у них есть модификаторы доступа, то есть нашу проблему с тем, что бы значения Name можно было задать только внутри класса Person, а получить откуда угодно извне, можно решить расстановкой модификаторов доступа на get и set! Нам достаточно сделать set private, а get public. Кстати, поумолчанию C# назначет методам get и set такие же модификаторы доступа, как и у самого свойства, так что в нашем случае мы и так имеем public set и public get, давайте сделам set private вот так:



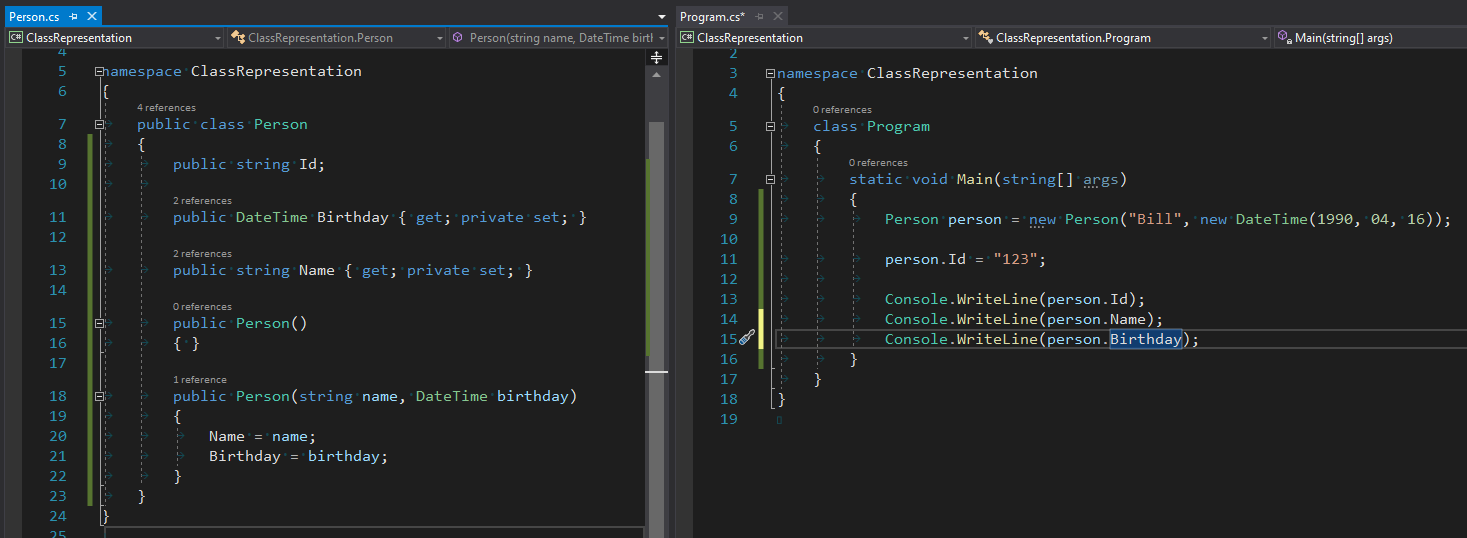
Довольно громоздко не правда ли? Если бы мы написали сложную логику валдации значений пригодных для Name в set методе. Или получали значение Name сожным образом в get методе, то такая громоздкость имела бы смысл, нам же просто надо хранить данные и не позволять их менять никому извне, Если никакой сложной логики в set и get не подразумевается, то свойство можно записать сокращенно, вот так:



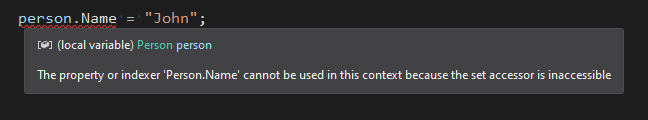
Подведем итоги.

* **Поле** – является элементом класса позволяющим хранить данные.
* **Свойство** – элемент класса позволяющий хранить данные и разграничивать права доступа на чтение и запись.
* **Метод** – является элементом класса позволяющим модифицировать данные .
* **Конструктор** – создает объект (экземпляр) класса.

Так что можно избавитьсят от специализированных методов на получение данных о дне рождения и имени, а заодно и от полей для их хранения. Вот, что получиться в итоге:



Обратите внимание, с точки зрения внешней системы свойство ничем не отличается от поля, но если мы попробуем записать извне значение в поле с private set, то получим вот такую ошибку:



Первый принцип ООП в действии! Класс ограничивает доступ к данным осуществляя тем самым их **инкапсуляцию.** Так же стоит помнить, что само объединение данных в логическую сущность (объект) так же является **инкапсуляцией.**

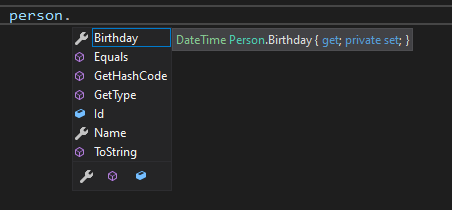
Вот полезные сылки по которым вы сможете подробнее ознакомиться с элементами классов:

* [Классы в C#](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/classes)
* [Поля](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/fields)
* [Свойства](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/properties)
* [Методы](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/methods)
* [Конструкторы](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/constructors)

С инкапсуляцией мы разобрались. А заодно познакомили вас с базовыми элементами из которых состои любой класс.

Теперь давайте рассмотрим концепции **наследования** и **полиморфизма**, о них стоит говорить только совместно, так как в C# наследование используется именно для достижения полиморфизма.

Для начала вспомним про нашу переменнную person класса Person в основном теле программы, перейдем в Program.cs и посмотрим какие элементы класса Person у этой перменной. Для этого напишем person и поставим точку. Вот что выдает нам visual studio



Как мы видим присутствуют свойства Name и Birthday, поле Id и четыре метода. Тип каждого элемета понятен из значка рядом с ним. Выше в лекции мы это уже разбирали. Свойства и поле мы написали сами, а вот откуда же взялись эти четыре метода и что он делают?

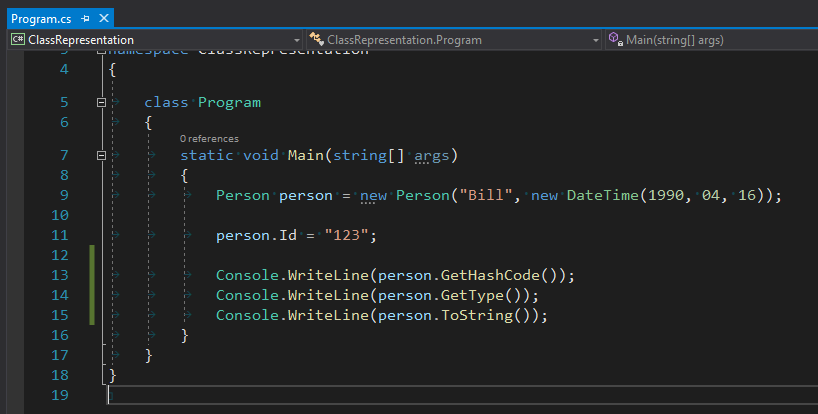
Дело в том, что любой класс в C# (и Person не исключение) является потомком класса Object, а данные методы:

* Equals
* GetType
* GetHashCode
* ToString

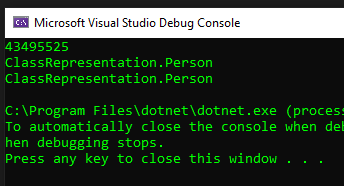
Являются методами класса Obgect и доступны из любого его потомка. Вот мы и стокнулись с первым проявлением наследования: публичные элементы предка доступны у его потомка.

Давайте подробно рассмотрим некоторые из методов доставшихся в нам в наследство от Object. Для этого в тело программы в файле Program.cs мы добавим их вызов и выведем результатты в консосль.

Вот так:

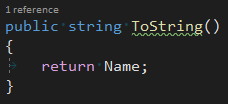


Запускаем, и вот что мы видим в результат:

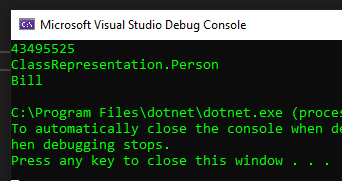


Первый метод вернул нам некое целое число (оно назвается Has кодом и пока нас не интересует), два друнгих вывели нам полное имя касса состоящее из пространства имени и собственно имени. И вот тут давайте начнем знакомство с полиморфизмом и поменяем поведение для метода ToString метод Equals пока не трогаем.

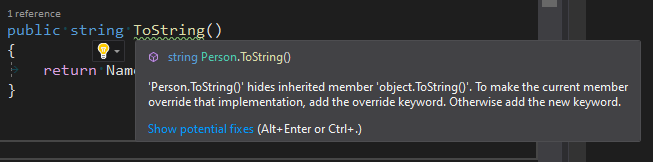
Идем в файл Person.cs и в классе Person пишем свой метод ToString() модификатор доступа у него будет public, возвращать он будет тип string и называться ToString, а входных параметров у него не будет, в теле метода пока просто вернем значение свойства Name и посмотрим что получилось:



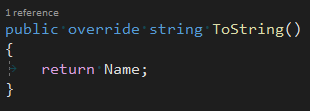
Как видим ошибки нет, хотя метод с точно таким же именем у нас уже есть в классе предке! Запустим код и псмотрим что будет:



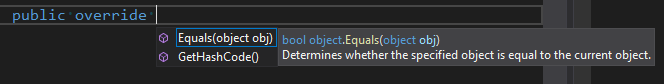
Был вызван метод потомка, получается, что метод потомка перекрыл собой метод предка. Такое перекрытие называется **переопределением** и если мы внимательно посмотрим на метод ToString в классе Person, то мы увидим. Что visual studio подчеркивает его зелёным, то есть у неё есть к нам какое-то предложение, мы можем как-то улучшить этот код, наведем мышку и увидим что нам предлагает visual studio



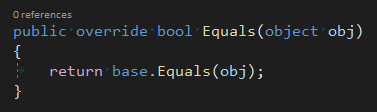
Visual studio нам сообщает, что метод с таким именем уже определен в классе предке и наше текущее определение будет перекрывать таковое у предка. Так же нам предлагают воспользоваться ключевым словом override для того, что бы однозначно определить наши намерения по **переопределению** данного метода. Последуем этому совету, вот что у нас получилось:



Давайте пойдем дальше и переопределим метод Equals, данный метод отвечает за сравнение объектов одного типа. В этот раз будем сразу декларировать намерение по переопределению, по этому при написании нового метода сразу напишем override после модификатора доступа (public как обычно). И что же мы увидим? Visual Studio сразу показала нам список методов предка которые мы можем переопределить:



Заметьте, метода ToString в списке нет, так как он уже переопределен нами выше, а метод GetType недоступен для переопределения в потомке! Для такого поведения существует специализированный модификатор доступа sealed но пока нам это неинтересно. Выберем Equals и нажмем Enter visual studio сделает за нас большую часть работы:



Здесь нам почти всё знакомо. Кроме типа входого параметра object и ключевого слова base, давайте разберем их подробно.

Тип object как раз и представляет собой тот самы класс Object от которого наследуются все остальные классы в C#, тут важно рассказать о другом свойстве наследования в C# по умолчанию наследника всегда можно использовать вместо предка. Это свойство называется **ковариантностью** и основанно оно какраз на том, что через наследника мы всегда можем получить доступ ко всем публичным элементам класса предка (его полям, свойствам, методам). А так как благодаря инкапсуляции нам доступны извне только публичные элементы, то с точки зрения внешнего кода совершенно неважно имеем мы дело с экземпляром предка или с экземпляром какого-то из его потомков. Нам в любом случае будет доступен один и тот же набор публичных элементов.

Еще раз:

**Ковариантность - возможность оперировать наследником вместо предка.**

Но постойте! Ведь мы только что поменяли реализацию метода ToString у нашего класса Person и теперь он будет вести себя подургому, не так как он это делал у предка! Да и в этом-то какраз и проявляется полиморфное поведение, передавая потомка вместо предка в метод для обработки мы можем НЕ МЕНЯЯ КОД добиваться разного поведения.

Чувствуете как у программы прибавилось гибкости?

Теперь поговорим о ключевом слове base благодаря ему мы можем получить доступ первоначальному элементу класса предка, то есть написанная visual studio реализация метода Equals принимает любой объект и отправляет его в первоначальный Equals, тот который был до того, как мы взялись писать свою реализацию.

Теперь мы понимаем visual studio написал за нас все обязательные компоненты для переопределения Equals и при этом ничего не сломал. Пользуясь пердоставленной заготовкой, давайте напишем свою реализацию медода Equals для класса Person. Для этого давате примем, что у всех людей в нашей системе будет уникальный Id который храниться в соответствующем поле, а что бы его никто не мог изменить после того, как экземпляр класса создан – сделаем поле приватным, и добавим его инициализацию в класс Person:



Мы заменили модификатор доступа с public на private и добавили инициализацию. Давайте рассмотрим подробнее запись, с помощью которой мы произвели инициализацию:

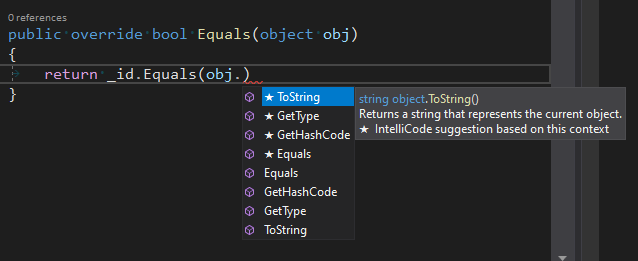
* Первым идет слово Guid – это обращение к статическому классу Guid (ели быть точным, то это структура, но в нашем контектсе можно считать её классом)
* После чего у статического класса Guid мы вызываем статический метод

Guid.NewGuid() который вернет нам новый экземпляр класса (структуры) Guid обратите внимание: здесь вместо конструктора мы обращаемся к специализированному статическому методу который создает новый объект.

* А уже у созданного такием образом объекта мы вызовем метод ToString()
* [Побробнее о том. что такое Guid](https://docs.microsoft.com/ru-ru/dotnet/api/system.guid?view=netframework-4.8)

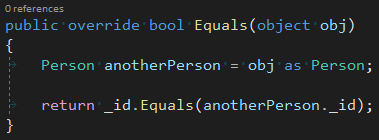
Вот сколько действий мы сделали написав цепочку: Guid.NewGuid().ToString()

Теперь каждый раз, когда мы будем создавать новый объект класса Person, ещё до вызова конструктора он будет получть значение для поля \_id. И пользуясь тем, что данное значение будет уникальным (для понимания почему перейдите по ссылке о Guid’ах) и неизменяемым в течении жизни объекта (благодаря инкапсуляции) мы можем испльзовать значение поля \_id для срвнения экземпляра класса Person с другим экземпляром в методе Equals. Давайте попробуем написать логику для такого сравнения:



И что мы видим? У значения переменной obj нет поля \_id так как она представляет собой перменную класса Object, который является предком всех классов в C# и ничего не знает своих потомках. Прежде чем делать сравнение объектов по значениям полей \_id нам необходимо привести перменную obj к классу Person и убедиться что это вообще возможно, так как obj может содержать объект любого класса который наследуется от Object.

Здесь важно понимать, что благодаря ковариантности мы можем передать в метод экземпляр потомка вместо экземпляра предка, но он сам не редуцируется до предка, он просто оборачивается в него как в обертку. То есть елси мы передатим в наш Equals другой экземпляр класса Person – он не деградирует до Object, а просто будет завернут в него как в оболочку. Давайте же развернем оболочку и получим Person. Так как оборачивание экземпляров одних классов в другие (особенно в их предков) достаточно стандартная процедура, то в C# есть для разворачивания таких обреток (распаковки) есть стандартная инструкция, вот так это выглядит:



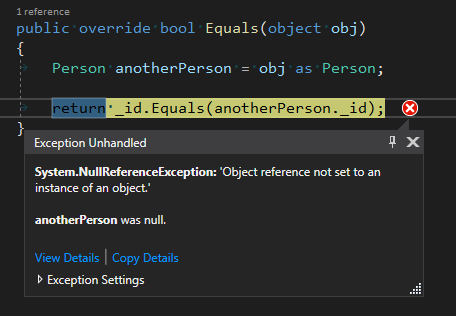
Теперь ключевое слово as позволяет попробовать распаковать obj в класс Person. Обратите внимаеие, вгутри класса Person мы имеем доступ к приветному полю \_id **другого экземпляра,** а не только того, которому принадлежит метод Equals, так как в C# инкапсуляция осуществляется на уровне классов. Но вернемся к распаковке, заметьте мы должны знать во что именно мы хотим распаковать полученный объект и явно это указать в инструкции, подробнее можно прочитать по ссылке:

* [Упаковка\Распаковка](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/types/boxing-and-unboxing)
* [Оператор as](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/type-testing-and-cast)

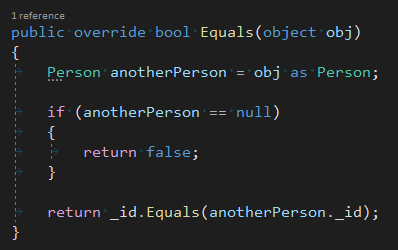
Хорошо, а что если распаковка закончиться неудачей? С точки зрения внешнего кода мы же не знаем как организована работа метода Equals. В Случае если мы передадим в метод что-то, что нельза распаковать в Person инструкция as не сможет проинициализировать переменную anotherPerson и в тот момет, когда мы попробуем обратиться к её полю \_id мы свалимся с исключением. Давайте убедимся в этом, в файл Program.cs добавим вот такой код:



Формально всё впорядке, экхземпляр класса string вполне можно упаковать в Obbject, а значит и передать в метод Equals экзепляра класса Person. Но в меоент выполнения мы получим вот такую ошибку:

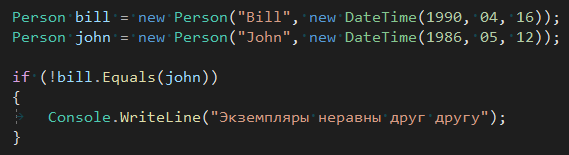


А произойдет она ровно потому, что инструкция as несмогла распаковать полученный объект в объект класса Person и переменная anotherPerson осталась непроинициализированна. Давайте дополним код метода Equals ещё одной проверкой, для того, что бы исключить подобные ситуации:

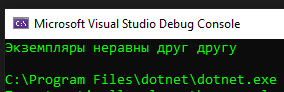


Как видите, нерпоинициализированная перменная будет содержать значение null в случае если в метод Equals был передан объект не того класса который мы ожидали внутри, воспользуемся этим и вернем false в этом случае. Как мы видим метод может содержать более одной инструкции return если это необходимо. Запустим программу и убедимся что все впорядке.

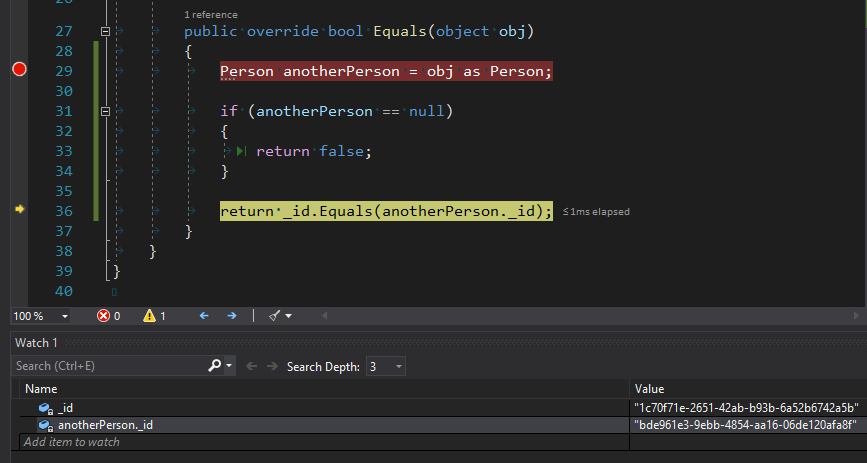
Теперь давайте в теле основной программы создадим вторую переменную класса Person и убедимся что метод Equals работает так, как мы ожидали. Так как метод Equals возващает тип bool мы можем использовать его операторе if в качестве условия и применять к нему любые другие логические операторы:



Дословно условие в if читается так в случае если перменная bill **не** (благодаря логическому отрицанию **!** мы инвертируем результат выполнения метода Equals) эквивалентна переменной john выполни код в фигурных скобках. Запускаем и убеждаемся что всё работает:

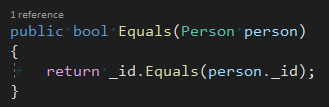


Поставим точку останова в методе Equals и посмотрим как именно будет происхлодить сравнение, я добавил окно watch для отображения значений полей \_id у текущего объекта и того, который был передан в метод Equals извне, подрпобнее о том, как добавить окно наблюдателя мы рассматривали ранее, вот что мы видим:



Их значения совершенно разные, соответственнго метод Equals вернет нам false, кторое затем будет инвертированно логическим отрицанием в true которое уже будет считанно оператором if.

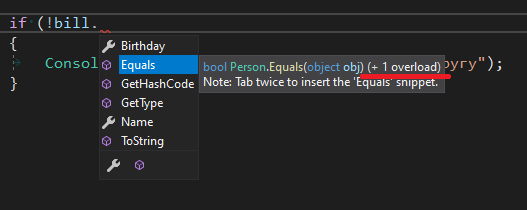
Теперь давайте познакомимся ещё с одним проявлением полиморфизма. Для класса Person мы переопределили базовый метод Equals в который можно передать объект любого класса который наследуется от object. Напишем же специфичный метод Equals который будет принимать только объекты класса Person.



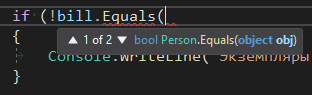
Как мы видим, visual studio не ругается, хотя новый метод называется точно так же как и предыдущий, дело в том, что методы в классе должны отличаться не по названию, а по сигнатуре: комбинации имени с типом параметров и их (параметров) колличеством. Подробнее вот тут:

[Методы и их сигнатуры](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/methods)

То есть Equals(object obj) и Equals(Person person) компилятор видит как разные методы класса Person. Давайте посмотрим в коде:



Как мы видим метод Equals во всплывающем окне указан один раз, но в описании для него написанно + 1 overload то есть есть ещё одна перегрузка! Таким орбразом компилятор подсказывает нам, что данный метод обладает полиморфным поведением: мы можем вызвать его с разными начальными параметрами и, в зависимости от этого, в нем может быть реализованна разная логика. Для того, что бы посмотреть сиггнатуры перегруженных методов нам достаточно написать после точки его имя и открыть круглую скобку, после чего visual studio покажет нам подсказку с всевозможными перегрузками выбранного метода:

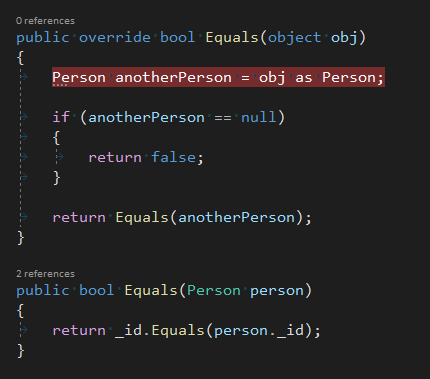


Навигация по перегрузкам осуществляется клавишами вверх вниз на клавиатуре:



Теперь, вот этот код из перимера выше будет автоматичекси использовать наиболее подходящую перегрузку, то есть вместо метода Equals(object obj) будет вызван метод Equals(Person person). Заметьте, мы никак не модифицировали внешний код, мы только написали более подходящую перегрузку к уже существующему методу и компилятор автоматичемски начал её использовать. Убедиться в том, что вызывается именно новый метод вы можете самостоятельно поставив точку останова в новом методе и запустив пограмму.

Вернемся в класс Person внутри метода Equals(object obj) мы используем ту же самую логику сравнения по значению поля \_id, что и в методе Equals(Person person) за исключением проверки типа упакованного объекта, мы можем улушить код вот таким образом:



Как и во внешнем коде, компилятор понимает. Что мы обращаемся к более специфичному методу благодаря тому, что мы переданный объект имеет тип Person.

Вот мы и познакомились с ещё одним видом полиморфизма: перегрузкой методов внутри класса.

А как нам получить два эквивалентных объекта? Давайте рассмотрим следующий пример, объявим переменную sam типа Person и проинициализируем её объектом из перменной bill:



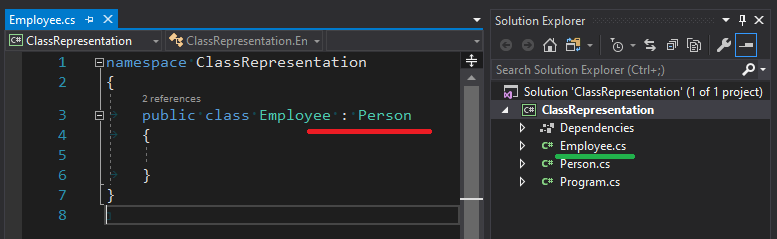
Передадим её в метод сравнения, для большей наглядности оставим остальное условие без изменений только добавим веточку else



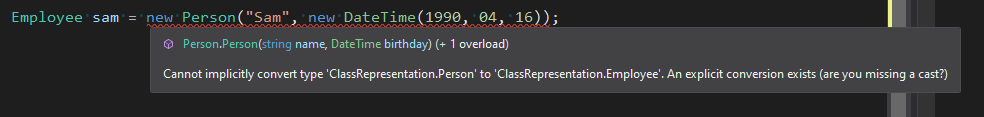
Запустим приложение и убедимся, что объекты эквивалентны:



Теперь усложним пример. До сих пор мы рассматиривали наследование в рамках дефолтного предка всех классов object, давайте создадим явного наследника класса Person. Для этого перейдем в solution explorer и, как мы уже делали до этого, создадим новый класс. Назовем его Employee, присвоим ему модификатор доступа public и укажем наследование в явном виде, вот так:



Наследник будет иметь доступ ко всем элементам класса предка, давате вернемся в тело основной программы и убедимся в этом. Для начала создадим экземпляр класса Employee и присвоим его новой переменной.



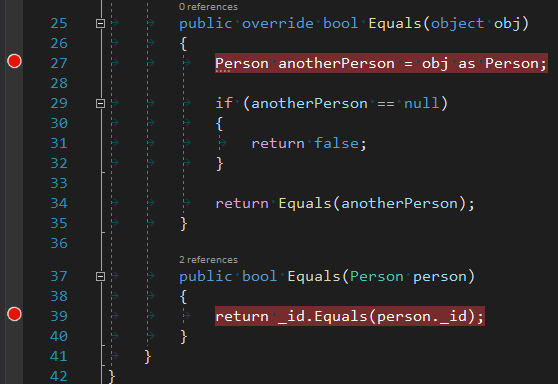
Компилятор отказывается пользоваться конструктором класса Person для создания его потомка! Хотя конструктор и является публичным элементом класса предка. Дело в том, что конструктор может создавать экземпляр только своего класса для конструктора ковариантность (использование потомка вместо предка) работать небудет, в C# конструкторы могут работать только с экземплярами своего класса такое свойство называется **инвариантностью.**

Инвариантность – свойство позволяющее работать только с экземплярами специфического класса. Не с потомками или предками, а только с тем классом который явно задан для данного действия.

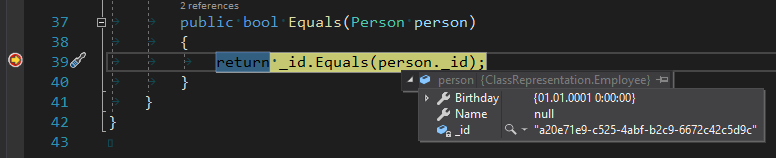
Раз конструкторы инвариантны, а созданный нами класс Employee лишен собственного конструктора, как же нам создать его экземпляр? Вспоминаем, что каждый класс имеет свой конструктор поумолчанию! Без параметров и специфических действий, используем его:



А теперь давайте посмотрим, что будет если передать переменную sam содержащую экземпляр класса Employee в метод Equals переменной bill которая содержит экземпляр класса Person. Так как метод Equals у класса Person имеет две реализации и мы (предположим) незнаем какая именно реализация будет использованна, поставим точки останова в обеих и посмотрим.

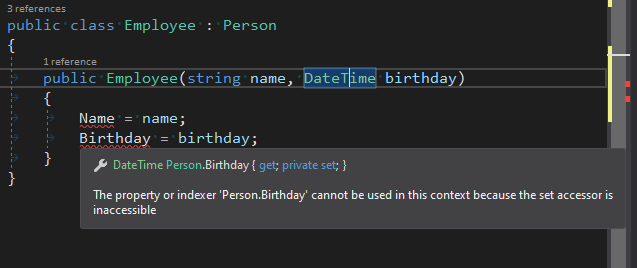


Сработает более специфичная реализация Equals так как благодаря ковариантности мы можем привести Employee к своему предку Person при передаче в метод:



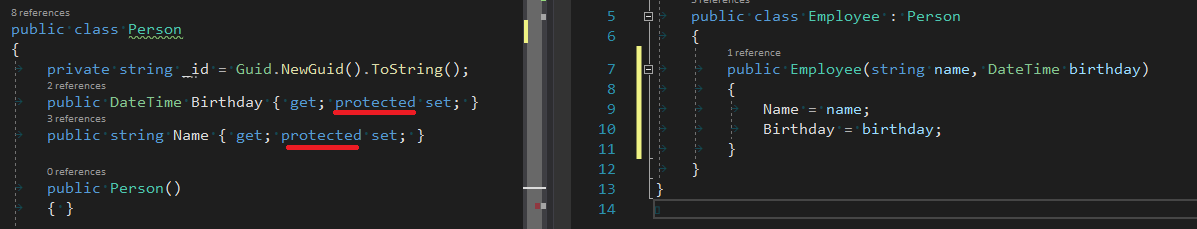
Методы, в отличии от конструкторов, в C# ковариантны. Наведем курсор на перменную person видите: C# понимает что это просто обертка для объекта потомка класса Employee (для удобства visual studio выводит нам полное имя класса вместе с пространством имен) и внутри себя данный объект содержит те же элементы, что и его предок.

Но, в отличии от предка, класс Employee не имеет возможности проинициализировать свойства Name и Birthday так как неимеет соответсвтующего конструктора, а конструктор из Person мы не можем использовать напрямую. Конечно мы можем написать для Employee точно такой же конструктор вот так:

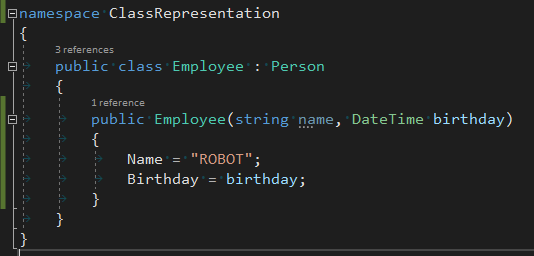


Но мы столкнемся с проблемой доступа, дело в том, что свойства Name и Birthday имют модификатор private на метод set то есть данный метод доступен ТОЛЬКО внутри своего класса! Только внутри класса Person можно задавать им значения.

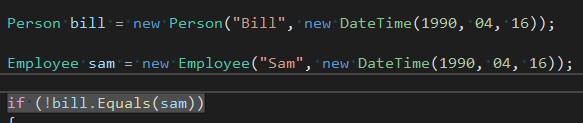
Тут можно пойти двуимя путями, во первых можно изменить модификатор доступа с private на protected, данный модификатор открывает доступ не только внутри класса который владеет данными элементами, но внутри всех его потомков, но всё ещё не дает модифицировать данные извне, полупаблик такой получается. Если мы меняем модификатор доступа у методов set, то это решает нашу проблему:

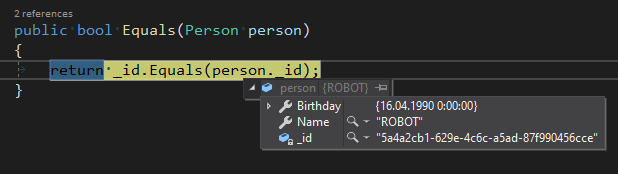


Но порождает новую, мы можем поменять данные свойств внтутри класса наследника и класс предок ничего не будет об этом знать! Например вот так:

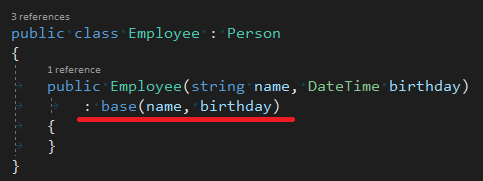


Выполним код. Зайдем в метод Equals с помощью точки останова и увидим довольно странное поведение:





Пока мы не заглянем в конструктор Employee мы не поймем что пошло нетак. Так что менять модификаторы доступа на метод set у класса предка может оказаться не такой уж хорошей идеей. Вместо этого мы можем прямо из конструктора потомка обратиться к конструктору предка используя всё то же ключевое слово base вот так:



Это позволяет нам при создании объекта класса Employee по цепочке вызвать конструктор его предка Person и выполнить в нем логику по присвоению значений свойствам Name и Birthday не нарушая их инкапсуляции и не отдавая контроль над их значениями за пределы класса Person.

В остальном написанный нами потомок может делать всё то же самое, что и Person поотношению к своему предку Object. Мы можем дописывать новые элементы, переопределять (с помощью ключевого слова override) поведение существующих и писать новые реализации к уже имеющимся методам с новым набором параметров. Вы можете самостоятельно проэксперементировать с этим. Ниже приведу несколько полезных ссылок:

* [Наследование в C#](https://docs.microsoft.com/ru-ru/dotnet/csharp/tutorials/inheritance)
* [Переопределение методов в C#](https://docs.microsoft.com/ru-ru/visualstudio/ide/reference/generate-override?view=vs-2019)
* [Модификаторы доступа в C#](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers)

Мы же перейдем к заключительной теме сегодняшней лекции. К Интрефейсам.