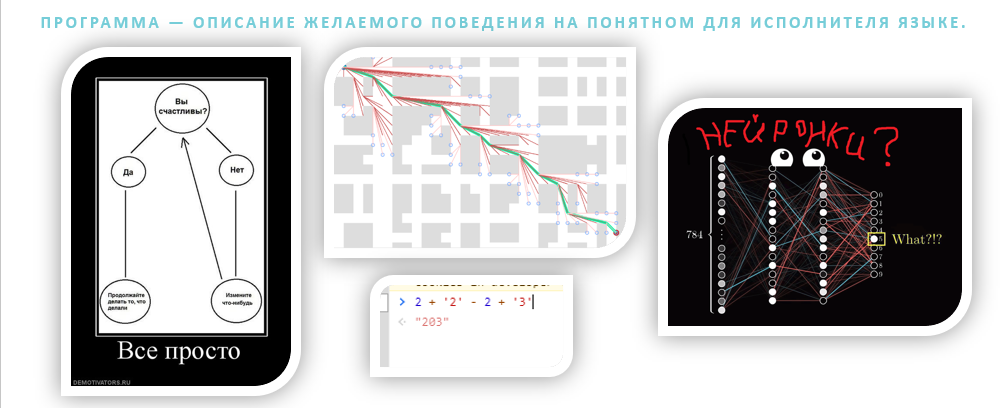
ОБЩАЯ ТЕОРИЯ И БАЗОВЫЕ КОНСТРУКЦИИ ЯЗЫКА C#

Все, что происходит когда мы включаем компьютер, телефон, да хоть бы и холодильник - является результатом действия программ. Любое, хоть сколько-нибудь сложное поведение электронного устройства – результат выполнения программы.

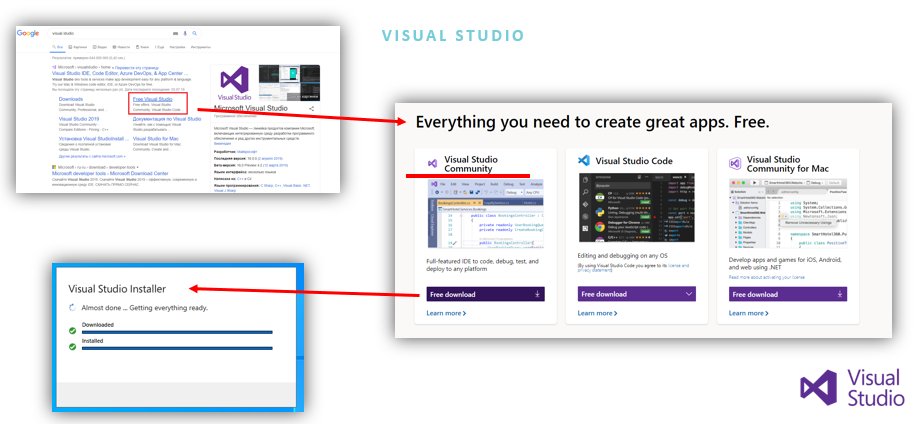


В основе любой программы лежит всего три действия:

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

В реальности, на пути написания универсальных программ на все случаи жизни лежит множество препятствий. Здесь я перечислил только некоторые из них.Для снижения сложности написания программ язык программирования должен уметь следующее:

* 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 программирования облегчает нам построение систем на уровне концепций.
* Кроссплатформенность позволяет запускать одни и те же программы под разными операционными системами.
* Специализированные среды разработки, существенно облегчают работу с кодом.

Начнем с самого начала, нам нужна среда разработки, будем сразу использовать подходящие инструменты. Заходим в google и набираем visual studio. Кликаем скачать, запускаем скачанный файл:Установщик загружен, проставляем галочки на нужных нам компонентах:

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

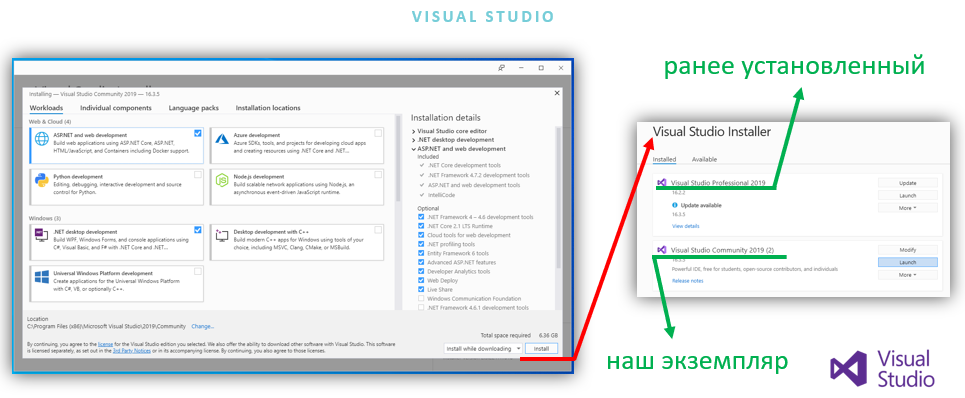
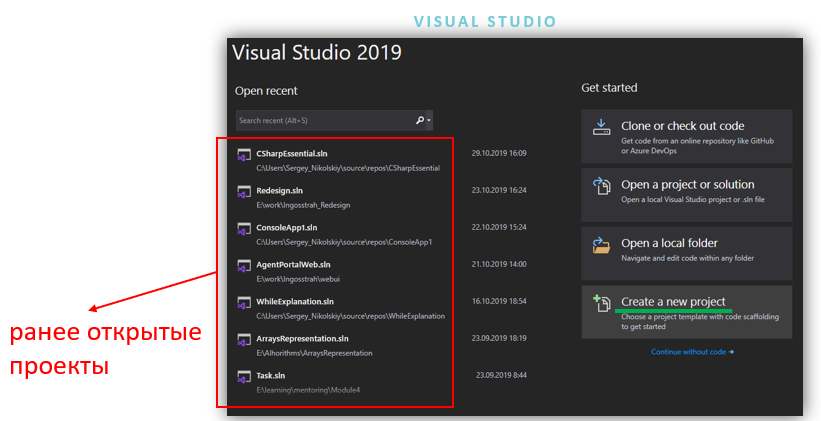
Обратите внимание есть

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

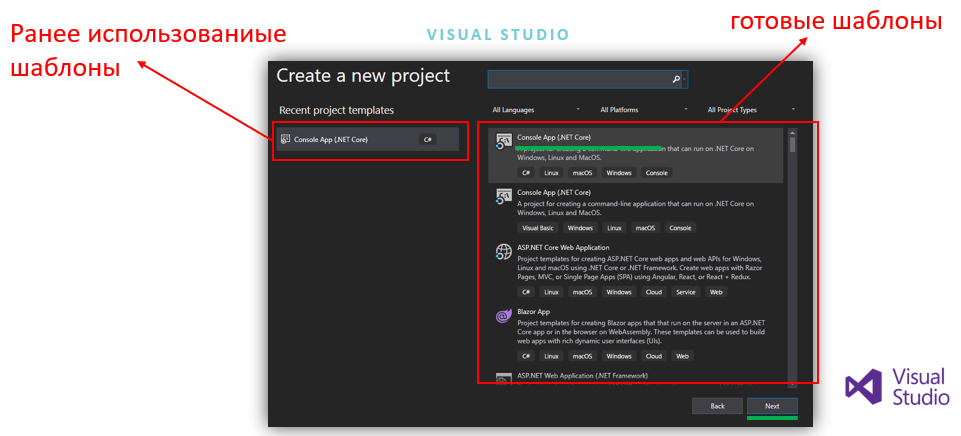
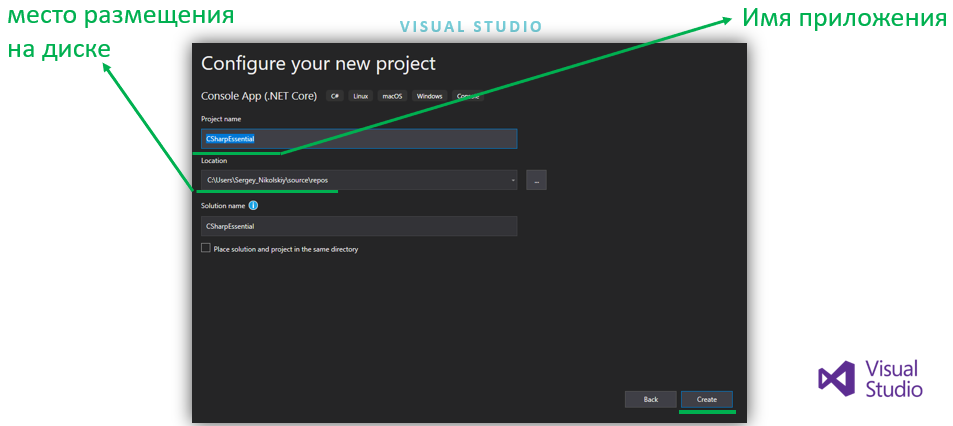
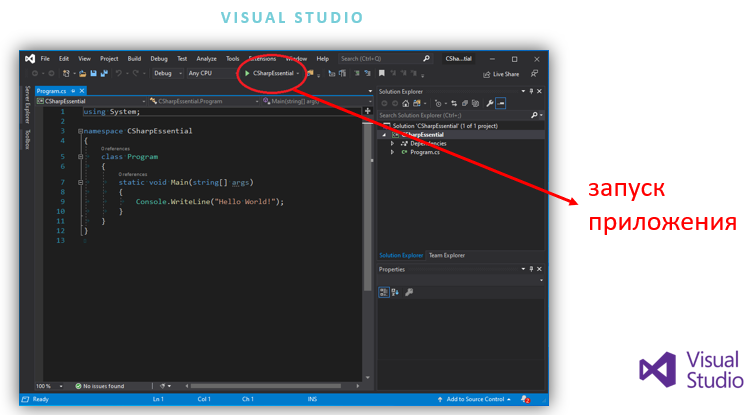
Другие языки программирования, такие как

* Python,
* C++.

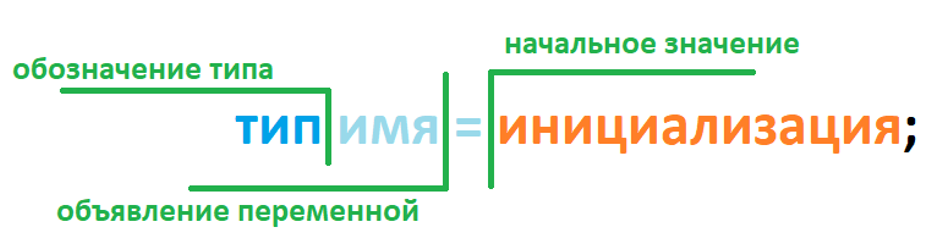
И всё это в бесплатной для личного пользования Community версии.

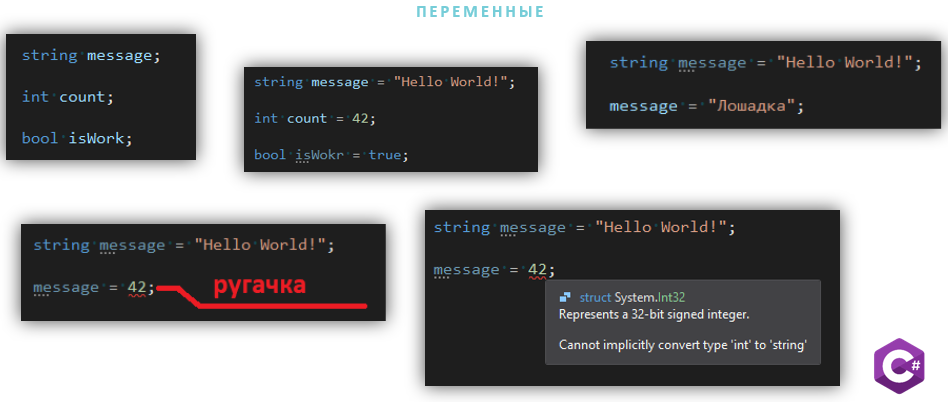
Нажимаем Install, по завершении установки запускаем visual studio. Обратите внимание, launcher знает о всех установленных на компьютере экземплярах visual studio.В начальном окне мы можем выбрать ранее запущенные проекты, сейчас же нам интересна вот эта кнопка: create a new project. Выбираем интересующий нас шаблон. Обратите внимание, каждый шаблон содержит краткое описание, которое включает в себя язык, целевые платформы и тип проекта. В нашем случае это:

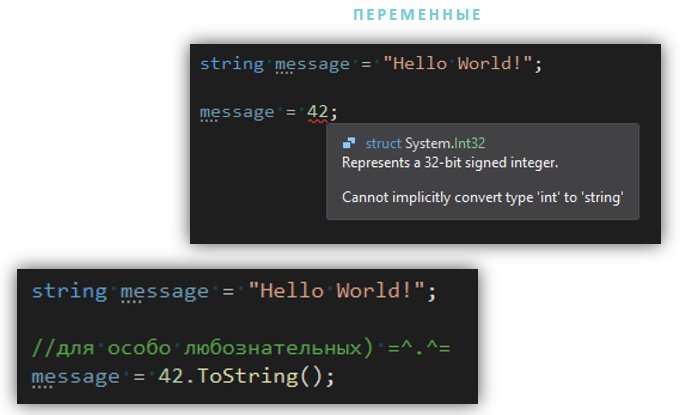
* язык C#
* платформы Linux macOs Windows
* тип приложения Console

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.Перед нами готовое консольное приложение. Жмем на зеленый треугольничек, и наслаждаемся своим первым Hello World приложением на написанным с помощью языка C# на платформе .Net Core.Несмотря на наличие такого инструмента как Visual Studio. Нам всё ещё нужно самим разрабатывать программное обеспечение и писать код. Начнем с азов языка C#

Одной из базовых конструкция языка C# является переменная, с этой концепцией вы наверняка знакомы по С++. Но, в отличии от C++, вам не придется вручную управлять памятью! За вас это делает сама платформа .Net, как это часто бывает в программировании – реальность несколько сложнее, но на данный момент примем, что управление памятью автоматическое.

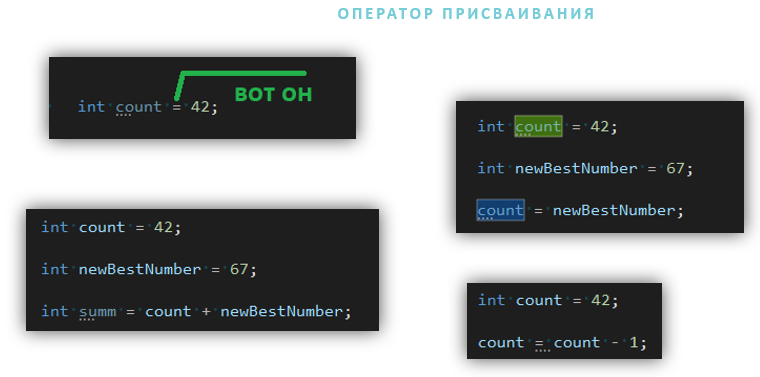
Вот общая схема объявления и инициализации переменных в рамках C#При объявлении переменных не обязательно сразу их инициализировать, это можно сделать во время выполнения программы. Главное, что бы новое значение имело тот же тип, что и предыдущее. Так как **C# - язык со статической типизацией**: однажды назначенный тип переменной не меняется в ходе выполнения программы.

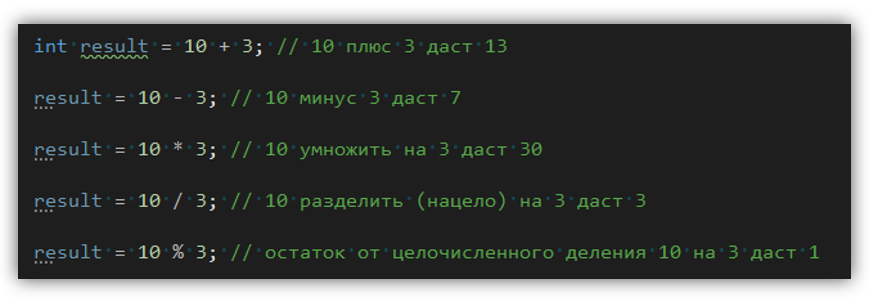
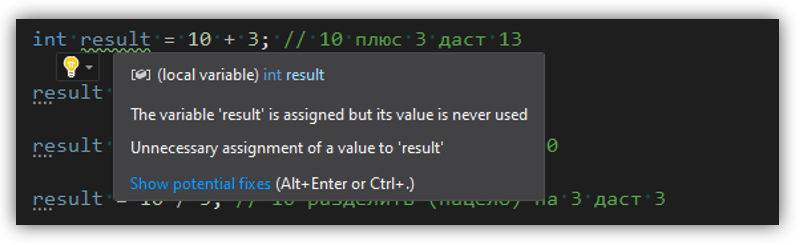
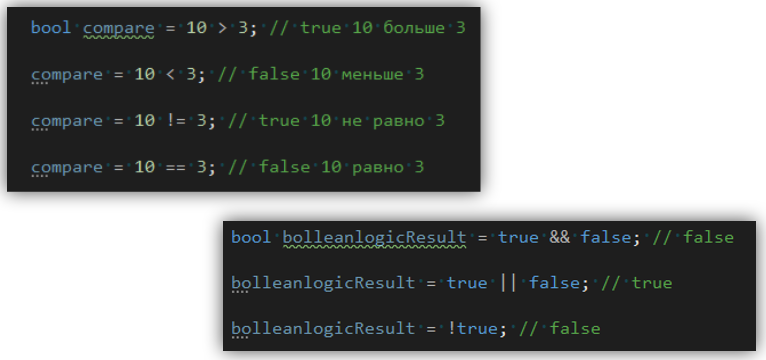
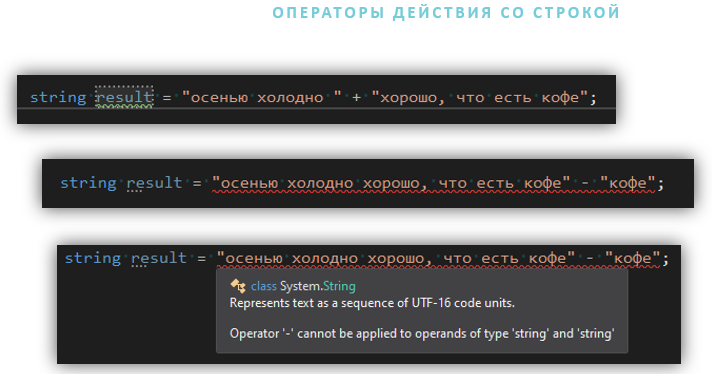
Как видите Visual Studio предупреждает нас об ошибке типизации ещё до того, как мы бы попытались запустить программу. Что бы понять, что пошло не так достаточно навести мышку подчеркнутую красным область.Превратить один тип в другой нельзя.

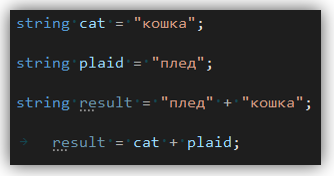
Но можно превратить значения одного типа в значения другого. Для этого есть специальный механизм конвертации: 

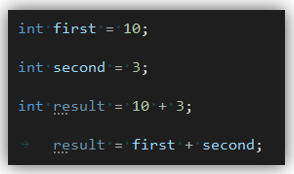


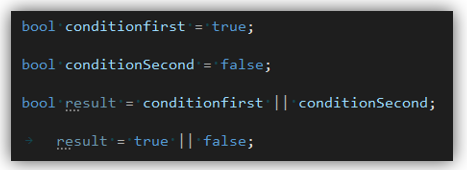
Так же ругачка появиться при попытке объявить в рамках зоны видимости (пока считаем, что зона видимости – это вся наша программа) две переменные с одинаковым именем. То есть компилятор отличает переменные по имени. Но переменные только хранят данные, а мы хотим добиться каких-то действий с данными. Тут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

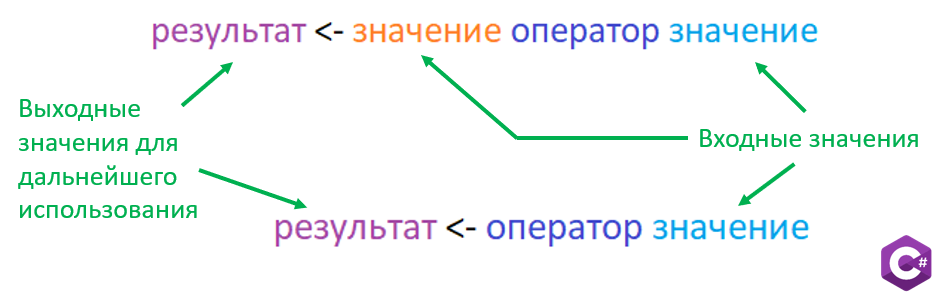
Для начала рассмотрим ОПЕРАТОР ПРИСВАИВАНИЯ он используется для передачи значения в переменную, как и в C++.Дословно написано: переменной с именем count типа int ПРИСВОИТЬ значение 42. Так же мы можем присвоить одной переменной значение другой. Или мы можем присвоить сумму значений двух переменных третьей.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

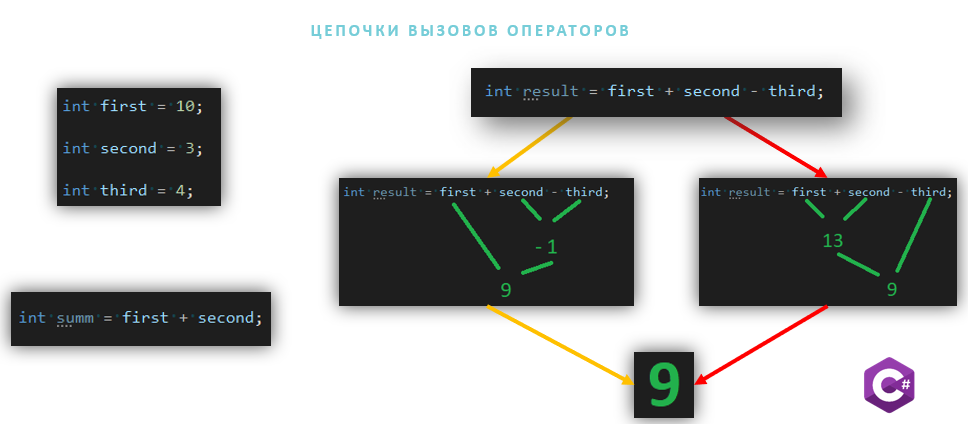
Начнем с математики, те самые плюс, минус, умножить, разделить (в нашем случае деление породило два оператора: целочисленное деление, которое просто отбрасывает остаток. И, наоборот, оператор, который возвращает остаток от деления).Visual Studio нам подчеркнула переменную result зеленым потому, что переменной result присвоено значение, но оно никак нами не используется в дальнейшем. Это не ошибка, поэтому и подчеркивание зеленое, это совет. Компилятор заботится о памяти компьютера, а любая переменная занимает место в памяти, размер этой ячейки динамически меняется в процессе выполнения программы и память освобождается в тот момент, когда переменная 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 присваиваем переменной compare того же типа. А так же операторы булевой алгебры: И(&&) ИЛИ(||) НЕТ(!). Именно они отвечают за формирование условий, без которых невозможно представить программу. Теперь давайте рассмотрим необычный оператор: конкатинатор строк. Он выглядит как обычный математический плюс, но он представляет собой совсем другое действие: он создает третью строку, которая содержит в себе две предыдущих. А вот оператора минус для строк не существует. Поэтому компилятор думает, что мы имели ввиду математический минус и любезно нам говорит, что в математический минус он не может передать строки – не получается.Операторы можно использовать не только со 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, логических – булевый, строковых – строковый.



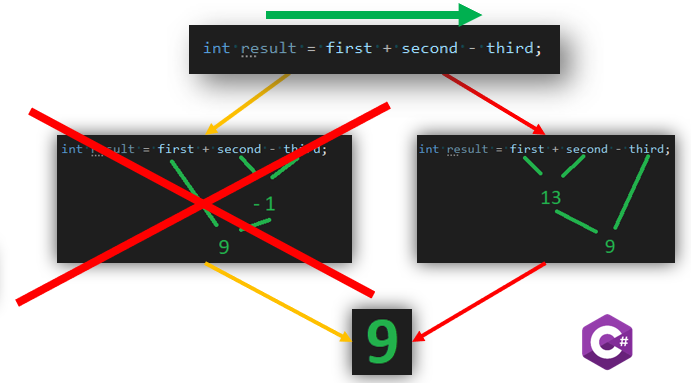




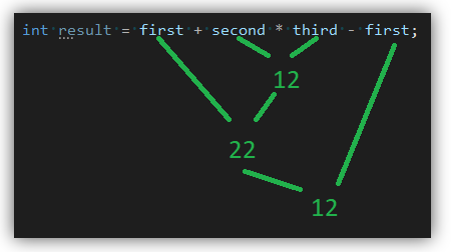
Оператор всегда возвращает значения своего типа. При одинаковом внешнем виде операторов, внутри они совершенно разные и обладают разной типизацией. До сих пор мы рассматривали операторы, которые укладывались в две простых схемы: Бинарные и Унарные. А раз результат выполнения любого оператора – это значение, и на вход операторы принимают значения, значит мы можем соединить операторы в цепочку.

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. Возьмем три проинициализированных переменных, и построим из них вот такую цепочку вызовов:Возможны два варианта поведения:

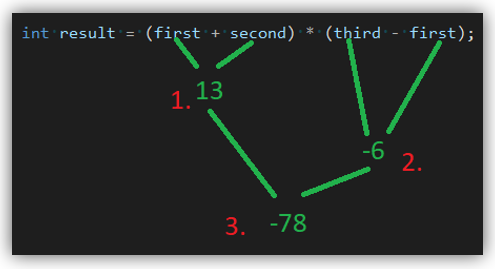
* Из 3 вычтут 4, а затем прибавят к 10
* К 10 прибавят 3, а потом вычтут из суммы 4

Да, результат в обоих случаях будет одинаковым, но вычисления-то шли разными путями. **Выполнение пойдет по второму сценарию:****При равном приоритете операторов вычисления пойдут слева на право.**

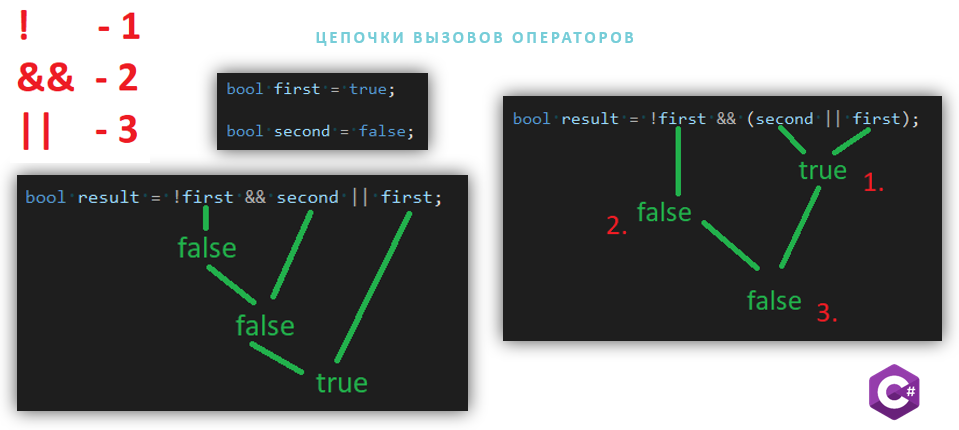
Это не всегда так, прежде чем делать такие выводы стоит свериться со справочной информацией, но большинство бинарных операторов ведут себя так. А оператор присваивания, в любом случае выполнится последним. Из этого правила мы делаем вывод о том, что у операторов есть приоритезация выполнения. Подробнее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



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

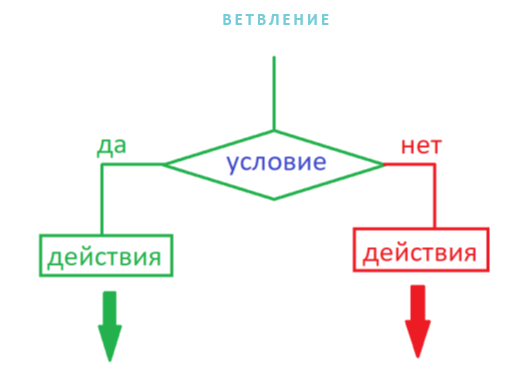


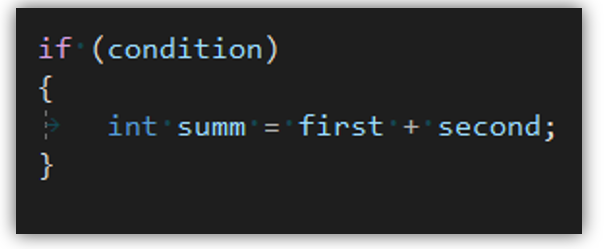
Мы добавили скобки, повысив тем самым приоритет операций сложения и вычитания по сравнению с операцией умножения. Слева направо: 10 складывается с 3, получаем 13. Затем и 4 вычитается 10, получаем -6, и только потом перемножаем -6 на 13 и получаем -78

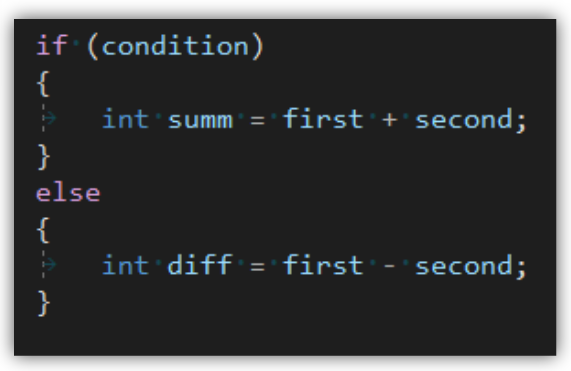
Логические операторы так же имеют свой порядок выполнения. Он представлен в левом верхнем углу. Сначала выполнится оператор отрицания, затем И, а в конце оператор ИЛИ. Скобки меняют порядок выполнения логических операторов так же, как и математических. Сначала выполнится оператор ИЛИ затем отрицание,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

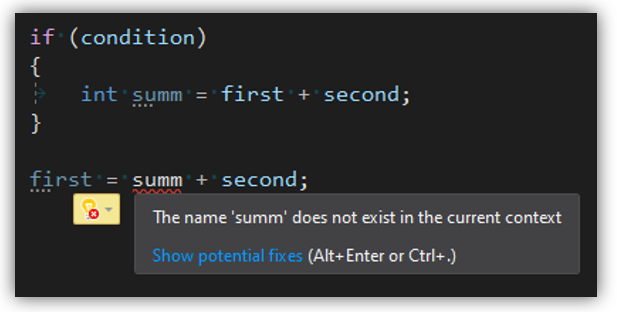
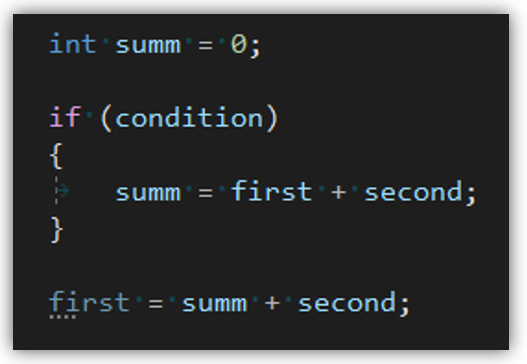
Начнем с ветвлений, В самом примитивном случае ветвление можно представить вот такой схемой, в которой условие – это некоторое выражение булевой алгебры (конструкция, которая вернет true или false), в соответствии с которым изменится ход выполнения программы. На ДА (true) выполнится зеленая ветка, на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

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

Для пояснения необходимости явного указания второй ветки ветвления нам необходимо ввести новое понятие.

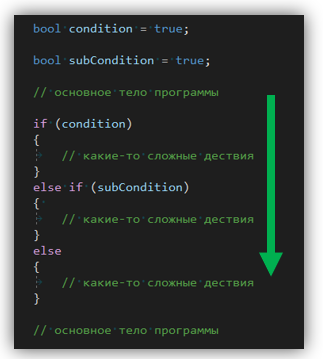
**контекст выполнения – это последовательность команд, выполняемых одна за другой.**

Посмотрим, что будет при попытке обратиться к summ за пределами фигурных скобок - компилятор не знает 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 переменные доступны только в нем. Мы видим, что переменная из родительского контекста доступна в дочернем. Объявим переменную summ до фигурных скобок. 

Проблема исчезла теперь значение summ будет влиять на дальнейшее исполнение программы в зависимости от того, выполнился блок кода в фигурных скобках или нет.

Теперь очевидно, что **else** позволяет изолировать код в рамках дочернего контекста вызова. В примере с явным указанием else код, относящийся к выполнению условия condition, изолирован от кода, который будет выполнен в случае, если condition == false. Как мы видим объявленные в рамках дочерних контекстов переменные доступны только внутри него и никак не повлияют на основное тело программы, тем самым мы создаем не зависящие друг от друга логические блоки.

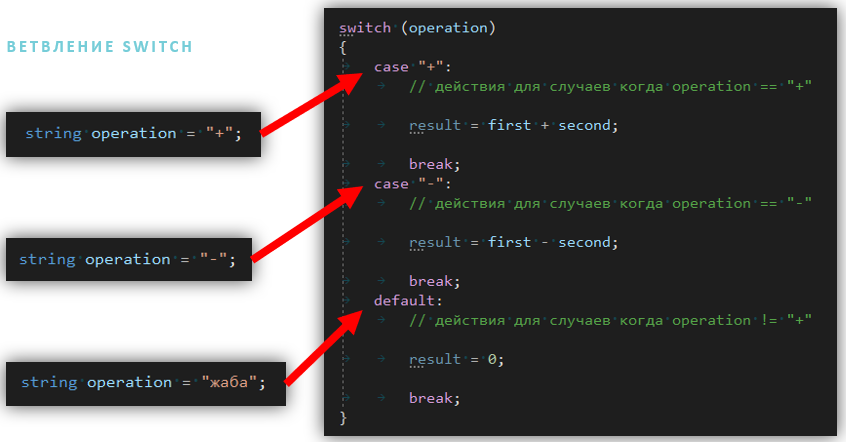
Так же мы вправе состыковывать if else операторы для проверки дополнительных условий, при этом приоритет проверки будет идти сверху вниз.



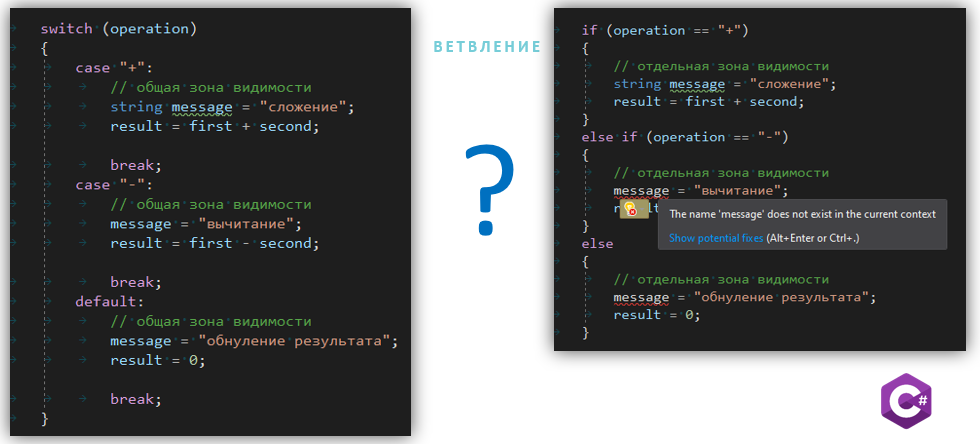
Условие subCondition будет проверено только если condition вернет false. Естественно ветвление может иметь сколь угодно вложенную структуру.

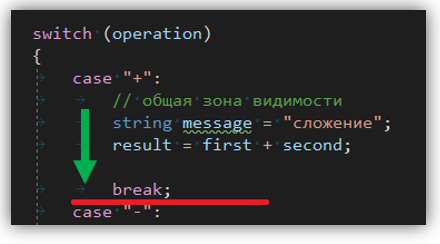
**Компьютер может понять куда более запутанную логику чем программист. Программа должна быть понятна в первую очередь программисту, а уже потом компьютеру.**

Представим ситуацию, когда мы хотим оперировать в ветвлении непосредственно значением переменной и выполнять действия в зависимости от её значений. Можно свести это к булевому условию и сделать конструкцию из состыкованных операторов if, но в C# есть специальный оператор для такого рода ветвлений.

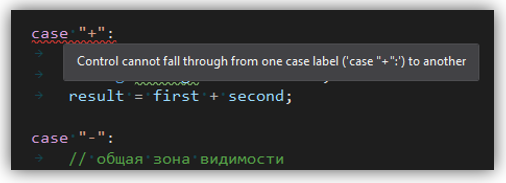
**switch** – ветвление в зависимости от значения переменной. Внутри себя он имеет множество кейсов, каждый из которых привязан к тому или иному значению переменной, указанной в скобках. Работает это так:

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

Оператор switch отличается от конструкции из состыкованных операторов if else тем, что switch создаст только один дочерний контекст. 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. В switch мы можем использовать только одно условие, в то время как пристыкованные if могут оперировать условиями, никак не зависящими друг от друга.Оператор switch отделяет один кейс от другого посредством ключевого слова break после чего выполнение кода внутри оператора switch прекращается.



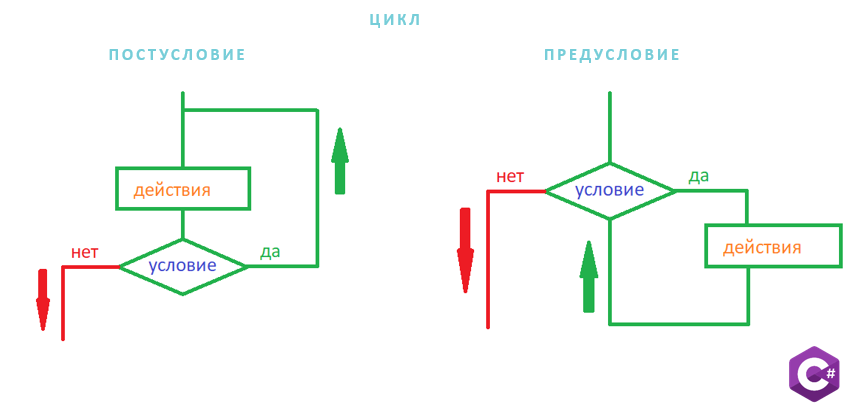
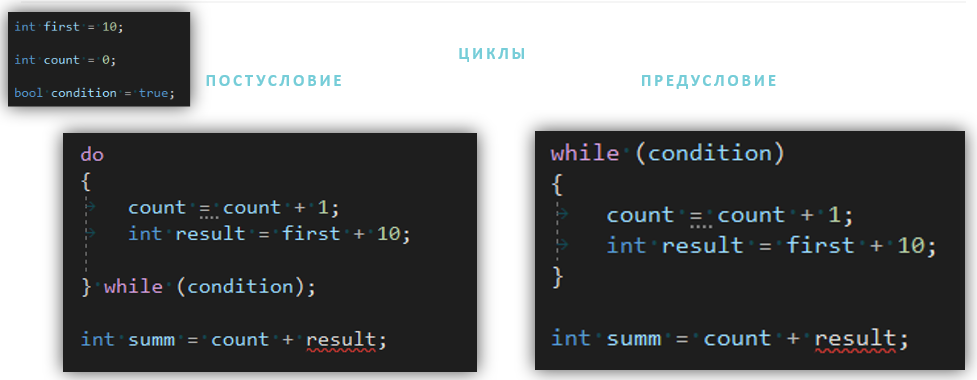
Для исключения недоразумений Visual Studio предупреждает нас о том, что не может отличить один кейс от другого. Без ключевого слова break

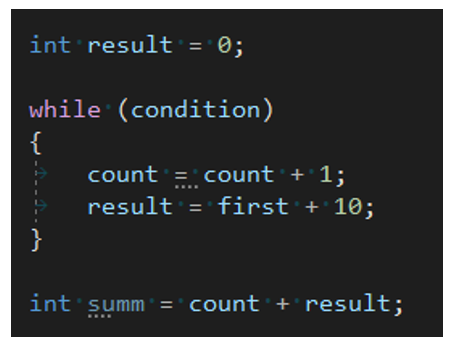


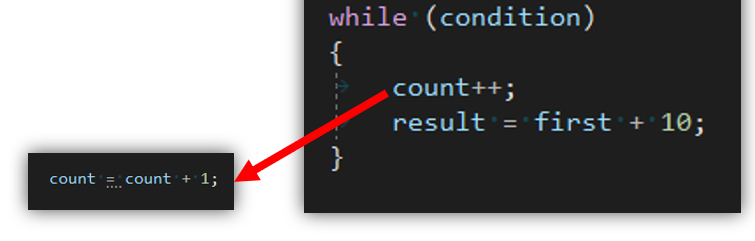
Цикл – это частный случай ветвления: **ветвление назад**, программа при выполнении условия возвращается к уже выполненному участку кода.

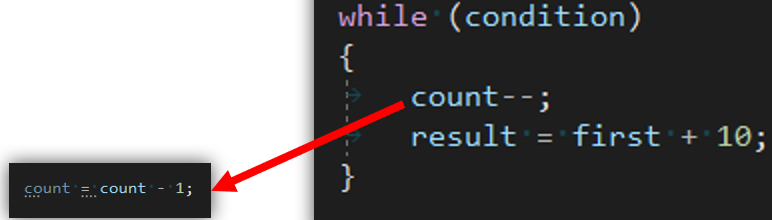
Здесь представлены две схемы организации циклов:

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

В C# есть конструкции для обоих случаев.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 выполнены 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.

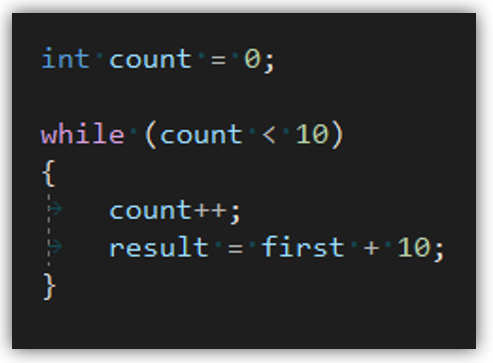


Обратите внимание на переменную **count**. Внутри циклов мы прибавляем 1 к её предыдущему значению и результатом этой суммы снова инициализируем count. В программировании это называется инкрементирование, в C# для этого есть специальный оператор:

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

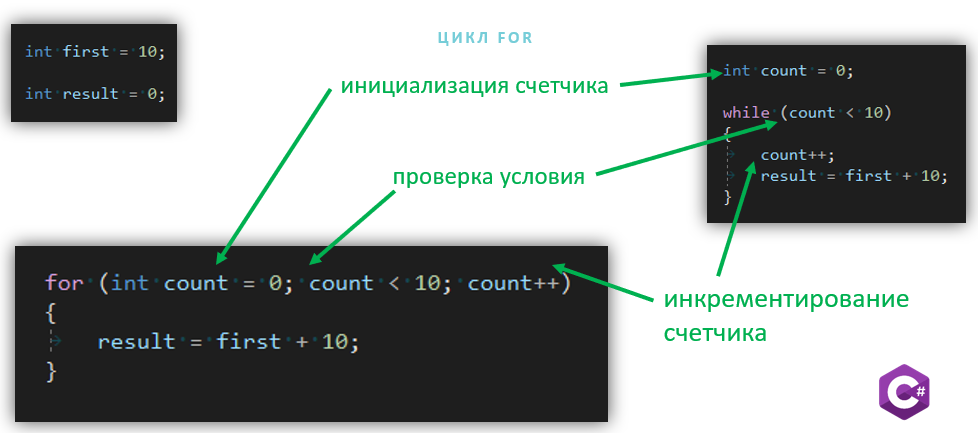
Для удобства есть и обратный оператор: **Декремент**, который уменьшает переданное в него значение на единицу.

Обратим внимание на условие повторения цикла, мы передавали в него переменную condition, заменим её на условие зависящие от значения **count.**



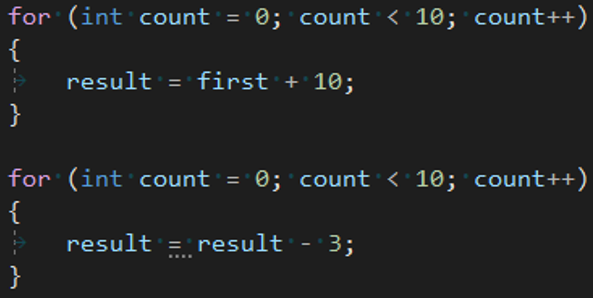
Цикл будет выполняться до тех пор, пока count меньше 10ти при этом каждую итерацию count будет прирастать на 1 изначально count проинициализирован 0 значит цикл будет выполняться 10-ть раз.

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

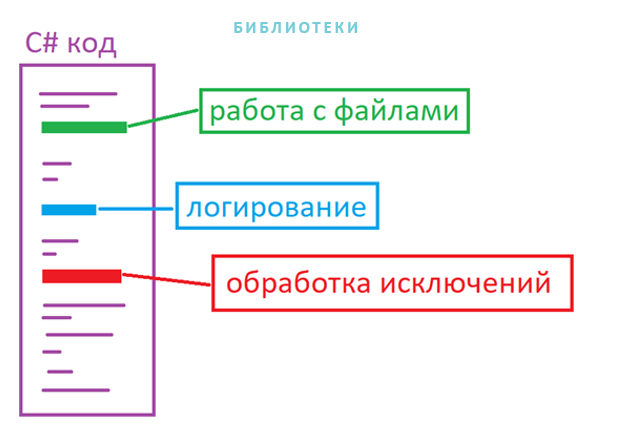


Он во всем аналогичен вот такому циклу while кроме одного: в конструкции while переменная count находится во внешнем контексте относительно цикла и может применяться за его рамками. В конструкции for переменная count объявляется во внутреннем контексте.

Что позволяет нам писать вот так:

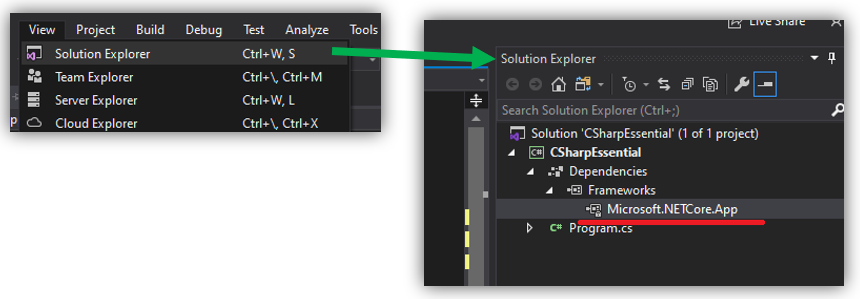


Без побочных эффектов. Цикл For объявляет и инициализирует счетчик во внутреннем контексте. Тем самым обеспечивая изоляцию счетчика от основного контекста выполнения программы.

Мы почти готовы писать нашу первую программу. Остался последний штрих: Интеграция с вводом выводом. Что бы наша программа умела принимать пользовательские данные и отдавать результат. Для этого нам необходимы внешние библиотеки.

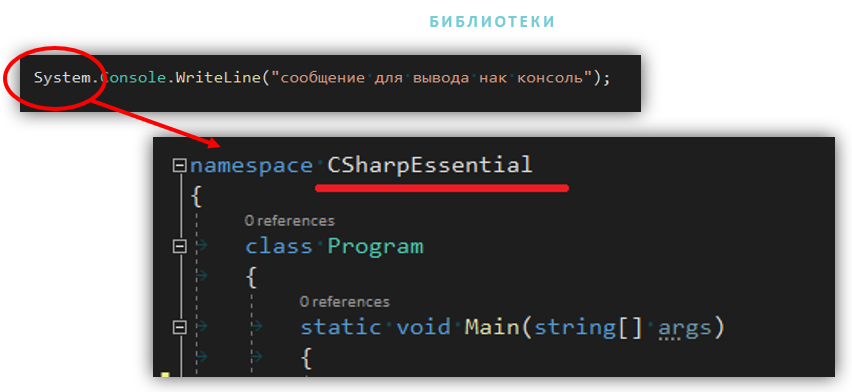
C# умеет взаимодействовать со множеством готовых библиотек концептуально они мало отличаются от библиотек C++. Они так же предоставляют нам набор компонентов, которые мы можем вызвать из написанного нами кода, передать туда необходимые параметры и получить ожидаемый результат. Некоторые библиотеки поставляются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

Откроем ранее созданное приложение и добавим 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 создания. Вот где это можно увидеть.

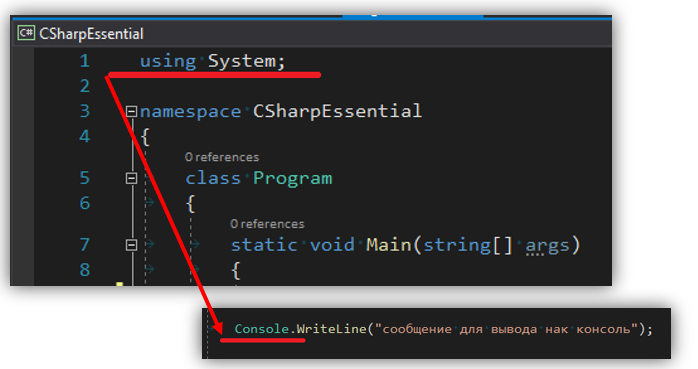
В меню View верхней панели visual studio выбираем окно solution explorer. Именно там надо искать его, если случайно закрыли. Он открывается с левой стороны: находим там наше приложение и раскрываем вкладку зависимости. 

В этой вкладке находятся 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 компонентов.

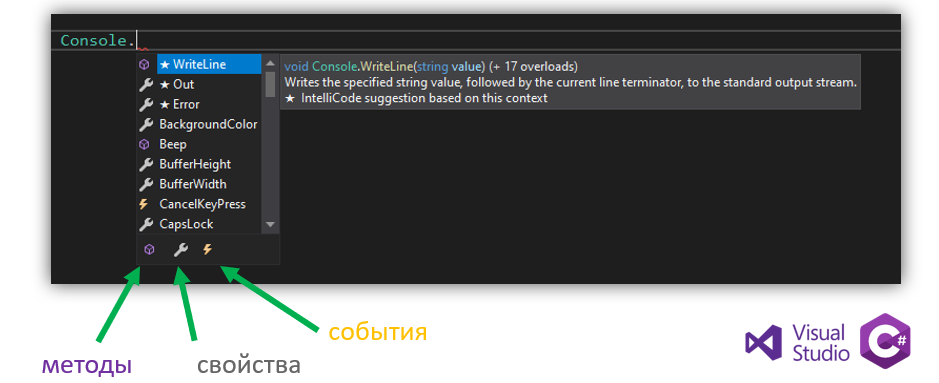
Нужный нам компонент находится в пространстве имен System в классе Console и называется WriteLine так и пишем: System.Console.WriteLine(“сообщение”) здесь System – это пространство имен, в котором находится статический класс Console у которого мы вызываем метод WriteLine. 

В нашем приложении тоже есть пространство имен.

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:

Visual Studio располагает механизмом подсказок. Для доступа к нему достаточно написать имя переменной (в данном случае Console) и поставить точку.

После чего откроется список, содержащий описание всех методов, свойств и событий доступных для обращения у объекта содержащегося в данной переменной.

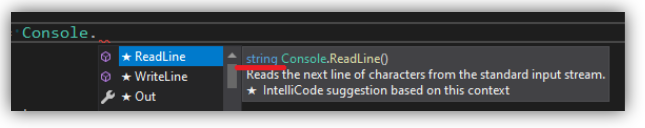
Более того, выбрав соответствующий компонент доступно его описание, список параметров и число перегрузок.Справочная информация, за которой не надо лезть в глубины MSDN.

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 избавлены от необходимости собственноручно связываться с консолью windows или linux для обработки выходного потока нашей программы.

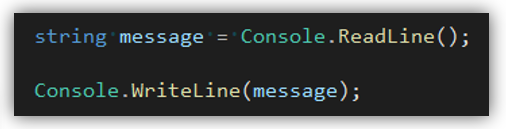
На примере работы данного механизма мы можем наблюдать **АБСТРАКЦИЮ** в действии: нам нет нужды знать, как именно работает Console.WriteLine нам достаточно знать, как им оперировать в рамках написанной нами программы.

Получить данные из консоли мы можем с помощью другого метода у объекта класса Console

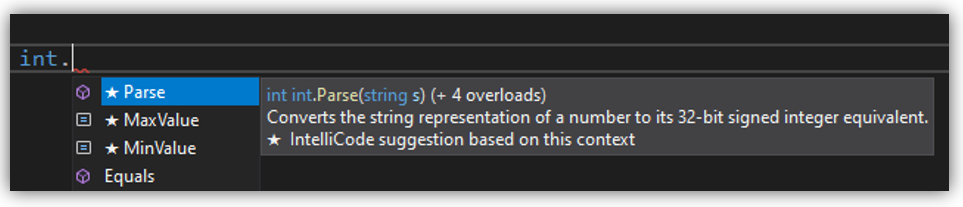


Console.ReadLine() имеет доступ к стандартному входному потоку, а значит на пару с Console.WriteLine они обеспечат нам связь с внешним миром. У метода Console.ReadLine() перед именем указан тип string, это тип строковых переменных. А у метода Console.WriteLine тип void – это означает, что данный метод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

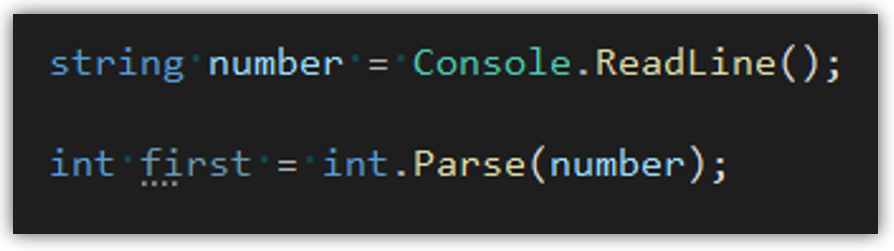
Таким образом мы можем проинициализировать переменную message значением, которое вернет Console.ReadLine() и передать её в метод Console.WriteLine



Теперь мы знаем, как передать строку для вывода на консоль и как получить строку из консоли. С# язык строго типизированный, поэтому строка 2 не равна числу 2 у них разные типы, а считать из входного потока мы можем только строку.

Таким образом, что бы получить от пользователя число, нам необходимо сначала получить строку, а затем конвертировать её значение в число. Для этих целей подойдет метод int.Parse 

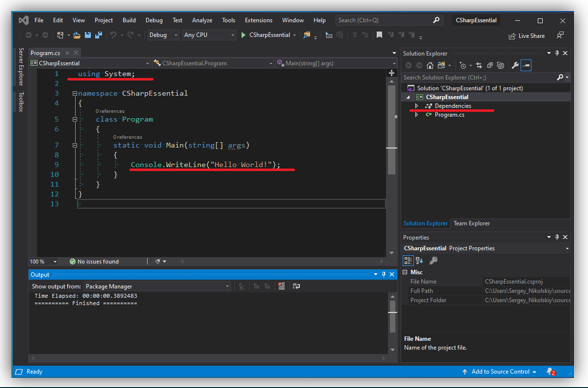
Метод int.Parse служит для конвертации строк в числовые объекты, по его описанию мы видим, что он принимает строку, а возвращает число.

Соберем следующую конструкцию: получили строку из консоли, и конвертировали её в число. Все будет в порядке для строк 2, 35 или 678. В случае если строку в число конвертировать не удастся речь пойдет позже.

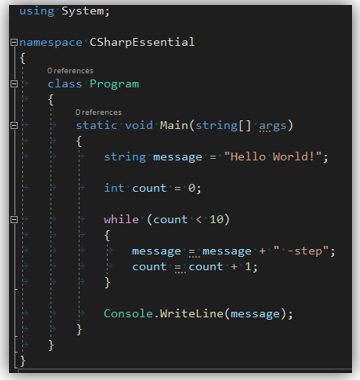
Вот ссылки на MSDN где вы сможете подробнее познакомиться со спецификацией C#.

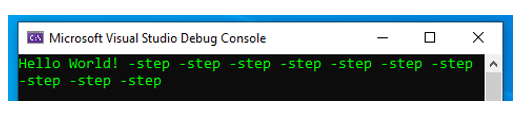
* [Введение в C# и платформу .NET](https://docs.microsoft.com/ru-ru/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework)
* [Обзор C#](https://docs.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/)
* [Руководство по программированию на C#](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/)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

Вернемся к Hello World который сгенерировала Visual Studio. Вот список зависимостей, вот объявлено использование пространства имен System, а вот обращение к объекту класса Console и вызов метода WriteLine в который передана строка привет мир.



Вынесем эту строку в отдельную переменную. Добавим счетчик и цикл.

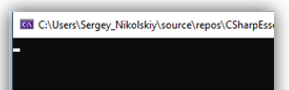


Запустим на выполнение:

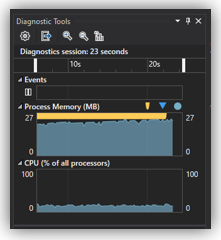
Условие count меньше 10 успешно выводит программу из цикла, присвоим это условие переменной.



Переменная condition проинициализирована результатом выполнения оператора сравнения и передаем в цикл. Запускаем.



На экране диагностики видно, что программа работает: память выделяется, процессор исправно выполняет инструкции, но консоль пустая. Что-то пошло не так.



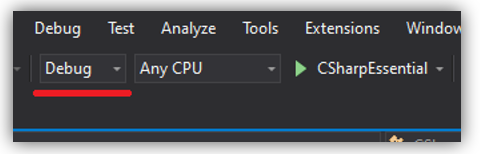
Судя по тому, что компилятор не ругается на нашу программу, все конструкции мы использовали верно. Значит дело в самой логике, необходимо посмотреть, что происходит во время выполнения программы.

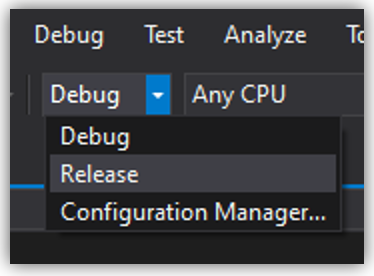
В этом помогут точки остановки. Левее нумерации строк расположена панель для точек остановки. Достаточно кликнуть мышкой на этой панели напротив той строчки кода, в которой мы можем разместить точку остановки, и она появится. Поставим их на линии 13 где мы объявили и проинициализировали переменную condition. И внутри цикла на строке 17



Помимо красной точки, Visual Studio выделила ту область кода, в которую мы попадем при остановке выполнения программы. А непосредственно в момент создания точки остановки, мы видим сообщение о том, где именно в программе находится проставляемая точка и получаем доступ к её настройке.

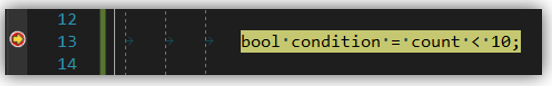
На верхней панели, рядом с зеленым треугольником, через который мы запускаем приложение, есть перечисление: Дебаг, релиз и менеджер конфигураций. Здесь мы можем выбрать тип сборки, который будет создан и запущен в момент нажатия зеленого треугольника.



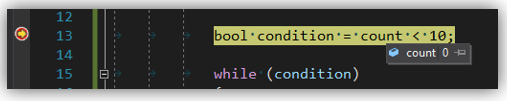


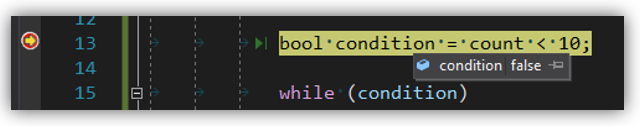
Подробнее о сборках поговорим в далее. Пока просто запомним, что в дебаг сборке точки останова сработают, а в релизной нет.

Выбираем дебаг и запускаем программу в отладочной сборке. Срабатывает первая точка останова.

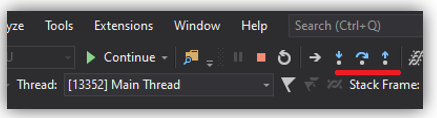


Наводим мышку на переменную count и видим её значение: 0.



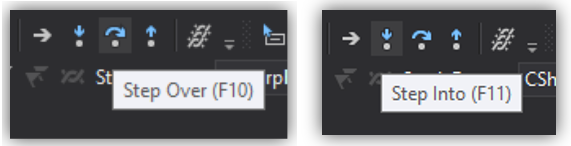
0 меньше 10 значит переменная condition должна быть проинициализирована значением true. Наведем на неё мышь и узнаем значение.

condition выдает false, но 0 явно меньше 10. Дело в том, что точка останова срабатывает до того, как отработал оператор присваивания. А по умолчанию значение для всех переменных булевого типа false. Двинемся на шаг вперед. Для этого снова глянем на верхнюю панель Visual Studio:

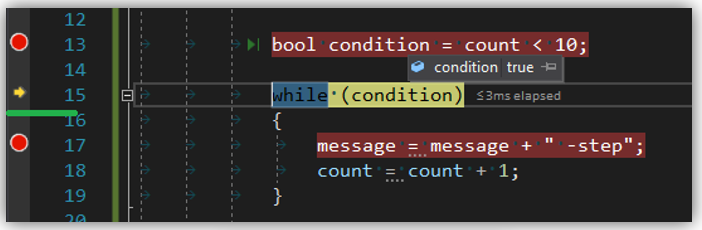


Зеленый треугольник остался на месте, но вместо имени приложения там теперь слово продолжить, нажатие на треугольник продолжит выполнение программы до следующей точки останова.

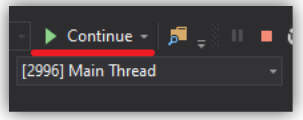
Обратим внимание на значки рядом:



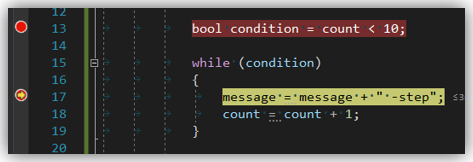
* F11 и мы попадаем внутрь исполняемого метода.
* F10 и мы переходим дальше по коду - **наш случай**.

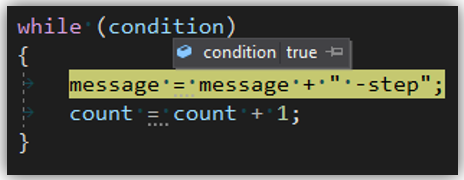
F10 и переходим к следующей строчке. Наводим мышь на condition и видим, что она приняла нужное нам значение. 

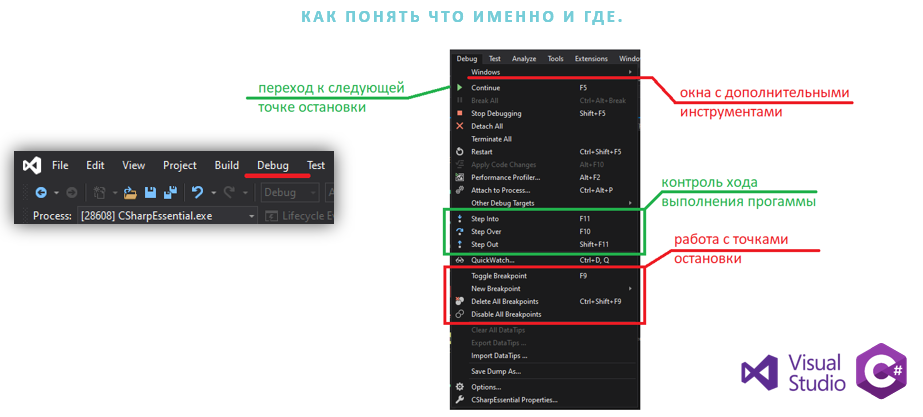
Нажмем продолжить (зелёный треугольник), для перехода к следующей точке остановки, которая находится в цикле.



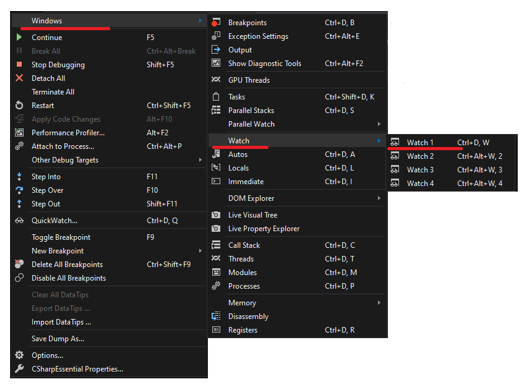
Выполнение программы остановилось на первой итерации цикла (там была размещена вторая точка останова). Нажав на продолжить ещё несколько раз, мы убеждаемся в том, что переменная condition не меняет свое значение от итерации к итерации.



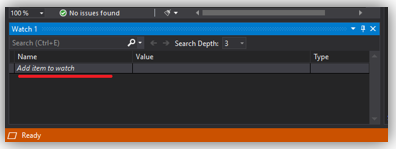


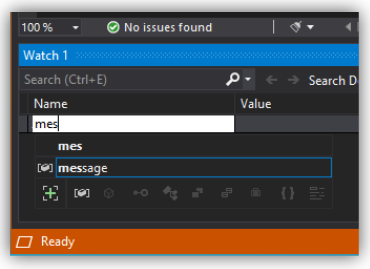
Здесь нужен инструмент для отслеживания значений переменных. В верхней панели находим вкладку Debug, мы видим уже знакомые нам иконки: зеленый треугольник продолжить, стоп для остановки выполнения программы. F11 F10 для контроля хода выполнения программы, выключение\удаление всех точек останова и, в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

Во всем перечисленном многообразии нас интересует вот это окошко, наблюдатель. Их можно открыть 4 штуки, откроем первый.

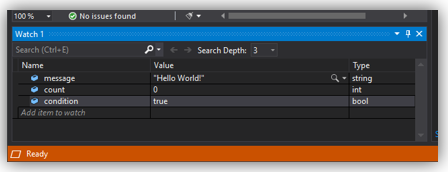


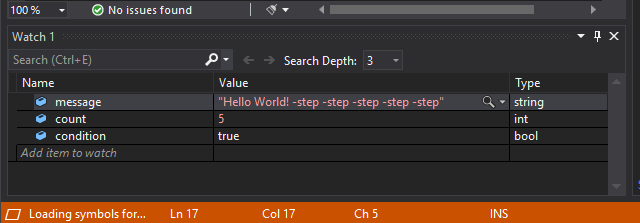
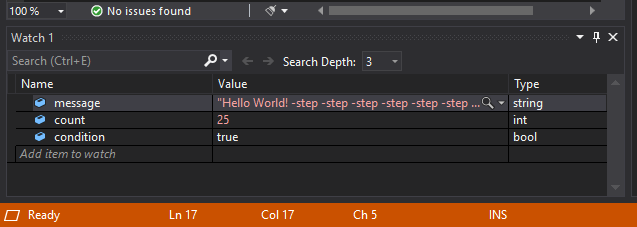
Внизу экрана появляется вот такое окошко.



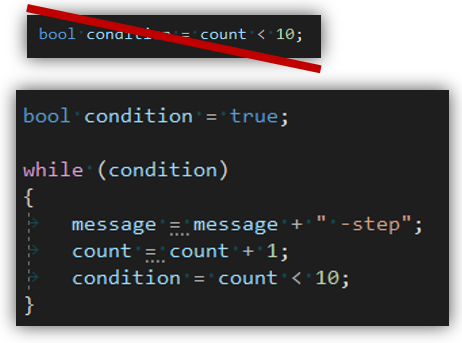
Для добавления переменной к просмотру в поле для ввода имени помещаем имя интересующей нас переменной. Как только мы начинаем вводить имя, Visual Studio выводит подсказку с доступными в данном контексте выполнения переменными.

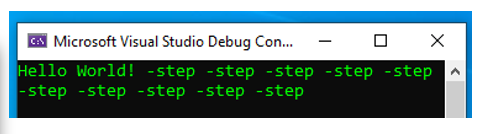
Добавим все три переменных в окно наблюдателя. Вот что мы имеем на начальном этапе.



А вот какие значения будут после пятой итерации циклаНа двадцать пятой. condition всё ещё равен true, программа вошла в бесконечный цикл.

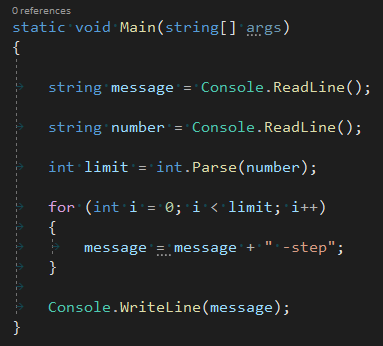
Проблема в инициализации переменную condition:Мы присвоили ей результат выполнения оператора сравнения, в который передали значение переменной count и сравнили его с 10. При инициализации программа считала значение count, оно было меньше 10 в момент инициализации, и, соответственно, condition был проинициализирован значением true. Дальнейшие изменения значения count никак не повлияли на переменную condition.

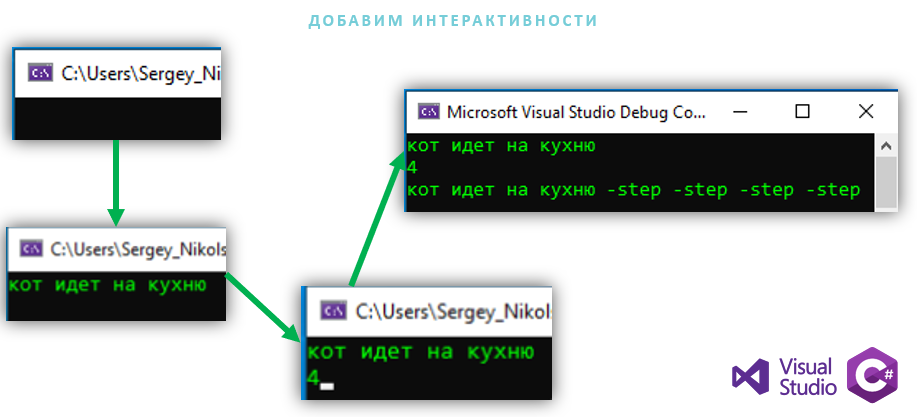
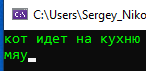


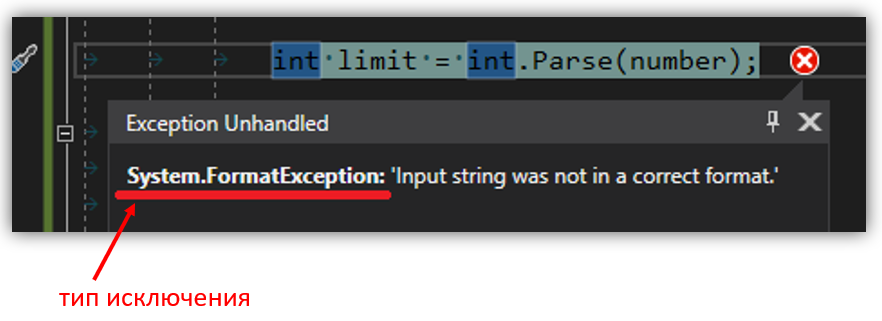
Вместо того что бы динамически менять значение condition каждую итерацию цикла, мы задали его один раз. Вносим изменения в код и запускаем программу.

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

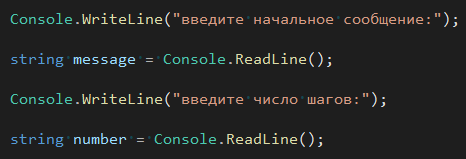
Пришло время добавить в программу конструкции, которые мы рассмотрели выше:



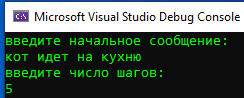
Запускаем, консоль пуста и ждет ввода сообщения, ведем его, нажмем enter, в этот момент метод Console.ReadLine считал входной поток из консоли windows и проинициализировал полученным сообщением переменную message, запустилось выполнение второго вызова метода Console.ReadLine который снова ждет от нас информацию: передаем 4 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 сделаем так:

Все будет хорошо ровно де тех пор, пока мы не попытаемся распарсить (задействовать метод Parse – это стандартное название для подобных конвертаций) “мяу” в число, ничего не получилось и метод int.Parse выдает нам ошибку:Ошибка ввода данных – это вполне конкретный объект, который имеет свой тип и называется исключением, это не ошибка в логике программы, как в предыдущем случае, когда мы попали в бесконечный цикл, это ошибка в обработке данных. Исключение, которое привело к остановке выполнения программы является специальным объектом, который, в свою очередь содержит информацию об ошибке.

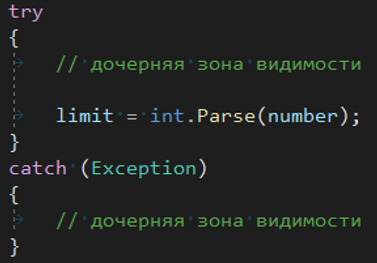
Если мы введем те данные, которые от нас ожидает программа – всё будет в порядке, то есть наша программа стала хрупкой, она неявно для пользователя зависит от того, какие данные в неё передают. Снизим хрупкость программы путем добавления пояснений.



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

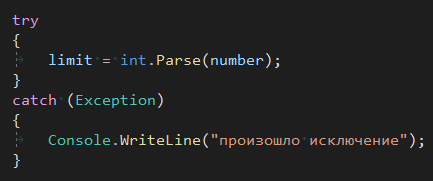
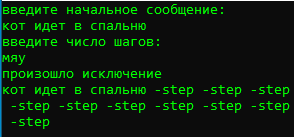


Для дальнейшего снижения хрупкости программы обработаем объект исключения, что бы он не ломал логику, а органично в неё вписывался. Этой цели служит конструкция try catch



Как и в случае операторов ветвления и циклов она образует дочерний контекст вызова, каждый раз, когда мы видим фигурные скобки в операторе – у нас образуется дочерний контекст вызова. Поэтому переменную limit которую мы собираемся использовать в дальнейшем мы проинициализируем значением по умолчанию,

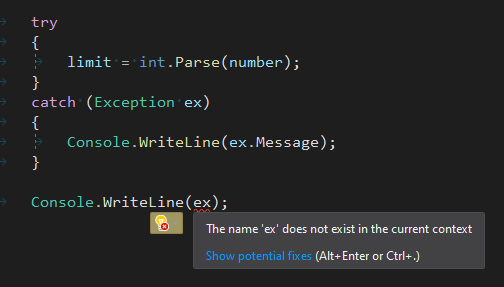


которое будет ей присвоено в случае, если Parse не будет выполнен корректно. Вот итоговая реализация: Запускаем нашу программу. И снова “мяу” вместо числа шагов

Получаем те самые 10 шагов, которые соответствуют изначальному значению переменной limit.

**try catch** работает так: внутрь оператора try помещается код при выполнении которого возможно появление исключения, и если таковое случилось – срабатывает код внутри оператора catch.

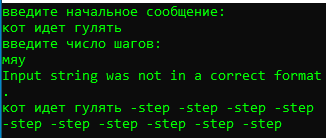
В случае если всё хорошо и переданная строка парсится в число – программа выполняется как обычно. В случае же “мяу” выполняется код обработки исключения. Выше фигурировал объект исключения, а в catch явно что-то есть в круглых скобках.



Это то самое место, где мы можем получить доступ к самому объекту исключения, для этого надо к объявлению типа Exception добавить название переменной, которая будет проинициализирована в блоке catch и может быть обработана в его дочернем контексте.

Зона видимости переменной ex не выходить за границы catch У объекта исключения, который теперь лежит в переменной ex. Так же, у объекта исключения есть свойство Message содержащее текстовую информацию, которую можно вывести в консоль.

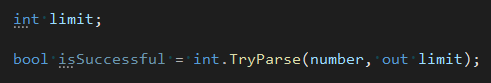
Запустим программу, введем некорректные данные, поймаем объект исключения и обработаем его. Как видите внутри исключения есть легко читаемая информация, которая понятна пользователю.

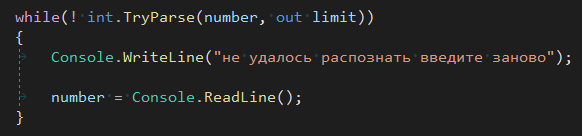


Программа стала куда устойчивее, когда мы добавили обработку исключения.

Теперь дадим пользователю возможность ошибаться и вводить данные снова. Для этого можно использовать метод парсинга который не приводит к исключению, когда в него попадает неправильное значение он называется Int.tryParse

В этот метод передается два параметра, первым из которых является исходная строка, а вот вторым инициализируемая методом переменная. Слово out в данном случае гарантирует нам, что в ходе выполнения данного метода переменная будет проинициализирована. А возвращает метод значение типа bool.

Проинициализируем переменную isSuccessful значением, которое вернет метод int.tryParse и убререм инициализацию у переменной limit (так как ключевое слово out гарантирует инициализацию). Теперь в случае, если пользователь ввел что-то что нельзя превратить в число на этапе парсинга, перменной limit будет присвоен 0.

В отличии от работы с исключениями 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 получено. В результате программа будет просить ввести корректное число до тех пор. Пока пользователь этого не сделает. Подобного поведения можно было бы добиться и с помощью объекта исключения – но это не очень хорошая практика.

**Исключения нужны для исключительных случаев.**

Рассмотрим ситуацию, когда нам нужно написать приложение, которое гораздо больше 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, мы будем писать систему отвечающую за выдачу денег через банкомат. Или робота, который будет рассылать оповещения.

Делать нечто такое, что нельзя описать в рамках одной инструкции. Или для чего нужно будет написать такую большую и сложную инструкцию, что при малейших исправлениях мы получим трудно уловимую ошибку.

В написании по настоящему сложных приложений нам поможет следование объектноориентированной парадигме программирования (ООП), она позволяет нам строить достаточно гибкие и легко масштабируемые приложения, содержащие при этом сложную логику.

**Парадигма - подход к решению задачи.**

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

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 определяются их классами (следует из вышенаписанного), то для понимания ООП необходимо рассмотреть понятие класс.

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

О реализации классов 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 единообразно в пределах одного класса. Какие же сейчас выделяют основные концепции в ООП?

A picture containing text, book

Description automatically generated

**Инкапсуляция** - свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе и скрыть детали  
реализации 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** - 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: 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.

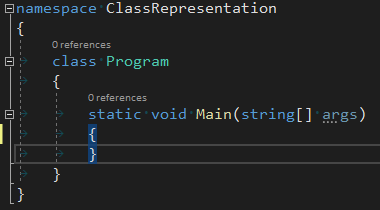
A picture containing game, light

Description automatically generated

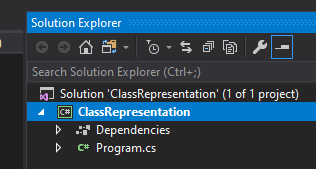
ООП достаточно прост в освоении, поскольку сознание человека в общем-то объектно-ориентированно. Основные концепции всех языков на земле при построении предложений – подлежащее и сказуемое, то есть существительное (данные) и глагол (действие с этими данными).

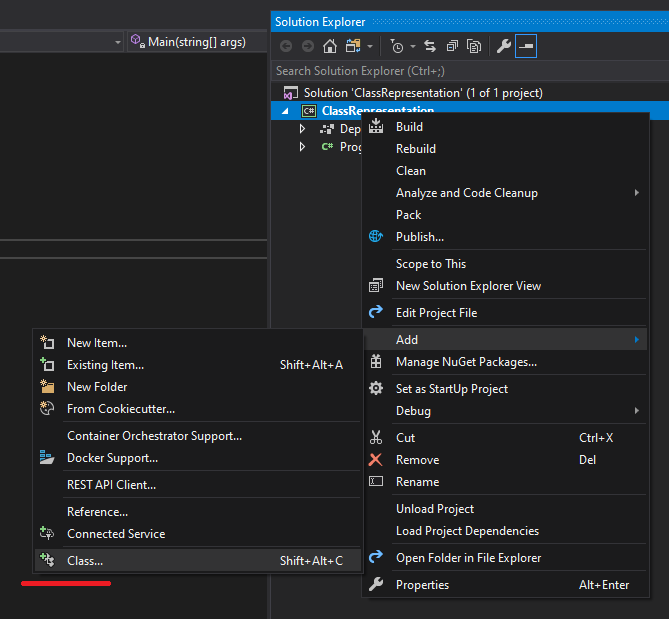
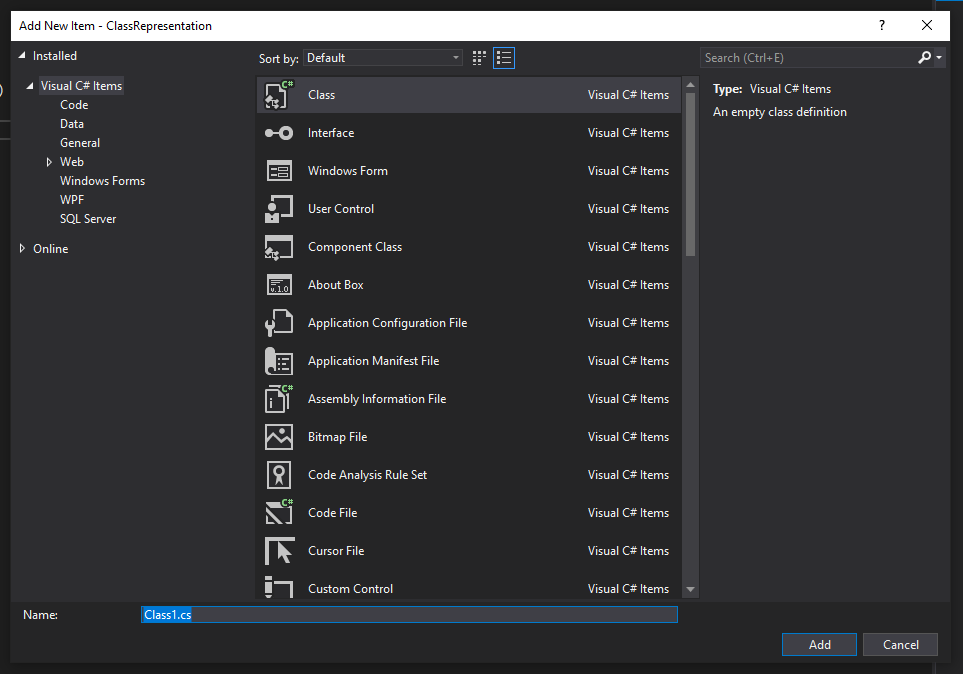
1. При решении задачи методами ООП мы выделяем какие-то важные данные, задаем им желаемое поведение и объединяем в классы. Для которых мы выстраиваем иерархию. Например мы создаем класс студента содержащий имя, возраст, группу, факультет, массив оценок - данные. И методы: ходить на занятия, делать задания и сдавать экзамены - действия. У нас получится универсальный чертеж для описания любого студента в нашей бизнес модели.
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.

Перейдем к реализации классов и их объектов в C#. Создадим новое консольное приложение ClassRepresentation, и сотрем все внутри метода Main вот что мы увидим:

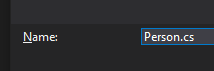


Создадим наш первый класс, откроем solution explorer

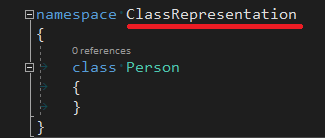


Кликнем правой кнопкой мыши на имени приложения и выберем add -> classПо клику открывается окно для добавления новых элементов в приложение:В нем сразу выбран элемент класс и visual studio предлагает нам его имя, меняем имя на Person.

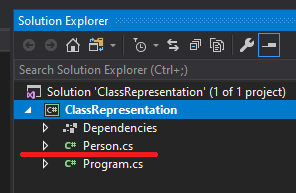
Обратите внимание: вновь созданный класс имеет расширение **.cs** которое однозначно нам сообщает, что это элемент написанный на C#, при переименовании следим что бы не стереть расширение.



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



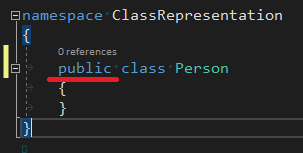
Созданный нами класс имеет то же самое пространство имен, что и основная программа, более того, в нем точно так же фигурирует ключевое слово class, а в solution explorer он отображается вот так:



Имя файла полностью совпадает с именем созданного нами класса, данное поведение является общепринятым, никто не мешает нам внутри одного файла на создавать множество классов, поменять имя файла на отличное от имени 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.

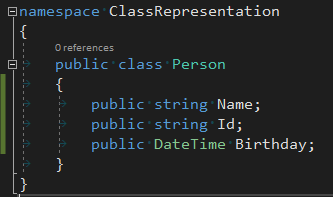
Модификаторы доступа есть у всех программных элементов: у классов, полей, констант, методов. Благодаря модификаторам доступа мы можем инкапсулировать данные – объединять их в значимые сущности и настраивать доступ к ним.

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, начнем с данных:

Объект Person должен обладать:

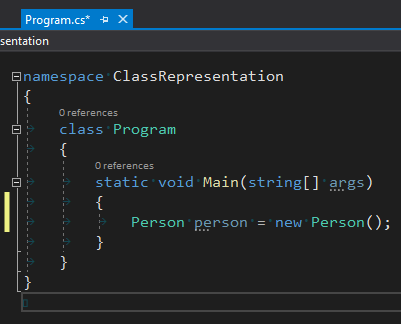
* Идентификационным номером.
* Именем
* Датой рождения

Добавим в класс соответствующие поля. 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 точка с запятой, которая завершает декларацию поля.



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

Класс Person наполнился смыслом, создадим 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 напишем вот такую конструкцию:



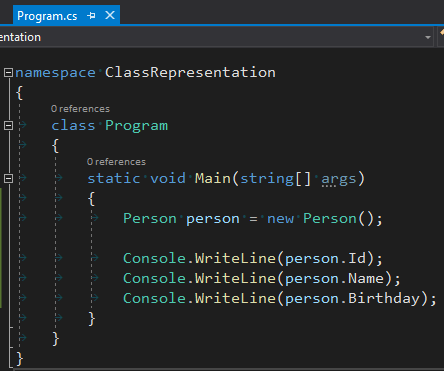
Мы объявили переменную с именем person класса Person и проинициализировали её путём вызова конструктора по умолчанию для класса Person.

Но в классе Person были только три поля для хранения данных и все.

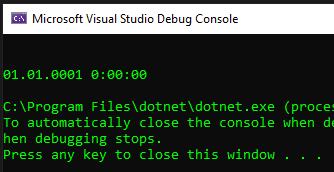
Дело в том, что каждый класс в C# по умолчанию содержит специальное действие, которое создаёт объект этого класса и называется это действие конструктор.

Если мы не написали в классе конструктор явно, то он будет создан без нашего участия, 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:

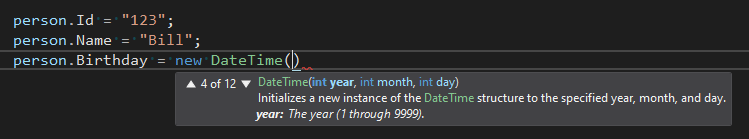


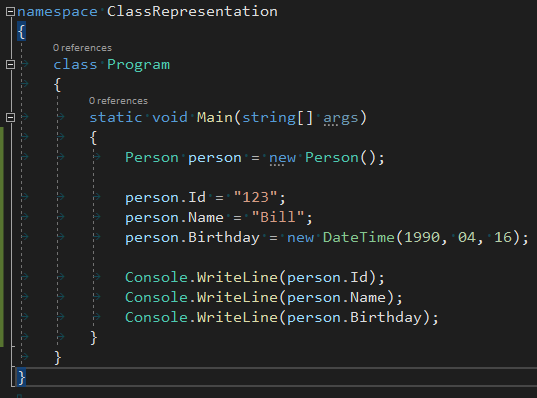
И запустим приложение:



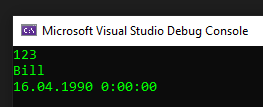
Приложение отработало без ошибок и вернуло нам две пустых строки (значения string по умолчанию) для Id и Name соответственно, и значение по умолчанию типа DateTime для поля Birthday.

**Конструктор по умолчанию для класса Person создал объект типа Person и пронициализировал его поля значениями по умолчанию.**

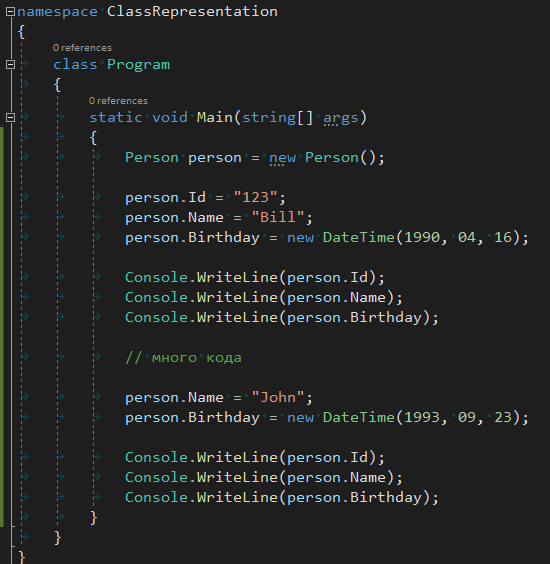
Обратимся к полям объекта person и запишем в них значения. При инициализации переменной Birthday я обращаюсь к конструктору типа DateTime выбираю его четвертую перегрузку (о перегрузках конструктора и как их писать поговорим ниже) которая принимает на вход три значения целочисленного типа: год, месяц и дату.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 объекта person в его текущем исполнении – без проблем:





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.

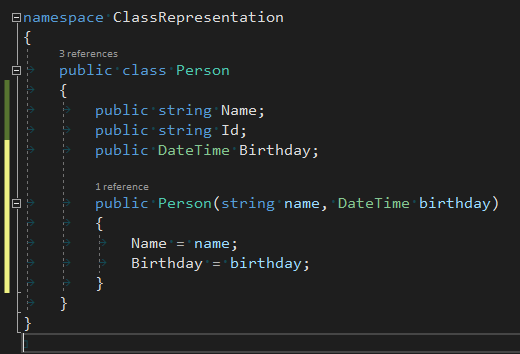
Давайте признаем такое поведение нежелательным и, попробуем, каким-то образом защитить данные Birthday и Name в person от модификации после создания объекта.

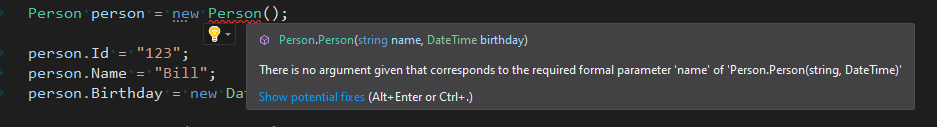
Было бы идеально сделать так что бы имя и дату рождения можно было задать в момент создания объекта. Напишем такой конструктор в классе Person (всё поведение объекта person задается в его классе, а значит и меры надо предпринимать там) который бы ещё на моменте создания определял имя и дату рождения.

Снова открываем solution explorer и переходим на файл Person.cs в котором мы прописывали класс Person.

Класс Person уже содержит конструктор по умолчанию, иначе мы бы не смогли создать объект этого класса, так что нам нужно написать ещё один конструктор для Person.

**В C# класс может содержать сколько угодно конструкторов, пока они отличаются параметрами.**

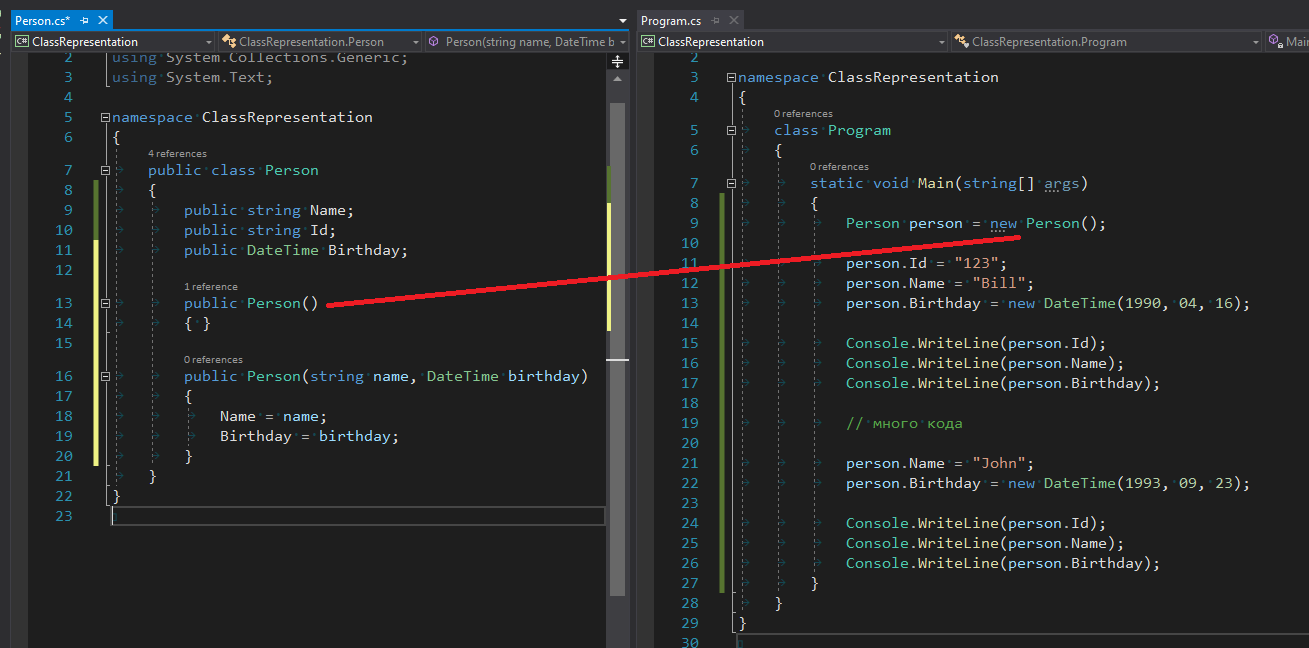
Объявим модификатор доступа, так как мы собираемся вызывать конструктор извне, то это снова будет public. Затем напишем имя класса, в нашем случае это Person откроем круглые скобочки и перечислим через запятую необходимые нам параметры с их типами string name, DateTime birthday. Обратите внимание, параметры 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 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.

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

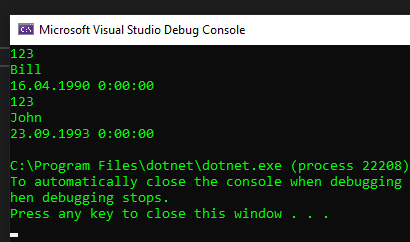
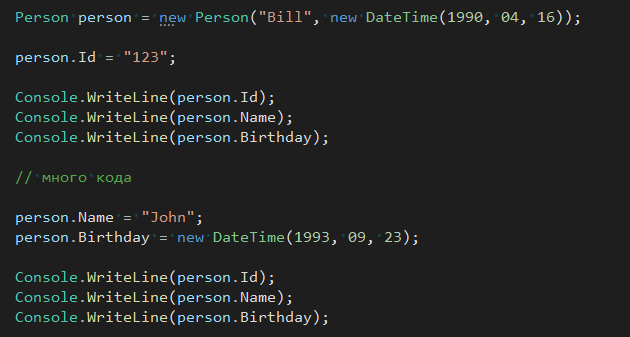
Дело в том, что как только мы написали руками в классе конструктор, C# решил, что конструктор по умолчанию для нашего класса больше не нужен (мало-ли мы не хотим иметь возможность создавать пустые объекты класса Person) и убрал к нему доступ.

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.

Вернемся в класс Person и просто напишем конструктор без параметров. Снова начинаем с модификатора доступа и снова он у нас public, затем снова пишем имя класса и круглые скобки, на этот раз в фигурных скобках не будет никаких действий, значения по умолчанию характерные для их типов полям присвоятся сами, как видим ошибка в Program.cs исчезла!

Компилятор снова использует конструктор без параметров, но на этот раз это не конструктор по умолчанию 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 значения и убедитесь, что данные сохраняются:

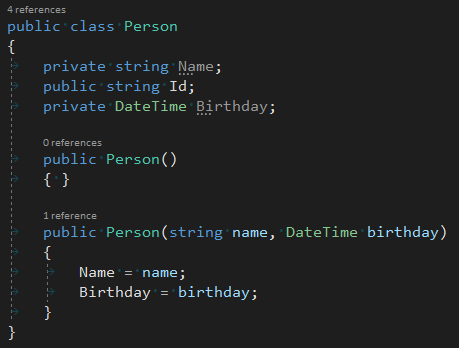
Но вернемся к нашей проблеме: поля Name и Birthday позволяют менять свои значения во время жизни объекта.

Начнем с использования в Program конструктора с параметрами

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.

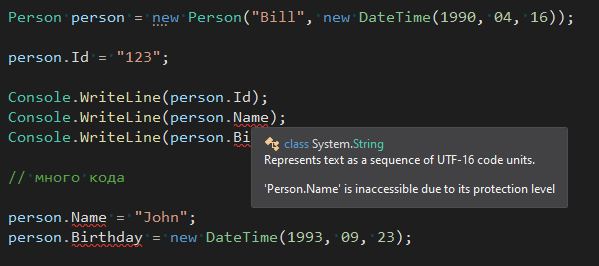
Необходимо изолировать поля Name и Birthday таким образом, что бы их значение можно было изменять только внутри класса Person, но при этом они были бы доступны для чтения снаружи.

Очевидным решением является изменить модификатор доступа у этих поле. Вернемся в Person.cs и поменяем его с public на **private**



Данный модификатор доступа позволяет оперировать полями внутри класса, компилятор ни на что не ругается.

Вот только в Program.cs 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.

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 содержащуюся в них информацию.

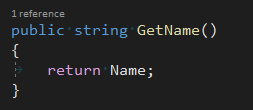
Чтение данных и запись данных – это действия с данными, мы можем снабдить класс Person специальными методами (действиями) которые позволят нам получать данные из полей Name и Birthday и не позволят туда ничего записывать.

И вот уже эти самые действия мы снабдим модификаторами 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.

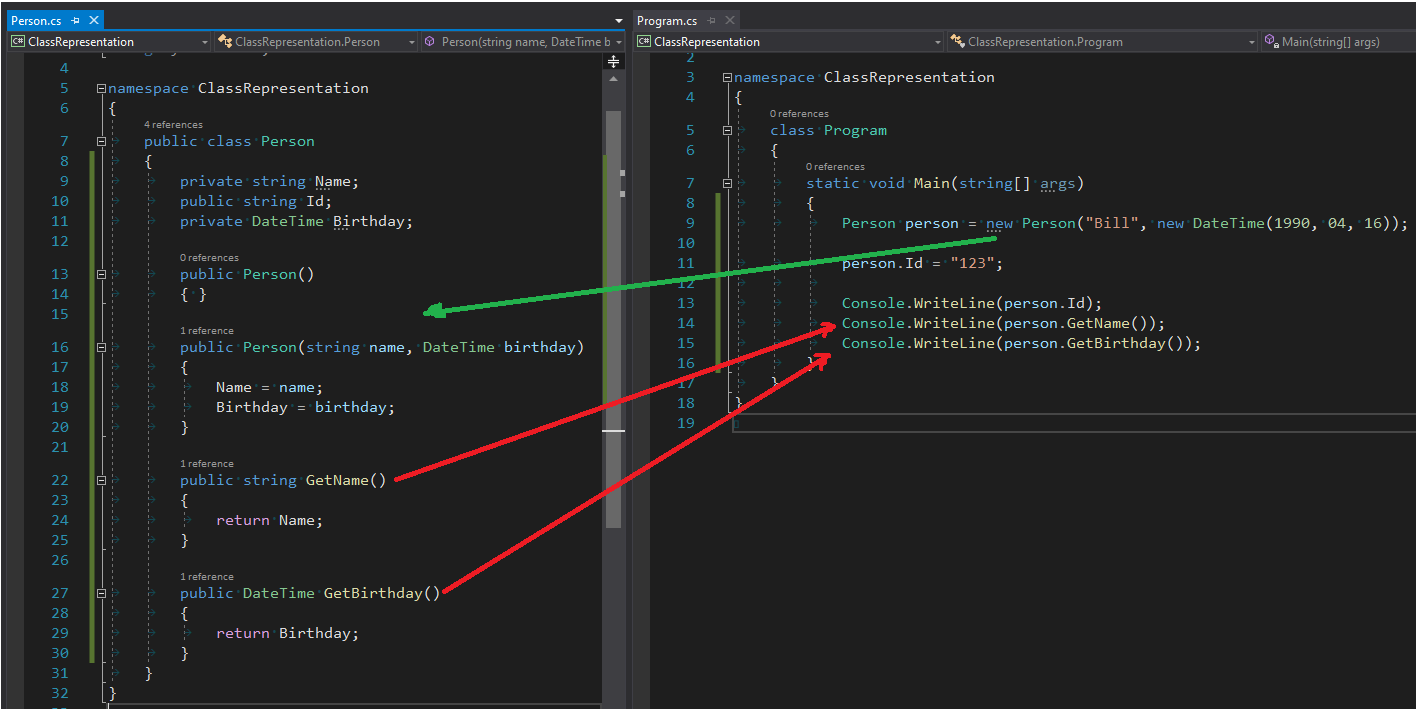
Перейдем в Person.cs и напишем методы чтения данных, начнем с Name.

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 рассказывать о том, что он делает, поэтому мы назовем его GetString а параметров нам не надо, 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:



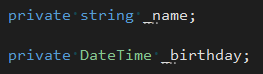
Метод для получения значения поля Birthday будет написан аналогично.

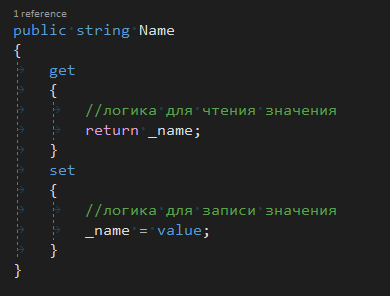
Теперь в Program.cs для получения значений Name и Birthday мы будем обращаться не к полям, которые у нас стали private, а к методам, которые значения этих полей возвращают.

Мы принимаем данные в конструкторе, на этапе создания объекта, а получаем их значения через специальные публичные методы. Таким образом данными можно свободно оперировать внутри 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, иными словами: инкапсулировали эти данные.

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, напишем свойство для поля Name.

Переименуем Name, в соответствии с общепринятыми практиками поля с модификатором доступа private начинаются с нижнего подчеркивания и пишутся с маленькой буквы. Перепишем поля Name и Birthday

Конструкция любого свойства строиться следующим образом:

При этом get и set это обычные методы! Предназначение 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#.

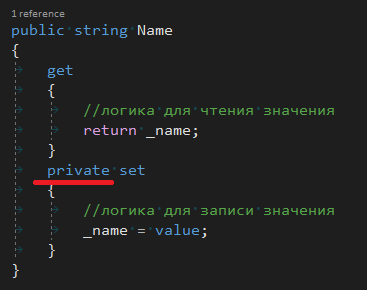
Важно помнить: set не может возвращать значение, а get всегда должен возвращать значение того же типа, что и тип свойства в котором он объявлен.

Обратите внимание на особенную переменную **value** которую студия выделила синим что бы подчеркнуть её статус. Переменная value существует только в контексте 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 возвращать 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, по умолчанию C# назначает методам get и set такие же модификаторы доступа как и у самого свойства. В нашем случае мы и так имеем public set и public get, сделаем set private



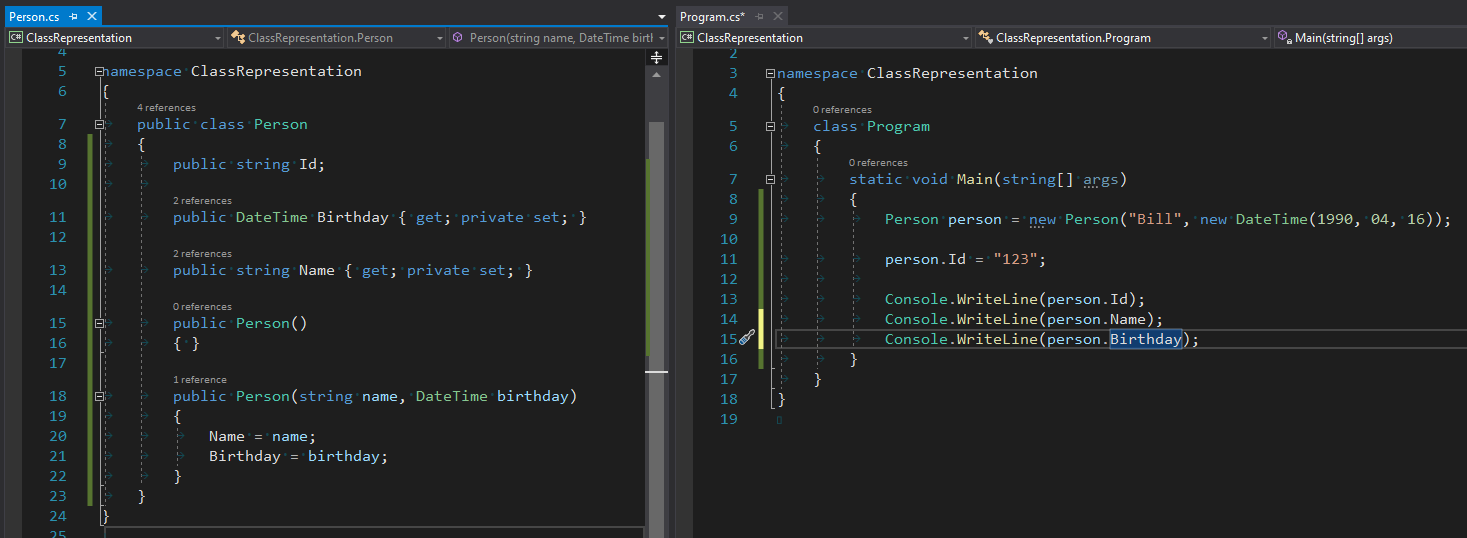
Довольно громоздко не правда ли? Если бы мы написали сложную логику валидации значений пригодных для Name в set методе. Или получали значение Name сложным образом в get методе, то такая громоздкость имела бы смысл, нам же надо только хранить данные и не позволять их менять извне.

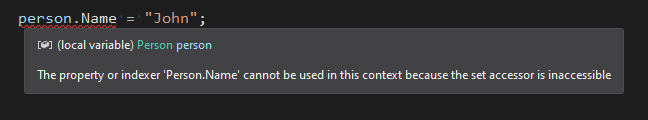
Если никакой сложной логики в set и get не подразумевается, то свойство можно записать сокращенно, вот так:



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.

Теперь можно избавится от специализированных методов на получение данных о дне рождения и имени, а заодно и от полей для их хранения. Сточки 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:



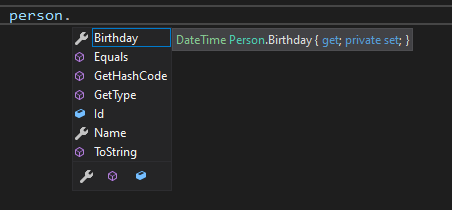
Первый принцип ООП в действии: Класс ограничивает доступ к данным осуществляя тем самым их **инкапсуляцию.** Так же стоит помнить, что само объединение данных в логическую сущность (объект) так же является **инкапсуляцией.**

Вот полезные ссылки 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)

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

Перейдем в Program.cs и посмотрим какие элементы класса Person у этой переменной.

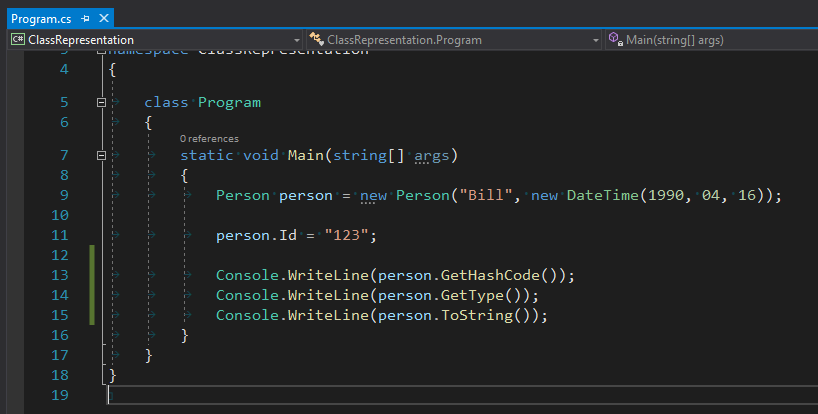


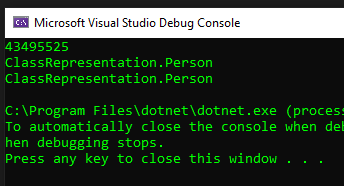
Присутствуют свойства Name и Birthday, поле Id и четыре метода. Тип каждого элемента понятен из значка рядом с ним. Выше мы это уже разбирали. Свойства и поле мы написали сами, но есть ещё четыре метода.

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 Object и доступны из любого его потомка. Вот мы и столкнулись с первым проявлением наследования: публичные элементы предка доступны у его потомка.

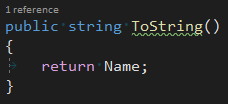
Давайте подробно рассмотрим некоторые из методов доставшихся в нам в наследство от Object. Для этого в тело программы в файле Program.cs мы добавим их вызов и выведем результаты в консоль.



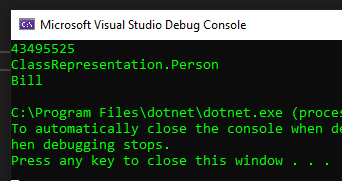
Первый метод вернул нам целое число (оно называется Has кодом и пока нас не интересует), два других вывели нам полное имя класса, состоящее из пространства имени и собственно имени.

И вот тут давайте начнем знакомство с полиморфизмом и поменяем поведение для метода ToString, метод Equals пока не трогаем. В классе Person пишем свой метод ToString().

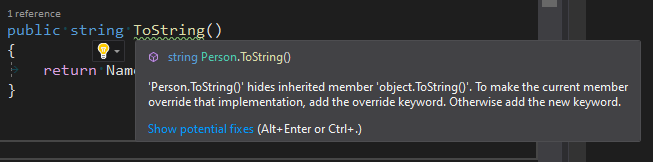
Модификатор доступа у него будет public, возвращать он будет тип string и называться ToString, а входных параметров у него не будет, в теле метода пока просто вернем значение свойства Name



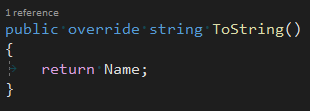
Как видим ошибки нет, хотя метод с точно таким же именем у нас уже есть в классе предке! Запустим код:

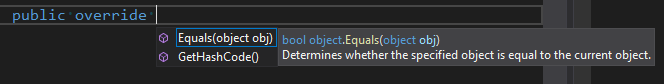


Был вызван метод потомка, получается, что метод потомка перекрыл собой метод предка. Такое перекрытие называется **переопределением** и если мы внимательно посмотрим на метод ToString в классе Person, то мы увидим. Что visual studio подчеркивает его зелёным, то есть у неё есть к нам какое-то предложение,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

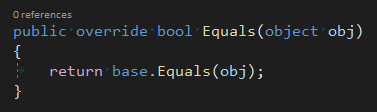


Visual studio нам сообщает, что метод с таким именем уже определен в классе предке и наше текущее определение будет перекрывать таковое у предка. Так же нам предлагают воспользоваться ключевым словом override что бы однозначно определить наши намерения по **переопределению** данного метода. Последуем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



Пойдем дальше и переопределим метод Equals, данный метод отвечает за сравнение объектов одного типа. В этот раз будем сразу декларировать намерение по переопределению, поэтому при написании нового метода сразу напишем override после модификатора доступа (public как обычно). 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 почти всё знакомо. Кроме типа входного параметра **object** и ключевого слова **base**, давайте разберем их подробно.

Тип **object** как раз и представляет собой тот самый 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 основано оно как раз 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.

**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 подругому, не так как он это делал у предка! Да и в этом-то как раз 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 экземпляру класса предка и его реализации публичных элементов, 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.

Visual studio написал за нас все обязательные компоненты для переопределения Equals и при этом ничего не сломал. Пользуясь предоставленной заготовкой, напишем свою реализацию метода Equals для класса Person. Для этого 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 изменим его инициализацию по умолчанию в классе Person:



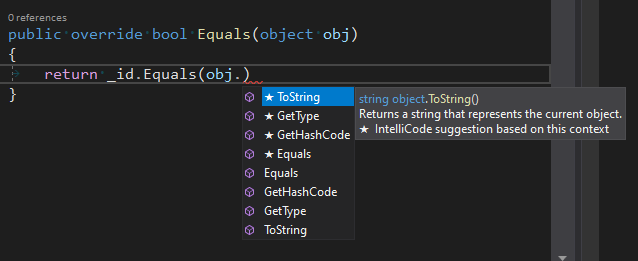
Мы заменили модификатор доступа с public на private и добавили инициализацию, которая сработает при создании экземпляра класса Person ещё до того, как будет вызван конструктор.

Рассмотрим подробнее запись, с помощью которой мы произвели инициализацию:

* Первым идет слово Guid – это обращение к статическому классу Guid (если быть точным, то это структура, но мы пока можем 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 таким 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 получать значение для поля \_id.

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 использовать значение поля \_id для сравнения текущего экземпляра класса Person с другим экземпляром в методе Equals. 

У значения переменной obj нет поля \_id так как она представляет собой переменную класса Object, который является предком всех классов в C# и ничего не знает своих потомках.

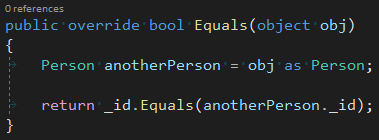
Ещё одно важное свойство наследования в C# - предки ничего не знают о своих потомках, например в сложных приложениях нет никакой гарантии, что от вашего публичного класса не отнаследуются, после чего не добавят в наследнике какой-нибудь метод. Но это никак не изменит поведение вашего кода, так как экземпляр вашего класса ничего не будет знать о дополнительных элементах, появившихся в его потомках. Куда опаснее ситуация, когда некий метод вашего класса переопределят в потомке, а затем (благодаря ковариантности) не передадут во внешний обработчик экземпляр потомка с новым поведением. Но вернемся к сравнению объектов класса Person

Прежде чем делать сравнение объектов по значениям полей \_id нам необходимо привести переменную 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 (как неоднократно говорилось выше), но он сам не редуцируется до предка, он просто оборачивается в него как в обертку.

Если мы передадим в 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 классов в их предков достаточно стандартная процедура, то в C# есть для разворачивания таких обёрток (распаковки) стандартная инструкция



Ключевое слово **as** позволяет попробовать распаковать obj в класс Person.

Внутри класса Person мы имеем доступ к приватному полю \_id **другого экземпляра,** а не только того, которому принадлежит метод Equals, так как

**в C# инкапсуляция осуществляется на уровне классов, а не их объектов.**

Но вернемся к распаковке, 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)

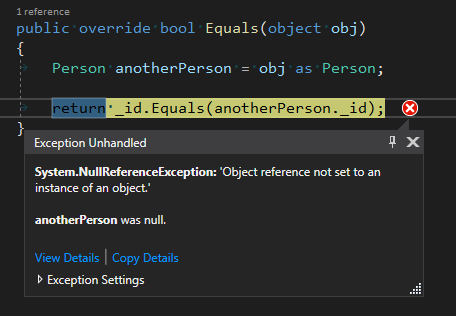
А если распаковка закончиться неудачей? С точки зрения внешнего кода мы же не знаем как организована работа метода Equals.

В случае если мы передадим в метод что-то, что нельзя распаковать в Person инструкция as не сможет проинициализировать переменную anotherPerson и в тот момент, когда мы попробуем обратиться к её полю \_id мы свалимся с исключением.

Давайте убедимся в этом, в файл Program.cs добавим вот такой код:

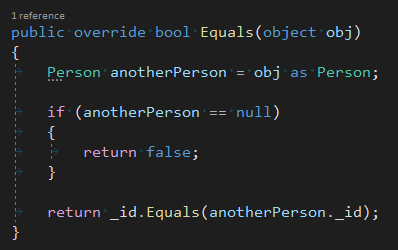


Формально всё в порядке, экземпляр класса string вполне можно упаковать в Object, а значит и передать в метод Equals экземпляра класса Person. Но в момент выполнения мы получим вот такую ошибку:



А произойдет она ровно потому, что инструкция as не смогла распаковать полученный объект в объект класса Person и переменная anotherPerson осталась не проинициализирована.

Дополним код метода Equals ещё одной проверкой, что бы исключить подобные ситуации:

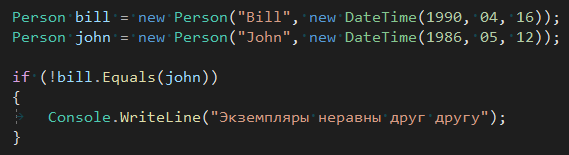


Как видите, не проинициализированная переменная будет содержать значение null в случае если в метод Equals был передан объект не того класса, который мы ожидали внутри инструкции as.

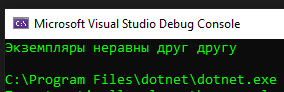
Воспользуемся этим и вернем false в этом случае. Как мы видим метод может содержать более одной инструкции return если это необходимо.

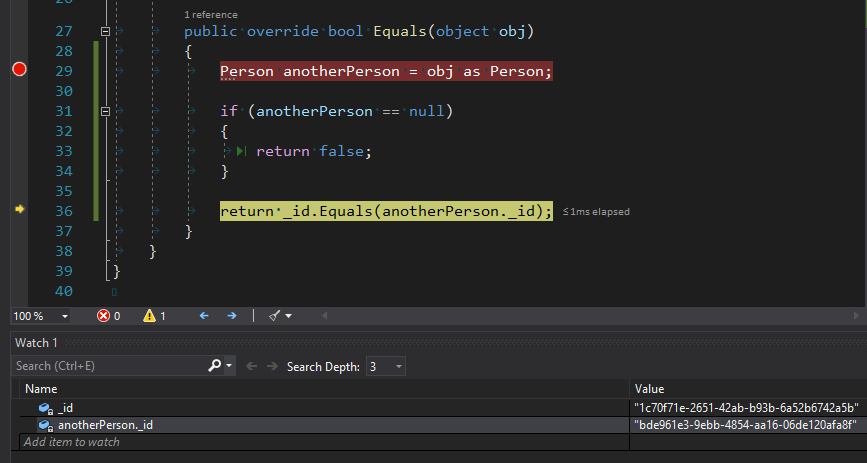
Теперь давайте в теле основной программы создадим вторую переменную класса Person и убедимся, что метод Equals работает так, как мы ожидали.

Так как метод Equals возвращает тип bool мы можем использовать его в операторе if в качестве условия и применять к нему любые другие логические операторы:



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

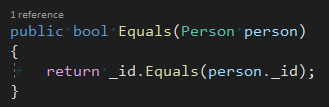


Разместим точку останова в методе Equals и посмотрим как именно будет происходить сравнение. Так же добавим окно watch для отображения значений полей \_id у текущего объекта и того, который был передан в метод Equals извне. Так как окно watch будет получать данные в контексте выполнения метода Equals внутри класса Person, то оно вполне сможет отобразить значения private полей \_id. В то время как размести мы точку останова вне контекста класса Person. Значения приватных полей были бы недоступны в окне watch.

Их значения совершенно разные, соответственно метод Equals вернет нам false, которое затем будет инвертировано логическим отрицанием в true которое уже будет считано оператором if. Переопределенный нами метод работает так, как мы от него ожидали.

Теперь давайте познакомимся ещё с одним проявлением полиморфизма.

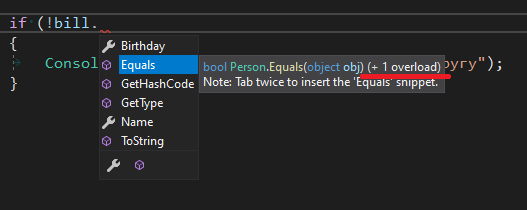
Для класса Person мы переопределили базовый метод Equals в который можно передать объект любого класса, который наследуется от object. Напишем же специфичный метод Equals который будет принимать только объекты класса Person.



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) количеством. Подробнее вот тут:

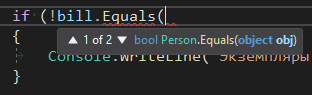
[Методы и их сигнатуры](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/methods)

То есть Equals(object obj) и Equals(Person person) компилятор видит, как разные методы класса Person.



Как мы видим метод Equals во всплывающем окне указан один раз, но в описании для него написано + 1 overload то есть, есть ещё одна перегрузка! Таким образом 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 реализована разная логика.

Для того, что бы посмотреть сигнатуры 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:

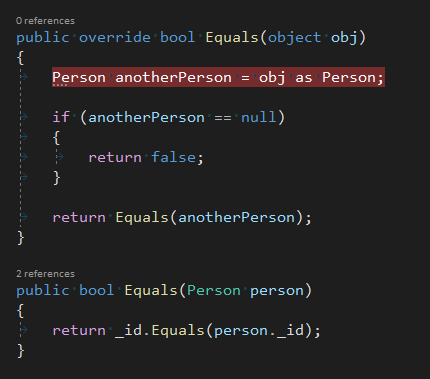


Теперь, вот этот код из примера выше будет автоматически использовать наиболее подходящую перегрузку, то есть вместо метода Equals(object obj) будет вызван метод Equals(Person person).

Мы никак не модифицировали внешний код, мы только написали более подходящую перегрузку к уже существующему методу, и компилятор автоматически начал её использовать.

Убедиться в том, что вызывается именно новый метод вы можете самостоятельно поставив точку останова в новом методе и запустив программу.

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 улучшить этот код вот таким образом:



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

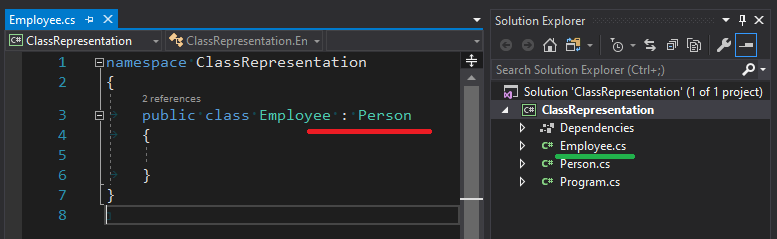
Вот мы и познакомились с ещё одним видом полиморфизма: перегрузкой методов внутри класса.

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

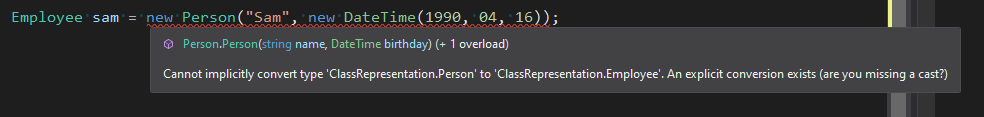


Передадим её в метод сравнения, для большей наглядности оставим остальное условие без изменений только добавим веточку else

До сих пор мы рассматривали 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, вернемся в тело основной программы и убедимся в этом.

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.

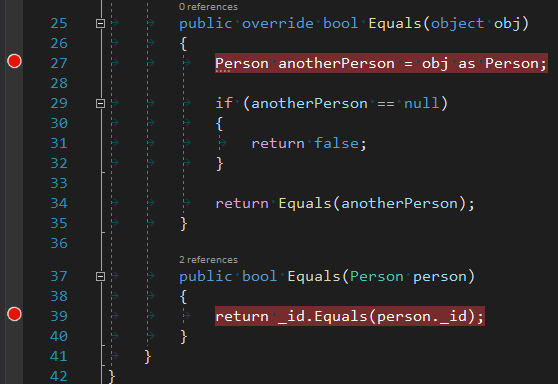
Дело в том, что конструктор может создавать экземпляр только своего класса для конструктора ковариантность (использование потомка вместо предка) работать не будет, в C# конструкторы могут работать только с экземплярами своего класса такое свойство называется **инвариантностью.**

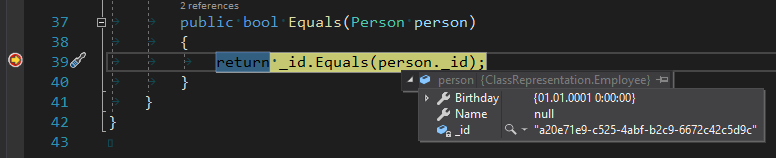
**Инвариантность – свойство позволяющее работать только с экземплярами специфического класса. Не с потомками или предками, а только с тем классом, который явно задан для данного действия.**

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



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 использована, поставим точки останова в обеих и посмотрим.

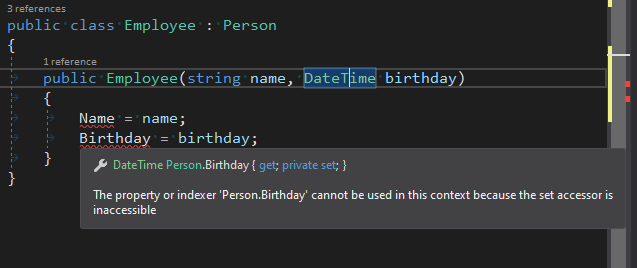


Сработает более специфичная реализация Equals так как благодаря ковариантности мы можем привести Employee к своему предку Person при передаче в метод:

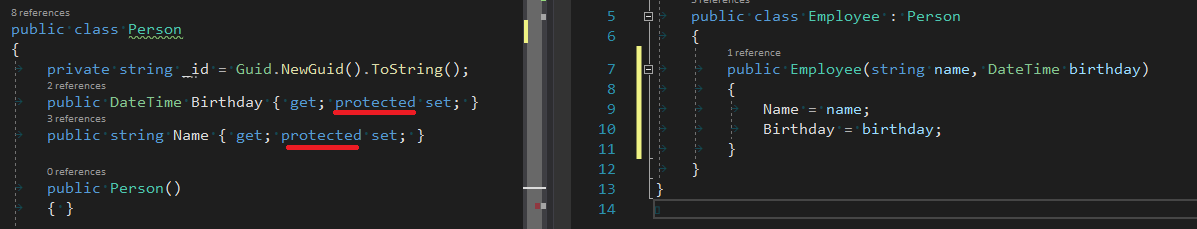
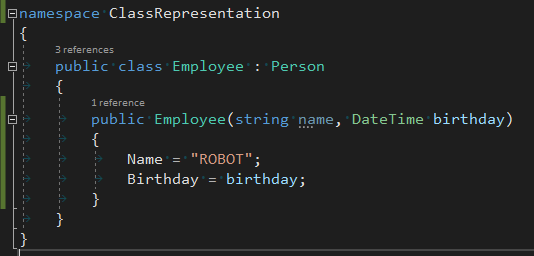
**Методы в C# ковариантны.**

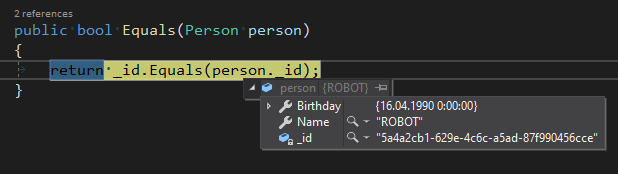
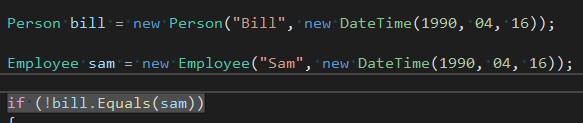
Наведем курсор на переменную person: C# понимает что это просто обертка для объекта потомка класса Employee (для удобства visual studio выводит нам полное имя класса вместе с пространством имен) и внутри себя данный объект содержит те же элементы, что и его предок.

Но, в отличии от предка, класс Employee не имеет возможности проинициализировать свойства Name и Birthday так как не имеет соответствующего конструктора, а конструктор из Person мы не можем использовать напрямую.

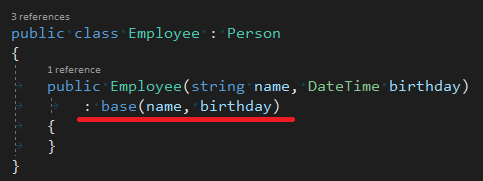
Конечно, мы можем написать для Employee точно такой же конструктор:Но мы столкнемся с проблемой доступа, дело в том, что свойства Name и Birthday имеют модификатор private на метод set то есть данный метод доступен ТОЛЬКО внутри своего класса! Только внутри класса Person можно задавать им значения.

Тут можно пойти двумя 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. Если мы меняем модификатор доступа у методов set, то это решает нашу проблему:Но порождает новую, мы можем поменять данные свойств внутри класса наследника и класс предок ничего не будет об этом знать!

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 не так.

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** точно так же, как при переопределении метода Equals которое сгенерировала visual studio

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 по отношению 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 экспериментировать с этим. Ниже приведу несколько полезных ссылок:

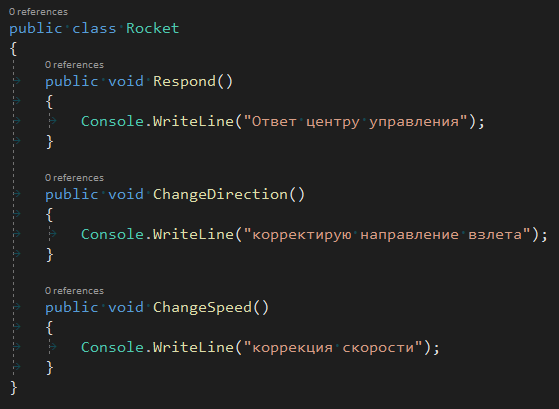
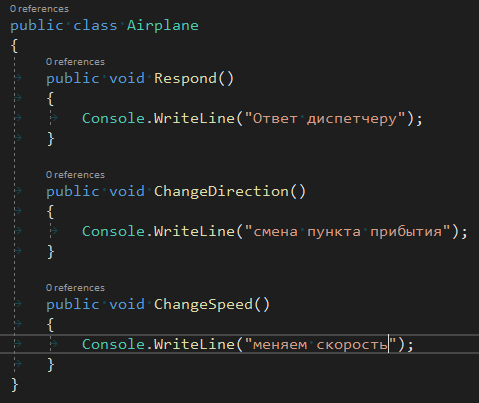
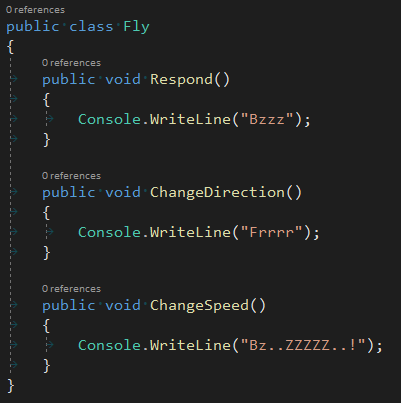
* [Наследование в C#](https://docs.microsoft.com/ru-ru/dotnet/csharp/tutorials/inheritance)
* [Переопределение методов в C#](https://docs.microsoft.com/ru-ru/visualstudio/ide/reference/generate-override?view=vs-2019)
* [Модификаторы доступа в C#](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers)

Мы же перейдем к заключительной теме. К **интрефейсам**. Создадим для этого новое приложение, по примеру того, как мы создавали ClassRepresentation. Назовем его InterfaceRepresentation Внутри него создадим три класса

* Fly
* Airplane
* Roсket

Сделаем их публичными, и в каждом классе сделаем три метода:

* ChangeDirection
* Respond
* ChangeSpeed

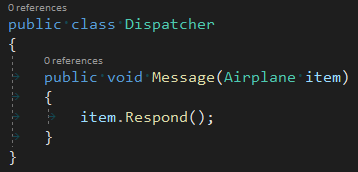
В качестве реализации все они будут выводить текстовые сообщения, так как для наших целей это не принципиально. Получаем три вот таких класса:

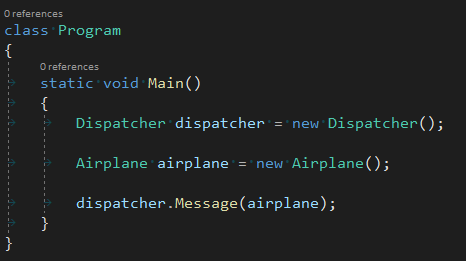
Как мы видим классы имеют общее поведение, но совершенно разные с точки зрения сущностей: и самолет, и муха, и ракета могут менять направление и скорость полета, так же они могут отвечать на внешний запрос.

Но наследовать их друг от друга или от некого общего предка будет неправильно с точки зрения логики – да они ведут себя похоже, но в будущем (когда классы будут обрастать новыми элементами) их поведение будет отличатся все сильнее и сильнее.

Так что унаследовать их друг от друга или от некого общего предка – не самое лучшее решение.

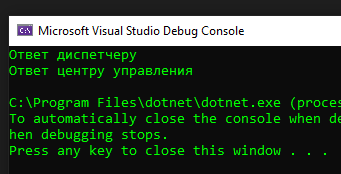
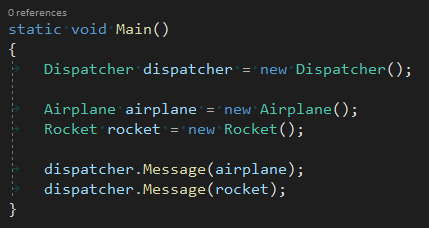
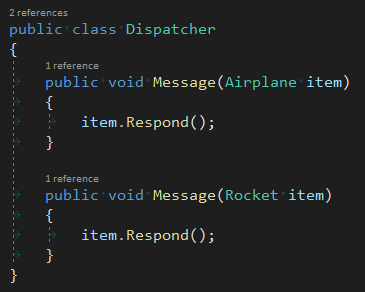
Теперь создадим класс Dispatcher который будет иметь следующую структуру:



Мы принимаем объект типа Airplane и вызываем у него метод Respond. Применим этот код в основном теле программы:

А теперь представим, что нам надо передать тому же диспетчеру не самолет, а ракету. А что, метод Respond есть и у неё.

Можно решить эту проблему перегрузкой метода Message у класса Dispatcher

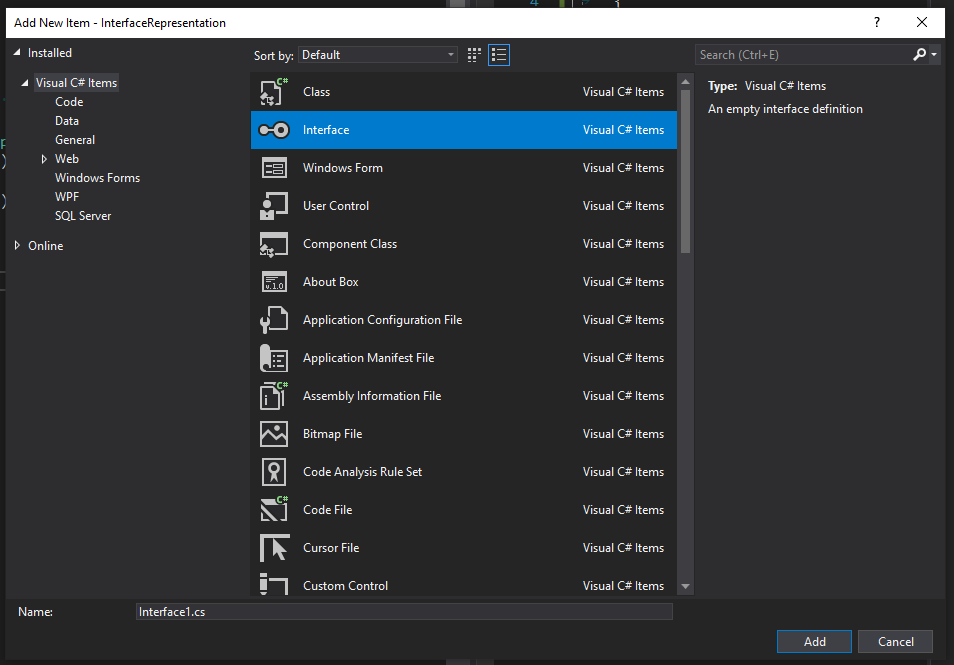


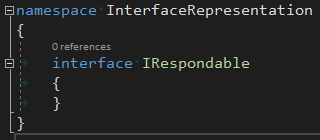
Это вполне рабочий вариант, но что если логика обработки в методе Message никогда не будет меняться, а типов обрабатываемых объектов будет 200 или 2000?

Класс Dispatcher очень быстро обрастёт огромным количеством перегрузок метода Message. Можно извернуться и унаследовать все эти объекты от некого общего предка и принимать предка в Message. Но мы уже пришли к соглашению что наследовать все подряд от общего корня – не самая хорошая идея с точки зрения архитектуры и разделения бизнес сущностей.

В C# есть удобный инструмент для ситуаций, когда нам нужно выделить общее поведение у объектов несвязанных иерархией наследования. Это **интерфейсы**,в общем смысле интерфейсы предоставляют нам контракт на предоставление набора публичных элементов, реализация которого гарантируется у всех классов, которые реализуют данный интерфейс.

Как раз наш случай! У нас есть три никак не связанных класса которые реализуют метод Respond который не возвращает значений, является публичным и не принимает параметров.

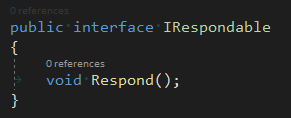
Это и есть наш контракт, осталось его записать на языке C#. Открываем solution explorer, открываем меню (правая кнопка мыши на имени приложения) выбираем пункт Add -> New Item. те же действия мы проделывали при добавлении класса. Но, на этот раз, в открывшемся окне выбираем не класс, а интерфейс: Назовем его IRespondable.cs так как он будет содержать один единственный метод в своём контракте, обратите внимание I в качестве первой буквы в именах интерфейсов является неформальной договоренностью в среде C# разработчиков, так же, как и именование интерфейсов по принципу **Iделающийчтолибо** при написании вашего кода желательно придерживаться данного стандарта. Жмем Add и получаем пустой интерфейс:

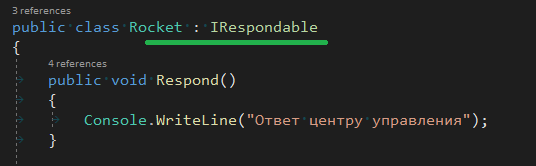
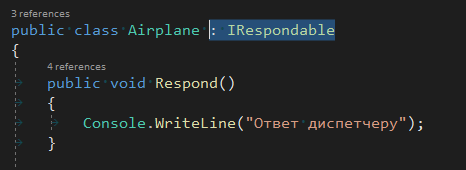


Как и класс интерфейс имеет модификатор доступа, сделаем его public и добавим контракт на метод Respond.

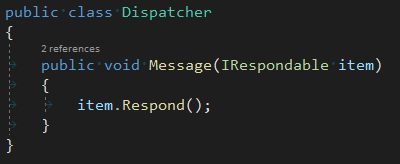
Для этого начнем с типа возвращаемого значения: void в нашем случае, затем укажем имя метода с пустыми параметрами и поставим точку с запятой.

Мы не указываем у контракта модификатор доступа, так как всё, что мы перечислим в интерфейсе по умолчанию имеет модификатор доступа public и, в отличии от метода, контракт не имеет реализации, реализовывать его будут классы, которые данный контракт будут выполнять. Получиться вот такой интерфейс:



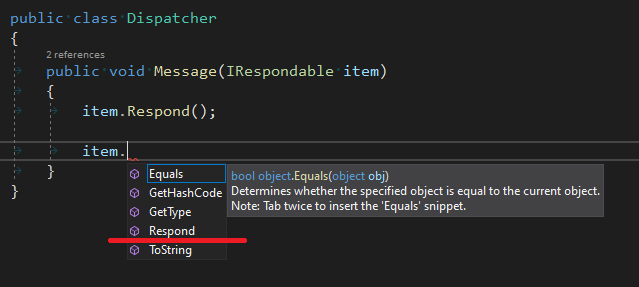
Перейдем к классы Airplane и Rocket и объявим в них реализацию созданного нами интерфейса:

Как видите синтаксис в этом случае не отличается от наследования, теперь мы можем в классе Dispatcher в методе Message поменять сигнатуру таким образом что бы метод принимал любой объект который реализует интерфейс IRespondable



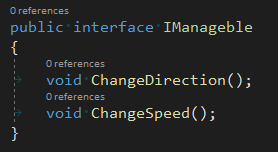
Вернемся в тело основной программы – никаких ошибок! Объекты обоих классов (и Airplane и Rocket) передаются в метод Message у объекта класса Dispatcher.

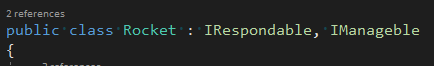
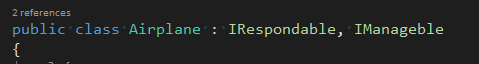
Это стало возможно потому, что метод Message оперирует не объектами конкретных классов Airplane и Rocket, а любым объектом, реализующим интерфейс IRespondable.

Объекты классов Airplane и Rocket используются только в том объеме, в котором они реализуют данный интерфейс. То есть, внутри метода Message мы можем оперировать только теми элементами классов Airplane и Rocket, которые объявлены внутри интерфейса IRespondable убедимся в этом:Ничего кроме метода Respond переменная item не содержит.

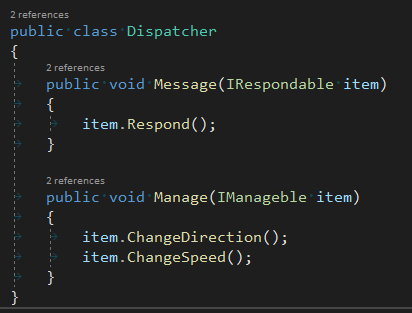
В отличии от наследования, при котором в C# у класса потомка может быть только один предок, в реализации интерфейсов таких ограничений нет.

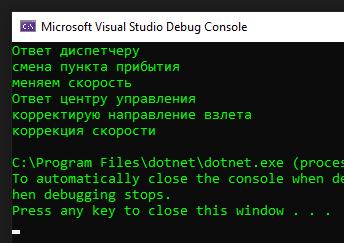
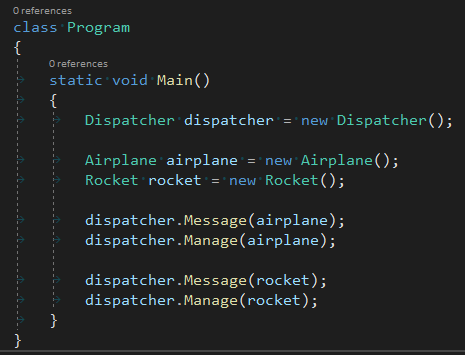
Сделаем ещё один интерфейс, который будет содержать два других метода классов Airplane и Rocket

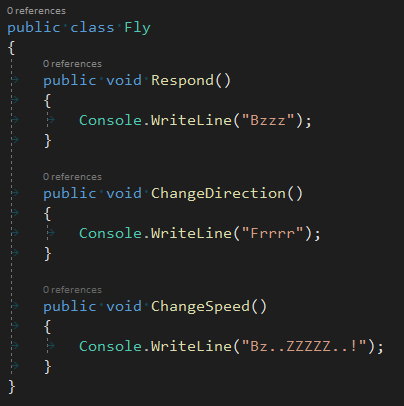


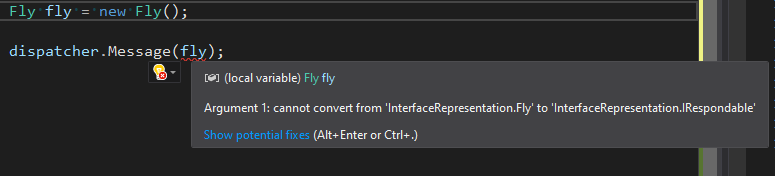
Объявим в классах Airplane и Rocket его явную реализацию

Мы можем просто перечислить через запятую все интерфейсы, которые реализует класс.

Добавим классу Dispatcher новый метод Manage для взаимодействия с объектами, реализующими интерфейс IManagebe:

Ничего необычного, всё то же самое что мы проделывали до этого с интерфейсом IRespondable

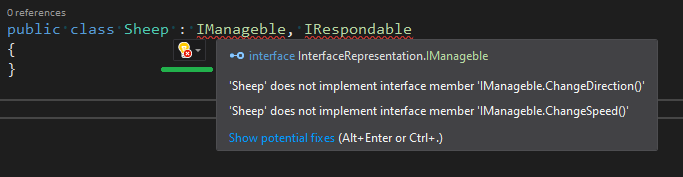
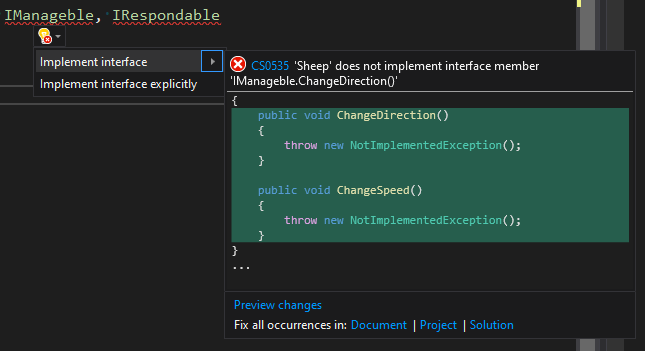
А что если мы попробуем передать в созданные нами методы класса Dispatcher объект класса Fly? Он же тоже имеет все необходимые методы

Давайте попробуем:

Несмотря на то, что данный объект реализует все те же самые методы, компилятор отказывается рассматривать его как объект реализующий интерфейс IRespondale.

Необходимо в явном виде указать реализацию интерфейса для того что бы использование объекта класса с соответствующими методами стало возможно.

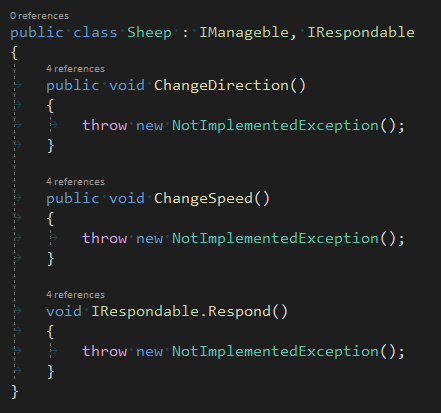
Вы можете самостоятельно это сделать и передать муху под управление диспетчера.

Напишем новый класс реализующий имеющиеся у нас интерфейсы, причем теперь мы знаем какие методы мы от него ожидаем. Так как интерфейсы у нас уже написаны. Нам достаточно написать вот так:Получить ругачку, навести курсор на лампочку, и мы увидим вот такую помогалку:Visual studio предлагает нам написать реализации указанных в инетерфейсе методов.

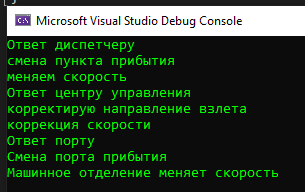
Они не будут содержать никакой логики и будут выбрасывать исключение при вызове, но они уже будут иметь правильное наименование, необходимый набор параметров и правильный тип возвращаемого значения.

Мы можем получить такую реализацию указанных в интерфейсах методов средствами visual studio.

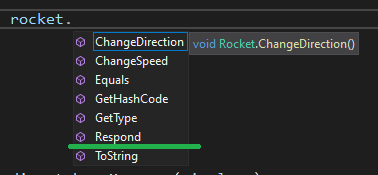
Обратите внимание, что в подсказке фигурировало два способа реализации интерфейсов: обычный и некий explicitly давайте проверим что будет, если метод Respond для интерфейса IRespondable мы реализуем вторым способом.

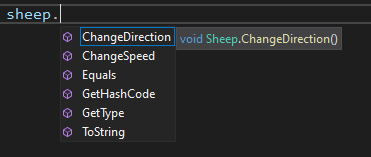


Логика внутри методов будет аналогична той, которую мы использовали для классов Airplane и Rocket мы не будем на этом останавливаться, добавим переменную нового класса в тело основной программы и отдадим её на обработку диспетчером. Мы уже неоднократно это делали, так что я не буду приводить здесь скриншот основного тела программы, покажу только результат выполнения:



Как мы видим несмотря на разницу в реализации интерфейса IRespondale поведение объекта класса Sheep аналогично поведению объектов классов Airplane и Rocket так в чем же разница?

Попробуем обратиться к методам объектов напрямую

У объекта класса Rocket метод Respond доступен как при обращении к объекту непосредственно, так и в случае, когда данный объект используется в рамках интерфейса IRespondable в то время как объект класса Sheep не предоставляет доступа к методу Respond напрямую:

Только в случае использования его в рамках интерфейса IRespondable вот в этом и будет заключаться отличие в обычной и explicitly реализации интерфейса.

**Под итожим:** когда классы не объединены общим предком мы можем выделить общую логику путем объявления интерфейса, содержащего контракты на соответствующее поведение.

Затем данный интерфейс присваивается классам, которые по нашей задумке должны обладать общей логикой.

А затем в тех местах кода, где мы хотим данной логикой воспользоваться мы обращаемся к объектам соответствующих классов в рамках интерфейса, который они реализуют.

Таким образом интерфейсы помогают добиться полиморфного поведения на уровне классов избегая наследования.

Полезные ссылки:

[Интерфейсы в C# хабрхабр](https://habr.com/ru/post/343852/)

[Интерфейсы в C# MSDN](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/interfaces/)