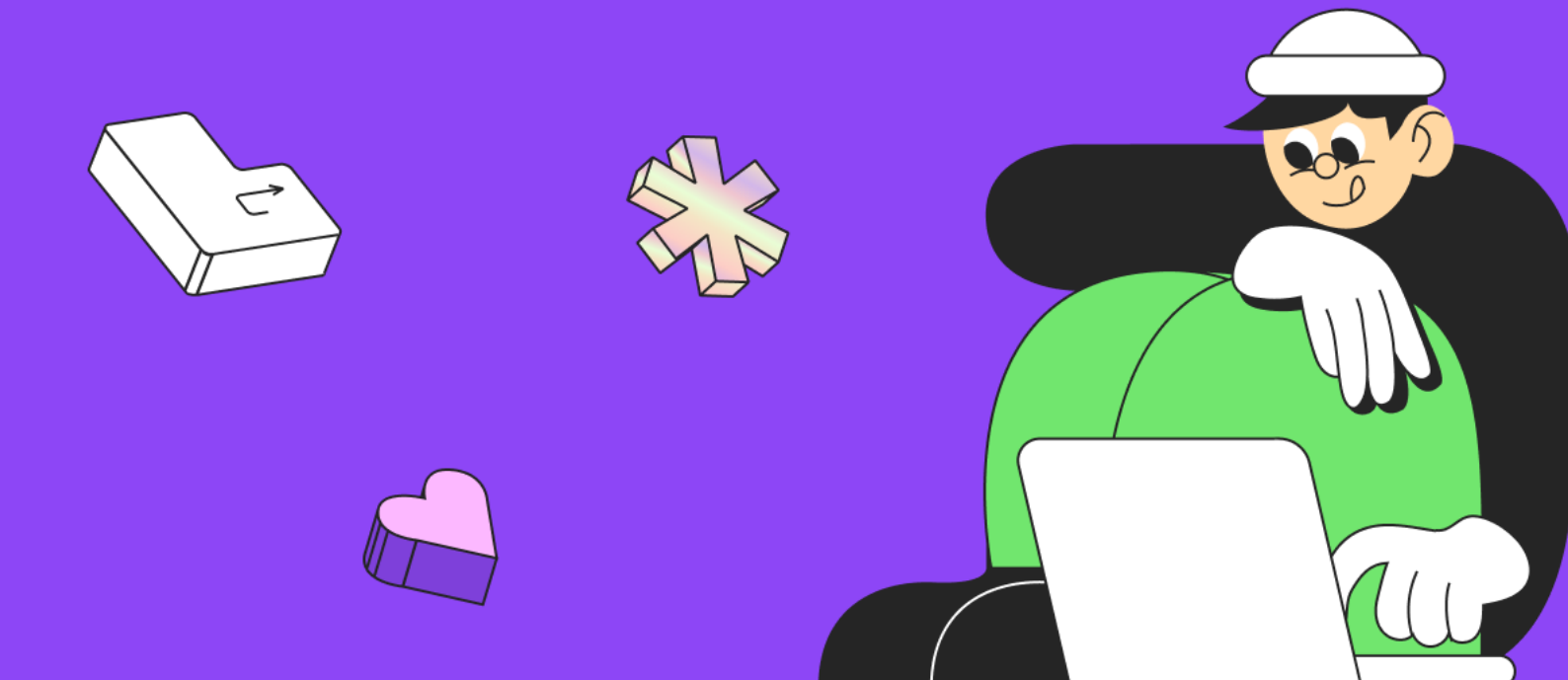
GeekBrains

Знакомство с функциями и массивами



Оглавление

[00.01.36]

Приветствие

Приветствуем вас на очередной лекции по программированию.

[00.01.39]

На этом уроке

1. Узнаем, что такое функции и массивы, а также для чего они нужны в программировании.

[00.01.47]

История

Зачем это и как пришли

На заре программирования, в начале 60-х годов прошлого века код писался так называемым стихийным способом. Сначала делалось что-то, а если это действие требовалось повторить, просто брался кусок кода, выполнялся его копипаст и затем использовался. В результате приложения выглядели как объёмный однотипный лапшевидный код. И со временем программисты поняли, что это не самый лучший подход и перешли к следующему этапу, который назвали процедурным подходом. Именно с таким подходом мы и познакомимся.

[00:02:29]

Задача

Для начала поставим перед собой задачу. Если на предыдущей лекции мы искали максимум из 5 чисел, то сейчас надо найти максимум из 9 чисел. Технически эта задача не сложнее предыдущей. То есть мы можем добавить 4 строчки кода, и всё будет хорошо. Но чтобы решить задачу со 109 числами, как раз потребуется, с одной стороны, функция, а с другой стороны — массивы.

[00:03:01]

Функции

Функции в обычной жизни

Начнём с функции. Посмотрим, где функция применяется в обычной жизни. Далее вспомним математику и применение функций в ней. После этого посмотрим, как использовать их в программировании.

В жизни, например, когда хотите поставить будильник или попросить кого-то сделать это, то говорите: «Поставь будильник на 19:00». Но, если начнёте описывать полный перечень действий, которые требуется сделать, допустим, установить будильник на телефоне, придётся взять телефон, разблокировать его, а затем открыть приложение «Будильник». Сначала

понадобится прокрутить колёсико часов, далее — минут, выбрать мелодию и нажать «Включить будильник». Это и есть те самые действия, которые обрамляются одним простым выражением: «Поставь будильник на нужное время».

Или, если мы хотим приготовить борщ, и для этого требуются различные продукты, всё, что надо сделать — взять деньги и пойти в магазин. Далее выбираем нужное, покупаем, приходим домой и начинаем что-то готовить. Если перечислять полный перечень действий, это бы выглядело следующим образом:

1. Возьми кошелёк.
2. Отсчитай нужное количество денег. *Или возьми карту, зайди в банковское приложение и проверь, что там хватает денег для покупки продуктов.*
3. Выйди в подъезд.
4. Нажми кнопку лифта.
5. Спустись на первый этаж.
6. Найди первый магазин.
7. Зайди в этот магазин.
8. Пройдись по всем продуктовым полкам.
9. Найди там все необходимые продукты. *Их также потребуется перечислить, потому что у всех разные запросы в приготовлении борща.*

То есть чтобы постоянно не делать одно и то же, мы проговариваем обычное предложение. Далее используем его, чтобы описать некий довольно большой перечень действий.

Или возьмём, например, приготовление омлета. Идея заключается в том, что надо взять яйца, щепотку соли, затем всё это перемешать и добавить молока. В результате через какое-то время на сковородке получается омлет. Если мы хотим накормить одного человека, то возьмём 2 яйца. А если же нам надо накормить 10 человек, очевидно, вместо 2 придётся использовать уже 15 или 20 яиц.

Как следствие, мы получаем такое понятие, как **аргументы функции**. Забегая немного вперёд, скажу, что в языке C# понятия **функции** как таковой не существует, но есть понятие

метода. Поэтому во время лекции функция иногда будет называться методом или **методом функции**. Это абсолютно нормально, и вы понемногу привыкнете к понятию **метод**.

[00:05:35]

Функции в математике

С жизненными функциями разобрались. Теперь поговорим о математике.

Допустим, в 7 классе записывались такие страшные на тот момент, а, возможно, для кого-то и сейчас, выражения:

$$f(x) = x^2 + 4$$

$$y = x^2 + 1$$

Это писалось по-разному: в младшей школе — через **y**, в старшей — через **f(x)**. Теперь представьте ситуацию, в которой вам требуется посчитать значение суммы в разных точках.

Допустим, этой точкой будет 1.2.3.4.5:

$$(1^2 + 1) + (2^2 + 1) + (3^2 + 1) + (4^2 + 1) + (5^2 + 1)$$

В общем случае наше, казалось бы, простое выражение становится объёмным. Но, используя математические символы, мы можем переписать его как:

$$f(1) + f(2) + f(3) + f(4) + f(5)$$

Таким образом, когда вы видите **f(1),** то сразу воспринимаете так: вместо x надо подставить 1, возвести 1 в квадрат и прибавить 1. А **f(2)** — как $2^2 + 1$ и так далее.

Но математики — хитрые люди. Они придумали более короткую запись:

$$\sum_{i=0}^{5} f(i)$$

В итоге всё, что написано на трёх строчках, представляет собой запись одного и того же.

Что любопытно, в математике есть и другие, более сложные функции. Соответственно, если бы у нас не было таких коротких обозначений — действий через **f(x)**, приходилось бы постоянно писать большие выражения. И чтобы это не делать, в математике используются функции. Для вас это могут быть простые флешбэки из школы, например, x^2, x^3 и так далее. Но

в высшей математике всё гораздо веселее и сложнее. Но не пугайтесь, мы не будем использовать их в рамках нашего курса. Это всего-навсего пример.

[00:07:25]

Функции в программировании

Теперь перейдём к применению функций в программировании, а используются они там постоянно. Дело в том, что практически всё программирование строится на этих функциях.

Например, когда мы генерировали псевдослучайные числа, то использовали такой механизм, как Next и в скобках указывали аргументы. Это и есть функция, которая в качестве аргументов принимает два числа: минимальной диапазон и максимальный диапазон, и выдаёт случайное число из этого диапазона. А использование оператора WriteLine не что иное, как метод, который в качестве аргумента принимает строку, выводя её в консоль или в терминал.

Таким образом, функцию, или в общем случае программирования — метод, мы можем воспринимать как некую коробку, которая на вход получает некие аргументы (входные аргументы функции), а на выходе выдаёт какой-то результат. По-другому эти функции или методы называются подпрограммами. Потому что если есть какая-то большая задача, то пытаться полностью её решать в рамках одного какого-то модуля не надо. Гораздо выгоднее разбить её на маленькие кусочки, сделать так называемую декомпозицию задачи. Таким образом, большая задача делится на перечень мелких подзадач, выполняя каждую из которых, получается общее решение и, соответственно, написанная вами программа.

Разберёмся, что ещё требуется, чтобы начать использовать функции. В первую очередь мы воспринимаем **функцию** как **часть программного кода, которая описывается самим разработчиком**. У каждой функции есть идентификатор или имя. Кстати, для именования используются те же принципы, что и для переменных. То есть в наименованиях используются только латинские символы латинские: маленькие и большие буквы. Вы можете использовать также цифры и символ подчёркивания. Но цифра не может стоять на первом месте, а символ подчёркивания в языке C# не принят.

Функция также имеет входные аргументы и технически может возвращать значение. В общем случае она может и не возвращать его, тогда в некоторых языках такие конструкции называются **процедурами**. В языке C# не принято разделение на функции и процедуры, а просто говорится **метод**. И этот метод либо возвращает какое-то значение, либо не возвращает никаких значений.

На слайде показано, каким образом выглядит общее описание функции в контексте языка C#. Оно может показаться сложным, но рассмотрим простой пример.

Мы только что взяли математическую функцию $f(x) = x^2 + 1$. Если переписать её на языке программирования, то увидим примерно следующее:

```
double f(double x)
{
        double result = x*x+1;
       result result
}
```

Значит, в первую очередь мы определяем наименование и указываем это как ***f***. Если она так называется в математике, то и здесь назовём её ***f***. Далее указываем, какие аргументы это функция принимает. Учитывая то, что в программировании абсолютно всё построено на типах данных, для x надо указать, каким типом данных он считается. В нашем случае это вещественное значение, так как в качестве аргумента можем передать, например, точку 2,5.

Как следствие, если 2,5 умножать на себя, снова получим вещественное число. Поэтому вся функция так же будет отдавать нам вещественное значение. На слайде это показано сноской «Возвращаемый тип».

Внутри фигурных скобок описывается тело метода. В общем случае здесь может быть больше двух строчек кода, но пока это функция простая, поэтому двух строчек достаточно:

double result = x*x+1 — вычисление необходимого значения

result result — то, что вернёт функция, чтобы в дальнейшем использовать это как $f(2)$ и куда-то сохранить значение, которое посчитает функция.

[00:11:31]

Поищем максимум из 9 чисел

Итак, мы усвоили теоретический минимум, чтобы напрямую использовать методы и самостоятельно их описывать. Теперь найдём максимальное из 9 чисел. Сначала решим эту задачу стихийно, а затем — с использованием методов.

Воплотим в жизнь идею нахождение максимума из 9 чисел. Для начала определим и опишем 9 переменных. Далее определим переменную максимум, куда положено значение первого аргумента. На следующем этапе проверим: если значение новой переменной, то есть **b1**, больше **max**, то в **max** положим **b1**:

If(b1>max) max=b1;

Технически это надо повторить много раз:

If(c1>max) max=c1;

Далее воспользуемся копипастом и допишем следующее:

If(a2>max) max=a2;

If(b2>max) max=b2;

If(c2>max) max=c2;

Главное — избавиться от всех потенциальных ошибок, которые могут быть допущены. Поэтому везде вместо 1 поставим 2.

Таким же образом находим максимум из тройки игроков:

If(a3>max) max=a3;

If(b3>max) max=b3;

If(c3>max) max=c3;

Далее аккуратно отформатируем код и выведем на экран:

Console.WriteLine(max);

Смотрим на вводные данные, находим значение 39 и снова проверяем написание переменных. Далее переходим к терминалу и вводим команду **dotnet run**, чтобы получить 39. Мы видим, что происходит сборка проекта, и на экране появляется 39.

Введём для теста какое-нибудь новое максимальное число. Пусть это будет 313. Проверяем и убеждаемся, что всё верно. Ещё раз меняем какое-нибудь значение — снова всё работает хорошо. То есть наш алгоритм реализован правильно.

[00:14:42]

Функция max

Таким образом выглядит решение, если использовать классический стихийный подход. Теперь воспользуемся функциями.

Итак, вначале опишем некоторую функцию, назовём её **max**. Так как мы работаем с целыми числами, то будем так же возвращать **int**:

int Max(int)

Далее наша функция будет проделывать такую работу: возьмёт три числа и найдёт из них максимальное. Затем мы возьмём следующие три числа и найдём максимальное, повторим это действие несколько раз, а потом устроим суперфинал. Это небольшая отсылка к игре «Поле чудес», наверняка некоторые из вас её помнят. Сначала пойдёт первая тройка игроков, затем — вторая и третья. После этого наступит финал.

Итак, возьмём некоторый аргумент 1, опишем некоторый аргумент 2 и 3.

int Max(int ard1, int ard2, int ard3)

После этого опишем тело метода:

```
{
        int result = arg1;
        if(arg2>result) result=arg2;
        if(arg3>result) result=arg3;
        return result;
}
```

Сначала у нас была подзадача, где мы искали максимум из трёх чисел. Теперь идёт всё то же самое, только в контексте метода. Определяем внутренний **result**, где будет храниться значение **max**. Далее, если arg2>result, то в result надо положить arg2. То же самое проделываем с третьей строчкой кода. Теперь требуется, чтобы наш метод возвращал **result**.

Таким образом, у нас появляется вспомогательный механизм, который ищет максимум из трёх чисел. Посмотрим, как можно это использовать.

Если в предыдущем примере мы писали 8 строчек кода, то теперь они не нужны. Удаляем их:

If(b1>max) max=b1;

If(c1>max) max=c1;

If(a2>max) max=a2;

If(b2>max) max=b2;

If(c2>max) max=c2;

If(a3>max) max=a3;

If(b3>max) max=b3;

If(c3>max) max=c3;

Но у нас должно быть определено три финалиста. Поэтому создадим переменную **max1** и скажем, что результатом работы функции **max** будет максимальная из трёх чисел: **a1, b1** и **c1**:

int max1 = Max(a1, b1, c1);

После того как отработает этот кусочек кода, мы получим максимум из первой тройки игроков. Запустим команду и убедимся в этом. Находим в системе 15, 21, 39. Ожидаем увидеть 39. Проверяем и убеждаемся, что всё хорошо.

Аналогичным образом проделаем эти действия для второй и третьей тройки игроков:

int max1 = Max(a1, b1, c1);

int max1 = Max(a2, b2, c2);

int max1 = Max(a3, b3, c3);

Поэтому заводим отдельные переменные, а после того как всё будет сделано, устроим окончательный финал. Попросим max принять на вход **max1, max2 и max3**:

int max1 = Max(max1, max2, max3);

Console.WriteLine(max);

Теперь покажем результат. Запустим программу, чтобы увидеть 2311. Далее возьмём ещё число, убедимся, что всё работает должным образом — да, снова выходит максимальное число. Посмотрим, что стало лучше.

Теперь у нас есть кусочек кода, который чётко отвечает за поиск максимума из трёх. Если возникает какая-то ошибка, то эту ошибку мы будем исправлять в одном месте, а не бегать по всему коду и в каждой строке вносить правки. Это первый плюс.

Рассмотрим следующий плюс. Сейчас мы применяем промежуточные переменные, чтобы использовать их в конечном финале. Но посмотрим, что может произойти на самом деле. Закомментируем это код, чтобы не мешал:

```
// int max1 = Max(a1, b1, c1);

// int max1 = Max(a2, b2, c2);

// int max1 = Max(a3, b3, c3);

// int max1 = Max(max1, max2, max3);
```

Перепишем результирующий вариант. Возьмём значение **max** переменной **max** и то, что посчитали. Но вместо того, чтобы использовать дополнительные переменные, внутри аргумента передадим функцию.

```
int max = Max(

Max(a1, b1, c1)

Max(a2, b2, c2)

Max(a3, b3, c3));
```

Это небольшая ссылка к функциональному программированию. Можно было оставить всё и на одной строке, и система бы сработала также хорошо.

Теперь введём новое максимальное значение, например, 112125 и перезапустим систему, чтобы увидеть на экране это число. Да, всё получилось. Таким образом, это самое простое введение в функции.

[00:20:01]

Массивы

Устройство массивов внутри компьютера

Мы познакомились с функциями, которые в языке C# принято называть методами. Теперь наша задача — избавиться от того количества переменных, что есть сейчас. Для этого

понадобятся **массивы**. О них говорилось на первом курсе по введению в программирование. А сейчас узнаем, как использовать массивы при написании кода.

Начинаем с указания типа данных. После этого открываем квадратные скобки и даём идентификатор или имя нашему массиву. Далее определяем значение, которыми оно будет наполнено. На экране показано несколько возможных вариаций того, как определять массивы. Сейчас на практике рассмотрим несколько вариантов, а уже на семинарских занятиях будем тестировать и использовать их все.

Итак, подправим наше приложение, чтобы оно уже превратилось во что-то более красивое. Создадим новый проект, который теперь будет называться IntroArray — это уже девятый пример. Далее повторим все действия, введя в программу dotnet new console. Воспользуемся этим кодом, немного его подправив:

```
// See https://aka.ms/new-console-template for more information

Console.WriteLine("Hello, World");
```

Итак, тот кусочек кода, представленный на экране, теперь не нужен. Нам требуется определить массив. Пишем **int**, затем ставим квадратные скобки, даём какое-то наименование и после этого перечисляем значения, которые хотим использовать.

```
int [] array = {1,2,3,4,5,6,7,8,9};
```

Сейчас это 9 значений, идущих по порядку. Далее правим значения и убираем лишнее:

```
int [] array = {11,21,31,41,15,61,17,18,19};
```

Теперь посмотрим, как заставить работать имеющийся код. Технически мы можем описать обращение к нужному элементу массива. Например, если у нас есть массив **array**, то первый элемент имеет индекс 0. Следующий элемент будет определяться через индекс 1, дальше — через 2, 3 и так далее.

Внимание! *Если в массиве 9 элементов, то индекс последнего будет равен 8, а индекс первого — 0. Это важно, потому что большая часть программирования всегда нумеруется с 0.*

Теперь, чтобы обратиться к конкретному элементу, напишем имя нашего массива, а в квадратных скобках укажем его индекс, например, 0. Присвоим нулевому элементу массива значение 12, так как после 11 текущего кода будет использоваться 12. Снова пишем **Console.WriteLine**, используя встроенный функционал:

Console.WriteLine(array[0]);

И ожидаем получить на экране 12.

Сделаем компиляцию проекта через dotnet run.

array[0] = 12

Console.WriteLine(array[0]);

На 9 строчке описано, как обратиться к массиву и записать в него значения, а на 10 — как обратиться к массиву и получить значение соответствующего элемента по указанному индексу. То есть, если бы надо было поставить элемент 4, написали **array 4**:

Console.WriteLine(array[4]);

А сейчас на экране ожидаем увидеть 15. Обратите внимание, что по порядку надо считать не количество, а индекс, в нашем случае это число 15. В результате всё получилось.

Мы заменили наши 9 переменных, чтобы сделать код короче. Теперь посмотрим, как использовать поиск максимума из 9.

Технически мы можем воспользоваться уже описанным функционалом. Сделаем это так же, как в предыдущем примере, то есть укажем функцию. Далее трижды вызываем **array 0**, **array 1** и **array 2.** После этого ставим запятую, здесь уже появляются автоматические подсказки:

int max = Max(

Max(array[0], array[1], array[2],

Max(array[3], array[4], array[5],

Max(array[6], array[7], array[8]

)

На следующей строке указываем элемент 3, 4, 5, а на третьей строке — 6,7,8. Убираем запятую и после выполнения этого кода в консоли ожидаем максимальное число нашего массива. Проверим это через **WriteLine**, указав значение максимума, и ставим точку с запятой в коде выше:

int max = Max(

Max(array[0], array[1], array[2],

Max(array[3], array[4], array[5],

Max(array[6], array[7], array[8]

)

Console.WriteLine(max);

Обратите внимание, что функция **Max** называется с большой буквы, а переменная **max** используется с маленькой буквы. Если такое написание смущает, то можно написать, например, значение **result** и положить в переменную **result** значение, которое возвращает функция:

int result = Max(

Max(array[0], array[1], array[2],

Max(array[3], array[4], array[5],

Max(array[6], array[7], array[8]

)

Console.WriteLine(result);

Запускаем и ожидаем получить число 61. Мы получили ошибку, потому что ожидается конец файла, то есть здесь есть что-то лишнее. Убираем лишнее и снова вводим **dotnet run**. Теперь всё работает хорошо. Результат действительно равен 61. Поменяем значение, поставим 211 и запустим систему.

Итак, мы решили задачу поиска максимума из 9. Но если в нашем массиве будет не 9 чисел, а, например, 7, возникнут проблемы. Больше проблем ждёт, если будет не 9 чисел, а 109. Как с этим справиться, поговорим позже. А пока воспользуемся методами и потренируемся в использовании массивов на примере задачи по поиску позиции нужного элемента.

[00:27:27]

Алгоритм

Допустим, у нас есть массив **array**, в котором **n** элементов. Найдём элемент, совпадающий с некоторым значением, который определяет пользователей.

1. Сохраним его в переменную **find**. Затем установим счётчик в нулевую позицию.
2. Если на текущей позиции элемент совпал с **find**, операцию можно завершить, потому что мы нашли позицию.

3. Если элемент не совпал с **find**, увеличиваем значение счётчика **index** на 1 и переходим на предыдущий шаг.
4. Сравниваем снова. И если элемент совпал с **find,** значит, алгоритм закончил работу.
5. Если элемент совпал с **find**, снова увеличиваем индекс. Смотрим результат.
6. В результате если элемент находится, операция завершается успешно. А если этого элемента так и нет, надо сообщить об этом.

[00:28:12]

Реализация алгоритма

Итак, реализуем теперь этот алгоритм кодом. Сначала определим новый массив. Дадим **int** имя **array**. Далее определим какое-то количество чисел, например, 8. Технически их может быть сколько угодно, потому сейчас мы реализовываем задачу для любого количества элементов. Отформатируем написанное:

int[] array = {1,12,31,4,15,16,17,18};

Далее по алгоритму требуется n элементов. Чтобы получить n, напишем:

int n = array.Length;

Внутри массива есть информация о том, сколько элементов в нём содержится. В частности, **array.Length** возвращает длину или количество элементов массива. Определили.

Далее надо, чтобы пользователь мог ввести число. Выберем число 4:

int find = 4;

Теперь по нашему алгоритму требуется установить некоторый счётчик **index**, поэтому определим его так же. Индекс, равный 0. Помним, что элементы в нашем массиве начинаются с 0, то есть 1 стоит под нулевой позицией:

Int index = 0;

Далее нам потребуется цикл **while**, в котором будем проверять: если **index < n**. Отмечаем, что на каждом этапе надо увеличивать значение индекса, поэтому прописываем:

while (index < n)

```
{
        index = index + 1;
}
```

Программисты пишут это в более короткой форме:

```
while (index < n)
{
        index + +;
}
```

Теперь выполняем второй пункт алгоритма. Если **array[index]** совпал с **find**, то алгоритм завершает свою работу. Можно при этом показать значение позиции.

```
while (index < n)
{
        if(array[index] == find)
        index + +;
}
```

Обратите внимание, что понятие эквивалентно равенству левой части и правой. То есть элемент, находящийся по нужному индексу, равен элементу find. Укажем **index.**

```
while (index < n)
{
        if(array[index] == find)
        {
                Console.WriteLine(index);
        }
        index + +;
}
```

Пока всё идёт хорошо. Сделаем запуск нашего проекта через **dotnet run** — получаем число 3.

Теперь в качестве искомого возьмём число 18 и ожидаем увидеть последнюю позицию. Последняя позиция соответствует 7. Не забываем, что индексы начинаются с нуля.

Посмотрим, что можно улучшить. Если у нас будет несколько одинаковых элементов, наш алгоритм покажет их все:

int[] array = {1,12,31,4,18,15,16,17,18};

Перезапустим систему и убедимся, что сначала появится позиция 4, а затем последняя. Разберёмся, как это исправить.

Технически нам потребуется новый оператор. Посмотрим, как он выглядит.

Если выполняется это условие, то просто добавляется **break** («прервать»):

```
if(array[index] == find)
{
        Console.WriteLine(index);
        break;
}
```

Перезапустим программу и убедимся, что будет найден первый элемент. Находим первый элемент и завершаем на этом свою работу. Мы выполнили условие пункта 2 нашего алгоритма. То есть, если **array[index]** совпал с **find**, значит, алгоритм завершил работу успешно, мы узнали индекс и решили задачу.

[00:32:20]

Синтаксис языка

Итак, мы потренировались в описании алгоритма и перевода его напрямую в код. Теперь надо потренировать синтаксис языка.

В текущем примере мы указывали значение массива вручную. А сейчас перепишем этот код с использованием генератора псевдослучайных чисел с использованием методов. Мы потренируем то, каким образом можно взять, например, метод, передать в него массив и заполнить массив нужным количеством элементов. На следующем этапе опишем метод, который будет выводить все элементы по порядку. Затем превратим наш код поиска нужного индекса в метод.

Сначала определим новый массив. Пусть это будет массив под именем **array**. Далее укажем, что в этом массиве будет по умолчанию 10 элементов. Запомним новую конструкцию **new int [10]**, которая дословно означает «создай новый массив, где будет 10 элементов». По умолчанию. Кстати, он будет наполнен нулями. Чтобы заполнить массив случайными числами, воспользуемся методом, который опишем сами:

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

Начнём с ключевого слова **void**. Далее дадим наименование нашему методу. В качестве аргумента будет приниматься какая-то коллекция, то есть аргумент **collection**. Затем нам надо получить длину нашего массива. Делается это посредством **collection.Length**. На следующем этапе возьмём позицию **index**, которая по умолчанию будет начинаться с 0. Теперь в цикле while пропишем **index < length**. В фигурных скобках пропишем **index + 1**, но такую конструкцию принято на языке C# писать более компактно.

```
void FillArray(int[] collection)
{
        int length = collection.Length;
        int index = 0;
        while (index < length)
        {
                //index = index + 1;
                Index++;
        }
}
```

После этого обратимся к аргументу **collection** на позицию **index** и положим туда новое случайное число — целое число из диапазона 1–10.

```
void FillArray(int[] collection)
{
        int length = collection.Length;
        int index = 0;
        while (index < length)
```

```
        {
            collection[index] = new Random().Next(1, 10);
            //index = index + 1;
            Index++;
        }
}
```

Запустим этот код и убедимся, что ошибок, которые возникают в процессе написания, например, текста, нет. Кодом это пока сложно назвать.

Таким образом, если в аргументе вместо **collection** записать условный **array** и запустить проект, Visual Studio Code сразу покажет потенциальные ошибки. В нашем случае система подсказывает, что на строке 7 есть проблема — нет локальной переменной с именем **array**. Это надо исправить.

Итак, первый метод готов. Пока тестировать его не будем.

Теперь сделаем метод **void**, который будет печатать массив. Аналогичным образом в качестве аргумента здесь будет приходить массив. Обратите внимание, что здесь мы специально не даём одинаковые имена, чтобы привыкнуть называть разные аргументы различными именами. То есть в первом случае будет **collection**, а во втором, например, **col**:

```
void FillArray(int[] collection)
void PrintArray(int[] col)
```

Количество элементов обозначим таким образом:

```
int count = col.Length;
```

Далее обозначим текущую позицию не через именование перемен **index**, а через **position**^

```
int position = 0;
```

Затем возьмём новый цикл **while** и укажем **position < count**:

```
while (position < count);
```

Затем выведем через **Console.WriteLine** значение текущего элемента, то есть **col[position]**. Далее запишем **position++**. То есть увеличиваем значение текущей позиции:

Console.WriteLine(col[position]);

position++;

Теперь разберёмся, для чего используется ключевое слово **void**. Дело в том, что в контексте языка C# есть методы, которые могут возвращать или не возвращать какие-то значения. **Если метод ничего не возвращает, он называется void-методом**. Обратите внимание, что в этом случае в коде оператор return, отвечающий за поиск максимума из 3, не используется.

Протестируем наш метод. Для начала напишем **FillArray** и в качестве аргумента передадим наименование нашего массива. Затем вызовем следующий метод **PrintArray**, который будет распечатывать наш массив. Запускаем и видим, что нет **Run**. Обычно код запускается со второго раза, но иногда бывают исключения, и он срабатывает с первого.

Итак, сначала мы определили массив из 10 элементов. Далее вызвали метод **FillArray**, который заполнил массив, а отдельный метод **PrintArray** нам его распечатал. В нашем случае это выглядит так:

FillArray(array);

PrintArray(array);

Если перезапустим наш пример, то с вероятностью 100% у нас появится новый набор чисел. Затем будет ещё один набор чисел, и так далее.

[00:38:37]

Метод IndexOf

Мы попробовали написать свои первые методы: метод заполнения массива и метод его печати на экран. Теперь попробуем адаптировать решение предыдущей задачи, в которой находили нужные элементы и позицию нужного элемента в массиве.

Для этого потребуется описать метод, отличный от **void**. Он будет возвращать позицию, то есть **index**. Назовём этот метод **IndexOf**, а в качестве аргумента будет приходить массив **collection** и какой-то элемент **find**.

Далее определяем количество элементов через **count** — **collection.Length**. Нам потребуются индексы, чтобы щёлкать массив, пусть это будет переменная **index**.

Затем возьмём цикл **while**, который будет проверять **index < count**. Ведём в фигурные скобки **index++**. Напишем следующее: если **collection[index]** совпал с **find**, потребуется куда-то

сохранить позицию. Чтобы её куда-то сохранить, определим новую переменную **position**. И пусть по умолчанию это будет 0. Это не совсем общее решение, но пока подойдёт.

В **position** положим значение нашего индекса. После того как этот цикл отработает, ожидаем **return position**, нашу позицию элемента.

```
int IndexOf(int[] collection, int find)
{
        int index = collection.Length;
        int count = 0;
        int position = 0;
        while (index < count)
        {
                If(collection[index] == find)
                {
                        position = index;
                }
                index++;
        }
        Return position;
}
```

Теперь протестируем. Введём **Console.WriteLine**. Определим переменную **pos** и положим в неё результат работы метода **IndexOf**. В качестве аргумента будет передаваться наш массив. Например, будем искать число 4.

```
int pos = IndexOf(array, 4);
Console.WriteLine(pos);
```

После того как метод отработает, на экране покажем **pos**. Запустим приложение и посмотрим, что получится.

Видим перед собой массив, где первая 4 находится на 3 позиции, а вторая — на 4. Получаем последнее вхождение.

Посмотрим, как изменить код, чтобы получать 1 вхождение. Всё очень просто — через добавление оператора **break**. Снова перезапускаем программу и видим, что 4 находится на нулевой позиции, а других четвёрок нет.

Принудительно добавим пару четвёрок. Пусть, например, на четвёртой позиции будет 4. После того как мы заполнили всё случайными числами, принудительно добавим пару 4. Например, на четвёртую позицию и на шестую. Снова запустим **dotnet run**.

Обратите внимание, что сейчас 3 находится на нулевом индексе, а 4 — на первом индексе. Встречаются и другие четвёрки, но нам действительно показали первую позицию.

Технически кажется, что мы всё сделали. Но есть проблема: если поискать элемент, которого точно не существует, например, элемент 444, и запустить этот код, выйдет позиция 0.

Посмотрим, как обходить такие ситуации. Программисты придумали одно изящное решение. По умолчанию мы указывали позицию 0. Очевидно, что нулевая позиция есть. Если мы ищем какой-то элемент, то либо он будет равен 0, либо больше 0. Но если не встречается ни одного элемента, то договоримся, что по умолчанию станет возвращаться значение -1. Это искусственный приём. То есть, если элемента нет, значит, выйдет -1. Таким образом, если запустить наш код, обнаружится значение позиции, равное -1. Это значит, что такой элемент не найден.

[00:43:40]

Заключение

Итак, на этой лекции мы познакомились с массивами и немного поработали с методами. Узнали, что массивы в языке C# описываются разными способами. Посмотрели, как описываются методы, попробовали написать собственные методы, использовали методы, которые работают с массивами, заполняющими массивы. Подключили знания предыдущей лекции. Мы уже вовсю используем циклы и генератор псевдослучайных чисел, поэтому продолжаем практиковаться и решать больше задач. А на этом всё, до встречи на семинарских занятиях.