GeekBrains

Введение в C#

Базовые типы C#

Операторы ветвления. Область видимости

На этом уроке

1. Познакомимся с базовыми типами языка C# и применяемыми к ним операторами.
2. Научимся добавлять логику в программы с помощью операторов ветвления.

Оглавление

Область видимости переменных

Переменные в языке C# имеют свою область видимости (scope или блок). Область видимости — часть программы, где переменная, поле, класс или иное объявление будет доступно.

Областям видимости чаще всего предшествует какое-либо объявление: название пространства имён, определение класса или функции, объявление условного оператора или цикла. Области видимости позволяют нам создавать иерархию кода, а также управлять выполнением отдельных частей кода.

Мы можем выделить набор операций в отдельную область видимости с помощью фигурных скобок. Распространённая ошибка — нарушение баланса фигурных скобок, что приводит к невозможности скомпилировать программу. Visual Studio автоматически добавляет закрывающуюся скобку, когда мы печатаем открывающуюся, но при этом нужно быть внимательными при копировании и вставке частей кода.

Важно помнить, что объявления должны быть уникальными в области видимости. Когда мы подключаем к программе новые namespace, их объявления также попадают в нашу область видимости, поэтому мы можем использовать их объявления (например, класс System.Console). Рассмотрим пример:

```
using System;
namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            int someAnotherValue = 42;
            {
                int scopedValue = 5;
                Console.WriteLine($"{someAnotherValue} {scopedValue}");
            }
            {
                int scopedValue = 2;
                Console.WriteLine($"{someAnotherValue} {scopedValue}");
            }

            Console.WriteLine($"{someAnotherValue}");
            Console.WriteLine($"{scopedValue}");
        }
    }
}
```

Как мы помним из предыдущего урока, функция `Main` — точка входа в приложение, Это означает, что после запуска программы начнётся выполнение всех операций, описанных в теле функции `Main`.

Пространство имен `HelloWorld` содержит класс `Program`, который в свою очередь содержит функцию `Main`. В теле функции `Main` мы объявляем переменную `someAnotherValue` и делаем два блока. В каждом из этих блоков нам доступно значение переменной `someAnotherValue`, объявленной в блоке выше. Также, каждый из этих блоков объявляет внутри себя переменную `scopedValue`, причём это две абсолютно разные переменные. Каждая из них доступна только внутри того блока, в котором объявлена. Мы не имеем доступа к этим переменным из вышестоящего или из соседнего блока.

Доступные нам объявления состоят из тех, что объявлены в нашем блоке или в вышестоящих блоках. Объявления, описанные в соседних или в нижестоящих блоках, нам недоступны.

Внимание! *В дополнение к текущему документу прилагается изображение «Области видимости» с наглядным изображением описанных деталей.*

Типизация в языке C#

Язык C# обладает строгой типизацией. Это означает, что переменные, аргументы функций и возвращаемые ими значения обладают типом. Тип (например, строка или число) задаёт множество значений и определяет возможные операции. Например, мы можем присвоить переменной с уже знакомым нам типом `string` любые строковые значения, а также можем объединять значения нескольких переменных в одно значение.

Далее мы рассмотрим основные типы данных в языке C# и научимся выполнять с ними базовые операции.

Числовые типы данных

C# задаёт несколько числовых типов данных. Выбор типа зависит от определенной условий — числового диапазона, необходимой точности и прочих. Мы рассмотрим наиболее часто применяемые типы числовых данных. В конце будут даны ссылки, содержащие перечень всех имеющихся числовых типов. Числовые типы данных C# можно разделить на целочисленные типы и типы с плавающей запятой. Стоит учитывать, что каждый такой простой тип имеет свой диапазон значений, который стоит соблюдать, чтобы не получить ошибку.

Целочисленные типы данных

Целочисленные типы подходят для хранения целых чисел. Наиболее часто мы будем использовать типы данных `int`, `long`, `byte`. У каждого из типов собственные диапазон значений и операции преобразования (их рассмотрим далее)

Тип данных	Диапазон значений
byte	0..255
int	-2 147 483 648..2 147 483 647
long	От -9 223 372 036 854 775 808..9 223 372 036 854 775 807

Представление чисел в разных системах счисления

C# поддерживает представление чисел в десятичной, шестнадцатеричной и двоичной форме. Система счисления задаётся в записи значения числа:

```
int decimalLiteral = 42; // просто число в десятичной форме
int hexLiteral = 0x2A; // 0x указывает на шестнадцатеричную форму
int binaryLiteral = 0b_0010_1010; // 0b указывает на двоиччную форму.
```

Знаки подчёркивания можно использовать в любой форме для наглядного разделения разрядов. Их использование никак не влияет на значение переменной:

```
int million = 1_000_000;
int oneMillion = 1000000;
Console.WriteLine(million); // 1000000
Console.WriteLine(oneMillion); // 1000000
```

Числовые типы с плавающей запятой

Числовые типы с плавающей запятой используют для хранения действительных чисел (все положительные числа, отрицательные числа и нуль). Они отличаются друг от друга точностью — количеством знаков после запятой. Наиболее точный тип — `decimal`, он используется в денежных операциях, т. к. в них требуется высокая точность.

Тип данных	Приблизительный диапазон	Точность
float	$\pm 1,5\ x\ 10^{-45} .. \pm 3,4\ x\ 10^{38}$	6–9 цифр
double	$\pm 5,0\ x\ 10^{-324} .. \pm 1,7\ x\ 10^{308}$	15–17 цифр
decimal	$\pm 1,0\ x\ 10^{-28} .. \pm 7,9228\ x\ 10^{28}$	28–29 цифр

Диапазон значения и как его найти

Как было написано выше, диапазон у простых типов ограничен и не всегда можно вспомнить, особенно в начале своего пути разработчика. И именно для такого случая всегда можно воспользоваться специальным свойством у типа - `MinValue` и `MaxValue`

```
// Тип int, минимальное значение -2147483648
Console.WriteLine($"Тип int, минимальное значение {int.MinValue}");

// Тип int, максимальное значение 2147483647
Console.WriteLine($"Тип int, максимальное значение {int.MaxValue}");

// Тип long, минимальное значение -9223372036854775808
Console.WriteLine($"Тип long, минимальное значение {long.MinValue}");

// Тип long, максимальное значение 9223372036854775807
Console.WriteLine($"Тип long, максимальное значение {long.MaxValue}");
```

Представление даты и времени

Для представления даты и времени в C# используется `DateTime`. Чтобы задать дату, нужно создать экземпляр этой структуры. Наиболее часто применяются два варианта создания конструктора: содержащая дату и время (годы, месяцы, дни, часы, минуты, секунды) и содержащая только дату (годы, месяцы, дни).

```
DateTime dateAndTime = new DateTime(2015, 8, 4, 16, 23, 42);
DateTime date = new DateTime(2021, 1, 1);

Console.WriteLine(dateAndTime); // 04.08.2015 16:23:42
```

```
Console.WriteLine(date); // 01.01.2021 0:00:00
```

Мы можем добавлять и убавлять произвольные значения составляющих даты:

```
DateTime jan01 = new DateTime(2020, 1, 1);
DateTime jan15 = jan01.AddDays(14);
```

По умолчанию при выводе в консоль дата и время форматируются в соответствии с региональными настройками. Вывод дат можно форматировать, используя определенные обозначения форматов дат и времени. Полный перечень составляющий строки форматирования приведен в дополнительных источниках к этому уроку. Вот некоторые примеры форматирования дат:

```
DateTime date = new DateTime(2015, 8, 4, 16, 23, 42);

Console.WriteLine(date.ToString("dd.MM.yy")); // 04.08.15
Console.WriteLine(date.ToString("HH:mm:ss")); // 16:23:42
```

Логический тип данных

Логический тип данных может иметь одно из двух значений — `true` или `false`. Логические значения часто используют вместе с логическими операторами для задания условий в работе программы. В языке C# логический тип данных называется `bool`:

```
bool isWinterComing = true;
```

Переменные с типом данных `bool` часто называют флагами. Если значение флага `true`, говорят, что флаг поднят (или выставлен), если значение флага `false` — флаг снят.

Операторы языка C#

Арифметические операторы, класс `Math`

К числовым типам данных можно применять арифметические операторы — сложение, вычитание и прочие. Язык C# предоставляет набор арифметических операторов, а платформа .NET содержит класс `Math`, предоставляющий некоторые распространенные математические функции.

Для лучшего понимания операторов нужно отметить, что операторы — это функции, а операнды — аргументы функций. Эти функции описаны в реализациях .NET, а некоторые из них можно переписать в своей программе.

Арифметические операции возвращают результат того же типа, что и его операнды — если сложить два числа с типом `int`, то результат также будет числом типа `int`.

Арифметические операции, в которых один аргумент целочисленный, а второй — вещественный, приводятся к вещественному типу:

```
int + float = float, int + double = double, int + decimal = decimal
```

Арифметические операции, в которых один аргумент — `float`, а второй — `double`, имеют в результате `double`, как более точный тип.

Во избежание ошибок, арифметические операции с типами `float` или `double` с одной стороны и `decimal` с другой, невозможны. В таких случаях необходимо производить приведение типов (далее в этом уроке).

Простые математические операторы

Операторы сложения, вычитания, умножения и деления вычисляют значение для двух операндов:

```
int a = 5;
int b = 3;
Console.WriteLine(a % b); // 2
Console.WriteLine(a + b); // 8
Console.WriteLine(a - b); // 2
Console.WriteLine(a * b); // 15
Console.WriteLine(a / b); // 1
```

Обратите внимание: так как оба значения целочисленные, операция деления также вернула целочисленное значение.
Оператор, обозначенный символом процента (`%`), — оператор взятия остатка от деления.

Инкремент и декремент

Они увеличивают или уменьшают значение переменной на единицу. Разделяют постфиксную и префиксную форму записи операторов.

Префиксная форма записи (`++a` или `--a`) в начале применяет оператор, а затем возвращает результирующее значение, в то время как постфиксная форма записи (`a++` или `a--`) возвращает текущее значение переменной, а затем увеличивает её значение. Рассмотрим на примерах:

```
int a = 5;
int b = 3;
```

```
Console.WriteLine(a++); // 5
Console.WriteLine(a); // 6
Console.WriteLine(++a); // 7

Console.WriteLine(b--); // 3
Console.WriteLine(b); // 2
Console.WriteLine(--b);// 1
```

Вначале в переменной `a` содержится значение 5. При выводе на консоль `a++` вернёт текущее значение (5), а затем увеличит его на единицу (6).
При выводе на консоль `++a` в начале увеличит значение переменной на единицу (7) и только потом отдаст это значение. Поэтому при выводе на экран мы увидим новое значение переменной.
Оператор декремента работает таким же образом.

Внимание! *Об отличиях префиксной и постфиксной формы операторов часто спрашивают на собеседованиях.*

Составные операторы

Составные операторы содержат знак равенства и знак применяемой операции. Составные операторы сложения и вычитания используются для увеличения или уменьшения значения переменной **на какое-то число**. Составные операторы умножения и деления используются для увеличения или уменьшения значения переменной **в несколько раз**. Любой составной оператор обладает полной формой и может быть записан при помощи обычных арифметических операторов:

```
int a = 5;

a += 1; // a = a + 1
a -= 2; // a = a - 2
a *= 3; // a = a * 3
a /= 4; // a = a / 4
Console.WriteLine(a); // 3
```

На этапе компиляции компилятор разбивает составные операторы на две операции и применяет обычные операторы вместе с операцией присваивания. По сути, составные операторы являются своего рода синтаксическим сахаром — такое определение даётся операторам или ключевым словам языка, которые никак не меняют поведение программы, но позволяют записать операции более удобным образом.

Внимание! *Составные операторы сложения и вычитания не принято использовать в случае, если вы хотите увеличить или уменьшить значение переменной на 1 (как сложение в коде выше). Для таких случаев нужно воспользоваться операторами инкремента/декремента.*

Класс `System.Math`

`System.Math` предоставляет прочие часто применяемые математические функции и константы. Например, тригонометрические операции (`Math.Sin()`, `Math.Cos()`, ...), значение числа π и экспоненты (`Math.PI`, `Math.E`), взятие числа по модулю (`Math.Abs`).

Логические и условные операторы

C# предоставляет логические операторы и условные логические операторы (logical / conditions). Для лучшего понимания мы будем называть их побитовыми операторами и условными операторами, чтобы не путаться в определениях. Побитовые операторы применяются к числам в байтовом представлении и возвращают новые числовые значения.

Условные операторы применяются к операндам с типом `bool`. Результатом также будет логическое значение (true/false). Чаще всего применяются операторы условного И и условного ИЛИ. Для описания условных операторов используют специальные таблицы истинности, в которых демонстрируется зависимость операндов и выходного значения оператора.

Оператор условного И

Оператор условного И определяется знаком двойного амперсанда (`&&`) и возвращает true только в том случае, если оба операнда равны true. В противном случае возвращается false. Оператор применяется, когда нам необходимо удостовериться в истинности всех условий для осуществления операций. Например, чтобы дать пользователю доступ в административный раздел сайта, необходимо, чтобы пользователь был авторизован **И** имел роль администратора. Несоблюдение хотя бы одного из этих условий должно служить отказом:

```
bool showAdminPage = isAuthorized && isAdmin;
```

Таблица истинности для оператора условного И:

a	**b**	**a && b**
false	false	false
false	true	false
true	false	false
true	true	true

Оператор условного ИЛИ

Оператор условного ИЛИ определяется знаком двойной вертикальной черты (`||`) и применяется в тех случаях, когда нам достаточно истинности хотя бы одного из условий. Оператор возвращает false, только когда оба операнда равны false. В противном случае возвращается true. Например, мы выбираем тип транспорта для поездки на работу — личный или общественный. Мы можем исходить из нескольких факторов. Допустим, мы решаем воспользоваться общественным транспортом в том случае, если:

- на улице хорошая погода (не хотим идти под дождём до остановки);
- дороги перегружены (не хотим стоять в пробке за рулём);
- личный автомобиль сломался.

Хотя бы один из этих критериев сделает решение воспользоваться общественным транспортом истинным:

```
bool useBus = isGoodWeather || isRoadLoaded || isCarBroken;
```

Таблица истинности для оператора условного ИЛИ:

a	b	a || b
false	false	false
false	true	true
true	false	true
true	true	true

Побитовые операторы И, ИЛИ

Побитовые операторы И, ИЛИ похожи по своему поведению на условные варианты операторов, с той разницей, что применяются к значениям чисел побитово (бит за битом). Такие операторы обладают высокой производительностью, так как они — простейшие операции, которые поддерживает процессор.

Мы уже знаем, что оператор && возвращает `true` только в одном случае, как и оператор || возвращает `false` только в одном случае. Если принять за `true` значение «включённого» бита (единица), а за `false` — значение «выключенного» бита (нуль), мы получим определение побитовых операторов:

1. Оператор побитового И (`&`) — сравнивает биты двух чисел и возвращает число, у которого **включены** только те биты, которые равны единице у обоих операндов.

2. Оператор побитового ИЛИ (`|`) — сравнивает биты двух чисел и возвращает число, у которого **отключены** только те биты, которые равны нулю у обоих операндов.

Побитовое И

Побитовое И чаще всего применяется в так называемых битовых масках (bitmask) — приёма в программировании, позволяющего получить значения определённых бит числа. Битовые маски применяются в сетевых задачах, а также в тех случаях, когда мы хотим проверить на истинность сразу несколько значений.

Ранее мы рассматривали пример с доступом пользователя в административный раздел сайта с помощью условного И:

```
bool showAdminPage = isAuthorized && isAdmin;
```

В реальных задачах мы можем столкнуться с необходимостью проверять большое число флагов для принятия решения. Допустим, мы хотим протестировать выпускников вуза по нескольким областям: знания алгоритмов, знание баз данных, знание C#, знание JS, способности в дизайне и умение работать в Git. После прохождения тестов мы хотим разбить всех студентов на три группы: backend-разработчики, frontend-разработчики и дизайнеры. Критерии отбора следующие:

1. Backend-разработчики должны обладать знаниями в области баз данных и алгоритмов, C# и Git.
2. Frontend-разработчики должны обладать знаниями в области алгоритмов, JS и Git.
3. Дизайнеры должны уметь работать в Git и разбираться в дизайне.

Если бы мы решали эту задачу на флагах, решение получилось бы громоздким:

```
bool isBackender = hasDBKnowledges && hasAlgorithmsKnowledges && hasCSharpKnowledges && hasGitKnowledges;

bool isFrontender = hasAlgorithmsKnowledges && hasJSKnowledges && hasGitKnowledges;

bool isDesigner = hasDesignKnowledges && hasGitKnowledges;
```

При этом для каждого студента нам бы пришлось хранить значение всех шести флагов — шесть булевых значений. Использование битовых масок облегчает такие сценарии.

Так как у нас шесть критериев, нам понадобится шесть бит информации. Каждый бит будет соответствовать определённому критерию. То есть, каждый критерий будет описан числом,

содержащим в двоичной записи только одну единицу (только один включённый бит). Сопоставим критерии и их двоичное представление:

Навык	Число (двоичное представление)						Число (десятичное представление)
Знание баз данных	0	0	0	0	0	1	1
Знание алгоритмов	0	0	0	0	1	0	2
Знание C#	0	0	0	1	0	0	4
Знание JS	0	0	1	0	0	0	8
Знание Git	0	1	0	0	0	0	16
Знания в дизайне	1	0	0	0	0	0	32

Обратите внимание, что каждому навыку соответствует степень двойки, что позволяет нам суммировать представления навыков, тем самым объединяя их.

Теперь попробуем с помощью битовых масок определить профессию студента, имеющего навыки в алгоритмах и базах данных, со знаниями JS, Git и C#. Для начала получим число, которое будет означать все эти навыки — возьмём шесть нулей и «включим» соответствующие биты:

0	1	1	1	1	1

Действительно, студент обладает всеми навыками, кроме знаний в дизайне. Теперь составим для наших профессий битовые маски:

Backend-разработчик (БД, алгоритмы, C#, Git):

0	1	0	1	1	1

Frontend-разработчик (алгоритмы, JS, Git):

0	1	1	0	1	0

Дизайнер (Git, дизайн):

1	1	0	0	0	0

Далее мы должны наложить каждую из этих масок (отсюда название) на навыки нашего студента. Здесь важно понять принцип отбора — в каждой профессии мы хотим иметь тех студентов, которые обладают требуемыми навыками. Требования — это единицы в маске профессии. Навыки студента — это единицы в его числовом представлении навыков. Если все единицы в требованиях профессии и в навыке студента совпадут, он подходит под эту профессию.

Если мы наложим маску профессии через оператор побитового И на навыки студента и в результате получим такие же единицы, что и в маске профессии, значит, студент подходит под требования профессии. Напишем небольшую программу:

```
int knowledges = 0b011111; // знания студента

// Маски профессий:
int backenderMask = 0b010111;
int frontenderMask = 0b011010;
int designerMask = 0b110000;

// Те навыки из каждой профессии, которые присутствуют у студента:
int backenderKnowledges = knowledges & backenderMask;
int frontenderKnowledges = knowledges & frontenderMask;
int designerKnowledges = knowledges & designerMask;

// Если навыки полностью совпали с маской, мы получим True, иначе False
Console.WriteLine(backenderKnowledges == backenderMask);
Console.WriteLine(frontenderKnowledges == frontenderMask);
Console.WriteLine(designerKnowledges == designerMask);
```

Поначалу может показаться, что решение на флагах было более простым, чем решение с помощью битовых масок, но представьте, что мы хотим расширить требования ко всем профессиям. В случае с флагами нам бы пришлось добавлять новый флаг и изменять все имеющиеся проверки, а второе решение потребует лишь обновления трёх битовых масок.

Побитовое ИЛИ

Рассмотрим применение побитового ИЛИ. После того, как студент успешно сдал определённый тест, нам нужно выставить соответствующий флаг в его навыках. Представим, что студент успешно прошёл первый тест — на знание C#. Так как тест первый по списку, у студента ещё нет проверенных навыков:

```
int knowledges = 0; // знания студента
```

За знания C# мы ранее взяли число `0b000100` (см. таблицу). Теперь нам следует выставить нужную единицу в навыках студента:

```
knowledges = knowledges | 0b000100;
```

Теперь навыки студента обозначаются числом `0b000100`.

Добавим студенту знания Git (`0b010000`):

```
knowledges = knowledges | 0b010000;
```

Теперь значение знаний студента содержит два включённых бита:

```
knowledges == 0b010100; // True
```

Оператор равенства

Оператор равенства (==) возвращает значение `true`, если его операнды равны. В противном случае возвращается значение `false`.

```
int two = 2;
two == 2; // true
2 == two; //true
two == 1; // false
```

Оператор неравенства

Оператор неравенства (`!=`) возвращает значение `true`, если его операнды не равны. В противном случае возвращается значение `false`:

```
int two = 2;
two != 2; // false
2 != two; //false
two != 1; // true
```

В сочетании с булевыми переменными оператор неравенства возвращает противоположное значение:

```
Console.WriteLine(!true); // false
Console.WriteLine(!false); // true
```

Операторы сравнения

Операторы сравнения работают с числовыми значениями и возвращают булевый результат:

```
Console.WriteLine(5 > 2); // true
Console.WriteLine(5 >= 5); // true
Console.WriteLine(5 < 2); // false
Console.WriteLine(5 <= 5); // true
int x = 5;
Console.WriteLine( 2 < x && x <= 10); // true
```

Оператор условия `if-else`

Оператор `if` выполняет ближайший блок кода в том случае, если описанное в операторе условие истинно. В противном случае выполняется блок `else` (при его наличии):

```
string username = Console.ReadLine();
if(username == "admin")
{
    Console.WriteLine("Администратор");
}
else
{
    Console.WriteLine("Пользователь");
}
```

Если в условии оператора `if-else` указывается булевая переменная, то её **не нужно** явно сравнивать с `true`/`false`:

```
string username = Console.ReadLine();
bool isAdmin = username == "admin";
if (isAdmin)
{
    Console.WriteLine("Администратор");
}
else
{
    Console.WriteLine("Пользователь");
}
```

В зависимости от ситуации, мы можем применять отрицание в условии оператора `if-else`:

```
string username = Console.ReadLine();
bool isAdmin = username == "admin";
if (!isAdmin)
{
    Console.WriteLine("Доступ разрешён только администратору");
}
else
{
```

```
    Console.WriteLine("Добро пожаловать!");
}
```

Напомним, что блок `else` — необязательный:

```
string username = Console.ReadLine();
bool isAdmin = username == "admin";
if (isAdmin)
{
    Console.WriteLine("Это сообщение увидит только администратор");
}
Console.WriteLine("Это сообщение увидит администратор и все пользователи");
```

Операторы `if-else` можно комбинировать несколько раз:

```
int i = Convert.ToInt32(Console.ReadLine());
if (i == 0)
{
    Console.WriteLine(0);
}
else if (i == 1)
{
    Console.WriteLine(1);
}
else if (i == 2)
{
    Console.WriteLine(2);
}
else
{
    Console.WriteLine("i > 2");
}
```

Для таких случаев предпочтительнее использовать оператор `switch-case`.

Оператор `switch-case`

Оператор `switch-case` выполняет блоки тех меток `case`, значение выражения которых совпадает со значением выражения, указанного в операторе `switch`:

```
int dayOfWeek = 2;
string dayTitle = "";
switch (dayOfWeek)
{
    case 0:
        dayTitle = "Понедельник";
```

```
        break;
    case 1:
        dayTitle = "Вторник";
        break;
    case 2:
        dayTitle = "Среда";
        break;
    case 3:
        dayTitle = "Четверг";
        break;
    case 4:
        dayTitle = "Пятница";
        break;
    case 5:
        dayTitle = "Суббота";
        break;
    case 6:
        dayTitle = "Воскресенье";
        break;
}

Console.WriteLine(dayTitle);
```

В этом примере у оператора указано выражение `dayofWeek`, а у меток указаны значения от 0 до 6. В результате будет выполнен блок с меткой `case` 2, т. к. значение выражения `dayOfWeek` равно двум.

Обратите внимание, что в конце каждого блока `case` обязательно должен стоять оператор `break`. Этот оператор завершает выполнение оператора, в котором он описан (в нашем случае это switch-case). По правилам языка C#, у оператора switch-case может выполняться только один блок с командами. В то же время, если метка не содержит блока с кодом, мы можем описать несколько меток над одним блоком кода:

```
int dayOfWeek = 5;
switch (dayOfWeek)
{
    case 0:
    case 1:
    case 2:
    case 3:
    case 4:
        Console.WriteLine("Будний день");
        break;
    case 5:
    case 6:
        Console.WriteLine("Выходной");
        break;
}
```

Оператор `switch-case` может включать в себя ветку по умолчанию (`default`), которая будет выполнена только в том случае, если ни одна из веток не выполнялась:

```
int dayOfWeek = 7;
switch (dayOfWeek)
{
    case 0:
    case 1:
    case 2:
    case 3:
    case 4:
        Console.WriteLine("Будний день");
        break;
    case 5:
    case 6:
        Console.WriteLine("Выходной");
        break;
    default:
        Console.WriteLine("Укажите значение от 0 до 6");
        break;
}
```

Перечисления

Перечисления (`enum`) — это тип значения, определённый набором именованных констант применяемого целочисленного типа. Пример перечисления времён года:

```
class Program
{
    enum Season
    {
        Spring,
        Summer,
        Autumn,
        Winter
    }

    static void Main(string[] args)
    {
        Season currentSeason = Season.Autumn;
    }
}
```

Значения перечислений имеют числовые представления, которые можно задать явно. Если числовое значение элемента не задано явно, то он представляется как нуль в том случае, если он является первым элементом перечисления. Если элемент не является первым, он будет представлен числовым значением предыдущего элемента + 1:

```
enum Season
{
    Spring, // = 0, т.к. это первый элемент, а значение не указано явно
    Summer, // = 1, т.к. предыдущий элемент представлен как 0
    Autumn = 30, // 30, т.к. значение указано явно
    Winter // = 31, т.к. предыдущий элемент представлен как 30
}
```

Применение перечислений

Битовые маски

Перечисления хорошо сочетаются с битовыми масками. Ранее мы уже применяли битовые маски для распределения студентов по будущим профессиям. Рассмотрим тот же пример, но с использованием перечислений и сравним удобство обоих вариантов:

```
using System;
namespace HelloWorld
{
    class Program
    {
        [Flags]
        public enum Knowledges
        {
            // Для читаемости разряды можно разделять знаком подчёркивания.
            // Они никак не влияют на значение.
            Database   = 0b_000001,
            Algorithms = 0b_0001_0,
            CSharp     = 0b_000100,
            JavaScript = 0b_001000,
            Git        = 0b_010000,
            Design     = 0b_100000,
        }

        static void Main(string[] args)
        {
            // Маски профессий
                        Knowledges backendRequirements = Knowledges.Database |
Knowledges.Algorithms | Knowledges.CSharp | Knowledges.Git;
                    Knowledges frontendRequirements = Knowledges.Algorithms |
Knowledges.JavaScript | Knowledges.Git;
            Knowledges designRequirements = Knowledges.Git | Knowledges.Design;
```

```
            //Знания студента в числовой записи
            Knowledges allKnowledges = (Knowledges)0b011111;

            // Те навыки из каждой профессии, которые присутствуют у студента:
                        Knowledges backenderKnowledges = allKnowledges &
backendRequirements;
                       Knowledges frontenderKnowledges = allKnowledges &
frontendRequirements;
            Knowledges designerKnowledges = allKnowledges & designRequirements;

            bool isBackender = backenderKnowledges == backendRequirements;
            bool isFrontender = frontenderKnowledges == frontendRequirements;
            bool isDesigner = designerKnowledges == designRequirements;

            Console.WriteLine($"Знания студента: {allKnowledges}");

            if (isBackender)
            {
                Console.WriteLine("Студент может стать backend-разработчиком");
            }

            if (isFrontender)
            {
                Console.WriteLine("Студент может стать frontend-разработчиком");
            }

            if (isDesigner)
            {
                Console.WriteLine("Студент может стать дизайнером");
            }
        }
    }
}
```

Использование перечислений освобождает нас от необходимости объявлять маски в двоичном представлении, достаточно один раз задать соответствующие значения элементам перечисления. При этом мы можем применять побитовые операторы к элементам перечисления, а также задавать их значение в числовом виде (см. `allKnowledges`).

Скопируйте и запустите программу. Обратите внимание, что благодаря атрибуту Flags знания студента при выводе на консоль отображаются в более удобном строковом виде:

```
Знания студента: Database, Algorithms, CSharp, JavaScript, Git
```

Без атрибута Flags будет выведено числовое представление. Мы вернёмся к атрибутам в следующих курсах.

switch-case

Перечисления часто применяются вместе с оператором `switch-case`. Если в качестве значения оператора указать переменную типа enum, то в качестве значения меток можно будет указывать элементы этого перечисления:

```
switch (DateTime.Today.DayOfWeek)
{
    case DayOfWeek.Monday:
    case DayOfWeek.Tuesday:
    case DayOfWeek.Wednesday:
    case DayOfWeek.Thursday:
    case DayOfWeek.Friday:
        Console.WriteLine("Сегодня будний день");
        break;
    case DayOfWeek.Saturday:
    case DayOfWeek.Sunday:
        Console.WriteLine("Сегодня выходной");
        break;
}
```

Преобразования типов

В языке C# после объявления переменной её нельзя объявить повторно или назначить ей значения другого типа, если этот тип невозможно неявно преобразовать в тип переменной. Например, `string` невозможно неявно преобразовать в `int`. Поэтому после объявления переменной типа `int` нельзя назначить ей строку.

Но иногда нам может потребоваться скопировать значение в переменную другого типа. Такого рода операции называются преобразованиями типа. В C# можно выполнять несколько видов преобразования.

Неявное преобразование

Неявное преобразование происходит для числовых типов в том случае, когда целевой тип содержит больший диапазон значений и большую точность. Например, мы можем неявно преобразовать тип `byte` в тип `int`, так как тип `int` имеет больший диапазон. Но неявное преобразование не сработает в обратную сторону, так как при переводе `int` в `byte` велика вероятность потери значений. То же самое касается вещественных типов данных — неявное преобразование возможно из `float` в

`double`, так как тип `double` имеет большую точность, но неявное преобразование `double` в `float` не сработает:

```
byte x = 200;
int a = x;
byte y = a; // Ошибка: Cannot implicitly convert type 'int' to 'byte'

float f = 1.23F;
double d = f;
float z = d; // Ошибка: Cannot implicitly convert type 'double' to 'float'
```

Явное преобразование

Для тех случаев, когда неявное преобразование повлечет за собой потерю данных, существует явное преобразование типов, при котором мы указываем перед исходной переменной выражение приведения — требуемый тип, указанный в круглых скобках:

```
int i = 12345;
byte b = (byte)i;
Console.WriteLine(b); // 57

double d = 3.14159265358979;
float f = (float)d;
Console.WriteLine(d);
Console.WriteLine(f); // 3,141593

i = (int)d;
Console.WriteLine(i); // 3
```

Переменная `b` имеет тип `byte`, следовательно может хранить значения в интервале `0..255`. Переменная `i` имеет тип `int` и его диапазон значения гораздо больше, чем может уместить в себя переменная с байтовым типом, поэтому преобразование может быть осуществлено только с явным указанием типа — мы даём компилятору понять, что готовы осуществить преобразование с возможной потерей данных. В результате в переменной `b` окажется результат операции `i % 256`, где 256 — это величина диапазона `byte`. Если значение переменной `i` попадало в интервал типа `byte`, то преобразование произошло без потерь.

Преобразование с использованием вспомогательных классов

Не во всех случаях типы, участвующие в преобразовании, могут быть совместимыми. До этого мы рассматривали примеры с числовыми типами. Если мы попробуем явно преобразовать тип `string` в тип `int`, то это приведёт к ошибке компиляции:

```
string s = "12345";
int i = (int)s; // Cannot convert type 'string' to 'int'
```

В таких случаях мы можем воспользоваться функциями стандартного класса `Convert`. Если строка содержит корректное числовое значение, то конвертация произойдет успешно:

```
string s = "12345";
int i = Convert.ToInt32(s);
Console.WriteLine(i); // 12345
```

У класса `Convert` есть ещё одно преимущество: он округляет значения при преобразовании чисел с плавающей запятой в тип `int` по математическим правилам, в то время как явное преобразование просто игнорирует дробную часть:

```
double d = 4.9;
int truncated = (int)d;
int converted = Convert.ToInt32(d);

Console.WriteLine(truncated); // 4
Console.WriteLine(converted); // 5
```

Порядок выполнения операций

В некоторых случаях нам бывает необходимо задать приоритет выполнения операций (так же, как и в математических операциях). Для этого в языке C# используются круглые скобки. Выражения, написанные внутри круглых скобок, выполняются в первую очередь, слева направо, а затем их значения подставляются в оставшуюся часть общего выражения:

```
int result = (2 + 2) * 2; // 8
```

Задание порядка выполнения операций бывает полезно в сложных выражениях. Вспомним, как мы решали задачу с определением профессии студента:

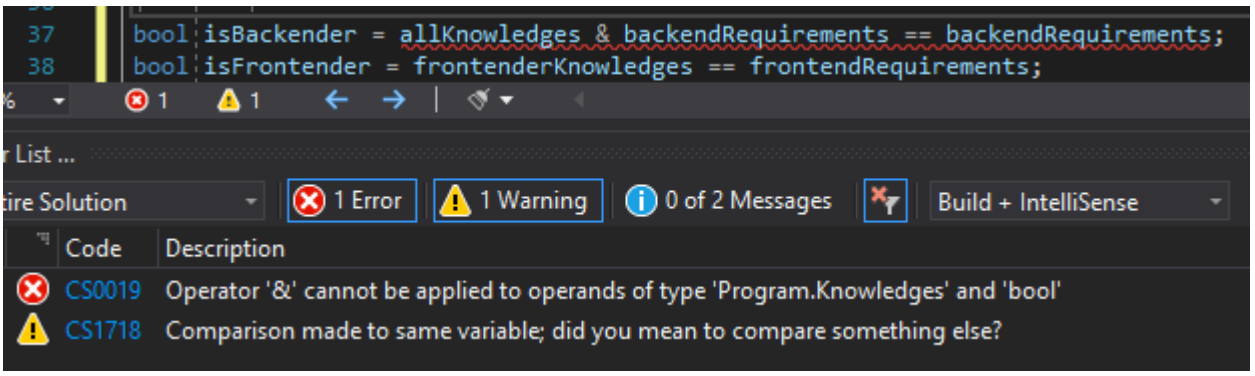
```
Knowledges backenderKnowledges = allKnowledges & backendRequirements;

bool isBackender = backenderKnowledges == backendRequirements;
```

Так как нигде далее мы не использовали переменную backenderKnowledges и подобные ей, перепишем эти строки в однострочное выражение:

```
bool isBackender = allKnowledges & backendRequirements == backendRequirements;
```

Теперь у нас нет промежуточных переменных, но без оператора приоритета этот код не будет корректным и не скомпилируется.

Дело в том, что по правилам приоритета, компилятор сначала сравнивает между собой переменную `backendRequirements` и `backendRequirements`, а затем пытается применить побитовое И между `allKnowledges` и результатом сравнения:

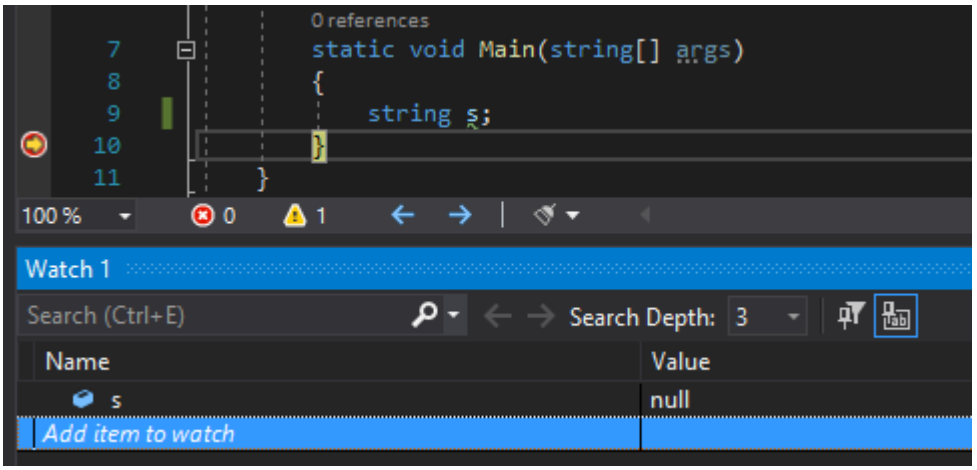


Чтобы вычисления стали корректными, нам нужно задать приоритет операциям. Сначала мы накладываем маску `backendRequirements` на знания студента, а уже затем сравниваем этот результат с маской:

```
bool isBackender = (allKnowledges & backendRequirements) == backendRequirements;
bool isFrontender = (allKnowledges & frontendRequirements) == frontendRequirements;
bool isDesigner = (allKnowledges & designRequirements) == designRequirements;
```

Значение `null`

`null` — это особое значение, которое будет значением по умолчанию для переменных со сложными типами. Проще всего понять применение этого значения — провести аналогию с заполнением анкеты: в ней уже есть своего рода переменные для фамилии, возраста и прочих данных. Но до тех пор, пока эти переменные не заполнены, они не хранят никакого значения, иначе говоря, они хранят `null`. Если мы объявим строку, но не зададим ей значение, она также будет хранить `null`:



Мы будем часто сталкиваться со значением `null` в следующих курсах.

Завершение программы оператором `return`

Оператор `return` позволяет прервать выполнение программы:

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            string login = Console.ReadLine();

            if (login != "admin")
            {
                Console.WriteLine("Доступ запрещен");
                return;
            }

            Console.WriteLine("Сообщение для администратора");
        }
    }
}
```

Если будет указан логин `admin`, то блок `if` не выполняется и мы увидим сообщение для администратора. В противном случае, мы увидим сообщение об отсутствии доступа, а затем оператор `return` завершает выполнение программы.

Практическое задание

Подготовить 4 проекта, которые будут выполнять следующие действия:

1. Запросить у пользователя минимальную и максимальную температуру за сутки и вывести среднесуточную температуру.
2. Запросить у пользователя порядковый номер текущего месяца и вывести его название.
3. Определить, является ли введённое пользователем число чётным.
4. Для полного закрепления понимания простых типов найдите любой чек, либо фотографию этого чека в интернете и схематично нарисуйте его в консоли, только за место динамических, по вашему мнению, данных (это может быть дата, название магазина, сумма покупок) подставляйте переменные, которые были заранее заготовлены до вывода на консоль.
5. (*) Если пользователь указал месяц из зимнего периода, а средняя температура > 0, вывести сообщение «Дождливая зима».
6. (*) Для полного закрепления битовых масок, попытайтесь создать универсальную структуру расписания недели, к примеру, чтобы описать работу какого либо офиса. Явный пример - офис номер 1 работает со вторника до пятницы, офис номер 2 работает с понедельника до воскресенья и выведите его на экран консоли.

Используемые источники

1. Справочник по C#. Целочисленные типы.
2. Числовые типы с плавающей запятой — справочник по C#.
3. Класс Math.
4. Справочник по C#. Типы перечислений.
5. Оператор switch C#.
6. DateTime Структура (System).
7. DateTime Format In C#.