

Введение в C#

Методы стандартных классов .NET



На этом уроке

1. Углубимся в тему работы с методами и типами .NET.
2. Изучим основные принципы выполнения методов в памяти компьютера.
3. Познакомимся с несколькими новыми операторами и ключевыми словами.

Оглавление

[На этом уроке](#)

[Функции и методы](#)

[Роль функций в программировании](#)

[Функции в объектно-ориентированном программировании](#)

[Объявление методов](#)

[Зачем разделять код на методы. Принцип DRY.](#)

[Стандартные методы классов](#)

[Дополнительные операторы языка](#)

[Тернарный оператор](#)

[Память в C#](#)

[Stack](#)

[Heap](#)

[Категории типов](#)

[Сравнение переменных с типами ValueType и ReferenceType](#)

[Особенности сравнения переменных типа String](#)

[Упаковка и распаковка ValueType](#)

[Ключевое слово ref](#)

[Передача аргументов по ссылке](#)

[Возвращение ссылок на значения](#)

[Ключевое слово out](#)

[Ключевое слово params](#)

[Рекурсивный вызов метода](#)

[Анализ стека при помощи точек останова](#)

[Домашние задания](#)

[Используемые источники](#)

Функции и методы

Роль функций в программировании

Функция — это фрагмент программного кода, к которому можно обратиться из другого места программы. Из определения можно сделать вывод, что в функции выносятся операции, которые должны быть доступны из разных мест программы. Функция может содержать входные аргументы, с помощью которых мы можем изменять её поведение, а также передавать ей необходимые значения. Также функция может содержать результирующее значение.

Функции в программировании решают ряд задач:

1. Позволяют разбить сложную программу на набор маленьких функций, с которыми проще взаимодействовать (вносить правки и разбираться в их работе).
2. Позволяют переиспользовать код и избежать излишнего дублирования кода (о дублировании кода мы поговорим далее в этом уроке).

Функции в объектно-ориентированном программировании

Ранее мы старались избегать терминов «класс», «объект» и «объектно-ориентированное программирование» — ООП. Определение ООП будет дано в следующих курсах, после того, как мы разберёмся с основами языка. Но так как C# — объектно-ориентированный язык, нам необходимо разобраться с некоторыми базовыми терминами из мира объектно-ориентированных языков. Это поверхностные определения, достаточные, чтобы мы могли двигаться вперед.

В объектно-ориентированном языке функции описываются внутри классов и называются [методами](#). Класс — зачастую отражение какой-либо модели из реального мира. Например, класс `Dog` может содержать метод `Bark` (лаять), а класс `Calc` может описывать методы `Add/Subtract/Multiply/Divide`.

Ранее мы называли методы функциями для упрощения. Далее мы будем называть их методами, как это принято в языке C#.

Объявление методов

Ранее мы рассматривали только метод `Main` (точку входа в приложение). Также мы использовали методы класса `Convert` в уроке №2:

```
int i = Convert.ToInt32(Console.ReadLine());
```

Здесь у класса `Convert` вызывается метод `ToInt32`. Метод принимает на вход строковый аргумент (результат выполнения метода `Console.ReadLine`) и отдает результат в виде целого числа.

Наряду с использованием стандартных методов, можно также объявлять свои. Рассмотрим применение простейших методов:

```
static void Main(string[] args)
{
    string name = GetUserName();
    string greeting = MakeGreeting(name);
    Console.WriteLine(greeting);
}

static string GetUserName()
{
    Console.WriteLine("Привет! Как тебя зовут?");
    string userName = Console.ReadLine();
    return userName;
}

static string MakeGreeting(string userName)
{
    return $"Привет, {userName}!";
}
```

Этот фрагмент кода содержит для нас сразу несколько нововведений. Разберёмся с ними по очереди. Код начинается с привычного нам метода `Main`, в котором мы получаем имя пользователя путём вызова метода `GetUserName`. Далее мы передаём имя пользователя в метод `MakeGreeting`, возвращающий персонализированное приветствие, которое мы потом выводим на экран.

Далее мы видим описание методов `GetUserName` и `MakeGreeting`. Так же, как и метод `Main`, объявление методов начинается с ключевого слова `static` (будет рассмотрено в следующих курсах). Затем следует указание типа возвращаемого значения. В обоих случаях это тип `string`. Далее мы указываем имя метода и перечисляем входные аргументы в круглых скобках. Возвращаемое значение, имя и набор аргументов называют сигнатурой метода. Сигнатура метода должна быть уникальной в пределах класса.

Код обоих методов достаточно простой и состоит из уже знакомых нам операций. Разница заключается в поведении оператора `return`, который в данном случае не завершает работу программы, а возвращает значение в вызывающий метод (в данном случае, в метод `Main`). Важно, чтобы тип возвращаемого значения совпадал с типом, указанным в объявлении метода. В противном случае, произойдет ошибка сборки, а в некоторых случаях - ошибка во время выполнения программы.

Мы можем возвращать как значение переменной, так и результат выполнения выражения. Например, метод `GetUserName` можно переписать так:

```
static string GetUserName()
{
    Console.WriteLine("Привет! Как тебя зовут?");
```

```
        return Console.ReadLine();
    }
```

Поведение метода при этом никак не изменится.

Метод может возвращать значение. В том случае, когда метод ничего не возвращает, а просто выполняет набор операций, в качестве типа возвращаемого значения следует указывать `void`:

```
static void ShowGreeting(string greeting)
{
    Console.WriteLine("Hello, world!");
}
```

Бывают ситуации, когда из метода следует вернуть несколько значений. На практике в таких случаях применяют классы, но также можно воспользоваться возможностью языка C# под названием `named tuple` (именованный кортеж):

```
static void Main(string[] args)
{
    (string userName, int userAge) = AskUserInfo();
}

static (string name, int age) AskUserInfo()
{
    Console.WriteLine("Как тебя зовут?");
    string name = Console.ReadLine();
    Console.WriteLine($"Приятно познакомиться, {name}! Сколько тебе лет?");
    int age = Convert.ToInt32(Console.ReadLine());
    return (name, age);
}
```

Метод `AskUserInfo` возвращает кортеж значений `string` и `int`. Обратите внимание, что названия переменных, объявленных в теле метода или в вызываемой функции, не обязательно должны совпадать с теми, что указаны в сигнатуре метода.

Зачем разделять код на методы. Принцип DRY.

Взглянем на программу по созданию пользователей, в которой весь код написан в методе `Main`:

```
static void Main(string[] args)
{
    (string name, int age)[] users;
    Console.WriteLine("Введите количество пользователей:");
    int count = Convert.ToInt32(Console.ReadLine());
    users = new (string name, int age)[count];
}
```

```

for (int i = 0; i < users.Length; i++)
{
    Console.Write("Введите имя пользователя: ");
    string userName = Console.ReadLine();
    Console.Write("Введите возраст пользователя: ");
    int userAge = Convert.ToInt32(Console.ReadLine());
    users[i] = (userName, userAge);
    Console.WriteLine($"Пользователь {userName} ({userAge}) сохранен");
}

Console.WriteLine("Ввод данных завершён. Нажмите любую клавишу...");
Console.ReadKey();
Console.Clear();
int choice;
do
{
    Console.WriteLine("0 - Завершение работы");
    Console.WriteLine("1 - Просмотр всей базы данных");
    Console.WriteLine("2 - Просмотр пользователя");
    choice = Convert.ToInt32(Console.ReadLine());
    if (choice == 1)
    {
        Console.WriteLine("Вывод базы данных:");
        for (int i = 0; i < users.Length; i++)
        {
            Console.WriteLine($"{users[i].name} ({users[i].age})");
        }
    }
    if (choice == 2)
    {
        int userIndex;
        do
        {
            Console.WriteLine($"Введите идентификатор пользователя - от 0 до {users.Length - 1}");
            userIndex = Convert.ToInt32(Console.ReadLine());
        } while (userIndex < 0 || userIndex >= users.Length);

        Console.WriteLine($"{users[userIndex].name} ({users[userIndex].age})");
    }
} while (choice != 0);
}

```

В этой программе менее 50 строчек кода, что довольно мало, если сравнивать с реальными приложениями. При этом программа довольно примитивна, она заполняет массив значений и выводит их на экран. Но даже такая простая программа кажется излишне загромождённой. Её код не структурирован и нам сложно с первого взгляда понять, что он делает. Более того, нам приходится каждый раз писать один и тот же набор операций для вывода данных пользователя на консоль, а

также для преобразования введённой строки в число. Это вносит дополнительную сложность и затрудняет чтение и редактирование кода. Попробуем структурировать код с помощью методов:

```
static void Main()
{
    (string name, int age)[] users = CreateUsers();
    ShowMenu(users);
}

static (string name, int age)[] CreateUsers()
{
    Console.WriteLine("Введите количество пользователей:");
    int count = ReadInt();
    (string name, int age)[] users = new (string name, int age)[count];
    for (int i = 0; i < users.Length; i++)
    {
        users[i] = CreateUser();
        Console.WriteLine($"Пользователь {FormatUserData(users[i])} сохранен");
    }

    Console.WriteLine("Ввод данных завершён. Нажмите любую клавишу...");
    Console.ReadKey();
    Console.Clear();

    return users;
}

static void ShowMenu((string name, int age)[] users)
{
    int choice;
    do
    {
        choice = GetUserChoice();
        switch (choice)
        {
            case 0: return;
            case 1:
                PrintUsers(users);
                break;
            case 2:
                PrintSelectedUser(users);
                break;
            default:
                break;
        }
    } while (choice != 0);
}

static (string userName, int userAge) CreateUser()
{
    Console.Write("Введите имя пользователя: ");
    string userName = Console.ReadLine();
}
```

```

        Console.WriteLine("Введите возраст пользователя: ");
        int userAge = ReadInt();
        return (userName, userAge);
    }
    static int GetUserChoice()
    {
        Console.WriteLine("0 - Завершение работы");
        Console.WriteLine("1 - Просмотр всей базы данных");
        Console.WriteLine("2 - Просмотр пользователя");
        return ReadInt();
    }

    static void PrintSelectedUser((string name, int age)[] users)
    {
        int userIndex;
        do
        {
            Console.WriteLine($"Введите идентификатор пользователя - от 0 до
{users.Length - 1}");
            userIndex = ReadInt();
        } while (userIndex < 0 || userIndex >= users.Length);

        PrintUser(users[userIndex]);
    }

    static void PrintUsers((string name, int age)[] users)
    {
        Console.WriteLine("Вывод базы данных:");
        for (int i = 0; i < users.Length; i++)
        {
            PrintUser(users[i]);
        }
    }

    static void PrintUser((string name, int age) user)
    {
        Console.WriteLine(FormatUserData(user));
    }

    static string FormatUserData((string name, int age) user)
    {
        return $"{user.name} ({user.age})";
    }
    static int ReadInt()
    {
        return Convert.ToInt32(Console.ReadLine());
    }
}

```

Методы позволяют дать подсказки другим разработчикам — они выделяют небольшие куски кода в отдельную область и позволяют озаглавить её. При чтении структурированного кода нам не нужно читать его с начала и до конца так, как это делает компьютер. Люди проще ориентируются в

информации, когда она связана и структурирована. Так ли легко отыскать логику отображения меню в первом примере кода? Даже зная, с какой строки начинается эта логика, нам каждый раз придётся искать её глазами по всему коду. Где находится логика вывода меню во втором примере кода? В методе `ShowMenu`. Если нам нужно внести изменение в отображение меню, мы можем абстрагироваться от других частей программы и сконцентрироваться на этом методе. При чтении метода `ShowMenu` мы обнаружим для себя два новых метода: `PrintUsers` и `PrintSelectedUser`. В зависимости от поставленной задачи, мы можем углубиться в один из этих методов и изучить его более подробно.

Внимание! Чтобы перейти к определению метода, который используется в текущей строке, наведите на него каретку (или кликните мышкой) и нажмите `F12`. Например, можно навестись на название метода `GetUserChoice` в данной строке (IDE подсветит название метода):

```
choice = GetUserChoice();
```

и после нажатия `F12` мы переместимся ниже к описанию метода.

В первом варианте программы мы часто запрашивали числовые значения у пользователя — считывали строку из консоли и потом конвертировали в число. Во втором варианте мы вынесли эту операцию в отдельный метод `ReadInt`. Также мы вынесли в отдельный метод форматирование данных пользователя. Эти методы содержат мало кода и может показаться, что в них нет большого смысла. Но представим, что нам нужно изменить форматирование данных пользователя. Сейчас мы выводим на экран его имя и добавляем возраст в круглых скобках. Как только мы изменим это поведение внутри метода `FormatUserData`, оно изменится везде, где использовался этот метод. В предыдущем варианте программы нам бы пришлось вносить изменения во все строковые выражения в исходном коде. Это не только утомительно, но и может привести к ошибкам. Что если наше приложение состоит из множества файлов, в которых производится форматированный вывод данных пользователя? Мы будем вынуждены изменять все эти вхождения вручную, либо придумывать хитрые команды поиска и замены. Во втором варианте приложение лишено этого недостатка.

В программировании существует принцип **DRY (Don't Repeat Yourself)**, который сводится к тому, что каждая единица знания должна быть описана в коде единожды. Например: только один метод должен знать, как форматировать данные пользователя, а все остальные части приложения должны использовать этот метод.

В процессе разбиения кода стоит помнить также и о сохранении его простоты. Не стоит злоупотреблять разбиением на методы, которые используются лишь в одном месте, а также не стоит создавать методы, которые не востребованы прямо сейчас. Существует принцип **KISS (Keep it simple stupid)**, который призывает делать системы настолько простыми, насколько это возможно.

Стандартные методы классов

В .NET существует класс `Object`, базовый для всех классов .NET. Это означает, что любой другой класс обладает теми же методами, что и класс `Object`. Например, класс `Object` определяет метод `ToString` для строкового представления содержимого объекта. Когда мы выводим на консоль объекты, отличные от строковых, у них вызывается метод `ToString`. Он всегда определен в любом объекте, так как описан в `Object`:

```
class Program
{
    enum Numbers
    {
        Five = 5
    }
    static void Main(string[] args)
    {
        object[] objects = { 1, 2, "3", "Четыре", Numbers.Five, false };
        for (int i = 0; i < objects.Length; i++)
        {
            Console.WriteLine(objects[i].ToString());
        }
    }
}
```

Дополнительные операторы языка

Тернарный оператор

Вспомним пример, в котором мы определяли роль пользователя:

```
static void Main(string[] args)
{
    string username = Console.ReadLine();
    string role = GetUserRole(username);
    Console.WriteLine(role);
}

static string GetUserRole(string username)
{
    if (username == "admin")
    {
        return "Администратор";
    }
    else
    {
        return "Пользователь";
    }
}
```

```
}
```

Рассмотрим подробнее метод определения роли пользователя. В нём есть зависимость между входными и выходными значениями. В таких случаях мы можем упростить код с помощью тернарного оператора.

Тернарный оператор работает так же, как и оператор `if-else`. Название оператора означает, что он обладает тремя операндами. Если значение первого операнда — истина, возвращается значение и выполняется второй операнд. В противном случае выполняется третий операнд. При этом важно помнить, что типы второго и третьего операндов должны совпадать. Отличие от `if-else` в том, что тернарный оператор может содержать только по одному выражению во втором и третьем операнде, в то время как в операторе `if-else` можно указать блок операций. Результат выполнения тернарного оператора можно присвоить переменной или вернуть как результат выполнения метода.

Запись тернарного оператора выглядит так:

```
<условие> ? <операнд2> : <операнд3>
```

Метод `GetUserRole` можно переписать следующим образом:

```
static string GetUserRole(string username)
{
    return username == "admin" ? "Администратор" : "Пользователь";
}
```

При всех преимуществах у тернарного оператора есть недостаток — неудобство отладки. При использовании оператора `if-else` мы можем установить точку останова на любой строке, но так как тернарный оператор по сути — одна операция, мы не можем установить точку останова на какой-либо из его операндов. Поэтому стоит применять тернарный оператор для возвращения той или иной переменной, избегая сложных конструкций в операндах.

Память в C#

Stack

Стек (*stack*) — специальная область памяти компьютера, в которой хранятся локальные переменные, созданные каждым методом (включая `Main`). Название этой области происходит из-за использования одноименной структуры данных. Стек — это структура данных, организованная по принципу LIFO (*last in, first out*). Дословно стек переводится как стопка. Как и в стопке книг, первой сверху будет та,

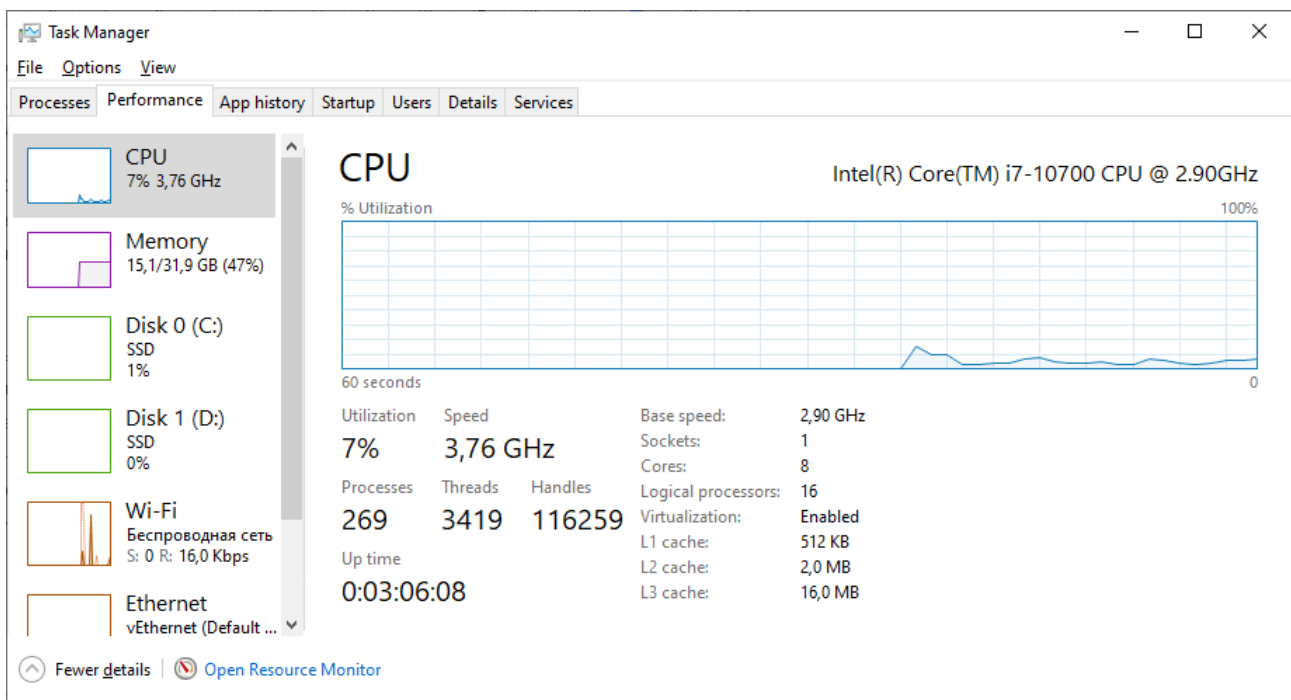
которую мы положили туда последней. Также и стек — те данные, которые попали в него последними, будут считаны в первую очередь. Стек можно считывать только сверху.

Стек состоит из кадров (stack frame). Каждый раз, когда мы вызываем какой-нибудь метод, в области стека создается новый стековый кадр, в который по мере выполнения метода будут помещаться локальные переменные, объявленные внутри этого метода. Затем каждый раз, когда метод завершает работу, стековый кадр удаляется из памяти, а следовательно, и все переменные, объявленные внутри метода. Как только стековый кадр удаляется, эта область памяти становится доступной для других стековых кадров. Соответственно, если мы вызываем метод внутри другого метода и так далее, то под каждый создаётся отдельный стековый кадр, который удаляется после завершения метода.

Важно помнить, что в C# размер стека ограничен. Независимо от того, сколько оперативной памяти установлено в компьютер, размер стека составляет от одного до нескольких мегабайт. Конкретный размер зависит от множества факторов, в том числе от разрядности приложения и версии операционной системы. Обычно для 32-битных приложений размер стека составляет 1 Мб, а для 64-битных — 4 Мб. В некоторых случаях размер стека можно исправить вручную, поменяв значение в заголовке бинарного файла, но это довольно редкая практика. Эти значения могут показаться небольшими, но стоит помнить, что стек хранит только переменные и аргументы методов, выполняемых в данный момент. Когда стек заполняется, происходит ошибка переполнения стека и программа экстренно завершает работу. Мы не сможем каким-либо образом обработать такую ситуацию (например, написать отдельный метод-обработчик таких ситуаций, который сохранит важные данные на диск). Так как стек переполнен, компьютер физически не может выделить в нем место для вызова ещё одного метода. Далее в этом уроке мы подробнее разберём работу со стеком и отладку приложений.

После знакомства со стеком может показаться, что его наличие нецелесообразно. Казалось бы, у современных компьютеров есть десятки гигабайт оперативной памяти, но по каким-то причинам нам приходится оперировать несколькими мегабайтами. Рассмотрим некоторые из них.

Одна из причин использования ограниченного участка памяти в качестве стека — многозадачность современных процессоров. На компьютере могут быть запущены сотни программ. Например, в момент написания методички на компьютере с запущенным браузером, парой мессенджеров и IDE выполняется 269 процессов:



Если бы размер стека не был фиксированным и малым, то можно представить ситуацию, когда один процесс выделит настолько большой участок памяти под стек, что другие просто не смогут запуститься.

Также наличие стека даёт гарантии разработчику, что после запуска приложению будет доступно пространство в памяти в соответствии с размером стека. Это спасает нас от ситуации, когда из-за нехватки памяти приложение экстренно завершится в момент выполнения. Если на компьютере недостаточно памяти под стек программы, она попросту не запустится изначально.

Heap

Стоит отметить, что, кроме стека, приложению доступна и другая категория памяти, называемая кучей (Heap). Название связано с тем, что память — динамическая и не делится на какие-либо размеры. Когда какому-либо приложению требуется память, запрашиваемый размер памяти выделяется под данное приложение из общей кучи. В отличие от стека, в котором кадры расположены друг за другом, в куче нет строгости в порядке выделения памяти. Мы просто запрашиваем определённый объём памяти и получаем его в каком-либо участке оперативки. В отличие от стека, у нас нет каких-либо ограничений на запрашиваемый объём динамической памяти, разумеется, кроме объёма свободной оперативной памяти.

Чаще всего в динамической памяти хранятся прикладные ресурсы приложения. Например, компьютерные игры хранят текстуры в памяти для быстрого действия, проигрыватель мультимедиа хранит в памяти ту часть файла, которая воспроизводится в данный момент, а также следующие несколько секунд.

В управляемых языках (таких, как C#) в динамической памяти автоматически размещаются некоторые типы данных.

Категории типов

В C# существует две категории типов — ссылочные типы (reference type) и типы значений (value type). Из известных нам типов данных, типы `object` и `string` — ссылочные. А все числовые типы данных, а также `enum`, `bool` и `char` — типы значений. Так как массивы — производные от класса `Object`, то они тоже имеют ссылочный тип данных.

У типов значений небольшой размер, который известен компилятору на момент сборки и, как правило, они хранятся в стеке. То есть при заполнении стека значениями определённого метода, все объявленные в этом методе переменные `ValueType` напрямую содержат свои значения. Существует возможность разместить типы значений в динамической памяти и мы рассмотрим её немного позднее.

Ссылочные типы, такие как строки или массивы, потенциально могут хранить большой объём данных, превышающий размер стека. Поэтому такие данные автоматически размещаются в куче. При заполнении стека, переменные с ссылочным типом хранят адрес памяти, с которого начинается размещение самих данных.

Для визуализации работы стека и кучи представим, как будет выполняться такая программа:

```
class Program
{
    static void Main()
    {
        int answer = 42;
        PrintSomething();
        Console.WriteLine(answer);
    }

    static void PrintSomething()
    {
        int number = 5;
        string greeting = "Hello";
        Console.WriteLine(number);
        Console.WriteLine(greeting);
    }
}
```

Выполнение начнётся с точки входа, в стеке разместится стек-фрейм для функции `Main`. В этом фрейме будет лежать переменная `answer`, и так как она обладает типом `ValueType`, в стеке напрямую будет лежать значение 42.

Затем в стеке разместится стек-фрейм для метода `PrintSomething`. В этом фрейме будет лежать переменная `number` (`ValueType`) со значением 5 и переменная `greeting` (`ReferenceType`) с каким-либо числом — адресом в памяти, обратившись по которому, мы увидим последовательно размещённые символы `H,e,l,l,o`.

Следующие две строки — вызовы методов `Console.WriteLine`. Под каждый из этих вызовов также будут созданы фреймы в стеке. В первый фрейм попадет числовой аргумент, а следовательно, во фрейме будет лежать число 5. Во втором фрейме будет лежать адрес из переменной `greeting`.

После выполнения всего метода `PrintSomething` мы снова вернёмся во фрейм функции `Main`, а значит, во фрейме будет храниться значение переменной `answer`, равное 42.

Сравнение переменных с типами `ValueType` и `ReferenceType`

Сравнение `ValueType` происходит по их значениям:

```
int a = 2;
byte b = 0b10;
double c = 2.0;
Console.WriteLine(a == b ); // true
Console.WriteLine(a == c); // true
Console.WriteLine(b == c); // true
```

При сравнении `ReferenceType` сравниваются сами ссылки, а не значения, расположенные по ним:

```
int[] a = { 1, 2, 3 };
int[] b = { 1, 2, 3 };

int[] linkToA = a;

Console.WriteLine(a == linkToA); // True
Console.WriteLine(a == b); // False
Console.WriteLine(b == linkToA); // False
```

Стоит помнить, что на одно и то же значение в памяти могут указывать несколько ссылочных переменных. Изменить значение можно через любую переменную:

```
int[] a = { 1, 2, 3 };
int[] linkToA = a;
linkToA[0] = 10;

Console.WriteLine(a[0]); // 10
```

Особенности сравнения переменных типа String

Как упоминалось ранее, `string` — ссылочный тип данных, но, в свою очередь, строковый тип имеет некоторые нюансы реализации, которые отражаются на поведении строковых переменных в операциях сравнения.

Ранее мы рассмотрели пример, в котором два идентичных массива не будут равны друг другу, так как они ссылаются на разные участки в памяти:

```
int[] a = { 1, 2, 3 };  
int[] b = { 1, 2, 3 };  
  
Console.WriteLine(a == b); // False
```

Может показаться, что строковые переменные должны вести себя таким же образом:

```
string first = "123";  
string second = "123";  
string third = "1" + "2" + "3";  
  
Console.WriteLine(first == second);  
Console.WriteLine(first == third);  
Console.WriteLine(second == third);
```

Но несмотря на ссылочную природу типа `string`, все три сравнения будут истинными. Все дело в оптимизационном процессе компилятора, который называется интернированием. На этапе сборки программного кода, компилятор создает пул (своего рода базу) всех встречающихся в коде строк. При этом, если строка будет результатом какого-либо простого выражения (например, конкатенацией строк, как в переменной `third`), то такие выражения вычисляются на месте и их значение также попадает в пул интернированных строк. Особенность пула строк — то, что каждая строка в нём уникальна, и все переменные, имеющие такое значение, указывают на адрес этой строки в памяти.

Во время компиляции вышеуказанного примера, компилятор встречает переменную `first` со значением `123` и размещает это значение в пуле. Переменная `first` будет указывать на адрес строки. Далее компилятор встречает переменную `second`, и так как в пуле уже есть значение `123`, переменная `second` будет указывать на адрес уже имеющейся в пуле строки. После этого компилятор переходит к переменной `third`. Её значение — результат простой строковой операции, поэтому компилятор вычисляет это значение (оно равно `123`) и также пытается разместить его в пуле уникальных строк. Но так как в пуле уже есть значение `123`, переменная `third` будет указывать точно на тот же адрес, что и две предыдущие. В результате во время выполнения программы все три переменные указывают на один и тот же участок в памяти, поэтому при сравнении все эти переменные равны друг другу.

Зная, как хранятся строки в памяти программы, решим небольшую задачу:

```
string name1 = "John";

string name2 = name1;

name1 = "Now I'm Bob!";

Console.WriteLine(name2);
```

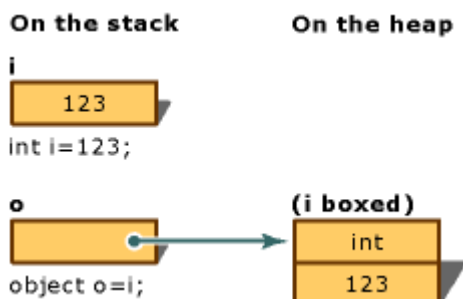
Попробуем разобраться, что выведется на консоль. Во время компиляции в пул интернирования попадают две строки — `John` и `Now I'm Bob!`. Во время выполнения, `name1` ссылается на строку `John`. Затем переменной `name2` присваивается то же значение, что и `name1`, то есть она также ссылается на строку `John`. Затем, мы присваиваем переменной `name1` адрес строки `Now I'm Bob!`. Заметьте, мы не меняем значения в памяти, а просто указываем переменной новый адрес. Следовательно, `name2` никак не изменяется и в консоль выводится значение `John`.

Упаковка и распаковка `ValueType`

В некоторых случаях нам может понадобиться обрабатывать типы значений наряду со ссылочными типами. В таких случаях мы можем применить к ним операцию упаковки — процесс преобразования типа значения в тип `object`:

```
int i = 123;
object o = i;
```

Так как `object` — ссылочный тип, в результате упаковки значение исходной переменной копируется из стека в кучу, а её адрес присваивается переменной `o`:



[Источник](#).

В этом примере значения массива — различные типы, приведённые к `object`:

```
object[] values = { 1, 2, "3", Values.Four };
```

```
for (int i = 0; i < values.Length; i++)
{
    Console.WriteLine(values[i].ToString());
}
```

Метод `ToString` описан в классе `Object` и поэтому доступен у всех элементов массива.

Для возвращения упакованных значений к исходному типу необходимо произвести распаковку. При этом значение будет скопировано из кучи в стек, так как целевой тип — `ValueType`. Внешне распаковка выглядит как обычное приведение типов:

```
int i = 1; // в стеке
object o = i; // в куче
int one = (int)o; // в стеке
```

Стоит понимать, что распаковка работает иначе, чем приведение типов. Необходимо распаковывать значение в конкретный тип. Например, нельзя упаковать числовое значение `int`, а затем распаковать его как `long`. При упаковке в объект информация об исходном типе сохраняется в объекте (это можно увидеть при отладке приложения). И если попытаться распаковать значение в отличный от исходного тип, произойдет ошибка времени выполнения.

Ключевое слово `ref`

Ключевое слово `ref` применяется, когда нам нужно передать или изменить значение по ссылке. Рассмотрим наиболее распространённые способы использования ключевого слова.

Передача аргументов по ссылке

В примере мы используем два метода: `Increment` и `IncrementByRef`.

В случае с `Increment` значение аргумента копируется в стек-фрейм по значению, а значит, при инкременте значения внутри метода значение исходной переменной (`number`) никак не изменяется.

В случае с `IncrementByRef` значение аргумента передаётся по ссылке, то есть аргумент `value` указывает на то же число, что и переменная `number`. В результате инкремента значение переменной `number` также инкрементируется:

```
static void Main(string[] args)
{
    int number = 1;

    int incremented = Increment(number);
}
```

```

        Console.WriteLine($"Исходное значение: {number}. Новое значение: {incremented}");

        int incrementedByRef = IncrementByRef(ref number);
        Console.WriteLine($"Исходное значение: {number}. Новое значение: {incrementedByRef}");
    }

    static int Increment(int value)
    {
        return ++value;
    }

    static int IncrementByRef(ref int value)
    {
        return ++value;
    }

```

Результат работы программы:

```

Исходное значение: 1. Новое значение: 2
Исходное значение: 2. Новое значение: 2

```

При использовании ключевого слова `ref` с переменными ссылочного типа мы получаем возможность изменять значение в памяти по адресу, хранящемуся в переменной:

```

static void Main(string[] args)
{
    string[] array = { "One", "Two", "Three" };
    string[] replacedByRef = ReplaceArrayByRef(ref array);
    string[] replaced = ReplaceArray(array);

    Console.WriteLine("array: ");
    for (int i = 0; i < array.Length; i++)
    {
        Console.Write($"{array[i]} ");
    }
    Console.WriteLine();

    Console.WriteLine("replaced: ");
    for (int i = 0; i < replaced.Length; i++)
    {
        Console.Write($"{replaced[i]} ");
    }
    Console.WriteLine();

    Console.WriteLine("replacedByRef: ");
    for (int i = 0; i < replacedByRef.Length; i++)
    {

```

```

        Console.Write($"{replacedByRef[i]} ");
    }
    Console.WriteLine();
}

static string[] ReplaceArray(string[] data)
{
    data = new[] { "replaced", "replaced", "replaced" };
    return data;
}

static string[] ReplaceArrayByRef(ref string[] data)
{
    data = new[] { "ref", "ref" , "ref"};
    return data;
}

```

В случае с `ReplaceArray` мы создаём новый массив и возвращаем его. Переменная `array` всё ещё указывает на массив `{ "One", "Two", "Three" }`, несмотря на то, что внутри `ReplaceArray` мы изменили значение аргумента `data`.

В случае с `ReplaceArrayByRef` мы передаём массив по ссылке, поэтому операции присваивания применяются также и к массиву `array`, на который ссылается аргумент `data`. Поэтому теперь переменная `array` указывает на тот же массив, что и `data` — `{ "ref", "ref", "ref" }`.

Результат работы программы:

```

array:
ref ref ref

replaced:
replaced replaced replaced

replacedByRef:
ref ref ref

```

Возвращение ссылок на значения

По аналогии с тем, как мы можем передавать в качестве аргументов методов ссылки на значения, мы также можем возвращать ссылки на значения вместо самих значений. Рассмотрим механизм на примере:

```

static void Main(string[] args)
{
    int[] numbers = { 1, 2, 3, -4, -7, 12, 24 };
    ref int min = ref GetMin(numbers);
}

```

```

    min = -1 * min;
    for (int i = 0; i < numbers.Length; i++)
    {
        Console.WriteLine(numbers[i]);
    }
}

static ref int GetMin(int[] data)
{
    ref int minValue = ref data[0];
    for (int i = 0; i < data.Length; i++)
    {
        if (data[i] < minValue)
        {
            minValue = ref data[i];
        }
    }

    return ref minValue;
}

```

Метод `GetMin` возвращает ссылку на минимальное значение в переданном массиве. В результате переменная `min` хранит не самое минимальное значение, а ссылку на какой-либо элемент массива `numbers`, поэтому присвоение этой переменной значения на самом деле меняет значение элемента массива. Элемент с минимальным значением (-7) сменит знак.

Результат работы программы:

```

1
2
3
-4
7
12
24

```

Ключевое слово `out`

Ключевое слово `out` схоже по использованию с ключевым словом `ref`, за исключением того, что при использовании `ref` перед передачей переменную необходимо инициализировать. Один из вариантов использования `out` — передача наружу дополнительных сведений для булевых методов:

```

static void Main(string[] args)
{
    string login = Console.ReadLine();
    if (Authorize(login, out string error))

```

```

    {
        Console.WriteLine("Вы успешно авторизованы");
    }
    else
    {
        Console.WriteLine(error);
    }
}

static bool Authorize(string login, out string error)
{
    if (login != "admin")
    {
        error = "Доступ разрешен только администратору";
        return false;
    }

    error = "";
    return true;
}

```

Обратите внимание, что внутри метода необходимо обязательно задавать возвращаемое значение аргументов с ключевым словом `out`.

Ключевое слово `params`

Ключевое слово `params` позволяет методу принимать массив однотипных аргументов, при этом во время вызова метода мы будем указывать значения через запятую так, как если бы это были отдельные аргументы:

```

static void Main(string[] args)
{
    int sum = Sum(1, 2, 3, 4, 5, 6, 7, 8, 9);
    Console.WriteLine(sum);
}

static int Sum(params int[] values)
{
    int sum = 0;
    for (int i = 0; i < values.Length; i++)
    {
        sum += values[i];
    }
    return sum;
}

```

Рекурсивный вызов метода

Рекурсия — это вызов метода внутри тела этого же метода. Рекурсивный метод — это метод, содержащий рекурсию.

Рекурсивный метод состоит из двух наиболее важных частей — рекурсивного вызова и терминального условия — условия, при наступлении которого происходит выход из метода без очередного рекурсивного вызова. Если указать терминальное условие, которое никогда не наступит или не указать условие вовсе, то рекурсивные вызовы заполнят весь стек процесса и приложение экстренно завершит работу. Поэтому важно начинать описание рекурсивного метода с условия выхода.

В качестве примера рассмотрим подсчёт факториала числа (произведения всех чисел от единицы до заданного числа) рекурсивным методом:

```
static int GetFactorial(int number)
{
    if (number == 1) // терминальное условие
    {
        return number;
    }
    return number * GetFactorial(number - 1); // рекурсивный вызов
}
```

На практике рекурсия иногда применяется для обхода древовидных структур (материал следующего курса).

Анализ стека при помощи точек останова

Visual Studio позволяет нам просматривать текущий стек вызовов во время отладки. Это означает, что мы можем перемещаться по фреймам вперед и назад, просматривая значения локальных переменных и аргументов методов. Воспользуемся программой из предыдущего примера и запустим её под отладкой:

```
using System;

namespace HelloWorld
{
    class Program
```

```

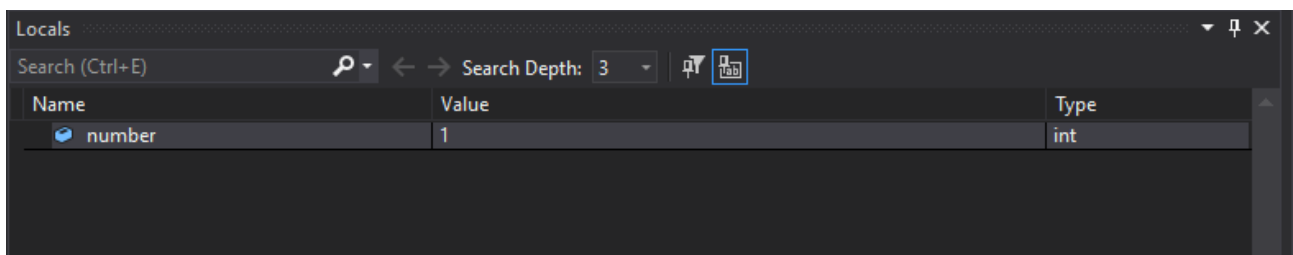
{
    static void Main(string[] args)
    {
        Console.WriteLine(GetFactorial(5));
    }

    static int GetFactorial(int number)
    {
        if (number == 1)
        {
            return number;
        }
        return number * GetFactorial(number - 1);
    }
}

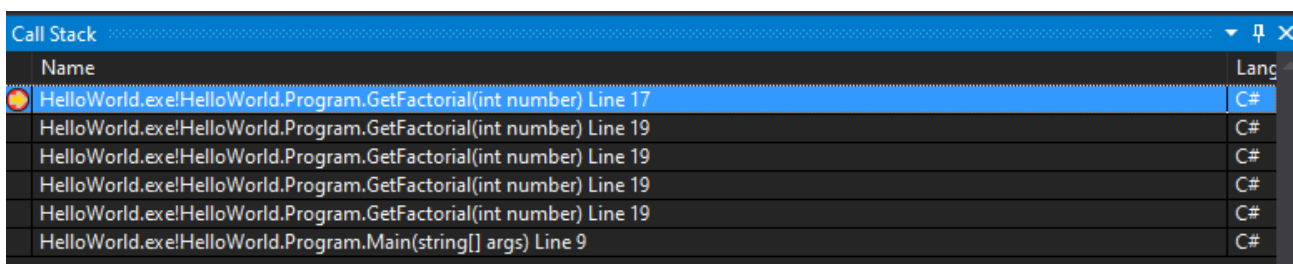
```

Установим точку останова (F9) на строке с терминальным выходом (`return number;`) — и запустим наш код (F5).

В панели Locals мы видим аргумент метода — `number`:



Справа от панели Locals расположена панель Call Stack:



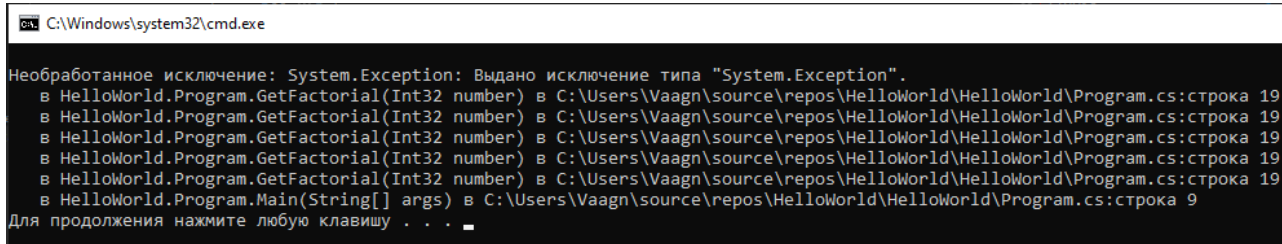
Если мы кликнем дважды на какой-либо из фреймов, то панель Locals отобразит значения аргументов и переменных из этого кадра стека.

Набор всех кадров стека, представленных в виде строки, называют `stacktrace` — трассировкой стека. `Stacktrace` порой содержит полезную информацию о том, в каком кадре возникла ошибка при

выполнении программы. Трассировка стека формируется автоматически в случае ошибки времени выполнения. Попробуем симитировать ошибку:

```
static int GetFactorial(int number)
{
    if (number == 1)
    {
        throw new Exception();
    }
    return number * GetFactorial(number - 1);
}
```

Теперь при наступлении терминального условия произойдёт исключительная ситуация. Если запустить программу без отладки (Ctrl + F5), мы увидим следующее:



C:\Windows\system32\cmd.exe

Необработанное исключение: System.Exception: Выдано исключение типа "System.Exception".

в HelloWorld.Program.GetFactorial(Int32 number) в C:\Users\Vaagn\source\repos\HelloWorld\HelloWorld\Program.cs:строка 19

в HelloWorld.Program.GetFactorial(Int32 number) в C:\Users\Vaagn\source\repos\HelloWorld\HelloWorld\Program.cs:строка 19

в HelloWorld.Program.GetFactorial(Int32 number) в C:\Users\Vaagn\source\repos\HelloWorld\HelloWorld\Program.cs:строка 19

в HelloWorld.Program.GetFactorial(Int32 number) в C:\Users\Vaagn\source\repos\HelloWorld\HelloWorld\Program.cs:строка 19

в HelloWorld.Program.GetFactorial(Int32 number) в C:\Users\Vaagn\source\repos\HelloWorld\HelloWorld\Program.cs:строка 19

в HelloWorld.Program.Main(String[] args) в C:\Users\Vaagn\source\repos\HelloWorld\HelloWorld\Program.cs:строка 9

Для продолжения нажмите любую клавишу . . .

Мы рассмотрим исключительные ситуации в одном из следующих уроков.

Практическое задание

1. Написать метод `GetFullName(string firstName, string lastName, string patronymic)`, принимающий на вход ФИО в разных аргументах и возвращающий объединённую строку с ФИО. Используя метод, написать программу, выводющую в консоль 3–4 разных ФИО.
2. Написать программу, принимающую на вход строку — набор чисел, разделенных пробелом, и возвращающую число — сумму всех чисел в строке. Ввести данные с клавиатуры и вывести результат на экран.
3. Написать метод по определению времени года. На вход подаётся число — порядковый номер месяца. На выходе — значение из перечисления (enum) — `Winter`, `Spring`, `Summer`, `Autumn`. Написать метод, принимающий на вход значение из этого перечисления и возвращающий название времени года (зима, весна, лето, осень). Используя эти методы, ввести с клавиатуры номер месяца и вывести название времени года. Если введено некорректное число, вывести в консоль текст «Ошибка: введите число от 1 до 12».
4. (*) Написать программу, вычисляющую число [Фибоначчи](#) для заданного значения рекурсивным способом.

Используемые источники

1. [Справочник по C#. Типы значений.](#)
2. [Ссылочные типы. Справочник по C#.](#)
3. [Руководство по программированию на C#. Упаковка-преобразование и распаковка-преобразование.](#)