Справочный материал

1. Основы методов (функций)

Метод в C# — это именованный блок кода, выполняющий определенную задачу. Использование методов является основой **процедурного** и **объектно-ориентированного программирования**, позволяя реализовать принципы **декомпозиции** (разделение сложной задачи на более простые) и **повторного использования кода (Code Reuse)**.

Рассмотрим сигнатуру метода:
`public static int Add(int a, int b)`

- **public**: Модификатор доступа. Определяет, откуда можно вызывать этот метод. `public` означает отсутствие ограничений.
- **static**: Ключевое слово, указывающее, что метод принадлежит самому классу, а не его конкретному экземпляру (объекту). В рамках данной работы все методы будут статическими.
- **int**: Тип возвращаемого значения. Метод обязан вернуть (`return`) значение этого типа. Если метод ничего не возвращает, используется тип `void` .
- **Add**: Имя метода. Должно быть информативным и отражать суть выполняемой операции.
- **(int a, int b)**: Список параметров (аргументов). Это входные данные, которые метод принимает для своей работы. `int a` - объявление локальной для метода переменной, которая получит свое значение в момент вызова.

Механизмы передачи

Это фундаментальная концепция, объясняющая, как данные из вызывающего кода попадают внутрь метода. Поведение зависит от типа данных.

Передача по значению (`value types`)

Типы-значения (Value Types)

- **Что это:** `int`, `double`, `bool`, `char`, `struct`.
- **Где хранятся:** Непосредственно в стеке. Переменная и есть само значение.
- **Механизм передачи: По значению (By Value)**. При передаче в метод система создает **полную копию** значения переменной в стеке. Любые изменения этой копии внутри метода **не затрагивают** оригинальную переменную.

Передача по ссылке (`ref`)

Модификатор `ref` заставляет передавать аргумент по ссылке. Функция получает **прямой доступ** к исходной переменной. Все изменения, произведенные с параметром внутри функции, **отражаются** на переменной, переданной при вызове.

```
// тут нет ref, поэтому передается копия значения
static void AddTen(int a)
{
    a += 10;
}

// ref указывает на то, что мы передаем не значение, а ссылку на значение
static void AddTwenty(ref int a)
{
    a += 20;
}

static void Main(string[] args)
{
    int V = 10;

    AddTen(V); // V будет равно 10
    AddTwenty(ref V); // V будет равно 30,
    //так как используется ключевое слово ref
}
```

Ссылочные типы (Reference Types)

- **Что это:** `string`, `массивы` (`int[]`), любой класс.

- **Где хранятся:** Сами данные объекта хранятся в **куче**. В **стеке** хранится только **ссылка** (адрес), указывающая на местоположение объекта в куче.
- **Механизм передачи (по умолчанию): Копия ссылки по значению (By Value)**. При передаче в метод система создает **копию ссылки**, но не самого объекта. В итоге и оригинальная переменная, и параметр метода (её копия) указывают на **один и тот же объект в куче**.
    - **Следствие 1:** Изменяя содержимое объекта по этой ссылке внутри метода (например, `array[0] = 99`), мы меняем тот самый единственный объект. Эти изменения будут видны снаружи.
    - **Следствие 2:** Присваивая параметру-ссылке новую ссылку (`sourceArray = newArray`), мы меняем лишь **локальную копию ссылки**. Оригинальная переменная продолжит указывать на старый объект.

Выходной параметр (`out`)

Модификатор `out` также передает аргумент по ссылке, но используется специально для того, чтобы **вернуть** одно или несколько значений из функции. В отличие от `ref`, переменную, передаваемую как `out`-параметр, не обязательно инициализировать до вызова. Однако функция **обязана** присвоить значение этому параметру перед своим завершением.

Подходит для функций, которые должны возвращать несколько значений, например, результат операции (`bool`) и сам продукт этой операции (вычисленное значение).

```
/// <summary>
/// Пытается преобразовать строку в целое число.
/// </summary>
/// <param name="input">Входная строка для анализа.</param>
/// <param name="result">Выходной параметр. Если преобразование
успешно,
/// сюда будет записано число. В противном случае — 0.</param>
/// <returns>Возвращает true, если строка является корректным
числом, иначе false.</returns>
public static bool TryParseInt(string input, out int result)
{
    // Проверяем на самые простые случаи: строка пустая или ее
вообще нет.
    if (string.IsNullOrEmpty(input))
```

```
    {
        result = 0; // Обязаны что-то присвоить out-параметру
        return false;
    }
    // Пробуем стандартными средствами C# сделать преобразование.
    // Блок try-catch - это обработка ошибок. Если Parse упадёт, мы
его поймаем.
    try
    {
        // Если эта строка выполняется без ошибок...
        result = int.Parse(input);
        return true;
        // ...значит, все прошло успешно.
    }
    catch (FormatException)
    {
        // Если мы попали сюда, значит, в строке были не только
цифры.
        result = 0; // Снова присваиваем значение по умолчанию.
        return false; // И сообщаем о провале.
    }
}
```

Массив параметров (`params`)

Модификатор `params` позволяет функции принимать **произвольное количество** аргументов одного типа. Внутри функции эти аргументы представлены в виде обычного массива. Параметр с модификатором `params` должен быть последним в списке параметров метода.

```
static int GetItemsCount(params int[] items)
{
    //Вернет количество переданных аргументов
    return items.Length;
}

int t = GetItemsCount(1,2,3,4,5);
Console.WriteLine(t); // Выведет 5
```

```
static int GetItemsCount(int a, params int[] items)
{
    //Вернет количество переданных аргументов
    return items.Length;
}

int t = GetItemsCount(1,2,3,4,5);
Console.WriteLine(t); // Выведет 4, так как
                      // значение "1" уйдет в аргумент "a"
```

Ключевые слова для управления передачей параметров

Ключевое слово	Назначение	Механизм работы	Требует инициализации
(нет)	Стандартная передача (по значению)	Копируется либо само значение (для value-типов), либо ссылка (для reference-типов).	**Да**
params	Передача переменного числа аргументов	Компилятор собирает перечисленные аргументы в массив указанного типа и передает его в метод.	Да
out	Возврат дополнительных результатов из метода	Передается ссылка на переменную. Метод **обязан** присвоить значение этой переменной перед завершением.	Нет
ref	Принудительная передача параметра по ссылке	Передается ссылка на саму переменную (а не на ее значение или копию ссылки). Любые изменения влияют на оригинал.	**Да**

Подробнее:

- **params (Массив параметров)**: Синтаксический сахар, позволяющий вызывать метод более гибко. Может быть только один params-параметр, и он должен быть последним в списке.
- **out (Выходной параметр)**: Гарантирует, что метод вернет значение через этот параметр. Подходит для функций, которые должны возвращать несколько значений, например, результат операции (`bool`) и сам продукт этой операции (вычисленное значение).
- **ref (Передача по ссылке)**:
  - **Для value-типов:** В метод передается не копия значения, а ссылка на ячейку в стеке, где хранится оригинальная переменная. Это позволяет методу изменять значение оригинальной переменной.
  - **Для reference-типов:** В метод передается ссылка **на саму ссылку**. Это позволяет методу не только изменять содержимое объекта в куче, но и **перенаправить оригинальную ссылку** на совершенно другой объект. Именно этот эффект демонстрируется в Задании 5.

Перегрузка методов

Перегрузка методов - возможность создания в одном классе нескольких методов с **одинаковым именем**, но с **разными наборами параметров** (разным количеством, типами или порядком). Компилятор автоматически выбирает нужную версию метода на основе аргументов, переданных при вызове. Тип возвращаемого значения не входит в сигнатуру:

```
static void Sum(int a, int b)
{
    Console.WriteLine(a + b);
}

static int Sum(int a, int b)
{
    return a + b;
}
```

Получим следующую ошибку:

```
Тип "Program" уже определяет член "Sum" с такими же типами
параметров.
```

В следующем примере сигнатуры методов различаются (тип данных аргумента):

```
static double Sum(double a, double b)
{
    return a + b;
}

static int Sum(int a, int b)
{
    return a + b;
}
```

Делегат

Введение в концепцию

До этого момента мы передавали в методы данные: числа, строки, массивы. Делегаты вводят более мощную парадигму: **передачу в качестве параметра самого поведения (логики выполнения)**.

Делегат - ссылочный тип, который инкапсулирует ссылку на метод. Он является типобезопасным указателем на функцию. "Типобезопасный" означает, что делегат может ссылаться только на методы с совместимой сигнатурой (совпадающим набором параметров и типом возвращаемого значения).

Использование делегатов позволяет создавать компоненты, чью логику можно изменять во время выполнения, не меняя их исходного кода.

Пример: Универсальный обработчик текста

Рассмотрим задачу: нам нужен метод, который принимает строку и применяет к ней какое-либо форматирование перед выводом в консоль. Сначала нам может быть нужно перевести строку в верхний регистр, после - в нижний, и наконец - обернуть в символы.

Вместо того чтобы писать три разных метода (`PrintInUppercase` , `PrintInLowercase` и т.д.), мы создадим один универсальный метод, а конкретное правило форматирования будем передавать ему как параметр.

Шаг 1: Объявление "контракта" (делегата)

Сначала мы определим, как должен выглядеть любой метод-форматтер. Он должен принимать одну строку и возвращать одну строку.

```
// Объявляем делегат. Он определяет "шаблон" для всех методов,
// которые могут выполнять форматирование текста.
public delegate string TextFormatter(string inputText);
```

Шаг 2: Создание конкретных реализаций ("поведения")

Теперь напишем методы, которые соответствуют сигнатуре делегата `TextFormatter`.

```
/// <summary>
/// Преобразует текст в верхний регистр.
/// </summary>
public static string ConvertToUppercase(string text)
{
    return text.ToUpper();
}

/// <summary>
/// Оборачивает текст в символы "---".
/// </summary>
public static string AddDashes(string text)
{
    return $"---{text}---";
}
```

Шаг 3: Создание метода высшего порядка

Это универсальный метод, который принимает данные (текст) и поведение (делегат-форматтер). Он не знает, *как именно* будет отформатирован текст, он лишь знает, *как вызвать* переданный ему форматтер.

```
/// <summary>
/// Обрабатывает строку с помощью переданного форматтера и выводит
результат.
/// </summary>
/// <param name="text">Входной текст.</param>
/// <param name="formatter">Делегат, содержащий метод для
```

```
форматирования.</param>
public static void ProcessAndPrint(string text, TextFormatter
formatter)
{
    // Вызываем метод, на который ссылается делегат 'formatter'
    string formattedText = formatter(text);
    Console.WriteLine($"Результат: {formattedText}");
}
```

Шаг 4: Объединение всего в `Main`

Теперь в `Main` мы можем вызывать наш универсальный обработчик, передавая ему разные "стратегии" форматирования.

```
using System;

public class DelegateExample
{
    // 1. Объявление делегата
    public delegate string TextFormatter(string inputText);

    // 2. Методы, соответствующие делегату
    public static string ConvertToUppercase(string text)
    {
        return text.ToUpper();
    }
    public static string AddDashes(string text)
    {
        return $"---{text}---";
    }
    public static string AddTimestamp(string text)
    {
        return $"[{DateTime.Now:T}] {text}";
    }
    // 3. Метод высшего порядка
    public static void ProcessAndPrint(string text, TextFormatter
formatter)
    {
        string formattedText = formatter(text);
        Console.WriteLine(formattedText);
    }

    public static void Main(string[] args)
```

```
    {
        string message = "Hello, Delegates!";
        Console.WriteLine($"Исходное сообщение: {message}\n");

        // Создаем экземпляр делегата, который ссылается на
ConvertToUppercase
        TextFormatter upperFormatter = ConvertToUppercase;
        // Передаем поведение "преобразовать в верхний регистр"
        ProcessAndPrint(message, upperFormatter);

        // Теперь тот же метод ProcessAndPrint будет выполнять
другое действие
        TextFormatter dashFormatter = AddDashes;
        // Передаем поведение "добавить тире"
        ProcessAndPrint(message, dashFormatter);

        // Можно передавать метод напрямую, без создания отдельной
переменной
        // Передаем поведение "добавить временную метку"
        ProcessAndPrint(message, AddTimestamp);
    }
}
```

Ожидаемый вывод:

```
Исходное сообщение: Hello, Delegates!

HELLO, DELEGATES!
---Hello, Delegates!---
[18:30:15] Hello, Delegates!
```