

# Значимые и ссылочные типы, строки

## 1. Значимые (value) vs ссылочные (reference)

Кратко:

- **Значимые (value)** типы хранят значение **непосредственно**. При присваивании/передаче копируются. Примеры: `int`, `double`, `bool`, `char`, `enum`.
- **Ссылочные (reference)** типы хранят **ссылку** (адрес) на объект в куче (heap). При присваивании/передаче копируется ссылка — оба имени указывают на один и тот же объект. Примеры: `string`, массивы `int[]`, любые объекты (класс).

**Почему это важно:** поведение при присваивании и передаче в методы разное — изменение объекта через одну ссылку видно через другую; изменение значимого типа — нет (копия).

Примеры:

```
// Значимые типы — копируются
int a = 5;
int b = a;
b = 10;
Console.WriteLine(a); // 5

// Ссылочные типы — копируется ссылка
int[] arr1 = {1, 2};
int[] arr2 = arr1;
arr2[0] = 99;
Console.WriteLine(arr1[0]); // 99 — видно через обе переменные
```

## Стек (stack) и куча (heap)

- **Стек (stack)** — это быстрая временная память для текущей работы программы. В нём хранятся параметры и локальные переменные методов. Когда метод вызывается — создаётся «блок» в стеке, он называется контекстом метода; когда метод заканчивает выполнение — блок снимается и память освобождается автоматически.

Пример: `int i = 5;` — локальная переменная обычно хранится в стеке.

- **Куча (heap)** — это память для объектов, которые живут дольше одного метода. Когда вы создаёте объект через `new` (например, массив или строку), сам объект размещается в куче, а **в переменной хранится ссылка** на этот объект. Куча управляется специальной системой — сборщиком мусора (GC).

Пример: `int[] arr = new int[3];` — массив находится в куче, а `arr` (в стеке) — это ссылка на него.

## Сборщик мусора (GC)

- **GC (garbage collector)** — это часть .NET, которая автоматически **освобождает память** от объектов, на которые больше **нет ссылок** (то есть они недостижимы из кода).
- Вы как программист обычно **не освобождаете память вручную** — рантайм делает это сам. Это упрощает разработку и уменьшает ошибки утечек памяти в простых программах.
- Объект удаляется **только тогда, когда на него больше не ссылаются**. Если какая-то переменная или коллекция всё ещё держит ссылку — объект не будет удалён.

```
void Foo()
{
    var arr = new int[100]; // массив в куче
} // после выхода из Foo ссылка arr исчезает – массив может быть удалён GC
```

## null и nullable-типы

`null` — специальное значение ссылочных типов, означающее «нет объекта». Попытка вызвать метод на `null` приведёт к `NullReferenceException`. Оно является дефолтным для всех ссылочных типов.

**Nullable для значимых типов:** чтобы позволить значимым типам принимать `null`, используется `T?`:

```
int? maybe = null;  
maybe = 5;
```

### Проверка на null:

- Обычная:

```
if (s == null) { /* ... */ }  
if (s != null) { /* ... */ }
```

- Паттерн:

```
if (s is null) { /* ... */ }  
if (s is not null) { /* ... */ }
```

## Операторы для работы с `null` :

- `??` — null-coalescing (вернёт левый операнд, если он не `null`; иначе правый):

```
string name = input ?? "Аноним";
```

- `?.` — null-conditional (безопасный доступ к члену, вернёт `null`, если левый `null`):

```
int? len = s?.Length; // если s == null, len == null
```

- `??=` — присвоение при `null`:

```
s ??= "default";
```

- `!` — null-forgiving (оператор «я уверен, что не `null`» — убирает предупреждение компилятора; использовать осторожно):

```
int len = s!.Length;
```

## Пример nullable и проверок:

```
int? age = null;  
int effective = age ?? 0; // если age null => 0
```

## 2. Строки

`string` — **ссылочный тип**, но он **неизменяемый (immutable)**. Это значит: после создания строки её содержимое нельзя изменить.

Операции, которые «меняют» строку (конкатенация, `Replace` и т.п.), на самом деле создают **новую** строку в памяти. Из-за этого он похож по поведению на значимые типы.

Это сделано для безопасности и предсказуемости кода (строки можно безопасно передавать между частями кода). Однако это может приводить к проблемам с производительностью при многократном изменении строки.

### Часто используемые методы строк

Работа со строками в C# — одна из самых распространённых задач. Вот список методов, которые нужно знать с самого начала:

- `s.Length` — возвращает количество символов в строке.

```
string s = "Hello";  
Console.WriteLine(s.Length); // 5
```

- `s.Substring(start, length)` — возвращает часть строки, начиная с позиции `start` и длиной `length`.

```
string s = "HelloWorld";  
Console.WriteLine(s.Substring(0, 5)); // Hello
```

- `s.IndexOf("sub")` / `s.LastIndexOf("sub")` — ищет подстроку и возвращает её индекс (слева направо или справа налево). Возвращает `-1`, если не найдено.

```
string s = "banana";  
Console.WriteLine(s.IndexOf("na")); // 2  
Console.WriteLine(s.LastIndexOf("na")); // 4
```

- `s.Replace("a", "b")` — создаёт новую строку, в которой все вхождения "a" заменены на "b" .

```
string s = "cat";  
Console.WriteLine(s.Replace("c", "b")); // bat
```

- `s.ToLower()` / `s.ToUpper()` — возвращает строку в нижнем или верхнем регистре.

```
string s = "Hello";  
Console.WriteLine(s.ToLower()); // hello  
Console.WriteLine(s.ToUpper()); // HELLO
```

- `s.Trim()` / `s.Trim(chars[])` — удаляет пробелы (по умолчанию) или указанные символы в начале и в конце строки.

```
string s = "  hello  ";  
Console.WriteLine(s.Trim()); // "hello"  
  
string s2 = "...text...";  
Console.WriteLine(s2.Trim('.')); // "text"
```

- `s.Split(separator)` — разбивает строку на массив подстрок по разделителю.
- `string.Join(separator, parts)` — соединяет массив строк в одну строку с разделителем.

```
string s = "a,b,c";  
string[] parts = s.Split(',');  
// parts = ["a", "b", "c"]  
  
string joined = string.Join(";", parts);  
Console.WriteLine(joined); // "a;b;c"
```

- `s.Contains("x")` — проверяет, есть ли в строке указанная подстрока.
- `s.StartsWith("pre")` — проверяет, начинается ли строка с подстроки.
- `s.EndsWith("post")` — проверяет, заканчивается ли строка подстрокой.

## StringBuilder

`StringBuilder` (в `System.Text`) нужен там, где вы собираете строку по кусочкам — особенно в цикле.

Он хранит внутренний изменяемый буфер (масштабируемый массив символов) и добавляет куски в него без постоянного копирования всей строки. В конце вы один раз вызываете `ToString()` и получаете итоговую строку.

### Пример:

Этот простой код будет работать долго и потреблять много памяти:

```
string s = "";
for (int i = 0; i < 1000000; i++)
{
    s += i.ToString() + ","; // каждый шаг создаёт новую строку
}
```

После того как мы перепишем код на `StringBuilder`, он выполнится почти мгновенно:

```
using System.Text;

var sb = new StringBuilder();
for (int i = 0; i < 1000000; i++)
{
    sb.Append(i);
    sb.Append(','); // специальный Append для char
}
string result = sb.ToString();
```

## Спецсимволы

Наиболее часто используемые:

- `\n` — новая строка (перевод строки).
- `\r` — возврат каретки (используется вместе с `\n`, т.е. `\r\n`).
- `\t` — табуляция (горизонтальный отступ).
- `\\` — обратный слэш `\`.
- `\"` — кавычка внутри строки.

```
string s = "Hello\tWorld\n\"C#\"";
Console.WriteLine(s);
// Вывод:
// Hello      World
// "C#"
```

## Подробный текст ( `@""` )

Чтобы не писать много обратных слэшей, можно использовать подробный текст в котором отключены все спецсимволы. Она обозначается префиксом `@` перед строкой:

```
// вместо того, чтобы писать так
string path = "C:\\Users\\Admin\\Docs";
Console.WriteLine(path);

// можно писать так
string path = @"C:\Users\Admin\Docs";
Console.WriteLine(path); // C:\Users\Admin\Docs
```



## Особенности:

- Спецсимволы в нём не обрабатываются.
- Можно писать многострочные строки прямо в коде:

```
string multi = @"Первая строка  
Вторая строка  
Третья строка";  
Console.WriteLine(multi);
```

- Для того чтобы поставить двойные кавычки " " нужно написать их дважды "" :

```
string s = @"He said, ""This is the last chance!"  
Console.WriteLine(s);  
// He said, "This is the last chance!"
```

## Интерполяция строк ( \$" " )

Интерполяция позволяет подставлять значения переменных прямо в строку.

```
int age = 20;
string name = "Иван";
string s = $"Меня зовут {name}, мне {age} лет";
Console.WriteLine(s);
// Меня зовут Иван, мне 20 лет
```

Можно совмещать с @ :

```
string folder = "Temp";
Console.WriteLine($"@C:\{folder}\files"); // C:\Temp\files
```

## Форматирование

Иногда нужно вывести числа или даты в удобочитаемом виде:

- числа с фиксированным количеством знаков после запятой,
- даты в определённом формате,
- выравнивание по ширине,
- добавление разделителей разрядов и т.п.

В C# это делается через **форматные строки**. Они указываются после двоеточия : внутри фигурных скобок при интерполяции ( \$" " ) или в методах `string.Format` , `Console.WriteLine` .

# Форматирование чисел

Основные спецификаторы:

Спецификатор	Пример	Результат
F (Fixed-point, фиксированная точка)	\${3.14159:F2}	3.14
N (Number, с разделителями разрядов)	\${12345.6789:N2}	12,345.68
E (Exponential, экспоненциальная форма)	\${12345:E2}	1.23E+004
P (Percent, процент)	\${0.1234:P1}	12.3 %
C (Currency, денежный формат)	\${123.45:C}	123,45 ₺ (зависит от локализации)

Пример:

```
double x = 12345.6789;
Console.WriteLine($"{x:F2}"); // 12345.68
Console.WriteLine($"{x:N0}"); // 12,346
Console.WriteLine($"{x:E3}"); // 1.235E+004
```

# Форматирование дат и времени

Даты поддерживают свои коды:

Код	Значение	Пример
d	Краткая дата	19.08.2025
D	Полная дата	19 августа 2025 г.
t	Краткое время	15:45
T	Полное время	15:45:33
f	Полная дата + краткое время	19 августа 2025 г. 15:45
g	Краткая дата + краткое время	19.08.2025 15:45
o	ISO 8601	2025-08-19T15:45:33.1234567
yyyy-MM-dd	Год-месяц-день	2025-08-19
dd.MM.yyyy HH:mm	Дата и время вручную	19.08.2025 15:45

## Выравнивание

Можно задать ширину поля и выравнивание:

```
Console.WriteLine($"|{"Name",-10}|{"Age",5}|");
// -10 = выравнивание влево, 10 символов
// 5 = вправо, 5 символов

// Результат:
// |Name      | Age|
```

# Кодировка строк

Когда мы работаем со строками, на самом деле мы работаем не с «буквами», а с их **кодами** — числами, которые соответствуют символам в таблице **Unicode**.

## Как устроено хранение строк в .NET

- В .NET строки ( `string` ) хранятся во внутреннем формате **UTF-16**.
- Каждый символ ( `char` ) — это 16-битное число (от 0 до 65535), которое соответствует **кодовой единице**.
- Большинство привычных символов (латиница, кириллица, цифры, спецсимволы) укладываются в одну такую кодovou единицу.

## Проблема с «длинными» символами

- Но в Unicode есть символы, которым **не хватает 16 бит** (например, эмодзи 😊, редкие иероглифы, музыкальные ноты).
- Для них используется механизм **суррогатных пар**: символ хранится в виде двух `char` подряд.
- Поэтому:

```
string emoji = "😊";  
Console.WriteLine(emoji.Length); // 2, хотя мы видим один символ
```

То есть `Length` возвращает количество кодовых единиц (`char`), а не «символов, как видит человек».

# Возможные проблемы

## 1. Длина строки $\neq$ количество символов.

Особенно заметно с эмодзи, диакритическими знаками ( `é` может быть представлена как `e + ´` ).

## 2. Обрезка строк.

Если взять `Substring` и случайно «разрезать» суррогатную пару, то строка окажется повреждённой:

```
string s = "😊X";  
Console.WriteLine(s.Substring(0,1)); // ошибка: останется половина символа
```

## 3. Сравнение строк.

Два одинаково выглядящих символа могут иметь разное внутреннее представление (например, `é` может храниться как один код или как комбинация).

## 4. Неправильная кодировка при чтении/записи файлов.

Если файл сохранён в UTF-8, а вы откроете его как ANSI — русские буквы «сломаются» (будут `❓` или непонятные символы).

# Регулярные выражения

**Регулярные выражения (regex)** — это специальный язык для поиска и обработки текста по шаблону.

С их помощью можно:

- искать в строке определённые подстроки (например, все email-адреса);
- проверять, соответствует ли строка какому-то формату (например, дата, телефон, индекс);
- заменять части строки по условию.

В .NET регулярные выражения находятся в пространстве имён `System.Text.RegularExpressions`, главный класс — `Regex`.

## Основные методы

- `Regex.IsMatch(string, pattern)` — проверяет, соответствует ли строка шаблону. Возвращает `true/false`.
- `Regex.Match(string, pattern)` — находит первое совпадение, возвращает объект `Match`.
- `Regex.Matches(string, pattern)` — находит **все** совпадения, возвращает коллекцию `MatchCollection`.
- `Regex.Replace(string, pattern, replacement)` — заменяет найденные совпадения на указанный текст.

## Основные элементы синтаксиса

- `.` — любой символ (кроме новой строки).
- `\d` — цифра ( `0-9` ), `\D` — не цифра.
- `\w` — буква, цифра или `_` . `\W` — не буква/цифра.
- `\s` — пробельный символ, `\S` — не пробел.
- `+` — «один или больше», `*` — «ноль или больше», `?` — «ноль или один».
- `{n}` — ровно `n` раз, `{n,m}` — от `n` до `m` раз.
- `^` — начало строки, `$` — конец строки.
- `(...)` — группа (можно извлекать подстроки).
- `|` — «или».

# Примеры

## Проверка даты в формате dd-mm-yyyy :

```
using System.Text.RegularExpressions;

string text = "Сегодня 12-05-2020, а завтра 13-05-2020";
string pattern = @"^\b\d{2}-\d{2}-\d{4}\b"; // дата dd-mm-yyyy

// Проверка, есть ли дата
bool hasDate = Regex.IsMatch(text, pattern);
Console.WriteLine(hasDate); // True

// Первое совпадение
var match = Regex.Match(text, pattern);
if (match.Success)
    Console.WriteLine("Найдена дата: " + match.Value);

// Все совпадения
var matches = Regex.Matches(text, pattern);
foreach (Match m in matches)
    Console.WriteLine("Дата: " + m.Value);
```

## Проверка email-адреса

```
string email = "student@example.com";
string pattern = @"^[a-zA-Z0-9_-]+@[a-zA-Z0-9_-]+\.[a-zA-Z0-9]+";

Console.WriteLine(Regex.IsMatch(email, pattern)); // True
```



### 3. Параметры `ref` , `out`

В C# по умолчанию все **значимые типы** (например, `int` , `bool` , `struct` ) передаются в метод **по значению**. Это значит, что внутри метода мы работаем с **копией** переменной, и изменения не затрагивают оригинал.

Иногда нужно, чтобы метод **изменил исходную переменную**. Для этого и существуют модификаторы `ref` и `out` .

#### `ref`

- Передаёт переменную **по ссылке**.
- Переменная **должна быть инициализирована** до вызова.
- Метод может **изменить её значение**, и изменения будут видны снаружи.

Пример:

```
static void IncRef(ref int x)
{
    x++; // изменяем исходную переменную
}

int a = 5;
IncRef(ref a);
Console.WriteLine(a); // 6
```

## out

- Используется, когда метод должен **вернуть дополнительное значение** через параметр.
- Переменная **не обязана быть инициализирована** перед вызовом.
- Метод обязан **присвоить ей значение** до выхода.

Пример:

```
static bool ParsePositive(string s, out int number)
{
    if (int.TryParse(s, out number) && number > 0)
        return true;
    return false;
}

if (ParsePositive("42", out int n))
    Console.WriteLine(n); // 42
else
    Console.WriteLine("Ошибка");
```

## Разбор на примере TryParse

Методы TryParse (например, `int.TryParse`) — это классический пример использования `out`.

Обычный `int.Parse("abc")` выбросит исключение, если строка не число.

А `int.TryParse` работает безопасно:

- Возвращает `true`, если число удалось разобрать.
- Возвращает `false`, если строка некорректная.
- Через параметр `out` возвращает само значение.

Пример:

```
Console.Write("Введите число: ");
string line = Console.ReadLine();

if (int.TryParse(line, out int value))
{
    Console.WriteLine($"Введено: {value}, удвоенное значение: {value * 2}");
}
else
{
    Console.WriteLine("Ошибка: нужно ввести целое число.");
}
```

---

## Ссылки

- [Как работают кодировки текста. Откуда появляются «кракозябры». Принципы кодирования. Обобщение и детальный разбор / Хабр](#)
  - [Хороший сайт для изучения регулярных выражений на практике](#)
  - [C# и .NET | Типы значений и ссылочные типы](#)
  - [Пример: как хранятся данные в стеке и куче](#)
-

# Практическое задание

## Что нужно сделать:

1. Продолжайте работу в проекте **ToDoList**, созданном ранее.

2. Добавьте поддержку **флагов команд**.

- **Флаг** — это специальный параметр в консольном приложении, который изменяет поведение команды.
- Обычно флаги начинаются с двух дефисов ( `--flag` ) или имеют короткую форму с одним дефисом ( `-f` ), короткие формы можно комбинировать ( `-i -s` или `-is` ).
- Пример:

```
add --multiline  
view -is
```

3. Команда `add`

- Оставьте существующий режим (однострочный ввод).
- Добавьте **многострочный режим** с флагами `--multiline` или `-m`.
- При его активации:
  - Пользователь вводит строки задач в формате:

```
> строка 1  
> строка 2  
...
```

- Ввод продолжается, пока не введена строка `!end`.
- После этого задача сохраняется в массив как единый текст (объединение строк через `\n`).

4. Команда `view`

Добавьте поддержку флагов для гибкого отображения, флаги можно комбинировать:

- `--index` , `-i` — показывать индекс задачи.
- `--status` , `-s` — показывать статус задачи.

- `--update-date` , `-d` — выводить дату последнего изменения.
- `--all` , `-a` — выводить все данные одновременно.  
По умолчанию (без флагов) показывается только текст задачи.
- Реализуйте вывод **в виде таблицы**:
  - Колонки выравниваются по ширине.
  - В колонке с текстом задачи показывайте **только первые 30 символов**, остальное заменяйте на `...`.

#### 5. Команда `read <idx>`

- Добавьте новую команду для просмотра **полного текста задачи**.
- Выводите:
  - Полный текст задачи (без обрезки).
  - Статус ( `выполнена` / `не выполнена` ).
  - Дату последнего изменения.

#### 6. Обновите вывод команды `help` .

#### 7. Для парсинга команд используйте:

- Методы строк ( `Split` , `StartsWith` , `Contains` и т.д.),
- Либо регулярные выражения ( `Regex` ).

#### 8. Используйте `TryParse` для перевода строк в числа.

#### 9. Добавьте проверку на `null` везде, где это необходимо.

#### 10. Каждую команду оформите **отдельным методом**.

#### 11. После реализации протестируйте:

- добавьте задачи в обоих режимах ( `add` и `add --multi-line` ),
- выведите их через разные флаги `view` ,
- проверьте корректность работы `read` .

#### 12. Сделайте коммиты после **каждого изменения**. Один большой коммит будет оцениваться в два раза ниже.

#### 13. Обновите **README.md** — добавьте описание новых возможностей программы.

#### 14. Сделайте push изменений в GitHub.