

Strings

- Strings
 - Char
 - Char methods
 - String
 - Equals
 - Compare
 - Interning
 - Methods
 - Format
 - StringBuilder
 - Encoding

Char

char - 16 bite символ UTF-16

- Представляет собой структуру `System.Char`
- Value type

```
public struct Char : IComparable, IComparable<char>, IConvertible, IEquatable<char>
```

```
char[] chars = new char[4];
```

```
chars[0] = 'X';           // Character literal
chars[1] = '\x0058';      // Hexadecimal
chars[2] = (char)88;      // Cast from integral type
chars[3] = '\u0058';      // Unicode
```

- Разница между `\x`, `\u` - SOF Jon Skeet:
 - `\x` - 1-4 hex digits
 - `\u` - 4 hex digits
- implicit convert to `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`.

Char methods

```
char chA = 'A', ch1 = '1';  
string str = "test string";  
  
chA.CompareTo('B');           // "-1" (meaning 'A' is 1 less than 'B')  
chA.Equals('A');              // True  
Char.GetNumericValue(ch1);    // 1  
Char.IsControl('\t');         // True  
Char.IsDigit(ch1);            // True  
Char.IsLetter(',');           // False  
Char.IsLower('u');            // True  
Char.IsNumber(ch1);           // True  
Char.IsPunctuation('.');      // True  
Char.IsSeparator(str, 4);     // True  
Char.IsSymbol('+');           // True  
Char.IsWhiteSpace(str, 4);    // True  
Char.Parse("S");              // S  
Char.ToLower('M');            // m  
'x'.ToString();              // x
```

Unicode символ вообще кодируется 21 бит, поэтому иногда символ кодируется двумя **char**

- Если символ выходит за пределы стандартных
- Если используются глифы

```
char[] chars = { '\u0061', '\u0308' };  
string strng = new String(chars);  
Console.WriteLine(strng);           // ä  
  
string good = "Tab\x9Good compiler"; // Tab Good compiler  
string bad = "Tab\x9Bad compiler";   // Tab鮭 compiler
```

String

Строка - неизменяемая упорядоченная коллекция **char**

- ссылочный тип
- неизменяемые (immutable)
- класс sealed из-за оптимизации

```
String s = "Hi\r\nthere."; // лучше не надо так
s = "Hi" + Environment.NewLine + "there."; // а вот так красиво

s = ""; // так себе
s = string.Empty; // норм

s = "Hi" + " " + "there."; // Строки литеральные, выполнится при компиляции
s = string.Concat("Hi", " ", "there.");
s = string.Format("{0} {1}", "Hi", "there.");
s = $"{myVariable1} some text {myClassVariable.SomeProperty}"; // интерполяция
```

Escape последовательности не отличаются от C:

- `\'` single quote
- `\"` double quote
- `\\` backslash
- `\0` `null` character
- `\a` alert character
- `\b` backspace
- `\f` form feed
- `\n` new line
- `\r` carriage return
- `\t` horizontal tab
- `\v` vertical tab
- `\uxxxx` unicode character hex value `\u0020`
- `\x` is the same as `\u`, but you don't need leading zeroes `\x20`
- `\Uxxxxxxxx` unicode character hex value (longer form needed for generating surrogates)

- Для verbatim строк символ \ не рассматривается, как управляющий

```
string file = "C:\\Windows\\System32\\Notepad.exe";  
String file = @"C:\Windows\System32\Notepad.exe"; // Verbatim string
```

- Строки неизменяемы (Зий повтор фразы)
- Чем больше методов, тем больше объектов в куче
- На одинаковые строки могут ссылаться разные объекты
- Нет проблем с многопоточностью

```
if (s.ToUpperInvariant().Substring(10, 21).EndsWith("EXE"))  
{  
}
```

Equals

Сравнение строк

```
if (myStr == myStr2) // Не надо так
```

```
// Лучше так:
```

```
Boolean Equals(String value, StringComparison comparisonType);
```

```
static Boolean Equals(String a, String b, StringComparison comparisonType);
```

```
public enum StringComparison
```

```
{
```

```
    CurrentCulture = 0,
```

```
    CurrentCultureIgnoreCase = 1,
```

```
    InvariantCulture = 2,
```

```
    InvariantCultureIgnoreCase = 3,
```

```
    Ordinal = 4,
```

```
    OrdinalIgnoreCase = 5
```

```
}
```


- Ordinal сравнивает по unicode кодам
- Invariant по некому "дефолтному" списку символов

```
var s1 = "Strasse";  
var s2 = "Straße";  
  
s1.Equals(s2, StringComparison.Ordinal);    // false  
s1.Equals(s2, StringComparison.InvariantCulture); // true
```

MSDN Strings Best Practice, SOF Ordinal Vs Invariant:

- Используйте перегруженные версии Equals для сравнения строк (static версия не кидает исключений, если первая строка `null`)
- Используйте `StringComparison.Ordinal` or `StringComparison.OrdinalIgnoreCase` по-дефолту, когда вам не важна локаль
- Ordinal намного быстрее Invariant (to 10x)
- Используйте `CurrentCulture` для отображения пользователю
- Используйте `String.ToUpperInvariant` вместо Lower для нормализации сравнения
- Не используйте `Invariant` в большинстве случаев, кроме суперредких ситуаций, когда вам важны спец символы, но при этом не важны особенности культуры
- Используйте форматирование и интерполяцию - код должен быть красивым

Compare

```
static int Compare(String strA, String strB, StringComparison comparisonType);  
static int Compare(string strA, string strB, Boolean ignoreCase, CultureInfo culture);  
static int Compare(String strA, String strB, CultureInfo culture, CompareOptions options);
```

- Используйте Compare только для сортировки, не для равенства!

```
bool StartsWith(String value, StringComparison comparisonType);  
bool StartsWith(String value, Boolean ignoreCase, CultureInfo culture);  
  
bool EndsWith(String value, StringComparison comparisonType);  
bool EndsWith(String value, Boolean ignoreCase, CultureInfo culture);
```

```
var l = new List<string>
{ "0", "9", "A", "Ab", "a", "aB", "aa", "ab", "ss", "ß",
  "Ä", "Äb", "ä", "äb", "ぁ", "ぁ", "ァ", "ァ", "A", "亜", "Ё", "ё" };
```

```
Ordinal                // 0 9 A Ab a aB aa ab ss Ä Äb ß ä äb Ё ё ぁ ぁァァ 亜 A
OrdinalIgnoreCase      // 0 9 A a aa Ab aB ab ss Ä ä äb Äb ß Ё ё ぁ ぁァァ 亜 A
InvariantCulture       // 0 9 a A A ä Ä aa ab aB Ab äb Äb ss ß ë Ë ァ あ ア あ 亜
InvariantCultureIgnoreCase // 0 9 a A A ä Ä aa aB Ab ab äb Äb ss ß ë Ë ァ あ ア あ 亜
"da-DK" culture        // 0 9 a A A ab aB Ab ss ß ä Ä äb Äb aa ë Ë ァ あ ア あ 亜
"de-DE"                // 0 9 a A A ä Ä aa ab aB Ab äb Äb ss ß ë Ë ァ あ ア あ 亜
"en-US"                // 0 9 a A A ä Ä aa ab aB Ab äb Äb ss ß ë Ë ァ あ ア あ 亜
"ja-JP"                // 0 9 a A A ä Ä aa ab aB Ab äb Äb ss ß ë Ë ァ あ ア あ 亜
"ru-RU"                // 0 9 a A A ä Ä aa ab aB Ab äb Äb ss ß ë Ë ァ あ ア あ 亜
```

Interning

- Создается внутренняя хеш-таблица
- В ней содержатся ссылки на объекты строк в куче
- Строки указанные в хеш-таблице не освобождаются GC!

```
public static String Intern(String str); // Добавляет строку, если не нашел  
public static String IsInterned(String str); // Возвращает null, если не нашел
```

- По-умолчанию clr интернирует все литеральные строки, описанные в метаданных. Но не надо на это рассчитывать
- Можно рассчитывать только на ручной вызов `Intern`

```
String s1 = "Hello";  
String s2 = "Hello" + string.Empty;  
String s3 = "Hell" + string.Empty + "o";  
Console.WriteLine(Object.ReferenceEquals(s1, s2)); // True  
Console.WriteLine(Object.ReferenceEquals(s1, s3)); // false  
  
s1 = String.Intern(s1);  
s3 = String.Intern(s3);  
Console.WriteLine(Object.ReferenceEquals(s1, s3)); // True
```

Methods

```
string text = "hello world";
int indexOfChar = text.IndexOf('o'); // равно 4
text.IndexOf("orl"); // равно 6

text.EndsWith("ld") == true // true
string[] words = text.Split(new char[] { ' ' },
StringSplitOptions.RemoveEmptyEntries);

text = "  hello world ".Trim(); // результат "hello world"
text = text.Trim(new char[] { 'd', 'h' }); // результат "ello worl"
"  hello world ".TrimStart(); // "hello world "
"hell world".Insert(4, "o");
text.Remove(0,6); // "world"
text.Replace("hello", "my");
text.ToUpper(); // HELLO WORLD
text.Substring(1,4); // ello
```

Format

```
string output = String.Format("name: {0} , last name: {1}", name, lastname);
```

```
int number = 30;  
String.Format("{0:d}", number);    // 30  
String.Format("{0:d4}", number);   // 0030  
  
number.ToString("d4");             // 0030
```


MSDN [1](#), [2](#), [3](#):

- C / c Задаёт формат денежной единицы, указывает количество десятичных разрядов после запятой
- D / d Целочисленный формат, указывает минимальное количество цифр
- E / e Экспоненциальное представление числа, указывает количество десятичных разрядов после запятой
- F / f Формат дробных чисел с фиксированной точкой, указывает количество десятичных разрядов после запятой
- G / g Задаёт более короткий из двух форматов: F или E
- N / n Также задаёт формат дробных чисел с фиксированной точкой, определяет количество разрядов после запятой
- P / p Задаёт отображения знака процентов рядом с числом, указывает количество десятичных разрядов после запятой
- X / x Шестнадцатеричный формат числа

```
double number = 45.08;  
String.Format("{0:f4}", number); // 45,0800
```

```
long number = 12345678910;
String.Format("{0:+# (###) ###-##-##}", number); // +1 (234) 567-89-10

Decimal price = 123.54M;
String s = price.ToString("C", CultureInfo.InvariantCulture); // ¤123.54

value = 123;
Console.WriteLine(value.ToString("00000"));
Console.WriteLine(String.Format("{0:00000}", value)); // Displays 00123

value = 1234567890.123456;
Console.WriteLine(value.ToString("0,0.0", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture, "{0:0,0.0}",
value));
// Displays 1,234,567,890.1
```

StringBuilder

- Класс для создания и преобразования строк
- `using System.Text;`
- эффективен для многократного добавления строк
- Разбивает блоки по 8000 символов, чтобы объект не попадал в Large Object Heap и не пересоздавался для `Append` (начиная с .net 4)

```
StringBuilder sb = new StringBuilder("Text");
Console.WriteLine("Length: {0}", sb.Length); // 4
Console.WriteLine("Capacity: {0}", sb.Capacity); // 16

sb.Append(" 1 ");
sb.AppendFormat("{0} {1}", "First", "Second");
sb.Replace("Second", "Third");
sb.ToString(); // Text 1 First Third
sb.AppendLine();
// Insert
// Remove
// Replace
```

- **String**
 - Небольшое количество операций и изменений над строками
 - Фиксированное количество операций объединения (компилятор может объединить все в одну)
 - Надо выполнять масштабные операции поиска (например `IndexOf` или `StartsWith`)
- **StringBuilder**
 - Неизвестное количество операций и изменений над строками во время выполнения программы
 - Приложению придется сделать множество подобных операций
- Часто проще использовать **`string.Join`** вместо **StringBuilder**

Encoding

- UTF-16 (В C# **Unicode**) Каждый символ по 2 байта
 - некоторые символы идут парами для составления буквы
- UTF-8 кодирует символы от 1 до 4 байт в зависимости от кода.
- UTF-32 все символы в 4 байта
- UTF-7 древний формат. deprecated
- ASCII или производную кодовую страницу
- Понятно, что надо использовать UTF-16 / UTF-8
- Можно управлять ByteOrderMark
- **System.Text.Encoding**

```
String s = "Hi there.";
Encoding encodingUTF8 = Encoding.UTF8;
Byte[] encodedBytes = encodingUTF8.GetBytes(s);

String decodedString = encodingUTF8.GetString(encodedBytes);

String s = Convert.ToBase64String(encodedBytes);
bytes = Convert.FromBase64String(s);
```