



# Онлайн- образование





# Меня хорошо видно && слышно?

Ставьте +, если все хорошо  
Напишите в чат, если есть проблемы



Проверить, идет ли запись!



# Правила вебинара



Активно участвуем

Задаем вопрос в чат или голосом



Off-topic обсуждаем в Telegram



Вопросы вижу в чате, могу ответить не сразу





# Символы и Строки

---

Тызахян Луспарон



# Цели и смысл вебинара

- 1 Познакомимся с понятием кодировки и их вариантами
- 2 Рассмотрим типы `char` и `string`
- 3 Изучим особенности работы со строками в `c#`

# Преподаватель



Тызыхян Луспарон

.Net разработчик Mindbox

# Маршрут вебинара

Кодировки

char


string

Что можно делать с char и string

Кодинг

Рефлексия, ответы на вопросы





# Про кодировки



# Определение

Кодировка – таблицы сопоставления символа и последовательности байт

Примеры

- ASCII
- Unicode
- CP1251
- KOI-8



# ASCII

7-битная кодировка (т.е. доступно 7 бит для кодирования символов)

- Максимум – 128 символов
- Символы
  - Английский алфавит
  - Цифры
  - Знаки препинания
  - Управляющие символы



# ASCII

dec	hex	oct	char	dec	hex	oct	char	dec	hex	oct	char	dec	hex	oct	char
0	0	000	NULL	32	20	040	space	64	40	100	@	96	60	140	`
1	1	001	SOH	33	21	041	!	65	41	101	A	97	61	141	a
2	2	002	STX	34	22	042	"	66	42	102	B	98	62	142	b
3	3	003	ETX	35	23	043	#	67	43	103	C	99	63	143	c
4	4	004	EOT	36	24	044	\$	68	44	104	D	100	64	144	d
5	5	005	ENQ	37	25	045	%	69	45	105	E	101	65	145	e
6	6	006	ACK	38	26	046	&	70	46	106	F	102	66	146	f
7	7	007	BEL	39	27	047	'	71	47	107	G	103	67	147	g
8	8	010	BS	40	28	050	(	72	48	110	H	104	68	150	h
9	9	011	TAB	41	29	051	)	73	49	111	I	105	69	151	i
10	a	012	LF	42	2a	052	*	74	4a	112	J	106	6a	152	j
11	b	013	VT	43	2b	053	+	75	4b	113	K	107	6b	153	k
12	c	014	FF	44	2c	054	,	76	4c	114	L	108	6c	154	l
13	d	015	CR	45	2d	055	-	77	4d	115	M	109	6d	155	m
14	e	016	SO	46	2e	056	.	78	4e	116	N	110	6e	156	n
15	f	017	SI	47	2f	057	/	79	4f	117	O	111	6f	157	o
16	10	020	DLE	48	30	060	0	80	50	120	P	112	70	160	p
17	11	021	DC1	49	31	061	1	81	51	121	Q	113	71	161	q
18	12	022	DC2	50	32	062	2	82	52	122	R	114	72	162	r
19	13	023	DC3	51	33	063	3	83	53	123	S	115	73	163	s
20	14	024	DC4	52	34	064	4	84	54	124	T	116	74	164	t
21	15	025	NAK	53	35	065	5	85	55	125	U	117	75	165	u
22	16	026	SYN	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	ETB	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	CAN	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM	57	39	071	9	89	59	131	Y	121	79	171	y
26	1a	032	SUB	58	3a	072	:	90	5a	132	Z	122	7a	172	z
27	1b	033	ESC	59	3b	073	;	91	5b	133	[	123	7b	173	{
28	1c	034	FS	60	3c	074	<	92	5c	134	\	124	7c	174	
29	1d	035	GS	61	3d	075	=	93	5d	135	]	125	7d	175	}
30	1e	036	RS	62	3e	076	>	94	5e	136	^	126	7e	176	~
31	1f	037	US	63	3f	077	?	95	5f	137	_	127	7f	177	DEL



# Unicode

Кодировка, содержащая почти все символы мирового алфавита

- Пространство – 4 байта (около 4 млрд. возможных символов)
- Содержит в себе ASCII
- Есть блоки под национальные алфавиты
- Есть зарезервированные места под новые символы
- Есть даже эмодзи

# Примеры диапазонов

- U+0000...U+007F – Латиница
- U+0400...U+04FF – Кириллица
- U+0600...U+06FF – Арабское письмо
- U+16A0...U+16FF – Руны
- U+1F600...U+1F64F – Эмодзи



# UTF-8, UTF-16, UTF-32

- UTF-8 – кодирует символы Юникода в размере от 1 до 4 байт.  
Основная кодировка в веб-приложениях
- UTF-16 – кодирует каждый символ в виде одного или двух 16-битных слов. Используется в CLR Для кодирования
- UTF-32 – кодирует каждый символ в виде 32-битного числа  
(прямой номер символа в юникоде)

# UTF-8

Диапазон номеров символов	Требуемое количество октетов
00000000-0000007F	1
00000080-000007FF	2
00000800-0000FFFF	3
00010000-0010FFFF	4

Количество октетов	Значащих бит	Шаблон
1	7	0xxxxxxx
2	11	110xxxxx 10xxxxxx
3	16	1110xxxx 10xxxxxx 10xxxxxx
4	21	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx



# UTF-16



Символы Unicode до  $FFFF_{16}$  включительно (исключая диапазон для суррогатов) записываются как есть 16-битным словом.

Символы же в диапазоне  $10000_{16}..10FFFF_{16}$  (больше 16 бит) кодируются по следующей схеме:

- Из кода символа вычитается  $10000_{16}$ . В результате получится значение от нуля до  $FFFF_{16}$ , которое помещается в разрядную сетку 20 бит.
- Старшие 10 бит (число в диапазоне  $0000_{16}..03FF_{16}$ ) суммируются с  $D800_{16}$ , и результат идёт в ведущее (первое) слово, которое входит в диапазон  $D800_{16}..DBFF_{16}$ .
- Младшие 10 бит (тоже число в диапазоне  $0000_{16}..03FF_{16}$ ) суммируются с  $DC00_{16}$ , и результат идёт в последующее (второе) слово, которое входит в диапазон  $DC00_{16}..DFFF_{16}$ .



# UTF-16

\* = yyyyyyyyyyxxxxxxxxxx /  - 0x10000  
W1 = 110110yyyyyyyyyy / 0xD800 + yyyyyyyyyy  
W2 = 110111xxxxxxxxxx / 0xDC00 + xxxxxxxxxxxx





Base64





# Base64

Стандарт кодирования двоичных данных при помощи 64символа алфавита Используются

- 0-9 (10 символов)
- A-Z, a-z (52 символа)
- 2 символа в зависимости от реализации

Рассмотрим реализацию MIME (используется в электронной почте)

- Используем дополнительно '+' и '/' и добиваем для кратности 3 символом '='



# Преобразуем строку

1. Строка **IBM** → в байтах UTF-8 → **01001001 01000010 01001101**

2. Разобьем каждые 3 байта в 4 группы по 6 бит

**01001001 01000010 01001101** → **010010 010100 001001 001101**

3. Каждая группа - двоичный индекс строки

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/  
                                 **J****K****L****M****N****O****P****Q****R****S****T****U**

4. Таким образом **010010 010100 001001 001101** → **SUJN**

**Итого:** *Base64*(**IBM**) → **SUJN**

```
var privet = "Привет";

var bytes =
Encoding.UTF8.GetBytes(privet); var b64 =
Convert.ToBase64String(bytes);

var checkBytes =
Convert.FromBase64String(b64); var s =
Encoding.UTF8.GetString(checkBytes);

Console.WriteLine(s);
```





char





# char

**char** – тип символа

- 2 байта
- UTF-16

Можно указывать значения

- Явно

```
char c = 'j';
```

- Escape-последовательность \x с шестнадцатиричном представлении

```
char c = '\x6A'; // После \x от 1 до 4 символов
```

- Escape-последовательность \u с шестнадцатиричном представлении Unicode

```
char c = '\u006A';
```





string





# string

String – последовательность UTF-16 символов

```
var s =
```

```
"Привет"; var
```



```
em = " ";
```



# @ верbatim

@"Строка" – выводит строку «буквально»

```
var s1 = "c:\\Windows\\System32";  
var s2 = @"c:\Windows\System32";  
var s3 = @"Текст с ""двойной  
кавычкой""";  
;
```

# \$

## ИНТЕРПОЛЯЦИЯ

Позволяет писать строки с выражениями

```
var who = "otus";  
var s1 = "Привет {who}";  
var s2 = $"Привет {who}";  
Console.WriteLine(s1); // Привет {who}  
Console.WriteLine(s2); // Привет otus
```



# \$ интерполяция - формат

{<Выражение>[,<Выравнивание>][:<Формат>]}

- <Выражение> - Выражение (функция, переменная,
- <Выравнивание> - (необяз.) Количество символов для выравнивание (положительное – по правому краю, отрицательное – по левому)
- <Формат> - (необяз.) формат выражения

# \$ интерполяция - пример

Позволяет писать строки с выражениями

```
Console.WriteLine($"Сегодня | {DateTime.Now, 20:dd'/'мм'/'yyyy} |");
```



```
Сегодня | 24/11/2021 |
```



The background of the image is an aerial photograph of a city skyline, likely New York City, featuring numerous skyscrapers. The image is color-graded with a blue and green tint. A horizontal band across the middle contains a network of white lines and dots on a teal-to-blue gradient background. The text "StringBuilder" is centered within this band.

# StringBuilder



# Конкатенация строк

Можно так

```
var privet =  
"Привет"; privet +=  
" Otus";
```

И так

```
var privet = string.Concat("Привет", " Otus");
```

Но есть недостатки

- Если строки конкатенируются в цикле – создается много строк
- Как следствие – тратится лишняя память и время на создание строк



# Конкатенация строк

Решение - StringBuilder

# Конкатенация строк

StringBuilder – класс для конкатенации строк

```
var me = "Эдгар";  
// Объявляем класс - StringBuilder  
var sb = new StringBuilder();  
  
// Соединяем строк  
sb.Append("Привет")  
; sb.Append("Otus");  
sb.AppendFormat(", Меня зовут {0}", me);  
  
// Получаем итоговую строку  
var res = sb.ToString();  
Console.WriteLine(res);
```





# Особенности работы со строками



# Иммутабельность

Созданная строка – не меняется, совсем

```
var plus = "Плюс";  
var minus = plus.Replace("Плюс", "Минус"); // Создалась новая строка!  
Console.WriteLine($"{plus}, {minus}"); // Плюс, Минус  
  
plus[1] = 'У'; // Даже не скомпилируется (в языке Си – можно)
```



# Длина строки

String.Length – количество **char** в строке, не символов

```
var privet = "Привет";  
Console.WriteLine(privet.Length); // 6
```

```
var em = "🤖 ";  
Console.WriteLine(em.Length); // 2 – поскольку 🤖 один char  
// не помещается, нужно 2
```

```
// Нам поможет класс StringInfo  
var si = new StringInfo(em);  
Console.WriteLine(si.LengthInTextElements); // 1
```

# Сравнение строк

**String – класс**, но оператор `==` сравнивает **по значению**

```
var s1 =  
"Привет"; var s2  
= "Привет";  
var areEqual = s1 == s2;  
// В Java – false  
// В C# – true
```



# ReferenceEquals

**ReferenceEquals** – сравнение, что две ссылки ссылаются на один объект, а не конкретные значения

```
var vet = "ВЕТ";  
var s1 = $"При  
{vet}"; var s2 =  
$"При{vet}"; // true  
Console.WriteLine(ReferenceEquals(s1, // false  
s2));
```

# Интернирование

```
var s1 = "Привет";  
var s2 = "При" + "вет";  
Console.WriteLine(s1 ==          // true  
s2);  
Console.WriteLine(ReferenceEquals(s1, s2)); // ?
```



# Интернирование

```
var s1 = "Привет";  
var s2 = "При" + "вет";  
Console.WriteLine(s1 ==          // true  
s2);  
Console.WriteLine(ReferenceEquals(s1,          // True  
s2));
```

# Интернирование

- В CLR (среда выполнения C#) – оптимизирована работа со строками
- Существует таблица – пул интернирования
- Пул интернирования – список всех используемых строк в программе
- Если строка задана явным образом – автоматически помещается в пул интернирования
- Соответственно – если у двух переменных явно указаны значения строк, то они ссылаются на одну переменную
- Если строка создается динамически (ввод пользователем, генерация функцией) – они не интернируются → занимают доп. память
- Но строки можно интернировать – при помощи `string.Intern` – и получить ссылку на строку из пула
- Если строка уже в пуле – `string.Intern` возвращает



# Интернирование

```
var vet = "вет";  
var s3 = $"При{vet}"; // s3 и s4 – ссылаются на разные строки  
var s4 = $"При{vet}"; // выделена память для обеих
```

```
Console.WriteLine(s3 == s4); // true  
Console.WriteLine(ReferenceEquals(s3, s4)); // false
```

```
s3 = string.Intern(s3); // Помещаем s3«Привет» - в пул интернирования  
Console.WriteLine(ReferenceEquals(s3, s4)); // false, поскольку s4 – не в пуле
```

```
s4 = string.Intern(s4); // «Привет» уже в пуле – получаем существующую ссылку  
Console.WriteLine(ReferenceEquals(s3, s4)); // теперь s3 и s4 ссылаются на одну  
// переменную из пула
```





ИТОГИ





# Итоги - тезисы

1 Рассмотрели понятие кодировки и их примеры

2 Подробнее изучили понятия строки и символа в C#

3 Разобрались какие операции доступны для работы со строками







Заполните, пожалуйста,  
опрос о занятии по ссылке в  
чате

