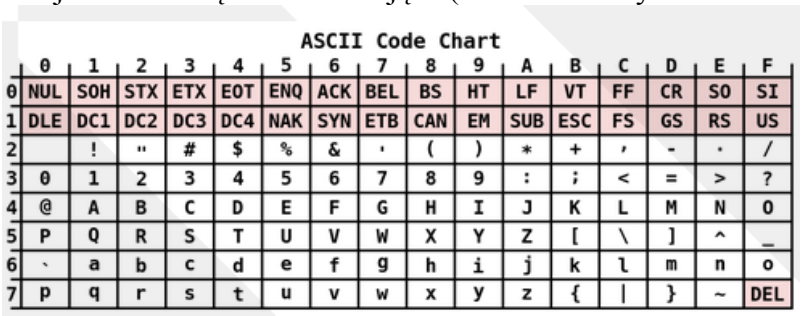
ITA2, ASCII, Unicode

ITA1 - pierwszy binarny kod zapisu znaków, powstał w 1874 r. Jego autorem jest Emile Baudot (od jego nazwiska pochodzi jednostka szybkości transmisji, 1 Bd = 1 bit/s).
Kod Badota był wykorzystywany do telegrafii (telegraf zbudował sam Baudot), we Francji od 1877 r., później w wielu innych krajach.
ITA2 (International Telegraph Alphabet No 2) – zmodyfikowany kod Baudota, którego autorem jest Donald Murray. Murray wprowadził znaki sterujące (m.in. CR, LF, BELL). Kod Murraya z 1901 r. został w 1924 r. międzynarodowym standardem (ITA2) i jest do dziś używany.
TTY (teletypewriter, teleprinter, teletype) – elektromechaniczna maszyna do pisania, która może nadawać i odbierać tekst. Telegraf drukował tekst na taśmie (np. telegraf Baudota, 1877 r.), dopiero ok. 1920 r. powstały urządzenia drukujące na kartkach (np. Creed, 1924 r.).
Telex – (telegraph exchange), sieć TTY pomysłu Bell Labs z 1930 r. z kodem ITA2, stała się międzynarodowym systemem łączności TTY jest też nazwą wirtualnego terminala w systemach Unix/Linux
ASCII (*American Standard Code for Information Interchange*) – rozszerzenie kodu ITA2, opracowane przez ANSI w 1963 r. Początkowo planowano zastąpienie ITA2 przez kod ASCII, jednak pomysł się nie przyjął. Standard wykorzystano w nowych systemach telekomunikacyjnych i w komputerach ASCII jest kodem 7-bitowym, przeznaczonym również do transmisji tekstu – stąd znaki sterujące (niektóre identyczne z ITA2)



ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

EASCII (Extended ASCII) – rozszerzenie kodu ASCII, kodowanie znaków 8-bitowe, stosowane w komputerach ze względu na zgodność z jednostką pamięci (1 byte = 8 bit). Dodatkowe 128 znaków (w porównaniu do ASCII) zostało wykorzystane m.in. do zapisu znaków diakrytycznych (np. ą, ó, …) Różne języki mają różne znaki diaktrytyczne – opracowano wiele wariantów kodowania znaków – tzw. strony kodowe (code page).
Istnieje wiele alternatywnych wariantów kodowania np. polskich znaków – zgodne z normami polskimi i międzynarodowymi oraz wprowadzane przez producentów sprzętu lub oprogramowania. Powstało co najmniej 30 wersji kodowania polskich znaków

Unicode – zestaw znaków, mający obejmować wszystkie rodzaje pisma i języki używane na świecie. Docelowo powinien zastąpić kod ASCII i strony kodowe.

- Każdy znak, w dowolnym języku, ma w Unicode unikalny numer.
  Znika problem zgodności strony kodowej i "wykrzaczania" tekstu.
- Unicode obejmuje ok. 140 tys. znaków, z czego pierwsze 65 tys. (tzw. BMP, Basic Multilingual Plane) zawiera znaki wszystkich współczesnych języków.
- Unicode może być zapisywany na kilka sposobów, m.in.:
  – UTF-16 – stosowany w Windows 7,
  znaki pierwszego BMP zapisywane w 2 bajtach (UInt16), pozostałe 4 bajty
  – UTF-8 – stosowany w Internecie,
  znaki ASCII – 1 bajt, alfabety europejskie – 2 bajty, inne 4+ bajty

Znaki i łańcuchy

- Do zapisu znaków używany jest Unicode (1 znak = 2 bajty, UInt16), łańcuchy to sekwencje znaków
  `Char (char)`
  `String (string)`
- Unicode eliminuje kłopoty ze znakami diakrytycznymi, stronami kodowymi itp.
- Obie struktury są wyposażone w liczne metody – większość zadań związanych z przetwarzaniem znaków i łańcuchów jest już gotowa
- W przetwarzaniu znaków i łańcuchów są uwzględniane ustawienia regionalne (tzw. lokalizacja) – domyślnie wg ustawień systemu,
  ale można to zmienić
- W C# łańcuchy są traktowane w specjalny sposób – raz nadana wartość nie może być modyfikowana, zamiast tego tworzona jest nowa struktura (z nową wartością), a stara jest usuwana z pamięci przez GC, zupełnie inaczej niż pozostałe typy proste, w których po prostu zmienia się wartość istniejącej cały czas, tej samej zmiennej.
  => Programy intensywnie przetwarzające łańcuch mogą z tego powodu działać wolniej – każda zmiana wartości łańcucha, choćby jednego jego znaku, powoduje utworzenie nowej struktury i pozbycie się starej. Aby tego uniknąć należy stosować obiekt StringBuilder
- Stałe znakowe są zapisywane w apostrofach, na kilka sposobów

```
Char a, b, c, d, e;
a = 'A';
b = '\u0042'; // "B", numer znaku Unicode
c = '\x0043'; // "C", notacja szesnastkowa
d = (Char)68; // "D", konwersja z liczby całkowitej
e = '\t';     // <tab>, tzw. sekwencja ucieczki
```

- Stałe łańcuchowe są zapisywane w cudzysłowach:

```
String s1, s2;
s1 = "Hello\r\nworld";
s2 = @"d:\archiwum\plik.zip";
```

- Normalnie wykrywane są sekwencje ucieczki, m.in.:
  "\t" – tabulator, "\r" – CR, "\n" – LF, "\\" – znak "\"
  Użycie "@" wyłącza sekwencje ucieczki – przydatne, gdy w łańcuchu występują znaki "\", np. w ścieżkach do plików
- Dla łańcuchów zdefiniowano operatory:
  = - łańcuch jest typem prostym, więc '=' kopiuje wartość
  + - dla łańcuchów oznacza konkatenację
  += - połączenie dwóch poprzednich operacji
  [ ] - indeksowanie, dostęp do znaków, READ ONLY
  == != - równość
  nie ma operatorów relacji – jest met. *Compare*

- Jako że każdy typ C# ma funkcję ToString (w pewnym sensie jest to konwersja na typ String), to do łańcucha można "dodać" wartość zmiennej lub stałej dowolnego typu

```
Int32 n = 13;
String s = "Liczba: " + n + ", kwadrat: " + n * n;
```

- Dla znaków nie ma żadnych specyficznych operatorów,
  natomiast wewnętrznie Char jest tożsamy z UInt16 – jest więc liczbą,
  obowiązują zwykłe zasady dotyczące konwersji:

```
Int32 n = ('#' + 'A') * 'd'; // 10000
Char x = 'A' + 1;            // błąd
Char B = (Char)('A' + 1);    // 'B'
Char a = (Char)('A' + 32);   // 'a'
```

- Każda operacja arytmetyczna UInt16 (lub Char) powoduje niejawną konwersję na Int32, natomiast konwersja w drugą stronę jest możliwa, ale musi być jawna
- Działając na zmiennych lub literałach znakowych, bez potrzeby
  jawnej konwersji można korzystać z operatorów:
  = - przypisanie
  == != - równość
  >= > < <= - relacja (kody znaków, nie pozycja w alfabecie!)

Metoda ToString

Wszystkie typy C# są wyposażone w metodę ToString()
(dziedziczoną po typie *object*, czyli wspólnym dla CTS):

- Domyślnie metoda zwraca łańcuch z nazwą typu – struktury lub klasy (zatem każda zmienna może być argumentem Console.WriteLine(), choć w wielu przypadków niewiele z tego wynika)
  Istnieje tylko jedna wersja tej metody, bezargumentowa
- Dla typów prostych metoda zwraca łańcuch z alfanumerycznym zapisem wartości, przy domyślnych opcjach formatowania
- Niektóre typy złożone mają specjalne wersje metody, adekwatne do zawartości informacyjnej, np. DateTime
  Są cztery wersje tej metody; można podać łańcuch formatujący i/lub obiekt *IFormatProvider*, co umożliwia stosowanie lokalizacji

Interfejs IFormatProvider

IFormatProvider jest wspólnym interfejsem kilku klas obiektów sterujących formatowaniem danych. Są to obiekty klas:

- NumberFormatInfo
  określa formatowanie liczb, m.in. symbol waluty, znak dziesiętny
- DateTimeFormatInfo
  określa formatowanie daty i czasu, m.in. rodzaj kalendarza, separatory w dacie i godzinie, kolejność danych w dacie,
- CultureInfo
  reprezentuje określoną "kulturę" – język i kraj, obejmuje wszystkie aspekty formatowania liczb, daty i czasu oraz sortowania łańcuchów

Konstruktor klasy CultureInfo ma dwa argumenty:

- Numer lub nazwa (łańcuch) kultury,
  - skrót nazwy języka, np. "pl", "en" albo
  - skrót nazwy języka i kraju, np. "pl-PL", "en-GB", "en-US"
- Flaga useUserOverride
  wartość typu Boolean (domyślnie *true*), decydująca o uwzględnieniu zmian w "kulturze", jeżeli takie są zapisane w systemie operacyjnym, (np. jeżeli użytkownik

zmieni format daty albo strefę czasową). MS zaleca uwzględniać ustawienia użytkownika, gdyż zapewnia to spójność danych przetwarzanych w różnych programach

```
CultureInfo culturePl = new CultureInfo("pl-PL");
```

Obiekt implementujący interfejs IFormatProvider, np. CultureInfo, można użyć m.in. przy wywołaniu ToString() oraz String.Format():

```
CultureInfo ciGB = new CultureInfo("en-GB");
DateTime dt = DateTime.Now;
String s;

s = dt.ToString(ciGB);
s = dt.ToString("dddd, dd MMMMM yyyy", ciGB);

s = String.Format(ciGB, "Data/czas: {0}", dt);
```

Metody Parse i TryParse

Wszystkie typy proste C# (oprócz String) są wyposażone w metody Parse() oraz TryParse(), działające odwrotnie niż ToString():

- Metoda Parse() może zakończyć się niepowodzeniem,
  co skutkuje rzuceniem wyjątku i zatrzymaniem programu:

```
Int32 a;
a = Int32.Parse("77");   // ok
a = Int32.Parse("dwa");  // błąd FormatException
```

- Metoda TryParse() nigdy nie rzuca wyjątku, zaś o powodzeniu lub błędzie konwersji świadczy wartość rezultatu:

```
Int32 a;
Boolean ok;
ok = Int32.TryParse("trzy", out a); // ok = false, a = 0
```

Metody Parse() oraz TryParse() są przeciążone:

- Parse()
- Parse(String, NumberStyles)
  Obiekt *NumberStyles* umożliwia określenie np. czy dozwolone są spacje wiodące, symbol waluty, separatory tysięcy itd.
- Parse(String, IFormatProvider)
  Obiekt IFormatProvider jest związany z lokalizacją; domyślnie ustawienia są takie jak w systemie operacyjnym, ale można podać obiekt dowolnego innego języka/kraju, służy do tego CultureInfo:

```
CultureInfo ciGB = new CultureInfo("en-GB");
Double x;
x = Double.Parse("1.5");           // błąd, pow. być 1,5
x = Double.Parse("1.5", ciGB);     // ok
```

- Parse(String, NumberStyles, IFormatProvider)

Klasa Convert

Klasa Convert dostarcza metody służące do konwersji typów prostych oraz typu DateTime na inne typy proste
(zawiera m.in. funkcjonalność metod Parse() oraz ToString())

- Konwersje z/na typ String może uwzględniać IFormatProvider:

```
String s;
Double x;
DateTime dt = DateTime.Now;

x = Convert.ToDouble("1,5");       //ok
x = Convert.ToDouble("1.5");       //błąd
x = Convert.ToDouble("1.5", ciGB);     //ok.

s = Convert.ToString(dt);          // 2018-12-10 13:45:25
s = Convert.ToString(dt, ciGB);    // 12/10/2018 3:45:25PM
```

Klasa Char

Klasa Char ma wiele statycznych metod, większość z nich służy do sprawdzania rodzaju znaku; działanie wynika wprost z nazwy:

- IsControl – czy jest znakiem sterującym
- **IsDigit** – czy jest cyfrą
- **IsLetter** – czy jest literą
- IsLetterOrDigit
- IsLower, IsUpper – czy jest małą/dużą literą
- IsNumber – czy jest liczbą (cyfry, ułamki, cyfry rzymskie itp.)
- IsPunctuation – czy jest znakiem interpunkcyjnym
- **IsWhiteSpace** – czy jest znakiem "białym" (spacja, tabulator, CR, LF)

Wszystkie te funkcje mają dwie wersje:

```
Boolean IsDigit(Char c)                 // znak c
Boolean IsDigit(String s, Int32 i)      // znak i łańcucha
```

Klasa String

Klasa String ma wiele statycznych i dynamicznych metod do manipulowania łańcuchami.
Metody (wywoływane przez obiekt):

- **CompareTo** – porównuje ze wskazanym łańcuchem; porządek alfabetyczny zgodnie z aktualną "kulturą"; rezultat: -1, 0, +1
- EndsWith, StartsWith – czy łańcuch kończy/zaczyna się łańcuchem
- Equals – czy jest równy wskazanemu łańcuchowi
- **IndexOf** – pozycja wskazanego łańcucha lub znaku (-1 gdy nie ma)
- LastIndexOf – pozycja ostatniego wystąpienia wskazanego łańcucha
- **Insert** – wstawia łańcuch we wskazane miejsce
- **Replace** – zamienia każde wystąpienie łańcucha na inny łańcuch
- **Split** – dzieli łańcuch na tablicę łańcuchów wg podanego separatora
- Substring – dostarcza wskazany fragment łańcucha
- **Trim** – usuwa "white space" z początki i końca
- ToLower, ToUpper – zamienia wszystkie litery na małe/duże

Funkcje (tj. metody statyczne, wywoływane przez klasę):

- Compare – porównuje dwa łańcuchu, porządek alfabetyczny;
  opcjonalna flaga *ignoreCase* – ignorowanie różnic wielkości liter
- **Concat** – łączy łańcuchy lub rezultaty *ToString()* dowolnych argumentów (oddzielonych przecinkami, tablicy lub listy)
- **Format** – tworzy łańcuch z dowolnej listy wartości, na podstawie łańcucha formatującego i opcjonalnie "kultury"
- **Join** – łączy łańcuchy lub rezultaty *ToString()* podobnie jak Concat, ale dodając separator, podawany jako pierwszy argument

Funkcja String.Format

Funkcja String.Format

- Metryczka:

```
String Format (
  IFomatProvider provider,    // "kultura"
  String format,              // łańcuch formatujący
  params object [] args       // dane (dowolnie wiele)
);
```

- Szczególne znaczenie ma atrybut *params* – oznacza, że można jako argument podać tablicę elementów albo ich listę (oddzielonych ",");
  Wywołania *String.Format* poniżej są równoważne:

```
object[] obiekty = { "Liczba", 123, 7.7, true, 'x' };
s = String.Format("{0} {1} … {4}", obiekty);

s = String.Format("{0} {1} … {4}",
      "Liczba", 123, 7.7, true, 'x' );
```

Łańcuch formatujący i formatowanie

- Łańcuch formatujący **może** zawierać dowolny tekst – jest on przenoszony do rezultatu bez zmian – oraz **musi** zawierać formatowanie dla każdego argumentu:

```
"dowolny {0} tekst {1}", a, b
```

  kolejnym argumentom odpowiadają formatowania, zaczynając od {0}
- Formatowania mogą wystąpić w dowolnej kolejności (np. {2} {0} {1}) i mogą się dowolnie wiele razy powtarzać (np. {0} {0} {1} {0})
- Formatowanie ma postać:

```
{ numer, wyrównanie/długość : format}
```

  przy czym długość i format są opcjonalne, np.

```
{0} {1,3} {2:F}, {3,-5:X}
```

- Długość oznacza **minimalną** liczbę znaków, jakie zajmie argument; domyślne jest wyrównanie do prawej (wynik jest w razie potrzeby uzupełniany spacjami), a jeżeli długość jest ujemna, to do lewej:

```
{0,6}     // "   123"
{0,-6}    // "123   "
```

- Formaty można podawać na dwa sposoby:
  - jednoliterowy kod formatowania, opcjonalnie uzupełniony przez liczbę oznaczającą "precyzję", np.:

```
{0:C2} {1:X4}
```

  - ciąg znaków formatowania użytkownika, np.:

```
{0:#,##0.00} {1:dddd, dd MMMMM yyyy}
```

- Znaczenie formatu (tak kodów, jak i znaków formatu użytkownika) zależy od typu argumentu, który zostanie mu przyporządkowany

Kody formatowania typów prostych;

- C – waluta (currency), zgodnie z "kulturą", która określa symbol waluty, użycie go przed albo po liczbie i odstęp od niej, precyzję, znak dziesiętny, separatory tysięcy, formatowanie wartości ujemnych; precyzja określa liczbę cyfr po przecinku
- D – wartości całkowite; precyzja – liczba cyfr, ew. uzup. zerami
- E – format "naukowy" (np. 1,23e-6); precyzja – liczba cyfr po ","
- F – format stałoprzecinkowy (np. 0,0001); precyzja jw.
- G – format uniwersalny, E albo G, zależnie od wartości i precyzji
- N – format numeryczny, z separatorami tysięcy, reszta jak F
- P – format procentowy, np. 0,25 -> 25%; precyzja jak w F
- R – format precyzyjny (*round robin*), gwarantuje że po konwersji łańcucha do liczby, liczba będzie miała dokładnie tę samą wartość
- X – format szesnastkowy, reszta jak D, np. {0:X4} -> 00F7

Wybrane znaki formatowania typów prostych:

- # – opcjonalna cyfra
- 0 – obowiązkowa cyfra
- . – znak dziesiętny (z bieżącej "kultury")
- , – separator (z bieżącej "kultury"), może wystąpić wielokrotnie, albo skalowanie (/1000), np. {0:#0,,.00} wyświetli wynik w milionach (skalowanie jest tylko gdy "," lub seria "," jest tuż przed ".")
- E e – formatowanie wykładnika notacji naukowej (np. {0:0.00e+#0} -> 1,23e+2)

Struktura DateTime

Struktura DateTime jest jedną z kilku podobnych struktur do zapisu daty i/lub czasu. Ma też bogate formatowanie, dla wielu "kultur"

- Właściwość Now (*DateTime.Now*)– bieżąca data i czas:

```
DateTime dt = DateTime.Now;
```

- Rok, miesiąc, dzień, godzina, itd. – właściwości obiektu DateTime
- Konstruktor tworzy obiekt z liczb (rok, miesiąc, dzień, godzina, minuta sekunda) albo liczby Int64 (liczba odc. 0,1μs od 1.01.001, 00:00; obecnie, 12.2018, w tej mierze mamy ok. 636,808e15 ≈2017,9 lat):

```
dt = new DateTime(2018, 12, 10, 14, 32, 55);
dt = new DateTime(2018, 12, 10);
```

- Metody (statyczne i dynamiczne) umożliwiające manipulowanie wartościami, np.:

```
DateTime teraz = DateTime.Now;
DateTime zaRok = teraz.AddYears(1);
Boolean przestępny = DateTime.IsLeapYear(zaRok.Year);
```

- Metody Parse() i ToString()

Kody formatowania typu DateTime:

- d – krótki wzorzec daty — 2018-12-10
- D – długi wzorzec daty — 10 grudnia 2018
- f – pełen wzorzec daty i krótki czasu — 10 grudnia 2018 21:12
- F – pełen wzorzec daty i czasu — 10 grudnia 2018 21:12:45
- g – krótki wzorzec daty i krótki czasu — 2018-12-10 21:12
- G – krótki wzorzec daty i długi czasu — 2018-12-10 21:12:45
- m – dzień miesiąca — 10 grudnia
- y – miesiąc roku — grudzień 2018
- t – krótki wzorzec czasu — 21:12
- T – długi wzorzec czasu — 21:12:45
- s – wzorzec daty i czasu ISO 8601 — 2018-12-10T21:12:45
- u – wzorzec czasu uniwersalnego — 2018-12-10 21:12:45Z

Wybrane znaki formatowania typu DateTime :

- d, dd – dzień miesiąca (jedna/dwie cyfry)
- ddd, dddd – dzień tygodnia (skrót/całość)
- M, MM, MMM, MMMM, MMMMM – miesiąc
  (jedna/dwie cyfry, nazwa skrót/cała/dopełniacz)
- h, hh, H, HH – godzina (format 12/24 godzinny, jedna/dwie cyfry)
- mm – minuta
- ss – sekunda
- f … ffffff, F … FFFFFF – różne warianty ułamka sekundy
- K – strefa czasowa
- tt – "AM/PM" – w języku polskim nie występuje

```
// wtorek, 10 grudnia 2018, 21:12:45
DateTime.Now.ToString("dddd, dd MMMMM yyyy, HH:mm:ss")
```

Klasa Console

- Definiująca szereg statycznych metod do obsługi konsoli (odpowiednik strumieni cin oraz cout w C++)
- Najważniejsze metody:
  `Write`
  `WriteLine`
  `ReadLine`
  `ReadKey`

Write i WriteLine – wyświetla tekst, ta druga dodatkowo przechodzi do nowej linii

- Wyświetlenie wartości zmiennych i literałów:

```
Console.WriteLine("Hello");
Console.WriteLine(x);
Console.WriteLine("Wynik = " + x);
```

  Można użyć jawnie ToString, żeby wskazać formatowanie:

```
Console.WriteLine(x.ToString("format"));
```

- Wyświetlenie danych formatowanych

```
DateTime now = DateTime.Now;
Console.WriteLine("Dziś {0:dddd}, godz. {0:T}", now);
```

  Działanie Write i WriteLine (w tym ostatnim ujęciu) jest identyczne jak String.Format, tylko nie można wskazać "kultury"

ReadLine – wczytuje tekst wpisany z klawiatury

- Powoduje wstrzymanie działania programu do naciśnięcia <Enter>, następnie zwraca wpisany tekst – ale bez znaków CR-LF
- Pojedyncze wartości można łatwo konwertować metodą Parse:

```
Double x = Double.Parse(Console.ReadLine());
```

- Większe ilości danych można podzielić metodą Split obiektu String, następnie przetwarzać oddzielnie:

```
String ln = Console.ReadLine();
String[] src = ln.Split(',');
for (…
```

- Jeszcze wygodniejsze jest użycie funkcji rozszerzających klasy Array i/lub kolekcji; Trzeba pamiętać, aby na koniec użyć ToArray:

```
String ln = Console.ReadLine();
Int32[] src = ln.Split(',').
  Select(s => Int32.Parse(s)).ToArray();
```

ReadKey – wczytuje znak wpisany z klawiatury

- Powoduje wstrzymanie działania programu do naciśnięcia dowolnego klawisza lub kombinacji klawiszy, które powodują wpisanie znaku alfanumerycznego lub sterującego, a także niektórych specjalnych
  (np. <Ctrl> nie, ale <Ctrl>+A albo <F12> już tak)
- Rezultat jest typu ConsoleKeyInfo – struktura zawierająca dokładne informacje zarówno o wciśniętym klawiszu, jak i wpisanym znaku:
  - KeyChar (typu Char) – wpisany znak lub '\0', np. dla <F1>
  - Key (typu wyliczeniowego ConsoleKey) – nazwa wciśniętego klawisza każdy klawisz ma inną nazwę, np. 1 = <D1> albo <NumPad1>, która nie zależy np. od użycia <Shift>, np. 2 = @ = <D2>
  - Modifiers (flaga bitowa typu ConsoleModifiers) – informacja które spośród klawiszy <Shift>, <Ctrl> i <Alt> były wciśnięte podczas wpisywania znaku (np. "@" = <Shift><2>):

```
Boolean alt = (key.Modifiers & ConsoleModifiers.Alt) != 0
```

Klasa Array

- Klasa Array jest klasą uniwersalną – reprezentuje tablicę elementów dowolnego typu
- Jest to klasa abstrakcyjna – nie można utworzyć obiektu Array
- Każda tablica, niezależnie od typu elementów składowych, należy do klasy Array (do zmiennej typu Array można przypisać dowolną tablicę), odwrotne przypisanie wymaga rzutowania

```
Int32[] t = new Int32[3];
Array a;
```

- Klasa Array udostępnia wiele przydatnych właściwości i metod;
- Większość z nich to metody statyczne
  (wywoływane przez klasę Array, a nie obiekt tej klasy;
  pierwszym argumentem metody jest tablica)

```
Int32[] t1 = { 1, 2, 3, 4 };
Int32[] t2;
Int32 i;
i = Array.IndexOf(t1, 3);             // 2
Array.Reverse(t1);                    // {4,3,2,1}
t2 = Array.FindAll(t1, n => n%2==0); // {4,2}
Array.Sort(t2);                       // {2,4}
```

Klasa Array – metody statyczne

- CreateInstance – tworzy tablicę obiektów wskazanego typu
- Copy – kopiuje fragment tablicy do innej tablicy
- Find, FindAll – znajduje elementy spełniające określony warunek
- FindIndex – znajduje element spełniający warunek i zwraca jego indeks
- **IndexOf** – znajduje element o określonej wartości i zwraca jego indeks
- Resize – zmienia rozmiar tablicy
- **Reverse** – odwraca kolejność elementów
- **Sort** – sortuje tablicę (jest kilkanaście wersji tej metody, można m.in. podać niestandardowy komparator)

Klasa Enumerable

- Enumerable to klasa statyczna, dostarcza kilkadziesiąt (!) statycznych metod przetwarzania dowolnych obiektów implementujących interfejs IEnumerable, tj. różne kolekcje (klasa Array też)
- Wszystkie metody są zdefiniowane jako tzw. rozszerzenia (extension), mogą być wywoływane na 2 sposoby:
  – przez klasę Enumerable
  – przez obiekt dowolnej klasy implementującej IEnumerable
  (znacznie wygodniej, można wykonać kilka operacji)

```
Int32[] t1 = { 1, 2, 3, 4 };
Double avg;
avg = Enumarable.Average(t1);
avg = t1.Average();
```

- Wiele metod daje jako rezultat kolekcję, jednak aby zapisać ją jako tablicę, należy użyć metody ToArray

```
Int32[] t1 = { 1, 2, 3, 4 };
Int32[] t2;
t2 = t1.Reverse().ToArray();          // {4,3,2,1}
```

- Wywołania wielu kolejnych metod Enumerable można połączyć w łańcuch – kolejna metoda łańcucha przetwarza rezultat dostarczony przez metodę poprzednią

```
Int32[] t3 = { 2, 1, 2, 3, 2, 4 };
Int32[] t4;
t4 = t3.Distinct().Where(n => n<4).
     Reverse().ToArray();              // {3,1,2}
```

Klasa Enumerable, wybrane metody:

- All – sprawdza czy wszystkie elementy spełniają wskazane kryterium
- Any – sprawdza czy choć jeden elementy spełnia wskazane kryterium
- Average – oblicza wartość średnią (elementy muszą być liczbami)
- Concat – łączy dwie kolekcje
- Contains – sprawdza czy kolekcja zawiera wskazany element
- Distinct – zwraca kolekcję elementów unikalnych (bez powtórzeń)
- Except – zwraca kolekcję za wyjątkiem wskazanych (różnica zbiorów)
- Max – zwraca wartość maksymalną (elementy muszą być liczbami)
- Min – zwraca wartość minimalną
- OrderBy – sortuje według wybranego klucza i opcjonalnie komparatora
- **Select** – zwraca kolekcję elementów przetworzonych p. zał. funkcję
- Sum – oblicza sumę (elementy muszą być liczbami)
- **Where** – zwraca podzbiór elementów spełniających zał. kryterium
- ToList – przekształca kolekcję w listę
- **ToArray** – przekształca kolekcję w tablicę

Zadania

Proszę napisać program, który…

1. Wczytuje łańcuch zawierający liczby pomieszane z tekstem, następnie wyszukuje i drukuje wszystkie cyfry; należy użyć metody IsDigit (dostępna przez obiekt typu Char)
2. Jak w zad. 1., ale składa liczby całkowite z występujących bezpośrednio po sobie cyfr
3. * Jak w zad. 2., ale liczby rzeczywiste w notacji zmiennoprzecinkowej
4. ** Jak w zad. 2., ale liczby rzeczywiste w notacji zmiennoprzecinkowej lub naukowej
5. Wczytuje łańcuch z dowolnie wieloma liczbami rozdzielonymi przecinkami, dzieli na podłańcuchy wg pozycji przecinków i konwertuje na liczby, następnie drukuje liczby i ich kwadraty (Dane wejściowe mogą zawierać spacje - są ignorowane przez Parse); należy użyć metod Substring oraz Remove (obie dostępne przez obiekt typu String)
6. Wczytuje łańcuch z dowolnie wieloma liczbami rozdzielonymi przecinkami, zamienia ją na tablicę łańcuchów i zamienia na tablicę liczb Int32. Następnie oblicza kwadraty tych liczb, drukuje dane i wynik obliczeń; należy użyć funkcji rozszerzających Split, Select oraz ToArray (dostępne przez obiekt typu String)
7. Wczytuje łańcuch z dowolnie wieloma liczbami rozdzielonymi przecinkami, zamienia ją na tablicę, zamienia elementy puste na "0", wreszcie zamienia wszystko na tablicę Int32. Następnie oblicza kwadraty tych liczb, drukuje dane i wynik obliczeń (jak w p. 2)
8. Wyświetla wartość np. 1234,5678 z formatowaniem „waluta” dla różnych kultur (polska, angielska, USA, hebrajska, arabska, chińska, japońska, …); efekt należy obserwować w oknie Locals podczas pracy krokowej
9. Jak w zad. 8, ale bieżącą datę i czas
10. Wyświetla informacje o wciśniętym klawiszu i wpisanym z klawiatury znaku, jakie można odczytać ze struktury ConsoleKeyInfo, zwracanej przez Console.ReadKey Program powinien działać w „nieskończonej” pętli, aż do wciśnięcia określonej kombinacji klawiszy, np. <Ctrl>+<End>
11. Pracując w „nieskończonej” pętli wykonuje proste zadania (np. ? – wyświetla menu, a – wczytuje liczbę a, b – wczytuje liczbę b, + – oblicza sumę a i b, * – oblicza iloczyn a i b, x – kończy pracę
12. * Jak w zadaniu 11, ale wykorzystując informacje o wciskanych klawiszach, a nie wpisywanych z klawiatury znakach