# Platforma .NET – Wykład 5
# Programowanie w C# – Część 3/3

Osoba prowadząca wykład, laboratorium i projekt:
dr hab. inż. Marek Sawerwain, prof. UZ

Instytut Sterowania i Systemów Informatycznych
Uniwersytet Zielonogórski

e-mail : M.Sawerwain@issi.uz.zgora.pl
tel. (praca) : 68 328 2321,
pok. 328a A-2,
ul. Prof. Z.Szafrana 2,
65-246 Zielona Góra

Ostatnia kompilacja pliku: Monday 5th June, 2023, t: 23:05



Notatki



## Spis treści





Notatki



## Plan wykładu – spotkania tydzień po tygodniu

(1) Informacje o wykładzie, pojęcie platformy, podstawowe informacje o platformie .NET
(2) Składowe platformy .NET: CLR, CTS, języki programowania, biblioteki klas, pojęcie podzespołu (ang. assembly)
(3) Programowanie w C# – środowisko VS, MonoDevelop, syntaktyka C#, wyjątki, współpraca z DLL
(4) Programowanie w C# – model obiektowy, typy uogólnione, lambda wyrażenia
(5) Programowanie w C# – aplikacje „okienkowe", programowanie wielowątkowe
(6) Programowanie w F# – podstawy, przetwarzanie danych tekstowych,
(*) "Klasówka I", czyli egzamin cześć pierwsza
(7) Dostęp do baz danych



Notatki



## Plan wykładu – tydzień po tygodniu

(8) Język zapytań LINQ, Entity Framework
(9) Obsługa standardu XML
(10) Technologia ASP.NET 1/2
(11) Technologia ASP.NET 2/2
(12) Model widok i kontroler – Model View Controller
(13) Tworzenie usług sieciowych SOAP i WCF (komunikacja sieciowa)
(14) Wykład monograficzny .NET 1
(15) Wykład monograficzny .NET 2
(*) "Klasówka II", czyli egzamin cześć druga



Notatki

## Plan wykładu

1. przeciążanie operatorów i konwersje typów,
   1.1 przeciążanie operatorów, typy anonimowe,
   1.2 operatory „is", „as",
   1.3 zdarzenia oraz delegacje,
2. wątki i wyjątki,
   2.1 wyjątki,
   2.2 przeznaczenie wątków,
   2.3 podstawowe konstrukcje.
3. główne klasy i obiekty aplikacji okienkowych,
   3.1 hierarchia klas,
   3.2 okno aplikacji,
   3.3 system grafiki GDI+.



Notatki



## Przeciążanie operatorów

Język C# pozwala na przeciążanie operatorów w przypadku typów użytkownika poprzez definicję statycznych składowych poprzez użycie słowo kluczowego operator.

```
public static Complex operator +(Complex c1, Complex c2)
```

Nie wszystkie operatory mogą być przeciążone:

| Operatory | Opis |
|---|---|
| `+, -, !, ~, ++, --, true, false` | wymienione operatory unarne mogą zostać przeciążone |
| `+, -, *, /, %, &, \|, ^, <<, >>` | wymienione operatory binarne mogą zostać przeciążone |
| `==, !=, <, >, <=, >=` | operatory relacji mogą zostać przeciążone |
| `&&, \|\|` | operatory logicznie nie mogą zostać przeciążone, choć przeciążenie bitowych operatorów logicznych jest możliwe |
| `[ ]` | operator indeksowania nie może zostać przeciążony, należy wykorzystać indekser |
| `( )` | operator konwersji nie może zostać przeciążony, należy stosować nowe operatory konwersji |
| `+=, -=, *=, /=, \%=, \&=, \|=, ^=, <<=, >>=` | operatorów przypisania nie można przeciążać |
| `=, . , ?: , ->, new, is, sizeof, typeof` | tych operatorów nie można przeciążyć |

Operatory porównania muszą być przeciążane w parach tzn.: == oraz != . Podobnie < i >, oraz <= i >=.



Notatki



## Znane i lubiane liczby zespolone

Krótki program do testowania liczb zespolonych:

```
class TestComplex {
    static void Main() {
        Complex num1 = new Complex(2, 3);
        Complex num2 = new Complex(3, 4);

        Complex sum = num1 + num2;

        System.Console.WriteLine("Liczba zespolona:  {0}", num1);
        System.Console.WriteLine("Liczba zespolona: {0}", num2);
        System.Console.WriteLine("Suma dwóch liczb: {0}", sum);
    }
}
```



Notatki



## Znane i lubiane liczby zespolone

Klasa reprezentująca liczby zespolone:

```
public struct Complex {
    public int real;
    public int imaginary;

    public Complex(int real, int imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }

    public static Complex operator +(Complex c1, Complex c2) {
        return new Complex(c1.real + c2.real, c1.imaginary + c2.imaginary);
    }

    public override string ToString() {
        return (System.String.Format("{0} + {1}radius", real, imaginary));
    }
}
```



Notatki

## Operatory is oraz as

Słowo kluczowe **is** jest wykorzystywane do sprawdzenia, czy podana zmienna jest kompatybilna z określonym innym typem.

```
if (obj is TypeName) {
// kod zwiazany z obj
}
```

Warunek będzie prawdziwy jeśli wyrażenie będzie różne od null oraz możliwa będzie konwersja obiektu do podanego typu bez zgłoszenia wyjątku. Możliwe konwersje, to konwersja referencji (w wyniku dziedziczenia) i konwersje związane z techniką „pudełkowania".

```
class Class1 { }
class Class2 { }
class Class3 : Class2 { }

public static void Test (object o) {
    Class1 a; Class2 b;

    if (o is Class1) {
         Console.WriteLine ("obiekt o jest typu Class1");
         a = (Class1)o;
         // reszta kodu dla typu Class1
    }
    ...
}
```



Notatki



## Operatory is oraz as

Słowo kluczowe „as" jest stosowane do wykonania konwersji pomiędzy kompatybilnymi typami. Ogólnie syntaktyka przedstawia się następująco:

```
expression as Type
```

Jednakże wyrażenie to, jest równoważne następującej konstrukcji:

```
expression is Type ? (Type)expression : (Type)null
```

Przykład zastosowania operatora „as":

```
object[] objArray = new object[6];
objArray[0] = new ClassA(); objArray[1] = new ClassB(); objArray[2] = "hello";
objArray[3] = 123; objArray[4] = 123.4; objArray[5] = null;

for (int i = 0; i < objArray.Length; ++i) {
    string s = objArray[i] as string;
    if (s != null) {
      ...
    }
}
```



Notatki



## Implikowanie typu na podstawie wartości

Słowo kluczowe **var** stosuje wnioskowanie do określenia typu na podstawie podanej wartości np.:

1. `var i = 5;` // liczba int
2. `var s = "Dzień Dobry";` // ciąg znaków – `string`
3. `var a = new[] { 0, 1, 2 };` // tablica `int[]`
4. `var list = new List<int>();` // `List<int>`
5. `var expr = from c in customers where c.City == "London" select c;` // `IEnumerable<Customer>` albo `IQueryable<Customer>`

Wnioskowanie typu w kontekście **var** jest stosowane w następujących przypadkach:

- w przypadku przypisań lokalnych w kontekście słowa **var** ,
- w sekcji inicjalizacyjnej pętli **for** ,
- w sekcji inicjalizacyjnej konstrukcji **foreach**,
- wyrażenia ze słowem **using** (typy zgodne z interfejsem **IDisposable**).



Notatki



## Typy anonimowe

Typ anonimowy to udogodnienie, polegające na tym iż kompilator tworzy klasę z elementami tylko do odczytu na podstawie wartości podanych przy wykorzystaniu notacji inicjalizatora klasy { ... }:

```
var v = new { Amount = 108, Message = "Hello" };
```

Tego rodzaju deklaracje stosuje się zazwyczaj w przypadku klauzuli **select** w zapytaniach LINQ. Anonimowe typy oferują dostęp tylko do pól, metody i własności nie mogą być stosowane, typ anonimowy nie może być rzutowany na inne typy z wyjątkiem klasy **object**.

```
var productQRY =
    from prod in TBLProdukty
    select new { prod.Kolor, prod.Cena };

foreach (var v in productQRY) {
    Console.WriteLine("Kolor={0}, Cena={1}", v.Kolor, v.Cena);
}
```



Notatki

## Czym jest delegat?

Delegat to typ określający sygnaturę metody. Instancja (egzemplarz) delegata pozwala na przypisanie innej metody o kompatybilnej sygnaturze. Przykładowa definicja:

public delegate void DeleteMessage(string messageID);

Typ delegatu jest określany przez nazwę delegatu. Delegat to bezpieczna odmiana wskaźników do funkcji w językach C/C++. I tak dla przykładowej metody:

```
public static void DelegateMethod(string m) {
    System.Console.WriteLine(m);
}
```

Utworzenie instancji obiektu delegatu:

```
DeleteMessage handler = DelegateMethod;
```

Wywołanie podstawionej metody:

```
handler("Hello World");
```



Notatki



## Czym jest delegat?

Delegat to typ, a więc może stanowić typ dla argumentu metody:

```
public void MethodWithCallbackDelegate(int par1, int par2,
                                       DelegateMethod _cb) {
    _cb("Suma parametrów: " + (par1 + par2).ToString());
}
```

Wywołanie metody z argumentem w postaci delegatu:

```
MethodWithCallbackDelegate(2, 3, handler);
```

Uwagi i własności dotyczące delegatów:

1. delegat jest podobny do wskaźnika do funkcji w C/C++ ale oferuje bezpieczeństwo typu,
2. delegat może zostać przekazany jako parametr,
3. delegat jest stosowany do tworzenia wywołań zwrotnych (callback),
4. delegat może być łączony w łańcuch delegatow, jedno wywołanie może skutkować wywołaniem wielu metod,
5. delegat może wskazywać metody bez ścisłej zgodności z sygnatura (tzw. wariancja i kontrwariancja).



Notatki



## Kombinacja delegatow (multicase delegates)

Delegat choć jest typem pozwala na tworzenie kombinacji delegatów za pomocą operatorów sumy (i różnicy). Łączyć można „delegaty" tych samych typów.

delegate void DelegateTypeName(string s);

Prosty przykład tworzenia kombinacji delegatów:

```
DelegateTypeName a, b, c, d;

a = Mehtod1;
b = Method2;

c = a + b;
d = c - a;

a("A"); b("B");
c("C"); d("D");
```



Notatki



## Odczyt utworzonych wartości

Oddzielnym problemem jest odczyt wartości poszczególnych metod wywołanej kombinacji delegatów.

```
public static int Method1() {
    // zadania realizowane
    // w metodzie
    return 1;
}
```

Utworzenie delegatów przy wykorzystaniu wzorca delegatu, który nie przyjmuje argumentów ale zwraca liczbę całkowitą:

```
Func<int> DelInst1 = Class1.Method1;
Func<int> DelInst2 = Class1.Method2;
Func<int> DelInst3 = Class1.Method3;

Func<int> Instances = DelInst1 + DelInst2 + DelInst3;
```

Wywołanie i odczytanie wartości zwracanych przez poszczególne metody:



Notatki

## Metody anonimowe

W uproszczeniu jest to metoda zdefiniowana w miejscu użycia/przypisania bez określonej nazwy. Najczęściej stosowane są w kontekście delegatów i zdarzeń. Jednak w obecnym standardzie preferowane są lambda-wyrażenia, przy czym metody anonimowe mogą być stosowane bez listy argumentów.

```
delegate void Pokazywacz(string s);

static void Main() {
        Pokazywacz p = delegate(string j) {
            System.Console.WriteLine(j);
        };
        p("Delegat stosuje metodę anonimową");

        p = new Pokazywacz(Klasa.ZróbCośZTymStringiem);
        p("Ten delegat stosuje metodę ZróbCośZTymStringiem");
    }

    static void ZróbCośZTymStringiem(string k) {
        System.Console.WriteLine(k);
    }
```

Stosowanie metod anonimowych, pozwala na redukcję narzutu w kodzie, bowiem eliminuje konieczność tworzenia oddzielnych metod do realizacji niewielkich zadań.



Notatki



## Bezparametrowe metody anonimowe

Przykład przydatności metody anonimowej:

```
void StartThread()
{
    System.Threading.Thread t1 = new System.Threading.Thread
      (delegate()
            {
                System.Console.Write(Rozpoczęcie, ");
                System.Console.WriteLine("realizacji zadania!");
 StartTask();
            });
    t1.Start();
}
```

Powyższa metoda uruchamia nowy watek, gdzie początkowe instrukcję uruchamiające zadanie nie wymagają oddzielnej jawnie określonej metody.



Notatki



## Zdarzenia

Zdarzenie pozwalają na komunikację pomiędzy klasami lub obiektami, bowiem pewien obiekt/klasa może zgłosić „zdarzenie" które zostanie przekazane do innej klasy/obiektu. Klasa która wysyła (podnosi) zdarzenie jest nazywana publikatorem (publisher), a klasa która odbiera zdarzenie subskrybentem (subscriber).
Uwagi i własności dotyczące zdarzeń:

- Publikator określa, które zdarzenie będzie zgłoszone, natomiast subskrybent określa jaka akacja zostanie zrealizowana po otrzymaniu zdarzenia,
- zdarzenie może posiadać wielu subskrybentów, subskrybent może obsługiwać wiele zdarzeń z wielu publikatorów,
- zdarzenia, które nie posiadają subskrybentów nie są podnoszone,
- zazwyczaj zdarzenia są stosowane do sygnalizowania akcji użytkownika jak np.: kliknięcie na przycisk, wybranie opcji z menu,
- w przypadku, gdy zdarzenie posiada wielu subskrybentów, to poszczególne metody obsługujące zdarzenie są wywoływane synchroniczne, w momencie zgłoszenia zdarzenia (możliwa jest też obsługa asynchroniczna),
- zdarzenia mogą być stosowane do synchronizowania wątków,
- w .NET zdarzenie bazują na delegacie **EventHandler** oraz klasie bazowej **EventArgs**.



Notatki



## Podłączanie obsługi zdarzenia

Podłączenie obsługi przykładowego zdarzenia „Load":

```
this.Load += new System.EventHandler(this.Form1_Load);
```

Podobnie jak wyżej ale bez słowa **new**:

```
publisher.RaiseCustomEvent += HandleCustomEvent;
```

Wykorzystanie $\lambda$-wyrażenia:

```
this.Click += (s,e) => {
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
};
```

Usunięcie obsługi zdarzenia:



```
publisher.RaiseCustomEvent -= HandleCustomEvent;
```

Notatki

# Przykład zdarzenia – lista zgłaszająca zmiany

Utworzenie listy oraz dodatnie obsługi zdarzenia dodawania i usuwania elementów:

```
ListWithChangedEvent list = new ListWithChangedEvent();
list.Changed += new ChangedEventHandler(ListChanged);
```

Dodawanie elementów do listy:

```
list.Add("element 1"); list.Add("element 2"); list.Add("element 3");

list.Clear();
list.Changed -= new ChangedEventHandler(ListChanged);
```

Delegat do obsługi zdarzenia zmiany elementy w liście:

```
public delegate void ChangedEventHandler(object sender, EventArgs e);
```

Metoda do obsługi zdarzenia:



Notatki



Lista ze zdarzeniem OnChange wywoływanym w momencie zmiany listy (kiedy?):

```
public class ListWithChangedEvent: ArrayList {
    public event ChangedEventHandler Changed;

    protected virtual void OnChanged(EventArgs e) {
        if (Changed != null)
            Changed(this, e);
    }

    public override int Add(object value)  {
        int i = base.Add(value);
        OnChanged(EventArgs.Empty);
        return i;
    }

    public override void Clear() {
        base.Clear();
        OnChanged(EventArgs.Empty);
    }

    public override object this[int index] {
        set {
            base[index] = value;
            OnChanged(EventArgs.Empty);
        }
    }
}
```



Notatki



# Wyjątki

Wyjątki mają następujące własności:

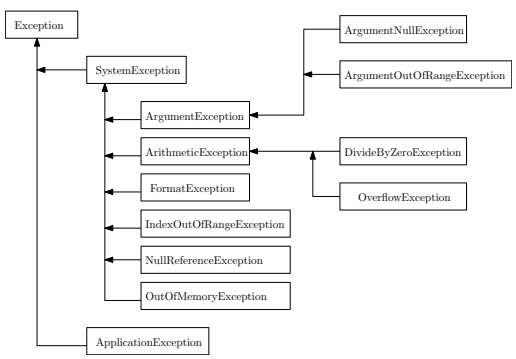
- wyjątki to typy, które dziedziczą z klasy System.Exception,
- wyjątki, które mogą być zgłaszane należy objąć sekcją try,
- wyjątki są zgłaszane przez stosowanie słowa kluczowego throw,
- w momencie zgłoszenia wyjątku, kontrola sterownia jest przekazywana do pierwszej obsługi wyjątku określonej słowem catch,
- jeśli nie ma obsługi zgłoszonego wyjątku, program zostaje zatrzymany i wyświetlany jest komunikat o błędzie,
- nie należy przechwytywać wyjątku, który nie jest możliwy do obsłużenia i wprowadzenie poprzez to aplikacji w stan nieokreślony. W przypadku przechwycenia System.Exception najlepiej zgłosić ten wyjątek ponownie na końcu bloku catch,
- warto w bloku wyjątku podawać nie tylko typ wyjątku ale także zmienną, gdyż klasa reprezentująca wyjątek może dostarczyć dodatkowych informacji o powodach wystąpienia sytuacji krytycznej,
- obiekty wyjątku zawierają szczegółowe informacje o błędzie, min. stan stosu wywołań oraz tekstowy opis błędu,
- kod w sekcji finally jest wykonywany nawet w przypadku zgłoszenia wyjątku, blok finally warto stosować do zwalniana użytych zasobów, np.: zamykania otwartych strumieni i plików,
- wyjątki zarządzane w platformie .NET są implementowane z użyciem struktury wyjątków Win32/Win64, jednak nie można ich traktować jako wyjątków systemowych.



Notatki



# Hierarchia wyjątków






Notatki

## Kiedy wyjątki powinny być zgłaszane

Zgłoszenie wyjątku powinno następować w następujących warunkach:

1. metoda nie może zrealizować postawionych zadań,
2. nastąpiło niewłaściwe wywołanie/odwołanie do obiektu,
3. w przypadku kiedy argument w metodzie nie jest poprawny.

Przykład zgłaszania wyjątku w przypadku przekroczeniu zakresu:

```
static int GetValueFromArray(int[] array, int index) {
    try {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex) {
        System.ArgumentException argEx = new System.ArgumentException(
                                    "Index is out of range", "index", ex);
        throw argEx;
    }
}
```



Notatki



## Przykład try ... catch ...

Konstrukcja try-catch zawiera blok try, w którym znajduje się kod, oraz jednej bądź więcej sekcji catch, w których znajdują się poszczególne sekcje obsługi różnych typów wyjątków.

```
try {
    string s = null;
    ProcessString(s);
}
catch (ArgumentNullException e) {
    Console.WriteLine("{0} Wyjątek numer jeden", e);
}
catch (Exception e) {
    Console.WriteLine("{0} Wyjątek numer dwa", e);
}
```



Notatki



## Przykład try ... finally ...

Blok **finally** jest zawsze wykonywany bez względu na to, czy zostały zgłoszone wyjątki.

```
int i = 123;
string s = "Jakiś ciąg znaków";
object o = s;

try {
    i = (int)o; // błędna konwersja
}
finally {
    Console.Write("i = {0}", i);
}
```



Notatki



## Przykład try ... catch ... finally ...

Zazwyczaj sekcje **catch** i **finally** stosuje się w kontekście wykrywania błędów w korzystaniu z poprawnie otworzonych zasobów, jednak po próbie skorzystaniu bez względu na pojawienie błędów oczekuje się iż dostęp do zasobu zostanie zamknięty:

```
string path = @"c:\users\public\test.txt";
System.IO.StreamReader file = new System.IO.StreamReader(path);
char[] buffer = new char[10];

try {
    file.ReadBlock(buffer, index, buffer.Length);
}
catch (System.IO.IOException e) {
    Console.WriteLine("Error reading from {0}. Message = {1}", path, e.Message);
}
finally {
    if (file != null) {
        file.Close();
    }
}
```



Notatki

## Definicja wyjątku użytkownika

Wyjątek jest reprezentowany przez nazwę klasy i musi dziedziczyć z klasy System.Exception:

```
[Serializable()]
public class ZłyIdentyfikatorWydziałuWyjątek : System.Exception {
    public ZłyIdentyfikatorWydziałuWyjątek() : base() { }
    public ZłyIdentyfikatorWydziałuWyjątek(string message) : base(message) { }
    public ZłyIdentyfikatorWydziałuWyjątek(string message,
                          System.Exception inner) : base(message, inner) { }

    protected ZłyIdentyfikatorWydziałuWyjątek(
           System.Runtime.Serialization.SerializationInfo info,
           System.Runtime.Serialization.StreamingContext context) { }
}
```



Notatki



## Sprawdzanie przekroczenia zakresu

Słowo kluczowe **checked** (jest to także operator) jest stosowane do wymuszenia testu przepełnienia zakresu w przypadku operacji arytmetycznych:

```
int dwanaście = 12;
int i2 = 2147483647 + dwanaście;

Console.WriteLine(checked(2147483647 + ten));

checked {
    int i3 = 2147483647 + ten;
    Console.WriteLine(i3);
}
```

Lepszym rozwiązaniem jest naturalnie przechwycenie wyjątku:

```
try {
    z = checked(maxIntValue + 12);
}
catch (System.OverflowException e) {
    Console.WriteLine("Przekroczenie zakresu:  " + e.ToString());
}
```



Notatki



## Czym jest wątek?

Środowisko wielowątkowe/wieloprocesorowe i tworzenie dla tego typu środowisk programów, to obecnie bardzo ważny obszar ze względu na rozwiązania wieloprocesorowe jakie stały się obecne niemal we wszystkich zastosowaniach począwszy od zagadnień naukowych, komercyjnych, a kończąc na rozwiązaniach domowych.

### Czym jest wątek

Podstawowym elementem w systemie operacyjnym jest proces, realizujący określone zadanie. W ramach procesu mniejszą jednostką jest wątek, którego przeznaczeniem jest realizacja określonego zadania w ramach procesu. Każdy wątek zawiera obsługę wyjątków, podlega systemowi priorytetów, posiada także mechanizmy do zatrzymania realizacji zadania bądź jego wznowieniu.

### Uwaga

Wątków w C# nie należy odnosić do systemu wątków danego systemu operacyjnego, w ramach którego działa maszyna CLR.



Notatki



## Kiedy warto stosować wątki

Stosowanie wątków jest zasadne w następujących przypadkach:

- komunikacja w sieci, np.: do serwera WEB bądź z bazą danych,
- wykonywanie operacji wymagających dużej ilości czasu,
- wątki pozwalają na zarządzanie zadaniami poprzez priorytety, ważne zadania mogą być realizowane przez wątki o wysokim priorytecie, a więc wykonywanym w pierwszej kolejności,
- wątki zwiększą także szybkość reakcji interfejsu użytkownika, poprzez realizacją zadań w tle wątku głównego.

### Przetwarzanie równoległe w .NET 4.0

Najnowsza wersja platformy .NET przyniosła duże zmiany w kontekście programowania wielowątkowego. Nowo wprowadzone klasy takie jak System.Threading.Tasks.Parallel oraz System.Threading.Tasks oraz nowy równoległy mechanizm zapytań (Parallel LINQ – PLINQ), ułatwiają tworzenie aplikacji wielowątkowych. Uzupełnieniem są także współbieżne kolekcje znajdujące się w klasie System.Collections.Concurrent.

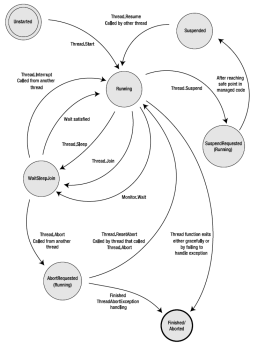


Notatki

## Stany wątków w .NET

Stany jakie wątek może przyjąć:

1. Unstarted
2. Running
3. SyspendedRequeste
4. Suspended
5. WaitSleepJoin
6. AbortRequested
7. Finised/Aborted






Notatki



## Podstawowa obsługa wątków w C#

Klasa która stanowi bazę to System.Threading.Thread. Jednak wcześniej trzeba zdefiniować funkcję, która będzie pracować w ramach wątku:

```
public static void ThreadRoutine() {
    for (int i = 0; i < 10; i++) {
        Console.WriteLine("ThreadRoutine: {0}", i);
        Thread.Sleep(0);
    }
}
```

Treść funkcji main:

```
Console.WriteLine("Wątek główny: uruchomienie wątku dodatkowego");
Thread t = new Thread(new ThreadStart(ThreadRoutine));

t.Start();

for (int i = 0; i < 4; i++) {
    Console.WriteLine("Wątek główny: ja coś robię");
    Thread.Sleep(0);
}

Console.WriteLine("Wątek główny: wywołanie Join(), oczekiwanie
                   na zakończenie pracy ThreadRoutine");
t.Join();
Console.WriteLine("Wątek główny:: ThreadRoutine -- zakończenie pracy");
```



Notatki



## Przekazanie danych do wątku

Przeciążanie metody Start, utworzenie obiektu wątku:

```
Thread newThread = new Thread(PracaDoWykonania);
newThread.Start( 1000 );
...
newThread.Start( "ciąg znaków" );
```

Metoda PracaDoWykonania jest następująca:

```
public static void PracaDoWykonania(object data) {
    // zrób coś z data;
}
```

Lepszy rozwiązaniem jest zastosowanie konstruktora i delegata **ThreadStart**:

```
public class BiggerThread {
    private string p1;
    private int p2;

    public BiggerThread(string t, int n) {
        _p1 = text;
        _p2 = number;
    }

    public void ThreadRoutine() {
        // zrób coś z _p1 i p_2
    }
}
...
BiggerThread bws = new BiggerThread( "tekst", 1000);
Thread t = new Thread(new ThreadStart(bws.ThreadRoutine));
t.Start();
```



Notatki



## Odczytanie danych z wątku

Odczytanie danych można zrealizować za pomocą metody zwrotnej utworzonej za pomocą delegata:

```
public delegate void ThreadCallback(int n);

public class ThreadWithState {
    private string _text;
    private int _value;

    private ThreadCallback callback;

    public ThreadWithState(string text, int number,  ThreadCallback callbackDelegate) {
        _text = text;
        _value = number;
        callback = callbackDelegate;
    }

    public void ThreadProc() {
        Console.WriteLine(_text, _value);
        if (callback != null)
            callback(10);
    }
}
```

Utworzenie wątku i definicja funkcji zwrotnej:

```
ThreadWithState tws = new ThreadWithState("ciag znaków", 100,
        new ThreadCallback(ResultCallback));

Thread t = new Thread(new ThreadStart(tws.ThreadProc));

...

public static void ResultCallback(int n) {
        Console.WriteLine( "Funkcja zwrotna wątku pomocniczego: liczba {0} ", n);
}
```



Notatki

## Programowanie asynchroniczne

Główny cel stosowania programowania asynchronicznego to uzyskanie wyższej wydajności, choć nie tylko, bowiem pozwala ono także na zwiększenie responsywności aplikacji, np. w obsłudze interfejsu użytkownika. W ramach API platformy .NET, następujące obszary wspierają przetwarzanie asynchroniczne:

- HttpClient, SyndicationClient – dostęp do usług WEB,
- StorageFile, StreamWriter, StreamReader, XmlReader – obsługa plików,
- MediaCapture, BitmapEncoder, BitmapDecoder – przetwarzanie obrazów (danych graficznych),
- Synchronous and Asynchronous Operations – obsługa technologii/frameworku WCF.

### Słowa kluczowe

Język C# oddaje do dyspozycji dwa słowa kluczowe **async** oraz **await** do realizacji programowania asynchronicznego.



Notatki



## Przykład 0 – Przygotowywanie śniadania

Synchroniczny algorytm przygotowania śniadania:

- przygotować filiżankę kawy,
- podgrzać patelnię, a następnie usmażyć dwa jajka,
- usmażyć trzy plasterki boczku.
- przygotować dwa kawałki chleba tostowego.
- nałożyć masło i dżem na tosty.
- nalać soku pomarańczowego do szklanki.

```
static void Śniadanie() {
    Kawa filiżanka = NalanieKawy();       Console.WriteLine("Kawa gotowa.");
    Jajka jajka = SmażenieJajek(2);       Console.WriteLine("Jajka gotowe.");
    Boczek plasterki = SmażenieBoczku(3); Console.WriteLine("Boczek gotowy.");
    Tost kromkaTostu = PieczenieTostu(2);
    UżyjMasła(kromaTostu);
    UżyjDżemu(toast);                     Console.WriteLine("Tost gotowy.");
    Sok sok = NalanieSoku();              Console.WriteLine("Sok gotowy");

    Console.WriteLine("Ufff, Śniadanie gotowe !!!");
}
```

Za dokumentacją ;-): `https://docs.microsoft.com/pl-pl/dotnet/csharp/programming-guide/concepts/async/`.



Notatki



Podstawowa wersja asynchroniczna:

```
static async Task Śniadanie()
{
    Kawa filiżanka = NalanieKawy();
    Console.WriteLine("Kawa gotowa.");

    Jajka jajka = await SmażenieJajek(2);
    Console.WriteLine("Jajka gotowe.");

    Boczek plasterki = await SmazenieBoczku(3);
    Console.WriteLine("Boczek gotowy.");

    Tost kromkaTostu = await PieczenieTostu(2);
    UżyjMasła(kromaTostu);
    UżyjDżemu(toast);
    Console.WriteLine("Tost gotowy.");

    Sok sok = NalanieSoku();
    Console.WriteLine("Sok gotowy");

    Console.WriteLine("Ufff, Śniadanie gotowe !!!");
}
```



Notatki



## Podstawowy przykład I

Problemy z kodem:

```
label1.Text = ""; label2.Text = ""; label3.Text = "";
...
DoWork1(); DoWork2(); DoWork3();
...
void DoWork1() {
    Thread.Sleep(2000); label1.Text = "robota 1 skonczona";
}

void DoWork2() {
    Thread.Sleep(2000); label2.Text = "robota 2 skonczona";
}

void DoWork3() {
    Thread.Sleep(2000); label3.Text = "robota 3 skonczona";
}
```

Poszczególne metody DoWork blokują aplikację na czas realizacji swojego zadania.



Notatki

## Podstawowy przykład I

Rozwiązanie problemu polega na zastosowaniu przetwarzania asynchronicznego:

```
private async void button1_Click(object sender, EventArgs e) {
    label1.Text = await DoWork1();
    label2.Text = await DoWork2();
    label3.Text = await DoWork3();
}
```

Metodę `DoWork<<X>>` należy zaimplementować w następujący sposób:

```
Task<string> DoWork<<X>>() {
    return Task.Run(() => {
        Thread.Sleep(2000);
        return "robota <<X>> skonczona";
    });
}
```

### Uwaga

Metoda wywołana ze słowem **await** musi zwracać typ `Task<T>`.



Notatki



## Zadanie z wynikiem void

Jeśli mamy następujące dwie linie kodu:

```
await MethodReturningEmptinessAsync();
MessageBox.Show("Done!");
```

To nie jest oczekiwana wartość powrotna, ponieważ metody asynchroniczna zwraca void. Wykorzystuje się w takim przypadku typ Task w wersji niegenerycznej:

```
private async Task MethodReturningEmptinessAsync() {
    await Task.Run(() => {
        Thread.Sleep(4000);
    });
}
```



Notatki



## Podstawowy przykład II

Prosty przykład pracy asynchronicznej z odczytem danych z podanego adresu WWW:

```
async Task<int> AccessTheWebAsync() {

    HttpClient client = new HttpClient();

    Task<string> getStringTask =
       client.GetStringAsync("http://www.address.com");

    DoSomeIndependentAndBoringWork();

    string urlContents = await getStringTask;

    return urlContents.Length;
}
```

Słowo **async** oznacza iż metoda AccessTheWebAsync będzie realizować swoje zadanie asynchronicznie.

Przykład pochodzący z dokumentacji MSDN:
`https://msdn.microsoft.com/pl-pl/library/hh191443.aspx`.



Notatki



## Podstawowy przykład

Następnie wywołujemy asynchroniczne zadanie odczytu danych z podanego adresu WWW:

```
HttpClient client = new HttpClient();
Task<string> getStringTask = client.GetStringAsync("http://www.address.com");
```

To zadanie jest wykonywane asynchronicznie niezależnie od sterownia w metodzie AccessTheWebAsync. Następnie po uruchomieniu zadania, następuje wywołanie metody: DoSomeIndependentAndBoringWork. Po zakończeniu pracy kolejna linia:

```
string urlContents = await getStringTask;
```

oczekuje za zakończenie realizacji zadania asynchronicznego. Oczekiwanie na zakończenie wymaga zastosowania słowa kluczowego **await**.

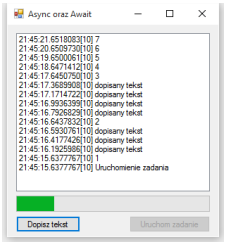
Przykład pochodzący z dokumentacji MSDN:
`https://msdn.microsoft.com/pl-pl/library/hh191443.aspx`.



Notatki

## Przykład z interfejsem GUI

Nieskomplikowana aplikacja, gdzie można dopisywać tekst do listy oraz równolegle i asynchronicznie wykonywane jest dodatkowe zadanie bez blokowania interfejsu użytkownika.






Notatki



Metoda WriteToList dopisuje komunikat tekstowy do listy. Główne zadanie wykonuje się po kliknięciu na przycisk Uruchom zadanie.

```
private async void TaskRun_Click(object sender, EventArgs e) {
    statusList.Items.Clear();
    ProgressBar.Maximum = TaskNUM;
    ProgressBar.Value = 0;
    TaskRunBTN.Enabled = false;
    WriteToList("Uruchomienie zadania");
    for (int i = 1; i <= TaskNUM; i++) {
        WriteToList(i.ToString());
        await Task.Run(() => {
            // tu wykonujemy jakąś operację
            // której wykonanie może zabrać
            // "nadnormatywnie" więcej czasu
            Thread.Sleep(1000);
        });
        ProgressBar.Value = i;
    }
    WriteToList("Zakończenie zadania");
    TaskRunBTN.Enabled = true;
}
```



Notatki



## Przygotowania śniadania – pełna wersja asynchroniczna – 1/2

Zakładając iż tworzymy zdania: `Task<Jajka> jajkaZadanie = SmażenieJajekAsync(2);`, to nowa wersja tworzenia śniadania przedstawia się następująco:

```
static async Task Śniadanie() {
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    var eggsTask = SmażenieJajekAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var allTasks = new List<Task>{eggsTask, baconTask, toastTask};
    while (allTasks.Any())
    {
        Task finished = await Task.WhenAny(allTasks);
        if (finished == eggsTask) {
            Console.WriteLine("eggs are ready");
        }
        else if (finished == baconTask) {
            Console.WriteLine("bacon is ready");
        }
```



Notatki



## Przygotowania śniadania – pełna wersja asynchroniczna – 2/2

```
        else if (finished == toastTask) {
            Console.WriteLine("toast is ready");
        }
        allTasks.Remove(finished);
    }
    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");

    async Task<Toast> MakeToastWithButterAndJamAsync(int number) {
        var toast = await ToastBreadAsync(number);
        ApplyButter(toast);
        ApplyJam(toast);
        return toast;
    }
}
```



Notatki

# Wybrane nowe rozwiązania w języku C# (wersje 6.0, 7.x, 8.x, 9.x, 10.x i 11.x)



Notatki



## Nowe konstrukcje językowe

Nowe konstrukcje językowe wprowadzone w wersji 6.0 języka C# (pozostawiono angielskie nazewnictwo):

- Auto-Property enhancements, Expression-bodied function members, using static,
- Null-conditional operators, String Interpolation, Exception Filters,
- nameof Expressions, Await in Catch and Finally blocks, Index Initializers,
- Improved overload resolution.

Nowe konstrukcje językowe wprowadzone w wersji 7.0 języka C#:

- Out variables, Pattern matching, Tuples,
- Deconstruction, Local functions,
- Literal improvements, Ref returns and locals,
- Generalized async return types, More expression bodied members, Throw expressions.



Notatki



## Syntaktyka dla ciągów znaków

Zawzyczaj operacje formatowania na ciągach znaków prezentują się następująco:

```
string sout = string.Format("{0} {1}", FirstName, LastName);
```

To dzięki notacji z symbolem$ to samo wyrażenie można zapisać nieco krócej:

```
string sout = $"{FirstName} {LastName}";
```

Możliwe jest stosowanie wywołań innych obiektów i metod:

```
string sout =  $"Name: {LastName}, {FirstName}.
     G.P.A: {Grades.Average()}";
```

A nawet bardziej skomplikowane wyrażenia:

```
string sout = $"Name: {LastName}, {FirstName}.
  G.P.A: {Grades.Any() ? Grades.Average() : double.NaN:F2}";
```



Notatki



## Nowe rozwiązania w C# – zmienne typu out

Typowy kod dla zmiennych wyjściowych:

```
Point pt;
int x, y; // zmienne muszą być
          // wcześniej zadeklarowene
pt.GetCoords(out x, out y);
WriteLine($"({x}, {y})");
```

Nowe rozwiązanie syntaktyczne pozwala na utworzenie nowych zmiennych w momencie wywołania funkcji:

```
pt.GetCoords(out int x, out int y);
WriteLine($"({x}, {y})");
```

Można także zastosować typ var:

```
pt.GetCoords(out var x, out var y);
```



Notatki

Co pozwala na łatwe tworzenie tego typu konstrukcji:

```
if (int.TryParse(s, out var i))  {
    WriteLine(new string('*', i));
}
else {
    WriteLine("To jednak nie była liczba int!");
}
```

Można także pominać zmienną wyjściową za pomocą znaku podkreślenia:

```
p.GetCoords(out var x, out _, out var z);
```



Notatki



## Dopasowanie wzorca (ang. pattern matching)

Język C# w wersji 7.0 wprowadza pojęcie dopasowania wzorca. Jest to dodatek syntaktyczny pozwalający na przeprowadzenie testu na wartości, czy spełnia odpowiednie warunki. Gdy określone warunki zostaną spełnione, następuje uzyskanie wartości z podanej zmiennej. Można wskazać trzy główne miejsca, gdzie wzorce można zastosować:

- wzorce stałe, pozwala to na sprawdzenie, czy wejście jest równe stałej,
- wzorce typu, pozwala na to na sprawdzenie, czy wejście jest określonego typu jeśli tak, to otrzymamy wartości zgodną z określonym typem,
- wzorce zmiennych (ang. var patterns), które zawsze są zgodne i pozwalają na otrzymanie wartości z wejścia.

Do obsługi wzorów C# dodaje słowo kluczowe is oraz pozwala w konstrukcji case stosować konstrukcje odnoszące się do wzorca, a nie tylko do stałych jak dotychczas.



Notatki



Przykład, jeśli dwa pierwsze testy nie zawiodą, to parametr o zawiera liczbę całkowitą:

```
public void fooMethod(object o)
{
    if (o is null) return;     // constant pattern for "null"
    if (!(o is int i)) return; // type pattern for "int i"

    WriteLine(new string('*', i));
}
```

Naturalnie wzorce są elastyczne, podobne zachowanie można zapisać w taki oto sposób:

```
if (o is int i || (o is string s && int.TryParse(s, out i))
{ /* a teraz można używać i ile się chce */ }
```



Notatki



Konstrukcja switch została rozszerzona o obsługę wzorców:

- możliwe jest stosowanie dowolnego typu, a nie jak dotąd tylko dla typów prymitywnych,
- wzorce mogą być użyte jak dodatkowe klauzule w przypadkach,
- klauzule przypadków mogą opisywać dodatkowe warunki.

```
switch(shape) {
    case Circle c:
        WriteLine($"circle with radius {c.Radius}");
    break;
    case Rectangle s when (s.Length == s.Height):
        WriteLine($"{s.Length} x {s.Height} square");
    break;
    case Rectangle r:
        WriteLine($"{r.Length} x {r.Height} rectangle");
    break;
    default:
        WriteLine("<kształt nieznany>");
    break;
    case null:
        throw new ArgumentNullException(nameof(shape));
}
```



Notatki

## Krotki

C# 7.0 wprowadza istotne poprawki w korzystaniu z krotek:

```
var letters = ("a", "b");
```

Określanie nazw pól dostępowych:

```
(string Alpha, string Beta) namedLetters = ("a", "b");
var alphabetStart = (Alpha: "a", Beta: "b");
```

Metoda zwracająca krotkę:

```
private static (int Max, int Min) Range(IEnumerable<int>
                numbers) {
    ...
    return (max, min);
}
```



Notatki



Krotka powstała z klasy za pomocą metody Deconstruct:

```
public class flPoint {
    public flPoint(float x, float y) {
        this.X = x; this.Y = y;
    }

    public float X { get; }
    public float Y { get; }

    public void Deconstruct(out float x, out float y) {
        x = this.X; y = this.Y;
    }
}
```

Co pozwala na podanie następującego przykładu:

```
var p = new flPoint(3.14, 2.71);
(float X, float Y) = p;
```



Notatki



## Zmiany w C# dla wersji 7.1

Nowe elementy dodane w wersji 7.1 języka:

- metoda Main może być asynchroniczna:

```
static async Task<int> Main() {  |  static async Task Main() {
    return await DoAsyncWork();  |      await SomeAsyncMethod();
}                                |  }
```

- wartości domyślne w wyrażeniach: Func¡string, bool¿ whereClause = default; (wcześniej należało powtórzyć postać typu w default),
- wnioskowania nazw w krotkach, na podstawie użytych zmiennych:

```
int count = 5;
string label = "Colors used in the map";
var pair = (count, label);
```



Notatki



## Nowy rodzaj dziedziczenia

C# 7.2 wprowadza dodatkowy rodzaj dziedziczenia: private protected.
Zatem dostępne są następujące rodzaje dziedziczenia:

- public – dostęp nie jest ograniczony,
- protected – dostep jest ograniczony tylko do klas lub klasy dziedziczącej,
- internal – dostęp jest ograniczony tylko do podzespołu,
- protected internal – dostęp jest ograniczony tylko do podzespołu bądź typów wyprowadzonych z danej klasy,
- private – ograniczenie dostęp tylko do danego typu,
- private protected – ograniczenie dostęp tylko do danego typu lub typów powstałych/deklarowanych przez dziedziczenie w ramach jednego podzespołu.



Notatki

Nowości w 7.3 dotyczące bezpiecznego i wydajnego kodu:

- it is possible to access fixed fields without pinning,
- it is possible to reassign ref local variables,
- it is possible to use initializers on stackalloc arrays,
- it is possible to use fixed statements with any type that supports a pattern,
- it is possible to use additional generic constraints.

Poprawki w istniejących rozwiązaniach:

- it is possible to test == and != with tuple types,
- it is possible to use expression variables in more locations,
- it is possible to attach attributes to the backing field of auto-implemented properties,
- method resolution when arguments differ by in has been improved,
- overload resolution now has fewer ambiguous cases.



Notatki



## Elementy wprowadzone w C# v8.x

W wersji 8.x najnowsze zmiany dotyczą m. in. następujących elementów:

- Pattern matching enhancements (switch expressions, property patterns, tuple patterns, positional patterns),
- Using declarations,
- Static local functions,
- Disposable ref structs.

A także:

- Nullable reference types,
- Asynchronous streams,
- Indices and ranges.



Notatki



Kolejne ułatwienia w konstrukcji switch:

```
public static RGBColor FromRainbow(Rainbow colorBand) =>
    colorBand switch
    {
        Rainbow.Red    => new RGBColor(0xFF, 0x00, 0x00),
        Rainbow.Orange => new RGBColor(0xFF, 0x7F, 0x00),
        Rainbow.Yellow => new RGBColor(0xFF, 0xFF, 0x00),
        Rainbow.Blue   => new RGBColor(0x00, 0x00, 0xFF),
        Rainbow.Indigo => new RGBColor(0x4B, 0x00, 0x82),
        Rainbow.Violet => new RGBColor(0x94, 0x00, 0xD3),
        _              => throw new ArgumentException(message:
            "invalid enum value", paramName: nameof(colorBand)),
    };
```



Notatki



Wykorzystanie wzorców w określeniu wartości danego pola:

```
public static decimal ComputeSalesTax(Address location,
                                      decimal salePrice) =>
    location switch
    {
        { State: "WA" } => salePrice * 0.06M,
        { State: "MN" } => salePrice * 0.75M,
        { State: "MI" } => salePrice * 0.05M,
        // other cases removed for brevity...
        _ => 0M
    };
```



Notatki

Switch oraz wzorce dopasowania do krotek:

```
public static string RockPaperScissors(string first, string second)
    => (first, second) switch
    {
        ("rock", "paper") => "rock is covered by paper. Paper wins.",
        ("rock", "scissors") => "rock breaks scissors. Rock wins.",
        ("paper", "rock") => "paper covers rock. Paper wins.",
        ("paper", "scissors") => "paper is cut by scissors. Scissors wins.",
        ("scissors", "rock") => "scissors is broken by rock. Rock wins.",
        ("scissors", "paper") => "scissors cuts paper. Scissors wins.",
        (_, _) => "tie"
    };
```



Notatki



Wzorce z pozycjonowaniem, tj. sprawdzanie przynależności punktu do danej ćwiartki układu współrzędnych. Definicja klasy Point:

```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);

    public void Deconstruct(out int x, out int y) =>
        (x, y) = (X, Y);
}
```



Notatki



Wzorce z pozycjonowaniem, tj. sprawdzanie przynależności punktu do danej ćwiartki układu współrzędnych:

```
static string Quadrant(Point p) => p switch
{
    (0, 0) => "origin",
    (var x, var y) when x > 0 && y > 0 => "Quadrant 1",
    (var x, var y) when x < 0 && y > 0 => "Quadrant 2",
    (var x, var y) when x < 0 && y < 0 => "Quadrant 3",
    (var x, var y) when x > 0 && y < 0 => "Quadrant 4",
    (var x, var y) => "on a border",
    _ => "unknown"
};
```



Notatki



Ułatwienia w stosowaniu konstrukcji using:

```
using var file = new System.IO.StreamWriter("output-log.txt");
foreach (string line in lines)
{
    ...
file.WriteLine(line);
...
}
// the file variable is disposed here
```



Notatki

## Funkcje lokalne

C# w nowych odsłonach pozwala na tworzenie funkcji lokalnych:

```
int M() {
    int y;
    LocalFunc();
    return y;

    void LocalFunc() => y = 10;
}
```

Funkcje mogą być statyczne podwarunkiem iż nieużywają zmiennych funkcji w której są osadzone:

```
int M() {
    int x = 7;
    int y = 3;
    return Suma(x, y);

    static int Suma(int l, int r) => l + r;
}
```



Notatki



Spis nowości w C# v9.x

- Records
- Init only setters, Top-level statements
- Pattern matching enhancements
- Performance and interop
  - Native sized integers
  - Function pointers
  - Suppress emitting localsinit flag
- Fit and finish features
  - Target-typed new expressions
  - static anonymous functions
  - Target-typed conditional expressions
  - Covariant return types
  - Extension GetEnumerator support for foreach loops
  - Lambda discard parameters
  - Attributes on local functions
- Support for code generators
  - Module initializers
  - New features for partial methods



Notatki



Typ rekord jest podobny do struktury, i jego przeznaczeniem jest opis danych:

```
public record Person(string FirstName, string LastName);
```

Rekord z polami do odczyty:

```
public record Person
{
    public string FirstName { get; init; } = default!;
    public string LastName { get; init; } = default!;
};
```

Rekord z polami do odczytu/zapisu:

```
public record Person
{
    public string FirstName { get; set; } = default!;
    public string LastName { get; set; } = default!;
};
```



Notatki



Rekordy z definicji nie powinny posiadać cechy mutacyjności, gdyż ich przeznaczeniem jest wspieranie modeli danych przeznaczonych tylko do odczytu. Rekordy cechują się też:

- spójna syntaktyka dla typu referencyjnego z danymi do odczytu (ang. immutable properties)
- zachowanie się rekordów zostało zaprojektowane dla typów odnoszących się do danych:
  - równość wartości,
  - syntaktyka dla zmian nieniszczących (ang. nondestructive mutation)
  - wspieracie dla formatowanie danych.
- Wsparcie dla typów danych opartych o dziedziczenie

```
public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}
```



Notatki

Rekord z danymi tylko do odczytu:

```
public struct WeatherObservation {
    public DateTime RecordedAt { get; init; }
    public decimal TemperatureInCelsius { get; init; }
    public decimal PressureInMillibars { get; init; }

    public override string ToString() =>
        $"At {RecordedAt:h:mm tt} on {RecordedAt:M/d/yyyy}: " +
        $"Temp = {TemperatureInCelsius},
                    with {PressureInMillibars} pressure";
}
```

Przypisanie danych jest możliwe tylko podczas inicjalizacji:

```
var now = new WeatherObservation {
    RecordedAt = DateTime.Now,
    TemperatureInCelsius = 20,
    PressureInMillibars = 998.0m
};
```



Notatki



Konstrukcje startowe bez konieczności stosowania klasy i funkcji main:

```
using System;

Console.WriteLine("Hello World !!!");
```

Zamiast dłuższego przykładu:

```
using System;

namespace HelloWorld {
    class Program  {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World !!!");
        }
    }
}
```



Notatki



Spis nowości w C# v10.x

- Record structs
- Improvements of structure types
- Interpolated string handlers
- global using directives
- File-scoped namespace declaration
- Extended property patterns
- Improvements on lambda expressions
- Allow const interpolated strings
- Record types can seal ToString()
- Improved definite assignment
- Allow both assignment and declaration in the same deconstruction
- Allow AsyncMethodBuilder attribute on methods
- CallerArgumentExpression attribute
- Enhanced #line pragma



Notatki



Uproszczenia w obsłudze dekonstrukcji obiektu. Dotychczas należało odpowiednio deklarować wszystkie zmienne:

```
// Initialization:
(int x, int y) = point;

// assignment:
int x1 = 0;
int y1 = 0;
(x1, y1) = point;
```

Od wersji 10 język C# dopuszcza zapis, gdzie tylko jedna zmienna deklarowana jest w sekcji dekonstrukcji:

```
int x = 0;
(x, int y) = point;
```



Notatki

Wybrane nowości w C# v11:

- Raw string literals, Generic math support, Generic attributes
- UTF-8 string literals, Newlines in string interpolation expressions
- List patterns, File-local types, Required members
- Auto-default structs, Pattern match Span¡char¿ on a constant string
- Extended nameof scope, Numeric IntPtr
- ref fields and scoped ref
- Improved method group conversion to delegate
- Warning wave 7

Nowa wersja C# 11 wymaga .NET 6/7.



Notatki



C# 11 upraszcza używanie uogólnionych atrybutów, zamiast:

```
public class TypeAttribute : Attribute
{
   public TypeAttribute(Type t) => ParamType = t;

   public Type ParamType { get; }
}
...
[TypeAttribute(typeof(string))]
public string Method() => default;
```

Można teraz napisać krócej bo:

```
public class GenericAttribute<T> : Attribute { }
...
[GenericAttribute<string>()]
public string Method() => default;
```



Notatki



## Raw string literal – surowe literały znakowe

C# 11 pozwala na stosowanie literałów znakowych, gdzie zachowane zostaną odstępy, przejścia do nowej linii oraz stosowanie pojedynczych cudzysłowów:

```
string longMessage = """
    Przykład dłużego tekstu.
    Zbudowanego z kilku linii, ze wcięciami.
        takimi tak to,
            oraz nieco większymi.
    A tu dla przykładu bez wcięcia.
    Można także stosowanie "cytowanie" tekstu w tekście ;-).
""";
```

Można stosować także nazwy zmiennych:

```
var location = $$"""
Koordynaty GPS: {{{Longitude}}, {{Latitude}}}
""";
```



Notatki



Wersja C# 12 aktualnie jest rozpatrywana jako eksperymentalna, i wymaga .NET SDK 8 Preview:

- Primary constructors
- Optional parameters in lambda expressions
- Alias any type

Krótki opis dwóch nowych rozwiązań:

- Rozszerzono stosowanie podstawowych konstruktorów nie tylko dla rekordów, ale możliwe jest ich stosowanie w klasach oraz strukturach.
- Można użyć dyrektywy using, aby wprowadzić alias dla dowolnego typu (dotychczas możliwe było to tylko dla nazwanych typów). Oznacza to, że można tworzyć aliasy semantyczne dla krotek, tablic, wskaźników, a także dla tzw. typów unsafe.



Notatki

Dodatkowe informacje o nowych elementach języka C# można odszukać pod adresami:

- dla wersji 6.0 – `https://docs.microsoft.com/pl-pl/dotnet/articles/csharp/whats-new/csharp-6#read-only-auto-properties`,
- dla wersji 7.0 – `https://blogs.msdn.microsoft.com/dotnet/2017/03/09/new-features-in-c-7-0/`,
- dla wersji 7.x – `https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-7?view=netframework-4.7.1`,
- i ogólnie dla 8.x – `https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-8?view=netframework-4.7.1`,
- dla wersji 9.x – `https://docs.microsoft.com/en-gb/dotnet/csharp/whats-new/csharp-9`,
- dla wersji 10.x – `https://docs.microsoft.com/en-gb/dotnet/csharp/whats-new/csharp-10`,
- dla wersji 11.x – `https://docs.microsoft.com/en-gb/dotnet/csharp/whats-new/csharp-11`,
- dla wersji 12.x – `https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-12`.



Notatki



## Windows Forms

Wersja platformy .NET 2.0 oferowała następujące rozwiązania w dziedzinie obsługi grafiki:

| Funkcjonalność | Stosowane API |
|---|---|
| Okno i kontrolki | Windows Forms |
| grafika 2D | GDI+ (System.Drawing.dll) |
| grafika 2D | DirectX |
| strumieniowe video | Windows Media Player API |
| wsparcie dokumentów | np.: dokumenty PDF |

Od wersji .NET 3.0 wprowadzono nową technologię WPF (Windows Presentation Foundation) która w całości zastępuje starsze rozwiązania. Jest to technologia przeznaczono nie tylko dla typowych aplikacji okienkowych ale także aplikacji WEB. Jednym z ważniejszych elementów jest XAML – Extensible Application Markup Language, język opisu interfejsu użytkownika. Jednakże, w najnowszej wersji 4.0 API Windows Forms nadal jest obecne i zostało uaktualnione o nowe funkcjonalności jak wykresy czy interfejs 3D.

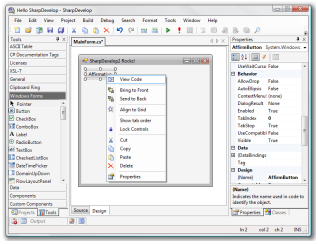


Notatki



## Przeznaczenie Windows Forms

Pierwsze wersja platformy .NET (rok 2001) oferowały zestaw klas Windows.Forms do tworzenia aplikacji z graficznym interfejsem użytkownika. Najważniejszym elementem było ukrycie detali związanych z Windows API i dostarczenie wielu podstawowych komponentów do tworzenia aplikacji typu GUI dla użytkownika. Oprócz tego Windows.Forms oferują narzędzia do zarządzania zasobami jak ikony czy tabele znaków. Środowiska dla programistów jak np.: Visual Studio czy SharpDevelop oferują także możliwość graficznego projektowania formatek dla aplikacji „okienkowych".





Notatki



## Przesteń nazw Windows Forms

Najważniejsza przestrzeń nazw jest System.Windows.Forms która zawiera wszystkie podstawowe klasy do tworzenia tzw. aplikacji okienkowych – aplikacji z interfejsem użytkownika. Przestrzeń ta jest podzielona na następujące obszary:

- „Core infrastructure": podstawowe typy reprezentujące najważniejsze operacje programów Windows Forms, i zawiera także różne dodatkowe klasy do współpracy z kontrolkami ActiveX oraz nowymi kontrolkami WPF,
- „Menus and Toolbars": aplikacje Windows Forms zawierają bogaty zbiór klas do tworzenia własnych menu oraz pasków z przyciskami, z nowoczesnym wyglądem oraz zachowaniem (ToolStrip, MenuStrip, ContextMenuStrip, StatusStrip),
- „Controls": zawiera typy do tworzenia graficznych interfejsów użytkownika jak np.: przyciski, menu, tabele z danymi, możliwa jest też dynamiczne konfiguracja w trakcie działania aplikacji.



Notatki

# Przestrzeń nazw Windows Forms

- „Layout": tzw. zarządca położenia, pomaga kontrolować położenie kontrolek w oknie lub innych kontrolkach, najważniejsze klasy które pomagają w realizacji tego typu zadań to np.: FlowLayoutPanel czy TableLayoutPanel gdzie można określić iż poszczególne kontrolki zostaną ułożone w sposób tabelaryczny, przydatny jest też SplitContainer, gdyż pozwala na podzielenie przestrzeni na dwie bądź więcej części,
- „Components": typy tego rodzaju nie dziedziczą z klasy Control, jednak dostarczają dodatkowe funkcjonalności jak np.: treść podpowiedzi, czy klasa Timer pozwalająca na wywoływanie zdarzeń w równych odstępach czasu,
- „Common dialog boxes": zarządzanie oknami dialogowymi w kontekście podstawowych operacji, podstawowe typu okien to np.: OpenFileDialog, PrintDialog, and ColorDialog, naturalnie możliwe jest tworzenie własnych okien dialogowych.

## Model programowania

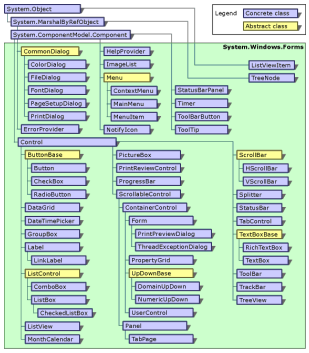
Aplikacje Windows.Forms są oparte o tzw. model zdarzeniowy (event-driven model), inaczej mówiąc poszczególne elementy programu reagują na pojawiające się zdarzenia jak np.: kliknięcie na przycisk, czy wybór elementu z menu.



Notatki



# Hierarchia klas Windows Forms






Notatki



# Proste okno

Utworzenie okna aplikacji przy wykorzystaniu klasy **Form**:

```
using System;
using System.Windows.Forms;

namespace SimpleWinFormsApp {
    class Program  {
        static void Main(string[] args) {
            Application.Run(new MainWindow());
        }
    }

    class MainWindow : Form {
    }
}
```



Notatki



# Parametry okna

Ustalenie podstawowych elementów okna:

```
class MainWindow : Form {
    public MainWindow() {}
    public MainWindow(string title, int h, int w) {
        Text = title;
        Width = w;
        Height = h;
        CenterToScreen();
    }
}
```



Notatki

## Dodanie menu

Obiekty reprezentujące menu oraz opcje:

```
private MenuStrip mnuMainMenu = new MenuStrip();
private ToolStripMenuItem mnuFile = new ToolStripMenuItem();
private ToolStripMenuItem mnuFileExit = new ToolStripMenuItem();
```

Prywatna metoda tworząca menu:

```
private void BuildMenuSystem() {
    mnuFile.Text = "&File";
    mnuMainMenu.Items.Add(mnuFile);

    mnuFileExit.Text = "E&xit";
    mnuFile.DropDownItems.Add(mnuFileExit);

    mnuFileExit.Click += (o, s) => Application.Exit();

    Controls.Add(this.mnuMainMenu);
    MainMenuStrip = this.mnuMainMenu;
}
```



Notatki



## Kontrolka ListBox

Utworzenie prostej kontrolki ListBox:

```
    private StatusBar sb;

    public MainWindow() {
        Text = "ListBox";
        Size = new Size(210, 210);

        ListBox lb = new ListBox();
        lb.Parent = this;
        lb.Items.Add("Maria");lb.Items.Add("Ryszark");
        lb.Items.Add("Angelica"); lb.Items.Add("Maciej");
        lb.Items.Add("Dorota"); lb.Items.Add("Katarzyna");

        lb.Dock = DockStyle.Fill;
        lb.SelectedIndexChanged += new EventHandler(OnChanged);

        sb = new StatusBar();
        sb.Parent = this;

        CenterToScreen();
    }

    void OnChanged(object sender, EventArgs e) {
        ListBox lb = (ListBox) sender;
        sb.Text = lb.SelectedItem.ToString();
    }
```



Notatki



## Kontrolka ListView – 1/4

Struktura pliku **Form1.cs**, najważniejsze elementy to metoda **InitializeListView** oraz konstruktor **Form1**:

```
using System;
using System.Windows.Forms;

namespace ListViewTest {

   public class Form1 : Form {
         private System.Windows.Forms.ListView myListView;

         private void InitializeListView() { ... }

         public Form1 () {
            this.Left = 100;
            this.Top = 100;
            this.Height = 280;
            this.Width = 400;
            InitializeListView();
         }
   }
}
```



Notatki



## Kontrolka ListView – 2/4

Zawartość metody **InitializeListView**:
Utworzenie obiektu kontrolki, ustalenie współrzędnych oraz wymiarów:

```
    myListView = new ListView();
    myListView.Location = new System.Drawing.Point(20, 20);
    myListView.Height = 200;
    myListView.Width = 280;
```

Ustalenie sposobu prezentacji danych:

```
myListView.View = View.Details;
```

Trzy główne kolumny w kontrolce:

```
myListView.Columns.Add("Key",    50, HorizontalAlignment.Left);
myListView.Columns.Add("Col A", 100, HorizontalAlignment.Left);
myListView.Columns.Add("Col B", 100, HorizontalAlignment.Left);
```



Notatki

## Kontrolka ListView – 3/4

Dodanie obiektu wiersza z danymi (dane Expense oraz Revenue) oraz ustalenie własności graficznych:

```
ListViewItem entryListItem = myListView.Items.Add("I1");
entryListItem.UseItemStyleForSubItems = false;

ListViewItem.ListViewSubItem expenseItem = entryListItem.SubItems.Add("Expense");
expenseItem.ForeColor = System.Drawing.Color.Red;
expenseItem.Font = new System.Drawing.Font(
                                "Arial", 10, System.Drawing.FontStyle.Italic );

ListViewItem.ListViewSubItem revenueItem = entryListItem.SubItems.Add("Revenue");
revenueItem.ForeColor = System.Drawing.Color.Blue;
revenueItem.Font = new System.Drawing.Font(
                            "Times New Roman", 10, System.Drawing.FontStyle.Bold );
```



Notatki



## Kontrolka ListView – 4/4

Dodanie danych, czyli drugi oraz trzeci wiersz danych

```
ListViewItem entryListItem2 = myListView.Items.Add("I2");
ListViewItem.ListViewSubItem expenseItem2 =
                                entryListItem2.SubItems.Add("Expense 2");
ListViewItem.ListViewSubItem revenueItem2 =
                                entryListItem2.SubItems.Add("Revenue 2");

ListViewItem entryListItem3 = myListView.Items.Add("I3");
ListViewItem.ListViewSubItem expenseItem3 =
                                entryListItem3.SubItems.Add("Expense 3");
ListViewItem.ListViewSubItem revenueItem3 =
                                entryListItem3.SubItems.Add("Revenue 3");
```

Ostatnia czynność, dodanie kontroli do hierarchii kontrolek formularza

```
Controls.Add( this.myListView );
```



Notatki



## Grafika 2D – GDI+

Interfejs GDI+ to podsystem systemu operacyjnego Windows XP (jest obecny także w późniejszych systemach) i jest odpowiedzialny za metody tworzenia grafikli na ekranach oraz urządzeniach drukujących.

| Przestrzeń nazw | Przeznaczenie |
|---|---|
| System.Drawing | Podstawowa przestrzeń nazw GDI+, zawiera definicje typów dla podstawowych metod rysujących jak czcionki, kolory, piksaki, i etc. |
| System.Drawing.Drawing2D | W tej przestrzeni umieszczono klasy i metody do bardziej zaawansowanej grafiki 2D jak np. wypełnienia, przekształcenia geometryczne, oferowane jest także wsparcie dla grafiki wektorowej. |
| System.Drawing.Imaging | Przestrzeń zawiera typy pozwalające na manipulację plikami graficznymi jak np.: zmiana palety, odczyt metadanych i etc. |
| System.Drawing.Printing | Obsługa urządzeń drukujących |
| System.Drawing.Text | Typ i definicje do manipulacji czcionkami |



Notatki



## Prosty rysunek

W konstruktorze klasy **MainWindow** należy dodać obsługę zdarzenia **Paint**:

```
Paint += new PaintEventHandler(OnPaint);
```

Obsługa zdarzenia jest następująca:

```
private void OnPaint(object sender, PaintEventArgs e) {
    Graphics g = e.Graphics;
    g.FillEllipse(Brushes.Blue, 10, 20, 150, 80);
    g.DrawString("Hello GDI+", new Font("Times New Roman", 30),
        Brushes.Red, 200, 200);
    using (Pen p = new Pen(Color.YellowGreen, 10)) {
        g.DrawLine(p, 80, 4, 200, 200);
    }
}
```



Notatki

## A w następnym tygodniu między innymi:

1. języki funkcyjne na przykładzie F#,
2. podstawowe typy F#,
3. wartości i funkcje,
4. analiza leksykalna wyrażeń tekstowych,
5. „leniwe" obliczenia.

Proponowane tematy prac pisemnych:

1. model programowania zorientowanego na zdarzenia,
2. delegaci, metody anonimowe, lambda wyrażenia, dlaczego w najnowszym standardzie zaleca się stosowanie $\lambda$-wyrażeń,
3. analiza zawartości przestrzeni nazw System.Windows.Forms oraz System.Drawing w środowisku Mono oraz sprawdzenie kompatybilności z implementacją .NET firmy Microsoft.

# Dziękuje za uwagę!!!



Notatki

Notatki

Notatki

Notatki