Politechnika Łódzka
Wydział Elektrotechniki, Elektroniki, Informatyki i Automatyki

Interfejsy platform mobilnych

Aspekty programowania obiektowego w języku C#

Instytut Informatyki Stosowanej
Politechniki Łódzkiej

Opracował: dr inż. Radosław Wajman

Rodzaje typów

- Bezpośrednie : (value type) są strukturami przechowywanymi (na stosie) jako wartości.
  np. int, bool, float, char ... oraz struktury.

- Referencyjne : (reference type) ich wartości są przechowywane na stercie (w pamięci rezerwowanej dynamicznie), na stosie znajdują się jedynie referencje (wskazania) do obiektów.
  wszystkie klasy (np. object, string), ale także tablice czy interfejsy.

- Wskaźnikowe : używane do jawnego manipulowania pamięcią.
  Zwykle się ich nie używa (ponadto wskaźniki mogą być używane tylko w blokach nienadzorowanych).

Rodzaje typów

- Zmienna typu referencyjnego może mieć wartość **null** oznaczającą, że nie wskazuje żadnego obiektu.
- Obiekt typu referencyjnego pozostaje na stercie tak długo, aż system nie stwierdzi, że nie ma już żadnych odwołań do tego obiektu (automatyczne zwalnianiem pamięci).
- Przypisanie (operator =) typu referencyjnego polega na skopiowaniu wskazania na obiekt (nie powstaje nowy egzemplarz obiektu, a tylko mamy kolejne wskazanie na ten sam obiekt).
- Przypisanie (i ogólnie odwołanie się) typu bezpośredniego polega na skopiowaniu wartości całego obiektu.

Rodzaje dostępu

- W C# zdefiniowano pięć rodzajów dostępu do typów i ich składowych:
  - **public** : składowa lub typ zadeklarowany jako publiczny są dostępne z dowolnego miejsca.
    Ten rodzaj dostępu jest domyślny dla interfejsów.
  - **private** : składowa zadeklarowana jako prywatna jest dostępna tylko z wnętrza typu, w którym została zadeklarowana.
    Ten rodzaj dostępu jest domyślny dla składowych klas i struktur.
  - **protected** : składowa zadeklarowana jako chroniona jest dostępna z wnętrza klasy, w której została zadeklarowana lub z wnętrza klasy pochodnej.
  - **internal** : typ lub składowa typu są dostępne tylko z wnętrza zestawu (ang. assembly), w którym nastąpiła ich deklaracja
    (podczas kompilacji pliki .cs z kodem źródłowym programu są kompilowane w moduły - zgodnie z podziałem na przestrzenie nazw - a następnie grupowane w zestawy)
  - **protected internal** : składowa zadeklarowana z takim rodzajem dostępu jest widoczna z wnętrza klasy, w której została zadeklarowana (lub klasy pochodnej od niej) oraz z wnętrza zestawu, w którym się znajduje.

Typy predefiniowane w C#

- Oprócz typów powszechnie znanych jak:
  int, uint, short, ushort, lung, ulong, byte, char, float, double
  mamy jeszcze:

- **bool** : (System.Boolean) typ logiczny mogący przyjmować wartości true lub false .
  Nie można dokonywać konwersji z bool na typ całkowity lub odwrotnie.

- **decimal** : (System.Decimal) liczba w systemie dziesiętnym zajmująca 12 bajtów (przechowuje 28 cyfr oraz pozycję punktu dziesiętnego w tych liczbach).
  Ma mniejszy zakres w porównaniu z liczbami zmiennopozycyjnymi jednak zapewnia bardzo dużą precyzję przechowywania liczb o podstawie 10. Liczba zapisywana jako dziesiętna wymaga przyrostka 'm' lub 'M', np. decimal d = 10.1m ;

- **object** : (System.Object) typ bazowy dla wszystkich innych typów (z wyjątkiem wskaźnikowych).

Typy predefiniowane w C#

- **string** : (System.String) reprezentuje łańcuch (zmiennej długości) znaków Unicode.

  Możemy używać tych samych sekwencji specjalnych, co w przypadku typu char, np. '\n', '\0', ...

  Pomimo tego, że **string** jest klasą (typem referencyjnym) ma pewne szczególne przywileje - można go tworzyć bez użycia operatora new,
  np. string s = " tekst \n " ;

Literały

- Poza zwykłymi literałami łańcuchowymi istnieją także tzw. literały dosłowne - zawarte wewnątrz @"...".
  Zawartość wewnątrz takiego literału jest brana 'dosłownie'. Tożsame są łańcuchy @"\\serwer\plik.txt" oraz "\\\\serwer\\plik.txt" .

- Zapisując literał oznaczający liczbę możemy określić typ literału itp. , np. 0x5 : liczba 5 w systemie szesnastkowym.

- Literały szesnastkowe mają przedrostek 0x

- 5UL : oznacza wartość 5 typu ulong (U - liczba bez znaku, L - liczba długa).

Zmienne – modyfikatory

- **static** : pozwala zadeklarować składową statyczną (istniejącą na rzecz całego typu, a nie pojedynczego obiektu tego typu).
  - Składowe statyczne nie wymagają istnienia obiektów.
  - Dana składowa statyczna istnieje jeszcze przed pojawieniem się pierwszego obiektu danej klasy
  - Ponadto istnieje tylko jedna jej wartość, wspólna dla wszystkich obiektów klasy,
    np. static string napisWspólnyDlaCałejKlasy = "..."

- **const** : pozwala zadeklarować stałą
  (daną składową, której wartość jest obliczana w czasie kompilacji i nie może być zmieniana w czasie działania programu).
  - Typ stałej musi być jednym z typów predefiniowanych
    (np. int, float, long, string),
    np. public const double PI = 3.14 ;

Zmienne – modyfikatory

- **readonly** : pozwala zadeklarować składową, której wartość nie będzie mogła być modyfikowana po początkowym nadaniu jej wartości.
  - W odróżnieniu od danych stałych (const) wartości danych tego typu są obliczane podczas działania programu, a nie podczas kompilacji. np. readonly MyClass a = new MyClass( ); readonly int b = 10;

- **volatile** : deklaruje pole typu nietrwałego.
  - Wartość pola w programie może być zmieniana przez kogoś innego jak: system operacyjny, sprzęt lub przez inny wątek (spoza programu). .

- Wartości domyślne to:
  - null – dla referencji (gdy tworzymy zmienną typu referencyjnego bez przypisania do niej obiektu).
  - 0 – dla wszystkich typów liczbowych (np. int, float) i wyliczeniowych.
  - false – dla typu logicznego bool.

Zmienne

- W przypadku typów referencyjnych nie wystarczy prosta deklaracja zmiennej.
- Deklaracja np. MyClass zm ; jest poprawna, jednak nie tworzy nowego obiektu typu, a jedynie samo wskazanie (zm nie wskazuje na żaden obiekt).
- Aby utworzyć egzemplarz typu referencyjnego konieczne jest użycie operatora **new**.
- **new** : alokuje pamięć na stercie oraz wywołuje konstruktor podanego typu, w celu utworzenia obiektu.
- Pozwala utworzyć dynamicznie obiekt (obiekty typu referencyjnego, takie jak klasy, tworzymy tylko dynamicznie).
- Aby utworzyć egzemplarz danej klasy używamy konstrukcji, np.

```
MyClass zm = new MyClass( );
```

Konwersje

- W C# istnieją konwersje jawne (przy użyciu rzutowania) i niejawne (przeprowadzane automatycznie w razie potrzeby).

- Rzutujemy stosując konstrukcję: *(typ) wartość* .

- Przykładowo:

```
int liczba = 100;
long długaLiczba = liczba ; // Konwersja niejawna
short krótkaLiczba = (short) liczba ; // Jawna konwersja
```

Konwersje

- W klasie możemy zdefiniować możliwość jej konwersji do innego typu.

- W tym celu należy zdefiniować metodę typu
  `public static implicit operator Typ ( ... ),`
  aby móc niejawnie przekształcać klasę do typu *Typ* lub funkcję
  `public static explicit operator Typ ( ... ),`
  aby móc to zrobić jawnie.

- Ponadto taka funkcja musi pobierać jeden parametr takiego typu jak definiowana klasa, np.

```
public class A {
     double wartość ;
     public static implicit operator double ( A a )
            { return a.wartość ; }
     public static explicit operator long( A a )
            { return (long)wartość; }
}
```

Tablice

- Tablice są obiektami (wywodzącymi się z **System.Array**) umożliwiającymi przechowywanie wielu elementów pewnego typu w ciągłym obszarze pamięci.

- Operator [ ] służy do tworzenia i indeksowania tablic (indeksy rozpoczynają się od zera).

- Po utworzeniu tablicy jej długość nie może być już zmieniona.

- W przypadku przekroczenia zakresu tablicy zgłaszany jest wyjątek *IndexOutOfRangeException.*

Tablice

- Tworzenie tablic typów bezpośrednich jest bardzo efektywne, np.

```
int[] tablInt = new int[ 100 ] ;
int[ ] tabl = { 1, 2 , 3 } ;
MojaStruktura[ ] tablStr = new MojaStruktura[100];
```

- Tworzenie tablic typów referencyjnych wymaga późniejszej inicjalizacji wartości tablicy, np.

```
mKlasa[] tablKlas = new mKlasa[100];
/* Na razie tablica jest wypełniona referencjami o wartości null. */
for ( int i=0; i < tablKlas.Length; i++ )
     tablKlas[i] = new mKlasa(); /* inicjalizacja tablicy. */
```

- Możliwe jest tworzenie tablic wielowymiarowych:

```
int [,,] tabl3D = new int [5][10][10]; // Tworzy regularną tablicę.
int [ ] [ ] tabl2D = new int [10] [ ] ; // Tworzy tablicę nieregularną.

for( int a=0; a < 10; a++ ) // tablice nieregularne są
     tabl2D[a] = new int[ a ] ; // tablicami tablic
```

Tablice

- Tablice posiadają składowe informujące o ich długości:

- **Length** : informuje o długości tablic jednowymiarowych, np.

```
for( int i=0; i < tabl.Length ; i++) ... ;
```

- **GetLength** : metoda pozwalająca uzyskać informację o długości tablicy w danym wymiarze (dla tablic wielowymiarowych), liczonym od 0. np.

```
for( int i=0; i < tabl3d.GetLength(2); i++) ... ;
```

Typ wyliczeniowy

- **enum** : słowo kluczowe pozwalające zdefiniować typ wyliczeniowy (nowy typ danych, składający się z nazwanych stałych numerycznych).

- Przykładowo:

```
// Definicja typu
public enum DniTyg { Pn, Wt, Sr, Czw, Pt, Sb, Nd } ;

. . .

DniTyg dzien = DniTyg.Pn ; // Deklaracja zmiennej. Nazwy składowych
                           // wyliczenia muszą być poprzedzone nazwą
                           // typu wyliczeniowego.
```

- Domyślnie kolejne elementy typu wyliczeniowego (stałe numeryczne) mają przyznawane wartości liczbowe w kolejności 0, 1, 2, 3 itd.
  Można jednak zdefiniować własny porządek, np.

```
public enum Kierunki { Północ = 1, Południe=2, Wschód=4, Zachód=8 }
Kierunki k = Kierunki.Północ | Kierunki.Wschód ;
```

Metody – modyfikatory

- **public** : (oraz private, protected, internal, protected internal). Modyfikuje dostęp do metody – zasada, jak przy składowych.

- **virtual** : pozwala tworzyć metodę wirtualną
  (przydatne przy dziedziczeniu– klasa pochodna musi metodę zdefiniować).

- **abstract** : pozwala utworzyć metodę abstrakcyjną
  (przydatne przy dziedziczeniu– klasa pochodna musi metodę zdefiniować)

- **extern** : wskazuje, że metoda jest zaimplementowana z użyciem kodu nienadzorowanego.

- **static** : metoda statyczna działa na rzecz całego typu, a nie pojedynczego obiektu wywołujemy ją według wzoru :
  `nazwaTypu.nazwaFunkcjiStatycznej()`
  Aby wywołać metodę statyczną nie potrzebujemy żadnego obiektu (klasy lub struktury).

Metody – modyfikatory

- Na liście parametrów poza typem możemy podać modyfikatory:
  - **ref** : pozwala przekazać dany parametr przez referencję (domyślnie parametry są przekazywane przez wartość.
    Modyfikator ref musi się pojawić na liście parametrów metody oraz podczas jej wywoływania).
  - **out** : parametr tego typu służy przekazaniu wartości z metody na zewnątrz (domyślnie parametr przekazuje wartość z zewnątrz do metody).
    Przekazując do metody zmienną w trybie out przekazujemy ją przez referencję, oczekując jednocześnie, że wewnątrz metody zostanie jej nadana odpowiednia wartość.
    Modyfikator out musi się pojawić na liście parametrów metody oraz podczas jej wywołania.
  - **params** : dzięki temu modyfikatorowi metoda może przyjmować dowolną liczbę parametrów.
    Modyfikator params może być zastosowany tylko do ostatniego parametru metody.

```
public class A {
      public void f_1 ( ref int parametr ) { ... }
      public void f_2 ( params int[ ] tabl ) { ... }
} ...
A a = new A( ) ;  // Utworzenie obiektu klasy
a.f_1( ref x ); // Przekazanie przez referencję zmiennej int x ;
a.f_2( 1, 2, 3 ); // Tablica tabl (wewnątrz metody f_2) będzie
                  // zawierała 3 elementy
```

- Możliwe jest przeciążanie metod

Właściwości

- Właściwości są bardzo podobne do pól ... oraz metod.
- Ogólna deklaracja właściwości:

```
[modyfikatory] typ nazwaWłaściwości {
      [modyfikatory] get { ... }
      [modyfikatory] set { ... } ...
}
```

- Modyfikatorami dla właściwości mogą być modyfikatory określające rodzaj dostępu (**public, private**, ...) oraz **virtual, static i abstract**. np.

```
public class A {
      float wartość ; // To jest pole prywatne
      public int wart { //nie może być nazwa wartość
                get { return (int)wartość ; }
                set { wartość = (float)value ; } //opcjonalnie
      }
}
...
A a = new A( );
a.wart = 10 ;              // użycie części set deklaracji właściwości
Console.Write( a.wart ); // użycie części get
```

Właściwości

- Część *get* właściwości musi zwracać wartość o takim samym typie jak typ właściwości,
  natomiast część *set* właściwości posiada parametr value, zgodny z typem właściwości.

- W rzeczywistości, w podanym wyżej przykładzie właściwość wart()
  zostanie przez kompilator przekształcona do dwóch metod – get_wart( ) oraz set_wart( ).

- Ponadto metody te będą typu inline (będą rozwijane w miejscu wywołania – co zapewnia większą szybkość ich 'wywołania').

- Definiując właściwość możemy określić samą część *get* – w takim wypadku będzie to właściwość tylko do odczytu

Właściwości

- Istnieje także specjalny rodzaj właściwości, który pozwala wprowadzić indeksowanie do tworzonej przez nas klasy.
- Od zwykłych właściwości różni się tym, że zamiast *nazwaWłaściwości(...)* wstawiamy *this[...]*. Np.

```
public class Tablica2D {
      ...
      int[ ] tablica = new int[ wlkX * wlkY ] ;

      public int this [ int x, int y ] {
               get { return tablica[ x + y*wlkX ] ; }
               set { tablica[x+y*wlkX] = (int) value ; }
      }
}
...
Tablica2D tabl = new Tablica2D( 10,10 );
tabl [0,0] = 0 ; // klasę można indeksować
```

Klasy

- Informacje ogólne
- Konstruktor
- Destruktor
- Przeciążanie operatorów

Klasy

- Klasy są podstawowymi typami referencyjnymi w C#.

- Ogólna deklaracja klasy:
  [modyfikatory] class nazwaKlasy [: nazwaNadklasy , interfejsy] { // Deklaracje składowych klasy }

- Przykładowo:

```
class Przykład {
     int wartość ;                          // deklaracja składowej
     public Przykład ( int wartość ) { // Konstruktor
             this.wartość = wartość ;
     }
}
```

Klasy

- Dostępne modyfikatory dla klasy:
  - **public** : (ew. internal) modyfikuje dostęp do klasy.
    Domyślnie klasa jest prywatna.
  - **sealed** : powoduje, że z danej klasy nie można dziedziczyć. np.
    `sealed class A { ... }`
  - **abstract** : tworzy klasę abstrakcyjną (mogącą zawierać metody abstrakcyjne).
    Przydatne przy dziedziczeniu.
- **this** : słowo kluczowe oznaczające referencję do obiektu, z którego wywołano metodę (inaczej mówiąc oznacza referencję do samego siebie). Oczywiście nie można używać **this** wewnątrz metod statycznych (one nie są wywoływane na rzecz obiektu).
- **base** : słowo kluczowe pozwalające na odwołania do składowych klasy bazowej (oznacza referencję do obiektu klasy bazowej "zaszytego" w obiekcie, z którego wywołano metodę).

Kontruktor

- Konstruktor jest specjalną metodą (o nazwie identycznej z nazwą klasy) wywoływaną automatycznie podczas tworzenia egzemplarza klasy (ew. struktury).
- Konstruktor nie posiada typu zwracanej wartości (nie zwraca nawet void).
- Można przeciążać konstruktory.
- Konstruktor nie pobierający żadnego argumentu jest konstruktorem domyślnym.
- Ponadto jeśli klasa nie definiuje żadnego konstruktora, to automatycznie jest dla niej tworzony konstruktor domyślny (inicjalizujący wszystkie pola domyślną wartością).
- Np.

```
class A { public A() {...} // Deklaracja konstruktora domyślnego }
```

Kontruktor

- Konstruktor klasy pochodnej musi najpierw wywołać konstruktor klasy bazowej.
- W przypadku, gdy klasa bazowa nie posiada konstruktora domyślnego, musimy jawnie wywołać jeden z jej konstruktorów.
  W tym celu używa się słowa kluczowego **base** .

- Ponadto definiując metodę, możemy wskazać, aby przed jej wykonaniem został wywołany jeden z konstruktorów (uzyskujemy to poprzez zapisanie po dwukropku słowa kluczowego **this** w formie wywołania metody).
- Przykład:

```
class MyClass : A {
     MyClass(int a): base(a){...} // Konstruktor wywołujący konstruktor nadklasy.
     MyClass(): this(0){...}      // Konstruktor domyślny, wywołujący
                                  // konstruktor z parametrem
}
```

Kontruktor

- Konstruktor może posiadać dowolny rodzaj dostępu.

- Ponadto możliwe jest tworzenie konstruktorów statycznych (jednak taki konstruktor nie może pobierać żadnych parametrów, np.

```
class Witacz {
     public static Witacz() {
             System.Console.WriteLine("Witaj");
     }
}
```

- Statyczny konstruktor wykonuje operacje potrzebne do inicjalizacji klasy.

- Statyczne konstruktory nie mogą być wywoływane bezpośrednio.

- Wywoływany jest automatycznie przy pierwszym dostępie do statycznych składowych klasy

Destruktor

- Destruktor jest specjalną metodą (o nazwie **~NazwaKlasy**) wywoływaną podczas usuwania obiektu z pamięci.
- Ma na celu posprzątanie po istnieniu obiektu danej klasy (np. zwolnienie zasobów systemowych).
- Należy jednak pamiętać, że w C# zwalnianiem nieużywanej pamięci zajmuje się system, a nie programista. np.

```
public class A {
    ...
    ~A( ) { ... } // Destruktor
}
```

  - Destruktory nie są dziedziczone.
  - Nie mogą pobierać żadnych parametrów.
  - Nie określamy także zwracanej wartości ani specyfikatora dostępu.
  - Destruktor jest wywoływany tylko automatycznie – nie można nakazać jego wywołania.
- Destruktory są implementowane poprzez przeciążenie metody *Finalize* z klasy *System.Object*.
- Programy w C# nie mogą bezpośrednio przeciążać ani wywoływać metody *Finalize*.

Destruktor

- Do wymuszania automatycznego wyczyszczenia wyczyścenia obiektów, do którego utracone juz zostały referencje służy wywołanie metody *Collect()* statycznego obiektu *Garbage Collectora*.

```
public class Klasa { Pen pen;
     public Klasa() { this.pen = new Pen(Brushes.Red); }
     ~Klasa() { }}

Klasa c = new Klasa();
c = null;
GC.Collect(); // uruchomienie GC (czyściciela) destruktora klasy
```

- Ale i tak system przeważnie sam decyduje o tym, kiedy uruchomić GC i może kompletnie pominąć wywołanie metody *Collect()*.

- Dodatkowo należy pamiętać, że obiekt GC wykonuje zwalnianie pamięci w osobnym wątku.
  Zwalnia zasoby utworzone w innym wątku naruszając ich przestrzeń adresową.
  Z uwagi na te niedogodności w C# zwykło się nie definiować metod destruktorów.

Wymuszanie zwalniania pamięci

- Dla potrzeb wymuszania zwalniania pamięci pamięcie wykorzystuje się interfejs IDisposable

```
public class Klasa : IDisposable {
      Pen pen;
      public Klasa() {
               this.pen = new Pen(Brushes.Red);
      }
      public void Dispose() {
               this.pen.Dispose();
      }
}
```

- następnie

```
using (Klasa c1 = new Klasa()) {
      // referencja c1 jest ważna tylko w obrębie klamer
      // każde wyjście po za klamrę, wyrzucenie wyjątku spowoduje
      // automatyczne uruchomienie przesłoniętej metody Dispose()
}
```

Przeciążanie operatorów

- C# pozwala przeciążać operatory.
- Aby to zrobić definiujemy funkcję statyczną nazywającą się *operator XX( )* gdzie w miejsce *XX* wstawiamy symbol jednego z przeciążalnych operatorów.
- Operatory, które można przeciążać to:

  + - * / % ! != == ~ << >> < <= > >= & | ^ ++ --

- Przykładowo:

```
class LZespol {
     double Re, Im ;
     LZespol ( int r, int i ) {
            Re = r ; Im = i ;
     } // Konstruktor
     public static LZespol operator + ( LZespol a, LZespol b) {
     // operator dodawania
            return new LZespol ( a.Re+ b.Re , a.Im + b.Im );
     }
}
```

Przeciążanie operatorów

- Pomimo przeciążenia operatory zachowują swój priorytet (np. zawsze operator mnożenia * ma wyższy priorytet niż operator dodawania).
- Operatory * oraz & dają się przeciążać tylko w wersji dwuargumentowej.
- Poza zwykłymi operatorami możemy także przeciążyć 'operatory' true oraz false .
- Operatory && oraz || są automatycznie rozwijane z operatorów & i |, dlatego też nie muszą być przeciążane.
- Operatory logiczne muszą być przeciążane 'parami'.
  To znaczy, że jeśli przeciążymy operator >, to musimy także przeciążyć operator < (pozostałe pary to == i != oraz <= i >=).

Struktury

- Struktury są podstawowymi typami bezpośrednimi tworzonymi przez programistę (klasy są typami referencyjnymi).
- Ponadto, w odróżnieniu od klas, struktury dziedziczą (niejawnie) tylko z klasy object i domyslnie są typu sealed (ze struktur nie można dziedziczyć).
- Struktura nie może także posiadać destruktora.
- Jednak struktury, tak samo jak klasy, mogą implementować interfejsy.

- Wzór deklaracji struktury:

```
[modyfikatory] struct nazwaStruktury [ : nazwyInterfejsów ] {
     // Deklaracje składowych struktury
}
```

- Przykładowo:

```
struct MojaStruktura {
     int x, y ;
     public void metoda() {...}
}
```

Klasy i metody abstrakcyjne

- **abstract** : słowo kluczowe pozwalające zadeklarować klasę / metodę jako abstrakcyjną.

- Metody abstrakcyjne są funkcjami nie posiadającymi implementacji i z założenia są wirtualne (gdyż będą przesłaniane w klasach pochodnych).

- Klasa abstrakcyjna może posiadać metody abstrakcyjne.
- Klasa pochodna od klasy abstrakcyjnej musi przesłaniać wszystkie jej metody abstrakcyjne lub sama musi być klasą abstrakcyjną.
  Np.

```
abstract class A {                              // Klasa abstrakcyjna
     public abstract void jakaśFunkcja( ) ;  // Metoda abstrakcyjna
}
```

Interfejsy

- Interfejs jest kolekcją metod abstrakcyjnych.
- Opisuje zbiór cech, które musi spełniać implementująca go klasa.
- Jest podobny do czystej klasy abstrakcyjnej (takiej, która zawiera tylko abstrakcyjne składowe).
- Interfejs z założenia nie dostarcza implementacji dla swoich składowych – są one implementowane dopiero w klasach lub strukturach dziedziczących z interfejsu (implementujących ten interfejs).
- Klasa (ewentualnie struktura) może implementować wiele interfejsów. Interfejs może dziedziczyć po wielu innych interfejsach.
- Wzór deklaracji interfejsu:

```
[modyfikatory] interface nazwaInterfejsu [ : nazwyNadinterfejsów ] {
        // Deklaracje składowych interfejsu
}
```

Interfejsy

- Przykład interfejsu:

```
public interface IZapisywalny { // Prosty interfejs.
     void Zapisz( ) ;           // Składowe interfejsu są
}                               // z założenia publiczne

public class MyClass : IZapisywalny { // Klasa implementująca interfejs.
     ...
     void funkcja( IZapisywalny z ) { // Funkcja przyjmująca parametr
               ...                    // typu implementującego interfejs
     }
}
```

- Klasa lub struktura może być niejawnie rzutowana do interfejsu, który implementuje.
- Interfejs może być niejawnie rzutowany do interfejsu, po którym dziedziczy.
- Interfejs może być jawnie rzutowany do każdego innego interfejsu lub klasy.

Szablony

- Szablony (ang. *Generics*) pozwalają na parametryzację klas, struktur, delegacji lub metod, w zależności od tego, jaki typ danych mają przechowywać lub przetwarzać.
- Przykładowo:

```
public class Stos<T> {
      T[ ] elementy;
      public void Push( T element) {...}
      public T Pop( ) {...}
}
...
// utworzenie obiektu stosu przechowującego wartości int.
Stos<int> stos = new Stos<int>( );
// włożenie nowej wartości na stos
stos.Push(3);
// zdjęcie liczby ze stosu (nie wymaga rzutowania)
int x = stos.Pop( );
```

- Wyrażenie `Stos<int>` jest typem gotowym do użycia – wewnątrz deklaracji klasy/szablonu **Stos** wszystkie wystąpienia parametru **T** są zamieniane na **int** i dopiero wtedy powstaje typ `Stos<int>`.

Szablony

- Oczywiście metody klasy `Stos<int>` (czyli `Pop` i `Push`) operują na klasie `int`.
- Eliminuje to konieczność rzutowania wartości zdejmowanej ze stosu (tym bardziej, że próba włożenia na taki stos wartości innego typu niż `int`, spowodowała by błąd już podczas kompilacji).
- Deklaracja szablonu może zawierać dowolną ilość parametrów (np. `class Łącznik<...>`).

- **where** : słowo kluczowe pozwalające nadawać ograniczenia na typ przekazywany jako argument szablonu.
  Dzięki temu możemy zapewnić silniejszą kontrolę typów podczas kompilacji oraz zminimalizować potrzebę rzutowania.

- Przykładowo:

```
public class Słownik<Klucz, Wartość > where Klucz: IComparable {
     public void Dodaj( Klucz k, Wartość w) {
              if( k.CompareTo(x) < 0 ) {...} // Nie jest wymagane rzutowanie
     }                                       // k do IComparable. }
```

Szablony

- Każdy typ, który ma wystąpić w miejscu parametru danego szablonu, musi w całości wypełniać wszelkie ograniczenia nałożone na dany parametr (w przeciwnym razie wystąpi błąd kompilacji).

- **default** : pozwala pobrać domyślną wartość danego typu.
  Dzięki temu możemy zainicjować wewnątrz szablonu w jednolity sposób zarówno zmienne typów bezpośrednich (*value type*) jak też referencyjnych (*reference type*).

```
class Wzorzec<T > {
     T t = default( T );
     ...
}
```

Szablony metod

- Czasami parametry nie są potrzebne dla całej klasy szablonu, ale tylko dla pewnych szczególnych metod.

- Zdarza się to często, gdy metoda operuje na parametrze, który jest szablonem. Np.

```
void PushMultiple<T>( Stos<T> stos, params T[] wartości) {
     foreach( T wartość in wartości)
              stos.Push( wartość );
}

Stos<int> stos = new Stos<int>();
PushMultiple<int>(stos, 1, 2, 3 ); // <-- wywołanie szablonu metody.
```

- W wielu wypadkach kompilator sam jest w stanie odkryć jaki typ powinien być obsłużony przez konkretne wywołanie szablonu metody.
  I tak w przykładzie powyżej można by metodę `PushMultiple` wywołać jako

```
     PushMultiple(stos, 1, 2, 3);
```

  (kompilator sam odkryje, że chodzi o `PushMultiple<int>`).

Typy częściowe

- Modyfikator `partial` pozwala rozbić definicję klasy (ew. struktury, interfejsu lub szablonu) na wiele kawałów, przechowywanych w różnych plikach.
- Przykładowo:

```
public partial class Klient { // Pierwszy kawałek definicji
     private int imie;
     private string adres;
     ...
}
public partial class Klient { // kolejny kawałek definicji klasy.
     public void zamów( ) { ... }
}
```

- Kiedy kompilujemy razem oba kawałki kodu, efekt jest taki sam jakby cała definicja klasy była zapisana w jednym kawałku.
- Jednak należy pamiętać o tym, że wszystkie części klasy muszą być skompilowane wspólnie (wszystkie kawałki definicji muszą być ze sobą scalone już podczas kompilacji).

Typy częściowe

- Każdy fragment typu częściowego musi zawierać modyfikator **partial**.
- Jeżeli deklarujemy klasę z tym modyfikatorem to wskazujemy, że jej pozostałe części mogą pojawić się w innym miejscu – ale wcale nie muszą.

- Ponadto wszystkie części tej samej klasy muszą być zdefiniowane w tej samej przestrzeni nazw oraz posiadać ten sam rodzaj dostępu (public, protected, internal czy private).

- Jeżeli którakolwiek z części klasy zawiera modyfikator **abstract** lub **sealed** to odnosi się on do całej klasy.

- Podobnie z dziedziczeniem – wystarczy, aby wskazanie klasy nadrzędnej wystąpiło jedynie w jednym miejscu.

Dziedziczenie

- Klasa może dziedziczyć (bezpośrednio) tylko z jednej nadklasy.

- Jednak każda klasa (a także struktura i interfejs) może implementować wiele interfejsów.

- Na szczycie hierarchii dziedziczenia znajduje się klasa **object** , po której niejawnie dziedziczą wszystkie klasy i struktury.
- Przykładowo:

```
class Przykład2 : Przykład {    // <- Klasa Przykład2 jawnie
     ...                        // dziedzicząca z klasy Przykład
     public Przykład2( ) : base(1) {    // <- Konstruktor, który jawnie
     }                                   // wywołuje konstruktor nadklasy }
```

- Klasa B dziedzicząca z klasy A może być niejawnie rzutowana (w górę hierarchii) do klasy A.
- Natomiast klasa A może być jawnie rzutowana (w dół hierarchii) do klasy B

  (w przypadku niepowodzenia jest zgłaszany wyjątek *InvalidCastException*).

Dziedziczenie

- **is** : operator pozwalający sprawdzić, czy obiekt należy do danej klasy, np.

```
if (p is Przykład2){...}
// Sprawdza czy p wskazuje na obiekt typu Przykład2
```

- **as** : operator pozwalający rzutować w dół w taki sposób, że gdy rzutowanie nie powiedzie się, zwracane jest null (a nie rzucany wyjątek), np.

```
Przykład p;
Przykład2 p2;
p2 = p as Przykład2;
```

Polimorfizm

- Oznaczając metodę jako wirtualną wskazujemy, że może ona (ale nie musi) być zrealizowana jeszcze raz w klasach pochodnych.

- Przed każdym wywołaniem takiej funkcji musi zostać sprawdzony rzeczywisty typ obiektu, na rzecz którego jest ona wywoływana
  (zostanie wywołana funkcja właściwa dla obiektu –
  nawet jeśli wskazujemy na niego referencją klasy podstawowej).

- Gdyby funkcja nie była wirtualna, została by po prostu wywołana funkcja odpowiednia dla typu, jakiego jest sama referencja – bez sprawdzania typu obiektu).

- **virtual** : słowo kluczowe pozwalające zadeklarować funkcję (w klasie bazowej) jako wirtualną
- **override** : słowo kluczowe pozwalające na przesłonięcie (w klasie pochodnej) funkcji wirtualnej.

Polimorfizm

- Przykład:

```
class A {
     public virtual void wypiszNazwę( ) {
               System.Console.Write("A");
     }
}

class B : A { // Klasa B dziedzicząca z klasy A
     public override void wypiszNazwę( ) {
               System.Console.Write("B");
     }
}
...

A a = new B( ) ;   //rzutowanie na typ bazowy A
a.wypiszNazwę( ); // W tym kodzie operacja a.wypiszNazwę() wypisze na ekranie "B",
                  // gdyż a wskazuje na obiekt klasy B
                  // (gdyby wipiszNazwę nie była funkcją wirtualną
                  // zostałoby wypisane "A" ).
```

Wyjątki

- Wyjątki to obiekty informujące o wystąpieniu pewnej wyjątkowej sytuacji (np. przekroczenia indeksu tablicy, ...).
- Wszystkie wyjątki muszą pochodzić od klasy **System.Exception** .
- Wyjątek może być zgłoszony przez funkcję biblioteczną lub wewnątrz metody użytkownika.
- Metoda, która wykryła błąd może zgłosić wyjątek (dzięki **throw**).
- Nie ma gwarancji, że będzie cokolwiek, co wykryje i obsłuży ten wyjątek (czyli odpowiednia instrukcja **catch**).
- Blok instrukcji obsługujących sytuacje wyjątkowe:

```
try {
      ...                       // W tym bloku może wystąpić wyjątek
} catch ( TypWyjątku w ) {  // Blok wychwytujący wyjątki i obsługujący sytuacje
wyjątkowe.
      ...                       // Może wystąpić wiele bloków catch.
} finally {                     // Instrukcje w tym bloku są wykonywane zawsze przed
      ...                       // opuszczeniem bloku try. Jest to blok opcjonalny.
}
```

  - **throw** : słowo kluczowe umożliwiające zgłoszenie wyjątku.
    Gdy zostanie zgłoszony wyjątek stos wywołań funkcji jest zwijany do momentu, aż wyjątek zostanie przechwycony przez jeden z bloków **catch**.
  - **try** : umożliwia utworzenie bloku programu, który jest w stanie wychwytywać wyjątki.

Wyjątki

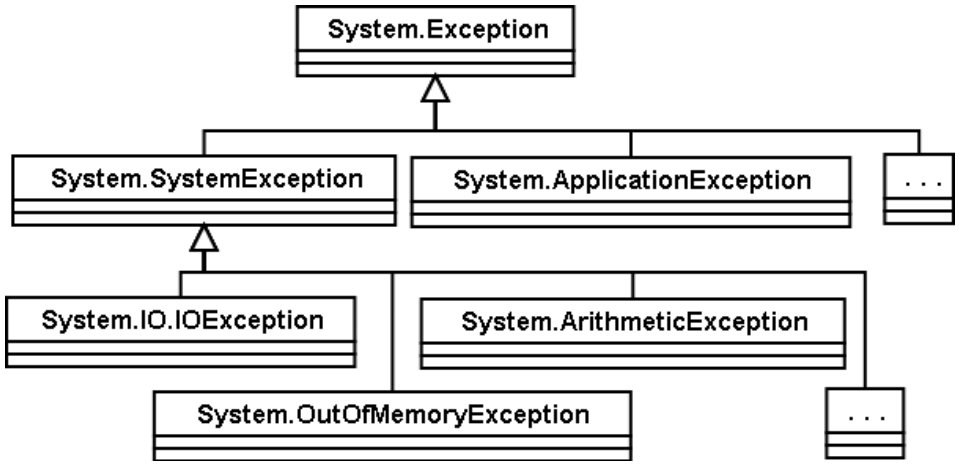
- Jako argument dla **catch** podajemy typ wychwytywanego wyjątku (np. `catch ( IOException e) { ...}`)
  – jeśli pominiemy typ wyjątku, to będą wychwytywane wszystkie wyjątki (skrócony zapis wychwytywania wyjątków typu **System.Exception**).

- Po bloku **try** może się pojawić dowolna ilość bloków **catch**, ale wykonywany jest tylko pierwszy pasujący blok
  (np. nie możemy umieścić bloku **catch** wychwytującego wyjątek typu pochodnego od E za blokiem wychwytującym wyjątek typu E, gdyż nie miał by on szans na wykonanie).

- Wszystkie instrukcje skoku (**throw**, **return**, **break**, **continue**, **goto**) są ograniczone przez instrukcję **try**.
- Skok poza blok **try** zawsze powoduje wykonanie bloku **finally** w danym bloku **try**.
- Ponadto nie można wykonać skoku z wnętrza na zewnątrz bloku **finally**.

Wyjątki

- Fragment drzewa dziedziczenia predefiniowanych klas wyjątków:



- Użytkownik tworzy wyjątki jako klasy pochodne od predefiniowanych wyjątków
  (wszystkie wyjątki muszą dziedziczyć po klasie **System.Exception**).
- Według konwencji typy wyjątków użytkownika powinny dziedziczyć z **System.ApplicationException**
- Ważne składowe klasy **System.Exception**:
  - **Message** : tekst opisujący błąd, używany do poinformowania użytkownika o jego naturze.
  - **StackTrace** : ścieżka wywołań funkcji do momentu wystąpienia wyjątku. Informuje, w którym dokładnie miejscu pojawił się błąd. Jest typu **string**.

Delegacje

- Delegacja pozwala utworzyć nowy typ danych, który definiuje sygnaturę metody.

- Obiekty delegacji mogą wywoływać metody zgodne z ich sygnaturą. Ogólna deklaracja delegacji:

```
[modyfikatory] delegate typ nazwaDelegacji ( ... ) ;
```

- Np.

```
delegate int Policz (string s); // <- Deklaracja (typu) delegacji mogącej wywoływać metody
                                // pobierające parametr typu string i zwracające int.

Policz p = Policz( nazwaMetody );// <- Utworzenie obiektu delegacji
p( ) ;                           // <- Wywołanie metody wskazywanej przez delegację
```

Delegacje

- Jeśli utworzymy delegację typu **void**, to będzie ona mogła wskazywać i wywoływać wiele metod jednocześnie (delegacje inne niż **void** nie mogą rejestrować wielu metod, gdyż nie miało by sensu zwracanie wyniku wywołania wielu metod na raz).

- W takim wypadku rejestrujemy kolejne metody, które ma wywołać delegacja dzięki operatorowi +=, natomiast wyrejestrowujemy za pomocą -=.
- Przykładowo:

```
class A {

   delegate void ListyFunkcji ( );              // Deklaracja delegacji typu void.

   public static void Main( ) {
      ListyFunkcji listaInicjalizacyjna = null;// <- Utworzenie obiektu delegacji
      listaInicjalizacyjna += przywitaj;        // <- Zarejestrowanie dodatkowej funkcji
      ...                                       // Następnie obiekt listaInicjalizacyjna
   }                                            // moglibyśmy przekazać do funkcji lub
                                                //wywołać.
   void przywitaj() {...}                       // <- Funkcja zarejestrowana w delegacji.
}
```

Operator lambda – anonimowość

- Operator lambda jest funkcją anonimową służącą do tworzenia delegatów,
- Za jego pomocą można pisać funkcje lokalne, które mogą być przekazywane jako argumenty lub zwracane jako wartość wywołania funkcji,
- Aby utworzyć wyrażenie lambda, należy określić parametry wejściowe (jeśli istnieją) po lewej stronie operatora lambda =>, i umieścić blok wyrażenia lub instrukcji po drugiej stronie.
- Wyrażenie lambda x => x * x określa parametr o nazwie x i zwraca wartość x kwadratów, a następnie całość można przypisać do typu delegata:

```
delegate int del(int i);
static void Main(string[] args)
{
    del myDelegate = x => x * x;
    int j = myDelegate(5); //j = 25
}
```

Operator lambda – anonimowość

- Operator lambda (=>) ma takie samo pierwszeństwo jak przypisania (=) i jest łączność do prawej strony,

- Ogólna forma to:

```
(input parameters) => expression
```

- Nawiasy są opcjonalne tylko wtedy, gdy lambda ma jeden parametr wejściowy; w przeciwnym razie są wymagane. Co najmniej dwa parametry wejściowe są rozdzielane przecinkami w nawiasach:

```
(x, y) => x == y
```

- Czasami jest trudne lub wręcz niemożliwe dla kompilatora wywnioskowanie typów wejściowych dlatego można określić typy jawnie:

```
(int x, string s) => s.Length > x
```

- Określanie braku parametrów wejściowych za pomocą pustych nawiasów:

```
() => SomeMethod()
```

Instrukcja lambda

- Lambda instrukcji jest podobna do wyrażenia lambda, z tym że instrukcje są ujęte w nawiasy klamrowe:

```
(input parameters) => {statement;}
```

- Treść lambda instrukcji może składać się z dowolnej liczby instrukcji

```
delegate void TestDelegate(string s);
…
TestDelegate myDel = n => { string s = n + " " + "World"; Console.WriteLine(s); };
myDel("Hello");
```

Zdarzenia

- Zdarzenia używane są do oprogramowania czynności zachodzących w trakcie działania aplikacji,
- Takim zdarzeniem może być ruch myszą, kliknięcie w obrębie komponentu, przesunięcie kontrolki i wiele innych,
- Po umieszczeniu przykładowej kontrolki na formie i kliknięciu w niego generowany jest automatycznie kod:

```
private void button1_Click(object sender, EventArgs e)
{

}
```

Zdarzenia

- Jak to się dzieje, że po naciśnięciu przycisku wykonywany jest kod znajdujący się w tej metodzie?

- Zdarzenia w dużej mierze opierają się na delegatach,

- W pliku **.designer.cs* można odnaleźć instrukcje, które odpowiadają za utworzenie przycisku Button na formularzu oraz za przypisanie określonej metody do danego zdarzenia:

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

- Zdarzenie Click zadeklarowane jest w klasie Control

```
public event EventHandler Click;
```

- Delegaty deklarujemy z użyciem słowa kluczowego delegate, a zdarzenia — z użyciem słowa event. Fraza EventHandler nie należy do słów kluczowych języka C#. Jest to nazwa delegatu określonego w przestrzeni System:

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

Zdarzenia

- Nic nie stoi zatem na przeszkodzie, aby przypisać do danego zdarzenia więcej niż jedną metodę:

```
private void button1_Click(object sender, EventArgs e)
{
    MessageBox.Show("Procedura zdarzeniowa #1");
}

private void myButton_Click(object sender, EventArgs e)
{
    MessageBox.Show("Procedura zdarzeniowa #2");
}

private void Form1_Load(object sender, EventArgs e)
{
    button1.Click += button1_Click;
    button1.Click += myButton_Click;
}
```

Lambda asynchronicznie

- Można łatwo tworzyć wyrażenia lambda i instrukcje, które obejmują wywołania asynchroniczne za pomocą **async** i **await** słów kluczowych,

- na przykład w poniższym przykładzie Windows Forms zawiera program obsługi zdarzeń, który wywołuje i czeka na metodę asynchroniczną

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        // ExampleMethodAsync returns a Task.
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\r\n";
    }

    async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

Lambda asynchronicznie

- Sam program obsługi zdarzeń można dodać, używając lambdy asynchronicznej,

- aby dodać taką obsługę, należy użyć **async** i modyfikator przed listą parametrów lambda:

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            // ExampleMethodAsync returns a Task.
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event handler.\r\n";
        };
    }

    async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

Przestrzenie nazw

- Przestrzenie nazw pozwalają grupować powiązane ze sobą typy (praktycznie cały program w C# jest zbiorem deklaracji typów).

- Różne przestrzenie nazw mogą się zagnieżdżać.
- Przestrzeń nazw deklarujemy używając słowa kluczowego **namespace**, np.

```
namespace Serverside { // Zamiast zagnieżdżania w sobie kolejnych deklaracji
      namespace DD {    // przestrzeni nazw możemy uzyskać ten sam efekt stosując
class Test { ... }     // kropki w nazwie przestrzeni nazw. Np. aby uzyskać taki sam
                }      // efekt jak po lewej stronie, można zapisać :
}                      // namespace Serverside.DD { ... }
```

- Pełna nazwa typu obejmuje także nazwę przestrzeni nazw, w której jest on zawarty.
- Tak więc, aby odwołać się do klasy zadeklarowanej w powyższej przestrzeni nazw **Serverside.DD** powinniśmy użyć nazwy: *Serverside.DD.Test*

Przestrzenie nazw

- **using** : słowo kluczowe pozwalające uniknąć konieczności stosowania pełnych nazw w stosunku do typów zawartych w innych przestrzeniach nazw, np.

  ```
  using Serverside.DD;
  ```

  spowoduje, że będzie można używać wszystkich nazw zawartych w przestrzeni *Serverside.DD* bez konieczności poprzedzania ich nazwą tej przestrzeni.

- Ponadto dzięki **using** możemy określić zastępczą nazwę dla typu lub przestrzeni nazw, np.

  ```
  using dd = Serverside.DD ;
  ```

  Od tej pory będzie można wpisywać **dd** zamiast **Serverside.DD**.

Przestrzenie nazw

- Najbardziej zewnętrzna przestrzeń nazw, w której zdefiniowane są wszystkie inne przestrzenie nazw, jest nazywana globalną przestrzenią nazw.

- Jeśli nie zadeklarujemy typu jawnie w przestrzeni nazw, to znajdzie się on w przestrzeni globalnej
  – będzie się można do niego odwoływać bezpośrednio z dowolnej innej przestrzeni nazw
  (aczkolwiek nie jest to w dobrym stylu programistycznym).

Zewnętrzne aliasy

- Przypuśćmy, że chcemy użyć w programie dwóch różnych wersji tej samej biblioteki.
- Prawdopodobnie znajdą się klasy, które w obu wersjach są umieszczone w tej samej przestrzeni nazw i posiadają te same nazwy.
- Odwołanie się do takiej klasy spowodowało by błąd (dla kompilatora obie klasy, ze starej i nowej biblioteki, są mają identyczne nazwy – oczywista dwuznaczność).

- Rozwiązaniem jest użycie zewnętrznych aliasów, np.

```
extern alias Curr;
extern alias Old;
class Test {
     Curr::NameSpc.MyClass a;  // Klasa MyClass pochodząca z najnowszej
     Old::NameSpc.MyClass b;   // ... i troche starszej biblioteki.
}
```

- Ponadto w momencie kompilacji musimy powiązać biblioteki z ich aliasami.

  A zatem podajemy odpowiednie parametry dla kompilatora, np. `csc /r:Old=a1.dll /r:Curr:a2.dll test.cs`

Iteratory

- Iteratory pozwalają w łatwy sposób wyspecyfikować jak instrukcja **foreach** ma poruszać się (iterować) po elementach pewnego typu (pewnej kolekcji).
- Iterator udostępnia uporządkowaną sekwencję wartości.
- Aby umożliwić iterację po elementach danej klasy należy implementować w niej interfejs **IEnumerable** lub **IEnumerable<T>** (z przestrzeni nazw **System.Collections** lub **System.Collections.Generic**), a co za tym idzie także metodę **GetEnumerator** tego interfejsu.
- Wewnątrz funkcji implementującej iterację zawieramy deklarację **yeld return** (zwraca kolejny element kolekcji) lub **yeld break** (wskazuje, że iteracja dobiegła końca).
- Przykład :

```
class Stack : IEnumerable {
     int[ ] table;
     ...
     public IEnumerator GetEnumerator() {
       for (int i = 0; i < 10; i++) {
         yield return table[i];
       }
       yield break; // Deklaracja opcjonalna
     }
}
```

Iteratory

- Możemy także tworzyć różne iteratory dla tej samej klasy (np. iterujące po elementach w innej kolejności) – w tym takie, które pobierają parametry.
- Przykładowo, gdyby dodać do klasy stosu dodatkowy iterator:

```
public IEnumerable FromToBy(int from, int to, int by) {
     for (int i = from; i <= to; i += by) {
             yield return table[i];
     }
}
```

można by iterować po wybranych elementach stosu zapisując:

```
foreach(int i in stos.FromToBy(10,20,2 ))
{...}
```

- Enumerowana kolekcja powinna działać jako fabryka enumeratorów. Jeśli jest poprawnie zaimplementowana to zwracane enumeratory są od siebie niezależne.

Kod nienadzorowany

- **unsafe** : słowo kluczowe pozwalające utworzyć blok nie-nadzorowany. W blokach nie-nadzorowanych możemy używać typów wskaźnikowych. Ponadto operacje w tych blokach zwykle działają szybciej (np. nie ma sprawdzania zakresów).

- **fixed** : pozwala blokować obiekty (zmienne) w pamięci.
  Informuje system, by nie zmieniał położenia w pamięci danego obiektu (standardowo system dowolnie umieszcza i przesuwa obiekty w obrębie sterty
  – jednak, gdybyśmy wskazywali na jakiś obiekt wskaźnikiem, a w międzyczasie zmienił by on swoje położenie, doprowadziło by to do błędów).

Kod nienadzorowany

- Przykładowo:

```
unsafe void mojaFunkcja( int[] tab ) {
    fixed( int* wsk = tab ) {
      int *w = wsk;
      for( int i=0; i<tab.Length; i++) {
         int a = *w++ ;
      }
    }
}
```

- Wskaźniki są to zmienne zawierające adres w pamięci innej zmiennej (wskazują na inną zmienną).
- Możemy ich używać tylko w blokach nie-nadzorowanych (unsafe).
- Każdy typ bezpośredni lub pośredni posiada odpowiadający mu typ wskaźnikowy – o takiej samej nazwie, ale zakończonej znakiem gwiazdki * (np. typowi *int* odpowiada typ wskaźnikowy *int**).
- Typy wskaźnikowe mogą być pomiędzy sobą rzutowane.

Kod nienadzorowany

- Operator * jest w stosunku do wskaźników operatorem wyłuskania – pozwala pobrać wartość znajdującą się pod adresem wskazywanym przez wskaźnik (czyli wartość zmiennej, którą wskazuje wskaźnik).
  Np. `int a = *wsk; //` gdzie `wsk` jest wskaźnikiem typu `int*`

- Operator `&` jest używany jako operator pobrania adresu.
  Wyrażenie `&zm` zwraca adres w pamięci, pod którym znajduje się zmienna zm.
  Np. `int* wsk = &liczba ; //` `liczba` jest zmienną typu `int`

- W C# zwracane są wskaźniki tylko do typów bezpośrednich, a nigdy bezpośrednio do typów referencyjnych
  (wyjątkowo w przypadku tablic i łańcuchów zwracany jest adres pierwszego elementu – oczywiście także jako typ bezpośredni).

- Operator -> służy odwoływaniu się do składowych struktury poprzez wskaźnik.
  Wyrażenie `a->b` jest równoważne wyrażeniu `(*a).b`

Kod nienadzorowany

- W stosunku do wskaźników możemy także używać operatorów
  ++ (wskaźnik będzie wskazywał kolejny element),
  -- (przesunięcie wskaźnika do poprzedniego elementu),
  + (ponieważ działanie operatora jest skalowane do rozmiaru wskaźnika, to np. (wsk + 5) oznacza piąty element od elementu wskazywanego przez wskaźnik. Nie można do siebie dodawać dwóch wskaźników.),
  - (od wskaźnika możemy odjąć wartość typu całkowitego lub inny wskaźnik),
  [ ] (umożliwia indeksowanie, tzn. wsk[5] jest równoważne *(wsk+5))
  oraz +=
- **stackalloc** : słowo kluczowe pozwalające alokować bloki pamięci na stosie (ponieważ są umieszczone na stosie, po wyjściu z bloku, w którym zostały zadeklarowane, przeznaczona na nie pamięć jest zwalniana), np.

```
int* wsk= stackalloc int[10];
System.Console.Write( wsk[5] ) ;
```

- **sizeof** : zwraca rozmiar struktury. Może być użyty jedynie w blokach unsafe, np. `int r = sizeof( int );`

Dokumentacja

- Komentarze możemy wstawiać pomiędzy znaki /* ... */ .
- Dozwolone jest także użycie //... pozwalającego na wstawienie komentarza do końca bieżącego wiersza.
- Ponadto w C# możemy wstawiać w kodzie źródłowym komentarze dokumentacji.
  Dzięki nim automatycznie, podczas kompilacji, zostanie wygenerowana dokumentacja.

- Komentarze dokumentacji wstawiamy używając /// ... (pozwala to wstawić komentarz do końca bieżącego wiersza).
  Służą one opisaniu typów oraz składowych.

- Kompilator sprawdza spójność komentarzy dokumentacji i tworzy z nich plik dokumentacji w XML (w tym celu musimy podczas wywoływania kompilacji przekazać do kompilatora parametr /doc:<nazwaPliku>).

Dokumentacja

- Przykład kodu źródłowego:

```
// Plik Program.cs
class MojaKlasa {
     /// <summary>
     /// Metoda wywoływana z
     /// <see cref="Main"> Main </see>
     /// </summary>
     /// <param name="a"> a to </param>
     void funkcja( int a ){ ... } ... }
```

- Wygenerowana dokumentacja:

```
<?xml version="1.0"?>
<doc>
  <assembly>
     <name> Program </name>
  </assembly>
  <members>
     <member name="M:MojaKlasa.funkcja( System.int )">
       <summary> Metoda wywoływana z
         <see cref="M:MojaKlasa.Main"> Main </see>
       </summary>
       <param name="a"> a to... </param>
     </member>
  </member>
  ...
</doc>
```

Predefiniowane znaczniki xml

- **<summary>** : wewnątrz tego znacznika krótko opisujemy typ lub jego składową
- **<remarks>** : wewnątrz tego znacznika dokładnie opisujemy typ lub składową
- **<param>** : służy opisaniu parametru metody.
  Zawiera obowiązkowy parametr name.
  Jeśli stosujemy znacznik param w stosunku do któregokolwiek z parametrów metody, to musimy także opisać wszystkie inne jej parametry,
  np. `<param name="nazwa"> opis parametru </param>`
- **<returns>** : Opisuje wartość zwracaną przez metodę,
  np. `<returns> ... </returns>`
- **<example>** : Pozwala podać przykłady kodu w dokumentacji.
  Znacznik ten z reguły zawiera pewien opis oraz zagnieżdżone znaczniki <c> oraz <code>.
- **<c>** : służy podaniu w dokumentacji przykładowego kodu.
  Może być umieszczony wewnątrz znacznika <example>, ale nie musi.
- **<code>** : służy podaniu przykładowego kodu.
  Znacznika <code> powinno się używać zamiast <c> w przypadku, gdy fragment kodu ma więcej niż jedną linijkę.

Predefiniowane znaczniki xml

- **<see>** : wstawia do dokumentacji odnośnik do miejsca opisującego inny typ lub składową.
  Posiada parametr cref, który służy podaniu nazwy typu/składowej.
  Np. `<see cref="MojTyp"> MojTyp </see>`
- **<seealso>** : to samo co <see>.
  Najczęściej jednak <see> jest formatowane jako odnośnik w opisie, natomiast <seealso> tworzy osobną sekcję 'zobacz też'.
- **<value>** : służy opisaniu właściwości należącej do klasy,
  np. <value> jest to... </value>
- **<para>** : oznacza paragraf.
  Nie ma żadnego znaczenia logicznego, pozwala tylko ładniej sformatować tekst.
  Jest używany wewnątrz takich znaczników jak <remarks> czy <returns>
- **<list>** : wstawić listę.
  Nie ma żadnego znaczenia logicznego, pozwala tylko ładniej sformatować tekst.
  Wewnątrz znacznika listy możemy użyć dodatkowych znaczników <item> oraz <listheader>.
  Ponadto określamy parametr type listy ("bullet", "table", "number").
  Np. `<list type="number"> <item> Struktura </item>  </list>`

Predefiniowane znaczniki xml

- **<exception>** : pozwala opisać wyjątek zgłaszany przez metodę.
  Opcjonalny parametr cref służy podaniu typu wyjątku.
  Np. `<exception cref="MojeWyjatki"> opis bledu </exception>`

- **<paramref/>** : Pozwala na wstawienie odnośnika do opisu parametru.
  Posiada parametr name identyfikujący parametr.
  Np. ... `Parametr <paramref name="Str1" /> jest ważny, gdyż ...`

- **<permission>** : opisuje uprawnienia wymagane do dostępu do składowej lub typu.
  Opcjonalny parametr cref służy podaniu typu reprezentującego uprawnienia.