Klasy cd.
Struktury
Interfejsy
Wyjątki

Struktury

- Struktura pozwala na zdefiniowanie typu danych, który nie charakteryzuje się zbyt złożoną funkcjonalnością (np. punkt, kolor, etc).

- Do definiowania struktury służy słowo kluczowe ***struct***.

- Składnia definicji struktury wygląda następująco:

[modyfikatory] struct Identyfikator [: lista_interfejsów]
{
składowe_struktury
}

Składnia definicji struktury

[modyfikatory] struct Identyfikator [: lista_interfejsów]
{
składowe_struktury
}

- *modyfikatory*

Dozwolony jest modyfikator *new* oraz modyfikatory dostępu (opcjonalne);

- *Identyfikator*

Nazwa struktury (wymagane);

- *lista interfejsów*

Lista zawierająca implementowane interfejsy oddzielone przecinkami (opcjonalne);

Struktury przykład

```
struct Punkt
{
    public int x, y;

    public Punkt(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Porównanie struktur z klasami

- Struktura jest typem wartości, a nie jak klasa typem referencyjnym.

- Strukturę można deklarować jak zmienną typu wartości:

```
Punkt  pkt;
pkt.x = 10;
pkt.y  = 100;
```

- Można również utworzyć instancję struktury:

```
Punkt pkt = new Punkt(10, 20);
```

Porównanie struktur z klasami cd. 1

- struktura nie wspiera mechanizmu dziedziczenia (wspiera mechanizm implementowania interfejsów);
- operacje na strukturze charakteryzują się większą wydajnością niż operacje na klasach (dane przechowywane są bezpośrednio);
- w strukturze nie można zdefiniować destruktora;
- w strukturze nie można zdefiniować konstruktora bezparametrowego;
- wewnątrz struktury nie można inicjować pól składowych:

```
struct Punkt
{
   public int x = 10,  y = 200;   // BŁĄD
}
```

Struktury - grupowanie pól

- Strukturę często wykorzystuje się jako typ grupujący inne pola:

```
struct Pracownik
{
    public string Imię;
    public string Nazwisko;
    public int Wiek;
}
```

Struktury - podsumowanie

- Mimo, że struktury mogą zawierać metody, nie zaleca się ich używania (struktury powinny przechowywać dane).

- Inaczej ma się sprawa definiowania operatorów, gdyż nie wprowadzają one nowych wzorców zachowania, a jedynie dostarczają funkcjonalność dla istniejących (np.: dodawanie).

Interfejsy

- Interfejs stanowi rodzaj kontraktu lub też specyfikacji funkcjonalności, jaka ma zostać zaimplementowana przez klasę (lub strukturę) go implementującą.

- Gdy dana klasa (struktura) obsługuje interfejs, to oznacza, że gwarantuje klientowi obsługę metody, właściwości czy też jakiegoś zdarzenia, które zostały wcześniej zdefiniowane w tym interfejsie. Innymi słowy, dzięki interfejsowi wymuszamy na klasie to co ona musi wykonywać, ale oczywiście nie określamy jak ma to robić.

- W interfejsie mogą znaleźć się jedynie sygnatury metod.

- Do definiowania interfejsów używa się słowa kluczowego ***interface***.

Składnia definicji interfejsu

[modyfikatory] interface Identyfikator [: lista_bazowa]
{
    ciało_interfejsu
}

- *Modyfikatory*
  Dozwolony jest modyfikator ***new*** oraz modyfikatory dostępu (opcjonalne),

- *Identyfikator*
  Nazwa interfejsu, zaleca się poprzedzać nazwę interfejsu dużą literą ***`I`*** (wymagane),

- *Listabazowa*
  Lista zawierająca interfejsy bazowe oddzielone przecinkami (opcjonalne),

- *ciało interfejsu*
  Sygnatury metod (opcjonalne).

Interfejsy - przykład

- Przykład:

```
interface ISlownik
{
   string Tlumacz(string Wyraz);
}
```

Cechy interfejsów

- domyślnie wszystkie sygnatury metod są publiczne, a jawne podanie modyfikatora dostępu nie jest dozwolone:

```
interface ISlownik
{
   public string Tłumacz(string Wyraz);   // BŁĄD
}
```

- w interfejsie można umieścić jedynie sygnatury metod (nie można ich implementować):

```
interface ISlownik
{
      string Tłumacz(string Wyraz)     // BŁĄD
      {
      }
}
```

Cechy interfejsów cd.1

- W C# dozwolone jest dziedziczenie tylko z jednej klasy, mimo to możliwe jest implementowanie wielu interfejsów w pojedynczej klasie:

```
interface IBazowy_l { }
interface IBazowy_2 { }
class Implementująca : IBazowy_l, IBazowy_2 { }
```

- Każdy interfejs może być interfejsem rozszerzającym dla wielu innych interfejsów

```
interface IBazowy_l { void Metodal(); }
interface IBazowy_2 { void Metoda2(); }
interface IRozszerzajacy: IBazowy_l, IBazowy_2 // wspólny kontrakt
{
}
```

- Żaden interfejs rozszerzający nie może być bardziej dostępny od interfejsu bazowego

Implementacja interfejsów

- Każda klasa implementująca dany interfejs musi zachować zgodność z kontraktem zawartym w interfejsie. Oznacza to, że wszystkie metody z implementowanych interfejsów muszą mieć zgodną sygnaturę z określoną w kontrakcie.

```
interface IBazowy
{
   void Metoda();
}
class Implementacja : IBazowy
 // BŁĄD:  musi być public
   protected void Metoda ()
   {
   }
}

class  Inplementacja2   :   IBazowy
// BŁĄD:  brak implementacji metody
{
   string Metoda()
   {
   }
}
```

```
interface IBazowy2
{
   void Metoda(int Liczba, string Nazwa);
}

class  Implementacja3   :   IBazowy2
// BŁĄD:  brak implementacji metody
{
   public void Metoda(string Nazwa, int Liczba)
   {
   }
}
```

Jawna implementacja metod interfejsów

- Alternatywnym sposobem implementacji metod interfejsów przez klasę jest ich tzw.: „jawna implementacja", czyli wskazanie, jaką metodę z jakiego interfejsu implementujemy.

```
interface IBazowy_I
{
   void Metoda();
}

interface IBazowy_2
{
   void Metoda();
}
```

```
class Implementacja : IBazowy_I, IBazowy_2
{
      public void IBazowy_I.Metoda()
      {
      }

      public void IBazowy_2.Metoda()
      {
      }
}
```

Wyjątki

- Każdy dobry program powinien umieć obsługiwać przypadki wystąpienia różnych błędów.

- Wyjątek oznacza zarówno błąd jak i nieoczekiwane zachowanie, występujące w czasie działania programu.

- Mogą one być zgłaszane zarówno jako rezultat: błędnie napisanego przez nas kodu, odwołania się do funkcjonalności umieszczonej w jakiejś bibliotece, wystąpienia błędu zgłoszonego przez system operacyjny (np.: braku zasobów) lub wystąpienia nieoczekiwanych warunków.

- W .NET Framework wyjątek jest obiektem, który dziedziczy z klasy wyjątku ***System.Exception***.

Wyjątki - bloki ***try*** i ***catch***

- Mechanizm obsługi wyjątków w C# realizowany jest za pomocą bloków ***try*** i ***catch***.

- Blok ***try*** jest blokiem, w którym umieszcza się logikę funkcjonalną programu, natomiast obsługa błędów odbywa się w bloku ***catch***.

Wyjątki - składnia

- Składnia obsługi wyjątków za pomocą bloków *try* i *catch*:

```
try
{
   logika_funkcjonalna
}
catch (klasa_wyjątku Identyfikator)
{
   obsługabłędów
}
```

  - logika_funkcjonalna - ciąg instrukcji programu realizujący zadaną funkcjonalność (opcjonalne);
  - *Klasawyjątku* - klasa obsługująca wyjątek, musi być klasą *System.Exception* lub dziedziczącą z niej (wymagane);
  - *Identyfikator* - nazwa instancji klasy obsługi wyjątku (opcjonalne);
  - *Obsługablędów* - ciąg instrukcji programu wykonywany w przypadku wystąpienia błędu (opcjonalne).

Wyjątki – przykład 1

```
try
{
    int  x  =   10,   y  =   0,   z;
    z = x / y;  // wyjątek: dzielenie przez zero
    z++;        // ta instrukcja nie zostanie już wykonana
}
catch (DivideByZeroException ex)
{
   Console.WriteLine(„Błąd: {0}”, ex.Message);
}
```

W momencie wystąpienia, wyjątku następuje skok do bloku *catch,* a pozostałe instrukcje znajdujące się w bloku *try* nie zostają wykonane.

Wyjątki - obsługa

- Ciąg instrukcji zawarty wewnątrz bloku try może zwrócić wiele różnych wyjątków, które mogą być obsługiwane przez wiele różnych klas.

- Umieszczając różne bloki *catch* należy pamiętać o właściwej kolejności. Wyjątki należy obsługiwać począwszy od szczegółowych do ogólnych (bazowa klasa *System.Exception* musi znaleźć się jako ostatnia)

Wyjątki – przykład 2

```
try
{
    int  x,   y =  0,   z;
    Console.Write("Podaj numer: ");
        // jeżeli wprowadzimy literę wystąpi wyjątek FormatException
    x = Convert.ToInt32(Console.ReadLine());
        // jeżeli wprowadzimy poprawny numer wystąpi wyjątek
        //DivideByZeroException
    z = x / y;
}
catch (FormatException)
{
    Console.WriteLine("Niepoprawny format.");
}
catch (DivideByZeroException)
{
    Console.WriteLine("Dzielenie przez zero.");
}
catch (Exception)
{
    Console.WriteLine("Nieznany błąd!");
}
```

Klasy wyjątków

- Wyjątki mogą być obsługiwane przez różne klasy.
- Podstawową klasą dla wyjątków jest *System.Exception.*
- W .NET Framework zdefiniowano ogromną liczbę klas wyjątków.

Składniki klasy *System.Exception*

Nazwa składnika	**Rodzaj składnika**	**Opis**
Exception	konstruktor	Tworzy instancję klasy.
HelpLink	właściwość	Pozwala na ustawienie i odczytanie linku do pliku związanego z wyjątkiem.
InnerException	właściwość	Pozwala na odczytanie instancji klasy *Exception,* która spowodowała wystąpienie aktualnego wyjątku.
Message	właściwość	Pozwala na odczytanie opisu tekstowego dla aktualnego wyjątku.
Source	właściwość	Pozwala na ustawienie i odczytanie nazwy aplikacji lub obiektu, który spowodował wystąpienie aktualnego wyjątku.
StackTrace	właściwość	Pozwala na odczytanie łańcucha znaków, będącego reprezentacją sekwencji wywołań metod w czasie wystąpienia aktualnego wyjątku.
TargetSite	właściwość	Pozwala na odczytanie nazwy metody, która spowodowała wystąpienie aktualnego wyjątku.

Podstawowe klasy wyjątków

Nazwa klasy wyjątku	**Nazwa klasy bazowej**	**Przeznaczenie**
Exception	*Object*	Bazowa klasa wyjątków, przechwytuje wszystkie błędy.
SystemException	*Exeeption*	Bazowa klasa wyjątków dla przestrzeni ***System.***
ArithmeticException	*SystemException*	Błędy: arytmetyczne, rzutowania i konwersji.
DivideByZeroException	*ArithmeticException*	Błąd dzielenia przez zero.
NotFiniteNumberException	*ArithmeticException*	Liczba zmiennoprzecinkowa jest dodatnio lub ujemnie nieskończona.
OverflowExeeption	*ArithmeticException*	Błąd przepełnienia w czasie działań arytmetycznych, rzutowania i konwersji.
FormatException	*SystemException*	Format argumentu nie jest zgodny ze specyfikacją wywołującej metody.
IndexOutOJRangeException	*SystemException*	Próba dostępu do elementu tablicy poprzez indeks spoza zakresu.
InvalidCastException	*SystemException*	Błąd rzutowania lub jawnej konwersji.

Rzucanie wyjątków

- W C# istnieje możliwość rzucania własnych wyjątków.

- Do tego celu służy instrukcja ***throw***, po której należy wskazać instancję klasy rzucanego wyjątku.

- Przykład:

```
throw new FileNotFoundException(NazwaPliku);
```

Rzucanie wyjątków - przykład

```
class KlasaTelewizor
    {
        private int glos;
        public int glosnosc        // właściwość
        {
          get { return glos; }
          set
         {
               if ((value < 0) || (value > 10))
                    throw new ArgumentOutOfRangeException();
              else
                    glos = value;
          }
      }
```

Własne klasy wyjątków

- Jeżeli chcemy stworzyć własną klasę wyjątku musimy pamiętać, aby dziedziczyła ona z *System.Exception* (lub innej potomnej).

Własne klasy wyjątków - przykład

```
class Test
{
    class NieAkceptowanaWartosc : Exception
    {
    }
    class Wartosci
    {
            public static void Procent(int Argument)
            {
                if (Argument < 0 ||  Argument > 100)
                    throw new NieAkceptowanaWartosc();  // rzucamy wyjątek
            }
    }
    static void Main()
    {
            try
            {
               Wartosci.Procent(101) ;
            }
            catch(NieAkceptowanaWartosc) // przechwytujemy nasz wyjątek
            {
                        Console.WriteLine("Niepoprawna wartość.");
            }
    }
}
```

Wyjątki - **Blok *finally***

- W przypadku wystąpienia wyjątku, dalsze wykonywanie instrukcji z bloku *try* nie jest kontynuowane, a sterowanie zostaje przekazane do bloku *catch.*

- Mechanizm wyjątków pozwala na dołączenie jeszcze jednego bloku, który jest wykonywany zawsze jako ostatni, bez względu na to czy wystąpi wyjątek czy nie.

- Jest to blok ***finally***. Blok ten współpracuje z blokiem *try* w podobny sposób jak blok *catch.*

Wyjątki - Blok *finally* – przykład

```
try
{
  //.....
}
catch (Exception)
{
 // obsługa wyjątku
}
finally
{
    // instrukcje np.: zwolnienie zasobów
}
```

- Blok *finalfy* jest przydatny, gdyż zapobiega sytuacji powielania się tych samych instrukcji w blokach *try* i *catch.*

Przestrzenie nazw

- Przestrzenie nazw grupują klasy w obrębie wspólnej logicznej struktury.

- Do deklarowania przestrzeni nazw służy słowo kluczowe ***namespace***.

Przestrzeń nazw - składnia

```
namespace Identyfikator[.Identyfikatorl[...]]
{
   definicja_typów
}
```

- *Identyfikator, Identyfikatorl, ...*
  Nazwa przestrzeni nazw (wymagane);

- *definicja_typów*
  Przestrzeń nazw może zawierać: inne przestrzenie nazw, klasy, struktury, interfejsy, delegacje i typy wyliczeniowe.

Przestrzeń nazw - przykład

```
namespace Matematyka
{
    interface IDzialania
    {
    }

    class Macierze
    {
    }

    class Wielomiany
    {
    }
}
```

Przestrzeń nazw wielokrotnie deklarowane

- Przestrzenie nazw mogą być wielokrotnie deklarowane w różnych plikach.
- Ponowna deklaracja przestrzeni o takiej samej nazwie, nazywa się otwarciem przestrzeni nazw.
- Po otwarciu przestrzeni nazw, można do niej dołączać kolejne klasy

Przestrzeń nazw - zagnieżdżenia

- Przestrzenie nazw można zagnieżdżać:

```
namespace Matematyka
{
    namespace Trygonomertia
    {
    }
}
```

- W przypadku zagnieżdżania przestrzeni nazw, można zastosować uproszczoną konwencję zapisu, która pozwala na zmniejszenie liczby linii kodu.

```
namespace Matematyka.Trygonometria
{
}
```

Przestrzeń nazw - cechy

- Przestrzenie nazw są publiczne. Nie można stosować w stosunku do nich żadnych modyfikatorów:

```
public namespace Matematyka.Trygonometria     // BŁĄD
{
}
```

- Zaletą przestrzeni nazw jest możliwość zdefiniowania typów o takich samych nazwach w różnych przestrzeniach (w tej samej przestrzeni nie mogą istnieć dwa typy o takiej samej nazwie).

Przestrzeń nazw
Nazwy kwalifikowane

- W przypadku używania klas wewnątrz tej samej przestrzeni, odwołanie do zdefiniowanego w przestrzeni typu odbywa się poprzez skróconą wersję zapisu nazwy typu. Takie odwołanie określa się mianem nazwy niekwalifikowanej.

- W przypadku odwoływania się do typów zdefiniowanych w innej przestrzeni nazw, mamy do czynienia z odwołaniem się poprzez pełną nazwę kwalifikowaną czyli całą ścieżkę począwszy od głównej przestrzeni nazw, poprzez podprzestrzenie aż do zdefiniowanego typu.

Przestrzeń nazw
Nazwy kwalifikowane - przykład

```
namespace Liczby
{
    public class Liczba
    {
    }
}

namespace Test
{
    static void Main()
    {
        Liczba lbl  = new Liczba();   // BŁĄD:  nieznany typ
        Liczby.Liczba lb = new Liczby.Liczba(); // kwalifikowana nazwa
    }
}
```

Przestrzeń nazw
Dyrektywa ***using***

- Dyrektywa ***using*** pozwala na skrócenie zapisu w przypadku odwoływania się do kwalifikowanej nazwy typu.

- Zamiast pisać przy każdym odwołaniu pełnej ścieżki, *możemy odwołać się bezpośrednio do typu zdefiniowanego* w innej przestrzeni nazw.

Przestrzeń nazw
Dyrektywa ***using*** - uwagi

- Używając dyrektywy *using* należy pamiętać o tym, że:
  - w przypadku zagnieżdżonych przestrzeni nazw należy użyć pełnej ścieżki począwszy od głównej przestrzeni poprzez kolejne zagnieżdżone aż do najbardziej zagnieżdżonej:

```
using System.Security.Cryptography;
```

  - dyrektywa *using* musi znajdować się w globalnym zasięgu (nie może znajdować się na końcu programu);

  - dyrektywa *using* może być umieszczona wewnątrz przestrzeni nazw:

```
namespace Test
{
    using Liczby;
    .....
}
```

  - w przypadku, gdy w dwóch różnych przestrzeniach zadeklarowanych poprzez *using* znajdują się typy o tych samych nazwach, należy odwoływać się do nich używając nazwy kwalifikowanej.

To już jest koniec ....