Opracował: L. Jung

Programowanie obiektowe
(w języku C#: Csharp)

Programowanie zorientowane obiektowo w języku C# (Csharp)

W ramach przedmiotu będą realizowane 3 projekty, składające się z dwóch części:

- laboratoryjnej: realizowanej wspólnie na zajęciach laboratoryjnych (które są obowiązkowe!),
- indywidualnej: realizowanej samodzielnie (wzorując się na części laboratoryjnej).

Ponadto, odbędą się 3 sprawdziany (na „podsumowanie" wykonanych Projektów) w terminach podanych przez wykładowcę.

Programowanie zorientowane obiektowo (object-oriented programing: OOP), to stosowanie paradygmatów programowania obiektowego bazujących na koncepcji klasy (class) opisującej konkretne obiekty fizyczne świata rzeczywistego lub obiekty konceptualne (pojęciowe) projektowanego programu.

W programowaniu zorientowanym obiektowo (OOP), obiekt jest deklarowany jako klasa (class) obejmująca łącznie, deklarację:

- odpowiednich pól (zmiennych), czyli struktur danych zawierających wartości cech (atrybutów) obiektu opisywanego daną klasą,
- metod opisujących dozwolone na nich działania.

Opracował: L. Jung

Uproszczony szablon zapisu deklaracji klasy (class):

modifikator dostępu (publicznego)

```
public class NazwaKlasy
{ // deklaracje pól (zmiennych) przechowujących wartości cech obiektu klasy
   Point PołożenieFigury;
   Color KolorFigury;
   float GrubośćLinii;
   .     .     .
   // deklaracja metod
   public Wykreśl ( . . . )  {. . . }
   public Wymaż ( . . . )  {. . . }
   public PrzesuńDoNowegoXY ( . . . )  {. . . }
.     .     ..
}
```

Projektując w języku C# programy, zarówno konsolowe jak i formularzowe, korzystaliśmy już z różnych klas (np. Math, Console, ...):

```
// Rozpoznanie wartości Delty
if (Delta > 0f)
{ // Obliczenie dwóch pierwiastkow równania
    X1 = (-b - (float)Math.Sqrt(Delta)) / (2f * a);
    X2 = (-b + (float)Math.Sqrt(Delta)) / (2f * a);
    // Wyprowadzenie wyników obliczeń
    Console.WriteLine("\n\t Równanie ma dwa pierwiastki rzeczywiste:\n");
    Console.WriteLine("\t X1 = {0, 8:G2} \t X2 = {1, 8:F3} \t Delta = " +
                                          "{2, 8:E3}", X1, X2, Delta);
}
else
   if (Delta < 0f)
```

udostępnianych przez odpowiednie przestrzenie nazw (np. System, System.Windows.Forms, System.Drawing, ...) środowiska narzędziowego języka Csharp.

Również otwierając nowy projekt, na przykład programu formularzowego, automatycznie zostaje utworzony szablon tego programu zapisany w konwencji programu zorientowanego obiektowego:

```
namespace ProgramFormularzowy
{
    3 references
    public partial class Form1 : Form
    {
        1 reference
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

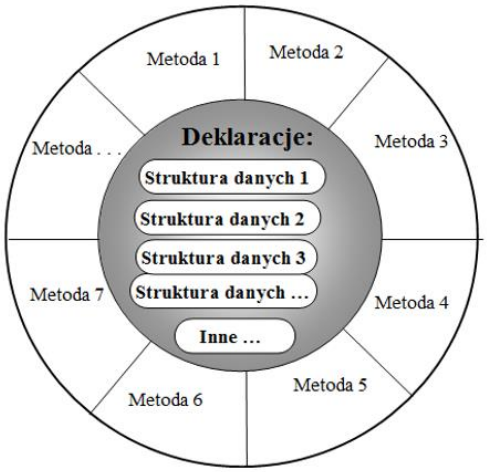
Programowanie zorientowane obiektowo (OOP: Object Oriented Programming).

Celem programowania zorientowanego obiektowo jest zapisanie:

- zhierarchizowanej struktury klas, która umożliwi korzystanie z właściwości trzech podstawowych paradygmatów programowania zorientowanego obiektowo: `hermetyzacji`, `dziedziczenia` i `polimorfizmu`,
- organizacji struktury programu jako zbioru współpracujących ze sobą obiektów, będących egzemplarzami klas.

`Paradygmat programowania`, to określony `wzorzec programowania`, który określa sposób implementacji struktur danych oraz przepływ sterowania w działającym programie komputerowym.

Obiekt (klasa) w programowaniu zorientowanym obiektowo jest rozumiany jako połączenie w jedną, spójną logicznie całość: struktur danych obiektu oraz metod (działań) opisujących funkcjonalność obiektu:

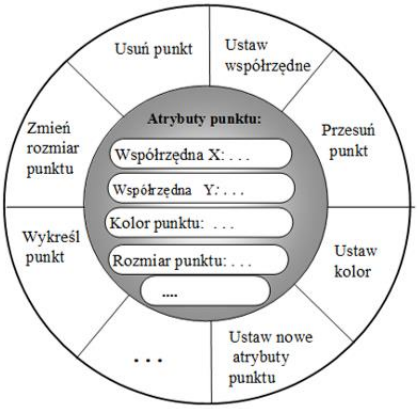


W programowaniu zorientowanym obiektowo, obiekt jest opisywany deklaracją odpowiedniej klasy obejmującej łączną deklarację:

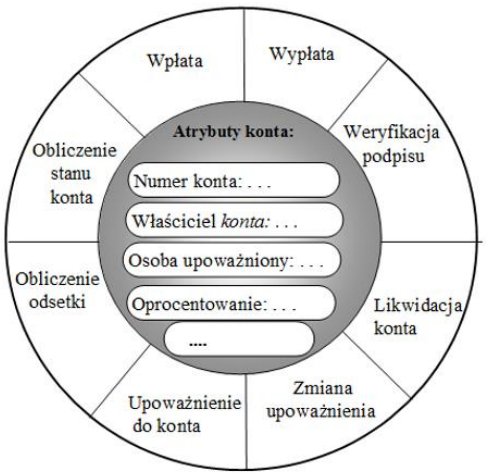
- struktur danych (zmiennych i właściwości), które umożliwiają przechowywanie wartości danych opisujących atrybuty egzemplarza klasy, (czyli obiektu klasy), a tymi atrybutami mogą być takie wielkości jak: wysokość, szerokość, kolor, waga, cena, rok produkcji, itd.,
- metod opisujących funkcjonalności przypisaną obiektowi klasy, czyli dozwolonych działań w odniesieniu do danego obiektu (czyli egzemplarza danej klasy), takich jak: Wykreśl (. . .), Wymaż (. . .), . . .

Na przykład, w programach:

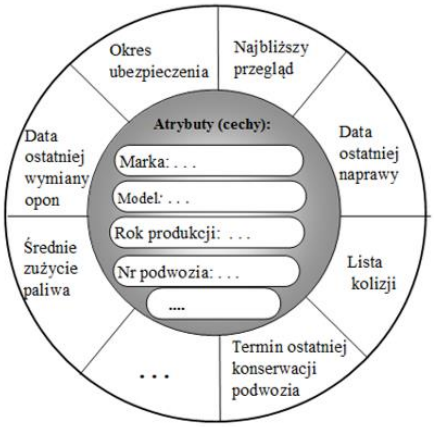
- z obszaru grafiki 2D (np. kreślenia figur geometrycznych): obiektami będą egzemplarze opisu figur geometrycznych, takich jak: punkt, okrąg, elipsa, kwadrat czy prostokąt):



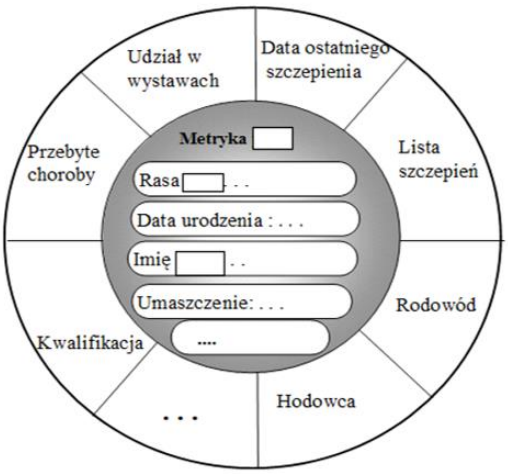
- obsługi kont bankowych: obiektami będą egzemplarze (metryki) opisu indywidualnych kont bankowych klientów banku:



- zarządzania kołowymi środkami transportu osobowego: obiektami będą egzemplarze opisu (metryki) osobowych środków transportu, takich jak: samochód osobowy, autobus, samolot czy statek:



- obsługi schronisk dla zwierząt: obiektami będą metryki opisu zwierząt:



Syntaktyczny zapis deklaracji klasy (class) w języku C#

```
modyfikator_dostępu class NazwaKlasy [: KlasaBazowa]
  { // deklaracje stałych
   modyfikator_dostępu const Typ_danej NazwaStałej = Wartość;
   . . .
   // deklaracje zmiennych klasy (pól klasy)
   modyfikator_dostępu Typ_danej NazwaZmiennej;
   . . .
   // deklaracja właściwości klasy
   public NazwaWłaściwości
   {
       // treść właściwości
   }
   . . .

    // deklaracja indeksatora klasy
    public float this[ushort . . ., ushort . . .]
     {
         set { . . . = value; }
         get { return . . .; }
     }
    . . .
    // deklaracje konstruktorów klasy
    public NazwaKlasy ( . . . )
      {
       . . .
      }
    . . .

   // deklaracje metod klasy
   modyfikator_dostępu void Nazwa_Metody1 ( . . . )
     {
      . . .
     }
  . . .
   modyfikator_dostępu TypDanych Nazwa_MetodyN ( . . . )
     {
      . . .
     }
   } // od class: NazwaKlasy
```

Modyfikatory dostępu

- modyfikator `public` udostępnia (nie ogranicza dostępu z zewnątrz klasy) deklarowane atrybuty klasy (stałe, zmienne, właściwości, metody, klasy):

```
public class ElementarneFiguryGeometryczne
{
    Odwołania: 3
    public class Punkt
    { // treść deklaracyjna klasy
      //  .    .    .

    }

    //  .    .    .

}
```

- modyfikator `private` ogranicza dostęp do deklarowanego atrybutu klasy z zewnątrz klasy, co oznacza, że jest on dostępny tylko w ramach treści deklarowanej klasy:

```
private void UstawNoweXY(Point NowaLokalizacja)
{
    X = NowaLokalizacja.X;
    Y = NowaLokalizacja.Y;
}
```

Modyfikator `private` jest jednocześnie modyfikatorem domyślnym, co oznacza, że jeżeli deklaracja zmiennej, stałej, metody oraz klasy nie jest poprzedzona żadnym modyfikatorem dostępu, to przyjmowane jest domyślnie wystąpienie modyfikatora `private`.

- modyfikator `protected` udostępniający (dostęp chroniony) deklarowany atrybut klasy tylko dla klas potomnych, a jednocześnie uniemożliwiający dostęp do niego z zewnątrz klasy:

```
public class Punkt
{ // treść deklaracyjna klasy
 // deklaracja pól klasy, które będą dostępne w klasach potomnych
    protected int X;
    protected int Y;
    protected int Grubość;  // linii
    protected Color KolorTła;
    // deklaracje atrybutów wspólnych dla wszystkich klas potomnych
    protected DashStyle StylLinii;
    protected Color KolorLinii;
```

- modyfikator `internal` określający dostęp tylko w obrębie tej samej przestrzeni nazw (komponentu, biblioteki, pliku *.dll):

```
namespace PrezenterFigurGeometrycznych
{
    1 odwołanie
    internal partial class FiguryKreśloneMyszą : Form
    {
        //  .     .     .
    }
    //  .     .     .
}
```

Elementy składowe klasy zadeklarowane jako `public` tworzą `publiczny interfejs klasy`, na który składają się: stałe, zmienne, właściwości (metody dostępu do pól prywatnych), metody (opisujące funkcjonalność egzemplarza danej klasy) i wewnętrzne klasy.

Podział klas w języku C#

- statyczne z kwalifikatorem (znacznikiem) `static`:

```
modyfikator_dostępu static class NazwaKlasyStatycznej
{

  // treść deklaracji klasy statycznej

}
```

Występujący w nagłówku deklaracji klasy kwalifikator `static` oznacza, że dana klasa będzie reprezentowana w programie tylko przez jeden jej egzemplarz, który tworzony jest automatycznie (nie wymaga dodatkowej deklaracji zmiennej referencyjnej danej klasy statycznej i tworzenie jej egzemplarza przy zastosowaniu operatora `new`).
W treści deklaracyjnej klasy statycznej wszystkie deklarowane jej atrybuty (zmienne, właściwości i metody) muszą też być statyczne (muszą być deklarowane z kwalifikatorem `static`.

Kwalifikator `static`

Kwalifikator `static` może być zapisany w nagłówku deklaracji klas, metod. zmiennych i właściwości, łącznie (w sąsiedztwie) wraz z zapisem innych modyfikatorów (`public`, `private`, `protected`, `internal`):

```
public static class EkstremumSzeregu
{   // deklaracje metod wyznaczających ekstremum wartości szeregu
    1 odwołanie
    public static float MinSx(float[,] TablicaWartościSzeregu)
    {   float WartoscMin;
        int i;
        WartoscMin = TablicaWartościSzeregu[0, 1];
        for (i = 1; i < TablicaWartościSzeregu.GetLength(0); i++)
            if (WartoscMin > TablicaWartościSzeregu[i, 1])
                WartoscMin = TablicaWartościSzeregu[i, 1];
        return WartoscMin;
    }
    1 odwołanie
    public static float MaxSx(float[,] TablicaWartościSzeregu)
    {   float WartoscMax;
        int i;
        WartoscMax = TablicaWartościSzeregu[0, 1];
        for (i = 1; i < TablicaWartościSzeregu.GetLength(0); i++)
            if (WartoscMax < TablicaWartościSzeregu[i, 1])
                WartoscMax = TablicaWartościSzeregu[i, 1];
        return WartoscMax;
    }
}
```

- dynamiczne bez żadnego dodatkowego kwalifikatora:

```
modyfikator_dostępu class NazwaKlasyDynamicznej
{

 // treść deklaracji klasy dynamiczne

}
```

Użycie klasy dynamicznej wymaga utworzenia jej egzemplarza, których w programie może być wiele (brak ograniczeń, co do liczby tworzonej liczby egzemplarzy klasy dynamicznej).

Utworzenie egzemplarza klasy dynamicznej wymaga:

- zadeklarowania zmiennej referencyjnej klasy dynamicznej: NazwaKlasyDynamicznej NazwaZmiennejReferencyjnej;
- utworzenie egzemplarza klasy dynamicznej przy zastosowaniu operatora `new`:

NazwaZmiennejReferencyjnej = `new` NazwaKlasyDynamicznej (...);

- abstrakcyjne z kwalifikatorem (znacznikiem) abstract:
  modyfikator_dostępu abstract class NazwaKlasyAbstrakcyjnej
  {
  // treść deklaracji klasy abstrakcyjnej
  }

  Występujący w nagłówku deklaracji klasy kwalifikator abstract oznacza, że dana klasa nie reprezentuje żadnego fizycznie istniejącego obiektu.
  Dla klasy abstrakcyjnej nie można utworzyć ich instancji (egzemplarzy), gdyż nie reprezentuje żadnego fizycznie istniejącego obiektu.
  Klasa abstrakcyjna jest klasą pomocniczą przy tworzenia klas dynamicznych dla których można tworzyć dowolną liczbę egzemplarzy.

Zmniejszanie stopnia złożoności procesu projektowania

Możliwość łączenia w klasie deklaracji struktur danych (zmiennych, właściwości) jak i metod (operacji) działania na nich jest ważnym mechanizmem językowym umożliwiającym dekomponowanie złożonego problemu na prostsze, czyli na praktyczne stosowanie zasady dekompozycji, tak ważnej w rozwiązywaniu zadań programistycznych.

Umożliwia bowiem koncepcyjne „opanowanie" złożoności tworzenia dużych projektów (systemów).

Program zaprojektowany według paradygmatu (reguł, zasad) projektowania zorientowanego obiektowo będzie bardziej elastyczny (łatwy w modyfikacji i w przyszłej rozbudowie) i adoptowalny do zmieniających się zastosowań, gdyż jego struktura będzie zorientowana na obiekty opisywanej rzeczywistości, a nie na funkcjonalności (akcji) jakie ma wykonywać.

Deklaracje zmiennych (pól) klasy

- służą do przechowywania wartości atrybutów danych klasy, które mogą być ustalane i zmieniane w trakcie działania programu,
- powinny być deklarowane z modyfikatorem `private` dla zapobieżenia możliwości zamiany ich wartości z zewnątrz klasy,
- mają ustalaną wartość domyślną przy tworzeniu egzemplarza danej klasy,
- jeżeli w konstruktorze klasy nie zostanie przypisana konkretna wartość deklarowanej zmiennej (polu) klasy, to zostanie jej przypisana wartość domyślna odpowiadającą jej typowi danych:

Domyślne wartości przypisywane deklarowanym zmiennym w klasie

(odpowiadające typowi danych deklarowanej zmiennej)

Typ	Wartość domyślna
byte	0
sbyte	0
short	0
ushort	0
int	0
uint	0
long	0
ulong	0
decimal	0.0
float	0.0
double	0.0
char	\0
bool	false
obiektowy	null

Egzemplarze (instancje) klas

Zadeklarowana klasa stanowi podstawę do utworzenia egzemplarza (obiektu) danej klasy (zwanego też jej instancją), który będzie reprezentował obiekt (instancję, egzemplarz) rzeczywisty:

- będą przechowywane w nim konkretne wartości danych opisujących atrybuty (cechy) obiektu rzeczywistego,
- będzie udostępniał metody opisujące zaprogramowaną funkcjonalność tego obiektu, czyli dozwolone działania jakie można na „nim” wykonać.

Stan (state) egzemplarza klasy (stan utworzonego obiektu) w danym momencie, opisany jest wartością jego pól (zadeklarowanych zmiennych i właściwości), które nazywane są atrybutami egzemplarza danej klasy (czyli obiektu).
Wartości pól obiektu mogą być zmieniane w czasie działania programu, co będzie skutkowało zmianą stanu obiektu.

Właściwości (w języku C#)

- Właściwość jest mechanizmem językowym umożliwiającym kontrolowany dostęp (zapisywania i odczytywania) do zmiennych (pól) klasy poprzez akcesory: get i set, które nazywane są odpowiednio: getterami i setterami.
- szablon syntaktyczny zapisu deklaracji właściwości:

```
private TypDanych NazwaZmiennej;
public TypDanych NazwaWłaściwości
    {
        get    // akcesor get
            {
            /* sekwencja instrukcji dla kontroli dostępu do aktualnej wartości
            danej właściwości  */
            return  NazwaZmiennej;;
            }

        set    // akcesor set
            {
            /*/ sekwencja instrukcji dla kontroli dozwolonej wartości, która
            ma być przypisana danej właściwości */
            NazwaZmiennej = value;
            /* value jest słowem kluczowym, które „przechowuje” wartość
            przypisywaną właściwości  */
            }
    }
```

Przykład deklaracji właściwości dla współrzędnej X (określającej położenie punktu na powierzchni graficznej utworzonej na ekranie laptopa)

```
private int _x;
public int X
 { set  { /* zezwolenie na ustalenie nowej wartości współrzędnej
             X tylko wtedy, gdy ta wartość należy do punktów
            ekranu; zapis użycia: X = 345; czyli value = 345 */
          if (value <= 0 || value >=
                                Screen.PrimaryScreen.WorkingArea.Width)
            throw new ArgumentOutOfRangeException("X");
          else   _x = value;
        }
   get  {// wartość współrzędnej X może być odczytywana bez
                                                        ograniczeń
          return _x;
        }
}
```

W definicji akcesora:

- get, umieszczamy instrukcje, które będą wykonywane w momencie, gdy użytkownik zażąda odczytu wartości danej właściwości (zmiennej),
- set umieszczamy instrukcje, które będą wykonywane w momencie, gdy użytkownik zażąda ustalenia nowej wartości danej właściwości (zmiennej), przy czym, przypisywana nowa wartość właściwości jest „przechowywana" w swoistym pojemniku value (ważnym w obrębie treści deklaracji właściwości).

Zarówno akcesor get jak i set mogą mieć podany modyfikator dostępu:

- public, wówczas sekwencja instrukcji przypisana danej sekcji będzie dostępna (do wykonania) z zewnątrz klasy,
- private, wówczas sekwencja instrukcji przypisana danej sekcji będzie wykonywana, tylko wtedy, gdy odwołanie do danej właściwości nastąpi wewnątrz danej klasy (na przykład w konstruktorze lub metodzie klasy),
- protected, wówczas sekwencja instrukcji przypisana danej sekcji będzie wykonywana, tylko wtedy, gdy odwołanie do danej właściwości nastąpi wewnątrz danej klasy (na przykład w konstruktorze klasy) lub w klasach pochodnych (od danej klasy, w której dana właściwość jest zadeklarowana).

Jeżeli akcesor nie ma jawnie określonego modyfikatora dostępu, to modyfikator dostępu określony dla właściwości jest automatycznie przenoszony dla jego akcesorów.

W skróconym zapisie deklaracji właściwości, przy akcesorach get i set można też podać inny modyfikator dostępu (private lub protected) dla uzyskania zwiększonej kontroli nad dostępem do deklarowanych zmiennych.

Gdy deklarowana właściwość ma umożliwić:

- tylko odczyt wartości prywatnej klasy, to powinna ona zawierać tylko akcesor get (właściwość typu "read_only"):

```
DashStyle RodzajLinii;
public DashStyle Styllinii
    {
      get
        {
          return RodzajLinii;
        }
    }
```

lub get i prywatny set, to akcesor set musi mieć wówczas przypisany modyfikator private (lub protected):

```
DashStyle RodzajLinii;
public DashStyle Styllinii
    {
      get
        {
          return RodzajLinii;
        }
     private set
        {
          RodzajLinii = value;
        }
    }
```

Konstruktory klas

Konstruktor klasy jest metodą, której nazwa jest taka sama jak nazwa klasy. Zapis deklaracji konstruktora klasy zawiera modyfikator dostępu `public`, nazwę klasy i listę parametrów formalnych, ale nie zawiera typu danych określającego zwracaną wartość, gdyż nie zwraca żadnej wartości.

```
public NazwaKonstruktoraKlasy (lista parametrów formalnych)
 {
  // treść algorytmiczna  konstruktora klasy
 }
```

`Konstruktor klasy` ma zadanie ustalenie wartości zmiennych (atrybutów) zadeklarowanych w klasie, czyli ustalenia „stanu początkowego" egzemplarza danej klasy. Jest wywoływany automatycznie przy tworzeniu egzemplarza klasy.

Klasa może mieć kilka konstruktorów (lub żadnego, ale wówczas tworzony jest automatycznie konstruktor bezparametrowy), które różnią się od siebie listą parametrów formalnych, co do liczby lub kolejności ich zapisu.

Właściwości języków programowania, zorientowanych obiektowo, zostały rozszerzone o możliwość:

- `przeciążanie metod i operatorów` (methods overloading, operators overloading), czyli definiowanie metod (i operatorów) o takich samych nazwach, ale różniących się między sobą listą parametrów formalnych (co do ilości, kolejności i ich typu danych):

```
public static Macierz operator +(Macierz a, Macierz b)
{if (a.LiczbaWierszy != b.LiczbaWierszy || a.LiczbaKolumn != b.LiczbaKolumn)
   throw new ArgumentException(„ERROR: niezgodny rozmiar macierzy");
  Macierz c = new Macierz(a.LiczbaWierszy, a.LiczbaKolumn);
  for (ushort i = 0; i < a.LiczbaWierszy; i++)
    for (ushort j = 0; j < a.LiczbaKolumn; j++)
      c.macierz[i, j] = a.macierz[i, j] + b.macierz[i, j];
  return c;
}
```

Po deklaracji klasy `Macierz` (nowy typ danych):
`class` Macierz
{ // treść deklaracyjna klasy Macierz }
utworzeniu i zainicjowaniu obiektów klasy Macierz:

```
float[,] MacierzA = new float[3, 3] { { 1.0F, 1.0F, 1.0F },
                                      { 2.0F,2.0F,2.0F },
                                      { 3.0F,3.0F,3.0F }};
float[,] MacierzB = new float[3, 3] { { 1.0F, 1.0F, 1.0F },
                                      { 2.0F,2.0F,2.0F },
                                      { 3.0F,3.0F,3.0F }};
float[,] MacierzC;
```

będziemy już mogli zapisywać działania rachunku macierzowego zgodnie ze znaną notacją matematyczną:

C = A + B; lub **C = A * B;** lub **C = A * B - A;**

- `programowania generycznego` (generic programming), czyli deklarowanie klas generycznych (zwanych też parametrycznymi), gdzie parametrem formalnym jest typ danych:
  - przykładem klas parametrycznych jest klasa generyczna `List`, która umożliwia przechowywanie kolekcji danych (wartości, obiektów klas) oraz efektywne ich przetwarzanie:
    `List`<typ danych(obiektu)> ListaObiektów=
    new `List`<typ danych(obiektu)> (**N**);
- klasę generyczną `List` stosujemy wtedy, gdy z góry nie możemy przewidzieć liczby elementów tworzonej kolekcji,

Gdy nie możemy przewidzieć liczby elementów tworzonej kolekcji, to możemy również zastosować (użyć) niegeneryczną kolekcję tablicy (której rozmiar dynamicznie ulega zmianie jeśli zachodzi taka potrzeba):
System.Collections.ArrayList
której elementy są typu `object`.

Lokalizacja zapisu deklaracji klas

Deklaracja nowej, „małej objętościowo" klasy może być zapisana w przestrzeni nazw lub w klasie projektowanego programu (również w pliku kodu formularza, np. o domyślnej nazwie Form1):

```
namespace LokalizacjaZapisuKlas
{
    Odwołania: 0
    class Klasa1...
    Odwołania: 3
    public partial class Form1 : Form
    {
        Odwołania: 0
        class Klasa2...
        1 odwołanie
        public Form1()...

        Odwołania: 0
        class Klasa3...
    }
    Odwołania: 0
    class Klasa4...
}
```

Deklaracja nowej klasy („większej objętościowo") może zapisywana również w oddzielnym pliku:

```
namespace ProjektNr2_XXXXX
{
    class FiguryGeometryczne
    {
        public class Punkt
        {
          . . .
        }
        public class Linia : Punkt
        {
          . . .
        }
    . . .
    }
```

W przypadku zapisu `złożonej klasy` (zawierającej deklaracje innych klas) w oddzielnym pliku, konieczne będzie udostępnienie jej treści deklaracyjnej w pliku (np. kodu formularza), który będzie miał korzystać z zapisanych w nim deklaracji, co należy zapisać :w sekcji using ze specyfikatorem static:

```
using static ProjektNr2_XXXXX.FiguryGeometryczne;

. . .
/* utworzenie egzemplarza Punktu i wpisanie jego referencji do
tablicy TFG (Tablica Figur Geometrycznych) */

TFG[IndexTFG] = new Punkt(Xp, Yp);
```