Advanced C# features

Dariusz Mikułowski

Instytut Informatyki Uniwersytetu Przyrodniczo-Humanistycznego W Siedlcach

29 grudnia 2020

1 Kolekcje

Kolekcje to wyspecjalizowane klasy pozwalające na łatwe przechowywanie i wyszukiwanie danych. Kolekcje zostały zdefiniowane w przestrzeni nazw System.Collections.
Jedną z popularniejszych kolekcji jest lista „ArrayList". Tak jak w tablicy, możemy w niej przechowywać wiele elementów.
Podczas deklarowania kolekcji nie musimy podawać typu przechowywanych w niej elementów. Jeśli jednak w kolekcji zapiszemy przypadkowo dane niepożądanego typu, może powstać trudny do wykrycia błąd, który pojawi się dopiero w czasie wykonania programu.
Trzeba również wiedzieć, że tablice działają szybciej niż kolekcje.

2 Kolekcje

Kolekcje dają większą swobodę stosowania, ponieważ dodawanie i usuwanie ich elementów jest realizowane w sposób dynamiczny. Kolekcja „ArrayList" posiada wiele metod, które upraszczają jej obsługę oraz udostępniają dodatkową funkcjonalność. Przykładem takiej metody jest metoda sort() sortująca elementy listy.
Do kolekcji możemy dodawać elementy wykonując metodę „Add()".

3 Przykład użycia kolekcji

```
using System;
using System.Collections;
namespace Kolekcje
{
  class Program
  {
    static void Main(string[] args)
    {
      ArrayList lista = new ArrayList();
      lista.Add(2);
      lista.Add(11);
      lista.Add(-2);
      lista.Add(4);
      Console.Write("Elementy nieposortowane: ");
      for (int i = 0; i < 4; i++)
      {
        Console.Write(lista[i] + " ");
      }
      lista.Sort();
      Console.Write("\nElementy posortowane: ");
      for (int i = 0; i < 4; i++)
      {
        Console.Write(lista[i] + " ");
      }
      Console.ReadKey();
    }
  }
}
```

4 Wstawianie elementu do kolekcji

Zawartość kolekcji możemy modyfikować w sposób dynamiczny. Do kolekcji możemy dodawać i wstawiać nowe obiekty, oraz je usuwać przy pomocy odpowiednich metod. Przykładowo:

```
using System;
using System.Collections;
namespace Kolekcje
{
  class Program
  {
    static void Main(string[] args)
    {
      Element[] tablica = new Element[2];
      tablica[0] = new Element(10);
      tablica[1] = new Element(20);
      ArrayList lista = new ArrayList(tablica);
      Console.Write("Elementy na liście: ");
      for (int i = 0; i < lista.Count; i++)
      {
        int liczba = ((Element)lista[i]).Liczba;
        Console.Write(liczba + " ");
      }
      Element nowyElement = new Element(123);
      lista.Insert(1, nowyElement);
      Console.Write("\nElementy na liście: ");
      ....
    }
  }
}
```

5 Modyfikowanie zawartości kolekcji

Metoda Insert() umieszcza element w kolekcji. Jako jej parametry podajemy indeks wstawianego elementu oraz wstawiany obiekt.
Zmienna Count określa, ile elementów zawiera kolekcja.
Jeśli chcemy usunąć jakiś element znając jego indeks, możemy użyć metody RemoveAt() podając indeks jako jej parametr.

6 Usuwanie elementu z kolekcji

```
  ...
  for (int i = 0; i<lista.Count; i++) {
    int liczba = ((Element)lista[i]).Liczba;
    Console.Write(liczba + " ");
  }
  lista.RemoveAt(0);
  Console.Write("\nElementy na   liście: ");
  for (int i = 0; i<lista.Count; i++) {
    int liczba = ((Element)lista[i]).Liczba;
    Console.Write(liczba + " ");
  }
  Console.ReadKey();
  }
}
  }
```

7 Standardowe kolekcje

- **ArrayList** - reprezentuje uporządkowaną kolekcję obiektów, które mogą być indeksowane indywidualnie.
- **Hashtable** - kolekcja w której dostęp do elementu jest realizowany za pomocą klucza;
- **SortedList** - lista ta jest kombinacją zwykłej tablicy (array) oraz tablicy mieszającej (Hashtable). Zawiera listę elementów do których dostęp uzyskujemy dzięki zastosowaniu klucza lub indeksu.
- **Stack** - reprezentuje stos zgodnie z zasadą last-in, first-out. Oznacza to, że ostatni dodany element jest pierwszy w kolejce do usunięcia z kolekcji.
- **Queue** - reprezentuje kolejkę zgodnie z zasadą first-in, first-out. Oznacza to, że pierwszy dodany element znajduje się na pierwszej pozycji w kolejce i jest pierwszym elementem do usunięcia.
- **BitArray** - reprezentuje tablicę binarnej reprezentacji z wykorzystaniem wartości 0 oraz 1. Jest stosowana gdy chcemy przechowywać bity, ale nie znamy z góry liczby bitów, które chcemy przechować.

8 Programowanie generyczne

Programowanie uogólnione (lub generyczne, z ang. generic programming). Jest paradygmatem programowania polegającym na tworzeniu kodu niezależnego od używanych typów danych. Typy te są przydzielane dopiero w czasie wykonywania się programu. Obecnie wiele języków ma możliwość wykorzystywania tego sposobu implementowania aplikacji. Są to np. C++, Java, Haskell. W językach C++ czy D programowanie uogólnione umożliwiają szablony. W językach Java, C#, VB.NET, Haskell, Eiffel służą do tego typy generyczne (lub inaczej uogólnione).

9 Zalety użycia typów generycznych

Programowanie generyczne polega na tym że deklarujemy zmienną klasę lub metodę nie podając jakich typów danych będzie ona używała. Te typy danych są ustalane dopiero w momencie tworzenia instancji danej klasy zmiennej czy metody, a więc na etapie wykonania.
Ponadto tej samej klasy czy metody generycznej możemy użyć wiele razy za każdym razem przetwarzając za jej pomocą inne typy danych. Przykładowo tej samej tablicy generycznej możemy za pierwszym razem użyć do przechowania liczb całkowitych a za innym razem przechowywać w niej znaki.

10 Zalety użycia typów generycznych

Główną zaletą użycia typów generycznych jest przyspieszenie działania programu.
W c# programowanie generyczne jest używane głównie w tzw. kolekcjach generycznych.
Korzystanie z typów generycznych pozwala także na uniknięcie wielu błędów na etapie wykonania programów i ich przesunięcie na etap kompilacji. Są one wtedy łatwiejsze do wykrycia.
Np. jeśli zamiast zwykłej kolekcji ArrayList użyjemy kolekcji generycznej „List" dopiero na etapie tworzenia egzemplarza tej kolekcji będziemy musieli zadeklarować - jakiego typu elementy ma ona przechowywać. Wtedy błąd związany z dodaniem złego typu będzie wykryty w czasie tworzenia instancji kolekcji i wykryje go kompilator, a nie pojawi się on w czasie wykonania programu.

11 Użycie typów generycznych

Kolekcję generyczną deklarujemy podając jednocześnie typy przechowywanych w niej obiektów. Możemy to zrobić w następujący sposób:

```
using System;
using System.Collections.Generic;
namespace KolekcjeGeneryczne
{
  class Program
  {
    static void Main(string[] args)
    {
      List<int> lista = new List<int>();
      lista.Add(2);
      lista.Add(5);
      int suma = lista[0] + lista[1];
      Console.WriteLine("Suma wynosi: " + suma);
      Console.ReadKey();
    }
  }
}
```

Jeśli spróbujemy dodać do kolekcji generycznej element innego typu niż ten, który został zadeklarowany, zostaniemy powiadomieni przez kompilator o błędzie.

12 Użycie typów generycznych

Ale tej samej kolekcji listy możemy też użyć do przechowania danych innego typu np string:

```
using System;
using System.Collections.Generic;
namespace KolekcjeGeneryczne {
  class Program {
    static void Main(string[] args) {
      List<string> lista = new List<int>();
      lista.Add("Pierwszy");
      lista.Add("drugi");
      string napis = lista[0] +" " + lista[1];
      Console.WriteLine("napis: " + napis);
      Console.ReadKey();
      }
  }
}
```

13 Użycie typów generycznych

Możemy też utworzyć własną klasę generyczną i użyć jej do utworzenia dwóch tablic: najpierw zawierającej liczby, a potem kolejnej zawierającej znaki:

```
using System;
namespace Generics
{
    // Definicja klasy generycznej, w tym przypadku własnej tablicy
    class MyGenericArray<T>
    {
        private T[] genericArray;
        public MyGenericArray(int size)
        {
            // ustalenie rozmiaru tablicy generycznej
            genericArray = new T[size + 1];
        }
        // zwracamy dowolny typ danych
        public T getGenericItem(int index)
        {
            return genericArray[index];
        }
        // ustawiamy dowolny typ danych
        public void setGenericValue(int index, T value)
        {
            genericArray[index] = value;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            // Tworzymy tablicę liczb całkowitych i wypełniamy ją
            MyGenericArray<int> intArray = new MyGenericArray<int>(5);
            for (int i = 0; i < 5; i++)
            {
                intArray.setGenericValue(i, i * 3);
            }
            // Wypisujemy  wszystkie dane
            for (int i = 0; i < 5; i++)
            {
                Console.WriteLine("Liczba: {0}", intArray.getGenericItem(i));
            }
            // Używając tej samej generycznej klasy deklarujemy inny typ danych
            MyGenericArray<char> charArray = new MyGenericArray<char>(5);
            for (int i = 0; i < 5; i++)
            {
                charArray.setGenericValue(i, (char)(i + 97));
            }
            for (int i = 0; i < 5; i++)
            {
                Console.WriteLine(charArray.getGenericItem(i));
            }
            Console.ReadKey();
        }
    }
}
```

14 Przeciążanie metod i konstruktorów

Przeciążenie metody (ang. Method overloading) – jest to utworzenie nowej wersji metody. Polega ono na zmianie sygnatury tej metody, czyli liczby parametrów jaką ta metoda przyjmuje, lub zmianie typu tych parametrów, oraz nowym zaimplementowaniu tej metody.
W języku C# możemy również przeciążać konstruktory.

15 Przykład przeciążenia metody

```
class Przeciazenie
{
  public void MojaMetoda()
  {
    System.Console.WriteLine("Brak parametrów.");
  }
  //przeciazenie metody - dodanie jednego parametru
  public void MojaMetoda(int param1)
  {
    System.Console.WriteLine("Jeden parametr w metodzie, którego wartość wynosi: { 0}
      ", param1 + ".");
      }
  //przeciazenie metody - dodanie dwoch parametrow
  public void MojaMetoda(int param1, int param2)
  {
    System.Console.WriteLine("Dwa parametry w metodzie. Parametr pierwszy to: { 0}, a parametr drugi to: { 1} ",
      param1,param2 + ".");
  }
  //przeciazenie metody - parametr typu string
  public string MojaMetoda(string param1)
  {
    return param1;
  }
}
```

16 Klasy zamknięte i abstrakcyjne

Klasy zamknięte charakteryzują się tym, że nie można tworzyć od nich klas pochodnych. Klasę zamkniętą definiujemy poprzez użycie przed jej nazwą słowa kluczowego sealed.
Przeciwieństwem klasy zamkniętej jest klasa abstrakcyjna.
Deklarację klasy abstrakcyjnej poprzedzamy słowem kluczowym abstract.

17 Klasy zamknięte i abstrakcyjne

Dalsze różnice pomiędzy klasami zamkniętymi a abstrakcyjnymi:

- w klasie zamkniętej nie jest możliwe tworzenie składowych chronionych (protected),
- klasa zamknięta nie może definiować metod abstrakcyjnych,
- niedozwolone jest jednoczesne zadeklarowanie klasy jako abstrakcyjnej i zamkniętej.
- klasa abstrakcyjna nie jest kompletna i dopiero w podklasach implementujemy jej zawartość (abstrakcyjne metody).

18 Klasy zamknięte

Klasy zamkniętej używamy w sytuacji, gdy chcemy zapobiec przeciążaniu (przesłanianiu) jej metod albo dziedziczeniu klasy.
W klasie zamkniętej wszystkie metody są domyślnie deklarowane jako zamknięte (ostateczne).

19 Klasa Object

Klasa Object jest to klasa główna (ang. root) w języku C#, od której zaczyna się hierarchia dziedziczenia klas. Wszystkie klasy używane w języku są traktowane jako klasy pochodne od klasy Object. Klasa ta udostępnia nam wiele wirtualnych metod, które często przesłaniamy w klasach pochodnych.
Wybrane metody klasy Object:

1. Equals() - sprawdza, czy dwa obiekty są sobie równe,
2. GetType() - sprawdza typ danego obiektu,
3. ToString() - zwraca łańcuch znaków, który reprezentuje dany obiekt.

Ważne jest również to, że podstawowe typy danych (takie jak np. liczby całkowite) są pochodnymi od klasy Object. Dzięki temu, możemy używać w nich takich metod jak np. ToString().

20 Interfejsy

W czasie programowania aplikacji w różnych językach takich jak Java czy C++ używamy pojęcia interfejsu. Interfejsy są również dostępne w języku C#.
Ogólna deklaracja interfejsu w C# jest następująca:

```
interface nazwa_interfejsu
{
  //składowe, zmienne, metody...
}
```

Składnikami interfejsu mogą być metody bez implementacji działania i specyfikatora dostępu, oraz zmienne.

21 Interfejsy

Przykład deklaracji interfejsu:

```
interface Figury {
 double Pole(); }
```

Do interfejsu możemy dodawać własności oraz indeksery. Przy dodawaniu własności lub indeksera powinniśmy wyspecyfikować akcesory, jakie ma on zawierać.

```
interface Figury {
 double Obwod { get; }
 double Pole(); }
```

22 Implementacja interfejsu

Interfejs implementujemy w postaci klasy. Przykład kodu interfejsu i jego implementacji:

```
using System;
interface Figury
{
  double Obwod { get; }
  double Pole();
}
class Kwadrat : Figury
{
  public int bok_a;
  public Kwadrat(int a)
  {
    bok_a = a;
  }
  public double Pole()
  {
    return bok_a * bok_a;
  }
  public double Obwod
  {
    get
    {
      return 4 * bok_a;
    }
  }
}
```

23 Wyjątki

W C#, podobnie jak w innych językach takich jak Java, istnieje mechanizm przechwytywania i obsługi wyjątków.
Popularnymi wyjątkami w .Net są:

1. ArrayTypeMismatchException - Typ wartości jaki chcemy przypisać jest niezgodny z typem docelowym,
2. DivideByZeroException - Próba dzielenia przez zero,
3. IndexOutOfRangeException - Przekroczenie indeksu,
4. InvalidCastException - Niepoprawne rzutowanie w czasie rzeczywistym,
5. OutOfMemoryException - Porażka wywołania new z powodu braku wolnej pamięci,
6. OverflowException - Arytmetyczne przepełnienie stosu,

24 Zgłoszenie wyjątku

Ogólna deklaracja zgłoszenia wyjątku w C#

```
throw new nazwa_wyjątku();
```

25 Zgłoszenie wyjątku

```
using System;
class liczby
{
  int[] tab;
  public liczby(int rozmiar)
  {
    tab = new int[rozmiar];
  }
  public int this[int index]
  {
    get { return tab[index]; }
    set
    {
      if (value % 3 == 0)
      {
        if (index <= tab.Length - 1) tab[index] = value;
        else throw new IndexOutOfRangeException();
        // Zgłaszamy wyjątek ;
      }
    }
  }
}
```

26 Instrukcje Checked i Unchecked

Instrukcje checked i unchecked służą do kontrolowania przepełnień arytmetycznych. Różnica między nimi jest taka, że checked podczas wystąpienia przepełnienia zgłasza wyjątek, a unchecked wyjątku nie zgłasza. Wartość przepełnienia jest wówczas dostosowywana do możliwego zakresu zmiennej, która je spowodowała. Gdy wartość mnożenia będzie większa niż 8 nastąpi wyjątek.

```
using System;
class Pokaz
{
  public static void Main()
  {
    byte x = 125;
    byte y = 7;
    byte z;
    try
    {
      checked
      {
        z = (byte)(x * y);
      }
    }
    catch (OverflowException)
    {
      Console.WriteLine("Nastąpiło przepełnienie");
    }
  }
}
```

27 Strumienie

W C#, podobnie jak w języku Java, zapis i odczyt danych z zewnętrznych źródeł takich jak np. plik odbywa się przy pomocy strumieni. Strumień z zasady musi posiadać początek i koniec. Początkiem strumienia jest to, co wysyła dane, a końcem miejsce do którego są one przesyłane. Miejscami takimi mogą być np.: plik na dysku, ekran monitora, klawiatura, łącze TCP itd.
Strumienie plikowe zostały zdefiniowane w klasie FileStream, która jest zadeklarowana w przestrzeni nazw System.IO. Ogólna deklaracja strumienia plikowego FileStream jest następująca:

```
FileStream nazwa = new FileStream(string ścieżka,
   FileMode tryb, FileAccess dostep);
```

Gdzie:

- ścieżka - jest zapisem ścieżki pliku na dysku np. „c: opowiadanie.txt".
- FileMode - określa tryb dostępu do pliku który mówi, co ma być zrobione z plikiem (otwarty lub utworzony),
- FileAccess - określa sposób dostępu do pliku (zapis, odczyt).

28 Strumienie plikowe

Możemy używać następujących wartości właściwości FileMode:

1. fileMode.Open - Otwiera plik, jeżeli taki nie istnieje, zgłasza wyjątek.
2. FileMode.OpenOrCreate - Otwiera plik, jeżeli taki nie istnieje, tworzy go.
3. FileMode.Append - powoduje dopisywanie danych na końcu pliku.
4. FileMode.CreateNew - Tworzy nowy plik, jeżeli plik już istnieje, zostanie zgłoszony wyjątek.
5. FileMode.Create - Tworzy nowy plik, jeżeli plik już istnieje, zostanie zastąpiony nowym.
6. FileMode.Truncate - Otwiera plik i kasuje jego zawartość, jeżeli plik nie istnieje, zostanie zgłoszony wyjątek.

29 Strumienie plikowe

Parametr FileAccess określa tryb dostępu do pliku. A oto jego możliwe wartości:

- FileAccess.Read - Plik tylko do odczytu.
- FileAccess.Write - Plik tylko do zapisu.
- FileAccess.ReadWrite - Plik do odczytu i zapisu

30 Użycie strumieni plikowych

```
using System;
using System.IO;
class Pokaz {
  public static void Main() {
    FileStream plik;
    int w;
    char znak;
    try {
      plik = new FileStream("c:\\tekst.txt", FileMode.Open);
    }
    catch (FileNotFoundException) {
      Console.WriteLine("BŁĄD! - Brak pliku tekst.txt");
      return;
    }
    do  {
      w = plik.ReadByte();
      if (w>0)             Console.Write((char)w);
    } while (w > 0);
    plik.Close();        Console.WriteLine("\n\n");
  }
}
```

31 Strumienie znakowe

Szczególnym przypadkiem strumieni, które są nieco łatwiejsze w użyciu są strumienie znakowe. Do strumieni znakowych nie trzeba przesyłać kolejnych bajtów, ale można od razu wysyłać/odbierać całe łańcuchy znaków. Strumienie znakowe dzielimy na strumienie zapisu i strumienie odczytu. Deklaracja strumieni znakowych ma postać:
Strumień zapisu:

```
StreamWriter nazwa = new StreamWriter(Filestream plik);
```

Strumień odczytu:

```
StreamReader nazwa = new StreamReader(FileStream plik);
```

32 Użycie strumieni znakowych

```
using System;
using System.IO;
class Pokaz
{
  public static void Main()
  {
    FileStream plik;
    string odczyt;
    try
    {
      plik = new FileStream("c:\\tekst.txt", FileMode.Open);
    }
    catch
    {
      Console.WriteLine("BŁĄD Podczas zapisu do zapis.txt");
      return;
    }
    StreamReader p = new StreamReader(plik);
    odczyt = p.ReadLine();
    do
    {
      Console.WriteLine(odczyt);
      odczyt = p.ReadLine();
    } while (odczyt != null);
    p.Close();
    plik.Close(); Console.WriteLine("\n\n");
  }
}
```

33 Swobodny dostęp do pliku

W czasie programowania operacji na plikach zazwyczaj zapisujemy i odczytujemy pliki sekwencyjnie od początku do końca. Często zachodzi jednak potrzeba dostępu do dowolnego miejsca w pliku. Plik o dostępie swobodnym pozwala ustawić miejsce czytania/zapisu w dowolnym miejscu pliku. Korzystamy tu z metody Seek, której deklaracja wygląda następująco:

```
plik.Seek(long pozycja, SeekOrigin odniesienie);
```

gdzie pozycja jest liczbowym określeniem pozycji znaku, a SeekOrigin określa odniesienie względem jakiego ma być uznana pozycja znaku. Wartości, jakie może przyjmować parametr SeekOrigin:

- SeekOrigin.Begin - Początek pliku,
- SeekOrigin.Current - Aktualna pozycja miejsca zapisu/odczytu,
- SeekOrigin.End - Koniec pliku.

34 Delegaty

W języku C# występują elementy, z którymi nie spotykamy się w innych językach. Takimi konstrukcjami są np. delegaty. Delegaty są to specjalnego rodzaju obiekty, które przechowują referencje do metod, a nie same metody, jak w "zwykłych" obiektach. Podczas wywoływania delegaty, wywołujemy metodę, której referencja jest przechowywana w tym delegacie. Można powiedzieć, że delegata jest wskaźnikiem do metody. Używanie delegaty jest bardzo wygodne bo za każdym razem gdy wywoływana jest powiązana z nim metoda sprawdzany jest adres, a więc nie ma problemu ze złym przydziałem pamięci.
Aby móc przypisać metodę do delegaty, musi istnieć zgodność deklaracji metody i jej delegaty. Deklaracja delegaty:

```
delegate typ_zwracany nazwa_delegaty(parametry) ;
```

Delegatę deklarujemy zawsze po deklaracjach przestrzeni nazw, nie jest on składnikiem klasy.

35 Użycie delegat

```
using System;
delegate void delegata(int liczba);
class Klasa_a
{
  public int a;
  public Klasa_a(int a)
  {
    this.a = a;
  }
  public void metoda(int cyferka)
  {
    Console.WriteLine(cyferka a);
  }
}

class Klasa_b
{
  public static void Liczba(int c); {
   Console.WriteLine(c);
  }
}

class Pokaz
{
  public static void Main()
  {
    Klasa_a A = new Klasa_a(4);
    delegata d1 = new delegata(A.metoda);
    delegata d2 = new delegata(Klasa_b.Liczba);
    d1(3);
    d2(7);
  }
}
```

36 Zdarzenia

Programowanie zdarzeniowe polega na tym, że pewne metody są wykonywane wówczas, gdy nastąpi jakieś zdarzenie. W języku C# metody wiążemy ze zdarzeniami za pomocą delegat. Są one wywoływane, gdy wywołamy zdarzenie. Z jednym zdarzeniem może być powiązanych kilka metod, które działają jak łańcuch wywołań. Ogólna deklaracja zdarzeń:

```
dostęp event nazwa_delegaty nazwa_zdarzenia;
```

Dzięki temu, że korzystamy z delegat, mamy pewność, że ze zdarzeniem zostaną powiązane tylko te metody, które będą spełniać warunek deklaracji narzucony przez delegatę.

37 Użycie zdarzeń

```
using System;
delegate void tablicowy_uchwyt();
class A
{
  public void Metoda1()
  {
    Console.WriteLine("Została wywołana metoda: Metoda1() klasy A");
  }
}

class B
{
  public static void Metoda2()
  {
    Console.WriteLine("Została wywołana metoda: Metoda2() klasy B");
  }
}
```

38 Użycie zdarzeń

```
class Tablica
{
  public event tablicowy_uchwyt zmiana;
  int[] tab;
  public Tablica(int rozmiar)
  {
    tab = new int[rozmiar];
  }
  public int this[int indeks]
  {
    get { return tab[indeks]; }
    set
    {
      tab[indeks] = value;
      Wywolaj_zmiana();
    }
  }
```

39 Użycie zdarzeń

```
private void Wywolaj_zmiana()
{
   if (zmiana != null)
   {
      zmiana();
   }
}
```

40 Metody anonimowe

Metody anonimowe są jedną z charakterystycznych konstrukcji występujących w języku c#. Pozwalają one na przekazanie bloku kodu jako parametru delegata.
Metody anonimowe nie mają nazw, a są jedynie blokiem kodu do wykonania. W metodzie anonimowej nie trzeba określać zwracanego typu. Określa się go poprzez instrukcję return wewnątrz bloku kodu tej metody.

41 Metody anonimowe

Metody anonimowe deklarujemy z wykorzystaniem instancji delegata, ze słowem kluczowym delegate. Ale w delegacie możemy też wykonać metodę nazwaną.

```
using System;
namespace AnonymousMethod
{
    delegate void ChangeNumber(int n);
    class Program
    {
        static int number = 10;
        public static void AddNumber(int a)
        {
            number += a;
            Console.WriteLine("Metoda nazwana: {0}", number);
        }
        public static void MultiplyNumber(int m)
        {
            number *= m;
            Console.WriteLine("Metoda nazwana: {0}", number);
        }
        public static int GetNumber()
        {
            return number;
        }
        static void Main(string[] args)
        {
            // Tworzymy instancję delegata używając metody anonimowej
            ChangeNumber cn = delegate (int x)
            {
                Console.WriteLine("Metoda anonimowa: {0}", x);
            };
            // wywołanie delegata używając metody anonimowej
            cn(10);
            // a teraz inicjowanie delgata używając metody nazwanej
            cn = new ChangeNumber(AddNumber);
            // wywołanie delegata używając metody nazwanej
            cn(10);
            // inicjowanie delegata przy użyciu innej metody nazwanej
            cn = new ChangeNumber(MultiplyNumber);
            // wywołanie delegata
            cn(2);
            // Zwrócenie liczby po dokonanych zmianach
            Console.WriteLine("Rezultat: {0}", Program.GetNumber());
            Console.ReadKey();
        }
    }
}
```

42 Atrybuty

Atrybut to znacznik, który służy do przekazywania informacji do środowiska wykonawczego o zachowaniu różnych elementów, takich jak: klasy, metody, struktury, typy wyliczeniowe czy poszczególne podzespoły naszego programu. Te informacje to są metadane opisujące dany element programu. Atrybut umieszczamy w nawiasach kwadratowych przed klasą, metodą lub strukturą. Platforma .NET udostępnia dwa rodzaje atrybutów - atrybuty predefiniowane i atrybuty niestandardowe. Programista może też definiować swoje atrybuty, których następnie może używać. Można powiedzieć, że atrybuty w C# są w działaniu podobne do adnotacji z języka Java.

43 Atrybuty

Środowisko .NET dostarcza trzy predefiniowane atrybuty:

- AtributeUsage służy do definiowania własnych atrybutów i opisuje sposób ich późniejszego użycia
- Conditional służy do wskazania fragmentów kodu np. metod które mogą być kompilowane w zależności od wartości zdefiniowanego identyfikatora np. debug albo trace.
- Obsolete służy do wskazania starych wersji metod, które nie są już zalecane do użycia.

44 Atrybuty

.NET Framework pozwala na tworzenie własnych atrybutów, które mogą być używane do przechowywania różnych informacji i pobrane w trakcie wykonywania programu. Tworzenie i używanie własnych atrybutów przeprowadza się w czterech krokach:

1. deklaracja własnego atrybutu,
2. skonstruowanie atrybutu,
3. użycie atrybutu na docelowym elemencie programu,
4. dostęp do atrybutu przez mechanizm refleksji.

45 Atrybuty

Nowy atrybut powinien dziedziczyć z klasy System.Attribute.
Przykład utworzenia nowego atrybutu:

```
[AttributeUsage(AttributeTargets.Class |
    AttributeTargets.Constructor |
    AttributeTargets.Method |
    AttributeTargets.Field |
    AttributeTargets.Property,
    AllowMultiple = true)]
// Pozwalamy użyć atrybutu w konstruktorach, metodach, polach, właściwościach
// pozwalamy na dziedziczenie atrybutu
public class DebugInfo : Attribute
{
    private string developerName;
    private string lastReviewData;
    public string message;
    public DebugInfo(string dev, string d)
    {
        this.developerName = dev;
        this.lastReviewData = d;
    }
    // właściwości
    public string DeveloperName
    {
        get
        {
            return developerName;
        }
    }
    public string LastReviewData
    {
        get
        {
            return lastReviewData;
        }
    }
}
```

46 Atrybuty

Atrybutu możemy użyć poprzez wstawienie go bezpośrednio nad elementem docelowym. Dzięki użyciu przykładowego atrybutu otrzymamy nazwisko programisty i datę ostatniej modyfikacji kodu

```
[DebugInfo("Paweł", "27/11/2018", message = "Zły zwracany typ")]
[DebugInfo("Paweł", "29/11/2018", message = "Nieprzypisana wartość")]
class Rectangle
{
    protected double length;
    protected double width;
    public Rectangle(double l, double w)
    {
        length = l;
        width = w;
    }
    [DebugInfo("Paweł", "22/11/2018", message = "Zły zwracany typ")]
    public double GetArea()
    {
        return length * width;
    }
}
```

47 Refleksje

Refleksja to mechanizm, który daje dostęp do danych o aplikacji w czasie jej działania. Refleksje znajdują się w przestrzeni nazw System.Reflection
Możliwości refleksji

- ⋆ podgląd atrybutów w trakcie wykonywania programu;
- ⋆ sprawdzenie różnych typów danych w danej bibliotece oraz utworzenie ich instancji;
- ⋆ wykonanie tzw. późnego wiązania do metod i właściwości (późne wiązanie oznacza, że np. docelowa metoda jest poszukiwana w trakcie wykonywania programu.)
- ⋆ tworzenie nowych typów w trakcie wykonywania programu, a następnie wykonywanie różnych zadań przy użyciu tych typów.

48 Refleksje

Przykład użycia refleksji:

```
using System;
namespace Refleksja
{
    class Program
    {
        static void Main(string[] args)
        {
            System.Reflection.MemberInfo info = typeof(MyClassToGetAttributeInfo);
            // pobranie listy atrybutów
            object[] attributes = info.GetCustomAttributes(true);
            for (int i = 0; i < attributes.Length; i++)
            {
                // Wypisujemy wszystkie atrybuty
                Console.WriteLine(attributes[i]);
                // Dodatkowo uzyskamy dostęp do opisu naszego atrybutu
                ExampleAttribute ea = (ExampleAttribute)attributes[i];
                Console.WriteLine("Info: {0}", ea.message);
            }
            Console.ReadKey();
        }
    }

    [AttributeUsage(AttributeTargets.All)]
    public class ExampleAttribute : Attribute
    {
        public readonly string message;
        private string topic;
        public ExampleAttribute(string Message)
        {
            this.message = Message;
        }
        public string Topic
        {
            get
            {
                return topic;
            }
            set
            {
                topic = value;
            }
        }
    }
    [ExampleAttribute("Informacja o mojej klasie")]
    class MyClassToGetAttributeInfo
    {
    }
}
```