.NET
Klasy, obiekty

ciąg dalszy

Przeciążanie operatorów 1

- W języku C# istnieje możliwość zdefiniowania funkcjonalności dużej części operatorów dla typów stworzonych przez użytkownika. Dzięki takiemu zabiegowi, działając na obiektach możemy używać czytelnego zapisu operatorowego.

- Zamiast definiować dla klasy *Liczba* metodę dodawania o nazwie *Dodaj*, możemy przeciążyć operator '+'.

- Przeciążanie operatora polega na zdefiniowaniu statycznej metody z użyciem słowa kluczowego ***operator.***

Przeciążanie operatorów - składnia

- Składnia definicji przeciążonego operatora może przyjąć jedną z dostępnych składni:

```
public static zwr_typ operator operator_unarny(typ_argumentu argument)
public static zwr_typ operator operator_binarny(typ_argumentul argument1,
                                                typ_argumentu2 argument2)
```

Przeciążanie operatorów - składnia

```
public static zwr_typ operator operator_unarny(typ_argumentu argument)
public static zwr_typ operator operator_binarny(typ_argumentu1 argument1,
                                                typ_argumentu2 argument2)
```

- ***zwr_typ***
  typ wyniku zwracany przez operator (wymagane);

- ***operator_unarny***
  jeden z dozwolonych operatorów unarnych: +, -, !, ~, ++, --**,**
  (wymagane);

- ***operator_binarny***
  jeden z dozwolonych operatorów binarnych:
  +, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, <=, >= ;(wymagane);
- ***typ_argumentu***
  typ argumentu wejściowego (wymagane);

Przeciążanie operatorów – składnia cd

```
public static zwr_typ operator operator_unarny(typ_argumentu argument)
public static zwr_typ operator operator_binarny(typ_argumentul argument1,
                                                typ_argumentu2 argument2)
```

- ***typ_argumentu***
  typ argumentu wejściowego (wymagane);
- ***typ_argumentu 1***
  typ pierwszego argumentu wejściowego (wymagane);
- ***typ_argumentu2***
  typ drugiego argumentu wejściowego (wymagane);
- ***argument, argument1, argutment2***
  nazwa argumentu (wymagane);

Przeciążanie operatorów – przykład 1

```
class Liczba
{
 private int Wartosc;

 public Liczba(int Wartosc)
 {
  this.Wartosc = Wartosc;
 }

 public static Liczba operator+(Liczba l1, Liczba l2)
 {
  return new Liczba(l1.Wartosc + l2.Wartosc); // zwracamy obiekt więc new
 }
}
```

Przeciążanie operatorów – przykład 1 wywołanie

```
Liczba l1 = new Liczba(100),  // inicjacja
       l2 = new Liczba(200),
       l3;                    // zmienna na wynik
l3 = l1 + l2;                 // dodanie wartosci obu typów
```

Przeciążanie operatorów cd.

- Nie wolno przeciążać operatorów:

- logicznych warunkowych: ***&&***, || (są rozwijane przy pomocy: ***&***, |, ***true*** oraz ***false***)
- konwersji: ***()*** (służą do tego celu składnie ze słowami kluczowymi: ***implicit*** i ***explicit***)
- indeksacji: ***[]*** (służą do tego celu indeksatory)
- przypisań: +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>= (są rozwijane automatycznie)
- oraz: =, ., ***?:***, ***new***, ***is***, ***sizeof***, ***typeof***

Przeciążanie operatorów relacji

- Operatory relacji muszą być przeciążane parami.

- Jeżeli przeciążymy operator == musimy również przeciążyć operator !=.
- To samo dotyczy pary <= oraz >= jak i pary < oraz >.

Przeciążanie operatorów relacji cd. 1

- Najlepszą techniką definiowania operatorów, jest stworzenie metody, która zbada relacje między obiektami i zwróci odpowiednią wartość. Taką metodę możemy następnie użyć w wyrażeniu w czasie definiowania par operatorów:

Przeciążanie operatorów relacji - przykład

```
class Liczba
{
 private int Wartosc;

 public Liczba(int Wartosc)
 {
  this.Wartosc = Wartosc;
 }
```

Przeciążanie operatorów relacji – przykład cd1

```
// metoda pomocnicza (dzięki niej unikamy redundancji)
private int Porownaj(Liczba Inna)
 {
  if (Wartosc < Inna.Wartosc)
   return -1;                    // mniejsza

  if (Wartosc > Inna.Wartosc)
   return 1;                     // większa

  return 0;                      // równe
 }
public static bool operator==(Liczba l1, Liczba l2)
{
 // gdy będą równe, metoda Porównaj zwróci wartość 0, warunek 0 == 0 da true
 return l1.Porownaj(l2) ==0;
}
```

Przeciążanie operatorów relacji – przykład cd2

```
public static bool operator!=(Liczba l1, Liczba l2)
{
 // gdy będą różne, metoda Porównaj zwróci wartość różną od 0 (1 lub -1),
 // warunek (1 lub -1) != 0 da true
 return l1.Porownaj(l2) != 0;
}
   public static bool operator<(Liczba l1, Liczba l2)
   {
    return l1.Porownaj(l2) < 0;
   }
    public static bool operator>(Liczba l1, Liczba l2)
    {
     return l1.Porownaj(l2) > 0;
    }
```

Indeksatory

- Indeksatory są mechanizmem pozwalającym na indeksowanie obiektu w taki sam sposób, w jaki indeksuje się tablice.

- W składni indeksatory przypominają właściwości z tą jednak różnicą, że zamiast nazwy, umieszcza się tu słowo kluczowe ***this*** oraz argument określający indeks w kwadratowych nawiasach.

- Składnia wygląda następująco:

```
 [modyfikatory] typ this[argumenty_indeksów]
{
 deklaracja_dostępu
}
```

Indeksatory - składnia

```
[modyfikatory] typ this[argumenty_indeksów]
{
 deklaracja_dostępu
}
```

- ***Modyfikatory***
  dopuszczalne są modyfikatory: ***new, virtual, sealed, override, abstract, extern*** oraz modyfikatory dostępu (opcjonalne);

- ***typ***
  typ danych elementu zwracanego przez indeksator (wymagane);

- ***argumenty_indeksów***
  lista argumentów indeksów, w formacie typ_indeksu nazwa_indeksu oddzielone przecinkami, nie dopuszcza się argumentów przekazywanych przez referencję i wyjście (wymagany co najmniej jeden argument indeksu);

- ***deklaracja_dostępu***
  blok ***get*** lub/i ***set,***
  dla bloku ***get*** wymaga się zwrócenia wartości elementu za pomocą ***return***,
  dla bloku ***set*** wartość przypisania dla elementu przechowywana jest w ***value***
  (wymagany co najmniej jeden z bloków).

Indeksatory - przykład

```
class CiagLiczb
{
 private int[] lista;

public CiagLiczb(int Dlugosc)
{
 lista = new int[Dlugosc];
}
```

```
public int this[int index]  // indeksator
{
 get
 {
  return lista[index]; // zwrócenie elementu tablicy
 }

 set
 {
  lista[index]=value; // przypisanie elementowi wartości
 }
}
}
```

Indeksatory - przykład

```
CiagLiczb ciag = new CiagLiczb(100);
ciag[0] = 1;
```

Indeksatory cd.

- Indeksatory używają takiej samej notacji jak tablice, ale w odróżnieniu od tablic mogą używać indeksów różnych typów (nie tylko typu całkowitego jak tablice).

Delegacje

- Delegacja jest typem referencyjnym, który pełni rolę bezpiecznego „wskaźnika do funkcji", dzięki któremu można stworzyć kod pozwalający na dynamiczną zmianę wołanych metod.

- Dzięki delegatom metody mogą być przekazywane jako argumenty.

- Do definiowania delegacji stosuje się słowo kluczowe ***delegate.***

Delegacje cd.1

Składnia delegacji wygląda następująco:

[modyfikatory] delegate typ Identyfikator([lista_argumentów]);

- ***modyfikatory***
  dozwolone są modyfikatory: *new* oraz modyfikatory dostępu (opcjonalne);

- ***typ***
  typ danych zwracany przez metodę, na którą wskazuje delegacja (wymagane);

- ***Identyfikator***
  nazwa delegacji musi być różna od nazwy klasy, w której ją zdefiniowano (wymagane);

- ***lista _argamentów***
  lista argumentów przekazywanych do metody, na którą wskazuje delegacja (opcjonalne).

Delegacje - przykład

```
class Metody
{
 // deklaracja delegacji dla metody
 private delegate void Metoda();

 private void DrukujText()
 {
  Console.WriteLine("Text");
 }

 private void DrukujLiczbe()
 {
  Console.WriteLine("Liczba");
 }
 private void Drukuj(Metoda metoda)
 {
  metoda(); // w czasie definicji nie
 }

 public void Wywolaj()
 {
  Drukuj(new Metoda(DrukujLiczbe));
  Drukuj(new Metoda(DrukujText));
 }
}
```

Zdarzenia

- W języku C# każdy obiekt może publikować zestaw zdarzeń, które następnie mogą być subskrybowane.

- Klasa, w której definiowane jest zdarzenie nazywamy ***klasą publikującą***, a wszystkie inne klasy, które zostały poinformowane o tym zdarzeniu są ***klasami subskrybującymi***.

  Łącznikiem – informatorem pomiędzy klasą publikującą a subskrybującą są ***delegaty***.

Zdarzenia

- Zdarzenia pozwalają obiektom na monitorowanie zmian zachodzących w innych obiektach.

- Wykorzystują one model wydawcy i czytelnika.
  - Wydawcą jest obiekt, w którym zachodzą pewne zmiany.
  - Czytelnikiem jest obiekt, który jest zainteresowany zmianami zachodzącymi u wydawcy.
  - W momencie, w którym u wydawcy zajdzie jakaś zmiana, zainteresowani czytelnicy (ci, którzy zgłoszą zainteresowanie zmianami) zostaną powiadomieni o jej wystąpieniu.
  - Zdarzenie jest wiadomością wysyłaną przez obiekt wydawcy do obiektu czytelnika, na którą czytelnik może zareagować wykonując jakąś czynność.

Zdarzenia cd.

- Zdarzenia wykorzystują delegacje do wywoływania metod w obiektach czytelników.

- Informacja o zmianie występującej u wydawcy, może spowodować wywołanie wielu delegacji. Nie ma jednak możliwości kontrolowania kolejności ich wywołania.

Zdarzenia - składnia

[modyfikatory] event typ nazwa_zdarzenia;

- ***modyfikatory***
  dozwolone są modyfikatory: ***abstract, new, override, static, virtual, extern*** oraz modyfikatory dostępu (opcjonalne);

- ***typ***
  delegacja, z którą ma być skojarzone zdarzenie (wymagane);

- ***nazwa _zdarzenia***
  nazwa zdarzenia (wymagane).

Zdarzenia przykład cz.1

```
using System;


class Delegacje
{
 // delegacja potrzebna do deklaracji zdarzenia
 public delegate void MetodaObslugi(int Parametr);


 public class Wydawca
 {
  // zdarzenie ukazania się książki "Piotruś pan"
  public event MetodaObslugi wydanie_ksiazki_Piotrus_Pan;
  // zdarzenie ukazania się książki "Czerwony kapturek"
  public event MetodaObslugi wydanie_ksiazki_Czerwony_Kapturek;
```

Zdarzenia przykład cz.2

```
// metoda wysłania informacji o ukazaniu się książek i ich cenie
public void WyslijInformacje()
{
 // jeżeli ktoś jest zainteresowany zdarzeniem ukazania się książki "Piotruś pan"
 // dostanie informację na ten temat
 if (wydanie_ksiazki_Piotrus_Pan != null)
  wydanie_ksiazki_Piotrus_Pan(120);

 // jeżeli ktoś jest zainteresowany zdarzeniem ukazania się książki
 // "Czerwony kapturek" dostanie informację na ten temat
 if (wydanie_ksiazki_Czerwony_Kapturek != null)
  wydanie_ksiazki_Czerwony_Kapturek(12);
 }
}
```

Zdarzenia przykład cz.3

```
public class Czytelnik
{
 // metoda określająca reakcję czytelnika
 public void Kupuje(int Cena)
 {
  if (Cena <= 100)
   Console.WriteLine("Kupuję");
  else
   Console.WriteLine("Proszę o rabat");
 }
}
```

Zdarzenia przykład cz.4

```
static void Main(string[] args)
{
 Wydawca wyd = new Wydawca();
 Czytelnik czyt = new Czytelnik();
 // czytelnik zgłasza chęć kupna książki "Piotruś pan" kiedy tylko się ukaże
 // pod warunkiem, że jej cena nie będzie większa od 100, jeżeli będzie,
 // poprosi o rabat
 wyd.wydanie_ksiazki_Piotrus_Pan += new MetodaObslugi(czyt.Kupuje);
 // wydawca wydaje obie książki i wysyła informacje o ich ukazaniu się, co spowoduje
 // reakcję czytelnika (czytelnik zareaguje jedynie na informacje o ukazaniu się
 // książki "Piotruś pan", gdyż nie zgłaszał chęci kupna innej książki)
 wyd.WyslijInformacje();
}
}
```

Dziedziczenie

- Dziedziczenie jest jednym z podstawowych mechanizmów obiektowości, który pozwala obiektowi przejąć (odziedziczyć) cechy (dane oraz funkcjonalność) od innego obiektu.

- Obiekt dziedziczący nazywa się **obiektem potomnym** lub obiektem **dziedziczącym**, natomiast obiekt, z którego się dziedziczy nazywa się **obiektem rodzica** lub obiektem **bazowym**.

- Dzięki mechanizmowi dziedziczenia można tworzyć nowe klasy w oparciu o istniejące, bez potrzeby ponownego pisania tej samej funkcjonalności. Klasa dziedzicząca stanowi w takim wypadku rozszerzenie klasy bazowej.

Dziedziczenie cd. 1

- W języku C# istnieje możliwość dziedziczenia tylko z jednej klasy bazowej (oraz wielu interfejsów).

- Aby wskazać, że nowa klasa dziedziczy z jakiejś klasy bazowej, w definicji nowej klasy należy podać nazwę klasy bazowej po znaku dwukropka:

```
class NowaKlasa : KlasaBazowa
{
   ....
}
```

Dziedziczenie cd. 2

- Klasa potomna nie może być bardziej dostępna od klasy rodzica (nie może mieć mniej restrykcyjnego modyfikatora dostępu)

```
class Przykład
{
 public class Bazowa1
  {
   ...
  }
protected class Dziedziczaca1 : Bazowa1
{
 ...
}
```

```
 private class Bazowa2
 {
  ...
 }

// BŁĄD: może być tylko private
public class Dziedziczaca3 : Bazowa2
{
 ...
}
```

Dostęp do składowych klasy bazowej

- Klasa potomna dziedziczy z klasy rodzica wszystkie elementy poza konstruktorami i destruktorami.
  Oznacza to że:
  - składowe publiczne ***(public)*** klasy rodzica stają się składowymi publicznymi klasy potomnej,
  - składowe chronione ***(protected)*** klasy rodzica stają się dostępne dla klasy potomnej,
  - składowe prywatne ***(private)*** klasy rodzica są dostępne tylko w klasie rodzica.

Wywoływanie bazowych konstruktorów

- Domyślnie w momencie wywołania konstruktora klasy potomnej następuje odwołanie do bezargumentowego konstruktora klasy bazowej.

- Można to jednak zmienić wskazując konkretny konstruktor klasy bazowej za pomocą słowa kluczowego ***base,*** umieszczonego w liście inicjującej konstruktora klasy potomnej.

Wywoływanie bazowych konstruktorów - przykład

```
class Bazowa
{
 public Bazowa()
 {
  ...
 }

 protected Bazowa(int Arg)
 {
  ...
 }
}
```

```
class Dziedziczaca : Bazowa
{
 public Dziedziczaca() // domyślnie :base()
 {
  ...
 }

 public Dziedziczaca(int Arg) : base(Arg)
 {
  ...
 }
}
```

Wywoływanie bazowych konstruktorów

- Odwołując się do konstruktora klasy bazowej, należy mieć na względzie jego dostępność.
- Jeżeli konstruktor w klasie bazowej będzie konstruktorem prywatnym, jego wywołanie nie powiedzie się. To samo dotyczy domyślnego odwołania, jeżeli sami definiujemy konstruktor bezargumentowy.

Słowo kluczowe ***base***

```
class Bazowa
{
 protected int liczba;
}

class Dziedziczaca : Bazowa
{
 public void Przelicz(int liczba)
 {
  base.liczba = liczba;
 }
}
```

- Słowo kluczowe ***base*** można również wykorzystać do odwołania się do składowych klasy bazowej (np.: gdy nazwy argumentów pokrywają się z nazwami pól klasy bazowej):

Przesłanianie metod Polimorfizm

- Przesłanianie metod zwane również polimorfizmem, jest mechanizmem, który pozwala na zmianę definicji metody z klasy bazowej w klasie pochodnej w taki sposób, iż w czasie wywołania nastąpi odwołanie do metody z właściwej klasy.

- Oznacza to, że w przypadku, w którym nastąpi rzutowanie z klasy pochodnej na klasę rodzica, nastąpi wywołanie metody przesłaniającej (znajdującej się w klasie pochodnej), a nie metody przesłanianej (znajdującej się w klasie rodzica).

Przesłanianie metod cd.1

- Do definiowania metod, które mogą być przesłaniane polimorficznie w klasach dziedziczących, służy modyfikator ***virtual.*** Metody te określa się mianem metod wirtualnych.

```
class Bazowa
{
 public virtual void MetodaPrzeslaniana()
 {
   ...
 }
}
```

Przesłanianie metod cd.2

- W czasie definiowania metod przesłaniających, należy pamiętać o tym, że:
  - muszą zawierać ciało (nawet gdy niczego nie implementujemy, należy umieścić pusty blok instrukcji)

```
public virtual MetodaPrzeslaniana();    // BŁĄD
public virtual MetodaPrzeslaniana()
{
}
```

Przesłanianie metod - klasa dziedzicząca

- nie mogą być definiowane jako statyczne (polimorfizm dotyczy obiektu nie klasy)

```
public static virtual MetodaPrzeslaniana()  // BŁĄD
{
  ...
}
```

- nie mogą być definiowane jako prywatne (nie będzie możliwości ich przesłonienia w klasie potomnej):

```
private virtual MetodaPrzeslaniana()  // BŁĄD
{
  ...
}
```

Przesłanianie metod - klasa dziedzicząca cd. 1

- W klasie dziedziczącej można przesłonić metodę wirtualną używając modyfikatora ***override.*** Metody te określa się mianem metody przesłaniającej lub kolejnej metody wirtualnej.

- Pojęcie kolejnej metody wirtualnej oznacza, że metoda przesłaniająca automatycznie staje się metodą wirtualną w klasie potomnej i może być przesłonięta w innej klasie, która będzie z niej dziedziczyć.

Przesłanianie metod
klasa dziedzicząca - przykład

```
class Bazowa
{
 // metoda wirtualna
 public virtual void MetodaPrzeslaniana()
 {
  ...
 }
}

class Dziedziczaca : Bazowa
{
 // metoda przesłaniająca metodę MetodaPrzeslaniana
 public override void MetodaPrzeslaniana()
 {
 }
}
```

```
class KolejnaDziedziczaca : Dziedziczaca
{
 // metoda przesłaniająca metodę MetodaPrz
 public override void MetodaPrzeslaniana()
 {
  ...
 }
}
```

Przesłanianie metod klasa dziedzicząca

- W czasie definiowania metod przesłaniających, należy pamiętać o tym, że:
  - muszą zawierać ciało (nawet gdy niczego nie implementujemy, należy umieścić pusty blok instrukcji),
  - muszą mieć składnię identyczną ze składnią metod wirtualnych z klasy bazowej,
  - nie można używać modyfikatora ***virtual,*** ponieważ modyfikator ***override*** oznacza, że metoda jest kolejną metodą wirtualną,
  - nie mogą być definiowane jako statyczne (polimorfizm dotyczy obiektu nie klasy),
  - nie mogą być definiowane jako prywatne (nie będzie możliwości ich przesłonienia w klasie potomnej).

Ukrywanie metod

- Metody dziedziczone oprócz tego, że mogą być przysłaniane, mogą również być ukrywane.
- Ukrycie metody oznacza zastąpienie metody odziedziczonej z klasy bazowej zupełnie inną metodą w klasie dziedziczącej.
- Do ukrywania metod służy słowo kluczowe ***new.***
- Jeżeli w klasie potomnej definiujemy metodę o takiej samej nazwie co w klasie bazowej, powinniśmy użyć słowa kluczowego ***new,*** aby uniknąć ostrzeżenia o pokrywaniu się nazw.

Ukrywanie metod - przykład

```
class Bazowa
{
 public void Metoda()
  {
  }
}
class Dziedziczaca : Bazowa
{
 new public void Metoda()
  {
  }
}
```

Ukrywanie metod – przesłanianie metod

- Mechanizm ukrywania może być używany zarówno w przypadku wirtualnych jak i niewirtualnych metod.

- Stosowanie mechanizmu ukrywania w przypadku metod wirtualnych niesie za sobą pewne komplikacje w stosunku do mechanizmu polimorfizmu. Ukrycie metody wirtualnej z klasy bazowej w klasie dziedziczącej spowoduje, że nie będzie ona traktowana jako kolejna metoda wirtualna tylko jako pierwsza metoda wirtualna w łańcuchu dziedziczenia.

Ukrywanie metod – przesłanianie metod - przykład

```
class Bazowa
{
 public virtual void Metoda()
 {
 }
}
class Dziedziczaca : Bazowa
{
 // ukrywamy metodę z klasy Bazowa
 // pierwszą funkcją wirtualną
 new public virtual void Metoda()
 {
 }
}
```

```
class KolejnaDziedziczaca : Dziedziczaca
{
 // przesłaniamy metodę z klasy Dziedziczac
 public override void Metoda()
 {
 }
}
```

Klasy ostateczne

- Czasami istnieje potrzeba zdefiniowania klasy ostatecznej (nierozszerzalnej), czyli takiej, z której nie można już dziedziczyć.

- Służy do tego modyfikator ***sealed,*** który umieszcza się przed słowem ***class***

Klasy ostateczne - przykład

```
public sealed class Ostateczna
{
}

// BŁĄD: nie można odziedziczyć
public class JakasPotomna : Ostateczna
{
}
```

Metody ostateczne

- Modyfikator ***seled*** może również być wykorzystany w mechanizmie przysłaniana do określenia, iż kolejna metoda wirtualna jest ostateczną metodą, której nie można już przysłaniać

Metody ostateczne - przykład

```
class Bazowa
{
 // metoda wirtualna
 public virtual void MetodaPrzeslaniana()
 {
 }
}

class Dziedziczaca : Bazowa
{
 // metoda przesłaniająca metodę MetodaPrzeslaniana
 public sealed override void MetodaPrzeslaniana()
 {
 }
}
```

```
class KolejnaDziedziczaca : Dziedziczaca
{
 // BŁĄD: nie można przesłonić ostatecznej metody
 public override void MetodaPrzeslaniana()
 {
 }
}
```

Klasy abstrakcyjne

- Klasy abstrakcyjne używane są do definiowania elementów, które będą implementowane w klasach potomnych.

- Do zdefiniowania klasy abstrakcyjnej służy modyfikator ***abstract,*** który umieszcza się przed słowem kluczowym ***class***.

```
abstract class KlasaAbstrakcyjna
{
}
```

Klasy abstrakcyjne cd. 1

Klasy abstrakcyjne różnią się od innych klas tym, że:

- nie można utworzyć instancji klasy abstrakcyjnej.

```
KlasaAbstrakcyjna kabs = new KlasaAbstrakcyjna(); // BŁĄD
```

- można utworzyć metodę abstrakcyjną w klasie abstrakcyjnej

```
abstract class KlasaAbstrakcyjna
{
 public abstract void Metoda();
}

class KlasaNieAbstrakcyjna
{
 // BŁĄD: metoda nie może być abstrakcyjna
 public abstract void Metoda();
}
```

Klasy abstrakcyjne cd. 2

- Klasy abstrakcyjne stosuje się do definiowania grupy cech wspólnych dla klas potomnych.

```
abstract class Przedmiot
{
 protected float waga;  //
}

class Kubek : Przedmiot
{
 public Kubek()
 {
  base.waga = 0.25f;
 }
}
```

Metody abstrakcyjne

- W każdej klasie abstrakcyjnej można zdefiniować metodę abstrakcyjną.

- Metoda abstrakcyjna jest metodą, która musi być zaimplementowana w klasie potomnej.

- Definiuje się je umieszczając modyfikator ***abstract*** przed typem zwracanej wartości

```
public abstract void Metoda();
```

Metody abstrakcyjne cd.2

- Metody abstrakcyjne są metodami wirtualnymi, z tym jednak wyjątkiem, że w klasie abstrakcyjnej nie mogą być implementowane (implementuje się je w klasie potomnej).

```
abstract classKlasaAbstrakcyjna
{
 // BŁĄD: nie może mieć ciała
 public abstract void Metoda()
 {
   ...
 }
}
```

Metody abstrakcyjne cd.3

- do implementacji metod abstrakcyjnych w klasie potomnej wykorzystuje się modyfikator ***override***

```
class KlasaDziedziczaca : KlasaAbstrakcyjna
{
 public override void Metoda()
  {
    ...
  }
}
```

Bazowa klasa System.Object

- Jeżeli w czasie definiowania nowej klasy, nie określimy żadnej klasy bazowej, domyślnie zostanie przyjęte, że nowa klasa dziedziczy z ***System.Object*** (która jest klasą bazową dla wszystkich innych klas).

Klasa ***System.Object*** składa z następujących elementów:

- publicznego bezparametrowego konstruktora ***Object*** tworzącego nową instancję klasy ***Object***,
- metod publicznych:
  - ***Equals -*** określającej czy dwa obiekty są równe,
  - ***GetHashCode*** - dostarczającej mechanizm haszowania charakterystyczny dla danego obiektu, który używany jest w algorytmach haszujących,
  - ***GetType -*** zwracającej typ aktualnej instancji klasy,
  - ***ToString -*** zwracającej napis reprezentujący aktualną instancję klasy.
  - ***ReferenceEquals -*** określającej czy instancje klasy są sobie równe.
  - ***Finalize -*** pozwalającej obiektowi na zwalnianie zasobów i wykonywanie czynności deinicjujących przed zwolnieniem pamięci zajmowanej przez obiekt (mechanizm dostępny za pomocą destruktora).
  - ***MemberwiseClone -*** tworzącej kopię aktualnego obiektu.

To już jest koniec ...