Opracował: L. Jung

Deklarowanie nowych typów danych

Klasy dynamiczne umożliwiają deklarowanie nowych typów danych dla ułatwienia wykonywania obliczeń (np. w rachunku macierzowym, czy rachunku liczb zespolonych).

Obliczenia macierzowe

Macierze są ważnym narzędziem algebry, które powstało na przełomie XIX i XX wieku, i szybko zaczęły być stosowane przez fizyków, techników i informatyków w wielu obliczeniach.

Prostokątną tablicę o *m* wierszach i *n* kolumnach złożoną z *m·x n* elementów zbioru **R** nazywamy macierzą:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

gdzie:

m - liczba wierszy macierzy

n - liczba kolumn macierzy

$a_{ij} \in \mathbf{R}$ - elementy macierzy A.

Wymiarem macierzy nazywamy iloczyn liczby jej wierszy i kolumn, co zapisuje się następująco:

$$m \times n$$

Macierz **A** o wymiarach $m \times n$ *oznaczamy następująco:*

$$\mathbf{A} = [a_{ij}]_{m \times n}$$

Zwyczajowo, macierze oznaczamy dużymi literami alfabetu łacińskiego: A, B, C,...

Macierze ułatwiają rozwiązywanie układów równań liniowych o wielu niewiadomych.:

Układ równań liniowych:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \dots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \dots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & \ddots & & \vdots & & \vdots \\ a_{m1}x_1 & + & a_{m2}x_2 & + & \dots & + & a_{mn}x_n & = & b_m \end{array}$$

Równanie wyrażające zadanie do rozwiązania:

$$A * X = B$$

mnożymy lewostronnie przez macierz odwrotną A:

$$A^{-1}$$

i otrzymujemy równanie postaci:

$$A^{-1} * A * X = A^{-1} * B$$

Korzystając z równości (właściwości):

$$A^{-1} * A = I$$

gdzie I jest macierzą jednostkową (ma 1 na głównej przekątnej)

zapisujemy równanie (wzór) dla wyznaczenie wektora niewiadomych X, które przyjmie postać:

$$X = A^{-1} * B$$

Chcemy zdefiniować nowy typ danych Macierz, który ułatwi rozwiązywanie zadań w klasycznych rachunku macierzowym.

Definiowanie nowych typów danych

Potocznie przez typ danych rozumiemy:

zbiór wartości (reprezentowany przez dany typ danych)

+

zbiór dozwolonych działań

Na przykład, typ danych int:

- reprezentuje 32-bitową liczbę całkowitą (z zakresu wartości: -2,147,483,648 do 2,147,483,647) zapisywaną w 4 bajtowej strukturze danych,
- udostępnia operacje arytmetyczne (+, -, *, /, . . .,++, −−), bitowe i logiczne (& - and, | - or, ^ - xor) oraz relacyjne (<, >, ...)

Dla zdefiniowania nowego typ danych (w języku C#) wybieramy klasę (class) lub strukturę (struct)

- klasa (class) jest typem referencyjnym obiektów (łączących listę pól składowych, listę metod i właściwości):

```
public class NazwaReferencyjnegoTypuDanych
   { .   .   .    }
```

których egzemplarze mają przydzieloną pamięć na kopcu (heap), zwanym też stertą,

- struktura (struct) jest typem wartościowym (również łączących listę pól składowych, listę metod i właściwości)

```
public struct NazwaWartościowegoTypuDanych
   { .   .   .   }
```

dla których pamięć przydzielana jest na stosie (stack)

Różnice między klasą a strukturą:

- klasy są typem referencyjnym a struktury typem wartościowym,
- klasa obsługuje mechanizmu dziedziczenia i pamięć przydzielana jest na stercie programu,
- struktura nie obsługuje mechanizmu dziedziczenia i pamięć dla wartości struktury jest przydzielana na stosie programu,
- struktury nie mogą mieć domyślnego konstruktora, a klasy mogą,
- struktury stosowane są do reprezentowania prostych typów danych, takich jak liczby czy wartości logiczne,
- klasy są stosowane do reprezentowania złożonych typów danych, takich jak tablice czy listy lub drzewa,
- struktury są bardziej efektywne (wydajne) niż klasy, ponieważ są przechowywane na stosie zamiast na stercie.

Deklaracja typu danych: Macierz

Dla obliczeń na macierzach wybieramy klasę (class), która jest typem referencyjnym.

Projektujemy klasę Macierz, w której deklarujemy:

- prywatną tablicę dwuwymiarową dla przechowywania w niej wartości elementów macierzy:

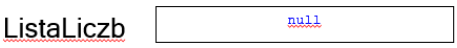
```
class Macierz
{   // deklaracja zmiennej prywatnych klasy Macierz
    private float[,] macierz;
```

Typ tablicowy jest typem referencyjnym, co oznacza, że sama deklaracja zmiennej tablicowej nie tworzy egzemplarza tablicy, czyli nie jest pobierany blok pamięci na przechowywanie elementów tablicy, a powoduje tylko rezerwację pamięci na przechowanie referencji (adresu) do egzemplarza tablicy.

Na przykład, deklaracja tablicy jednowymiarowej (czyli wektora) ListaLiczb:

```
Float[] ListaLiczb;
```

spowoduje tylko rezerwację pamięci na referencję (adres) do bloku pamięci reprezentującego egzemplarza tablicy:



gdzie null oznacza pustą referencję.

- konstruktora klasy Macierz:

```
public Macierz(ushort liczbaWierczy, ushort liczbaKolumn)
{
    macierz = new float[liczbaWierczy, liczbaKolumn];
}
```

zadaniem konstruktora klasy macierz jest utworzenie egzemplarza tablicy macierz.

Egzemplarz tablicy tworzymy przy użyciu operator new, po którym zapisujemy nazwę typu elementu tablicy, a w nawiasach kwadratowych podajemy liczbę elementów danego wymiaru tablicy.

- właściwości umożliwiających odczytywanie rozmiarów utworzonego egzemplarza klasy Macierz:

```
public ushort LiczbaWierszy
  {
    get { return (ushort)macierz.GetLength(0); }
  }

 public ushort LiczbaKolumn
  {
    get { return (ushort)macierz.GetLength(1); }
  }
```

Tablice w języku C# są strukturami samo opisującymi się, co oznacza, że podając nazwę zmiennej tablicowej i operator selekcji: . (kropkę), możemy wybrać odpowiedni atrybut opisujący egzemplarz tablicy. Na przykład, dla określenia liczby elementów tablicy możemy pobrać (odczytać!) wartość atrybutu Length:

ListaLiczb.Length

W przypadku tablic wielowymiarowych, liczbę elementów każdego z wymiarów określamy przy użyciu metody:

TablicaWielowymiarowa.GetLength(*NumerWymiaru*)

- indeksatora, który umożliwi indeksowanie (uzyskanie dostępu do) egzemplarza klasy (w sposób analogiczny do indeksowania tablic: Tablica[2, 3]):

```
public float this[ushort NrWiersza, ushort NrKolumny]
 {
   set  {if ((NrWiersza >= 0) && (NrWiersza < macierz.GetLength(0))
             && (NrKolumny >= 0) && (NrKolumny < macierz.GetLength(1)))
          macierz[NrWiersza, NrKolumny] = value;
        else // BŁĄD: "WYRZUCAMY" WYJĄTEK Z SYGNALIZACJĄ BŁĘDU
             throw new IndexOutOfRangeException("ERROR:
                         wartość jednego z indeksów " +
           "macierzy wykracza poza dozwolony zakres");
        }
```


```
get  { if ((NrWiersza >= 0) && (NrWiersza < macierz.GetLength(0)) &&
           (NrKolumny >= 0) && (NrKolumny < macierz.GetLength(1)))
         return macierz[NrWiersza, NrKolumny];
       else // BŁĄD: "WYRZUCAMY" WYJĄTEK Z SYGNALIZACJĄ BŁĘDU
        throw new IndexOutOfRangeException("ERROR: wartość
                  jednego z indeksów macierzy wykracza
                 poza dozwolony zakres");
       }
}
```

Ogólny szablon deklaracji indeksatora

[modyfikator dostępu] typDanej this[lista parametrów formalnych]
{ get{// zwracanie wartości wskazanej przez indeks}
  set{/* przypisanie, przekazanej wartości, do pozycji wskazanej
       przez indeks */}
}

- parametry formalne indeksatora nie muszą być liczbami całkowitymi (np. mogą być typu `string`),
- mogą być przeciążone,
- muszą zawierać co najmniej jedną deklarację dostępu **get** lub **set**,
- mogą być zdefiniowane z wieloma parametrami a każdy z parametrów może być innego typu,
- deklaracja konstruktora może występować zarówno w klasie (`class`), jak i strukturze (`struct`) oraz interfejsie (`interface`).

Deklaracje działań klasy Macierz
(przeciążanie operatorów arytmetycznych i relacyjnych)

W projektowanej klasie Macierz, przeciążamy definicje znanych operatorów arytmetycznych i relacyjnych dla działań obiektach typu Macierz.

Przeciążanie operatorów oznacza przypisywanie im nowego znaczenia (w stosunku do aktualnego ich znaczenia dla działań na typach danych: `int`, `float`, czy `double`), tak, aby można je również stosować w rachunku macierzowym:

D = A + B * C

F = D * B + C

E = F - D + A

Przeciążanie operatorów w języku C#

Przeciążane mogą być zarówno operatory działań arytmetycznych, logicznych, relacyjnych jak i metody, w tym również konstruktory klas.

Przeciążanie operatorów (operator overloading) działań w językach programowania (nie tylko w C#, ale również w języku C++ czy Java) oznacza przypisywanie znanym operatorom działań arytmetycznych (np. +, _, *, ?, . . .), logicznych (&, |, ^) i relacyjnych (==, !=, <, >, <=, >=), nowych działań.

Przy czym, operatory relacyjne (==, !=, <, >, <=, >=): muszą być przeciążane w parach: == i !=, < i >i >=<= .

Przeciążanie jest ważnym mechanizmem języków programowania zorientowanego obiektowo, gdyż umożliwia tworzenie wielu metod o takiej samej nazwie, ale różniących się ilością lub typem dozwolonych argumentów.

Jakie operatory mogą być przeciążane w języku C#?

- operatory unarne (jednoargumentowe):

  +, -, !, ~, ++, --, true, false

- operatory binarne (dwuargumentowe):

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

Operatory, których nie wolno przeciążać:

- logiczne: &&, ||,
- konwersji: (),
- indeksacji: [],
- operatorów przypisań: +=, -=, *=, %=, <<=, >>=, |=, ^=, &=, /=,
- pozostałe: =, ., new, is, sizeof, typeof, ?:.

Przeciążania operatorów: ogólne zalecenia

- przeciążanie operatorów stosujemy dla implementowanych (definiowanych) nowych typów danych, w których ma sens stosowanie operacji arytmetycznych, logicznych czy relacyjnych,
- przeciążanym operatorów powinno być nadawane znaczenia zgodne z naturalnym (intuicyjnym) ich znaczeniu,
- unikamy przeciążania operatorów dla typów referencyjnych, gdy może to budzić wątpliwości, czy nowe znaczenie operatora dotyczy referencji, czy do samego obiektu (egzemplarza) „wskazywanych" przez daną referencję,
- dobrze przemyślane stosowanie przeciążanie operatorów dla nowych typów danych umożliwia skrócenie zapisu obliczeń (kodu) i zwiększenie czytelności zapisu algorytmów obliczeń.

Szablon syntaktyczny przeciążanie operatorów w języku C#

```
public static TypWyniku operator SymbolOperatora
                                (Lista parametrów formalnych
    {
      // treść definicyjna operatora
    }
```

Przy czym, **przynajmniej jeden parametr formalny przeciążanego operatora musi być obiektem klasy**, dla której ten operator jest przeciążany (definiowany)!

Przeciążanie dwuargumentowego operatora '+' (w klasie Macierz)

Dwuargumentowy operator '+' ma umożliwić obliczenie sumy (dodania) dwóch macierzy!

Jeśli $\mathbf{A} = [a_{ij}]_{m\times n}$ i $\mathbf{B} = [b_{ij}]_{m\times n}$, to sumę (dodanie) dwóch macierzy wyznaczamy następująco:

$$\mathbf{A} + \mathbf{B} = [a_{ij} + b_{ij}]_{m\times n}$$

Warunek wykonalności operatora '+':

Dodawanie macierzy jest wykonywalne, gdy liczba wierszy jednej macierzy jest równa liczbie wierszy drugiej macierzy i liczba kolumn jednej macierzy jest równa liczbie kolumn drugiej macierzy.

Zapis w języku C# przeciążanie dwuargumentowego operatora '+' (w klasie Macierz)

```
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;
}
```

Przeciążanie dwuargumentowego operatora '-' (w klasie Macierz)

Dwuargumentowy operator '-' ma umożliwić obliczenie różnicy (odejmowania) dwóch macierzy!

Jeśli $\mathbf{A} = [a_{ij}]_{m\times n}$ i $\mathbf{B} = [b_{ij}]_{m\times n}$, to różnicę odejmowanie) dwóch macierzy wyznaczamy następująco:

$$\mathbf{A} - \mathbf{B} = [a_{ij} - b_{ij}]_{m\times n}$$

Warunek wykonalności operatora '-':

Odejmowanie macierzy jest wykonywalne, gdy liczba wierszy jednej macierzy jest równa liczbie wierszy drugiej macierzy i liczba kolumn jednej macierzy jest równa liczbie kolumn drugiej macierzy.

Zapis w języku C# przeciążanie dwuargumentowego operatora '-' (w klasie Macierz)

```
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;
 }
```

Przeciążanie dwuargumentowego operatora '*' (w klasie Macierz)

Dwuargumentowy operator '*' ma umożliwić obliczenie iloczynu (mnożenia) dwóch macierzy!

Jeśli $\mathbf{A} = [a_{ij}]_{m\times k}$ i $\mathbf{B} = [b_{ij}]_{k\times n}$, to iloczynem dwóch macierzy A oraz B, nazywamy macierz $\mathbf{C} = \mathbf{A} * \mathbf{B} = [cij]_{m\times n}$

gdzie:

$$cij = \sum_{p=1}^{k} a_{ip} b_{pj}$$

Warunek wykonalności operatora '*':

Mnożenie macierzy jest wykonywalne, gdy liczba kolumn w pierwszej macierzy A jest równa liczbie wierszy w drugiej macierzy B.

Zapis w języku C# przeciążanie dwuargumentowego operatora '*' (w klasie Macierz)

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

       }
     }
    return c;
}
```

Przeciążanie dwuargumentowego operatora '==' dla obliczania równości macierzy (w klasie `Macierz`)

Dwuargumentowy operator '==' ma umożliwić obliczenie równości dwóch macierzy!

Macierze A i B są równe (co zapisujemy w matematyce tak: **A = B**), gdy mają jednakowe wymiary i odpowiednie elementy są równe.

Warunek wykonalności operatora '==':

Wynikiem operacji równości jest:

- `true`, gdy obie macierze A i B mają jednakowe wymiary i odpowiednie ich elementy są równe,
- `false`, gdy macierze A i B mają różne wymiary lub gdy mają równe wymiary, ale odpowiednie elementy (co najmniej jeden) są różne od siebie,

Zapis w języku C# przeciążanie dwuargumentowego operatora '==' dla obliczania równości macierzy (w klasie `Macierz`)

```
public static bool operator ==(Macierz a, Macierz b)
        { if (a.LiczbaWierszy == b.LiczbaWierszy &&
                a.LiczbaKolumn == b.LiczbaKolumn)
            { for (ushort i = 0; i < a.LiczbaWierszy; i++)
                 for (ushort j = 0; j < b.LiczbaKolumn; j++)
                      if (a.macierz[i, j] != b.macierz[i, j])
                            return false;
              return true;
            }
          else
            return false;
        }
```

Zapis w języku C# przeciążanie dwuargumentowego operatora '!=' dla obliczania różności macierzy (w klasie `Macierz`)

```
public static bool operator !=(Macierz a, Macierz b)
        {
            return !(a == b);
        }
```

Nadpisanie metod Equals i GetHashCode

W przypadku przeciążania operatora równości '==' oraz '!=' należy również nadpisać deklarację (definicję) metody Equals (. . .) oraz metody GetHashCode(), które są dziedziczone z klasy System.Object.

Nadpisanie metody Equals (. . .):

```
public override bool Equals(object obj)
 {/* metoda Equals służy do porównania stanu wskazanej w argumencie
     instancji obiektu ze stanem obiektu bieżącego: this  */
  if (obj is null || !(obj is Macierz))
  // operator is umożliwia sprawdzenie czy 'obj' jest typu Macierz
     return false;
   Macierz m = (Macierz)obj; /* rzutujemy parametr (argument) "obj"
                                        na typ Macierz */

  // sprawdzenie zgodności wymiarów macierzy
  if (this.macierz.GetLength(0) == (m.LiczbaWierszy)
        && (this.macierz.GetLength(1) == (m.LiczbaKolumn)))
   { // badanie relacji równości na elementach macierzy
     for (ushort i = 0; i < m.LiczbaWierszy; i++)
      {
        for (ushort j = 0; j < m.LiczbaKolumn; j++)
          if (this.macierz[i, j] != m.macierz[i, j])
             return false;
      }
      /* wszystkie elementy egzemplarzy macierzy: 'this'
         oraz 'm' są równe */
     return true;
   }
  else
    return false;
```

Metoda **Equals()** musi działać tak, aby jej wynik odpowiadał znaczeniu operatora "==" .

```
UWAGA: Instrukcję:
                    Macierz m = (Macierz)obj;
  /*rzutujemy parametr (argument) 'obj'  na typ Macierz */
      możemy zapisać również tak:
                    Macierz m = obj as Macierz;
/* Operator 'as' jawnie konwertuje 'obj' (wyrażenie) na
dany typ wartości referencyjnej Macierz (lub na wartość
null). Jeśli konwersja nie jest  możliwa, operator zwraca
wartość null */
```

Nadpisanie metody GetHashCode()

- Metoda **GetHashCode()** zwraca wartość liczbową, służąca do identyfikacji obiektu podczas wykonywania relacji równości.
- Jeśli metoda **Equals()** zwraca wartość:
  - true, to metoda **GetHashCode()** zwraca taką samą wartość dla obydwu obiektów,
  - false, to metoda **GetHashCode()** zwraca różne wartości dla porównywanych obiektów, co oznacza, że porównywane obiekty mają co najmniej jeden element o różnej wartości.

Nadpisanie metody GetHashCode():

```
public override int GetHashCode()
 {/* zadaniem tej metody jest wygenerowanie 32-bitowej liczby całkowitej,
      której wartość będzie identyfikatorem obiektu    */
    return this.macierz.GetHashCode();
  }
```

W nadpisaniu metody GetHashCode() została wywołana metoda:

```
this.macierz.GetHashCode()
```

która zwróci tzw. hash code (typu int) obiektu 'macierz', który jest wyznaczany przez środowisko CLR (przy pierwszym wywołaniu metody GetHashCode), który zapisuje utworzony hash code w polu SyncBlockIndex obiektu 'macierz'.

Metoda GetHashCode udostępnia kod skrótu dla algorytmów, które wymagają szybkiego sprawdzenia równości obiektów.

Dwa obiekty, które są równe, zwracają kody skrótów, które są równe.

Jednak odwrotność nie jest prawdziwa: równe kody skrótu nie oznaczają równości obiektów, ponieważ różne (nierówne) obiekty mogą mieć identyczne kody skrótu.

Deklaracja operatora konwersji Macierz

```
public static explicit operator Macierz(float x)
    { // deklaracja i utworzenie egzemplarza macierzy pomocniczej dla zwrotu wyniku konwersji
        Macierz c = new Macierz(1, 1);
        // wpisanie wartości parametru (liczby) 'x' do macierzy 'c'
        c.macierz[0, 0] = x;
      // zwrócenie wyniku, którym jest utworzona nowa, jednoelementowa Macierz
        return c;
        // Użycie metody konwersji: Macierz YY = (Macierz)23;
     }
```

Przeciążanie operatorów konwersji

Można przeciążać operator konwersji w postaci jawnej (**explicit**) oraz w postaci niejawnej (**implicit**). Aby przeciążyć **operator niejawny** używamy słowa kluczowego **implicit**, natomiast aby przeciążyć **operator jawny**, używamy słowa kluczowego **explicit** w definicji operatora.

Deklarowanie zmiennych klasy Macierz

Utworzona klasa Macierz jest klasą dynamiczną, która umożliwia deklarowanie zmiennych referencyjnych (do egzemplarzy klasy Macierz) analogicznie do deklaracji zmiennych referencyjnych tablic, czyli następująco:

```
NazwaKlasy NazwaZmiennejReferencyjneKlasy;
```

Na przykład, deklarację dwóch zmiennych referencyjnych klasy Macierz zapisujemy następująco:

```
Macierz A, B;
```

Zadeklarowanym zmiennym referencyjnym A i B klasy Macierz zostaje przypisana domyślnie wartość null.

Tworzenie egzemplarzy klasy `Macierz`

Egzemplarze (obiekty) klas dynamicznych tworzymy przy użyciu operatora `new`:

1. dla już zadeklarowanych zmiennych referencyjnych klasy (co zapisujemy następująco):

   NazwaZmiennejReferencyjnejKlasy = `new NazwaKlasy ( . . . );`

   lista parametrów aktualnych dla konstruktora klasy

   Przykład (dla zadeklarowanych zmiennych referencyjnych: A i B):

```
A = new Macierz(3, 5);
B = new Macierz(5, 9);
```

Jeżeli konstruktor klasy:

- ma parametry formalne, to przy tworzeniu instancji (egzemplarza) klasy, po nazwie klasy podajemy listę parametrów aktualnych (odpowiadających, co do liczby, kolejności i typowi danych, liście parametrów formalnych jednego z jej konstruktorów),
- nie ma parametrów formalnych, to przy tworzeniu instancji (egzemplarza) klasy, po nazwie klasy zapisujemy tylko nawiasy (puste!): `()`.

Po utworzeniu egzemplarza (obiektu) klasy dynamicznej i jego zainicjowaniu możemy już odwoływać się do zadeklarowanych i udostępnianych (modyfikatorem `public`) metod, właściwości i operatorów przeciążonych:


```
. . .

C = A + B;

. . .

D = A * B;

. . .

If (A == B)

{

. . .

}
```

Deklaracja metod rozszerzeń klasy Macierz

Przeciążone operatory arytmetyczne, logiczne i relacyjne w klasie Macierz, umożliwiają programowanie obliczeń na obiektach typu Macierz, ale nie obejmują wykonywanie takich działań jak:

- inicjowanie wartości elementów obiektu typu Macierz,
- przepisywania wartości elementów z kontrolki DataGridView do obiektu typu Macierz i odwrotnie,
- wypisywania wartości elementów obiektu typu Macierz na konsolę i odwrotnie,
- . . .

Język C# umożliwia jednak uzupełnianie deklaracji klasy o statyczne metody rozszerzeń, które zapisujemy w klasie statycznej (zapisywanej poza klasą główną projektowanego programu):

```
static class MetodyRozszerzeńKlasy
{ . . . }
```

W klasie rozszerzeń deklarujemy metody rozszerzeń dla określonej klasy (opisującego typ danych):

```
static class MetodyRozszerzeńKlasy
{  . . .
   public static void NazwaMetody
    (this NazwaRozszerzanejKlasy NazwaParametruFormalnego,
                                          inne parametry formalne metody)
    { . . . }
  . . .
}
```

W liście parametrów formalnych deklarowanej metody rozszerzeń, deklaracja pierwszego parametru formalnego musi być poprzedzona autoreferencją `this` „wskazującą" typ danych (klasę), który będzie rozszerzony o deklarowaną metodę. I pierwszy parametr formalny będzie zmienną referencyjną do tego rozszerzanego typu danych (klasy).

Przykład, deklaracji `metod rozszerzeń` klasy `Macierz` (dla aplikacji formularzowej)

```
static class MetodyRozszerzeńKlasyMacierz
 { public static void LosoweInicjowanieMacierzy
                  (this Macierz X, double d, double g)
     {  /* wpisanie do  Macierzy X wartości elementów generowanych losowo
           z przedziału [d, g] */
        Random rnd = new Random();   //rnd - Generator liczb losowych
        for (ushort i = 0; i < X.LiczbaWierszy; i++)
          for (ushort j = 0; j < X.LiczbaKolumn; j++)
              X[i, j] = (float) (rnd.NextDouble() *
                                                (g - d) + d);
     }
```


```
public static void PrzepiszDataGridViewDoMacierzy
                                  (this Macierz X, DataGridView dgvMacierzX)
{ for (ushort i = 0; i < X.LiczbaWierszy; i++)
      for (ushort j = 0; j < X.LiczbaKolumn; j++)
          X[i, j] =
                   float.Parse((dgvMacierzX.Rows[i].Cells[j].Value).ToString());
}

public static void PrzepiszMacierzDoKontrolkiDataGridView
                                       (this Macierz X, DataGridView dgvMacierzX)
{  for (ushort i = 0; i < X.LiczbaWierszy; i++)
     for (ushort j = 0; j < X.LiczbaKolumn; j++)
        dgvMacierzX.Rows[i].Cells[j].Value = string.Format("{0:F2}", X[i, j]);
}
} // od static class MetodyRozszerzeńKlasyMacierz
```

Metody rozszerzeń:

- umożliwiają „dodawanie” nowych metod do istniejących typów (klas) bez konieczności tworzenia nowego typu potomnego (lub modyfikowania istniejącego typu w inny sposób) i ponownej kompilacji programu,
- stanowią specjalny rodzaj metod statycznych, ale są wywoływane tak, jakby były metodami zdefiniowanymi w rozszerzanej klasie typu danych (na przykład, w klasie Macierz).

Utworzenie egzemplarzy klasy Macierz

```
// .    .       .
 // utworzenie egzemplarzy macierzy: A, B i C
 A = new Macierz((ushort)dgvMacierzA.Rows.Count,
                                   (ushort)dgvMacierzA.Columns.Count);
 B = new Macierz((ushort)dgvMacierzB.Rows.Count,
                                   (ushort)dgvMacierzB.Columns.Count);
  C = new Macierz((ushort)dgvMacierzC.Rows.Count,
                                   (ushort)dgvMacierzC.Columns.Count);
```

Przykład użycia metod rozszerzających klasę Macierz (dla aplikacji formularzowej)

```
// przepisanie elementów z kontrolki DataGridView: dgvMacierzA  do macierzy A
    A.PrzepiszDataGridViewDoMacierzy(dgvMacierzA); // wywołanie metody rozszerzeń
// przepisanie elementów z kontrolki DataGridView: dgvMacierzB  do macierzy B
    B.PrzepiszDataGridViewDoMacierzy(dgvMacierzB); // wywołanie metody rozszerzeń
```

lub (bez pobierania wartości elementów macierzy z kontrolek DataGridView

```
// deklaracja przedziału wartości elementów macierzy
 const float DolnaGranicaPrzedziału = 10.0F;
 const float GórnaGranicaPrzedziału = 100.0F;


// inicjowanie macierzy  A: wywołanie metody rozszerzeń
A.LosoweInicjowanieMacierzy(DolnaGranicaPrzedziału,
                            GórnaGranicaPrzedziału);
// wpisanie (przepisanie) elementów macierzy A do kontrolki DataGridView:
   A.PrzepiszMacierzDoKontrolkiDataGridView(dgvMacierzA); // wywołanie
                                                          metody rozszerzeń
// inicjowanie macierzy  B: wywołanie metody rozszerzeń
B.LosoweInicjowanieMacierzy(DolnaGranicaPrzedziału,
                            GórnaGranicaPrzedziału);
// wpisanie (przepisanie) elementów macierzy B do kontrolki DataGridView:
   B.PrzepiszMacierzDoKontrolkiDataGridView(dgvMacierzB); // wywołanie
                                                          metody rozszerzeń
```