

## Metodyki programowania

Istnieją trzy główne metodyki programowania:

- Programowanie strukturalne
- Programowanie proceduralne
- Programowanie obiektowe

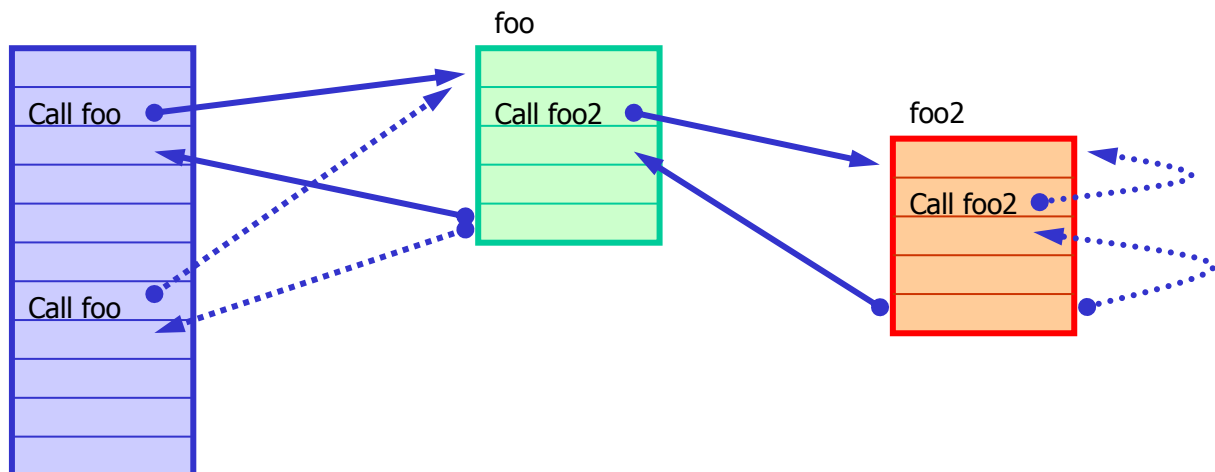
Zastosowanie "wyższych" metodyk programowania wiąże się zawsze z dodatkowym nakładem pracy, jednak przynosi znaczne korzyści – przy bardziej złożonych programach ten nakład pracy bardzo szybko się zwraca

DRY (ang. don't repeat yourself)

- Dodatkowa korzyść z programowania proceduralnego oraz obiektowego to możliwość wielokrotnego użycia kodu.
- Postulat DRY zaleca unikanie powtórzeń kodu (np. przez Ctrl-C, Ctrl-V), do czego właśnie służą wyższe metodyki: fragment kodu można "zamknąć" w postaci oddzielnej funkcji lub klasy i odwołać się do niego, zamiast go kopiować.
- Co ważne, DRY zmniejsza też liczbę błędów (zatem redukuje czas potrzebny na ich poprawianie) – jeżeli funkcja zawiera błędy, to trzeba je poprawić tylko w jednym miejscu, zaś raz opracowana i pozbawiona błędów funkcja już zawsze będzie działać poprawnie
- DRY ułatwia również utrzymanie kodu – jeżeli jakiś aspekt programu trzeba ulepszyć (np. zastosować szybszy algorytm), to wystarczy to zrobić w jednym miejscu
- Skrajne podejście DRY widać np. w języku Java, gdzie zaleca się stosowanie zasady jeden plik = jedna klasa (publiczna) – w ten sposób każda klasa może łatwo być użyta wielokrotnie
- Postulat DRY dotyczy też innych aspektów programowania, np. użycia narzędzi programistycznych, kompilacji warunkowej, a nawet definiowania stałych

## Podprogram

Podprogram to wydzielony fragment kodu, "leżący" poza programem głównym, który można **wywołać** – następuje wówczas skok do podprogramu, jego wykonanie, po czym tzw. skok powrotny, tj. powrót do programu głównego, dokładnie w to samo miejsce, z którego podprogram został wywołany. Mechanizm ten jest na tyle uniwersalny, że podprogram może wywołać inny podprogram, a nawet samego siebie (rekurencja), a powrót zawsze jest wykonywany bezbłędnie



Stosowanie podprogramów służy przede wszystkim dwóm celom:

- Wielokrotne użycie kodu  
Każda funkcja to spełnienie postulatu DRY – napisz raz, używaj wiele razy
- Dekompozycja  
Bardziej złożone zadania można rozbić na prostsze części, każda w postaci funkcji

Zalecenia DRY dotyczące funkcji:

- Czynność, która powtarza się choćby dwa razy, należy przenieść do funkcji
- Czynność, mającą więcej niż 20 linii kodu, należy podzielić na funkcje
- Czynność, która wymaga użycia instrukcji zagnieżdżonych 3 razy (np. for-for-if), dobrze jest robić na funkcje

## ***Metody statyczne i dynamiczne***

Metody (tj. funkcje będące elementami klasy) mogą być deklarowane jako statyczne bądź (domyślnie) dynamiczne:

- Metody **statyczne** mogą być wywoływane wyłącznie poprzez klasę (nie można wywołać metody statycznej przez obiekt danej klasy)
- Metody dynamiczne – odwrotnie niż statyczne – mogą być wywołane wyłącznie przez obiekt klasy

```
Int32 x, y = 7;  
String s;
```

```
x = Int32.Parse("13");    // ok., metoda statyczna  
x.Parse("13");            // błąd
```

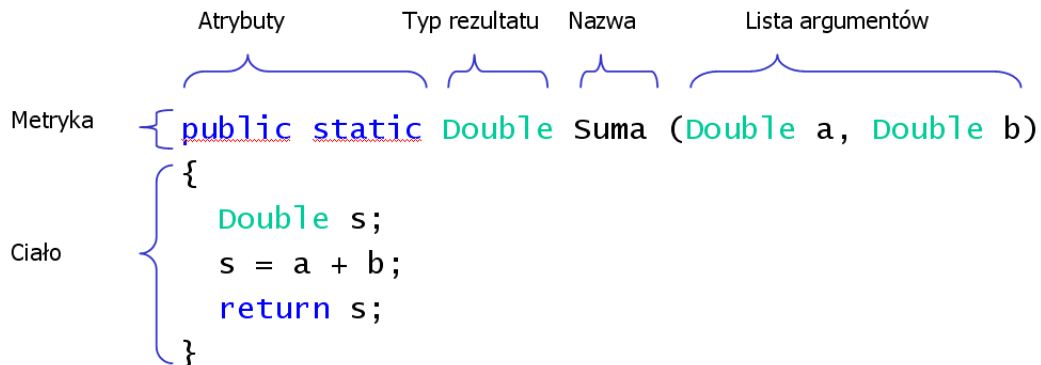
```
s = y.ToString();        // ok., metoda dynamiczna  
s = Int32.ToString(7);    // błąd
```

- Niektóre nowsze języki programowania (w tym m.in. Java, C#) wymuszają metodykę programowania obiektowego – nie można np. tworzyć funkcji, a wyłącznie metody – w ogóle wszystko, co tworzy się w C# jest, z zasady, częścią jakiejś klasy
- Jeżeli potrzebne są "funkcje", to grupuje się je w specjalnych klasach narzędziowych (ang. Utility), gdzie są deklarowane jako statyczne.

Klasa Math jest typowym przykładem takiej klasy – są w niej zawarte funkcje matematyczne, a klasa Math do niczego więcej nie służy. Podobnych klas w C# jest więcej, np. klasa File do operacjach na plikach albo Directory do operacji na katalogach. Klasy narzędziowe zawierają wyłącznie metody statyczne, a w konsekwencji nie da się utworzyć obiektu takiej klasy

## Definicja funkcji

- Definicja funkcji musi być umieszczona wewnątrz klasy  
Można wykorzystać klasę Program (w której jest funkcja Main) albo stworzyć własną; można też utworzyć nowy plik i tam utworzyć klasę (polecenie Project – Add class)
- Definicja składa się z metryki funkcji i ciała (treści) funkcji:



- Atrybuty**  
Oprócz atrybutu *static* można podać modyfikator dostępu:
  - *public* – funkcja będzie dostępna w dowolnej innej części programu
  - *private* – funkcja prywatna, dostępna tylko w klasie, gdzie jest zdefiniowanaDomyślnym modyfikatorem dostępu jest *private*
- Typ rezultatu**  
Dowolny typ języka C# albo słowo kluczowe *void* (ang. nic, brak) jeżeli funkcja nie dostarcza rezultatu (odpowiednik procedury języka Pascal). Może to być zarówno typ wartościowy (a w tym typ prosty), jak i referencyjny (a w tym tablica lub obiekt)
- Nazwa**  
Obowiązują zwykłe zasady dla nazw.  
Dla funkcji zaleca się stosowanie konwencji Pascal i czasowników lub wyrażień czasownikowych (np. WriteLine, Parse, Compare), chociaż są wyjątki, np. funkcje matematyczne (Sqrt, Pow, Sin, ...)
- Lista argumentów** (ściślej: parametrów formalnych)  
Lista oddzielonych przecinkami argumentów funkcji, zawsze w postaci par typ nazwa. Lista argumentów może być pusta. Argumenty w ciele funkcji są traktowane jak zmienne lokalne – można dowolnie odczytywać (używać) i modyfikować ich wartości; Każda funkcja to odrębny blok – nazwy argumentów i zmiennych lokalnych mogą się powtarzać w wielu funkcjach
- Ciało funkcji**  
Blok instrukcji (zatem "{" oraz "}"), zawierający dowolne instrukcje oraz wykonujący zadania przewidziane dla funkcji;  
Jeżeli funkcja zwraca rezultat – tj. ma typ rezultatu inny niż *void* – to ciało funkcji musi zawierać instrukcję *return*  
Funkcje "bezrezultatowe" mogą zawierać instrukcję *return*, jednak nie jest to wymagane

## Instrukcja return

- Dla funkcji mających rezultat inny niż *void* instrukcja return służy głównie do dostarczenia rezultatu i ma postać:

`return wyrażenie;`

przy czym *wyrażenie* musi mieć typ rezultatu taki sam, jak typ rezultatu funkcji (albo typ automatycznie konwertowany do niego)

- Dla funkcji "bezrezultatowych" return nie może dostarczać rezultatu, należy stosować wariant:

`return;`

- Instrukcja return ma jeszcze drugie zadanie – natychmiastowe zakończenie wykonywania funkcji – może wystąpić w kodzie funkcji więcej niż raz; Z uwagi na fakt, że wykonanie return kończy funkcję, ma to sens jedynie w połączeniu z instrukcją warunkową, np.:

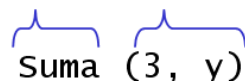
```
public static Int32 silnia(Int32 n)
{
    if (n <= 1)
        return 1;

    // obliczenia dla n>1 [...]
    return s;
}
```

## Wywołanie funkcji

Wywołanie funkcji wymaga podania nazwy i listy argumentów:

Nazwa    Lista argumentów



Lista argumentów (ściślej: parametry aktualne)

- Musi zawierać argumenty w liczbie zgodnej z metryką funkcji; Jeżeli lista argumentów jest pusta, to w wywołaniu należy użyć pustej listy (pustych nawiasów) – bez tego funkcja nie zostanie wywołana
- Typ argumentów musi być zgodny z metryką funkcji; Jako kolejne argumenty można podać dowolne wyrażenia, dla których typ rezultatu jest zgodny z typem argumentu funkcji lub jest automatycznie konwertowany;
- Wywołanie funkcji mającej rezultat **można** umieścić w dowolnym wyrażeniu lub instrukcji wyrażeniowej, gdzie ma typ identyczny z typem rezultatu funkcji

```
Int32 GetFive()
{
    return 5;
}
```

```
Int32 x = 13 / GetFive(); // 2 czy 2,6 ?
```

- Powyższe nie jest obowiązkowe, a dla funkcji bezrezultatowej zabronione

- Podczas wykonywania wyrażenia, najpierw zostanie wywołana funkcja (ma najwyższy priorytet ze wszystkich operatorów), a następnie wartość dostarczona jako rezultat funkcji zostanie użyta w wyrażeniu:

```
Double x = 13.0 + 2 * Math.Sqrt(9.0); // jak 13 + 2*3
```

Rezultatu funkcji (lub wyrażenia zawierającego taki rezultat) można użyć w **każdym** wyrażeniu C# - nie tylko w przypisaniu, ale też jako rozmiar tablicy, warunek pętli, inkrementacja pętli for itd. itp.

```
for (i=0; i < GetFive(); i++)           // jak i<5
Int32[] t = new Int32[ReadTabSize()];
do {
    // ...
} while (GetApproximationError()>errorTheshold);
```

- Argumenty są przekazywane do funkcji przez **wartość**:

```
Int32 Foo (Int32 x)
{
    x++;
    return x;
}

Int32 a = 7;
Foo (a);           // a=7 (a nie 8)
```

- Argumenty funkcji mogą mieć zdefiniowane **wartości domyślne**.  
Przy wywołaniu funkcji podawanie tych argumentów jest opcjonalne – jeżeli nie zostaną podane, brana jest wartość domyślna  
W metryce funkcji po takim argumentcie nie mogą wystąpić argumenty zwykłe, podobnie przy wywołaniu nie można pominąć jednego argumentu i podać wartość następnego

```
Int32 Foo (Int32 x, Int32 ix = 1, Int32 dx = 0)
{
    return x + ix - dx;
}

Int32 a = 7, b;
b = Foo(a);           // Foo(a, 1, 0)
b = Foo(a, 13);       // Foo(a, 13, 0), a nie Foo(a, 1, 13)
```

## **Przekazywanie argumentu p. referencję**

- Można napisać funkcję tak, aby przekazywać jej jako argument referencję do zmiennej – wówczas zmiana wartości argumentu jest tożsama zmianie wartości zmiennej, przekazanej do funkcji podczas jej wywołania – do tego celu służą atrybuty argumentów *ref* oraz *out*
- Argumenty *ref* służą do dwukierunkowej komunikacji z funkcją, przekazując wartość, która może być w funkcji zmodyfikowana;
- Przy wywołaniu funkcji z argumentem *ref* lub *out*, należy obowiązkowo powtórzyć ten atrybut – kod jest bardziej czytelny, nie ma wątpliwości że argument jest przekazywany przez referencję

- Argument **ref** jest referencją (inną nazwą tej samej zmiennej):

```
void Foo (ref Int32 x)
{
    x++;
}
```

```
Int32 a = 7;
Foo(ref a);           // a = 8 (a nie 7)
```

Każda zmiana x powoduje *de facto* zmianę wartości zmiennej, do której x jest referencją

- Przekazywanie do funkcji argumentów i dostarczanie przez funkcję rezultatu jest mechanizmem na tyle uniwersalnym, że w roli argumentów i rezultatu mogą też wystąpić zmienne typu referencyjnego – np. tablice lub obiekty klas  
W językach C/C++ przekazywanie tablicy jako argumentu lub rezultatu funkcji było kłopotliwe, ponieważ w tych językach tablica nie zawiera informacji o swoim rozmiarze; w C# ten problem nie występuje:

```
public static Foo (Int32[] t)
{
    Int32 len = t.Length;
    // ...
}
```

## **Funkcje rekurencyjne**

- Rekurencja polega na tym, że funkcja wywołuje siebie samą; Oczywiście wywołanie funkcji przez siebie samą musi nastąpić z inną wartością argumentu, a ciąg argumentów kolejnych wywołań musi być skończony, inaczej program wpadnie w nieskończoną pętlę:

```
void BadRecursion (Int32 a)
{
    BadRecursion(a);
}
```

- Istnieje szereg problemów, które można niemal równie łatwo rozwiązać za pomocą rekurencji, jak i bez niej, np. obliczanie silni, obliczanie NWD metodą Euklidesa:

```
NWD(A, B)
IF B=0
    return A
ELSE
    return NWD(B, A mod B)
```

- Wersja rekurencyjna wygląda bardzo elegancko (nie ma jawnej pętli), ale zużywa zasoby – należy stosować z rozważą.
- Są przypadki, kiedy rekurencja dramatycznie zwiększa złożoność – np. liczby Fibonacciego, algorytm z pętlą ma złożoność  $O(n)$ , rekurencja –  $O(2^n)$
- Są też przypadki, kiedy jest nieodzowna, np. algorytm quicksort bez rekurencji jest niemal niemożliwy do zrealizowania

## **Funkcje rozszerzające**

- Klasa, zgodnie z paradygmatem programowania obiektowego, jest zamknięta – jeżeli pochodzi z biblioteki, to nie można jej zmienić – nie można zmienić metod (funkcji) klasy ani dodać nowych;
- Można zdefiniować klasę potomną istniejącej klasy (choć są wyjątki, definicja klasy może zabraniać tworzenia klas potomnych) i dodać tam nowe metody albo nadpisać istniejące;  
Niestety nie zawsze można zastąpić klasę bazową przez klasę potomną
- Z drugiej strony – nie sposób przewidzieć komu i do czego klasa się przyda, więc nie można w klasie umieścić wszystkich możliwych metod (klasy powinny mieć ściśle określony i relatywnie wąski obszar odpowiedzialności)

➔ C# oferuje ciekawy mechanizm uzupełnienia definicji klas bez korzystania z mechanizmu dziedziczenia – funkcje rozszerzające (ang. extension functions)

- Funkcje rozszerzające muszą być definiowane w klasie statycznej (klasa Program, do której należy funkcja Main w aplikacjach konsolowych, NIE jest taką klasą)
- Funkcje rozszerzające muszą być definiowane jako statyczne; powinny też być publiczne, inaczej nie będzie jak z nich skorzystać poza klasą, w której są zdefiniowane
- Pierwszy argument funkcji rozszerzającej należy poprzedzić słowem kluczowym **this**; Funkcja staje się rozszerzeniem **typu argumentu**:

```
public static class DoubleUtils
{
    public static Double TheSame(this Double x)
    {
        return x;
    }
}
```

- Funkcje rozszerzające są nadal funkcjami statycznymi i można je wywoływać jak wszystkie inne funkcje statyczne: przez klasę, w której zostały zdefiniowane:

```
Double a = 13.0, b;
b = DoubleUtils.TheSame(a);
```

- Jeżeli funkcja rozszerzająca jest wywoływana jako rozszerzenie, to przy wywołaniu należy pominąć pierwszy argument – w jego miejsce wskazuje obiekt, na rzecz którego funkcja została wywołana:

```
Double a = 13.0, b;
b = a.TheSame();
```

## Zadania

Proszę napisać program, który...

- Wyświetla w zgrabnej tabelce wartości funkcji sinus i cosinus, dla kątów w zakresie od 0 do 90 stopni, co 10 stopni. Funkcje trygonometryczne w C# posługują się kątami wyrażonymi w radianach, należy zatem przeliczyć stopnie na radiany. Warianty:
  - Bezpośrednio, przeliczenie stopnie-radiany w wywołaniu funkcji
  - Z użyciem własnej funkcji stopnie-radiany, `Math.Sin(Tools.Deg2Rad(fi))`
  - Z użyciem własnej funkcji przyjmującej kąt w stopniach, `Tools.SinFmDeg(fi)`
  - Jak w c, ale funkcja zdefiniowana jako rozszerzająca, `fi.SinFmDeg()`
- Wyświetla w zgrabnej tabelce wartości logarytmów o podstawie podanej przez użytkownika, dla liczb z zakresu od 16 do 256, co 16. Warianty zmiany podstawy logarytmu:
  - Bezpośrednio,  $\log_{10}(x)/\log_{10}(p)$
  - Z użyciem własnej funkcji, `Tools.Log(x, p)`
  - Jak w b, ale funkcja zdefiniowana jako rozszerzająca, `x.Log(p)`
- Oblicza sumę dwóch liczb podanych przez użytkownika (nazwijmy je a i b), ale do obliczenia sumy używa własnej funkcji Suma. Następnie, nie zmieniając funkcji suma, tylko sposób jej wywołania, obliczyć:
  - $2a + 3b$
  - $ab + 7$
  - $\sqrt{a} + b^2$
  - $a + b + 1$  (nie wolno używać „+”!)
- Oblicza i wyświetla pierwiastki równania kwadratowego (wersja uproszczona, bierzemy pod uwagę tylko dwa przypadki, tj. delta większa lub równa zero – dwa pierwiastki, delta mniejsza niż zero – brak pierwiastków). W obliczeniach użyć własnych funkcji Suma, Różnica, Iloczyn i Iloraz zamiast dwuargumentowych operatorów arytmetycznych
- \* Wykonuje zadania z punktu 4, ale wykorzystując funkcje Suma, Różnica, Iloczyn i Iloraz jako funkcje rozszerzające
- Znajduje parę liczb względnie pierwszych (NWD=1) z zakresu liczb podanych przez użytkownika – wariant stochastyczny (obie liczby są losowane, do skutku). Do wyznaczania NWD należy zdefiniować własną funkcję
- \* Wykonuje zadania z punktu 6, ale funkcja NWD jest rekurencyjna
- \* Znajduje i wyświetla wszystkie pary liczb względnie pierwszych z zakresu liczb podanych przez użytkownika
- Wyświetla wartości silni dla liczb z zakresu podanego przez użytkownika. Do obliczenia silni należy zdefiniować funkcję
- Wykonuje zadania z punktu 8, ale funkcja obliczająca silnię jest rekurencyjna



11. \* Wykonuje zadania z punktu 8, ale funkcja obliczająca silnię jest rekurencyjna i użyta jako rozszerzająca: `n.Silnia()`
12. Oblicza i wyświetla liczby Fibonacciego, z zakresu od 1 do 50. Do obliczania liczb ciągu Fibonacciego zdefiniować funkcję rekurencyjną
13. \* Wykonuje zadania z punktu 10 i dodatkowo mierzy czas wykonania obliczeń, dla każdego elementu ciągu oddzielnie
14. \* Wykonuje zadania z punktu 10, ale wykorzystując algorytm bez rekurencji.
15. Oblicza i wyświetla **sumę** dwóch liczb podanych przez użytkownika. Do obliczania sumy należy użyć delegacji. Warianty:
  - a. Delegat w postaci zdefiniowanej wcześniej funkcji Suma
  - b. Delegat w postaci wyrażenia lambda
16. \* Oblicza i wyświetla **sumę oraz iloczyn** dwóch liczb podanych przez użytkownika. Do obliczeń należy użyć delegacji i wyrażenia lambda
17. Tworzy tablicę o rozmiarze podanym przez użytkownika, wypełnia ją wylosowanymi liczbami i sortuje, używając algorytmu gнома. Do oddzielnych funkcji należy przenieść (w tej kolejności):
  - a. Tworzenie tablicy i wypełnianie jej losowymi liczbami
  - b. Drukowanie tablicy
  - c. Sprawdzenie, czy tablica jest posortowana
  - d. Sortowanie
18. Wykonuje zadania z punktu 17, ale używa funkcji do dekompozycji algorytmu – należy zdefiniować jako prywatne funkcje odpowiedzialne za funkcjonalnie oddzielne części algorytmu gнома (uwaga na **ref!**)
  - a. Zamianę sąsiednich liczb miejscami
  - b. Wykonanie kroku w lewo, gdy  $t_i > t_{i+1}$
  - c. Wykonanie kroku w prawo, gdy  $t_i \leq t_{i+1}$
19. Wykonuje zadania z punktu 17, ale funkcja użyta jako rozszerzająca: `t.Gnom()`
20. \*\* Wykonuje zadania z punktu 17, ale sortuje według innego kryterium (np. nieparzyste przed parzystymi). Kryterium sortowania należy przekazać w delegacji
21. Oblicza całkę oznaczoną wybranej prostej funkcji (np. funkcji liniowej lub kwadratowej), w przedziale podanym przez użytkownika
22. \*\* Wykonuje zadania z punktu 21, ale funkcję podcałkową należy przekazać w delegacji