Część pierwsza

1. Wprowadzenie

Programowanie komputerowe jest w uproszczeniu zespołem czynności związanych z pisaniem interpretowalnych przez system obliczeniowy instrukcji (programu), prowadzącym do osiągnięcia określonego celu, lub innymi słowy – rozwiązania założonego problemu. Programowanie może być przeprowadzane przy użyciu różnych dostępnych języków i według różnych metodologii (paradygmatów).

Samo programowanie stanowi część znacznie szerszej dyscypliny jaką jest inżynieria oprogramowania, obejmująca wiele innych aspektów wytwarzania oprogramowania takich jak projektowanie, konserwacja czy kontrola jakości.

2. Zmienne i typy danych

Program składa się zasadniczo z szeregu instrukcji wykonujących określone operacje na danych . Dane przechowywane są w pamięci komputera w postaci tzw. *zmiennych* tj. przydzielonych obszarów pamięci posiadających przypisane *identyfikatory*, za pośrednictwem których program odwołuje się do ich zawartości. Z czysto logicznego punktu widzenia zmienna stanowi pewną nazwaną wielkość, która może się zmieniać w trakcie działania programu.

Z każdą zmienną jest ściśle powiązany *typ danych*, określający jakiego rodzaju wartości mogą w niej być przechowywane (np. czy zmienna przechowuje liczbę czy tekst; jeśli liczbę, to czy jest to liczba rzeczywista czy całkowita, jaki jest zakres przechowywanych danych itp.). W powszechnie używanych językach programowania wyróżnić możemy następujące główne typy danych (*typy proste*):

Typy danych	Nazwy typów w języku Pascal
Liczba całkowita *Typ zmiennych przechowujących liczby całkowite* *np. 13; -10; 25738*	**integer, longint** *(w zależności od zakresu)*
Liczba rzeczywista *Typ zmiennych przechowujących liczby rzeczywiste (zmiennoprzecinkowe)* *np. 123,45; 1,43e-13; -3,141593*	**real, single, double** *(w zależności od zakresu)*
Wartość logiczna *Typ zmiennych przechowujących wartości logiczne* *PRAWDA lub FAŁSZ*	**boolean**
Znak *Typ zmiennych przechowujących wartości znakowe* *np. 'a', '3', '@'*	**char**
Łańcuch znaków *Typ zmiennych przechowujących dane tekstowe (ciągi znaków),* *np. "Ala ma kota"*	**string, string[n]** *(w zależności od długości)*

W większości języków programowania dostępne są też następujące *typy złożone*, będące większymi strukturami danych składającymi się z wielu typów prostych:

Tablica *Struktura składająca się z wielu wartości określonego typu*	**array** [a1..b1,..,aN..bN] **of** typ_podstawowy;

prostego. Tablica może mieć zadaną liczbę wymiarów. Tablica jednowymiarowa odpowiada liście wartości, zaś dwuwymiarowa macierzy.	**array** [1..7] **of integer**; //[1,3,67,13,7,0,7] //(jednowymiarowa tablica wartości typu całkowitego o wymiarze 7) **array** [1..3,1..2] **of char**; //|d 7 _| //|a z c| //(dwuwymiarowa tablica wartości typu znakowego o wymiarze 3x2)

Rekord
Struktura przechowująca wiele wartości, z których każda może być różnego typu
	record
	pole1:typ_pola1;
	
	..
	
	poleN:typ_polaN;
	
	end;
	
	osoba=**record**
	imie:**string**;
	
	wiek:**integer**;
	
	end;
	
	//osoba: //nazwa rekordu
	
	// imię:typ_łańcuchowy //pola rekordu
	
	// wiek:typ_całkowity

Dalej

EITC/SE/CPF Wykład 2

Odpowiedziałeś poprawnie na 0 z 0 pytań

CZĘŚĆ PIERWSZA

3.2. Różne paradygmaty programowania: Programowanie proceduralne c.d.

Sposoby przekazywania parametrów do podprogramów (procedur i funkcji)

W ogólności procedury i funkcje nie powinny (w miarę możliwości) korzystać ze zmiennych globalnych, lecz pobierać i przekazywać wszystkie dane (czy też wskaźniki do nich) jako parametry wywołania. Rozróżniamy dwie metody przekazywania parametrów do funkcji:

- *przekazywanie przez wartość*

Przekazywanie przez wartość oznacza, że wewnątrz procedury tworzona jest lokalna kopia parametru (zmienna lokalna), której przypisywana jest wartość przekazanego parametru, i wszystkie przeprowadzane wewnątrz procedury operacje wykonywane są na tej zmiennej lokalnej.

```
procedure Do_kwadratu(a:integer);
// definicja procedury Do_kwadratu z deklaracją parametru a typu integer przekazywanego przez wartość
begin
  a:=a*a;   // pomnóż a przez siebie samą i zapisz wynik w a
  write(a); // wypisz wartość zmiennej a na ekranie
end;

// program główny
var b:integer;  // zadeklaruj zmienną b typu integer
begin
  b:=2; // przypisz zmiennej b wartość 2
  Do_kwadratu(b);   // wywołanie procedury Do_kwadratu z parametrem b
  write(b); // wypisz wartość zmiennej b na ekranie (wynik: 2)
end;
```

W powyższym przykładzie zdefiniowano procedurę `Do_kwadratu`, która podnosi przekazany jej parametr (metodą przekazywania przez wartość) do drugiej potęgi i wypisuje wynik na ekranie. Procedura ta wywoływana jest w programie głównym z podaniem zmiennej `b` w roli parametru. Oznacza to, że wewnątrz procedury parametr `a` (będący zmienną lokalną funkcji) uzyska **wartość** zmiennej `b`, czyli w tym wypadku `2`. Wartość zmiennej `a` jest następnie podnoszona do kwadratu dając wynik 4, który wyświetlany jest na ekranie w ostatniej linijce procedury. Można teraz zadać pytanie co się stało z wartością zmiennej zewnętrznej `b`, przekazanej jako parametr? Ponieważ parametr `a` został zadeklarowany jako przekazywany przez wartość, wszystkie operacje wewnątrz procedury wykonywane były na lokalnej zmiennej `a`, zaś wartość zmiennej `b` została nienaruszona i nadal jest równa 2.
Zasadniczą cechą przekazywania przez wartość jest możliwość podawania wartości stałej jako parametru wywołania procedury. W przypadku zdefiniowanej w powyższy sposób procedury możliwe jest na przykład następujące wywołanie: `Do_kwadratu(2)`.

- *przekazywanie przez zmienną (adres)*

W przypadku przekazywania przez zmienną, do procedury nie jest przekazywana wartość parametru, a adres zmiennej zewnętrznej podanej w wywołaniu w miejscu parametru i wszystkie operacje wykonywane na parametrze są tak naprawdę wykonywane na zmiennej zewnętrznej.

```
procedure Do_kwadratu(var a:integer);
// definicja procedury Do_kwadratu z deklaracją parametru a typu integer przekazywanego przez zmienną (słowo kluczowe var)
begin
  a:=a*a;   // pomnóż a przez siebie samą i zapisz wynik w a
  write(a); // wypisz wartość zmiennej a na ekranie
end;

// program główny
var b:integer;  // zadeklaruj zmienną b typu integer
begin
  b:=2; // przypisz zmiennej b wartość 2
  Do_kwadratu(b);   // wywołanie procedury Do_kwadratu z parametrem b
  write(b); // wypisz wartość zmiennej b na ekranie (wynik: 4)
end;
```

W powyższym przykładzie parametr `a` procedury `Do_kwadratu` zdeklarowany został jako parametr przekazywany przez zmienną (słowo kluczowe `var`). W konsekwencji, po wywołaniu procedury w programie głównym ze zmienną `b` w roli parametru, przekazana zostanie nie wartość, lecz adres tej zmiennej, w związku z czym wszystkie operacje wewnątrz procedury wykonywane będą bezpośrednio na zmiennej `b`, której wartość pozostanie trwale zmieniona po zakończeniu działania procedury (parametr `a` nie jest więc teraz zmienną lokalną procedury, tak jak to miało miejsce w przypadku przekazywania przez wartość, lecz rodzajem bezpośredniej referencji do podanej w wywołaniu zmiennej `b`).
Przekazywanie przez zmienną umożliwia efektywniejsze przekazywanie procedurze większych struktur danych, w przypadku których kopiowanie całej struktury zużywałoby niepotrzebne zasoby systemu, pozwalając jednocześnie na ich bezpośrednią modyfikację w obrębie procedury. W przypadku wyboru takiego typu przekazywania nie jest możliwe jednak wywołanie procedury z wartością stałą w roli parametru - wywołanie w stylu `Do_kwadratu(2)` spowoduje błąd kompilacji.
Języki programowania umożliwiające wyodrębnianie w programie procedur noszą nazwę języków proceduralnych. Do języków proceduralnych zaliczamy niemal wszystkie znane obecnie imperatywne języki programowania. Przykładem języków nieproceduralnych mogą być najprostsze języki asemblerowe służące do programowania nieskomplikowanych układów mikroprocesorowych oraz pierwsze wersje języka Basic.

3.3. Programowanie strukturalne

Programowanie strukturalne może być postrzegane jako pewna poddyscyplina programowania proceduralnego wprowadzająca stosowanie dodatkowych struktur, umożliwiających efektywniejsze sterowanie przepływem programu oraz zalecająca podział kodu na moduły realizujące poszczególne zadania.

Poniżej przedstawiony został przegląd głównych struktur kontroli przepływu stosowanych we współczesnych strukturalnych językach programowania. Struktury te różnią się nieco składnią pomiędzy poszczególnymi językami, jednak filozofia ich działania pozostaje jednakowa.

Bloki warunkowe (IF...THEN...ELSE, CASE)

Bloki warunkowe umożliwiają uzależnienie wykonania danego fragmentu kodu od spełnienia bądź nie spełnienia warunku logicznego.

Pseudokod

```
JEŚLI (warunek) TO {
  Instrukcja1;
  Instrukcja2;
} W PRZECIWNYM WYPADKU {
  Instrukcja3;
  Instrukcja4;
  Instrukcja5;
}
```

Składnia w języku Pascal

```
if (warunek) then begin
  Instrukcja1;
  Instrukcja2;
end else begin
  Instrukcja3;
  Instrukcja4;
  Instrukcja5;
end;
```

```
WYBIERZ (zmienna) SPOŚRÓD
  wartość1:
    Instrukcja1;
    Instrukcja2;
  wartość2:
    Instrukcja3;
    Instrukcja4;
  domyślnie:
    Instrukcja5;
    Instrukcja6;
KONIEC WYBIERZ
```

```
case (zmienna) of
  wartość1:
    Instrukcja1;
    Instrukcja2;
    break;
  wartość2:
    Instrukcja3;
    Instrukcja4;
    break;
  else
    Instrukcja5;
    Instrukcja6;
end;
```

Przykład:

Pseudokod

```
wczytaj liczbę z klawiatury
  i zapisz ją w zmiennej a
JEŚLI (a jest mniejsze od 0) TO {
  napisz na ekranie ‘Liczba ujemna’
} W PRZECIWNYM WYPADKU {
  napisz na ekranie ‘Liczba dodatnia’
}
```

Odpowiednik w języku Pascal

```
read(a);
if (a<0) then
  write(‘Liczba ujemna’);
else
  write(‘Liczba dodatnia’);
```

```
wczytaj liczbę z klawiatury
  i zapisz ją w zmiennej a
WYBIERZ a SPOŚRÓD
  1:
    napisz na ekranie ‘Wprowadzono 1’
  2:
    napisz na ekranie ‘Wprowadzono 2’
  domyślnie:
    napisz na ekranie ‘Wprowadzono
      inną liczbę niż 1 i 2’
KONIEC WYBIERZ
```

```
read(a);
case a of
  1:
    Write(‘Wprowadzono 1’);
    break;
  2:
    Write(‘Wprowadzono 2’);
    break;
  else
    Write(‘Wprowadzono inną
      liczbę niż 1 i 2’);
end;
```

Bloki powtórzeniowe (REPEAT...UNTIL, WHILE...DO)

Bloki powtórzeniowe umożliwiają wielokrotne powtarzanie wykonywanie tego samego fragmentu kodu, tak długo aż spełniony jest warunek.

Pseudokod

```
POWTARZAJ {
  Instrukcja1;
  Instrukcja2;
} DO MOMENTU AŻ (warunek);
```

```
DOPÓKI (warunek) WYKONUJ {
  Instrukcja1;
  Instrukcja2;
}
```

Składnia w języku Pascal

```
repeat
  Instrukcja1;
  Instrukcja2;
until (warunek);
```

```
while (warunek) do begin
  Instrukcja1;
  Instrukcja2;
end;
```

Poniższy przykład będzie tak długo pobierał liczbę od użytkownika, aż nie wprowadzi on liczby 5:

Pseudokod

```
POWTARZAJ {
  wczytaj liczbe z klawiatury
```

Odpowiednik w języku Pascal

```
repeat
  read(a);
```

```
    i zapisz ją w zmiennej a     until (a=5);
} DO MOMENTU AŻ (a jest równe 5);
```

Program realizujący dokładnie to samo zadanie można by zapisać w analogiczny sposób przy użyciu bloku (DOPÓKI .. WYKONUJ ..):

Pseudokod

```
nadaj zmiennej a wartość 0
DOPÓKI (a jest różne od 5) WYKONUJ {
  wczytaj liczbe z klawiatury
    i zapisz ją w zmiennej a
}
```

Składnia w języku Pascal

```
a:=0;
while (a<>5) do begin
  read(a);
end;
```

W tym przypadku konieczne jest najpierw nadanie zmiennej a wartości innej niż 5, aby zapewnić co najmniej jedno wykonanie pętli i tym samym co najpierw jedno pobranie liczby od użytkownika.

EITC/SE/CPF Wykład 3

Odpowiedziałeś poprawnie na 0 z 0 pytań

Część pierwsza

3.4. Różne paradygmaty programowania: Programowanie obiektowe

Programowania obiektowe jest paradygmatem programowania stosującym w tworzeniu programów komputerowych koncepcję tzw. *obiektów* i współoddziaływania między nimi.

Podstawowymi elementami programu obiektowego są *obiekty* opisywane przez *klasy*, podobnie jak zmienne opisywane są przez typy danych. Klasa definiuje strukturę danych przechowywanych w obiekcie, oraz dodatkowo wszystkie operacje, które mogą być na tych danych wykonywane, w formie funkcji składowych klasy zwanych *metodami*. Cały program (który sam może być traktowany jako obiekt pewnej klasy) składa się z wielu współpracujących ze sobą obiektów, które w trakcie działania programu przekazują między sobą informacje i zmieniają swój stan (określony przez wartości wszystkich atrybutów).

Idea programowania obiektowego idzie jednak znacznie dalej, dostarczając specyficznych mechanizmów umożliwiających dużo bardziej efektywne i intuicyjne modelowanie rozwiązywanych problemów. Mechanizmy te obejmują między innymi dziedziczenie, polimorfizm oraz dynamiczne wiązanie metod, które omówione zostaną dokładniej w dalszej części wykładu.

Programowanie zorientowane obiektowo odzwierciedla w znacznie lepszy sposób niż inne paradygmaty zjawiska i procesy zachodzące w świecie realnym i pozwala opisywać problemy w sposób bardzo zbliżony do tego, w jaki postrzega je człowiek . Paradygmat ten jest z tego powodu (zazwyczaj łącznie z paradygmatem zdarzeniowym) najpowszechniej stosowaną współcześnie metodologią programowania w środowisku aplikacji biznesowych.

3.5. Programowanie kierowane zdarzeniami

W programowaniu kierowanym zdarzeniami (ang. *event-driven programming*) przepływ sterowania nie jest ściśle określony przez kolejność poleceń, lecz determinowany poprzez tzw. zdarzenia wywoływane akcjami podejmowanymi przez użytkownika (np. kliknięcie myszką) lub zmianami stanów obiektów programu bądź programów zewnętrznych. W paradygmacie tym program oczekuje na zdarzenia, o których zaistnieniu informowany jest w postaci komunikatów przychodzących od innych obiektów, i na które musi odpowiedzieć podejmując odpowiednie działanie.

Poniższy rysunek przedstawia uproszczony schemat programu kierowanego zdarzeniami:

Jądro programu zaprojektowanego zgodnie z paradygmatem zdarzeniowym składa się zasadniczo z wykonywanej cyklicznie pętli, wewnątrz której następuje ciągłe sprawdzanie, czy z odpowiednich obiektów nie nadeszły komunikaty o zdarzeniach. W razie odebrania komunikatu o zdarzeniu wywoływany jest odpowiedni fragment kodu odpowiedzialny za obsługę danego zdarzenia (tzw. procedura obsługi zdarzenia; ang. *event handler*), po którego wykonaniu sterowanie powraca z powrotem do pętli głównej.

Poniżej przedstawione są (w pseudokodzie) rozwiązania prostego problemu odczytu dwóch liczb z klawiatury, zsumowania ich i wyświetlenia na ekranie zrealizowane w paradygmacie imperatywnym oraz zdarzeniowym:

Wersja imperatywna:

```
wczytaj liczbę (z klawiatury) i zachowaj ją w zmiennej A[0]
wczytaj liczbę (z klawiatury) i zachowaj ją w zmiennej A[0]

wydrukuj A[0]+A[1]
```

Wersja zdarzeniowa:

```
Ustaw licznik K na 0
powtarzaj {

  jeśli została wprowadzona liczba (z klawiatury) {

      // zarejestrowanie zdarzenia

      // obsługa zdarzenia

      zachowaj ją w A[K] i zwiększ K

      jeśli K jest równe 2 wydrukuj A[0]+A[1] i zresetuj K do 0

  }

}
```

Zdarzeniem w powyższym przykładzie będzie wprowadzenie liczby na klawiaturze, zaś procedurą jego obsługi – 2 linijki kodu wewnątrz środkowych nawiasów klamrowych.

Ponieważ część kodu odpowiedzialna za przechwytywanie zdarzeń oraz główna pętla programu nie są zależne od konkretnej aplikacji, wiele środowisk programistycznych przejmuje na siebie ich implementację i pozostawia programiście jedynie napisanie kodu procedur obsługi poszczególnych zdarzeń. W podanym na rysunku schemacie będzie to część programu w ramce "Reakcja na zdarzenie".

Programowanie sterowane zdarzeniami jest dominującym typem programowania stosowanym przy tworzeniu graficznych interfejsów użytkownika (GUI). Zdarzeniami w tym przypadku są naciśnięcia klawiszy klawiatury i myszy, kliknięcia w obiekty interfejsu, żądania odświeżenia obrazu ze strony poszczególnych okienek itp. Współczesne języki programowania wysokiego poziomu i realizowane przy ich pomocy aplikacje łączą obecnie najczęściej paradygmat obiektowy z paradygmatem programowania sterowanego zdarzeniami. Na takim właśnie modelu bazuje m.in. system operacyjny Microsoft® Windows®. Zdarzeniowy model programowania stosowany jest także w wysoko wydajnych serwerach sieciowych, gdzie zdarzeniami są żądania połączenia, nadejście danych do odbioru itp.

Istnieje wiele innych paradygmatów programowania, a w szczególności podklasy progra-mowania deklaratywnego (w tym programowanie funkcjonalnie i logicznie), w którym (w przeciwieństwie do programowania imperatywnego) zamiat specyfikacji sposobu rozwiązania danego problemu określa się jedynie zakładany cel, czyli to, co chcemy osiągnąć, nie wnikając w sam sposób (algorytm) osiągnięcia tego celu. Metodologie te wykorzystywanie są jednak głównie w specjalistycznych zastosowaniach inżynierskich i wykraczają one poza zakres omawianej w ramach niniejszego studium tematyki.

4.Idea programowania obiektowego

Programowanie obiektowe wprowadza zupełnie odmienny od programowania proceduralnego punkt widzenia na program jako rozwiązanie zakładanego problemu obliczeniowego, będący znacznie lepszym odzwierciedleniem sposobu postrzegania przez człowieka procesów zachodzących w realnym świecie i umożliwiający ich efektywniejsze modelowanie. W ramach niniejszego rozdziału przyjrzymy się bardziej szczegółowo koncepcji ogólnej oraz poszczególnym ideom związanym z tym paradygmatem programowania oraz ich realizacji na przykładach w języku programowania C++.

EITC/SE/CPF Wykład 6

Odpowiedziałeś poprawnie na 0 z 0 pytań

CZĘŚĆ PIERWSZA

4. Składnia i semantyka języka C# c.d.

4.5. Imperatywne struktury sterowania przepływem

Instrukcja warunkowa

Instrukcja warunkowa ma w języku C# następującą postać:

```
if (warunek_logiczny) instrukcja;
if (warunek_logiczny) instrukcja1 else instrukcja2;

// zamiast pojedynczej instrukcji można w obu miejscach użyć bloku
// instrukcji zamkniętego w nawiasy klamrowe:

if (warunek_logiczny) {
instrukcja1;
..
instrukcjaN;
} else {
instrukcjaM;
..
instrukcjaK;
}
```

Jeśli spełniony jest warunek logiczny wykonywane są instrukcje w ramach bloku znajdującego się zaraz za warunkiem. W przeciwnym wypadku wykonywany jest blok znajdujący się po opcjonalnym słowie kluczowym `else`.

Instrukcja wyboru

```
switch (wyrażenie)
{
case wartość1,wartość2:
// tu instrukcje dla tego przypadku
break;
case inna_wartość:
// tu instrukcje dla tego przypadku
break;
case jeszcze_inna_wartość:
// tu instrukcje dla tego przypadku
break;
default:
// tu instrukcje dla wartości domyślnej
}
```

Instrukcja `break` powoduje wyjście z bloku `switch` po wykonaniu ciągu instrukcji dla danego przypadku. Gdybyśmy jej nie użyli, program kontynuowałby wykonywanie przechodząc do kolejnej instrukcji w ramach bloku `switch`.

Pętle warunkowe

- Pętla WHILE

```
while (warunek_logiczny) instrukcja;
// lub
while (warunek_logiczny) {
instrukcja1;
..
instrukcjaN;
}
```

W konstrukcji tej tak długo, jak spełniony jest warunek logiczny, wykonywany będzie ciąg instrukcji wewnątrz bloku, przy czym warunek logiczny sprawdzany jest przed wejściem do bloku.

- Pętla DO..WHILE

```
do instrukcja while (warunek_logiczny);
// lub
do {
instrukcja1;
..
instrukcjaN;
} while (warunek_logiczny);
```

Konstrukcja ta działa analogicznie do poprzedniej, z tą różnicą, że warunek logiczny sprawdzany jest PO wykonaniu instrukcji zawartych wewnątrz bloku. Oznacza to, że instrukcje te zawsze zostaną wykonane co najmniej raz.

Pętla FOR

Pętla FOR ma w języku C# dosyć złożoną postać, która wygląda następująco:

```
for (<lista_wyrażeń_inicjujących>; <wyrażenie_warunkowe>; <lista_iteracji>) {
<instrukcje>
}
</instrukcje></lista_iteracji></wyrażenie_warunkowe></lista_wyrażeń_inicjujących>
```

Pętla FOR wykonywana jest w następujący sposób:

1. najpierw wykonywane są polecenia inicjujące, służące do nadania początkowych wartości zmiennym iteracyjnym
2. sprawdzane jest wyrażenie warunkowe i jeśli jest spełnione, wykonywane są instrukcje wewnątrz bloku (w przeciwnym wypadku pętla zostaje zakończona)
3. wykonywane są instrukcje z listy iteracji, służące z założenia do modyfikacji wartości zmiennych iteracyjnych
4. następuje powrót do punktu 2

Działanie to najlepiej wytłumaczyć na prostym przykładzie:

```
for (int i=1;i<11;i++) {
Console.Write("{0} ", i);   // wypisanie na ekranie bieżącej wartości
// zmiennej i
}
```

W powyższym przypadku zmienna `i` zainicjowana zostanie wartością `1` i będzie zwiększana o `1` po każdym przebiegu pętli, aż do osiągnięcia wartości `11`. W efekcie pętla wykona się `10` razy i spowoduje wypisanie na ekranie ciągu liczb od `1` do `10`. Ponieważ zmienna `i` została zadeklarowana w ramach listy wyrażeń inicjujących pętli, jest ona zmienną lokalną pętli widoczna tylko w jej obrębie.
W ramach pętli FOR możliwe jest stosowanie następujących instrukcji specjalnych:

- `break;` - powoduje natychmiastowe przerwanie wykonywania pętli
- `continue;` - powoduje natychmiastowe przejście do kolejnej iteracji pętli

Pętla FOR w języku C# daje dużo więcej możliwości niż modelowa pętla FOR i pozwala na konstruowanie różnorodnych struktur sterowania, częstokroć nie mających nic wspólnego z tą ostatnią.

Iterator FOREACH

FOREACH jest rodzajem pętli wykonującej się dla każdego elementu listy (kolekcji):

```
foreach (<typ> <nazwa_zmiennej_reprezentującej_element_listy> in <lista>) {
<instrukcje>
}
</instrukcje></lista></nazwa_zmiennej_reprezentującej_element_listy></typ>
```

Działanie tej konstrukcji znowu najlepiej rozpatrzeć na przykładzie:

```
// deklaracja tablicy zawierającej imiona
string[] imiona = {"Adam", "Jakub", "Zygmunt", "Piotr"};

foreach (string osoba in imiona)
{
Console.WriteLine("{0} ", osoba);
}
```

Powyższy kod spowoduje wykonanie instrukcji wypisania na ekranie wartości zmiennej `osoba`, która to wartość w każdym przebiegu pętli będzie przechodziła po kolejnych elementach tablicy `imiona`.

Dalej

EITC/SE/CPF Wykład 7 i 8

Odpowiedziałeś poprawnie na 0 z 0 pytań

Część pierwsza

5. Obiektowy aspekt języka C#

Język C# jest językiem w pełni zorientowanym obiektowo tzn. dostarcza składni i semantyki pozwalającej na tworzenie aplikacji w paradygmacie obiektowym. Co więcej, pomimo, iż zawiera on także elementy imperatywne, to w zasadzie wszystkie programy pisane w tym języku są programami obiektowymi, gdyż jak pokazaliśmy w ramach poprzednich wykładów, nawet najprostszy program będący zwykłym ciągiem instrukcji i tak stanowi wywołanie statycznej metody Main, będącej metodą pewnej klasy. Wynika to bezpośrednio z czysto obiektowej specyfiki samego środowiska .NET Framework, na którym operuje ten język.

5.1. Model obiektowy środowiska .NET

W środowisku .NET framework wszystkie klasy wywodzą się (dziedziczą) z klasy bazowej System.Object, zarówno te stanowiące integralną część środowiska, jak i klasy zdefiniowane przez użytkownika. Klasa ta zawiera podstawowe metody wspólne dla obiektów wszystkich klas takie jak kopiowanie, porównywanie czy przydzielanie i zwalnianie pamięci dla składowych klasy. Nawet proste typy danych są klasami pochodnymi System.Object, np. typ **string** jest tak naprawdę realizacją klasy System.String, **int** – System.Int32 itp. (wszystkie odpowiedniki znajdują się w tabeli poświęconej typom danych w jednym z poprzednich wykładów).
Platforma .NET Framework udostępnia standardowo w ramach swojej Biblioteki Klas szereg gotowych klas implementujących wstępnie zaprogramowane rozwiązania pogrupowane tematycznie w ramach przestrzeni nazw System oraz jej podprzestrzeni:

System	Wsparcie dla podstawowych potrzeb programistycznych. Obejmuje typy bazowe takie jak String, Boolean, Decimal, Object itp., wsparcie dla środowiska konsolowego, funkcje matematyczne oraz bazowe klasy dla atrybutów, wyjątków i tablic.
System.CodeDom	Funkcje umożliwiające pisanie i uruchamianie kodu w trakcie działania programu.
System.Collections	Definiuje różne rodzaje powszechnie stosowanych w programowaniu pojemników oraz kolekcji takich jak listy, kolejki, stosy, tablice haszowane oraz słowniki.
System.ComponentModel	Dostarcza szkieletu dla implementacji komponentów oraz kontrolek.
System.Configuration	Dostarcza infrastruktury do obsługi danych konfiguracyjnych
System.Data	Dostarcza komponentów pozwalających na dostęp do danych i usług w ramach architektury ADO.NET (następca ActiveX Data Objects). W ramach tej przestrzeni nazw zaimplementowane są m.in. mechanizmy dostępu do zewnętrznych baz danych.
System.Deployment	Umożliwia dostosowanie sposobu rozbudowywania i aktualizacji aplikacji po jej wdrożeniu.
System.Diagnostics	Dostarcza funkcjonalności umożliwiających diagnoszowanie aplikacji, takich jak dzienniki zdażeń, liczniki wydajności, śledzenie czy interakcja z procesami systemowymi.
System.DirectoryServices	Umożliwia prosty dostęp do Usług Katalogowych.
System.Drawing	Zapewnia dostęp do funkcji graficznych środowiska GDI+ (graficzne środowisku systemu Windows®), w tym wsparcie dla grafiki 2D i grafiki wektorowej, obróbki obrazów, drukowania oraz usług tekstowych.
System.EnterpriseServices	Dostarcza obiektom .NET dostępu do usług COM+.
System.Globalization	Dostarcza funkcjonalności upraszczających tworzenie aplikacji wielo-językcznych i wielokulturowych.

System.IO	Dostarcza interfejs dostępu do systemu plików oraz mechanizmy strumieniowej wymiany danych.
System.Management	Umożliwia dostęp do informacji systemowych takich jak ilość wolnego miejsca na dysku, aktualne wykorzystanie procesora itp.
System.Media	Dostarcza funkcjonalności umożliwiających odgrywanie plików dźwiękowych.
System.Messaging	Dostarcza funkcjonalności do zarządzania kolejkami wiadomości przesyłanych w obrębie sieci.
System.Net	Dostarcza interfejsu dla wielu protokołów sieciowych takich jak HTTP, FTP czy SMTP, w tym także protokołów bezpiecznej komunikacji (SSL).
System.Reflection	Dostarcza obiektowy widok na typy, metody i atrybuty, umożliwiając dynamiczne tworzenie i używanie typów w ramach działającej aplikacji.
System.Resources	Umożliwia zarządzanie różnorodnymi zasobami aplikacji (teksty, obrazy) uwzględniając aspekty takie jak wielojęzykowość i wielokulturowość.
System.Runtime	Umożliwia zarządzanie uruchomieniowym zachowaniem aplikacji oraz środowiska Common Language Runtime.
System.Security	Dostarcza funkcjonalności dla systemu bezpieczeństwa środowiska uruchomieniowego Common Language Runtime. Umożliwia wbudowanie w aplikacje odpowiednich mechanizmów zabezpieczeń na bazie polityk i praw dostępu. Dostarcza takich funkcjonalności jak szyfrowanie czy uwierzytelnianie.
System.ServiceProcess	Umożliwia tworzenie aplikacji działających jako usługi w systemie Windows.
System.Text	Dostarcza wsparcie dla różnych kodowań znaków, wyrażeń regularnych oraz bardziej efektywnych mechanismów manipulacji łańcuchami znaków.
System.Threading	Zapewnia wsparcie dla programowania wielowątkowego, w tym mechanizmy synchronizacji wątków.
System.Timers	Umożliwia wykonywanie określonych czynności w określonych momentach czasu.
System.Transactions	Dostarcza wsparcie dla lokalnych i rozproszonych transakcji.
System.Web	Dostarcza różnorodnych funkcjonalności związanych z siecią WEB, zapewniając wsparcie m.in. dla komunikacji przeglądarka-serwer czy tworzenie usług sieciowych XML Web Services. Większość funkcjonalności oferowanej przez tą bibliotekę implementuje architekturę webową ASP.NET.
System.Windows.Forms	Zawiera implementację architektury Windows Forms stanowiącą interfejs dostępu do graficznego środowiska okienkowego systemu Windows.

System.Xml	Dostarcza wsparcie do przetwarzania danych w formacie XML.

Pogrubioną czcionką zaznaczono przestrzenie nazw najistotniejsze z punktu widzenia większości typowych średniozaawansowanych zadań programistycznych.

W ramach niniejszego wykładu nie będziemy omawiać dokładnie wszystkich klas oferowanych przez platformę .NET Framework, gdyż mogłoby to stanowić tematykę obszernej książki. Dokładna dokumentacja środowiska dostępna jest natomiast na stronie Microsoftu w ramach Microsoft Developer Network (MSDN):

http://msdn2.microsoft.com/en-us/library/default.aspx, której zasoby przywoływane są także w ramach pomocy on-line środowiska programistycznego Visual Studiu. W dalszej części kursu przyjrzymy się bardziej szczegółowo jedynie klasom udostępnianym w ramach przestrzeni nazw System.Windows.Forms, służącym do obsługi graficznego interfejsu użytkownika.

EITC/SE/CPF Wykład 9 i 10

Odpowiedziałeś poprawnie na 0 z 0 pytań

Część pierwsza

6. Tworzenie aplikacji okienkowych

Visual Studio .NET jest przykładem środowiska typu RAD (Rapid Application Development), czyli środowiska programistycznego umożliwiającego szybkie tworzenie efektownych i funkcjonalnych aplikacji przy minimalnej ingerencji w kod programu. Szeroka tematyka wytwarzania aplikacji w tym środowisku znacznie wykracza poza zakres kursu – Celem niniejszego wykładu jest jedynie zapoznanie słuchacza z ideą RAD oraz projektowania wizualnego.

6.1. Model obiektowy Windows Forms

Okienkowy interfejs graficzny systemu Windows ma charakter obiektowy w intuicyjnym rozumieniu tego pojęcia. Obiektami są tu wszystkie okna, przyciski, pola tekstowe i inne element odpowiedzialne za komunikację pomiędzy systemem a użytkownikiem, których własności oraz zachowanie określane są poprzez definiujące je odpowiednie klasy.

Funkcjonalności tworzenia i operowania na okienkowym interfejsie systemu Windows w ramach platformy .NET Framework osiągane są za pośrednictwem interfejsu zwanego Windows Forms, dającego dostęp do natywnych elementów interfejsu systemu operacyjnego Microsoft Windows poprzez obudowanie istniejących funkcji Windows API w ramach biblioteki klas platformy .NET Framework. Interfejs Windows Forms zaimplementowany jest w obrębie przestrzeni nazw System.Windows.Forms biblioteki klas.

Środowisko programistyczne Visual Studio dostarcza narzędzi pozwalających na budowanie aplikacji okienkowych w sposób wizualny tzn. bez konieczności znacznej ingerencji w kod programu, który w większości generowany jest automatycznie. Tworzenie podstawowych aplikacji Windows Forms sprowadza się w zasadzie do umiejscowienia odpowiednich kontrolek (takich jak przyciski, pola tekstowe, listy wyboru itp.) w obrębie okien (zwanych formularzami lub formatkami, z ang. *form*) oraz zaprogramowaniu czynności, jakie mają zostać wykonane w wyniku zaistnienia określonych zdarzeń (np. kliknięcia na dany przycisk)

6.1. Tworzenie nowej aplikacji okienkowej

W celu utworzenia nowej aplikacji okienkowej Windows Forms należy w programie Visual Express 2012 wybrać "New project..." z menu "File", a następnie w oknie, które się pojawi wybrać "Windows Form Application" i wpisać nazwę dla nowej aplikacji analogicznie jak w przypadku aplikacji konsolowej.

Po kliknięciu przycisku OK ukaże się już jednak znacznie inna postać obszaru roboczego:

W miejscu edytora kodu programu widoczny jest mianowicie widok projektanta formularzy z domyślnie utworzonym (pustym) głównym formularzem aplikacji o nazwie Form1. Ponadto panel "Toolbox" po lewej stronie zawiera teraz cały zestaw kontrolek, które będą mogły być umieszczane w ramach projektowanego okna aplikacji.

6.2. Struktura aplikacji

Panel "Solution Explorer" w prawym górnym rogu ekranu zawiera pogrupowany hierarchicznie wykaz plików wchodzących w skład nowo utworzonej aplikacji.

Najistotniejszymi plikami nowo utworzonej aplikacji z punktu widzenia logiki działania programu są Program.cs – zawierający główny kod sterujący działaniem aplikacji, oraz Form1.cs – zawierający kod odpowiadającego mu formularza. Po dwukrotnym kliknięciu na plik Program.cs, w głównej części ekranu ukaże się okno edytora z jego zawartością:

```
using System;
using System.Collections.Generic;

using System.Windows.Forms;


namespace WindowsApplication1

{

   static class Program

   {

      /// <summary>

      /// The main entry point for the application.

      /// </summary>

      [STAThread]

      static void Main()

      {

         Application.EnableVisualStyles();

         Application.SetCompatibleTextRenderingDefault(false);

         Application.Run(new Form1());

      }

   }

}
```

Zawartość tego pliku stanowi kod sterujący aplikacją i jego zadanie sprowadza się w tym wypadku do utworzenia okna Form1 (poprzez utworzenie obiektu klasy Form1) oraz uruchomienie metody Run statycznej klasy Application ze wskazaniem tego obiektu jako parametru, co spowoduje uruchomienie wszystkich wewnętrznych mechanizmów środowiska Windows, wyświetlenie interfejsu użytkownika oraz zapewnienie odpowiedniego wyzwalania zdarzeń. Istotne jest, że kod pliku Program.cs generowany jest automatycznie przez środowisko Visual Express 2012 i w większości przypadków nie wymaga żadnej ingerencji ze strony programisty.
W większości przypadków jedynie zmiany w kodzie dokonywane przez programistę mają miejsce w plikach formularzy. Dwukrotne kliknięcie na plik Form1.cs spowoduje otwarcie widoku projektanta formularzy z tym formularzem – aby móc edytować kod formularza, należy kliknąć na pliku prawym klawiszem myszy i wybrać "View Code".

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication1
{
  public partial class Form1 : Form
  {
    public Form1()
    {
      InitializeComponent();
    }
  }
}
```

Jak widzimy powyżej, klasa Form1 dziedziczy z klasy Form, co od razu czyni obiekt tej klasy formularzem (oknem) Windows Forms, posiadającym pełną wspólną funkcjonalność formularzy, w tym między innymi pasek tytułu, przyciski minimalizacji/ maksymalizacji/ zamykania, menu systemowe oraz krawędzie.

EITC/SE/CPF Laboratorium 1

Instalacj1a i poznawanie środowiska Visual Express 2012 dla Windows Desktop

1. Pobrać ze strony firmy Microsoft® instalator środowiska Visual Express 2012 dla Windows Desktop (link do strony pobierania: http://go.microsoft.com/?linkid=9816758) i zainstalować program (instalator wymaga połączenia z Internetem). Następnie postępujemy wg instrukcji:
   - wybieramy miejsce docelowe instalacji oraz akceptujemy warunki licencji jak na rys poniżej,
   - Klikamy przycisk instal który pojawił się po wybraniu akceptacji warunków licejcni,
   - Instalator łączy się z witryną Microsoft® i pobiera wymagane pliki, w miedzy czasie będzie wymagany restart komputera,
   - Po restarcie komutera instalator dokończy proces instalacyjny i jeśli wszystko przebiegło poprawnie otrzymamy komunikat jak na rys poniżej,
   - Po instalacji otrzymujemy wersje 30 dniową, możemy jednak dokonać darmowej rejestracji, klikając przycisk register online, dzieki której otrzymamy klucz produktu sciągający limit czasowy użytkowania naszego środowiska,
   - Na stronie Microsoft® zakładamy nowe konto jeśli go nie posiadamy, wybierając link Sign up now,
   - Wypełniamy wsztstkie pola,
   - Następnie uzupełniamy kolejny formularz tym razem dotyczący zainstalowanego produktu, w naszym przypadku Visual Studio Express 2012 for Windows Desktop i klikamy przycisk Next
   - Po poprawnymm wypełnieniu frmularza powinien się wyświetlić klucz naszego produktu oraz powinniśmy dostać jego kopie na skrzynke emailową którą podaliśmy podczas rejestracji konta.

- Po skopiowaniu klucza w pole Product key: i kliknieciu przycisku Next otrzymamy komunikat

- Ostatecznie po wykonaniu wszystkich powyższych czynności możemy urucomić naszą darmową wersje Visual Studio Express 2012 gdzie powita nas ekran startowy jak na rys poniżej.

2. Zapoznać się z interfejsem aplikacji.

Tworzenie prostej aplikacji konsolowej

3. Utworzyć nową aplikację konsolową, a następnie wprowadzić kod programu „Hello world" omawianego w ramach wykładu i uruchomić aplikację.

Uwaga: Aplikacja konsolowa ulega zakończeniu natychmiast po wykonaniu kodu zawartego w metodzie `main()`, co skutkuje zamknięciem okna konsoli. Aby obejrzeć wynik działania aplikacji przez jej zamknięciem, można na końcu kody dodać linijkę `Console.ReadKey();`, która spowoduje, że aplikacja będzie oczekiwała na naciśnięcie dowolnego klawisza przez użytkownika. Wynik programu „Hello World":

4. Poćwiczyć proste elementy składni omawiane w ramach wykładu (definiowanie zmiennej, przypisywanie wartości, operacje arytmetyczne).

Przykładowe ćwiczenia znajdują się w dalszej części niniejszego pliku.

Przykładowe ćwiczenia

1. Operatory unarne:

```
using System;


class Unary

{
```

```
public static void Main()
{
    int unary = 0;
    int preIncrement;
    int preDecrement;
    int postIncrement;
    int postdecrement;
    int positive;
    int negative;
    sbyte bitNot;
    bool logNot;

    preIncrement = ++unary;
    Console.WriteLine("pre-inkrementacja: {0}", preIncrement);

    preDecrement = --unary;
    Console.WriteLine("pre-dekrementacja: {0}", preDecrement);

    postdecrement = unary--;
    Console.WriteLine("post-dekrementacja: {0}", postdecrement);

    postIncrement = unary++;
    Console.WriteLine("post-inkrementacja: {0}", postIncrement);

    Console.WriteLine("Finalna wartość zmiennej „unary”: {0}", unary);

    positive = -postIncrement;
    Console.WriteLine("Wartość dodatnia: {0}", positive);

    negative = +postIncrement;
    Console.WriteLine("Wartość ujemna: {0}", negative);
```

```
        bitNot = 0;

        bitNot = (sbyte)(~bitNot); // zapis (sbyte) oznacza wymuszenie, żeby wynik wyrażenia ~bitNot był
również typu sbyte (tzw. rzutowanie typów)

        Console.WriteLine("Negacja bitowa: {0}", bitNot);


        logNot = false;

        logNot = !logNot;

        Console.WriteLine("Negacja logiczna: {0}", logNot);

        Console.ReadKey();

    }

}
```

2. Operatory binarne:

```
using System;


class Binary

{

    public static void Main()

    {

        int x, y, result;

        float floatresult;


        x = 7;

        y = 5;


        result = x+y;

        Console.WriteLine("x+y: {0}", result);


        result = x-y;

        Console.WriteLine("x-y: {0}", result);
```

```
        result = x*y;
        Console.WriteLine("x*y: {0}", result);

        result = x/y;
        Console.WriteLine("x/y: {0}", result);

        floatresult = (float)x/(float)y;
        Console.WriteLine("x/y: {0}", floatresult);

        result = x%y;
        Console.WriteLine("x%y: {0}", result);

        result += x;
        Console.WriteLine("result+=x: {0}", result);
        Console.ReadKey();
    }
}
```

3. Tablice:

```
using System;

class Array
{
    public static void Main()
    {
        int[] myInts = { 5, 10, 15 };
        bool[][] myBools = new bool[2][];
        myBools[0] = new bool[2];
        myBools[1] = new bool[1];
        double[,] myDoubles = new double[2, 2];
        string[] myStrings = new string[3];
```

```
        Console.WriteLine("myInts[0]: {0}, myInts[1]: {1}, myInts[2]: {2}", myInts[0], myInts[1], myInts[2]);


        myBools[0][0] = true;

        myBools[0][1] = false;

        myBools[1][0] = true;

        Console.WriteLine("myBools[0][0]: {0}, myBools[1][0]: {1}", myBools[0][0], myBools[1][0]);


        myDoubles[0, 0] = 3.147;

        myDoubles[0, 1] = 7.157;

        myDoubles[1, 1] = 2.117;

        myDoubles[1, 0] = 56.00138917;

        Console.WriteLine("myDoubles[0, 0]: {0}, myDoubles[1, 0]: {1}", myDoubles[0, 0], myDoubles[1, 0]);


        myStrings[0] = "Joe";

        myStrings[1] = "Matt";

        myStrings[2] = "Robert";

        Console.WriteLine("myStrings[0]: {0}, myStrings[1]: {1}, myStrings[2]: {2}", myStrings[0],
myStrings[1], myStrings[2]);

        Console.ReadKey();

    }

}
```

STATUS PRZESŁANEGO ZADANIA

Status przesłanego zadania	To zadanie nie wymaga wysyłania niczego online

EITC/SE/CPF Laboratorium 2

Imperatywne struktury języka C#

1. Poćwiczyć omówione na wykładzie struktury wyboru (IF, SWITCH) oraz pętle (WHILE, DO, FOR, FOREACH). Przykładowe ćwiczenia znajdują się na następnych stronach.
2. Napisać program wypisujący liczby od 20 do 36 z krokiem 2, tzn. 20,22,24,..,34,36.

3. Napisać metodę obliczającą silnię z podanej liczby i program demonstrujący działanie metody. Podpowiedź: silnią liczby naturalnej `n` (w notacji matematycznej `n !`, co czytamy „n silnia”) nazywamy iloczyn wszystkich dodatnich liczb naturalnych nie większych niż n. Np. `6!=1*2*3*4*5*6=120`.

Przykładowe ćwiczenia oraz rozwiązania zadań znajdują się w dalszej części niniejszego pliku.

Przykładowe ćwiczenia

1. Konstrukcja IF:

```
using System;

class IfSelect
{
    public static void Main()
    {
        string myInput;
        int myInt;

        Console.Write("Wprowadź liczbę: ");
        myInput = Console.ReadLine();
        myInt = Int32.Parse(myInput);

        // Pojedyncza decyzja i akcja w nawiasach {}
        if (myInt > 0)
        {
            Console.WriteLine("Twoja liczba {0} jest większa od zera.", myInt);
        }

        // Pojedyncza decyzja i akcja bez nawiasów {}
        if (myInt < 0)
            Console.WriteLine("Twoja liczba {0} jest mniejsza od zera.", myInt);

        // Decyzja z użyciem else
        if (myInt != 0)
        {
            Console.WriteLine("Twoja liczba {0} jest różna od zera.", myInt);
        }
        else
       {
            Console.WriteLine("Twoja liczba {0} jest równa zero.", myInt);
        }


        // Decyzja z wielokrotnym użyciem else
        if (myInt < 0 || myInt == 0)
        {
            Console.WriteLine("Twoja liczba {0} jest większa lub równa zero.", myInt);
        }
        else if (myInt > 0 && myInt <= 10)
        {
            Console.WriteLine("Twoja liczba {0} jest w zakresie od 1 do 10.", myInt);
        }
        else if (myInt > 10 && myInt <= 20)
        {
            Console.WriteLine("Twoja liczba {0} jest w zakresie od 11 do 20.", myInt);
```

```
        }
        else
       {
            Console.WriteLine("Twoja liczba {0} jest większa od 20.", myInt);
        }
    Console.ReadKey();
    }
}
```

2. Konstrukcja SWITCH:

```
using System;

class SwitchSelect
{
    public static void Main()
    {
        string myInput;
        int myInt;

        Console.Write("Wprowadź liczbę z zakresu od 1 do 3: ");
        myInput = Console.ReadLine();
        // konwersja do liczby całkowitej
        myInt = Int32.Parse(myInput);

        // wybór wartości typu int
        switch (myInt)
        {
            case 1:
                Console.WriteLine("Twoja liczba to {0}.", myInt);
                break;
            case 2:
                Console.WriteLine("Twoja liczba to {0}.", myInt);
                break;
            case 3:
                Console.WriteLine("Twoja liczba to {0}.", myInt);
                break;
            default:
                Console.WriteLine("Twoja liczba {0} nie jest z zakresu od 1 do 3.", myInt);
                break;
        }
        Console.ReadKey();
    }
}
```

3 .Pętla WHILE:

```
using System;

class WhileLoop
{
    public static void Main()
    {
        int myInt = 0;

        while (myInt < 10)
        {
            Console.Write("{0} ", myInt);
            myInt++;
```

```
        }
        Console.WriteLine();
    Console.ReadKey();
    }
}
```

4. Pętla FOR

```
using System;

class ForLoop
{
    public static void Main()
    {
        for (int i=0; i < 20; i++)
        {
            if (i == 10)
                break;  // opuszczenie pętli przy i=10

            if (i % 2 == 0)
                continue;   // przejście do kolejnej iteracji dla liczb podzielenych przez 2 (liczby te nie
zostaną wypisane

            Console.Write("{0} ", i);
        }
        Console.WriteLine();
        Console.ReadKey();
    }
}
```

5. Pętla FOREACH

```
using System;

class ForEachLoop
{
    public static void Main()
    {
        string[] imiona = {"Adam", "Jakub", "Zygmunt", "Piotr"};

        foreach (string osoba in imiona)
        {
            Console.WriteLine("{0} ", osoba);
        }
    Console.ReadKey();
    }
}
```

Rozwiązania

2. Napisać program wypisujący liczby od 20 do 36 z krokiem 2, tzn. 20,22,24,..,34,36.

```
using System;

class Liczby
{
    public static void Main()
    {
        for (int i=20; i <= 36; i+=2)
```

```
        {
            Console.WriteLine("{0}", i);
        }
    Console.ReadKey();
    }
}
```

3. Napisać metodę obliczającą silnię z podanej liczby i program demonstrujący działanie metody.

Podpowiedź: silnią liczby naturalnej `n` (w notacji matematycznej `n !`, co czytamy „n silnia”) nazywamy iloczyn wszystkich dodatnich liczb naturalnych nie większych niż n.
Np. `6!=1*2*3*4*5*6=120`.

```
using System;

class Silnia
{
    static int silnia(int n)
    {
        int wynik = 1;
        for (int i = 2; i <= n; i++) wynik = wynik * i;
        return wynik;
    }

    public static void Main()
    {
        int n = 5;
        Console.WriteLine("{0}! = {1}", n, silnia(n));
        Console.ReadKey();
    }
}
```

STATUS PRZESŁANEGO ZADANIA

Status przesłanego zadania	To zadanie nie wymaga wysyłania niczego online

---

EITC/SE/CPF Laboratorium 3

Obiektowy aspekt języka C#

1. Poćwiczyć w ramach aplikacji konsolowej omówione na wykładzie elementy semantyki i składni obiektowych struktur języka C#.

2. Zaimplementować klasę Kalkulator, posiadającą jako atrybuty dwie liczby całkowite, oraz cztery metody zwracające wynik podstawowych działań arytmetycznych na tych liczbach (dodawanie, odejmowanie, mnożenie i dzielenie całkowite). Klasa powinna zawierać konstruktor inicjujący wartości liczb podane jako parametry wywołania. Napisać program demonstrujący wykorzystanie klasy.

3. Zaimplementować i uruchomić (w ramach aplikacji konsolowej) przykład z figurami podany przy omawianiu zagadnienia dziedziczenia i polimorfizmu.

4. Rozpatrzeć następującą hierarchię klas (klasa, od której wychodzi strzałka dziedziczy z klasy wskazywanej przez grot strzałki):

Dla poniższych zmiennych obiektowych:

```
burek Zwierze jakies_zwierze = new Zwierze();
burek Pies jakis_pies = new Pies();
burek Owczarek burek = new Owczarek();
burek Chihuahua reksio = new Chihuahua();
burek PajakPospolity arnold = new PajakPospolity ();
```

odpowiedzieć na pytanie, czy prawdziwe są poniższe relacje:

a. `burek is Zwierze`
b. `burek is Pies`
c. `burek is Owczarek`
d. `reksio is Owczarek`
e. `arnold is Pies`
f. `arnold is Zwierze`
g. `arnold is Owczarek`
h. `jakis_pies is Zwierze`
i. `jakies_zwierze is Pies`
j. `jakis_pies is Owczarek`

5. Zaimplementować hierarchię klas z poprzedniego zadania, uwzględniając następujące założenia:

- Klasa `Zwierze`jest klasą abstrakcyjną zawierającą atrybuty `imie`i `ilosc_lap`oraz metodę wirtualną `Przedstaw_sie()`, której realizacja w ramach tej klasy wypisuje na ekranie imię oraz ilość łap obiektu. Klasa ma posiadać konstruktor inicjujący wartości atrybutów.
- Klasa `Pies` wprowadza dodatkowy atrybut `kolor_siersci` oraz dodatkową metodę wirtualną `Szczekaj()` (która dla obiektu klasy `Pies` nie robi nic). Implementuje ona także własną wersję metody `Przedstaw_sie()` dodając do metody bazowej klasy `Zwierze` wypisywanie informacji, że obiekt jest psem oraz koloru sierści. Klasa ma posiadać konstruktor inicjujący nowy atrybut oraz przekazujący odpowiednie wartości atrybutów klasy bazowej `Zwierze` do konstruktora tej klasy.
- Klasa `Owczarek` implementuje metodę wirtualną `Szczekaj()`, wypisującą na ekranie odpowiednią wersję „szczeknięcia”. Implementuje ona także własną wersję metody `Przedstaw_sie()` dodając do metody bazowej klasy `Pies` wypisywanie informacji, że obiekt jest rasy owczarek. Klasa ma posiadać konstruktor przekazujący odpowiednie wartości atrybutów klasy bazowej `Pies` do konstruktora tej klasy.
- Klasa `Chihuahua` wprowadza dodatkowy atrybut `kolor_kokardki` oraz implementuje metodę wirtualną `Szczekaj()`, wypisującą na ekranie odpowiednią wersję „szczeknięcia” (inną niż dla klasy `Owczarek`). Implementuje ona także własną wersję metody `Przedstaw_sie()` dodając do metody bazowej klasy `Pies` wypisywanie informacji, że obiekt jest rasy chihuahua oraz koloru kokardki. Klasa ma posiadać konstruktor inicjujący nowy atrybut oraz przekazujący odpowiednie wartości atrybutów klasy bazowej `Pies` do konstruktora tej klasy.
- Klasa `PajakPospolity` (dziedziczą bezpośrednio z klasy `Zwierze`) wprowadza nową metodę `Tkaj_siec()`. Implementuje ona także własną wersję metody `Przedstaw_sie()` dodając do metody bazowej klasy `Zwierze` wypisywanie informacji, że obiekt jest pająkiem pospolitym. Klasa ma posiadać konstruktor przekazujący odpowiednie wartości atrybutów klasy bazowej `Zwierze` do konstruktora tej klasy.

  W ramach metody `Main()` stworzyć tablicę zoo obiektów klasy `Zwierze` i utworzyć oraz przypisać do kolejnych elementów tablicy obiekty klas `Owczarek`, `Chihuahua`, `PajakPospolity`. Następnie dla każdego elementu tablicy wywołać metodę `Przedstaw_się()`, zaś jeśli obiekt jest psem dodatkowo metodę `Szczekaj()` (wykorzystać rzutowanie typów).

Rozwiązania zadań znajdują się w dalszej części niniejszego pliku.

Rozwiązania

Zadanie 2.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class Kalkulator
    {
        // atrybuty klasy
        int liczba1;
        int liczba2;

        // konstruktor inicjujący wartości pól liczba1 i liczba2
        public Kalkulator(int l1,int l2)
        {
            liczba1 = l1;
            liczba2 = l2;
        }

        // metody wykonujące operacje arytmetyczne na atrybutach
        // liczba1 i liczba2
        public int Suma()
        {
            return liczba1 + liczba2;
        }
        public int Roznica()
        {
            return liczba1 - liczba2;
        }
        public int Iloczyn()
        {
            return liczba1 * liczba2;
        }
        public int Iloraz()
        {
            return liczba1 / liczba2;
        }
    }

    class Program
    {
        public static void Main()
        {
            // utworzenie obiektu kalk klasy Kalkulator z liczbami 21 i 4
            Kalkulator kalk = new Kalkulator(21, 4);
            // wypisanie wyników wywołania poszczególnych metod
            // obiektu kalk
            Console.WriteLine("Suma: {0}", kalk.Suma());
            Console.WriteLine("Różnica: {0}", kalk.Roznica());
            Console.WriteLine("Iloczyn: {0}", kalk.Iloczyn());
            Console.WriteLine("Iloraz całkowity: {0}", kalk.Iloraz());
            Console.ReadKey();
        }
    }
}
```

Zadanie 3.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class Figura
    {
        public virtual void Rysuj()
        {
            Console.WriteLine("Jestem nieokreśloną figurą.");
        }
    }
    class Linia : Figura
    {
        public override void Rysuj()
        {
            Console.WriteLine("Jestem linią.");
        }
    }

    class Okrag : Figura
    {
        public override void Rysuj()
        {
            Console.WriteLine("Jestem okręgiem.");
        }
    }

    class Kwadrat : Figura
    {
        public override void Rysuj()
        {
            Console.WriteLine("Jestem kwadratem.");
        }
    }

    class Program
    {
        public static void Main()
        {
            // 4-elementowa tablica obiektów ogólnej klasy Figura
            Figura[] figury = new Figura[4];

            // przypisanie do elementów tablicy nowych obiektów
            // klas pochodnych
            figury[0] = new Linia();
            figury[1] = new Okrag();
            figury[2] = new Kwadrat();
            figury[3] = new Figura();

            // wywołanie metody Rysuj() dla każdego elementu tablicy
            //(jako obiektu klasy Figura)
```

```
            foreach (Figura figura in figury)
            {
                figura.Rysuj();
            }
            Console.ReadKey();

        }
    }
}
```

Zadanie 4.

a. burek is Zwierze - prawda
b. burek is Pies - prawda
c. burek is Owczarek - prawda
d. reksio is Owczarek - fałsz
e. arnold is Pies - fałsz
f. arnold is Zwierze - prawda
g. arnold is Owczarek - fałsz
h. jakis_pies is Zwierze - prawda
i. jakies_zwierze is Pies - fałsz
j. jakis_pies is Owczarek - fałsz

Zadanie 5.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    // abstrakcyjna klasa Zwierze
    abstract class Zwierze
    {
        // atrybuty wspolne dla wszystkich zwierząt
        public string imie;
        public int ilosc_lap;

        // konstruktor inicjujący wartości atrybutów
        public Zwierze(string _imie, int _ilosc_lap)
        {
            imie = _imie;
            ilosc_lap = _ilosc_lap;
        }

        // wirtualna metoda wypisująca informacje wspólne dla
        // wszytskich zwierząt
        public virtual void Przedstaw_sie()
        {
            Console.WriteLine("Nazywam się: {0}. Mam {1} łapy.", imie,ilosc_lap);
        }
    }

    abstract class Pies : Zwierze
```

```
    {
        // dodatkowy atrybut specyficzny dla psów
        public int kolor_siersci;

        // konstruktor inicjujący wartość nowego atrybutu oraz
        // przekazujący odpowiednie wartości do konstruktora
        // klasy bazowej (każdy pies ma 4 łapy)
        public Pies(string _imie, int _kolor) : base(_imie,4)
        {
            kolor_siersci = _kolor;
        }

        // Zastąpienie metody wirtualnej wersją specyficzną dla psów
        public override void Przedstaw_sie()
        {
            // wywołanie wersji metody zdefiniowanej dla klasy
            // bazowej Zwierze
            base.Przedstaw_sie();
            // dodatkowy kod specyficzny dla klasy Pies
            Console.WriteLine("Jestem psem.");
        }

        // Dodatkowa wirtualna metoda abstrakcyjna specyficzna dla psów
        // (abstrakcyjna bo nie możemy zdefiniować jej zachowania dla
        // ogólnej klasy Pies(), moglibyśmy tu równie dobrze umieść słowo // virtual zamiast abstract i
zostawić puste ciało metody)
        public abstract void Szczekaj();
    }


    class Owczarek : Pies
    {
        // konstruktor przekazujący wartości inicjujące do konstruktora
        // klasy bazowej Pies
        public Owczarek(string _imie, int _kolor_siersci)
            : base(_imie, _kolor_siersci) {}

        // Zastąpienie metody wirtualnej wersją specyficzną dla psów
        // rasy Owczarek
        public override void Przedstaw_sie()
        {
            // wywołanie wersji metody zdefiniowanej dla klasy bazowej Pies
            base.Przedstaw_sie();
            // dodatkowy kod specyficzny dla klasy Owczarek
            Console.WriteLine("Jestem rasy Owczarek.");
        }

        // Zastąpienie metody wirtualnej wersją specyficzną dla psów
        // rasy Owczarek
        public override void Szczekaj()
        {
            Console.WriteLine("HAUUUUUUU!");
        }
    }

    class Chihuahua : Pies
```

```
{
    // dodatkowy atrybut specyficzny dla psów rasy Chihuahua
    int kolor_kokardki;

    // konstruktor inicjujący specyficzne atrybuty i przekazujący
    // wartości inicjujące pozostałych atrybutów do konstruktora
    // klasy bazowej Pies
    public Chihuahua(string _imie, int _kolor_siersci, int _kolor_kokardki)
        : base(_imie, _kolor_siersci)
    {
        kolor_kokardki = _kolor_kokardki;
    }

    // Zastąpienie metody wirtualnej wersją specyficzną dla psów
    // razy Owczarek
    public override void Przedstaw_sie()
    {
        // wywołanie wersji metody zdefiniowanej dla klasy bazowej Pies
        base.Przedstaw_sie();
        // dodatkowy kod specyficzny dla klasy Owczarek
        Console.WriteLine("Jestem rasy Chihuahua i mam kokardkę koloru: {0}.",kolor_kokardki);
    }

    // Zastąpienie metody wirtualnej wersją specyficzną dla psów
    // rasy Chihuahua
    public override void Szczekaj()
    {
        Console.WriteLine("Hau hau hau hau hau!");
    }
}

class PajakPospolity : Zwierze
{
    // konstruktor przekazujący odpowiednie wartości inicjujące do
    // konstruktora klasy bazowej Zwierze (pająk pospolity ma 6 łap)
    public PajakPospolity(string _imie) : base(_imie, 6) {}

    // Zastąpienie metody wirtualnej wersją specyficzną dla
    // pająków pospolitych
    public override void Przedstaw_sie()
    {
        // wywołanie wersji metody zdefiniowanej dla klasy bazowej
        // Zwierze
        base.Przedstaw_sie();
        // dodatkowy kod specyficzny dla klasy PajakPospolity
        Console.WriteLine("Jestem Pająkiem Pospolitym.");
    }

    // Dodatkowa metoda specyficzna dla pająków pospolitych
    public void Tkaj_siec()
    {
        Console.WriteLine("Właśnie utkałem sieć.");
    }
}

class Program
```

```
    {
        public static void Main()
        {
            // 4-elementowa tablica obiektów najbardziej ogólnej klasy
            // abstrakcyjnej Zwierze
            Zwierze[] zoo = new Zwierze[4];

            // utworzenie i przypisanie do elementów tablicy nowych
            // obiektów klas pochodnych
            zoo[0] = new Owczarek("Burek",1);
            zoo[1] = new PajakPospolity("Arnold");
            zoo[2] = new Chihuahua("Reksio", 5, 3);
            zoo[3] = new Owczarek("Szarik", 2);

            // wywołanie metody Przedstaw_sie() dla każdego elementu
            // tablicy (jako obiektu klasy Figura)
            foreach (Zwierze zwierze in zoo)
            {
                zwierze.Przedstaw_sie();

                // w przypadku gdy zwierzę jest psem wywołanie
                // metody Szczekaj()
                if (zwierze is Pies) ((Pies)zwierze).Szczekaj();
                // (Pies)zwierze oznacza potraktowanie obiektu
                // wskazywanego przez zmienną obiektową zwierze jako
                // obiektu klasy Pies (rzutowanie typów) – operacja ta
                // jest konieczna, gdyż wirtualna metoda Szczekaj()
                // pojawia się dopiero w klasie Pies, nie występuje
                // natomiast w klasie Zwierze

                // wypisanie linii oddzielającej
                Console.WriteLine("--------------------------------");
            }
            Console.ReadKey();
        }
    }
}
```

EITC/SE/CPF Laboratorium 4 i 5

Tworzenie aplikacji okienkowych

1. Przećwiczyć przykłady opisywane w ramach wykładu dotyczącego tworzenia aplikacji okienkowych Windows Forms.
2. Przyjrzeć się kontrolkom wylistowanym w tabeli w ramach wykładu, wstawiając je do przykładowego formularza. Przejrzeć atrybuty oraz zdarzenia poszczególnych kontrolek, próbując zmieniać wartości tych pierwszych.
3. Zaimplementować aplikację, która wykonuje operacje arytmetyczne na 2 liczbach w zależności od wybranego działania. Okno powinno zawierać następujące kontrolki:
    - dwa pola tekstowe do edycji wartości każdej z liczb oraz etykiety wskazujące na znaczenie tych pól
    - listę rozwijaną zawierającą wykaz 4 działań (dodawanie, odejmowanie, mnożenie, dzielenie oraz etykietę wskazującą na znaczenie tej listy
    - etykietę wyświetlającą wynik działania
    - przycisk powodujący obliczenie i wyświetlenie wyniku działania

Okno aplikacji powinno mieć wygląd zbliżony do następującego:

W szczególności powinny być spełnione następujące wymogi:

- pola tekstowe oraz lista rozwijana powinny być zgrupowane w obrębie panelu, którego kolor należy ustawić na żółty (atrybut `BackColor`).
- przycisk oraz etykieta wyniku powinny być zgrupowane w obrębie drugiego panelu koloru różowego
- oba pola tekstowe oraz etykieta wyniku powinny mieć wartości początkowe ustawione na 0 (atrybut `Text`)
- lista rozwijana powinna zawierać 4 elementy ("Dodawanie", "Odejmowanie", "Mnożenie", "Dzielenie) – atrybut `Item` – należy kliknąć na przycisk "..." przy wartości atrybutu i wpisać tekst każdej pozycji w nowej linii
- lista rozwijana powinna mieć ustawioną początkową wartość na "Dodawanie" (atrybut `Text`)
- etykieta wyniku powinna posiadać pogrubioną (atrybut `Font.Bold`) czcionkę wielkości 16pt (atrybut `Font.Size`)

Aplikacja powinna implementować następujące zachowanie:

- kliknięcie w przycisk powinno spowodować wyświetlenie wyniku wybranej operacji na liczbach z pól tekstowych w etykiecie wyniku.
- taki sam efekt powinno wywołać wybranie innego działania na liście rozwijanej (zdarzenie `SelectedIndexChanged`)

4. Rozszerzyć obszar okna i dodać panel koloru zielonego z przyciskiem w środku (jak na rysunku poniżej). Kliknięcie na przycisku powinno otwierać okno dialogowe wyboru koloru (`ColorDialog`), a następnie zmieniać kolor panelu na wybrany.

Wskazówki:

- kontrolkę okna dialogowego należy wstawić do formularza w taki sam sposób jak każdą inną kontrolkę – po wstawieniu nie pojawi się ona jednak na formularzu lecz w polu pod nim, ponieważ okno dialogowe nie jest widoczne przez cały czas, lecz jedynie po jego wywołaniu
- kliknięcie w przycisk zmiany koloru powinno powodować wyświetlenie okna dialogowego wyboru koloru (metoda `Show()` okna dialogowego). Po potwierdzeniu wyboru przez użytkownika kolor panelu (`BackColor`) powinien zostać zmieniony na kolor wybrany w oknie dialogowym (atrybut `Color` okna dialogowego).

- aby sprawdzić, w jaki sposób użytkownik zakończył interakcję z oknem dialogowym, należy sprawdzić wartość zwróconą przez metodę `Show()` okna dialogowego. W przypadku, gdy kliknięty został przycisk OK, zwrócona zostanie wartość `DialogResult.OK`.

5. Dodać drugi przycisk, po którego naciśnięciu:

- otwarte zostanie okno dialogowe wyboru pliku (`OpenFileDialog`)
- po wskazaniu przez użytkownika dowolnego pliku zawierającego obraz graficzny (.jpeg, .bmp, .gif) otwarte zostanie okno drugiego formularza (należy do aplikacji dodać nowy formularz) wyświetlające w całym swoim obszarze obraz wczytany z pliku (kontrolka `PictureBox`) zeskalowany do rozmiarów okna.

Wskazówki:

- wywołanie okna dialogowego `OpenFileDialog` i sprawdzanie akcji użytkownika przebiega analogicznie do okna `ColorDialog`
- pełna nazwa wybranego w oknie dialogowym pliku wraz ze ścieżką dostępu dostępna jest w atrybucie `FileName` okna dialogowego
- aby wczytać obraz z pliku do kontrolki `PictureBox` należy do wartości atrybutu `ImageLocation` kontrolki przypisać nazwę pliku odczytaną z atrybutu `FileName` okna dialogowego, a następnie wywołać metodę `Load()` kontrolki
- aby rozmiar kontrolki `PictureBox` zawsze obejmował cały obszar formularza i zmieniał się wraz ze zmianą jego rozmiarów należy ustawić atrybut `Dock` kontrolki na `Fill`.
- aby rozmiar obrazka był skalowany do rozmiarów kontrolki `PictureBox` należy ustawić atrybut `SizeMode` kontrolki na `StretchImage`

UWAGA: aby możliwy był dostęp do atrybutów kontrolki `PictureBox` znajdującej się w klasie `Form2` z poziomu klasy `Form1` (tu znajduje się metoda obsługi przycisku) konieczne jest zmienienie klasyfikatora dostępu do atrybutu `pictureBox1` klasy `Form2` z `private` (domyślna wartość dla wszystkich wstawianych do formularza kontrolek) na `public`. Atrybuty odpowiadające wstawionym do formularza kontrolkom zdefiniowane są w osobnym pliku o rozszerzeniu `.Designer.cs` (tutaj `Form2.Designer.cs`), w którym znajduje się część kodu aplikacji w całości generowana automatycznie przez projektanta formularzy. Interesujący nas wpis znajduje w ostatnim wierszu definicji klasy:

```
        public System.Windows.Forms.PictureBox pictureBox1;
    }
}
```

Kod źródłowy końcowej postaci aplikacji z zadania 5 (modyfikowane pliki):

Plik Form1.cs:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WindowsApplication1
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            double l1, l2, wynik;
            l1 = Convert.ToDouble(textBox1.Text);
            l2 = Convert.ToDouble(textBox2.Text);
            wynik = 0;
            switch (comboBox1.SelectedIndex)
            {
                case 0: wynik = l1 + l2; break;
                case 1: wynik = l1 - l2; break;
                case 2: wynik = l1 * l2; break;
                case 3: wynik = l1 / l2; break;
            }
            label4.Text = wynik.ToString();
        }
```

```
        private void button2_Click(object sender, EventArgs e)
        {
            if (colorDialog1.ShowDialog() == DialogResult.OK)
                panel3.BackColor = colorDialog1.Color;
        }

        private void button3_Click(object sender, EventArgs e)
        {
            Form2 f = new Form2();
            if (openFileDialog1.ShowDialog() == DialogResult.OK)
            {
                f.pictureBox1.ImageLocation = openFileDialog1.FileName;
                f.Show();
            }
        }

    }
}
```

Plik Form2.Designer.cs z zaznaczoną ręczną zmianą:

```
namespace WindowsApplication1
{
    partial class Form2
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code

        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.pictureBox1 = new System.Windows.Forms.PictureBox();
            ((System.ComponentModel.ISupportInitialize)(this.pictureBox1)).BeginInit();
            this.SuspendLayout();
            //
            // pictureBox1
```

```
            //
            this.pictureBox1.Dock = System.Windows.Forms.DockStyle.Fill;
            this.pictureBox1.Location = new System.Drawing.Point(0, 0);
            this.pictureBox1.Name = "pictureBox1";
            this.pictureBox1.Size = new System.Drawing.Size(284, 264);
            this.pictureBox1.SizeMode = System.Windows.Forms.PictureBoxSizeMode.StretchImage;
            this.pictureBox1.TabIndex = 0;
            this.pictureBox1.TabStop = false;
            //
            // Form2
            //
            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(284, 264);
            this.Controls.Add(this.pictureBox1);
            this.Name = "Form2";
            this.Text = "Obraz";
            ((System.ComponentModel.ISupportInitialize)(this.pictureBox1)).EndInit();
            this.ResumeLayout(false);

        }

        #endregion

        public System.Windows.Forms.PictureBox pictureBox1;
    }
}
```

STATUS PRZESŁANEGO ZADANIA
