MPDI

Wykład 12

Matlab

Inne typy danych

Programowanie obiektowe

Interfejs komponentowy w Matlabie

Typy danych

Poznaliśmy…

Liczbowe:

x=4.6 rzeczywiste

y= 3+4i zespolone

Tekstowe (łańcuchowe):

x= 'Politechnika'

Tablicowe (homogeniczne):

x= [1 2; 4 5]

y= ['a' 'b'; 'c' 'd']

ten sam typ danych w każdym elemencie

Logiczne:

x=true

y=false

w Matlabie logical 0 i logical 1

Tablice komórkowe (ang. cell - heterogeniczne)

Tablica, w której każdy element **może być innego typu**, w tym także typu złożonego.

Zawartość takiej tablicy wpisujemy podobnie jak w tablicach homogenicznych lecz w nawiasach klamrowych { } (nie prostokątnych).

```
oceny ={'fizyka', 3.5; 'informatyka', 4.0; 'mechanika',5.0}

przedmiot=oceny{1,1}
```

```
oceny =
    'fizyka'        [3.5000]
    'informatyka'   [     4]
    'mechanika'     [     5]

przedmiot =
            'fizyka'
```

Tablice struktur (ang. struct)

Jej elementami są **dane** złożone z wielu elementów przechowujących wartości różnego typu i identyfikowanych przez swoje **nazwy**.

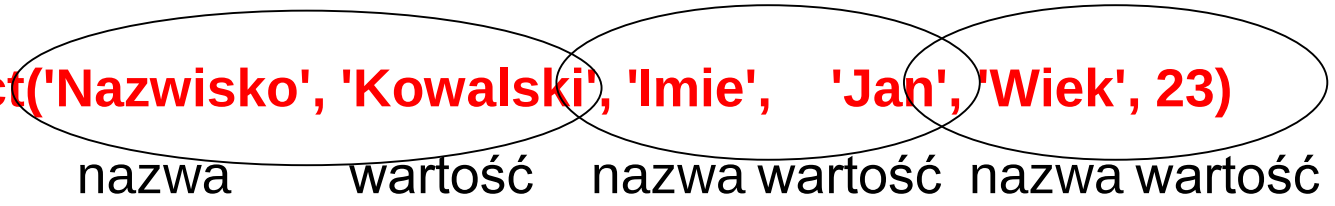
W innych językach, w tym w językach obsługi baz danych, noszą nazwę **typu rekordowego**

Tablice struktur można definiować na dwa sposoby:

a) Definiowanie z użyciem funkcji **struct**

Przykład:

student = struct('Nazwisko', 'Kowalski', 'Imie', 'Jan', 'Wiek', 23)



b) Definiowanie przez przypisywanie wartości kolejnych pól. Nazwy pól oddzielamy kropką od nazwy tablicy lub jej elementu:

student.Nazwisko='Kowalski'

student.Imie='Jan'

W ten sposób możemy zarówno definiować pola nowej struktury jak i dodawać nowe pole do już istniejącego rekordu:

student.Grupa=8

Typ zbiorowy

Typ zbiorowy to zbiór potęgowy danego typu porządkowego, czyli zbiór wszystkich podzbiorów tego typu. **Zmienna typu zbiorowego** może zatem zawierać zbiór pusty, jedno- lub wieloelementowy.

Zmienna typu zbiorowego zawierać może **dowolny podzbiór** elementów typu bazowego, od zbioru pustego do zbioru zawierającego wszystkie elementy.

W innych językach programowania występują typy danych:

Typ wyliczeniowy

Konieczna jest definicja typu

```
dni_tygodnia=(pn, wt, sr, cz, pi, so, ni)
```

I definicja zmiennej:

```
dzien: dni_tygodnia
```

zmienna **dzien** należy do typu **dni_tygodnia**

Teraz w programie można używać zmiennej x:

```
dzien = wt
```

Wartość *wt* ma charakter abstrakcyjny (umowny)

Zmienna **dzien** przyjmuje **jedną** (!) wartość ze zbioru dopuszczalnych wartości (nie może innej spoza wyliczonych)

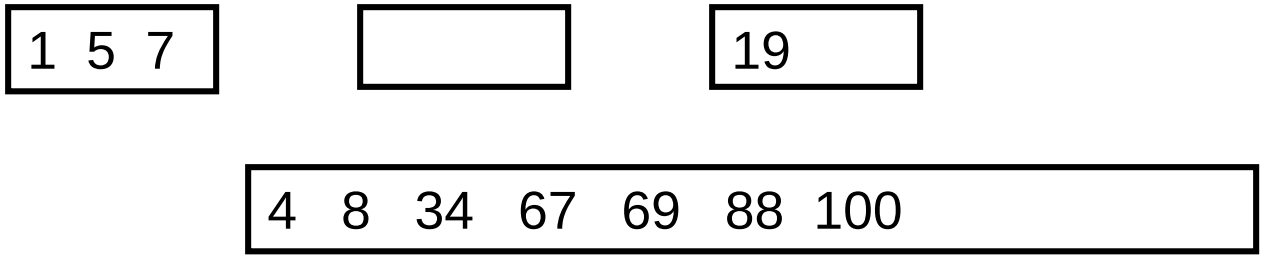
Typ zbiorowy

Zmienna typu **liczbowego** całkowitego może mieć wartość 1 lub 100 a nie może jednocześnie 1 i 100!

Zmienna typu **tablicowego** może zawierać wiele wartości i nawet jak komórki są puste, to rezerwują pamięć komputera

Zmienna typu zbiorowego może zawierać zero (zbiór pusty), jedną lub wiele wartości z danego zbioru

Zmienne typu zbiorowego



Typ zbiorowy może być oparty na typie wyliczeniowym

Przykłady:

Definicja typu | dni_tygodnia=(pn, wt, sr, cz, pi, so, ni)

dzien = wt — zmienna typu wyliczeniowego

weekend= (so, nie) — zmienna typu zbiorowego

miesiac = luty — zmienna typu wyliczeniowego

kwartal1= (styczen, luty, marzec) — zmienna typu zbiorowego

UWAGA:

Zmienna *miesiac* może przyjąć wartość tylko jednej z wymienionych – typ wyliczeniowy

Zmienna *typu zbiorowego* może przyjąć wartość dowolnego podzbioru z nazw bazowych – typ zbiorowy

Operacje logiczne wykonywane na zbiorach to relacje (porównania):

A = B równość zbiorów, te same elementy w obu zbiorach,

A < > B różność zbiorów, różne elementy w obu zbiorach (chociaż niektóre mogą się powtarzać)

A <= B **zawieranie** zbioru *A* w zbiorze *B* (wartość logiczna -*true* jeśli <u>każdy</u> element zbioru *A* jest w zbiorze *B*)

A >= B **zawieranie** zbioru *B* w zbiorze *A* (wartość logiczna - *true* jeśli <u>każdy</u> element zbioru *B* jest w zbiorze *A*)

c in A ma wartość logiczną - czy element **c** jest w zbiorze *A*

Wartość logiczną sprawdzamy oczywiście wykorzystując instrukcję warunkową ***if***

Przykład: Czcionka w aplikacjach Windows

definicja zmiennych (język *Pascal*)

x, y , z: **set of** (pochylony, pogrubiony, podkreslony);

wykorzystanie

x= [ ] //zbiór pusty

y= [pochylony, podkreslony]

z= [pochylony, pogrubiony]

Operatory działań na zmiennych typu zbiorowego

(znaczenie jak w **teorii mnogości**):

- **+** suma zbiorów (jeśli elementy się dublują, to w wyniku sumy występują raz)
- **–** różnica zbiorów (z pierwszego zbioru usuwamy elementy drugiego zbioru)
- * iloczyn zbiorów (**elementy wspólne!!!**)

```
x =(2, 3, 4) + (4, 5, 6)     %wynik      (2,3,4,5,6)
y = (2, 3, 4) – (4, 5, 6)    %wynik      (2,3)
v = (2, 3, 4) * (4, 5, 6)    %wynik      (4)      (zbiór wspólny)
y = v * x ;                  %przeanalizować wynik
```

W *Matlabie* istnieją funkcje działające na wektorach wykonujące odpowiednie działania jak w teorii zbiorów

```
A = [2,3,5,7] % wektor- zbiór A
B = [1,3,3] % wektor - zbiór B
x=union(A,B) % x to [1,2,3,5,7]
y=setdiff(A,B) % y to [2,5,7]
z=intersect(A,B) % z to [3]
v=ismember(4,A) % v to logical 0
```

suma

różnica

iloczyn

czy 4 jest w zbiorze

Podstawowe cechy programowania obiektowego

Programowanie strukturalne (proceduralne) – koncepcja tradycyjna.

Główną jego składową są instrukcje działające na danych.
Złożone programy korzystają z podprogramów (**funkcji, procedur**, a także **modułów** grupujących podprogramy) w celu uproszczenia zarządzania i kontroli nad programem.

Wady programowania strukturalnego

- dane są powszechnie dostępne – łatwo o błędy,
- sekwencyjność wykonywania programu,
- wszystkie sytuacje trzeba przewidywać i obsługiwać,
- konieczność testowania po każdej zmianie,
- wiele instrukcji, obszerny kod, trudność zrozumienia algorytmu

Zauważono „Kryzys oprogramowania” – programowanie strukturalne utrudnia panowanie nad bardzo złożonymi systemami informatycznymi SI (rozwój sprzętu wyprzedzał techniki budowania SI).

Potrzebne były metody **zwiększające wydajność i systematyczność** tworzenia SI,a następnie ich wydajność.

Poza tym powstały interfejsy graficzne (Windows)!

Korzenie technologii obiektowej – lata 60-te, Nygaard i Dohl, Simula 1, Simula67 (1967).

OBIEKTOWOŚĆ – filozofia tworzenia na podstawie rzeczywistych zjawisk otaczającego świata (nie tylko język programowania).

Obiekty (świata rzeczywistego a także systemu operacyjnego komputera – plik, ikona, przycisk, okno) – mają:
właściwości (nazwa, kolor itp.),
a także
zbiory operacji na nich czy przez nie wykonywanych, które są inicjowane jako reakcja na zdarzenia

Stworzono tzw. ADT – **abstrakcyjny typ danych** – podążanie w kierunku naturalnego języka (zbliżenie do rzeczywistości), nazwano **modułem** (język Modula) lub **klasą** (język Simula).

System reaguje na zdarzenia („siły sprawcze”), efektem są procesy:

- funkcje przetwarzania parametrów obiektów
- przesyłu informacji między obiektami
- oddziaływania jednych obiektów na inne

PROGRAMOWANIE OOP – podstawowe pojęcia

Programowanie zorientowane obiektowo (OOP – Object Oriented Programming) umożliwia przedstawienie problemu w postaci logicznie powiązanych ze sobą struktur danych zwanych obiektami, wymieniających informacje między sobą.

„**Obiektowość**” opiera się na koncepcyjnym (intuicyjnym) klasyfikowaniu rzeczywistości.
Na świat składają się obiekty i procesy w nich zachodzące.

Koncepcje (pojęcia)

KLASA = **typ obiektowy=encja (entity)**

OBIEKT = instancja w klasie = reprezentacja w klasie, element przechowujący dane

Podobnie jak typ zmiennej i zmienna

Klasa (typ obiektowy) jest to złożona struktura danych o określonej liczbie atrybutów.

Atrybuty klasy dzielimy na:

- **pola (właściwości)**
- **metody (funkcje)**

pola (fields) – atrybuty (właściwości opisane wartościami dowolnych typów, także strukturalnych)
Pole jest to zmienna o wartości będącej różnego typu.

metody (methods) – funkcje wykonywane na polach.
Metoda jest czynnością wykonywaną na obiekcie w postaci podprogramu (funkcji). Metoda obiektu operuje na polach (danych) obiektu, przy ich pomocy mamy dostęp do pól.

Czyli można powiedzieć, że typ obiektowy to typ rekordowy poszerzony o metody

Metoda jest to funkcja mająca deklarację w ramach typu obiektowego.

Nazwa jest **kwalifikowana**, tzn. wskazuje na obiekt, którego dotyczy i ma postać:

nazwaObiektu.nazwaMetody(argumenty)

.. identycznie jak w strukturach (czy rekordach)

Konstruktor – specjalna metoda używana przy tworzeniu (instancji) obiektu danej klasy – zmienna typu obiektowego

Destruktor – specjalna metoda wywoływana automatycznie tuż przed zakończeniem istnienia obiektu (niszczenie obiektu)

Przykład definicji typu obiektowego – plik Pies.m

```
classdef Pies<handle
    properties % właściwości
        imie='' %puste – nadawane przy pomocy metody
        nogi=4;
        ileHau = 0;
    end
   methods % metody
         function obj = Pies()
            % działania inicjacyjne
        end
        function nadajImie(obj,x)
              obj.imie=x;
        end
        function Hau(obj)
            fprintf('hauuuuuuuuuu');
            obj.ileHau = obj.ileHau + 1;
            fprintf( ' %d raz\n' ,obj.ileHau)
        end
        function ileNog(obj)
            disp(obj.ileNog);
        end

    end
end
```

Na bazie powyższej definicji można napisać nasz m-plik:

```
clc,clear
pies1=Pies(); %utworzenie obiektu
pies1.nadajImie('Burek');
pies1.Hau(); %wykonanie metody
pies1.Hau();
pies1.Hau();
fprintf('Ma nóg:%d\n',pies1.nogi);
pies2=Pies(); %utworzenie innego obiektu
pies2.nadajImie('Ciapek');
for k=1:5
        pies2.Hau(); %wykonanie metody
end
for i=1:5
        Hau(pies2); %inny sposób
end
fprintf('%s szczeknął %d razy\n',pies1.imie,pies1.ileHau);
fprintf('%s szczeknął %d razy\n',pies2.imie,pies2.ileHau);
```

Inny przykład

```
%definicja klasy
classdef Tczlowiek<handle
    properties % właściwości
        waga;
    end
    methods % metody
        function obj = Tczlowiek()
            % działania inicjacyjne
        end
        function jedz(obj, posilek)
             obj.waga=obj.waga+posilek/20;
        end
    end
end
```

```
clc,clear
osoba1=Tczlowiek( ); %utworzenie obiektu
osoba1.waga=90
osoba1.jedz(10)
osoba1.waga
osoba1.jedz(10)
osoba1.waga
```

aktywowanie metody **jedz** wpłynie na wagę obiektu osoba1, należącego do klasy T**czlowiek**

Przykład oddziaływania obiektów

```
classdef Liczba<handle
    properties % wlasciwosci
        wartosc=0;
        nr=0;
    end
   methods % metody
         function obj = Liczba(x,y)
            obj.nr=x;
            obj.wartosc=y;
        end
        function dodaj(obj,inny)
              obj.wartosc=obj.wartosc+inny.wartosc;
        end
        function wypisz(obj)
            fprintf('Liczba numer %d = %d \n',obj.nr, obj.wartosc);
        end
    end
end
```

```
clc,clear
L1=Liczba(1, 33); %utworzenie obiektu L1
L1.wypisz();
L2=Liczba(2, 7); %utworzenie obiektu L2
L2.wypisz();
L1.dodaj(L2) ;%dodanie obiektu L2 do L1
L1.wypisz();
```

Cechy obiektowości

- hermetyzacja
- polimorfizm
- dziedziczenie

HERMETYZACJA

Własność polegająca na dostępie do pól jedynie przy użyciu metod nazywa się hermetyzacją.

Tworzy to dyscyplinę programowania, w jednym miejscu mamy dane i dozwolone operacje na nich.

Ułatwia kontrolę poprawności złożonych programów.

DZIEDZICZENIE

Typ obiektowy może on być:

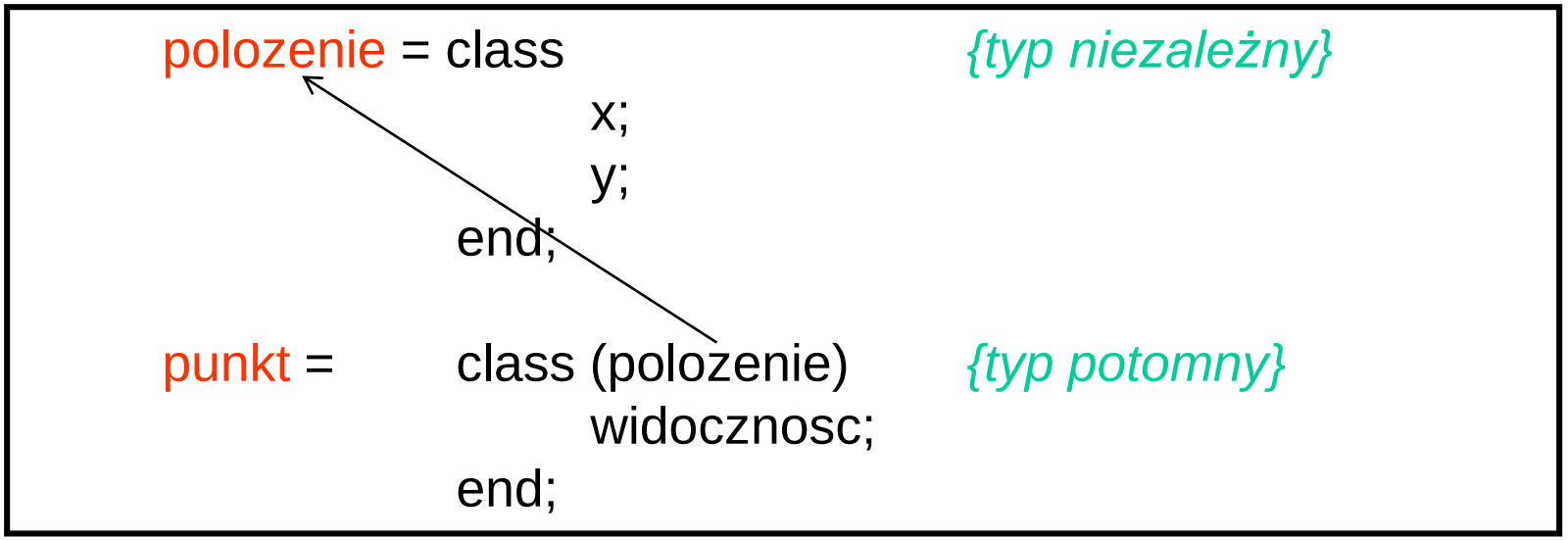
- **niezależny**, (zdefiniowany podobnie jak typ rekordowy) – rodzic drzewa
- jako potomek istniejącego. Wtedy mówimy, że obiekt dziedziczy wszystkie elementy (pola i metody) swojego przodka lub jest typem **potomnym**.

  Obiekty potomne mogą mieć własnego potomka (lub wielu).

Wzajemne zależności obiektów układają się w **drzewo hierarchii obiektów**.

Dziedziczność bezpośrednia – przodek jest niezależny (sam nie ma przodka)
Dziedziczność pośrednia – przodek już dziedziczy (sam ma przodka)

Przykład dziedziczenia klas:



POLIMORFIZM

Potomek może mieć **tę samą nazwę metody** jak przodek, „przykrywa” ona wówczas metodę przodka. Definiując metodę potomka (rozwijając ją lub modyfikując, np. gdy jest ona rozszerzeniem metody nadrzędnej), można odwołać się do metody przodka.
Jest to tzw. POLIMORFIZM

Polimorfizm (wielopostaciowość) - wykorzystanie tzw. metod wirtualnych.

TYPY DYNAMICZNE

Cel podstawowy – oszczędność pamięci, w języku Matlaba możemy zmieniać wielkość elementu

Np. tablice są dynamiczne – możemy zmieniać rozmiary):

```
M=[1 1;2 2]

M(3,3)=1;

M
```

```
1   1
2   2

1    1    0
2    2    0
0    0    1
```

zwiększyliśmy rozmiar tablicy – w innych językach może to być niemożliwe bez stosowania typów dynamicznych (wskaźnikowych)

Struktury w Matlabie też są dynamiczne – możemy dodawać nowe pola

ZASADY

Do zbioru (uporządkowanego według określonej metody) **możemy dołączać nowe elementy**.

Rozmiar elementu nie jest zdefiniowany, każde dołączanie nowego elementu powoduje nową **rezerwację pamięci**.

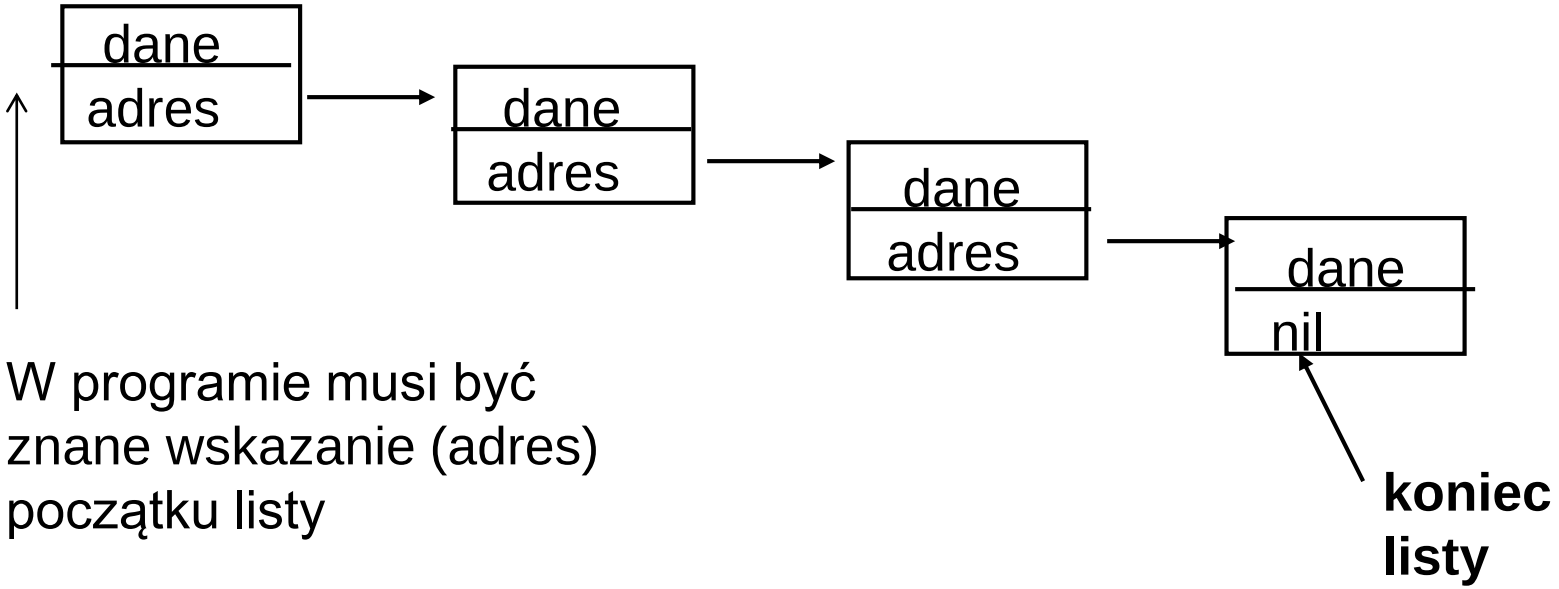
Pobieranie elementu, jego usuwanie, dołączanie nowego elementu – mogą być obarczone pewnymi kryteriami dostępu.

W językach programowania pozwalających na dostęp do pamięci stosuje się tzw. **zmienne wskaźnikowe** – wskaźnik jest logicznym adresem w pamięci

Zwykle istnieje też specjalny symbol, który określa wskazanie jako puste.
W językach C jest to **NULL**, w C++ **nullptr**, w Pascalu **nil**

Rodzaje typów dynamicznych

Lista jednokierunkowa

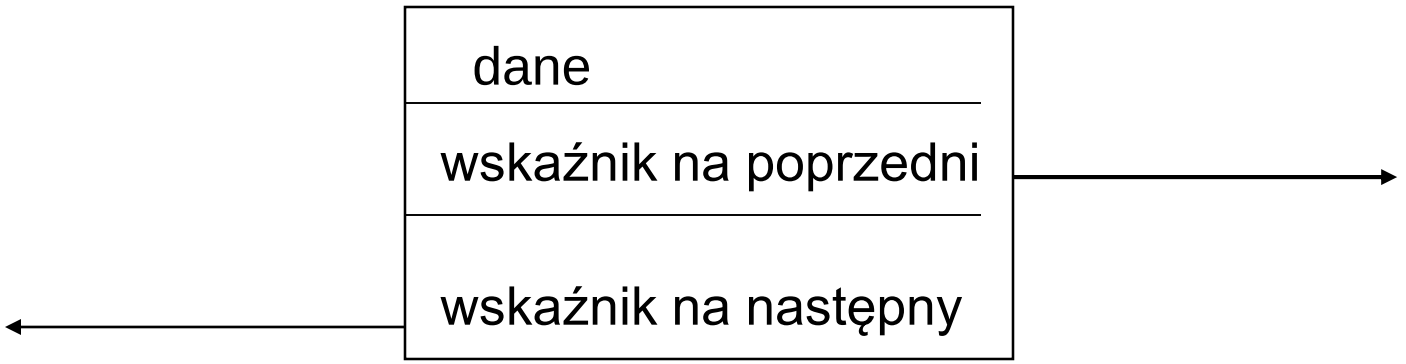


Lista jednokierunkowa może być "w przód" lub "wstecz", czyli element może zawierać wskaźnik na następny lub poprzedni element.

Lista dwukierunkowa

Jest to liniowo uporządkowany zbiór składników, w którym dla każdego składnika (poza pierwszym i ostatnim), jest określony składnik poprzedni i następny.

Dla ostatniego składnika listy dwukierunkowej jest określony tylko składnik poprzedni, a dla pierwszego tylko następny.



Stos (stack)

Jest to struktura danych, składająca się z liniowo uporządkowanych zbiorów składników (elementów), z których tylko ostatnio dołączony jest w danej chwili dostępny.

Miejsce dostępu to wierzchołek stosu. Jest to jedyne miejsce, do którego można dołączyć lub z którego można usunąć elementy.

FILO lub LIFO

first in – last out

last in - first out

Kolejka (queue)

Jest t struktura danych, składająca się z liniowo uporządkowanych zbiorów składników, do której można dołączyć składnik tylko na jednym końcu (koniec kolejki), a usunąć tylko w drugim końcu (początek kolejki).

Powiązanie między składnikami kolejki jest takie samo jak pomiędzy składnikami stosu.

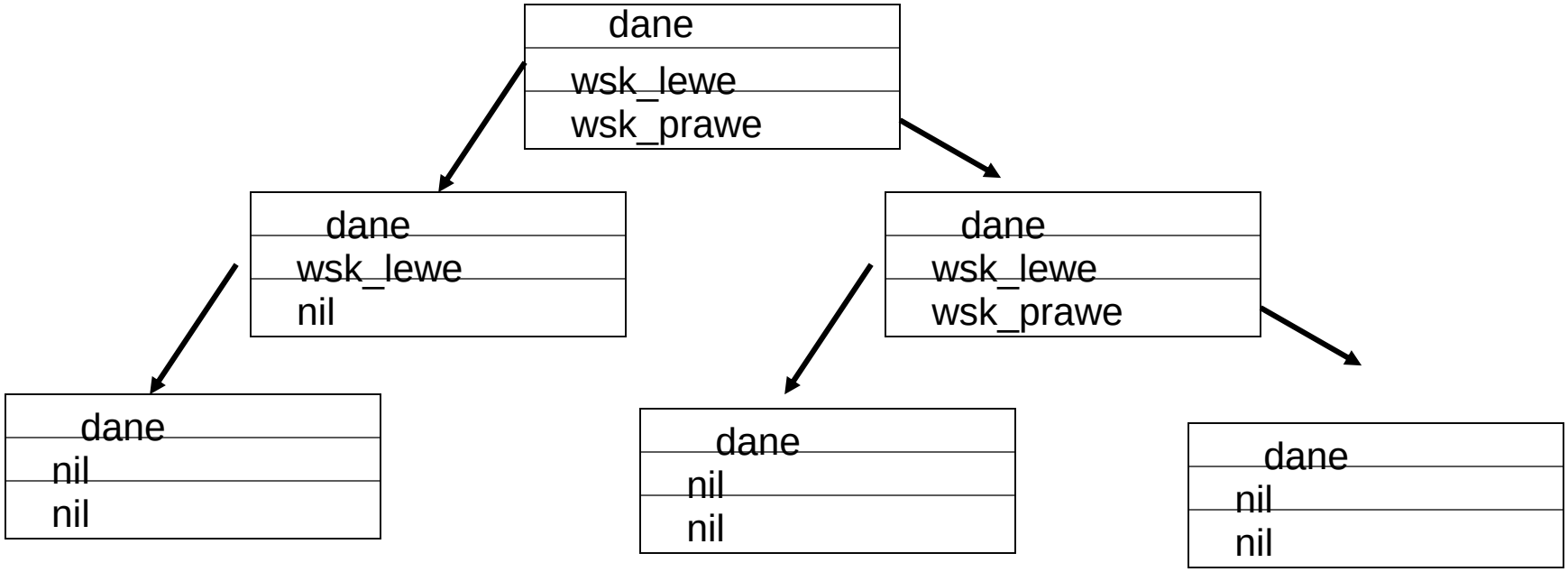
FIFO

first in – first out

Drzewo binarne

jest strukturą danych, składającą się nieliniowo uporządkowanych zbiorów składników.

Do każdego składnika można dołączyć jeden lub dwa składniki. Drzewo ma swój "korzeń" z którego wyrasta struktura.

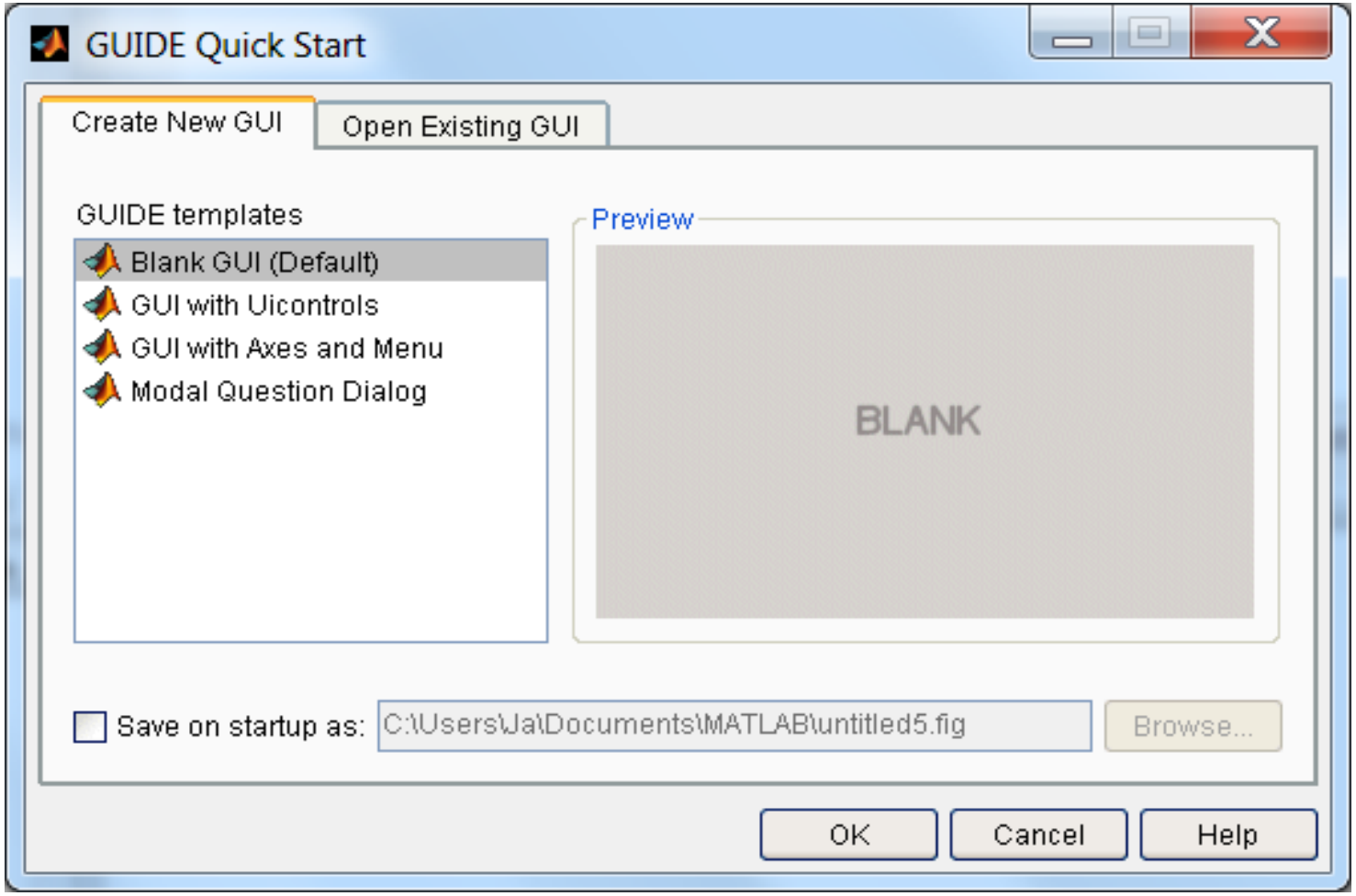


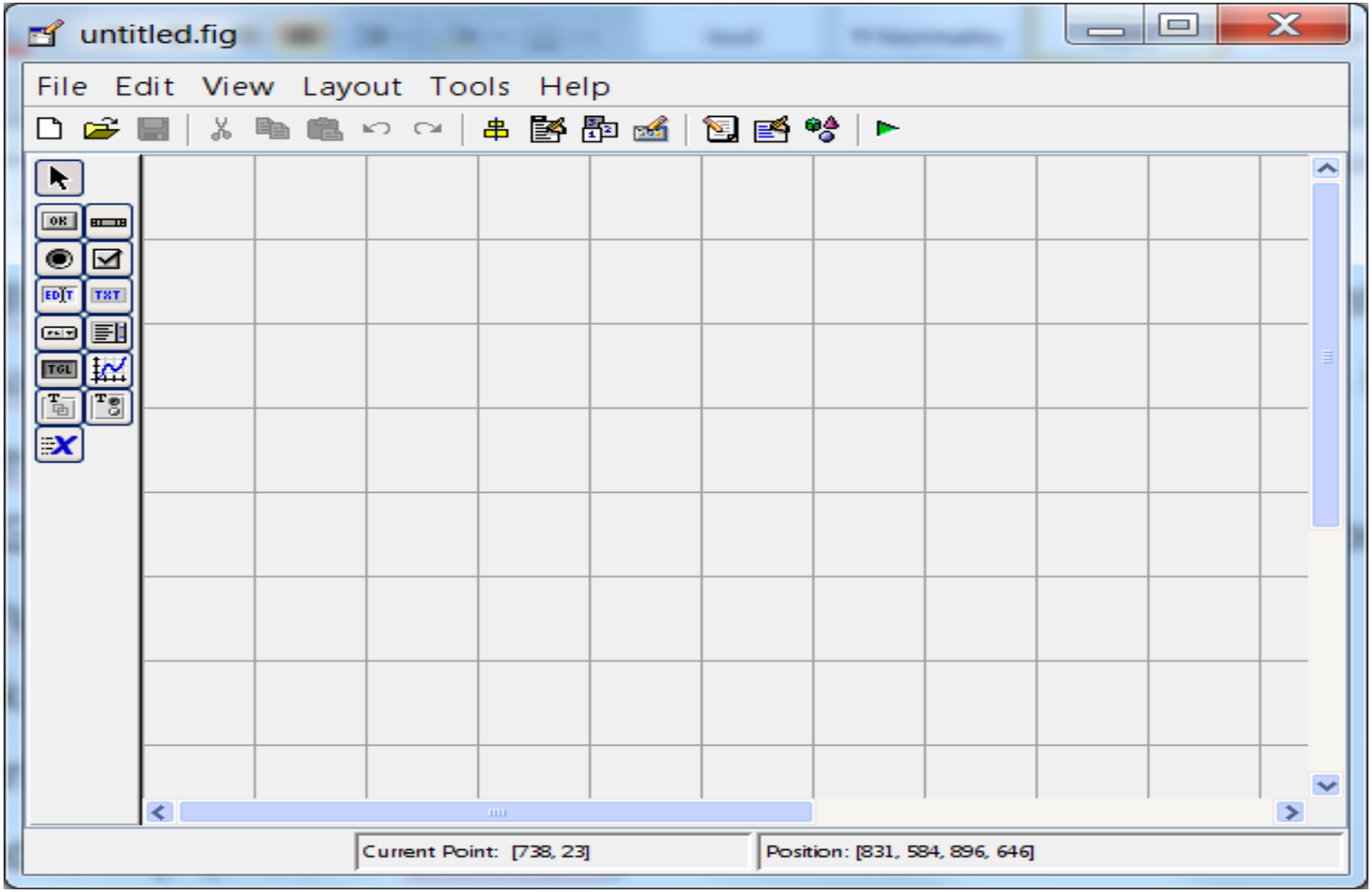
Drzewo binarne, w którym liczba synów każdego wierzchołka wynosi albo zero albo dwa, nazywane jest drzewem regularnym

Aplikacja obiektowa w Matlabie

Wpisujemy w *Command Window*:

>>guide

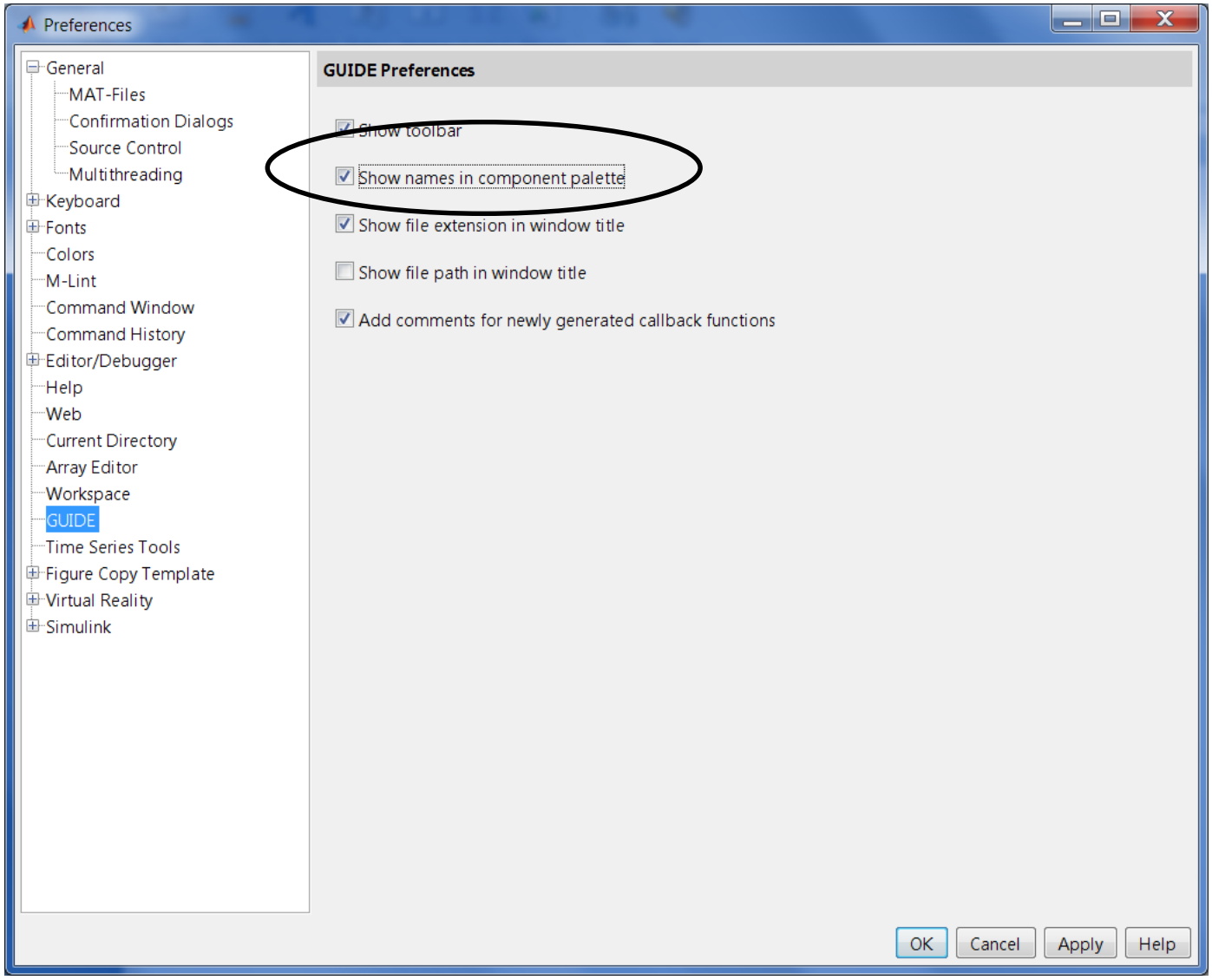


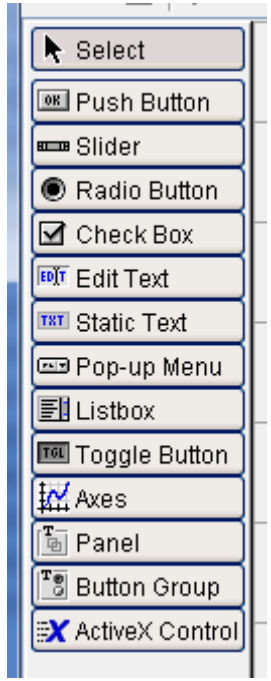


Okno pustego formularza

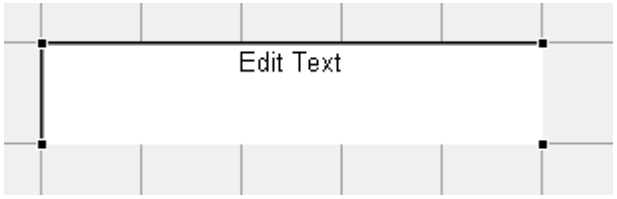
Mamy do wyboru następujące komponenty

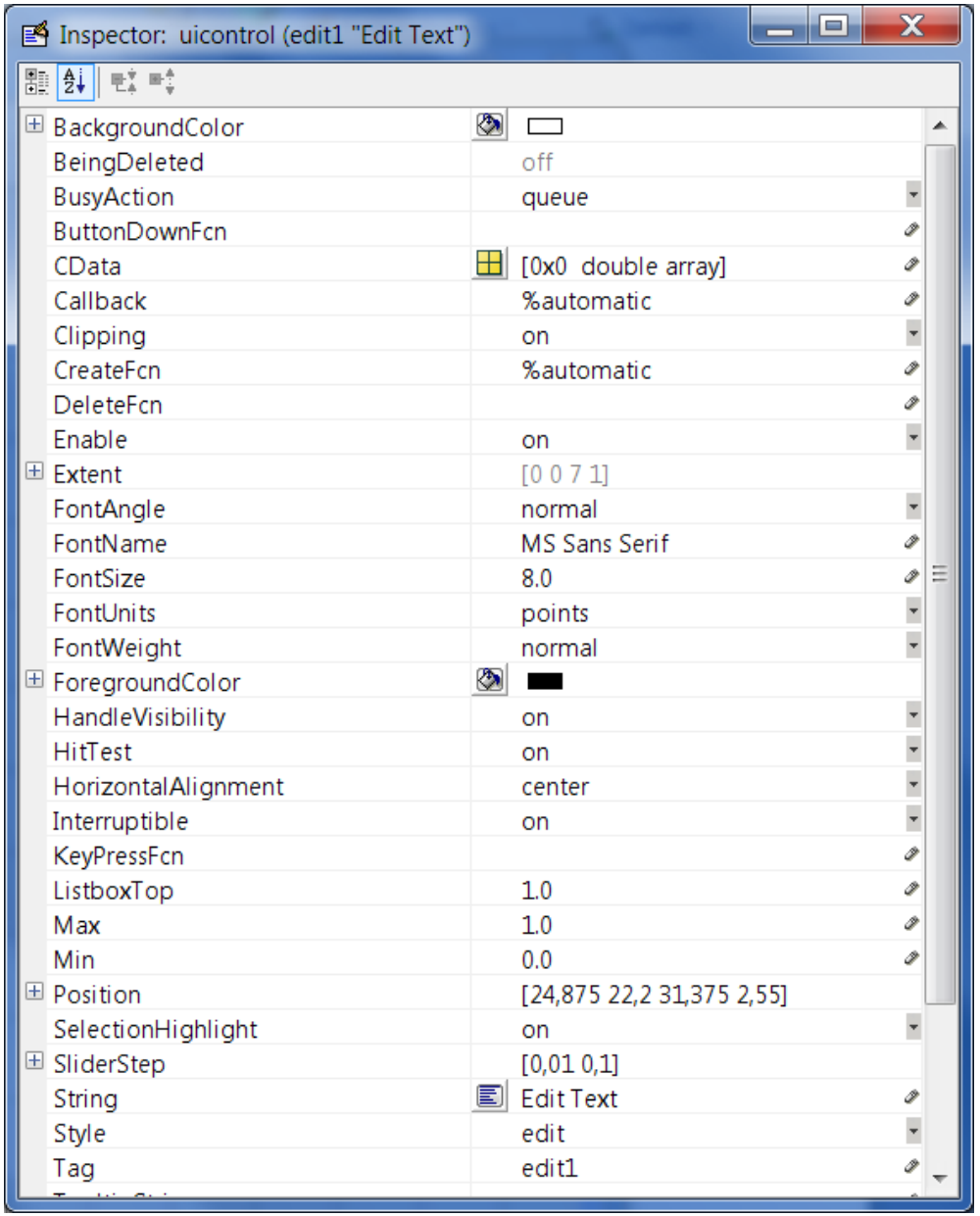
- Przycisk (*Push Button*)
- Suwak (*Slider*)
- Przycisk radiowy (*Radio Button*)
- Pole wyboru (*Checkbox*)
- Pole tekstowe (*Edit Text*)
- Etykieta (*Static Text*)
- Lista rozwijana (*Popup Menu*)
- Lista wyboru (*Listbox*)
- Przełącznik (*Toggle Button*)
- Wykres (Axes)
- Panel
- Grupa przycisków (*Button group*)
- Komponent ActiveX

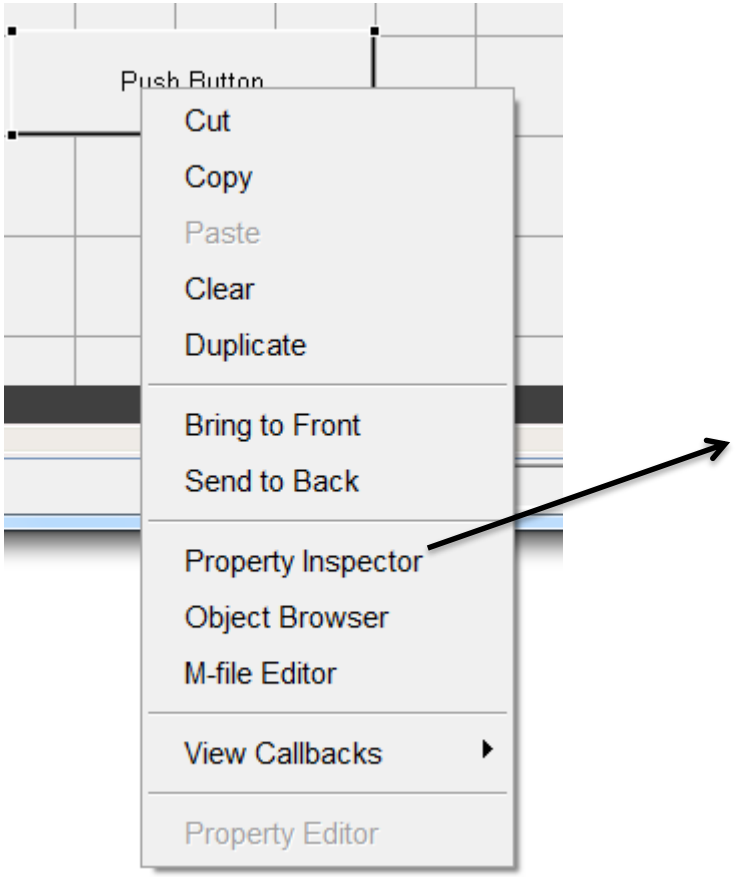

Preferences
General
MAT-Files
Confirmation Dialogs
Source Control
Multithreading
Keyboard
Fonts
Colors
M-Lint
Command Window
Command History
Editor/Debugger
Help
Web
Current Directory
Array Editor
Workspace
GUIDE
Time Series Tools
Figure Copy Template
Virtual Reality
Simulink
GUIDE Preferences
Show toolbar
Show names in component palette
Show file extension in window title
Show file path in window title
Add comments for newly generated callback functions
OK
Cancel
Apply
Help


Select
Push Button
Slider
Radio Button
Check Box
Edit Text
Static Text
Pop-up Menu
Listbox
Toggle Button
Axes
Panel
Button Group
ActiveX Control

Każdy komponent (obiekt) wprowadzony w okienku aplikacji ma ***Property Inspector*** (po dwukliknięciu danego elementu lub wybierany z menu kontekstowego), w którym można ustawić podstawowe informacje o danym komponencie na starcie aplikacji.







Inspector: uicontrol (pushbutton1 "Push Button")

Enable	on
Extent	[0 0 9,5 1]
FontAngle	normal
FontName	MS Sans Serif
FontSize	8.0
FontUnits	points
FontWeight	normal
ForegroundColor	
HandleVisibility	on
HitTest	on
HorizontalAlignment	center
Interruptible	on
KeyPressFcn	
ListboxTop	1.0
Max	1.0
Min	0.0
Position	[33,375 6,2 22,75 2,65]
SelectionHighlight	on
SliderStep	[0,01 0,1]
String	Push Button
Style	pushbutton
Tag	pushbutton1
TooltipString	
UIContextMenu	<None>
Units	characters
UserData	[0x0 double array]

W oknie *Property Inspector* można ustawić podstawowe informacje o danym komponencie (liczby, teksty, stałe)

Ważniejsze właściwości komponentów

String – napisy (na przycisku PushButton, , tekst w Edit, napisy na etykietach Static Text
Tag – nazwa komponentu w programie
FontSize – rozmiar czcionki
Position - położenie komponentu na formatce
Color lub **BackgroundColor** – kolorystyka
Value – wartość liczbowa, np. położenie suwaka
Units – jednostki miary

Funkcjonalność komponentu

Po zapisaniu pliku (np. wykresy.*fig)* z projektem formatki, tworzony jest też **m-plik** o identycznej nazwie i otwierany w edytorze *Matlaba*.

W m-pliku jest wiele funkcji dotyczących projektu i jego komponentów

Po zapisie aplikacji mamy dwa pliki:

nazwa.fig

nazwa.m

Aby zdefiniować działanie należy w m-pliku opisać działanie funkcji ***Callback*** dla danego komponentu

Funkcje **Callback**

Wpisane wewnątrz nich instrukcje wykonywane są:

function pushbutton1_Callback – po naciśnięciu przycisku

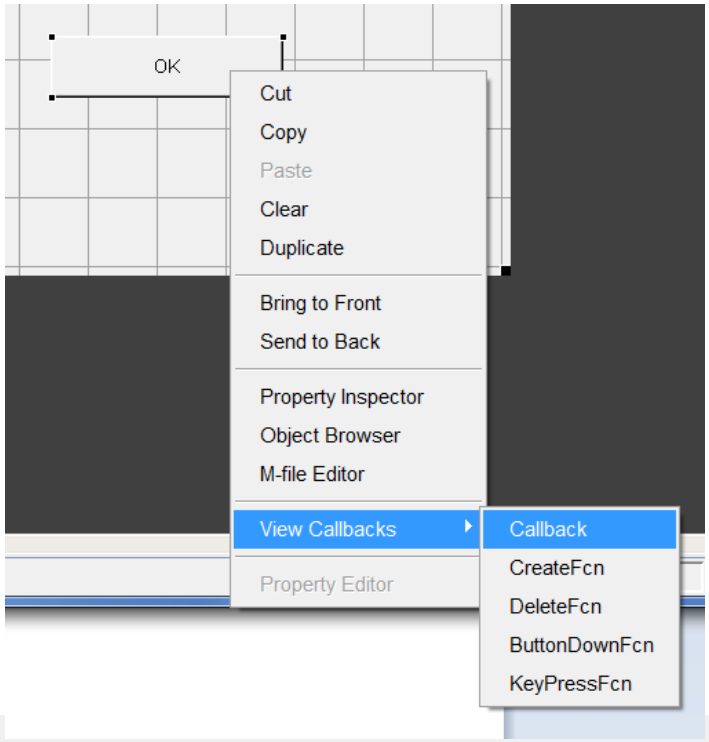
function edit1_Callback – po wpisaniu tekstu i naciśnięciu ENTER

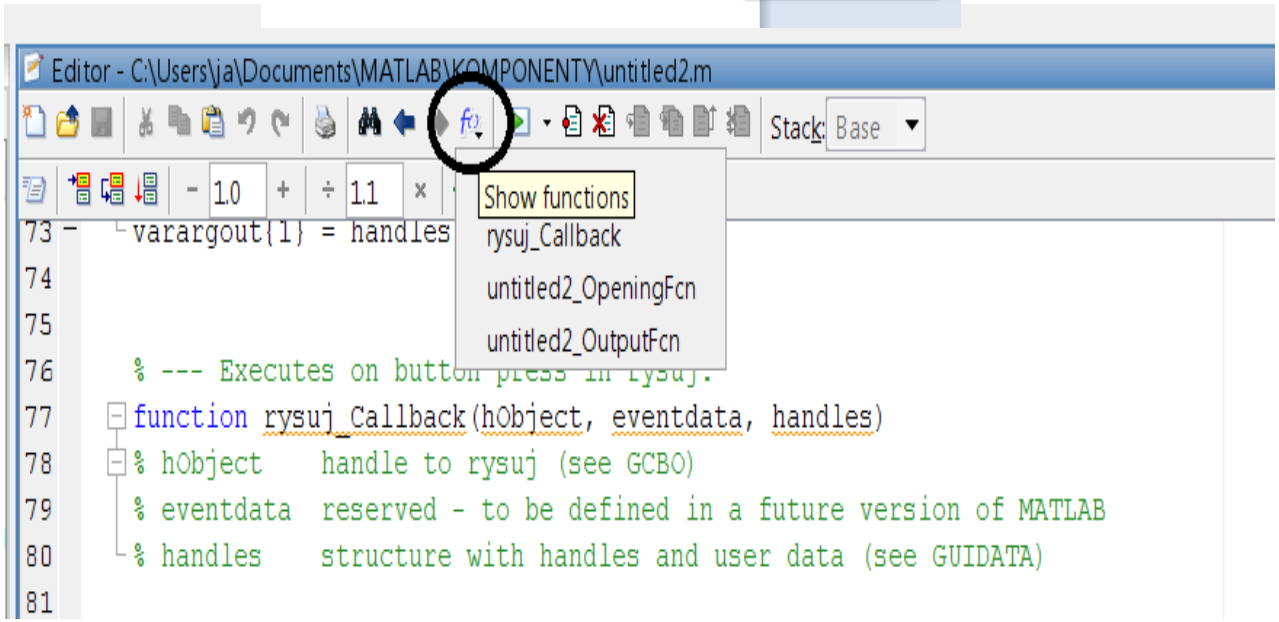
function slider1_Callback – na każde przesunięcie suwaka

itp.

Oczywiście możemy zmienić Tag elementu i wtedy funkcja może mieć nagłówek:
function suwak_Callback

lub


OK
Cut
Copy
Paste
Clear
Duplicate
Bring to Front
Send to Back
Property Inspector
Object Browser
M-file Editor
View Callbacks
Property Editor
Callback
CreateFcn
DeleteFcn
ButtonDownFcn
KeyPressFcn

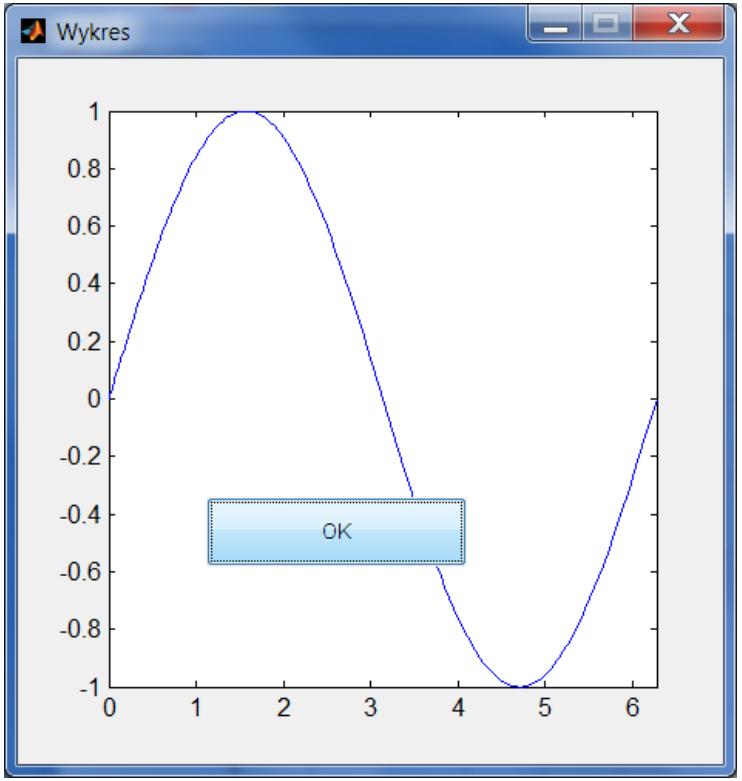

Editor - C:\Users\ja\Documents\MATLAB\KOMPONENTY\untitled2.m
Stack: Base
Show functions
rysuj_Callback
untitled2_OpeningFcn
untitled2_OutputFcn
varargout{1} = handles
% --- Executes on button press in rysuj.
function rysuj_Callback(hObject, eventdata, handles)
% hObject handle to rysuj (see GCBO)
% eventdata reserved - to be defined in a future version of MATLAB
% handles structure with handles and user data (see GUIDATA)

Jeżeli nawet nie umieścimy w formularzu komponentu wykresu *Axes*

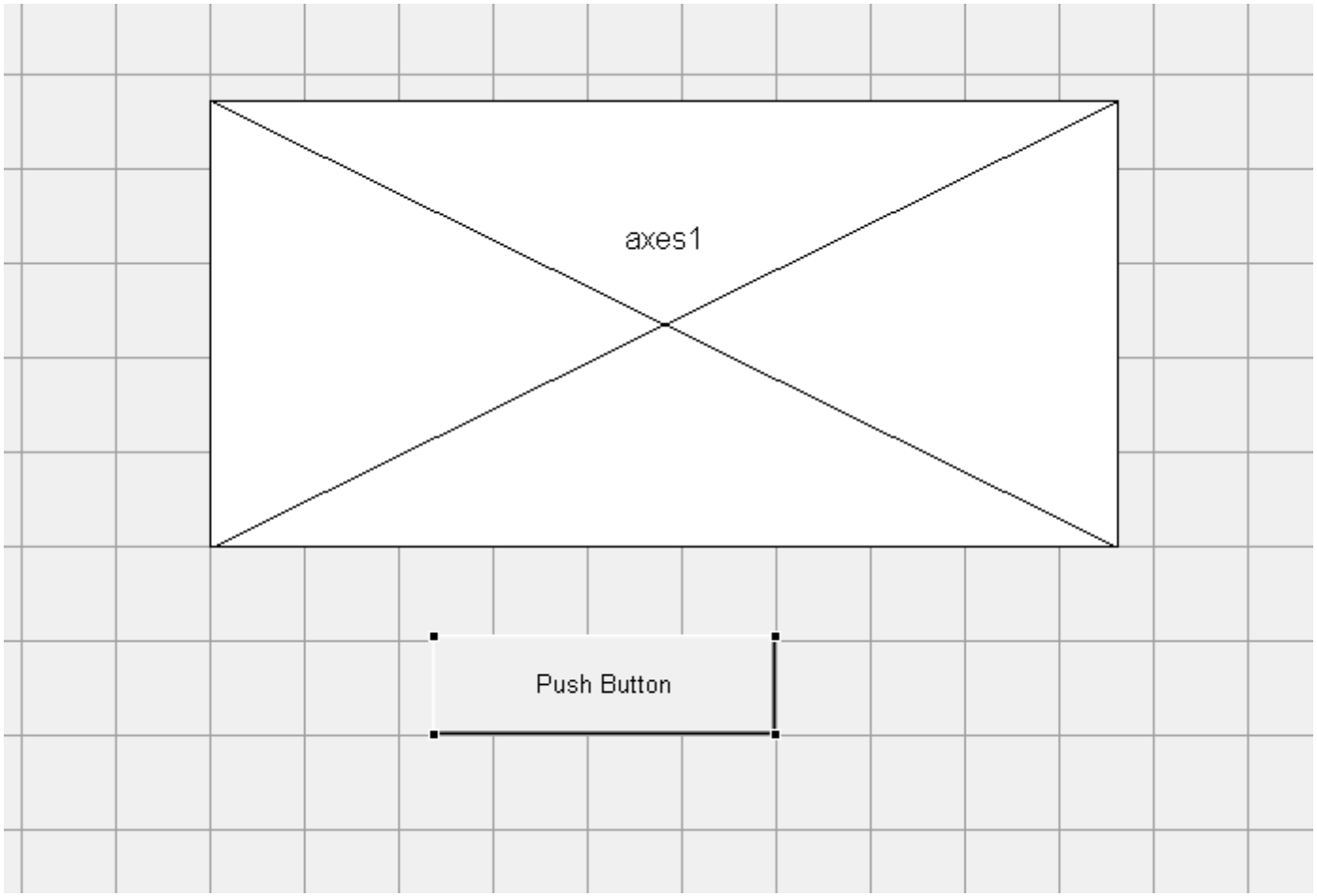
...

```
function rysuj_Callback (hObject, eventdata, handles)
% hObject    handle to rysuj (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
fplot('sin(x)',[0 2*pi])
```

i uruchomimy aplikację przyciskiem Run



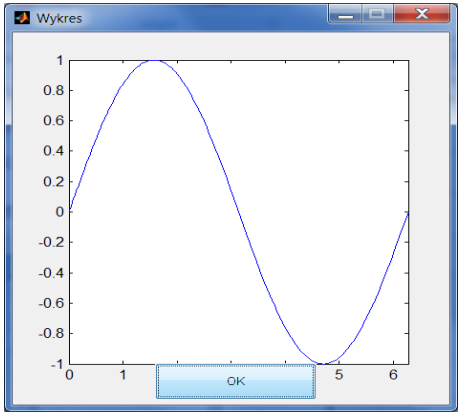
Projektujemy prostą aplikację rysującą wykres:



W naszej aplikacji wpisujemy kod:

```
function pushbutton1_Callback(hObject, eventdata, handles)
syms x
f=sin(x)
fplot (f ,[0 2*pi])
```

Zapisujemy zmiany w projekcie okna aplikacji klikając ikonę dyskietki
Uruchamiamy aplikację w znany sposób klikając zielony element w oknie projektu lub oknie edytora i sprawdzamy działanie przycisku.



Po uruchomieniu aplikacji wszystkie komponenty umieszczane są w zmiennej (strukturze obiektów) o nazwie:

handles

Dostęp do obiektu uzyskujemy zapisem:

handles.tag_obiektu

Zamiar pobrania (do zmiennej) lub zmiany właściwości dowolnego komponentu realizowany jest z wykorzystaniem dwóch funkcji:

Pobranie właściwości do zmiennej:

zmienna = get (handles.*tag_obiektu*, *właściwość*)

Ustawienie właściwości:

set (handles.*tag_obiektu*, *właściwość*, *wartość*)

Przykładowo, jeśli chcemy, żeby kod przycisku wykorzystywał położenie suwaka (inny komponent), piszemy w kodzie Callback dla przycisku **pushbutton**:

```
function pushbutton1_Callback(hObject, eventdata, handles)
pozycja=get(handles.slider1,'Value');
```

i możemy tę wartość wykorzystać w dalszej części kodu:

```
function pushbutton1_Callback(hObject, eventdata, handles)
pozycja=get(handles.slider1,'Value');
x=0:0.01:2*pi
y=sin(pozycja*x);
plot(x,y)
```

Uwaga: wcześniej projektując aplikację trzeba nadać suwakowi w Inspektorze wartości skrajne Min i Max oraz wartość startową Value

Inny prosty przykład – na kliknięcie przycisku przepisywanie tekstu wpisanego w jednym komponencie Edit Text do drugiego, kolejne kliknięcie powrót tekstu do pierwszego Edit Text

```
function pushbutton1_Callback(hObject, eventdata, handles)
if length(get(handles.edit1,'String'))~=0  %jeżeli długość napisu <>0
   tekst=get(handles.edit1,'String');     %pobranie tekstu z edit1
   set(handles.edit2,'String',tekst);      %ustawienie tektstu w edit2
   set(handles.edit1,'String','');          %wyczyszczenie edit1
else
  tekst=get(handles.edit2,'String');
  set(handles.edit1,'String',tekst);
  set(handles.edit2,'String','');
end
```

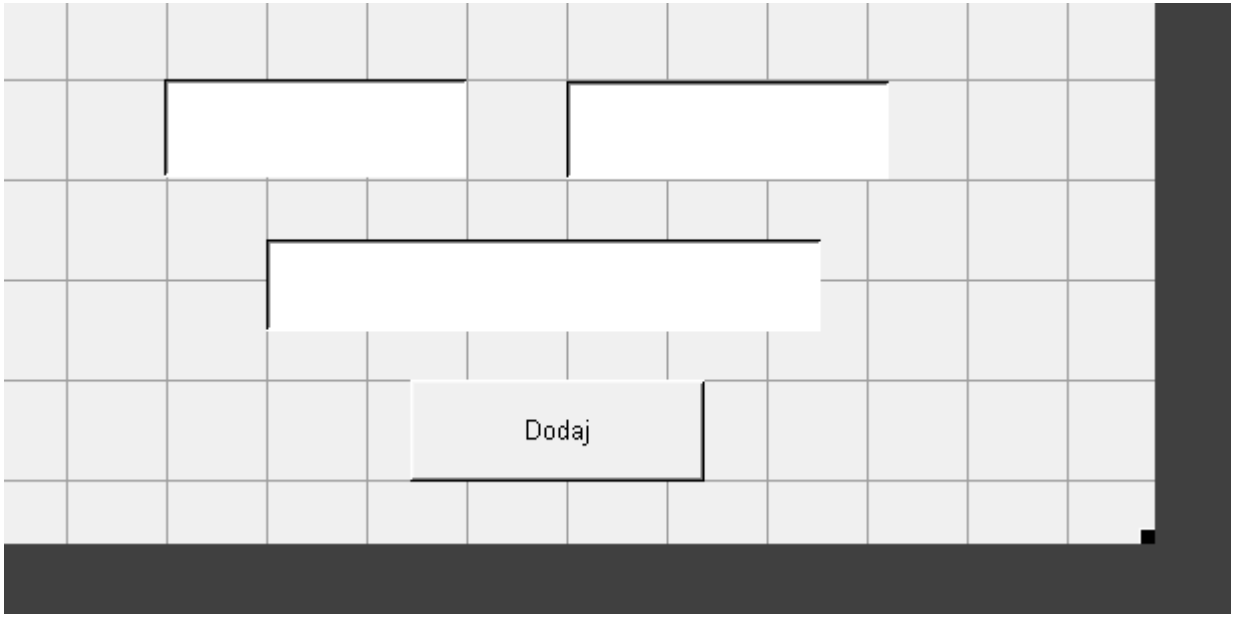
Operacje arytmetyczne w aplikacji komponentowej

Wykorzystując komponenty Edit Text do wpisywania liczb i wyświetlania wyników obliczeń musimy dokonywać **konwersji typu danych**, ponieważ dane wyświetlane w tych komponentach są typu tekstowego, a nam potrzebny jest typ liczbowy.

Korzystamy z dwóch standardowych funkcji:

str2double(*tekst*)	- konwersja tekstu na liczbę
num2str(*liczba*)	- konwersja liczby na tekst

Sumator



Kod *Callback* dla przycisku:

```
function pushbutton1_Callback(hObject, eventdata, handles)
    %pobranie 1 tekstu 1 i konwersja na liczbę
    s1=str2double(get(handles.edit1,'String'));
    %pobranie tekstu 2 i konwersja na liczbę
    s2=str2double(get(handles.edit2,'String'));
    %obliczenie sumy
    suma=s1+s2;
    % konwersja liczby na tekst
    tekst= num2str(suma);
    %wysłanie tekstu do 3 pola edycyjnego
    set(handles.edit3,'String',tekst);
```

KONIEC