**Pliki wchodzące w skład projektu**
**Rozmiar formatki, okna.**
**Wyświetlanie okien komunikatów**
**Rzutowanie**
**Konwersja**
**Tworzenie standardowych przycisków**

## *Pliki wchodzące w skład projektu.*

- Elementy projektu: plik projektu, plik modułu, plik formularza

Najprostszy projekt, na przykład domyślny Project1 zawiera plik z kodem w Pascalu (moduł) .PAS i plik zawierający zapisany w postaci tekstowej „obraz" formularza (.DFM). Oba te pliki są niezbędne do utworzenia formularza.

## Plik projektu (.DPR)

Jest również tworzony dla każdego projektu. Zawiera program główny, łączący w jedną całość wszystkie plik projektu. Domyślny plik projektu zawiera:

- Domyślną nazwę projektu (np. Project1).
- Deklarację modułów (także należących do projektu formularzy, które są modułami).
- Część główną programu uruchamiającą aplikację.

**Aby obejrzeć plik projektu .DPR:**
**1** Wybierz View|Project Source
Plik projektu jest tworzony automatycznie. W przypadku dodania nowego formularza plik projektu jest automatycznie aktualizowany.

Dodaj nowy, pusty formularz do projektu i obejrzyj zmiany w pliku projektu.
Zostanie dodany nowy moduł (*Unit2*), plik (UNIT2.PAS) oraz identyfikator formularza (*Form2*).

## Plik modułu (.PAS)

Domyślny plik modułu zawiera:

- Domyślną nazwę modułu.

Powinna być zmieniona na bardziej znaczącą np. główny moduł programu może się nazywać MainForm. Nazwę modułu można zmienić za pomocą polecenia File|Save As lub przy pierwszym zachowywaniu projektu.

- Część **interface** zawierającą:
  - Deklaracje modułów **uses**
  - Deklaracja typu formularza (klasa – formularz)
  - Deklaracja zmiennej obiektowej (obiekt – formularz)

Po utworzeniu nowego formularza deklaracja potrzebnych modułów jest wykonywana automatycznie.

- Część **implementation** z kodem źródłowym formularza.

n **Aby obejrzeć plik modułu formularza .PAS:**
**1** Wybierz zakładkę z nazwą odpowiedniego pliku w oknie edytora kodu (Code Editor). Jeśli w oknie Edytora kodu jej nie ma, wybierz View|Units....

## Plik formularza (.DFM)

Plik .DFM jest plikiem binarnym i na ekranie jest widoczna jego graficzna reprezentacja.
Można jednak oglądać plik formularza w postaci tekstowej otwierając go w oknie edytora kodu (Code Editor).

## Moduły nie związane z formularzem

Większość modułów jest związana z formularzem, ale można też tworzyć moduły niezależne.
Przy tworzeniu niezależnego modułu Delphi generuje tylko następujący kod

```
unit Unit2;
interface
implementation
end.
```

Sami musimy zadeklarować użycie potrzebnych modułów.

**Plik tworzone w trybie projektu**

| **Rozszerzenie pliku** | **Definicja** | **Opis** |
|---|---|---|
| .DPR | Plik projektu | Kod pascalowy dla programu głównego projektu. Zawiera listę wszystkich formularzy i plików modułów projektu oraz zawiera kod inicjalizujący aplikację. Tworzony, gdy projekt jest po raz pierwszy zachowywany. |
| .PAS | Kod źródłowy modułu (Object Pascal) | Dla każdego formularza jest tworzony dla każdego formularza (gdy zostanie zachowany). (W projekcie mogą być również moduły nie związane z żadnym formularzem). Zawiera wszystkie deklaracje i procedury, w tym procedury obsługi zdarzeń. |
| .DFM | Plik graficzny formularza | Binarny plik zawierający właściwości formularza. Dla każdego formularza jest tworzony oddzielny plik .DFM podczas pierwszego zachowywania projektu. |
| .OPT | Plik zawierający opcje projektu | Plik tekstowy zawierający bieżące ustawienia opcji projektu. Tworzony, gdy są zachowywane po raz pierwszy zmiany dokonane w opcjach projektu. |
| .RES | Plik zasobu kompilatora (Compiler resource file) | Binarny plik zawierający ikony aplikacji i inne zewnętrz zasoby wykorzystywane przez projekt. |
| .~DP | Kopia zapasowa projektu. | Tworzony podczas drugiego zachowywania projektu. Zawsze<br><br>zawiera poprzednią wersję pliku projektu. |
| .~PA | Kopia zapasowa pliku modułu | Kopia zapasowa poprzedniej wersji pliku modułu. (Poprzednia wersja pliku). |
| .~DF | Kopia zapasowa graficznego pliku formularza. | Kopia zapasowa binarnego pliku z ustawieniami właściwości. Zawiera poprzednią wersję pliku. |
| .DSK | Ustawienia desktopu | Plik zawiera informację o ustawieniach desktopu ustalonych dla pliku w oknie dialogowym *Environment options*. |

**Pliki tworzone podczas kompilacji**

| Rozszerzenie | Definicja | Opis |
|---|---|---|
| .EXE | Skompilowany plik wykonywalny | Plik wykonywalny aplikacji, zawiera również pliki .DCU. Wystarczy do rozpowszechniania aplikacji. |
| .DCU | Skompilowany moduł | Dla każdego pliku .PAS jest tworzony taki plik. |
| .DLL | Skompilowana biblioteka dynamiczna | |

## Rozmiar okna, formatki, położenie okna

**Metoda I – poprzez object inspektor**
Klikamy na formatkę i ustawiamy parametry width, height, left, top.
AutoScroll – włącz/wyłącz scrolle.
Align – położenie okna po uruchomieniu programu.

**Metoda II** – poprzez metodę ręcznego zmniejszenia rozmiaru okna formatki

# Wyświetlanie okien komunikatów

```
// MessageBox w Delphi
function MessageBox(hWnd: HWND; lpText, lpCaption: PChar;  uType: UINT): Integer; stdcall;
```

Funkcji MessageBox używamy, aby pokazać okno dialogowe zawierające wiadomość (treść komunikatu), tytuł oraz dowolną kombinację ikon i przycisków.

**Parametry:**

*hWnd*
Identyfikuje właściciela okna komunikatu. W przypadku, gdy parametr jest równy 0 (NULL) okno nie posiada właściciela. Jeśli parametr posiada wartość Handle, to właścicielem komunikatu jest okno aplikacji (formularz). Na przykład, jeśli właścicielem okna dialogowego jest nasza aplikacja (formularz), to podczas wyświetlania komunikatu okno programu jest nieaktywne. Sterowanie do aplikacji zostanie zwrócone dopiero po potwierdzeniu komunikatu.

*lpText*
Parametr określający tekst wyświetlanego komunikatu. W przypadku długiej treści komunikatu, jest ona automatycznie zawijana (do następnej linii) w oknie dialogowym.

*lpCaption*
Parametr opcjonalny, określający tytuł okienka dialogowego. W przypadku, gdy parametr ma wartość 0 (NULL), tytuł okienka przyjmuje domyślny tekst Błąd.

*uType*
Parametr uType zawiera flagi, które określają wygląd i zachowanie się okna dialogowego. Parametr ten może być kombinacją flag z dowolnych grup flag. W przypadku, gdy parametr posiada wartość 0, zostanie wyświetlone okno dialogowe z przyciskiem OK.

**Inna opcja :**
**ShowMessage('Błąd wejścia/wyjścia');**

```
ShowMessage(const Msg: string);
```

## Metoda MessageBox()

Metoda *MessageBox* wyświetla okno komunikatu, które zawiera tekst podany jako parametr oraz, opcjonalnie, różnorodne nagłówki i przyciski. Jest to wywołanie wbudowanej funkcji Windows API, z tym, że nie trzeba podawać uchwytu okna.
Składnia funkcji jest następująca:

```
function MessageBox(Text, Caption: PChar; Flags: Word): Integer;
```

Znaczenie parametrów jest następujące:

| **Parametr** | **Znaczenie** |
|---|---|
| Text | Tekst wyświetlany w oknie dialogowym |
| Caption | Tytuł na pasku tytułu w oknie dialogowym. |
| Flags | Typ przycisku (przycisków) w oknie dialogowym. |

Stałe, które możesz podać jako parametry można znaleźć w pomocy. Przykładowe to:

| ***Stała Flags*** | ***Znaczenie*** |
|---|---|
| MB_AbortRetryIgnore | Wyświetla zestaw przycisków: Przerwij, Ponów próbę, Zignoruj |
| MB_OK | Wyświetlany jest tylko przycisk OK |
| MB_OkCancel | Wyświetla zestaw dwóch przycisków: OK i Anuluj |

| | |
|---|---|
| MB_YesNo | Wyświetla zestaw dwóch przycisków: Tak i Nie |
| MB_YesNoCancel | Wyświetla zestaw przycisków: Tak, Nie i Anuluj |
| MB_IconExclamation | Wyświetlana jest ikona wykrzyknika |
| MB_IconInformation | Wyświetlana jest ikona z pochyloną literą *i* |
| MB_IconQuestion | Wyświetlana jest ikona ze znakiem zapytania |
| MB_IconStop | Wyświetlana jest ikona ze znakiem stopu |
| MB_DefButton1 | Pierwszy przycisk jest przyciskiem domyślnym |
| MB_DefButton2 | Drugi przycisk jest przyciskiem domyślnym. |
| MB_DefButton3 | Trzeci przycisk jest przyciskiem domyślnym. |
| MB_DefButton4 | Czwarty przycisk jest przyciskiem domyślnym. |

Flagi można łączyć za pomocą symbolu + np.

**MB_YesNo + MB_IconInformation + MB_DefButton1**

Metodę wywołujemy np. dla aplikacji :

**Application.MessageBox('Czy chcesz kontytuować tę operację', 'Potwierdzenie', MB_YesNoCancel + MB_IconQuestion);**

Metoda zwraca kod, który symbolizuje wciśnięty przycisk:

| ***Kod zwracany*** | ***Jego znaczenie*** |
|---|---|
| IDAbort | Został wciśnięty przycisk *Przerwij* |
| IDCancel | Został wciśnięty przycisk *Anuluj* |
| IDIgnore | Został wciśnięty przycisk *Zignoruj* |
| IDNo | Został wciśnięty przycisk *Nie* |
| IDOK | Został wciśnięty przycisk *OK* |
| IDRetry | Został wciśnięty przycisk *Ponów próbę* |
| IDYes | Został wciśnięty przycisk *Tak* |

Tak możesz wypróbować działanie funkcji *MessageBox*.

```
procedure TForm1.Button1Click(Sender: TObject);
var
   Button: Integer;
begin
   Button := Application.MessageBox('Witamy w świecie Delphi!', 'Message Box', mb_OKCancel +
      mb_DefButton1);
   if Button = IDOK then
      Label1.Caption := 'Wybrałeś OK';
   if Button = IDCANCEL then
       Label1.Caption := 'Wybrałeś Anuluj';
end;
```

## Rzutowanie

W poleceniu MessageBox trzeba podawać kilka parametrów. W przypadku drugiego i trzeciego parametru ma być to tekst, który zostanie wyświetlony w okienku. Ten tekst musi być typu PChar. Możesz więc napisać tak i będzie dobrze:

```
var
  Tekst, Tytul : PChar;
begin
  Tekst := 'Delphi jest fajne!';
  Tytul := 'Nie wiem?';
  MessageBox(0, Tekst, Tytul, MB_OK);
```

Tak, będzie dobrze. Dlaczego? Bo zmienne Tekst oraz Tytul są typu PChar. Teraz zamiast typu PChar napisz String:

```
var
  Tekst, Tytul : String;
begin
  Tekst := 'Delphi jest fajne!';
  Tytul := 'Nie wiem?';
  MessageBox(0, Tekst, Tytul, MB_OK);
```

Takie coś już nie przejdzie. Kompilator wyświetli błąd: [Error] Project1.dpr(21): Incompatible types: 'String' and 'PChar'. Kompilator próbuje Ci powiedzieć: "Co Ty wyprawiasz! Przecież typy PChar i typ String to dwie różne rzeczy!". Częściowo ma racje, ale to tylko głupia maszyna i da się ją oszukać. Do tego właśnie służy rzutowanie. Polega to na oszukaniu kompilatora. On "myśli", że jest to typ PChar, a ty stosujesz typ String:

```
var
  Tekst, Tytul : String;
begin
  Tekst := 'Delphi jest fajne!';
  Tytul := 'Nie wiem?';
  MessageBox(0, PChar(Tekst), PChar(Tytul), MB_OK);
```

Jest to rzutowanie typu String na typ PChar. Kompilacja się powiedzie i wszystko będzie dobrze. Fajnie, nie? Ale takie coś przejdzie tylko do czasu. Bo takie coś już nie wyjdzie:

```
var
  Licznik : Integer;

begin
  Licznik := 1;
  MessageBox(0, PChar(Licznik), '', MB_OK);
end.
```

Fachowca by powiedział: "To oczywiste. Przecież typy Integer i PChar zapisane są w odrębnych komórkach pamięci i bleblelble". Ktoś jednak pomyślał i stworzył:

# Konwersja

Konwersja polega na wykorzystaniu gotowych poleceń do zmiany typów. Jeżeli np. typ Integer chcesz przedstawić w postaci tekstu ( String ) stosujesz polecenie IntToStr. Zaraz przetestujemy to w praktyce. Żeby użyć tego polecenia do listu modułów uses musisz dodać moduł SysUtils. Gotowe? A, wiesz co? Dodaj jeszcze moduł Dialogs. Powinno to wyglądać tak:

```
uses
  Windows,
  SysUtils, { <-- ten moduł jest konieczny, aby zastosować konwersje }
  Dialogs; { <-- ten zawiera jedno polecenie - ShowMessage }
```

Moduł Dialogs zawiera polecenie ShowMessage, którego będziemy używali. Jak łatwo się domyśleć polecenie to wyświetla w oknie tekst ( to taka odmiana polecenia MessageBox ). Wpisz taki tekst do programu ( komentarze możesz oczywiście pominąć ):

```
var
  Data : TDateTime; { nowy typ - przechowuje datę }
  Rozdzial : Integer; { znany już typ Integer }
  Imie : String; { również znany typ String }
begin
  Data := Now; { przypisuje zmiennej aktualną datę }
  Rozdzial := 2;
  Imie := 'Komputer';
  ShowMessage('Cześć, mam na imię ' + Imie + ', a Ty właśnie czytasz ' + IntToStr(Rozdzial) +
  ' rozdział książki o Delphi! Dzisiaj jest: ' + DateToStr(Data));
end.
```

Hohhoho. To coś trudniejszego, ale jakże przydatne w programowaniu. Zacznijmy od początku. Polecenie ShowMessage ma tylko jeden parametr - jest nim tekst, który ma być wyświetlony w okienku. Tekst ten ma być typu String. Operator + służy do połączenia odpowiednich części ( zmiennych ). Po prostu gdy chcesz w dane miejsce wstawić wartość zmiennej to kończysz tekst stawiając apostrof, następnie znak + mówiący o podłączeniu zmiennej. Później wpisujesz zmienną, która ma być w to miejsce wstawiona. W powyższym wypadku jeżeli będziesz chciał wstawić zmienną, która nie jest typu String trzeba będzie zastosować konwersje. Program możesz uruchomić i sprawdzić jak działa.

Oto inne polecenia konwertujące typy:

**IntToStr** - konwertuje zmienna "Integer" na "String"
**StrToInt** - "String" na "Integer"
**StrPas** - "PChar" na "String"
**StrPCopy** - "String" na "PChar"
**TimeToStr** - zmienna "Time" na "String"
**DateToStr** - zmienna "Date" na "String"
**StrToDate** - "String" na "Date"
**StrToTime** - "String" na "Time"
**CurrToStr** - "Currency" ( zmiennoprzecinkowy ) na "String"
**StrToCurr** - "String" na "Currency"
**FloatToStr** - zmiennoprzecinkowy na "String"

## Tworzenie standardowych przycisków

Można szybko stworzyć standardowe przyciski poprzez dodanie komponentu BitBtn i ustawienie jego właściwości *Kind* . Zmiana tej właściwości powoduje jednoczesną zmianę właściwości *ModalResult* , z wyjątkiem rodzajów *bkCustom* , *bkHelp* i *bkClose* . Dla tych przypadków *ModalResult* przyjmuje wartość *mrNone*, a okno nie jest automatycznie zamykane po wybraniu takiego przycisku. Standardowe rodzaje przycisków zawierają nie tylko tekst, ale też i symbol graficzny.

Predefiniowane typy przycisków BitBtn

| **Wartość właściwości *Kind*** | **Efekt** | **Ustawienia innych właściwości** | **Komentarz** |
|---|---|---|---|
| BkAbort | Tworzy przycisk Cancel z tekstem Abort. | Caption := 'Abort'<br>ModalResult := mrAbort | Obok tekstu pojawia się czerwony X. |
| bkAll | Tworzy przycisk OK z tekstem All. | *Caption := '&All'*<br>*ModalResult := 8* | Obok tekstu pojawia się zielony podwójny check mark. |
| bkCancel | | *Caption := 'Cancel'*<br>*Cancel := True*<br>*ModalResult := mrCancel* | Obok tekstu pojawia się czerwony X. |
| bkClose | Tworzy przycisk Close z tekstem Close. Zamyka okno. | *Caption := '&Close'* | Obok tekstu pojawiają się drzwi wyjściowe. |
| bkCustom | Żaden | Nie dotyczy | Użyj tego ustawienia, aby stworzyć własny przycisk z wybranym obrazkiem (*Glyph* - bitmapa). |
| bkHelp | Tworzy przycisk z tekstem Help. | *Caption := '&Help'* | Obok tekstu pojawia się niebieski znak zapytania. Użyj procedury obsługi zdarzeń do wywołania pliku pomocy. (Jeśli okno dialogowe ma kontekst pomocy, Delphi zrobi to automatycznie). |
| bkIgnore | Tworzy przycisk służący do ignorowania zmian i kontynuowania akcji. | Caption := '&Ignore'<br>ModalResult := mrIgnore | Użyj do kontynuowania akcji po wystąpieniu błędu. |
| bkNo | Tworzy przycisk Cancel z tekstem No | *Caption := '&No'*<br>*Cancel := True*<br>*ModalResult := mrNo* | Obok tekstu pojawia się czerwone przekreślone koło. |
| bkOK | Tworzy przycisk OK z tekstem OK | *Caption := 'OK'*<br>*Default := True*<br>*ModalResult := mrOK* | Obok tekstu pojawia się zielony check mark . |
| BkRetry | Tworzy przycisk do ponawiania akcji. | Caption := '&Retry'<br>ModalResult := mrRetry | Obok tekstu pojawiają się dwie cykliczne zielone strzałki. |
| BkYes | Tworzy przycisk OK z tekstem Yes. | *Caption := '&Yes'*<br>*Default := True*<br>*ModalResult := mrYes* | Obok tekstu pojawia się zielony check mark . |