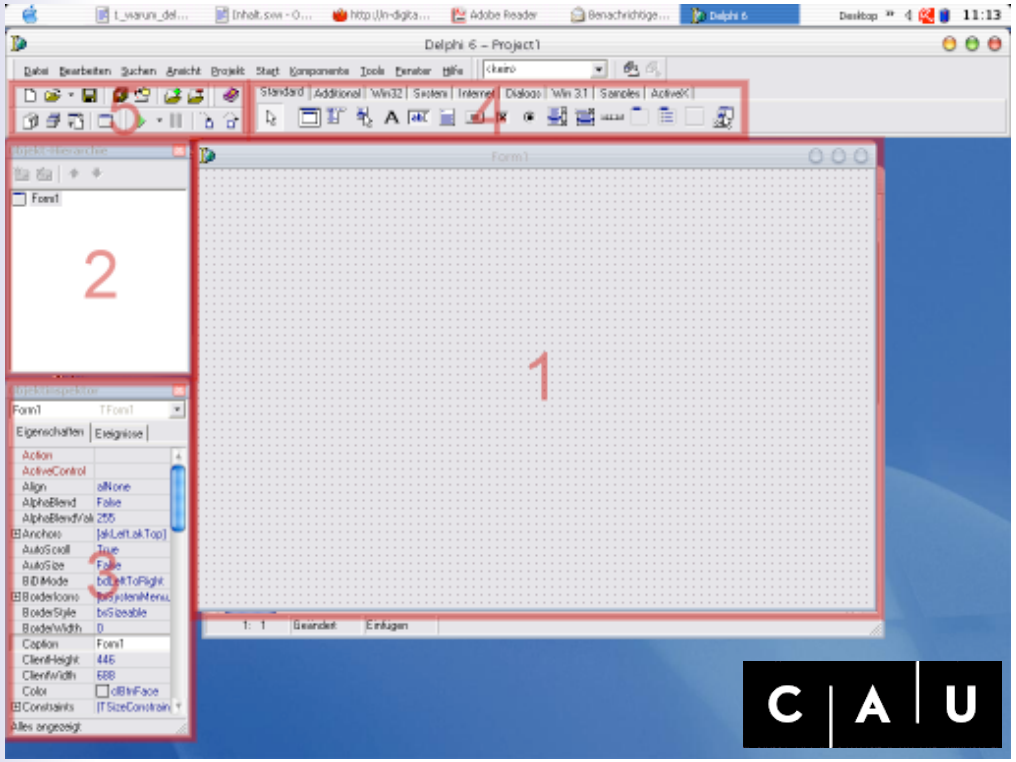
Borland Delphi

Einführung in die Programmiersprache

- Objective Pascal
- objektorienterte Programmiersprache
- Rapid Development
- schweizer Professor Niklaus Wirth (* 1934, Bild rechts) 1968-1972 an der ETH Zürich
- 1983 Pascal
- 1995 Delphi 1

RAD-Umgebung



1. Frame
2. Objekt Hierachie
3. Objekt Inspektor
4. Komponenten-palette
5. Symbolleiste

Inhalt

1. Syntax
2. Variablen, Konst.
3. String-Typen
4. Kompl. Datentypen
5. Zeiger
6. Schleifen
7. Verzweigungen
8. Operatoren
9. Prozeduren, Funkt.
10. Programmaufbau
11. Objektorientierung
12. Zugriff auf Objekte

Syntax

- Befehle werden durch `;` getrennt
- `meine_variable` und `Meine_Variable` die in *C/C++* unterschiedlich sind, sind sie für die Delphi-Sprache **gleich**
- Typstrenge:
  - Variablen müssen vor ihrer Verwendung in einem bestimmten Bereich deklariert werden
  - Ist einer Variable einmal ein Typ zugeordnet, kann sie keine Werte eines anderen Typs aufnehmen, nur Werte des gleichen Typs oder eines Untertyps.

    Bsp. ein `Integer` kann keine `Realzahlen` aufnehmen
  - Zuweisungen zwischen Variablen unterschiedlichen Typs sind meist nur über Konvertierfunktionen (wie z.B. `IntToStr`) möglich

Datentypen

Typ	**Wertebereich**	**Beispiel**:Deklaration	Zuweisung
Integer (ganze Zahlen)	-2147483648 bis 2147483647	`var zahl: integer;`	`zahl:=14;`
Real (Gleitkommazahlen)	5.0×10^{324} .. 1.7×10^{308}	`var zahl: real;`	`zahl:=3.4;`
String (Zeichenketten)	ca. 2^{31} Zeichen	`var text: string;`	`text:='Hallo Welt!';`
Char (Zeichen)	1 Zeichen	`var zeichen: char;`	`zeichen:='a';`
Boolean (Wahrheitswert)	true, false	`var richtig: boolean;`	`richtig:=true;`

Deklaration

```
var zahl1, zahl2, zahl3: integer;
    ergebnis: real;
    text, eingabe: string;
```

Zuweisung

```
x := x + 1;
```

Globale Variablen

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;

var
  Form1: TForm1;
  einezahl: integer;  // Diese Variable gilt in der ganzen Unit und
      // in allen Units, die diese Unit einbinden

implementation

{$R *.DFM}

var eine_andere_zahl: real; // Diese Variable gilt nur in dieser Unit
```

Globale Variablen können bei ihrer Deklaration mit einem Startwert belegt werden:

```
einezahl: integer = 42;
```

Lokale Variablen

```
procedure IchMacheIrgendwas;
var text: string;
begin
  ... //Irgendwas Sinnvolles
end;
```

Typumwandlungen

von	*nach*	*Funktion*	*Beispiel*

Integer	Real	kein Problem, einfache Zuweisung	`var zahl1: integer;`
			`zahl2: real;`
			`begin`
			`zahl2 := zahl1;`

Real	Integer	Möglichkeiten:
		- Nachkommastellen abschneiden (trunc)
		- kaufm. Runden (round)
		- aufrunden (ceil, Unit Math)
		- abrunden (floor, Unit Math)
			`var zahl1: real;`
			`zahl2: integer;`
			`begin`
			`zahl2 := trunc(zahl1);`
			`zahl2 := round(zahl1);`

Integer	String	IntToStr	`var textzahl: string;`
			`zahl: integer;`
			`begin`
			`textzahl := IntToStr(zahl);`

Real	String	FloatToStr
		FloatToStrF
			`zahl: real;`
			`begin`
			`textzahl := FloatToStr(zahl);`

String	Integer	StrToInt
		StrToIntDef
			`var textzahl: string;`
			`zahl: Integer;`
			`begin`
			`zahl := StrToInt(textzahl);`

String	Real	StrToFloat	`var textzahl: string;`
			`zahl: real;`
			`begin`
			`zahl := StrToFloat(textzahl);`

String	Char	Zugriff über Index
		(1 ist erstes Zeichen)
			`var text: string;`
			`zeichen: char;`
			`begin`
			`zeichen := text[1];`

Char	String	kein Problem, einfache Zuweisung	`var zeichen: char;`
			`text: string;`
			`begin`
			`text := zeichen;`

Teilbereichstypen

```
var kleineZahl: 0..200;

var kleinerString: string[10];
```

Aufzählungstypen

```
var farben: (blau, gelb, gruen, rot);
```

ord	gibt die Position des Bezeichners zurück
pred	gibt den Vorgänger zurück
succ	gibt den Nachfolger zurück
low	gibt den niedrigsten Wert zurück
high	gibt den höchsten Wert zurück

Konstanten

```
const version = '1.23';
```

String-Typen


```
var kurzertext: ShortString
```

```
var kurzertext: String[20];
```

- String
- ShortString
- AnsiString
- WideString
- Array[0..x] of Char
- PChar

Enumerierte Strings

```
var text: array [0..100] of Char;
```

Zeichenketten zusammenhängen

```
var text1, text2: String;
begin
  text1 := 'gut';
  text2 := 'Ich finde Delphi '+text1+'!!!';
  // text2 enthält nun den Text 'Ich finde Delphi gut!!'
```

Zugreifen auf ein bestimmtes Zeichen in einem String

```
var text: String;
  zeichen: Char;
begin
  text := 'Ich finde Delphi gut!';
            zeichen := text[1];
  // zeichen enthält nun den Buchstaben 'I'
```

Vergleichen zweier Strings

```
var text1, text2: string;
                ...
if text1 = text2 then ...
```

Mengentypen

```
var zahlen: set of 0..255;

zahlen := [5, 9];           // zahlen enthält die Zahlen 5 und 9
zahlen := [];               // zahlen enthält überhaupt keine Werte
zahlen := [1..3];           // zahlen enthält die Zahlen von 1 bis 3
zahlen := zahlen + [5];     // zahlen enthält die Zahlen 1, 2, 3, 5
zahlen := zahlen - [3..10]; // die Zahlen von 3 bis 10 werden aus der Menge
                            // entfernt, es bleiben 1 und 2

 if 7 in zahlen then ...
```

Arrays

```
var testwert: array [0..10] of integer;


testwert[0] := 15;
testwert[1] := 234;


for i := 0 to 10 do
   testwert[i] := 1;

var koordinate: array [1..10, 1..10] of integer;

koordinate[1,6]  := 34;
koordinate[7][3] := 42;
```

Dynamische Arrays

```
var dynArray: array of integer;

SetLength(dynArray, 5);

dynArray[0] := 321;


for i := 0 to high(dynArray) do
        dynArray[i] := 0;

SetLength(dynArray, 2);
```

Records

```
var Adresse: record
  name: string;
  plz: integer;
  ort: string;
end;
```

```
with adresse do begin
  name := 'Hans Müller';
       plz := 12345;
  ort := 'Irgendwo';
end;
```

```
adresse.name := 'Hans Müller';
     adresse.plz := 12345;
adresse.ort := 'Irgendwo';
```

Zeiger

Zeiger sind Variablen, die eine Specheradresse enthalten.

```
type
  PAdressRecord = ^TAdressRecord;
  TAdressRecord = record
    name: string;
    plz: integer;
    ort: string;
end;

var adresse: TAdressRecord;
  adresszeiger: PAdressRecord;
```

Steht das Symbol ^ vor einem `Typbezeichner`, wird daraus ein Typ, der einen Zeiger auf den ursprünglichen Typ darstellt. `^TAdressRecord` ist ein Zeigertyp auf einen Speicherbereich, an dem sich etwas vom Typ `TAdressRecord` befinden muss.

```
adresszeiger := @adresse;
```

`@` zur Ermittlung einer Speicheradresse

```
var gesuchterName: string;
begin
  gesuchterName := adresszeiger^.name;
```

^ zur Dereferenzierung

for-Schleife

```
for i := 1 to 10 do begin
  ... // Befehlsfolge, die öfters ausgeführt werden soll
end;

for i:=10 downto 1 do ...
```

while-Schleife

```
while x<>y do begin
  ... // Befehlsfolge, die öfters ausgeführt werden soll
end;
```

repeat-until-Schleife

```
repeat
  ...  // Befehlsfolge, die öfters ausgeführt werden soll
until x=y;
```

for-in-Schleife

```
for element in collection do statement;

var stringarr: array of String;
  s: String;
begin
  for s in stringarr do begin
          ShowMessage(s);
end;
```

if-Verzweigung

```
if  then ;

if  then  else ;

if x>0 then ...
else if x<0 then ...
else ...;
```

```
var eingabe: integer;
...
if eingabe=1 then begin
                              eingabe := 0;
  ausgabe := 'Sie haben eine 1 eingegeben';
end //kein Strichpunkt!
else if eingabe=2 then begin
                              eingabe := 0;
  ausgabe := 'Sie haben eine 2 eingegeben';
end
else begin
  eingabe := 0;
  ausgabe := 'Sie haben eine andere Zahl als 1 oder 2 eingegeben';
end;
```

case-Verzweigung

```
case eingabe of
  1: ausgabe := 'Sie haben 1 eingegeben';
  2: ausgabe := 'Sie haben 2 eingegeben';
  3: ausgabe := 'Sie haben 3 eingegeben';
  else ausgabe := 'Sie haben nicht 1, 2 oder 3 eingegeben';
                            end;
```

```
case eingabe of
  1,3,5,7,9: ausgabe := 'Sie haben eine ungerade Zahl kleiner als 10 eingegeben';
  2,4,6,8,0: ausgabe := 'Sie haben eine gerade Zahl kleiner als 10 eingegeben';
  10..20:    ausgabe := 'Sie haben eine Zahl zwischen 10 und 20 eingegeben';
end;
```

and, or, not, xor

```
(x>0) and (x<10)
```

```
((x>0) and (x<10)) or ((y>0) and (y<10))
```

```
if ((x>0) and (x<10)) or ((y>0) and (y<10)) then ...
```

```
if <Bedingung1> and <Bedingung2>
```

Prozeduren

```
procedure <Name>(<Parameter>);
 <Variablen- und Konstanten>
 begin
   <Anweisungen>
 end;
```

```
procedure Toene(Anzahl: integer);
  var i: integer;
  begin
    for i := 1 to Anzahl do
                  beep;
  end;
```

```
Toene(5);
```

Funktion

```
function <Name>(<Parameter>): <Rückgabetyp>;
 <Variablen- und Konstanten>
 begin
   <Anweisungen>
 end;
```

```
function SummeAusDrei(zahl1, zahl2, zahl3: integer): integer;
 begin
   result := zahl1 + zahl2 + zahl3;
 end;
```

```
ergebnis:=SummeAusDrei(3,5,9);
```

Programaufbau


```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

```
program Project2;
{$APPTYPE CONSOLE}
uses sysutils;

begin
  // Hier Anwender-Code
end.
```

```
unit ;

interface

uses ;



implementation

uses



initialization


finalization


end.
```

```
package ;

requires


contains


end.
```

```
unit Unit1;

interface
  function Brutto(netto: real): real;

implementation

function Brutto(netto: real): real;
begin
  result := netto * 1.16;
end;

end.
```

Klasse

Klassen stellen den "Bauplan" eines Objekts dar.

Sie definieren:

Eigenschaften/Attribute

Methoden

Klassennamen beginnen in Delphi üblicherweise mit einem großen **T** (Type).

Dabei handelt es sich um eine Vereinbarung, nicht um eine Regel.

Soll Programmcode weitergegeben werden, so sollte man sich an diesen Standard halten !

Instanzen

Instanzen sind nun "**lebendige**" Objekte, die nach dem Bauplan einer Klasse erstellt wurden.

Sie belegen bei der Programmausführung Arbeitsspeicher und können Daten aufnehmen. Von jeder Klasse kann es *beliebig viele* Instanzen geben, die alle gleich aufgebaut sind, aber unterschiedliche Daten enthalten können

Objekte, Methoden, Eigenschaften

Objekt wird synonym mit Instanz benutzt, Methoden sind Funktionen und Prozeduren einer Klasse, Eigenschaften oder Properties bestimmen Merkmale von Klassen vgl. unten.

```
type
  TAuto = class
  private
    FFarbe: string;
    FBaujahr: integer;
    procedure SetFarbe(Farbe: string);
  public
    property Farbe: string read FFarbe write SetFarbe;
  end;


 var MeinAuto: TAuto;
```

Konstruktor, Destruktor

```
Objektreferenz := Klasse.Create;
MeinAuto := TAuto.Create;
```

```
type
  TForm1 = class(TForm)
      Button1: TButton;
      procedure Button1Click(Sender: TObject);
    private
    public
  end;  //Ende der Klasse TForm1
  TAuto = class
    private
      FFarbe: string;
      FBaujahr: integer;
      procedure SetFarbe(Farbe: string);
    public
      property Farbe: string read FFarbe write SetFarbe;
  end; //Ende der Klasse TAuto
```

```
type
  TAuto = class
  private
    FFarbe: string;
    FBaujahr: integer;
    procedure SetFarbe(Farbe: string);
  public
    property Farbe: string read FFarbe write SetFarbe;
  end;
```

```
var MeinAuto: TAuto;
```

```
var MeinAuto: TAuto;
begin
  MeinAuto := TAuto.Create;
  MeinAuto.SetFarbe('rot');
  MeinAuto.Farbe := 'rot';
```

```
MeinAuto.FFarbe := 'rot';
  MeinAuto.FBaujahr := '1980';
```

```
property Farbe: string read FFarbe write SetFarbe;
```

```
procedure TAuto.SetFarbe(Farbe: string);
begin
  if (farbe = 'rot') or (farbe = 'blau') or (farbe = 'gruen') then
                          FFarbe := Farbe;
end;
```

Zugriffsrechte

private
Ein private-Element kann nur innerhalb der gleichen Unit verwendet werden. Aus anderen Units ist ein Zugriff nicht möglich.
protected
Ein protected-Element ist wie ein private-Element innerhalb der gleichen Unit verwendbar. Darüberhinaus haben alle abgeleiteten Klassen darauf Zugriff, unabhängig davon, in welcher Unit sie sich befinden.
public
public-Elemente unterliegen keinen Zugriffsbeschränkungen.
published
published-Elemente haben dieselbe Sichtbarkeit wie public-Elemente. Zusätzlich werden diese Element im Objektinspektor angezeigt, weshalb nicht alle Typen als published-Element eingesetzt werden können.
automated
automated ist nur noch aus Gründen der Abwärtskompatibilität vorhanden. Der Einsatz erfolgte in Zusammenhang mit OLE-Automatisierungsobjekten und nur in Klassen, die von TAutoObject (Unit OleAuto) abgeleitet waren. Für TAutoObject aus der Unit ComObj wird automated nicht mehr verwendet.