maXbox

maXbox Starter 3

Start with Modular Programming

1.1 First Step

Today we spend a little time in programming with modules or units. Hope you did already work with the Starter 1 and 2 at:

http://www.softwareschule.ch/download/maxbox_starter.pdf

This lesson will introduce you to code files and form files in a separate way. So what's modular programming?
Modular programming is subdividing your program into separate subprograms such as functions and procedures. It's more of this. A modular application, in contrast to one monolithic chunk of tightly coupled code in which every unit may interface directly with any other, is composed of smaller, separated chunks of code or own units that are well isolated. Those chunks can then be developed by separate teams with their own life cycles and their own schedules. The results can then be assembled together by a separate entity—the linker.
In maXbox a module represents an include file or a unit file.

The ability to use external libraries and compose applications out of them results in an ability to create more complex software with less time and work. The trade-off is the need to manage those libraries and ensure their compatibility. That is not a simple task. But there is no other practical, cost-efficient way to assemble systems of today's complexity.

1.2 Get the Code

As you already know the tool is split up into the toolbar across the top, the editor or code part in the centre and the output window at the bottom.

In maXbox you can't edit a visual form directly on top of it, you program the form in a separate file.

Before this starter code will work you will need to download maXbox from the website. It can be down-loaded from http://www.softwareschule.ch/maxbox.htm (you'll find the download maxbox2.zip top left on the site). Once the download has finished, unzip the file, making sure that you preserve the folder structure as it is. If you double-click `maxbox2.exe` the box opens a default program. Test it with F9 or press **Compile** and you should hear a sound. So far so good now we'll open our two examples.

```
66_pas_eliza_include_sol.txt
66_pas_eliza_form_sol.inc
```

If you can't find the two files try also the zip-file loaded from:
http://www.softwareschule.ch/download/maxbox_examples.zip

The best is to open two times maXbox and work with the two files in each box. The **Load** button will present you in `/examples` with a list of Programs stored within your directory as well as a list of `/exercises` with defective Pascal code to be fixed for training purposes.
Use the `Save Page as…` function of your browser[1] and load it from `examples` (or wherever you stored it). Now let's take a look at the code of this project. Our first line is

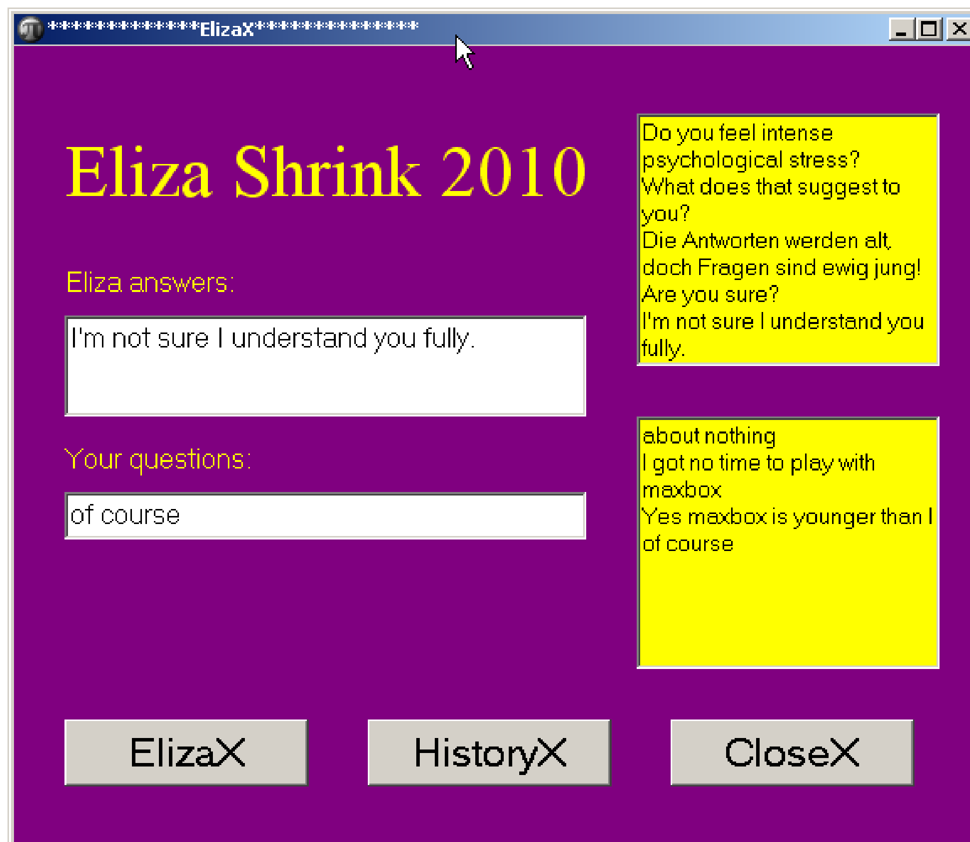
```
1 program ElizaX_Conunit;
```

We have to name the game, means the program's name is ElizaX_Conunit.

☞ This example requires a lot of objects from the classes: `TForm, TMemo, TEdit, TButton` and `TLabel;`
The program makes a series of calls to many event handlers of the form, which is triggered (invoked) from the onClick event of a button.

```
okBtn.onClick:= @OkBtnClick;
```

Most of the functions we use like Randomize() are implicit in a library (or unit). A library is a collection of code, which you can include in your program. By storing your commonly used code in a library, you can reuse code for many times in different projects and also hide difficult sections of code from the developer; another advantage of modular programming. Once a unit is tested it's stable to use.



1: The modular Eliza

Next we learn how a compiler directive works. Compiler directives are special-syntax comments we can use to control the features of a compiler. A compiler directive is an instruction to the compiler to complete a task before formally starting to compile the program, thus they are sometimes called pre-processor directives. Among other items, during the pre-processor step the compiler is looking for compiler directives and processes them as they are encountered.
After completing the tasks as directed, the compiler proceeds to its second step where it checks for syntax errors (violations of the rules of the language) and converts the source code into an object code

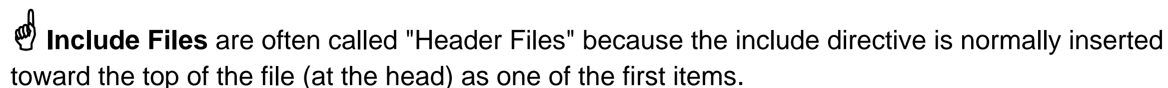
[1] Or copy & paste

that contains machine language instructions, a data area, and a list of items to be resolved when he object file is linked to other object files.
Directives can't be changed[2] while a program runs. A section starts with the symbol {$:

```
06 {$I ..\..\download\lib\66_pas_eliza_form_sol.inc}
```

The $Include compiler directive includes code from an external file in line into the current Unit or Program.
So the `66_pas_eliza_include_sol.txt` includes the `66_pas_eliza_form_sol.inc` which has by convention an inc extension. This file represents the form of picture 1.

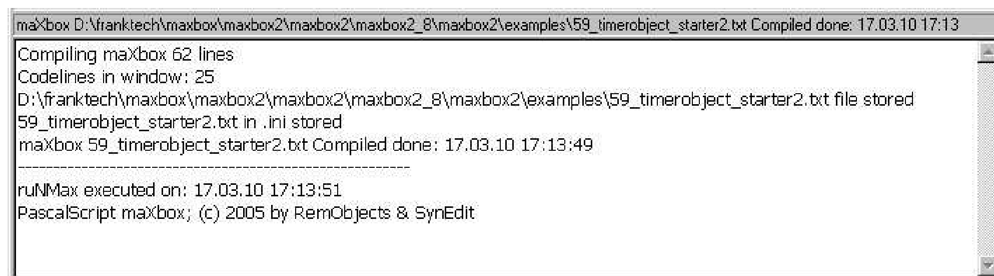
This is very useful for including compiler directives or common code into all of your units to ensure consistency, and a single point of control.
This directive can be used multiple times within your code and allows code in an include file to be incorporated into a Unit or Program.



Include Files are often called "Header Files" because the include directive is normally inserted toward the top of the file (at the head) as one of the first items.

So far we have learned something about modules and include directives and the difference between a pre-processor and a compiler. Now it's time to run your program at first with F9 (if you haven't done yet). The program generates a form and asks for input.

ELIZA was an early computer program that "chatted" with the user. It served as a parody of a Rogerian psychoanalyst. By incorporating parts of the user's statements in the responses or answers, it could be seen (with a lot of imagination) as conversing with the user. It has since been implemented on nearly every computer platform in existence.

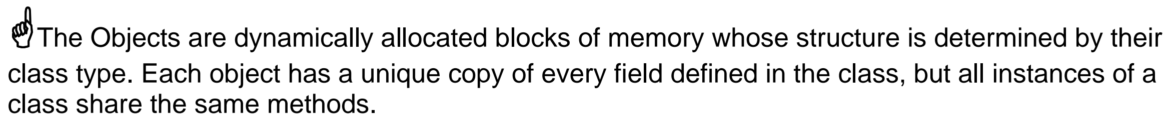


2: The Output Window

The **Compile** button is also used to check that your code is correct, by verifying the syntax before the program starts. When you run this code you will see the content of the include file `(66_pas_eliza_form_sol.inc)` first on the screen and then starts the main program with the form and buttons on it.
To stop this you can switch to the menu options and deactivate the `Show Include` check mark. With escape you close the form. Let's have a look at the form file:

```
6 var
    frm: TForm;
    yourMemo, elizaMemo: TMemo;
    TypeHere, ElizaReply: TEdit;
    histBtn, closeBtn, okBtn: TButton;
    elizalbl, youlbl, titelbl: TLabel;
```

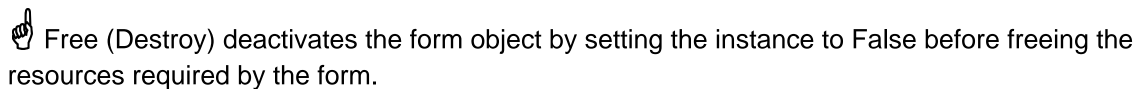
[2] You can only change before

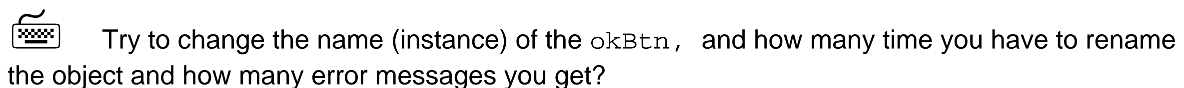

The Objects are dynamically allocated blocks of memory whose structure is determined by their class type. Each object has a unique copy of every field defined in the class, but all instances of a class share the same methods.
For example: `histBtn, closeBtn, okBtn` are 3 objects of the class `TButton`!
Objects are created and destroyed by special methods called constructors and destructors.
The constructor:

```
okBtn:= TButton.create(form);
histBtn:= TButton.create(form);
closeBtn:= TButton.create(form);
```

The destructor (just the Free method):

```
except
  frm.Free;
  result:= NIL;
  exit
end
```


Free (Destroy) deactivates the form object by setting the instance to False before freeing the resources required by the form.


Try to change the name (instance) of the `okBtn,` and how many time you have to rename the object and how many error messages you get?

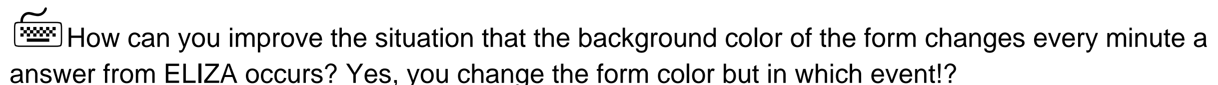
1.3 The Main Routine (ELIZA)

An OP program must have a main routine between begin and end. The main routine is run once and once only at the start of the program (after you compiled) and is where you will do the general instructions and the main control of the program.

```
461 //main program
462 begin
463   consultForm:= loadElizaForm;
464   okBtn.OnClick:= @OkBtnClick;
465   consultForm.Show;
466 end.
```

In Line 463 we create the form with the function `loadElizaForm` and we get and object back to control it namely the `consultForm`. In the next line the ButtonClick is known as an event handler because it responds to events that occur while the program is running. Event handlers are assigned to specific events (like onClick()) by the form files, in our example:

```
procedure OkBtnClick(Sender: TObject);//event handler
begin
  ElizaEngine;
end;
```


How can you improve the situation that the background color of the form changes every minute a answer from ELIZA occurs? Yes, you change the form color but in which event!?

maXbox

I would say that even OOP is tougher to learn and much harder to master than procedural coding, you learn it better after procedural thinking. But after you get in touch with OOP the modular programming is the next and for a long times your last step.
Some notes at last about event-driven. Event-driven programming is best-suited for a program that does not have control over when input may come, such as in a typical GUI or from program. A user can click on any button at any time!

Feedback @
max@kleiner.com

Literature:
Kleiner et al., Patterns konkret, 2003, Software & Support

Links of maXbox and ELIZA:

http://www.softwareschule.ch/maxbox.htm

http://www.softwareschule.ch/

http://www.ask.com/wiki/ELIZA

http://sourceforge.net/projects/maxbox

Aim

Introduction to a new topic

Important Information

Code example

Analysis of Code

Lesson

Test

Summary

Congratulation