Anonymous functions in Free Pascal

Michaël Van Canneyt

July 9, 2022

Abstract

Since some weeks, support for Anonymous functions has been introduced in Free Pascal. In this article, we'll take a look at this new and long-awaited feature.

1 Introduction

Delphi knows anonymous functions since quite some time now. It is one area in which the language compatibility of Free Pascal with Delphi was lacking. Scooter Software sponsored the development of this language feature. A programmer with the name Blaise (coincidence ?) developed it for them. It has been available for their private use since some time, and last year, they shared their code with the Free Pascal team. Sven Barth took it upon himself to integrate and adapt the code for the main compiler branch. Some weeks back, he announced general availability of this feature, much to the joy of many users.

With the exception of some small corner cases, the workings of Anonymous functions in Free Pascal are the same as in Delphi. But Free Pascal offers some extra functionalities, which we'll describe below.

2 What are anonymous functions anyway ?

Well, as the name implies, anonymous functions are functions that do not have a name. There is of course more to it than that, and we'll examine the consequences of this feature in detail.

In pascal, all methods and procedures are called by name, like this:

```
MyInterestingFunction(10,20);
```

Here, `MyInterestingFunction` is the name of a function which is defined prior to this piece of code. But if an anonymous function does not have a name, how can you call it?

The answer is simple: by assigning it to a procedural type variable and executing this procedural variable. The above example could be rewritten as follows with a procedural type since the days of ordinary pascal:

```
Type
   TMyProcedureType = procedure(X,y : integer);

var
  P :  TMyProcedureType;
```

```
begin
  P:=@MyInterestingFunction;
  P(10,20);
end;
```

The effect of this code is the same. The advantage is that you can assign different functions (for instance various color interpolation functions) to a variable of procedural type. Since it is a variable, you can also pass around this function variable to other functions.

As you can see, the variable is assigned using the name of the function. And here is where anonymous functions come into play: what if you could, instead of the name, simply type the declaration of the function you wish to assign ?

Anonymous functions allow just this. You can now type:

```
{$mode objfpc}
{$modeswitch anonymousfunctions}

Type
    TMyProcedureType = procedure(x,y : integer);

var
  P :  TMyProcedureType;

begin
  P:=procedure (x,y : integer)
    begin
      writeln(X,'+',Y,'=',X+Y);
    end;
  P(10,20);
end.
```

There are 2 things to note about this program:

1. The `anonymousfunctions` modeswitch. This enables the use of anonymous functions.
2. This code will not compile in Delphi. Delphi requires a special procedural variable type for this to work. But Free Pascal allows the above code to work because the anonymous function does not reference any symbols from outside the function scope.

Additionally, there are 3 things to note about the anonymous function:

1. There is no name - which was the whole point, of course.
2. There is no semicolon after the procedure or function header: the header is followed by the function body.
3. The anonymous function must appear in the statements of the surrounding block.

The last point means it is not possible to do the following:

```
{$mode objfpc}
{$modeswitch anonymousfunctions}
```

```
Type
    TMyProcedureType = procedure(x,y : integer);

procedure doit;

var
  doPlus : Boolean;
  P :  TMyProcedureType =  procedure (x,y : integer)
begin
  if DoPlus then
    writeln(X,'+',Y,'=',X+Y)
  else
    writeln(X,'-',Y,'=',X-Y)
end;

begin
  for doPlus:=False to True do
    P(10,20);
end;

begin
  Doit;
end.
```

So it is not possible to initialize a variable with an anonymous function. The assignment must happen in the statement block.

Pascal has always known local procedures: in local procedures, you can access symbols of the surrounding function, as shown in the following example, where the variable `doplus` declared outside of `DoPrint` is accessed inside `DoPrint`:

```
procedure DoIt;

var
  doPlus : Boolean;

  procedure DoPrint (x,y : integer);

  begin
    if DoPlus then
      writeln(X,'+',Y,'=',X+Y)
    else
      writeln(X,'-',Y,'=',X-Y)
  end;

begin
  for doPlus:=False to True do
    DoPrint(10,20);
end;

begin
  DoIt;
end.
```

This works, and will print (as expected) the following:

```
10-20=-10
10+20=30
```

But if we want to do this with an anonymous function like in the below:

```
{$mode objfpc}
{$modeswitch anonymousfunctions}

Type
    TMyProcedureType = procedure(x,y : integer);

procedure DoIt;

var
  P :  TMyProcedureType;
  doPlus : Boolean;
begin
  P:=procedure (x,y : integer)
    begin
      if DoPlus then
        writeln(X,'+',Y,'=',X+Y)
      else
        writeln(X,'-',Y,'=',X-Y)
    end;
  for doPlus:=False to True do
    P(10,20);
end;

begin
  DoIt;
end.
```

Then the Free Pascal compiler will complain:

```
ex5.pp(13,6) Error: Incompatible types:
got "anonymous procedure(LongInt;LongInt);"
expected "<procedure variable type of procedure(LongInt;LongInt);Register>"
ex5.pp(26,4) Fatal: There were 1 errors compiling module, stopping
```

Which is strange, because it was possible to use the variable `doplus` in the local procedure, so why not in the anonymous function?

The reason for this failure to compile is that we wish to assign the procedure to a variable. As shown in the following example, the local procedure `DoPrint` also cannot be assigned to a variable:

```
{$mode objfpc}

Type
    TMyProcedureType = procedure(x,y : integer);

procedure DoIt;

var
  P :  TMyProcedureType;
```

```
  doPlus : Boolean;

  Procedure DoPrint(x,y : integer);

  begin
    if DoPlus then
      writeln(X,'+',Y,'=',X+Y)
    else
      writeln(X,'-',Y,'=',X-Y)
  end;

begin
  P:=@DoPrint;
  for doPlus:=False to True do
    P(10,20);
end;

begin
  DoIt;
end.
```

The compiler will complain:

```
ex6.pp(23,6) Error: Incompatible types:
got
"<address of procedure(LongInt;LongInt) is nested;Register>"
expected
"<procedure variable type of procedure(LongInt;LongInt);Register>"
ex6.pp(31) Fatal: There were 1 errors compiling module, stopping
Fatal: Compilation aborted
Error: /usr/local/bin/ppcx64 returned an error exitcode
```

That is because a procedural variable is - behind the scenes - simply a pointer to the address of the function. But in the above case, this pointer is not sufficient to allow calling the function: the function needs to know the address of the `doPlus` variable to do its work, and this information cannot be present in simply one pointer.

In the example above, the compiler could probably still decide that the variable `P` lives only in the procedure `DoIt`, and allow it anyway. In general, the function variable could be passed on to other routines or in some other manner still be 'alive' when `DoIt` exits.

This reason is similar to the reason why you cannot assign a normal procedure to an event handler that is declared with the 'Of Object': the information about the object instance would be missing.

3 Closures: function references

This brings us to closures and function references. We speak of a closure when a function is defined and used together with some elements from its environment: in the above case the variable `DoPlus` is needed for the functioning of the anonymous function.

So, how to solve the above? How can we use `DoPlus` from the environment in our anonymous function?

The answer is a function reference. A new type of procedural variable is now possible:

```
{$ModeSwitch functionreferences}

Type
  TMyProcedureType = reference to procedure(x,y : integer);
```

Note that you need a mode switch for the compiler to accept this definition in `ObjFPC` mode.

When using a function reference, a 'Reference to procedure' type of variable, the compiler knows that it must be prepared to capture the environment of the function assigned to the procedural variable.

You can declare variables (or fields) directly as usual:

```
var
    P : reference to procedure(x,y : integer);
```

Note: this is not possible in Delphi, in Delphi you need to define a type (see later in this article).

Generics will of course also work:

```
Type
  Generic TMyProcedureType<t> = reference to procedure(x,y : T);
```

Armed with this new type, we now can change our program to:

```
{$mode objfpc}
{$modeswitch anonymousfunctions}
{$ModeSwitch functionreferences}

Type
    TMyProcedureType = reference to procedure(x,y : integer);

procedure doit;

var
  P :  TMyProcedureType;
  doPlus : Boolean;
begin
  P:=procedure (x,y : integer)
    begin
      if DoPlus then
        writeln(X,'+',Y,'=',X+Y)
      else
        writeln(X,'-',Y,'=',X-Y)
    end;
  for doPlus:=False to True do
    P(10,20);
end;

begin
  Doit;
end.
```

And now it will compile and run. Whenever `P` is used, the compiler will capture the environment (in this case the variable `DoPlus`) and pass it on to the called routine.

Note the 2 modeswitches at the start of the program:

```
{$modeswitch anonymousfunctions}
{$ModeSwitch functionreferences}
```

We are using 2 different features: anonymous functions, and function references.

To demonstrate that this is actually so, here is the same program, using a local procedure, no anonymous function:

```
{$mode objfpc}
{$modeswitch functionreferences}

Type
    TMyProcedureType = reference to procedure(x,y : integer);

procedure doit;

var
  P :  TMyProcedureType;
  doPlus : Boolean;

  Procedure DoPrint(x,y : integer);

  begin
    if DoPlus then
      writeln(X,'+',Y,'=',X+Y)
    else
      writeln(X,'-',Y,'=',X-Y)
  end;


begin
  P:=@DoPrint;
  for doPlus:=False to True do
    P(10,20);
end;

begin
  Doit;
end.
```

You can assign a local procedure to a 'reference to procedure'. Note that this code is not portable, as this construct is (unfortunately) not possible in Delphi.

In the above examples, the variables `P` and `DoPlus` are still used within the scope of the procedure `DoIt`. So, to show that they are actually used outside the scope of `DoIt`, we will change the program somewhat:

```
{$mode objfpc}
{$modeswitch anonymousfunctions}
{$ModeSwitch functionreferences}

Type
    TMyProcedureType = reference to procedure(x,y : integer);
```

```
Procedure ExecProcedure(A,B : integer;aProc :  TMyProcedureType);

begin
  aProc(4*A,2*B)
end;

procedure doit;

var
  doPlus : Boolean;
  P : TMyProcedureType;

begin
  P:=procedure (x,y : integer)
    begin
      if DoPlus then
        writeln(X,'+',Y,'=',X+Y)
      else
        writeln(X,'-',Y,'=',X-Y)
    end;
  for doPlus:=False to True do
    ExecProcedure(10,20,P);
end;

begin
  Doit;
end.
```

The following output is produced:

```
40-40=0
40+40=80
```

Demonstrating that the current value of `DoPlus` is actually used when executing `ExecProcedure`, while `DoPlus` is not defined in the scope of `ExecProcedure`.

The procedural variable `P` is actually superfluous, it can be omitted:

```
{$mode objfpc}
{$modeswitch anonymousfunctions}
{$ModeSwitch functionreferences}

Type
    TMyProcedureType = reference to procedure(x,y : integer);

Procedure ExecProcedure(A,B : integer;aProc :  TMyProcedureType);

begin
  aProc(4*A,2*B)
end;

procedure doit;

var
```

```
  doPlus : Boolean;

begin
  for doPlus:=false to True do
    ExecProcedure(10, 20, procedure (x,y : integer)
                          begin
                            if DoPlus then
                              writeln(X,'+',Y,'=',X+Y)
                            else
                              writeln(X,'-',Y,'=',X-Y)
                          end
                 );
end;

begin
  Doit;
end.
```

Again, this also works with a local procedure, without anonymous functions:

```
{$mode objfpc}
{$modeswitch functionreferences}

Type
    TMyProcedureType = reference to procedure(x,y : integer);

Procedure ExecProcedure(A,B : integer;aProc :  TMyProcedureType);

begin
  aProc(4*A,2*B)
end;

procedure doit;

var
  doPlus : Boolean;

  Procedure DoPrint(x,y : integer);

  begin
    if DoPlus then
      writeln(X,'+',Y,'=',X+Y)
    else
      writeln(X,'-',Y,'=',X-Y)
  end;


begin
  for doPlus:=false to True do
    ExecProcedure(10,20,@DoPrint);
end;

begin
  Doit;
```

```
end.
```

The above examples are of course very simple. More complex code can be thought of: threads could be started in the local procedure, the procedural variable can be passed on to other routines.

When using a 'reference to procedure', it is not necessary to assign a local procedure or anonymous function to it. It is of course possible to assign normal procedures or methods to a function reference:

```
{$mode objfpc}
{$modeswitch functionreferences}

Type
    TMyProcedureType = reference to procedure(x,y : integer);

Procedure ExecProcedure(A,B : integer;aProc :  TMyProcedureType);

begin
  aProc(4*A,2*B)
end;

var
  doPlus : Boolean;

Procedure DoPrint(x,y : integer);

begin
  if DoPlus then
    writeln(X,'+',Y,'=',X+Y)
  else
    writeln(X,'-',Y,'=',X-Y)
end;

begin
  for doPlus:=false to True do
    ExecProcedure(10,20,@DoPrint);
end.
```

And for methods, the same applies: they can be assigned to references to procedures.

4 3 types of procedural references

In Object Pascal, we now have 3 types of procedural variables at our disposal:

```
Type
  TMyProc = procedure;
  TMyMethod = procedure of object;
  TMyReferenceProc = Reference to Procedure;
```

One might ask: why 3 different types ? Which one must I use ? Why not use a single type that is usable for all cases: calling a plain procedure, a method or an anonymous function ?

After all, if a 'reference to procedure' is usable for all 3 cases, why not simply use that ?

The answer to these questions is two fold:

1. Historical reasons and backwards compatibility: Originally, in Pascal (and Turbo Pascal) only the procedural type existed. It was a simple pointer to an address to be called.

   With Delphi and the introduction of classes, the 'procedure of object' appeared, and under the hood, it was implemented differently (2 pointers are needed instead of 1). For backwards compatibility, the simple procedure reference had to be kept.

   With closures, even more information must be stored, and it depends on the actual function: this is not compatible to the previous 2 types, which again need to be kept for backwards compatibility.

2. Performance. a "Reference to procedure" tells the compiler that the environment of the function assigned to a referenced function must be captured. This capturing happens internally by creating a temporary instance of an interface and a class to back it up. This class is allocated on the heap, as it is an interface, there is reference counting involved etc: It implies lot of overhead which in many cases is simply not necessary and slows down the code.

These 2 reasons show why we now have 3 kinds of procedural types.

If you have an API which makes use of event handlers, you could consider converting the event handlers to 'reference to procedure'. If you are sure that your current API is not used by someone who uses the internal workings of the compiler to work with your API (i.e. uses the pointers directly) you will most likely not break existing code.

But before you rush to convert your APIs to enable anonymous functions, consider whether it is actually useful, because due to the need to capture the environment, there is a non-negligible performance penalty involved, both in Delphi and FPC.

5 Capturing the environment

So, why use function references and anonymous functions ? When you need to capture the environment to be available at a later time, then it makes sense to use function references.

In the above examples, all that was captured was the value of a local variable. But not only local variables are captured: also arguments to the outer function (`DoIt`) can be captured, or the value of `Self` in case of a method.

The following (fictitious) form sets an event handler for a logger class to show log messages. If the logger class can be called from various threads, the actual showing of the log message must be done in a `Synchronize` or `Queue` routine. Function references (and anonymous functions) allow to do this in a straightforward manner:

```
procedure TMainForm.SetupLog;

begin
  MyLogger.OnLog:=@DoGlobalLog
end;

procedure TMainForm.DoLog(Const Msg : String);

begin
  Memo1.Lines.Add(Msg);
end;
```

```
procedure TMainForm.DoGlobalLog(Sender : TObject; Const Msg : String);

begin
  TThread.Queue(TThread.Current,procedure
    begin
      DoLog(Msg)
    end
  );
end;
```

When `TThread.Queue` notices that the current thread is the main thread, it will execute the anonymous function in `DoGlobalLog` at once, so it is "part" of the `DoGlobalLog` routine. At that point, the `Msg` parameter will still be available.

When called from another thread, the `TThread.Queue` call `DoGlobalLog` will actually queue the call to `DoLog` and will return at once, after which `DoGlobalLog` will also exit.

At a later time, when the queued procedure is actually executed in the main thread, the `Msg` parameter to `DoGlobalLog` must still be available. This is what the capture process does: it has saved the value of `Msg` so it is available when the queued anonymous function is executed.

We can of course use a local procedure for this as well:

```
procedure TMainForm.DoGlobalLog(Sender : TObject; Const Msg : String);

  Procedure WriteLog;

  begin
    DoLog(Msg)
  end;

begin
  TThread.Queue(TThread.Current,@WriteLog);
end;
```

To make it clear what the compiler does for you, we'll implement the same functionality without function references. We need somehow to save the message so it is available when the queued call is executed, as well as the form `Self` pointer. The way to do this is to store them in an object, and to let the object call the form `DoLog` method. Here is such an object:

```
Type
  TLogTask = class
    FForm : TMainForm;
    FMsg : String;
    Constructor Create(aForm : TMainForm; const aMsg : String);
    Procedure Invoke;
  end;
```

The constructor is not very exciting, it stores the main form and message parameters:

```
Constructor TLogTask.Create(aForm : TMainForm; const aMsg : String);
```

```
begin
  FForm:=aForm;
  FMsg:=aMsg;
end;
```

The real work happens in the `Invoke` method. In that method, the saved message and form reference are used to actually log the message:

```
Procedure TLogTask.Invoke;
begin
  FForm.DoLog(FMsg);
  Destroy;
end;
```

Note that the `Invoke` calls the destructor `Destroy` : After the message was written, the object must destroy itself, or we would have created a huge memory leak.

Armed with this object, our `DoGlobalLog` routine now becomes:

```
procedure TMainForm.DoGlobalLog(Sender : TObject; Const Msg : String);

var
  aTask : TLogTask;

begin
  aTask:=TLogTask.Create(Self,Msg);
  TThread.Queue(TThread.Current,@aTask.Invoke);
end;
```

The first line creates the `TLogTask` object, passing it the `Self` pointer and the `Msg` message parameter. The second line queues the call to `WriteLog`. Since the reference to `aTask` is lost when `DoGlobalLog` exits, the `TLogTask` object needs to destroy itself.

Another approach would be to store it in an object list and let it remove itself from the list.

The above is actually performing exactly the same tasks as the capturing process, but manually; It gives you an idea of the work the compiler does for you: behind the scenes it creates a class with fields to store all references, and a method to do the work. It uses an interface to handle automatic reference counting - so the object destroys itself when the method has been invoked, whereas the simple case above can use a manual call to destroy: in more complex scenarios it becomes a little more difficult to decide when to free the object.

6 Interfaces

It was hinted at before, that the internal implementation of 'reference to' procedures uses interfaces. In Delphi, this is an implementation detail which is opaque to the user. In free pascal, this is made explicit. The definition

```
Type
  TMyProcedureType = reference to procedure(x,y : integer);
```

Is in fact equivalent to the definition of a reference-counted interface with a single `Invoke` method which has the same signature as the reference type:

```
Type
```

```
  TMyProcedureType = interface(IInterface)
    procedure Invoke(x,y : integer); stdcall; overload;
  end;
```

You can actually use the procedural 'reference to' type as an interface, i.e. you can declare a class with it:

```
Type
  TMyProcedureType = reference to procedure(x,y : integer);

  TMyClass = Class(TInterfacedObject,TMyProcedureType)
    procedure Invoke(x,y : integer);
  end;
```

Because the `Invoke` procedure is declared with 'Overload' it means you can include multiple such implicit interfaces in your class.

Why is this useful? It allows for more control over the capturing process. In the first place, sometimes you may wish to capture more environment than the compiler will capture automatically. You can use this to avoid declaring local variables to provide an environment.

In the following example, an interface is constructed and used as the callback:

```
{$mode objfpc}
{$modeswitch functionreferences}

Type
    TMyProcedureType = reference to procedure(x,y : integer);

Procedure ExecProcedure(A,B : integer;aProc :  TMyProcedureType);

begin
  aProc(4*A,2*B)
end;

Type
  TMyPrecisionImpl = class(TInterfacedObject, TMyProcedureType)
     Prec: LongInt;
     DoPlus : Boolean;
     procedure Invoke(X,Y: integer);
  end;

procedure TMyPrecisionImpl.Invoke(X,Y : Integer);

begin
  if DoPlus then
    writeln(X:Prec,'+',Y:prec,'=',(X+Y):Prec)
  else
    writeln(X:prec,'-',Y:prec,'=',(X-Y):prec)
end;

procedure DoIt;

var
   P : TMyPrecisionImpl;
```

```
   F : TMyProcedureType;

begin
   P:=TMyPrecisionImpl.Create;
   F:=P;
   P.Prec:=3;
   P.Doplus:=True;
   ExecProcedure(10,20,F);
   P.Prec:=5;
   P.DoPlus:=False;
   ExecProcedure(10,20,f);
   F:=Nil;
end;

begin
  DoIt;
end.
```

A second advantage is that because you have the possibility to use an interface, it means you can also control the lifetime of the interface and can hence improve performance: The above function constructs and destroys the interface only once. if the above function was written with an anonymous function, 2 instances of a compiler-generated interface would be constructed and destroyed.

Since the definition of an API determines whether or not a 'reference to' procedure is used, this mechanism can be used to work around some of the drawbacks that this implies.

The use of an interface as the backing mechanism has some consequences:

- it can be implemented by a class (as demonstrated above)
- Inheritance can be used.
- It is a reference counted interface and hence a managed type (So in most cases you class will descend from `TInterfacedObject`).
- The interface has RTTI associated with it.
- The `$M` directive applies to it.
- Contrary to a normal interface, it has no valid GUID and hence cannot be used in `QueryInterface`.

7 Delphi compatibility

FPC's anonymous functions implementation is compatible to Delphi: if it works in Delphi, it will work in Free Pascal. In the previous paragraphs, we've shown that more is possible:

- Anonymous functions can be assigned to variables of regular procedural types. In Delphi, this is not allowed.
- Local procedures can be assigned to function references. This is also not allowed in Delphi.
- You can actually make use of the fact that behind the scenes, a closure is implemented using an interface.

- In Free Pascal you can declare a variable using an anonymous reference function type:

  ```
  var
    P : reference to procedure;
  ```

  In Delphi this is not allowed, you must explicitly declare a type.

  ```
  Type
    TRefProc = reference to procedure;
  var
      P : TRefProc ;
  ```

There is a small corner case where Free Pascal takes a different approach than Delphi.

```
procedure A;
begin
  Writeln('We are in procedure A');
end;

procedure B;
begin
  Writeln('We are in procedure B');
end;

Type
  TProc = reference to procedure;

procedure Test;
var
  p: TProc;
  p2: procedure;
begin
  p2:=A;
  p:=p2;
  p();
  p2:=B;
  p();
end;
```

When compiled with Delphi, this generates the following output:

```
We are in procedure A
We are in procedure B
```

While, when compiled with FPC, it generates the following output:

```
We are in procedure A
We are in procedure A
```

The reason for this is the use of temporary variables during assignments. Delphi generates code that amounts to the following:

```
procedure Test;
```

```
var
  p: TProc;
  p2: procedure;
begin
 p2 := A;
 p := procedure
   begin
     p2();
   end;
 p();
 p2:=B;
 p();
end;
```

While Free Pascal generates code equivalent to the following (note the `tmp` variable):

```
procedure Test;
var
  P : Tproc;
  p2, tmp: procedure;

begin
  p2:=A;
  tmp:=p2;
  p := procedure
    begin
      tmp();
    end;
  p();
  p2:=B;
  p();
end;
```

The example is contrived and unlikely to occur in practice, but it is worth mentioning.

8 A word on readability

We've shown that in Free Pascal, you can use a local procedure instead of an anonymous function when passing it as a 'reference to' procedural type.

Whether you use a local procedure or an anonymous function is a matter of taste: the effect is the same, the only difference is the readability of the code.

As a more elaborate example, take the following piece of code (taken from a Pas2JS RTL unit):

```
Procedure DoFetchURL(URL : String);

  function doOK(response : JSValue) : JSValue;

  var
    Res : TJSResponse absolute response;

  begin
```

```
    Result:=Null;
    If (Res.status<>200) then
      begin
      DoLoadError(Format(SErrUnknownError,[URL,res.StatusText]));
      end
    else
      Res.text._then( @LoadLanguageJson );
  end;

  function doFail(response{%H-} : JSValue) : JSValue;

  begin
    Result:=Null;
    DoLoadError(Format(SErrFailedToLoadURL,[URL]));
  end;

begin
  Window.Fetch(URl)._then(@DoOK,@DoFail).catch(@DoFail);
end;
```

And compare it with the following code which is functionally the same, but uses anonymous functions instead:

```
Procedure DoFetchURL(URL : String);

begin
  Window.Fetch(URl)._then(
    function (response : JSValue) : JSValue

    var
      Res : TJSResponse absolute response;

    begin
      Result:=Null;
      If (Res.status<>200) then
        begin
        DoLoadError(Format(SErrUnknownError,[URL,res.StatusText]));
        end
      else
        Res.text._then( @LoadLanguageJson );
    end,
    function (response{%H-} : JSValue) : JSValue;

    begin
      Result:=Null;
      DoLoadError(Format(SErrFailedToLoadURL,[URL]));
    end).catch(
      function (response{%H-} : JSValue) : JSValue;

      begin
        Result:=Null;
        DoLoadError(Format(SErrFailedToLoadURL,[URL]));
      end
    );
```

```
end;
```

Note that the `doFail` function is duplicated when using anonymous functions, so there may be some advantages to using locally named procedures.

Luckily, in Free Pascal, every programmer can use what he likes most.

9 Conclusion

With the long-awaited arrival of anonymous functions, Free Pascal is again closer to Delphi compatibility. It does more than that and improves the Delphi implementation in some details. We've attempted to show the reasoning behind the implementation, and tried to explain the benefits (and disadvantages) of some of the constructs. To make use of this functionality, you need the main branch compiler: this functionality is not available in any release. How to get and use the main branch compiler has been shown in other articles.