Phil Gilmore

Anonymous Methods and Closures in Delphi 2010

Published 17 June 10 5:9 PM | Phil Gilmore

06/17/2010
Phil Gilmore

What are anonymous methods?
Anonymous methods are a new language feature introduced in Delphi 2010. To describe them in one sentence is unfair, but I'll try. They are methods which are defined inline (you might call them literal methods) and hence are not called by name. Because they are not called by name they do not have names, hence the "anonymous" title.

Like generics they are simple enough in concept, but the reasons behind them can be elusive. The application of the concept will require some real-world exposure. I will start by showing them and then addressing the questions that I think will enter the reader's mind by showing ever more realistic examples.

What do they look like?

Right off the bat, here is an example.

```
procedure
begin
  writeln('Anonymous method has executed');
end
```

This isn't very different from a normal procedure. You'll notice that the procedure has no name. You'll also notice the missing semicolon after the final end. Without a name, we have to call them by reference. To do this, we need a reference to one.

```
var
  myAnonymousMethod: TProc;
```

That's easy. But what's a *TProc*? Just like a reference to a procedure or procedure of object, a method reference must have a type. The type must be a matching procedure or function declaration (I'll stick with procedures for now). Instead of procedure or procedure of object, anonymous methods match signatures declared as *reference to procedure*. Here is the definition of *TProc*, found in *SysUtils.pas*:

```
type
  TProc = reference to procedure;
```

methods. Several are defined in *SysUtils.pas*. They are listed below.

```
type
  TProc = reference to procedure;
  TProc<T> = reference to procedure (Arg1: T);
  TProc<T1,T2> = reference to procedure (Arg1: T1; Arg2: T2);
  TProc<T1,T2,T3> = reference to procedure (Arg1: T1; Arg2: T2; Arg3:
T3);
  TProc<T1,T2,T3,T4> = reference to procedure (Arg1: T1; Arg2: T2; Arg3:
T3; Arg4: T4);

  TFunc<TResult> = reference to function: TResult;
  TFunc<T,TResult> = reference to function (Arg1: T): TResult;
  TFunc<T1,T2,TResult> = reference to function (Arg1: T1; Arg2: T2):
TResult;
  TFunc<T1,T2,T3,TResult> = reference to function (Arg1: T1; Arg2: T2;
Arg3: T3): TResult;
  TFunc<T1,T2,T3,T4,TResult> = reference to function (Arg1: T1; Arg2: T2;
Arg3: T3; Arg4: T4): TResult;

  TPredicate<T> = reference to function (Arg1: T): Boolean;
```

You can also define your own. I've been prone to use the traditional *TGetStrProc* for events, but this is a procedure of object and therefore does not match anonymous method types. I could use *TProc<string>*, but that's not very descriptive. If I wanted something more descriptive, I could define one like this:

```
type
  TAnonGetStrProc = reference to procedure(value: string);
```

Comparison to other method types

There are 3 basic types of subroutines in Delphi 2010. Going forward, it would be useful for you to be familiar with them if you aren't already. Each of the three types are available as both procedures or as functions. The first is the traditional Pascal procedure or function. This is not technically a method, but can be used as a function pointer nonetheless, so I'll cover it. The second type is a regular method. This is the type to which events are bound. The third type is anonymous methods.

Here is an example of the traditional Pascal procedure or function:

```
type
  TWriter = procedure;

procedure CustomWriter;
begin
  writeln('Display this text');
end;

var
  x: TWriter;
begin
  x := CustomWriter;
  x;
  readln;
end.
```

Here is an example of a method:

```
type
  TWriter = procedure of object;

type
  TCustomClass = class(TObject)
  public
    procedure CustomWriter;
  end;

procedure TCustomClass.CustomWriter;
begin
  writeln('Display this text');
end;

var
  x: TWriter;
begin
  with TCustomClass.Create do
  begin
    x := CustomWriter;
    x;
    Free;
  end;
  readln;
end.
```

Here is an example of an anonymous method. Ignore the details for now and just pay attention to the declaration as it compares to the other two types.

```
type
  TWriter = reference to procedure;

var
  x: TWriter;
begin
  x :=
    procedure
    begin
      writeln('Display this text');
    end;

  x;
  readln;
end.
```

We will cover the anonymous method example in the next section. But for now, you can see that each of these method types has a different way of declaring its type and the actual method that matches that type's signature is also declared differently for each type. The firs two types have always been there and should be familiar to you. If not, this is something you should learn before reading further.

Anonymous methods as variables

An anonymous method doesn't have a name, so it must be used with a reference. That reference can be a variable to which the method is assigned. Here is an example.

```
var
  myAnonymousMethod: TProc;
begin
  myAnonymousMethod :=
```

```
    begin
      writeln('This was written from an anonymous method');
    end;

  // ...
end.
```

You can now make further assignments by assigning the *myAnonymousMethod* to another reference. This reference can be another variable, a method parameter, a class field, etc. As long as they are all the same type, assignments can be made as needed. In this assignment, the line ends with a semicolon. This semicolon is denotes the end of the assignment line itself; it is not part of the anonymous method.

Invoking an anonymous method

Let's finish our variable reference example and invoke the method.

```
var
  myAnonymousMethod: TProc;
begin
  myAnonymousMethod :=
    procedure
    begin
      writeln('This was written from an anonymous method');
    end;

  myAnonymousMethod;  // Invoke the method.
end.
```

You can see that this code is not really useful, but it illustrates the following:

- Declaration of an anonymous method reference
- Assignment of an anonymous method to a reference
- Invocation of an anonymous method through its reference.

Because anonymous methods are so natively tied to a reference, it makes sense to use them mostly where passing references is easy to do. Method parameters are class fields are good examples. Let's expand our example to use the *TAnonGetStrProc* prototype and a simple logging example.

Anonymous methods as parameters

So far, our references have only been variables. Modifying the variable example, we can pass a variable reference to an anonymous method as a parameter to another method. This is far more useful than previous examples, as you'll see.

```
type
  TAnonGetStrProc = reference to procedure(value: string);

type
  TMyClass = class
  public
    procedure DoStuff(aLogMethod: TAnonGetStrProc);
  end;


procedure TMyClass.DoStuff(aLogMethod: TAnonGetStrProc);
var
```

```
begin
  try
    // Throws an exception.
    i := StrToInt('invalid integer');
  except
    // Log the error using the method that was passed in.
    // We don't have to define it in this class.
    aLogMethod('Error encountered, logged by anonymous method');
  end;
end;

var
  LogMethod: TAnonGetStrProc;
begin
  // Log method body is defined outside of any code that uses it.
  LogMethod :=
    procedure(value: string)
    begin
      Writeln(value);
    end;  // Semicolon is NOT part of anonymous method.

  // Here, we execute some code and PASS a reference to the log method.
  with TMyClass.Create do
  begin
    DoStuff(LogMethod);
    Free;
  end;

  readln;

end.
```

This is our first useful example. It shows that a method can be passed around and used by other methods in other places. These methods can then invoke this foreign code and pass it parameters if appropriate. We have therefore illustrated inversion of control, callback functionality and code reuse.

Inline declaration

In the previous example, the log method's body is defined OUTSIDE the method that calls it. Instead of the reference being a local variable, it is a method parameter. Okay, so what's so great about that? We can define regular event methods that do that, right? Remember that I said that anonymous methods are declared inline. They don't have to live inside a class at all like a real method. This means that we could change up the initialization section to get rid of the *LogMethod* variable altogether.

```
initialization
  with TMyClass.Create do
  begin
    DoStuff(
      procedure(value: string)
      begin
        Writeln(value);
      end);

    Free;
  end;
end.
```

See that the first parameter is not a variable, but the procedure body itself is stuffed in the call parameter list. This is the cool thing about them. They can be declared on the spot and passed around like any piece of data. They are first-class citizens of the Delphi language. the *DoStuff* procedure could even pass it to another method that it calls.

Inversion of Control (IoC)

Let's focus on the essence of the logging example for a moment. The *DoStuff* method has the ability to log its errors. It has no knowledge of how this logging is done. A method was given to it which presumably knows what to do with a string it is given by the *DoStuff* method.

This method may write the string to a log file on disk, write it to the Windows Event Log, a database, or send it to a logging service like SmartInspect or CodeSite. This method may do nothing with the string at all. The point is that the *DoStuff* method knows nothing about that. It has no control over the logging. It can't decide whether to log to a file, the event log or a logging service. The decision lies elsewhere. This is the fundamental concept of *Inversion Of Control* (or IoC for short).

Inversion of control was possible using standard function pointers, events and object methods in all previous versions of Delphi. However, the concept of IoC was not in vogue until a few years ago. Anonymous methods don't do anything you couldn't do before, but they can be easier than the alternatives.

Class fields

Instead of method parameters which must be passed around and can affect existing method signatures, refactoring may be done by storing anonymous methods in class fields. Here is our previous example, converted.

```
type
  TAnonGetStrProc = reference to procedure(value: string);

type
  TMyClass = class
  protected
    fLogMethod: TAnonGetStrProc;
  public
    constructor Create(aLogMethod: TAnonGetStrProc);
    procedure DoStuff;
  end;

implementation

constructor TMyClass.Create(aLogMethod: TAnonGetStrProc);
begin
  fLogMethod := aLogMethod;
end;

procedure TMyClass.DoStuff;
var
  i: integer;
begin
  try
    // Throws an exception.
    i := StrToInt('invalid integer');
  except
    // Log the error.
    fLogMethod('Error encountered, logged by anonymous method');
  end;
end;


initialization
  with TMyClass.Create(
    procedure(value: string)
    begin
      Writeln(value);
```

```
    begin
      DoStuff;
      Free;
    end;
  end.
```

In the approach above, the *DoStuff* signature does not have to have a method parameter for the log method. Instead, the method can use the *fLogMethod* field, which is initialized in the constructor. No cleanup is needed in the destructor. Now any methods that you add to the class can use the log method without a special parameter just for that. Most importantly, existing methods don't have to have their signatures modified when adding this log method functionality to an existing class.

Note too (as indicated in the comments of the above example) that the anonymous method itself is not followed by a semicolon. I've mentioned this before but it can take some reminding to get used it. It can be confusing because you have seen me repeatedly end an anonymous method with a semicolon, but if you go back and look, the semicolon is there to terminate the assignment of an anonymous method to a variable. It is NOT part of the anonymous method itself.

Generic anonymous type declarations

Before going further, I will briefly cover the generic procedure types which I mentioned earlier. In the next section, I have an example in which I use a variable declaration like this:

```
var
  ShowCounter: TProc<Integer>;
```

This is the notation used by generics. To make sense of it, look for the declaration of *TProc<T>* in *SysUtils.pas*. When I declare something of type TProc<integer>, it just means that I use the declaration of *TProc<T>*, but Delphi replaces all the *T*s in the declaration with integer when I use the *ShowCounter* variable. In other words, given this declaration:

```
TProc<T> = reference to procedure (Arg1: T);
```

... my declaration of *TProc<integer>* tells Delphi that *ShowCounter* uses the following declaration (which the compiler creates at runtime):

```
reference to procedure (Arg1: integer);
```

This is just a way of using an existing method prototype declaration for my anonymous method instead of declaring a new one for myself. It is the same as creating a new type...

```
type
  TIntegerProc = reference to procedure (Arg1: integer);
```

But because *TProc<T>* is already declared, I can use that instead.

Closures

Before I get into closures, I want to address the constant use of the word as a misnomer. The term *Closures* is very frequently treated as a synonym for the term *anonymous methods*. Though closures are specific to anonymous methods, they are not the same concept. Anonymous methods are possible without closures. Do not use the terms interchangeably.

methods. This is because anonymous methods have some considerations for variable scope. The first rule is that all variables which are declared inside the anonymous method are safe and will adhere to all the rules with which you are familiar. Variables declared outside the anonymous method are the ones you need to examine.

In traditional Pascal, anything in a *VAR* block which exists physically above the code which references is visible. These are called nested methods.

```
procedure OuterMethod;
var
  upperScope: string;

  procedure InnerMethod;
  var
    innerScope: string;
  begin
    // Here, upperScope and innerScope are visible.
    // lowerScope is not visible.
    writeln(text);
  end;

var
  lowerScope: string;
begin
  // upperScope and lowerScope are visible here.
  InnerMethod;
end;
```

This works because the *InnerMethod* only lives in one place and is not visible anywhere except in the *OuterMethod* body. But when using anonymous methods, the methods can be passed to another place where the *upperScope* variable is not declared. To accommodate this, Delphi provides closures. Closures create another scope which includes any symbols used by the anonymous method but which are not declared inside the method itself. This scope is then bound to the anonymous method and follows it everywhere it goes.

Without closures, You would use an an upper variable using parameters, like this:

```
procedure TMyClass.DoStuff;
var
  j: integer;
  ShowCounter: TProc<Integer>; // We covered this type in the previous
section.
begin
  ShowCounter :=
    procedure(value: integer)
    begin
      // j is unknown here because it is out of scope.
      // We output a parameter instead.
      writeln(IntToStr(value));
    end;

  for j := 1 to 10 do
    ShowCounter(j);  // Pass j as a parameter
end;
```

You see that the anonymous method has a parameter through which I pass the current counter value. The *writeln* statement will output the value parameter instead of the *j* variable. This would be necessary because *j* does not exist inside of the anonymous method which we assign to *ShowCounter*.

We don't have to do things that way. Because Delphi provides closures, we can have direct access to *j* counter from with the anonymous method without using a parameter. Delphi will place *j* in the scope of

```
procedure TMyClass.DoStuff;
var
  j: integer;
  ShowCounter: TProc; // Method no longer has a parameter.
begin
  ShowCounter :=
    procedure
    begin
      // j is known here because it is wrapped in a closure
      // and made available to us!
      writeln(IntToStr(j));
    end;

  for j := 1 to 10 do
    ShowCounter; // j is no longer passed
end;
```

So now you're wondering why I just spent all this time explaining closures when it works just like a nested method anyway and you don't have to take any steps to make it work or turn it on etc. The reason is because the closure is not really direct access to the symbols in the outer scope. There are limitations. The compiler has to make decisions about where the data comes from and what it can do with that data. Sometimes it will determine that a closure is not possible and will give you an error.

For example, you cannot have a for loop which uses a counter which exists in the closure. Again, modifying the previous example...

```
procedure TMyClass.DoStuff;
var
  j: integer;
  ShowCounter: TProc;
begin
  ShowCounter :=
    procedure
    begin
      for j := 1 to 10 do
        writeln(IntToStr(j));
    end;
  ShowCounter;
end;
```

This seems perfectly safe, but the compiler knows better. Some compiler optimizations for loops require the counter to live on the stack and this one lives inside a closure instead. When you run the code above, the compiler yields this error:

```
[DCC Error] Project1.dpr(30): E1019 For loop control variable must be
simple local variable
```

The bad news? It won't work. The good news? There is a very good compiler error to tell you what's wrong and there is an alternate way to do the same thing. The counter must be local, as the error indicates.

```
procedure TMyClass.DoStuff;
var
  ShowCounter: TProc;
begin
  ShowCounter :=
    procedure
    var j: integer;  // j is now local.
    begin
      for j := 1 to 10 do
        writeln(IntToStr(j));
```

```
  ShowCounter;
end;
```

Because of such considerations, it might be a good idea to avoid closures where possible. But it is important to understand how they work so that you know when you're using them. Because they are so effectively transparent, a solid understanding is necessary to use or avoid them effectively.

Result in closures

The *Result* value cannot live in a closure. For example, the compiler tells you you cannot do this:

```
function TMyClass.AddDays(aDays: integer): TDateTime;
var
  rc: TDateTime;
  p: TProc;
begin
  rc := now;

  p :=
    procedure
    begin
      // Result can't live in a closure.
      result := rc + aDays;
    end;

  p;
end;
```

You get this error, again very descriptive and deliberate:

```
[DCC Error] Project1.dpr(34): E2555 Cannot capture symbol 'Result'
```

Instead, you must fix it like this:

```
function TMyClass.AddDays(aDays: integer): TDateTime;
var
  rc: TDateTime;
  p: TProc;
begin
  rc := now;

  p :=
    procedure
    begin
      // rc can live in a closure.
      rc := rc + aDays;
    end;

  p;
  result := rc;
end;
```

Note that this means that result cannot live only in CLOSURES; the limitation does not apply to anonymous methods themselves. If you are using a an anonymous function (discussed next), *result* is allowed because it is local to the anonymous function, not something which needs to be captured in the closure. Since *result* does not apply to procedures, it is assumed that a reference to *result* inside an anonymous procedure is an attempt to reference the return value in outer scope, hence it requires

Anonymous Functions

So far, I have been careful to use only anonymous procedures. But as you have seen in *SysUtils.pas*, there are also anonymous functions. They work identically, except that there must be a return type in the type definition and in the inline method body itself.

Here is an anonymous function type definition.

```
type
  TIntegerConvertFunc = reference to function(s: string): integer;  //
Has a return type
```

Here is an anonymous function.

```
var
  myFunction: TIntegerConvertFunc;
begin
  myfunction :=
    function(s: string): integer
    begin
      result := IntToStr(s);
    end;
  // ...
```

Generic function declarations warrant mentioning. We've seen the *TProc* and *TProc<T>* declarations in *SysUtils.pas*. However, functions have a caveat. Because the functions have a return type, generic function declarations usually have a parameterized return type. This means that by convention, the last parameter type of any generic function declaration is the function return type, and therefore every generic function declaration has at least one parameter because a function must always have a return type.

Note that I say that this is by convention only. Non-generic function declarations obviously don't apply, and even generic function declarations may have a concrete return type. Furthermore, the type parameter order may not adhere to this convention. You'll find that VCL and .NET both adhere to this convention.

Again here is an overview of the VCL function types:

```
TFunc<TResult> = reference to function: TResult;
TFunc<T,TResult> = reference to function (Arg1: T): TResult;
TFunc<T1,T2,TResult> = reference to function (Arg1: T1; Arg2: T2):
TResult;
TFunc<T1,T2,T3,TResult> = reference to function (Arg1: T1; Arg2: T2;
Arg3: T3): TResult;
TFunc<T1,T2,T3,T4,TResult> = reference to function (Arg1: T1; Arg2: T2;
Arg3: T3; Arg4: T4): TResult;
```

First, note that there are no *TFunc* declarations with no parameters. That is, they all have a return type parameter. The result parameter is always named *TResult*, also by convention. From top to bottom, all these function prototypes are identical except for the number of input parameters. There is a function for all input parameter counts from 0 to 4. If you need more than that, you can declare a new type or perhaps refactor to simplify things.

As an example, an anonymous method matching the last prototype might look like this:

```
  i: integer;
  myFunction: TFunc<string, string, string, string, integer>;
begin
  myFunction :=
    function(s1, s2, s3, s4: string): integer
    begin
      result :=
        StrToInt(s1) +
        StrToInt(s2) +
        StrToInt(s3) +
        StrToInt(s4);
    end;

  i := myFunction('1', '2', '3', '4');
  writeln(inttostr(i));
  readln;
end.
```

You might find, however, that using generics it is easy to simplify this example using a list.

```
myFunction :=
  function(aValues: TList<string>): integer
  var
    j: integer;
  begin
    result := 0;
    for j := 0 to aValues.Count - 1 do
      result := result + StrToInt(aValues[j]);
  end;
```

Anonymous method names

I know... I said they don't have names. More specifically, the programmer does not give them names nor can someone refer to them by name. But internally, the compiler generates code for them just as any other method. When it reports an error, it tries to give you some rough idea of the method to which the error applies. It uses some information from its internal symbol tables to create this. It is useful for you to have an idea what they look like in case you get any errors and don't know what you're seeing. To see one of these, compile this code:

```
var
  f: TFunc<integer>;
begin
  f :=
    function: integer
    begin
    end;
end.
```

There is no return value in the anonymous function, so you will get a warning about it. It shows you the following.

```
[DCC Warning] Project1.dpr(72): W1035 Return value of function
'Project1$ActRec.$0$Body' might be undefined
```

'Project1$ActRec.$0$Body' is the name that it gives to the method in the main body of a console application saved as Project1.dpr. There is no predicting these names and they will change in unexplained ways between projects, etc. Do not attempt to use RTTI to get a handle on these puppies by name. Let the compiler keep them to itself.

Prototype mismatch

If your anonymous method definition doesn't match the prototype of the parameter or variable to which you are assigning it, you will receive a less than perfect error. For example, here is the prototype for *Generics.Defaults.TComparison<T>:*

```
TComparison<T> = reference to function(const Left, Right: T): Integer;
```

Note that the parameters are declared with the *const* keyword. If you don't notice, you might declare a method like the following and assign it to a *TComparison<string>* variable.

```
var
  c: TComparison<string>;
begin
  c :=
    function(left, right: string): integer
    begin
      if (left < right) then result := -1
      else if (left > right) then result := 1
      else result := 0;
    end;
end.
```

In this case, I get an internal error.

```
[DCC Fatal Error] Project1.dpr(16): F2084 Internal Error: AV21F2707E-
R0000000C-0
```

We go back to see what we did wrong and find that we left out that const keyword in the function body. Furthermore, if we make this declarative mistake in other contexts, we'll get a different error. In the following case (indeed most cases), we'll get a slightly more useful error.

```
type
  TMyClass = class(TObject)
  public
    procedure DoStuff;
  end;

var
  c: TComparison<string>;
{ TMyClass }

procedure TMyClass.DoStuff;
var
  c: TComparison<string>;
begin
  c :=
    function(left, right: string): integer
    begin
      if (left < right) then result := -1
      else if (left > right) then result := 1
      else result := 0;
    end;
end;

begin

end.
```

```
[DCC Error] Project1.dpr(33): E2010 Incompatible types:
'TComparison<System.string>' and 'Procedure'
```

This error could probably use improvement because the *Procedure* part never gets any more specific. Obviously, our anonymous method isn't a procedure; it's a function. So the first thing you may do is wonder if the error is referring to some other piece of code (a bug). No, indeed it is referring to your anonymous function. There is nothing wrong with that anonymous function by itself. It just doesn't match *TComparison<string>*.

It would be most useful if the error read something like this instead:

```
Incompatible types: 'TComparison<System.string>' and 'reference to
function(string, string): integer'
```

... or even better ...

```
Incompatible types: 'reference to function(const string, const string):
integer' and 'reference to function(string, string): integer'
```

Again, adding the const keyword fixes it up. In cases like this, copy and paste can save you some confusion. It's usually a dumb mistake or oversight like that.

Predicates

A special type of method is a predicate. A predicate is a filter condition. This means that conventionally, the implementation of a predicate is a function with any number of parameters which returns boolean. *SysUtils.pas* defines the following for you to use for predicates:

```
type
  TPredicate<T> = reference to function (Arg1: T): Boolean;
```

Unfortunately, a grep reveals that this type is used nowhere in the entire VCL. As useful as it is, I find it a tragedy that the new *Generics.Collections* unit isn't riddled with methods accepting parameters of this type. That can't stop you though. Any time you want to allow a method to filter items in a collection, a predicate is a good pattern to use.

Lambda Expressions

Unfortunately, Lambda expressions are not available in Delphi. While I am confident that they will be made available eventually, it is one of the features which I miss the most.

Because I think that it might be useful for users of other languages, I will cover them a bit here, in Prism and C# Syntax. I suspect that Delphi will adhere to the Prism syntax if this feature ever appears

One of the questions that comes up when learning anonymous methods is how they can be useful when they're no less verbose as traditional methods of IoC and delegation. The answer is that anonymous methods don't have to be so verbose. Lambda expressions are a popular shorthand syntax for anonymous methods.

I do a lot of C# and have reaped the benefits of Lambda Expressions and anonymous methods together. You have seen how well anonymous methods play with generics. Add Lambda expressions to the mix and things get very fancy very quickly.

Given a function like this:

```
function(s: string): integer
begin
  result := StrToInt(s);
end;
```

There are three main parts of the function.
- The parameter list
- The return type
- The method body

A Lambda expression is just a shorthand expression of these three elements. The first element is a parameter list. Fundamentally enclosed in parentheses, the parentheses can be omitted if there is only one parameter.

```
s: string
```

The next is the method body. Because there is only one line and it is the result assignment, the assignment to result can be omitted, leaving just an expression to be evaluated. Hence, the "*result :=*" is removed, leaving:

```
StrtoInt(s);
```

Put them together using the "hat" operator, which is read out loud as "such that" or "is a function of" (or whatever else you want), you get this in Prism and C#, respectively:

```
// Prism
s: string -> StrToInt(s)
```

```
// C#
string s => Int.Parse(s)
```

In C# and to some extent in Prism, the type of the parameters and the function result can be determined very effectively by the compiler based on context, alleviating the need to specify these types. This is called "*Type Inference*" and further simplifies the code:

```
// Prism
s -> StrToInt(s)
```

```
s => Int.Parse(s)
```

Multiple statements and parameters can be used, but much of the simplification goes away. Here are examples again in Prism and C#, respectively.

```
// Prism
(s1, s2) ->
begin
  var i := StrToInt(s1) + StrToInt(s2);
  result := i;
end
```

```
// C#
(s1, s2) =>
{
  int i = Int.Parse(s1) + Int.Parse(s2);
  return i;
}
```

While functions are usually used with Lambda expressions, procedures can be used too. If the expression does not evaluate to anything, it generally requires that the result not be assigned to anything.

Parameterless functions and procedures can also be used. Usually, an empty parameter list is provided with a pair of parentheses, (*()* –> ...).

Phil Gilmore (www.interactiveasp.net)

Filed under: Programming, Development Tools, Delphi

Comments

[Spring-DI] MemoryLeak bei Einsatz von DelegatedConstructor - Delphi-PRAXiS said on February 1, 2012 1:42 AM:

> Pingback from [Spring-DI] MemoryLeak bei Einsatz von DelegatedConstructor - Delphi-PRAXiS

Search

- Go

This Blog

- Home

Tags

- .NET 3.5
- Announcements
- ASP.NET MVC
- C#
- Code Generation

- Development Tools
- Linq
- MyGeneration
- Programming
- Source Control
- Visual Studio

Community

- Home
- Blogs
- Media

Archives

- December 2010 (1)
- June 2010 (1)
- May 2010 (1)
- March 2010 (1)
- December 2009 (2)
- June 2009 (2)
- May 2009 (1)
- April 2009 (1)
- March 2009 (1)
- February 2009 (1)
- January 2009 (3)
- November 2008 (1)

Syndication

- RSS for Posts
- Atom
- RSS for Comments

Email Notifications

- Your Email Address Go

Windows Phone 7