Melbourne ADUG Meeting - June 2016

Things you can do with Generics and Anonymous Methods

Colin Johnsun
www.github.com/colinj
@chillijay

Delphi Routine Types

- Standalone functions and procedures
- Nested functions and procedures
- Methods
- Anonymous methods

Procedure Types

- procedure / function pointers

```
TFunc = function(I: Integer): Integer;
TProc = procedure(S: string);
```

- method pointers

```
TFuncMethod = function(I: Integer): Integer of object;
TProcMethod = procedure(S: string) of object;
```

- method references

```
TFuncRef = reference to function(I: Integer): Integer;
TProcRef = reference to procedure(S: string);
```

Procedure Types

	Procedure Pointer	Method Pointer	Method Reference
Nested	No	No	No
Standalone	**Yes**	No	**Yes**
Method	No	**Yes**	**Yes**
Anonymous	No	No	**Yes**

Anonymous Methods

- procedure or function that does not have a name associated with it.

- can be assigned to a variable.

- can be used as a parameter.

- can be returned by functions.

- can access and bind to local variables defined outside of the anonymous function.

Anonymous Methods

- procedure or function that does not have a name associated with it.

```
Add5 :=
function (I: Integer): Integer
begin
  Result := I + 5;
end;
```

Anonymous Methods

- can be assigned to a variable.

```
Add5 :=
function (I: Integer): Integer
begin
  Result := I + 5;
end;
```

Anonymous Methods

- can be used as a parameter.

```
// Filter the list of integers to only even numbers
// Filter function has a parameter of the type
//    reference of function (N: Integer): Boolean
```

```
IntegerList.Filter(
    function (N: Integer): Boolean
    begin
     Result := N mod 2 = 0;
    end
 );
```

Anonymous Methods

- can be returned by functions.

```
type
  TIntFunc = reference of function(I: Integer): Integer;
function Add5Func: TIntFunc;
begin
  Result :=
    function (I: Integer): Integer
    begin
      Result := I + 5;
    end;
end;

...
var Add5: TIntFunc;

Add5 := Add5Func();  // Add5 is an integer function
X := Add5(7);        // Result is X = 12;
```

Anonymous Methods

- can access and bind to local variables defined outside of the anonymous function.

```
function Add(X: Integer): TIntFunc;
var
  N: Integer;
begin
  N := X;
  Result :=
    function (I: Integer): Integer
    begin
      Result := I + N;
    end;
end;

...

Add5 := Add(5);     // Add5 is a function. Captures the value 5.
X := Add5(7);
```

Anonymous Methods

- can access and bind to local variables defined outside of the anonymous function.

```
function Add(X: Integer): TIntFunc;
var
  N: Integer;
begin
  N := X;
  Result :=
    function (I: Integer): Integer
    begin
      Result := I + X;
    end;
end;

...

Add5 := Add(5);     // Add5 is a function. Captures the value 5.
X := Add5(7);
```

Anonymous Methods

Demo

(IncDemo)

Refactoring with functions

```
// A = [1, 2, 3, 4, 5]

  for I := 0 to Length(A) - 1 do
  begin
    A[I] := A[I] * 2;
  end;

// S = ['cat', 'dog', mouse']

  for I := 0 to Length(S) - 1 do
  begin
    S[I] := UpCase(S[I]);
  end;

// Refactor to...

procedure Map(anArray, Fn)
begin
  for I := 0 to Length(AnArray) - 1 do
  begin
    AnArray[I] := Fn(AnArray[I]);
  end
end;
```

Refactoring with functions

```
procedure Map(anArray, Fn)
begin
  for I := 0 to Length(AnArray) - 1 do
  begin
    AnArray[I] := Fn(AnArray[I]);
  end
end;
```

```
// A = [1, 2, 3, 4, 5]

Map(A, function(x): Integer
       begin
         Result := x * 2
       end);


// S = ['cat', 'dog', mouse']

Map(S, function(x): string
       begin
         Result := UpCase(x)
       end);
```

Refactoring with functions

```
// A = [1, 2, 3, 4, 5]

  Sum := 0;

  for I := 0 to Length(A) - 1 do
  begin
    Sum := Sum + A[I];
  end;

  Result := Sum;


// S = ['cat', 'dog', mouse']

  Join := '';

  for I := 0 to Length(S) - 1 do
  begin
    Join := Join + S[I];
  end;

  Result := Join;
```

```
// Refactor to...

procedure Fold(anArray, InitValue, Fn)
begin
  Value := InitValue;
  for I := 0 to Length(AnArray) - 1 do
  begin
    Value := Fn(Value, AnArray[I]);
  end
  Result := Value;
end;
```

Refactoring with functions

```
procedure Fold(anArray, InitValue, Fn)
begin
  Value := InitValue;
  for I := 0 to Length(AnArray) - 1 do
  begin
    Value := Fn(Value, AnArray[I]);
  end
  Result := Value;
end;
```

```
// A = [1, 2, 3, 4, 5]

Sum := Fold(A, 0, function(Acc, X): Integer
                  begin
                    Result := Acc + X;
                  end);


// S = ['cat', 'dog', mouse']

Join := Fold(S, '', function(Acc, X): string
                    begin
                      Result := Acc + X;
                    end);
```

Processing Lists using Functions

- Map - maps a value to another value
- Filter - filter values based on some condition
- Take/Skip - take the first *n* values or skip the first *n* values
- TakeUntil/SkipUntil - take / skip values based on some condition
- ForEach - do some action on items in a list
- ToList - convert resultant items into a new list
- Fold - combine items into a single value

Processing Lists

Demo

(SeqDemo)

Iterating through a List

```
procedure (const DoActionOn: TPredicate<T>)
var
  Item: T;
  Stop: Boolean;
begin
  for Item in aArray do
  begin
    Stop := DoActionOn(Item);
    if Stop then Break;
  end;
end;
```

Define the ValueType

```
TValueState = (vsStart, vsSomething, vsNothing, vsFinish);

TValue<T> = record
  FValue: T;
  FState: TValueState;
end;
```

The Method Signatures

```
TValueFunc<T, U> = reference to function (const Item: TValue<T>): TValue<U>;

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

TIteratorProc<T> = reference to procedure (const P: TPredicate<T>);

TFoldFunc<T, U> = reference to function (const Item: T; const Acc: U): U;
```

Thank you!

www.github.com/colinj