

Understanding and Using Morfik Helper Methods

A General Overview

Mauricio Longo

This paper offers a brief overview of what are Helper Methods and how they can be used in Morfik source code. Both a formal definition of Helper Methods as well as tips and samples of their usage are included.

Copyright © 2008 itTrends Ltda.

All rights reserved. Morfik and its respective logo are trademarks or registered trademarks of Morfik Technology Pty. Ltd. All other registered or unregistered trademarks referenced herein are the property of their respective owners and no trademark rights to the same are claimed.

itTrends Ltda.

www.ittrends.com.br

Overview

Helper methods are one of the new features introduced in Morfik AppsBuilder 2.0. This feature is built-into the Morfik language which provides the underlying semantic structure to both the Object Pascal language syntax of Morfik FX and the Basic language syntax of Morfik BX.

There were so many new features added in Morfik AppsBuilder 2.0 and so many of them visual in nature that it is easy to overlook a feature which is primarily related to coding. Helper Methods are, however, a powerful addition to the toolset available to the Morfik developer allowing for easier, faster coding and much greater flexibility.

Morfik Type System

The Morfik language, whose structure provides the basis for all the supported language syntaxes in AppsBuilder, is a hybrid object-procedural language. This means that even though Morfik supports all concepts of object oriented programming and has an extensive class framework, the language still consider some types as primitives and not as objects. This is usually the case with basic types such as string, integer, double, Boolean, etc.

Morfik has been implemented this way because this has always been the language design which imposes less of a performance penalty when performing operations on such base types as strings and integers. When designing the Morfik language and framework consideration is always given to best empower the Morfik developer so that there is no compromise of the power of the language for ease of use which is then addressed in a separate iteration of the design. What this means is that sometimes there is some time lag between a feature being technically available and it becoming easy to use in the Morfik Framework.

Morfik Watch

For tips and updated information on everything related to Morfik AppsBuilder visit our site.

www.morfikwatch.com

The Function Library

In deciding that basic types in the Morfik language would not be represented by objects, a decision was made to make the language impose as little overhead as possible, by default. In consequence an extensive library of utility functions for working with basic data types was created. While this library allowed developers to create powerful applications it seemed obvious that the non-object data types were not as easy to use while coding because code completion would not be aware of the type of the variable you wanted to work with, since you have to first specify the function you want to call and then pass it a variable as a parameter.

In languages which are totally object oriented (i.e. have no primitive types which are not objects) code completion can be more effective as typing a dot after the name of a variable displays all methods which can be invoked by that variable. Several people at Morfik were not satisfied with the ease of use of the basic data types in the Morfik Framework and runtime library and started to consider

how the same level of productivity could be achieved from code completion using these types as you get when working with classes.

The Morfik Language

The theoretical basis for programming in Morfik is a hypothetical Morfik language. This is a language that has no syntax of its own, but which exists on a semantic level and has all the characteristics of a hybrid object/procedural language. Morfik implements different syntaxes on top of this semantic basis: one for Object Pascal and one for Basic.

The semantic characteristics of this Morfik language is most closely related to Object Pascal which makes this syntax the most logical candidate to be the prime focus of development for the product. This is essentially the reason why Morfik early beta versions show changes to the Object Pascal language while only late in the Beta testing cycle does the Basic language get updated.

An Extensible Type System

The Morfik language allows the user to define new data types based on primitive types such as string, integer, Boolean, class, etc. This is nothing new and it follows in the tradition of hybrid languages such as Object Pascal and C++. This allows developers to have a lot of flexibility but for these new types to be really useful, functions that work with them must also be created.

When you work with classes there is an established way of extending the functionality of an existing class which is to create a new class that descends from the existing one and adding to this new class the desired new features. While this method works quite well, allowing developers to reuse existing code easily it does not really lend itself well to extending existing components of a framework.

Morfik Watch

For tips and updated information on everything related to Morfik AppsBuilder visit our site.

www.morfikwatch.com

How to Extend a Framework?

Consider that in creating a framework of classes we have created a class called *StringList* which is used within over fifty other classes whenever we need to work with multiple strings. A specific developer that has a nice idea for a new feature to be added to the *StringList* class can descend a class called *MyStringList*, of his own, from *StringList*, and add that new feature, however, everywhere in his application wherever he receives a *StringList* from the existing framework classes he will not be able to use his new feature since the object he is receiving is of type *StringList* and not *MyStringList*.

The issues of how to allow a developer to extend the classes in the Morfik Framework and how to make working with primitive types easier for all developers came together in the concept of Helper Methods.

What are Helper Methods?

Helper Methods are methods which can be attached to any pre-existing data type in Morfik Language. What does this mean? It means that the developer can create a Helper Method for a primitive type, the *String* type, for example, which will, then, be callable from any variable of the *String* type in the application. Developers can, also, create new methods for existing classes thus extending the functionality of those classes in a way which is immediately available to them anywhere in an application where that class is used.

Morfik Watch

For tips and updated information on everything related to Morfik AppsBuilder visit our site.

www.morfikwatch.com.

A Helper Method is declared using a very simple syntax which is like the declaration of a global level function whose name has been prefixed with the name of the type to which it applies. Helper Methods take an implicit parameter called *Self* in Object Pascal or *Me* in Basic which can be used within its code to refer to the variable from which it was called.

Creating Helper Methods

Creating a Helper Method is almost identical to creating a method for a class. The following example is a simple Helper Method defined for the *Boolean* type.

```
function Boolean.ToString: string;
begin
  if Self then
    Result := 'True'
  else
    Result := 'False';
end;
```

Notice that this function has the type name prefixed to its own name. Inside the code for the function you can see the *Self* variable being used to determine the value of the variable which was used to call this method. The following Helper Method is an example of how the *ToString* method for the *Boolean* type can be called.

```
procedure Params.Add(AName: String; AValue: Boolean); overload;
begin
  Self.Add(AName, AValue.ToString);
end;
```

Note that this routine is itself a helper method for another type.

Creating Records/Structures with Associated Behavior

Helper Methods can be used to extend existing types or to improve the process of creating useful new types. In the following example we use *Records* (or *structures*) to create an Enumerator type. Enumerators offer a different manner of traversing lists, without using direct numeric indexing of elements. In this case we are creating an enumerator (a record type called *TStringLengthFilter*) which filters the content of a *TStringList* object based on the length of the strings.

```

Type
  ComparisonOps = (GT, LT, EQ);
  ComparisonSet = Set of ComparisonOps;
  TStringLengthFilter = Record
    SpecifiedLength: Integer;
    Comparison: ComparisonSet;
    CurrentItem: integer;
    TheList: TStringList;
  End;

class function TStringLengthFilter.Create(AList: TStringList;
                                         AComparisonSet: ComparisonSet;
                                         ALength: Integer): TStringLengthFilter;

begin
  Result.TheList := AList;
  Result.CurrentItem := -1;
  Result.SpecifiedLength := ALength;
  Result.Comparison := AComparisonSet;
end;

function TStringLengthFilter.MoveNext: Boolean;
begin
  Result := False;
  while (Self.CurrentItem < (Self.TheList.Count-1)) do
  begin
    Inc(Self.CurrentItem);
    if (EQ in Self.Comparison) and
      (Self.TheList[Self.CurrentItem].Length =
       Self.SpecifiedLength) then
      begin
        Result := True;
        Exit;
      end;
    if (GT in Self.Comparison) and
      (Self.TheList[Self.CurrentItem].Length >
       Self.SpecifiedLength) then
      begin
        Result := True;
        Exit;
      end;
    if (LT in Self.Comparison) and
      (Self.TheList[Self.CurrentItem].Length <
       Self.SpecifiedLength) then
      begin
        Result := True;
        Exit;
      end;
  end;
end;

function TStringLengthFilter.GetCurrent: String;
begin
  Result := Self.TheList[Self.CurrentItem];
end;

function TStringList.GetLengthFilter(AComparison: ComparisonSet;
                                       ALength: integer):
  TStringLengthFilter;

begin
  Result := TStringLengthFilter.Create(Self, AComparison, ALength);

```

end;

This code adds a Helper Method, called *GetLengthFilter*, to the *TStringList* type so that it is easy to get an enumerator and apply it to that type. This method takes three parameters which are used in initializing an enumerator for the *TStringList* type. In these parameters the developer will specify which type of comparison (*ComparisonOps*) and a specific length. This enumerator will then filter all the strings that match the provided comparison type and length (i.e. > 5, >=6). The following are some examples of how to use this type.

The following code displays all strings in the *SL (TStringList)* object which have length smaller than 6:

```
Filter := SL.GetLengthFilter([LT], 6);
while Filter.MoveNext do
    ShowMessage(Filter.GetCurrent);
```

The following snippet will list all strings in the *SL (TStringList)* object with length greater than or equal to 5:

```
Filter := SL.GetLengthFilter([EQ, GT], 6);
while Filter.MoveNext do
    ShowMessage(Filter.GetCurrent);
```

Previously, the creation of a type with methods, such as *TStringLengthFilter* would require the usage of classes. With the introduction of Helper Methods this can now be accomplished with the simpler *Record (structure)* type which requires less memory allocation and has simpler initialization.

Type Helper Methods

Morfik allows the creation of Helper Methods which can be called from a Type reference instead of a variable of that type. This is useful when you do not have an initialized variable of the type in the scope of the code you are writing.

Type Helper Methods, for language consistency, follow the same syntax as the Class Methods of the Class type. In the code which showed how to create a *Record/Structure* with behavior, a Type Helper Method was defined to initialize variables of the *TStringLengthFilter* type.

Variable Initialization through Helper Methods

When working with variables of class types, or objects, there is little doubt about how they are initialized; a call to a constructor of the type is required. Working with the primitive types, in the Morfik language, there is no required way of initializing a variable and most of the time this is done just by assigning a value to the variable. When the type is a structure which is used in many different places in an application a procedure or function sometimes is created for the sole purpose of initializing all the structure's fields.

Morfik Watch

For tips and updated information on everything related to Morfik AppsBuilder visit our site.

www.morfikwatch.com

In Morfik you can now use Type Helper Methods to initialize variables of any type. The following snippet shows the code of a Type Helper Method.

```
class function TStringLengthFilter.Create(AList: TStringList;  
                                         AComparisonSet: ComparisonSet;  
                                         ALength: Integer): TStringLengthFilter;  
begin  
    Result.TheList := AList;  
    Result.CurrentItem := -1;  
    Result.SpecifiedLength := ALength;  
    Result.Comparison := AComparisonSet;  
end;
```

This Helper Method is used in another Helper Method, this time for the *TStringList* type in order to initialize an enumerator for a specific list. Notice in the following code snippet how the Create Helper Method is called from a type reference, returning an initialized variable of the *TStringLengthFilter* type.

```
function TStringList.GetLengthFilter(AComparison: ComparisonSet;  
                                     ALength: integer):  
                                     TStringLengthFilter;  
begin  
    Result := TStringLengthFilter.Create(Self, AComparison, ALength);  
end;
```

Conclusion

The introduction of Helper Methods in the Morfik language opens the possibility of enriching the existing Morfik Framework through the addition of methods to existing types. Through the use of this new feature developers can create new and useful types without the need to base all complex types on classes.

The ability to add more power to the primitive types of the Morfik language allows developer's to better take advantage of the hybrid nature of Morfik which allows for the usage of simpler types and when correctly used, more efficient native code.