# The Kylix™ Object Model

*by Ray Konopka, Raize Software, Inc.*

### Table of Contents

## Overview

Borland® Kylix,™ the Borland development environment for the Linux® platform, has been modeled after Borland® Delphi,™ the award-winning Windows® RAD environment. At the core of both environments is the Object Pascal language—a rich, object-oriented language wonderfully suited for development in a RAD environment. Over the years, Borland has continued to enhance the Object Pascal language, with Kylix representing the latest incarnation of the Object Pascal compiler.

A true object-oriented language, Object Pascal has its own object model, which defines the structure and capabilities of objects created using the language. This paper introduces the object model used in Kylix; it does not provide an introduction to object-oriented programming. As a result, this paper is geared towards familiarizing an experienced Delphi, C++, Java,™ or Visual Basic® programmer with the Kylix object model.

Kylix™

white paper

## The object model

Taken together, the object-oriented features of a language are commonly referred to as the language's object model. Kylix inherits its object model from its underlying language. The Object Pascal object model shares many features found in other languages such as C++ and Java, and it introduces some of its own unique features. For example, Kylix classes can be defined to include properties, which are formal declarations of attributes for a class. Properties are the most powerful addition to the object model and represent a significant portion of the CLX™ (component library for cross-platform development) component architecture.

Properties are not the only feature supporting the component architecture. In fact, the entire object model was specifically designed with components in mind. That is, virtually all of the features in Object Pascal originally were introduced to support the construction of components. As a result, every aspect of the object model is somehow affected by components, and vice-versa.

In this paper, we'll take a closer look at the mechanics behind the Kylix object model. The goal of this paper is not to teach you object-oriented programming but to familiarize you with the object-oriented features and syntax of Kylix. Therefore, it is necessary to have at least a basic understanding of object-oriented programming.

## Visibility directives

A key concept in object-oriented programming is *information hiding*. The objective is to create classes in which the implementation details are hidden from the outside world. That is, direct access to the underlying data of a class should be restricted to only those methods within the same class. These methods provide the external interface to the data. Therefore, if the implementation of the underlying data changes, users of the class are unaffected because the method interface that is used to access the data remains constant.

To support information hiding, Kylix provides four visibility directives: **public**, **private**, **protected**, and **published**. If you are familiar with other object-oriented languages, you should recognize most of these directives. It is important to note, however, that the scoping rules in Object Pascal are slightly different than in Java and C++. To help explain the various directives, the following sample units will be used:

```
unit Frames;

interface

type
  TSimpleFrame = class
  private
    FSides: TRect;
    FVisible: Boolean;
  protected
    function GetSides: TRect;
  public
    constructor Create( Bounds: TRect );
    function IsVisible: Boolean;
    procedure Hide;
    procedure Show;
    procedure Draw; virtual;
  end;

  TColorFrame = class( TSimpleFrame )
  private
    FColor: TColor;
  protected
    function GetColor: TColor;
  public
    constructor Create( Bounds: TRect;
                        AColor: TColor );
    procedure Draw; override;
  end;
implementation
end. {=== Frames Unit ===}


unit TxtFrame;

interface

uses
  Frames;

type
  TTextFrame = class( TColorFrame )
  private
```

```
    Text : string;
  public
    constructor Create( Bounds: TRect;
                        AColor: TColor;
                        AMsg: string );
    procedure Draw; override;
  end;
implementation
end. {=== TxtFrame Unit ===}
```

Public is the least restrictive directive. Public items are visible to any program or unit that has access to the unit in which the corresponding class is defined. For example, the IsVisible function of TSimpleFrame is visible to the TxtFrame unit because it uses the Frames unit.

The most restrictive directive is private, which restricts visibility to the *unit* in which the class resides. For example, a TSimpleFrame object declared outside of the Frames unit does not have access to the FSides field. Likewise, the TTextFrame.Draw method (in the TxtFrames unit) does not have access to the FSides field. Since the TColorFrame class resides in the Frames unit, however, its Draw method does have access to FSides. Thus, within a unit, private has the same visibility rules as public. (C++ developers will recognize this as friend classes.)

The rules for the new protected directive fall in between the extremes of public and private. That is, within a unit, protected fields and methods follow the same rules as private and public. But outside of the unit, only the methods of descendant classes have access to protected items. So, a derived class in another unit can access the protected data fields defined in a class. You don't even need the source code for the ancestor class! As an example, the TTextFrame.Draw method has access to its ancestor's GetColor method, but a TTextFrame object does not.

When creating classes, private should be used only for fields and methods that are truly class-dependent or to be hidden from all derived classes. Any fields or methods a programmer might want to access via a descendant class should be declared as protected. If you are unsure, it is better to declare items as protected, thus giving access to descendant classes.

The final directive is published. It was not used in the sample units because its visibility rules are identical to public. The difference between the two is that the published directive instructs the compiler to add extra runtime type information (RTTI) for the items that appear in that section. Because of this, fields defined in the published section must be of a class type, or they must define a property. The published section is used to specify which properties of a CLX component will appear in the Kylix Object Inspector.™ Runtime type information and properties are described in more detail in subsequent sections.

Note that fields and methods declared immediately following the class type heading have a default visibility of published if the class is compiled in the {$M+} state, or if the class descends from a class that was compiled in the {$M+} state. In all other cases, the default visibility is public.

For example, if you create a new form in Kylix and drop a button on the form, you will see that the button declaration appears immediately following the class type heading. In this situation, Button1 is a published field because TForm is a descendant of TPersistent, which is declared using the {$M} directive. Therefore, any unspecified fields, such as Button1, are published.

## Object reference model

In C++, an object's memory can be allocated statically or dynamically. With Kylix, all object instances are dynamically allocated from the heap and referenced via a pointer. However, since all objects are referenced on the heap, Kylix does not require the use of a pointer de-reference operator to access fields of the object instance.

In particular, Kylix uses a reference model that simplifies the syntax for referencing the fields and methods of objects. Specifically, when you reference an object instance, Kylix automatically assumes that you want to de-reference the pointer. Therefore, the caret symbol (^) used to de-reference pointers in Object Pascal is not necessary when de-referencing an object. The following code fragment demonstrates how classes are declared and used within Kylix.

```
program ReferenceModel;

type
  TSample = class
    ID : Integer;
    constructor Create;
  end;

var
  Sample: TSample;

begin
  Sample := TSample.Create;
  Writeln( 'Sample.ID = ', Sample.ID );
  Sample.ID := 101;
  Writeln( 'Sample.ID = ', Sample.ID );
  Sample.Free;       // Call Free to Clean Up
end.
```

By convention, constructors are called Create. When attributes (called properties) of the Sample object are accessed, there is no need to use the de-referencing operator (^) because de-referencing is implied. (For simplicity, an exception block was not used in the above example.)

The implicit de-referencing of Kylix occurs only with objects of class types. Standard pointer variables still must be de-referenced using the caret symbol. To create a class type object, the constructor works as a function returning a pointer to the created object. It is not necessary to use New and Dispose to create and destroy Kylix objects.

## A common ancestor

Take another look at the previous code example, specifically the class declaration for TSample. Notice that this class does not specify an ancestor class. Like Java, all classes in Kylix have a common ancestor. More precisely,

```
type
  TSample = class
    . . .
  end;
```

is equivalent to:

```
type
  TSample = class( TObject )
    . . .
  end;
```

Although there are several advantages to having a common ancestor, the most significant is the fact that all objects can be treated polymorphically. For example, the AddObject method of the TStringList class takes two parameters. The first is a string, and the second is a TObject. Therefore, a TStringList can be used to manage a list of objects. This is significant because the list is able to manage an instance of *any* class.

In addition to its polymorphic advantage, TObject provides a default constructor and destructor. The Create constructor allocates memory for the object instance and initializes all data fields to zero, while the Destroy destructor releases the memory and destroys the object instance. However, as seen in the earlier code fragments, the Free method of TObject should be called when an object is to be destroyed. So free first checks to make sure the object was actually instantiated before attempting to call Destroy.

## Forward class declarations

One of the most frequently asked questions regarding class declarations is how to declare two classes where each class contains a data field of the other class. This particular problem is handled by creating a forward class declaration. The following statement is a forward declaration for the TSample class.

```
type
  TSample = class;
```

Now TSample can be referenced within another class without having to be completely declared. Remember that the complete declaration of TSample must be specified in the same type declaration block in which the forward reference appears.

The QControls unit in Kylix has a fine example of how to use forward class declarations. In the code fragment that follows (taken from that unit), the TWidgetControl class must be declared as forward because the TControl class references the TWidgetControl class through its FParent field. The order of the class declarations cannot be changed to eliminate the need for the forward declaration. Since TWidgetControl descends from TControl, it must come after the declaration of TControl, or TControl must be declared as a forward reference before TWidgetControl.

```
type
  TWidgetControl = class;

  TControl = class( TComponent )
    FParent : TWidgetControl;
    . . .
  end;

  TWidgetControl = class( TControl )
    . . .
  end;
```

Do not be confused by the notation here. This does not result in an infinite recursion of class declarations, as one might suspect. Recall the reference model that is utilized in Kylix—FParent is not a static object but a pointer to an object.

## Virtual methods

Kylix supports two types of virtual methods: virtual and dynamic. The difference between virtual and dynamic is the structure of their corresponding method tables (VMTs and DMTs), the classic speed-versus-size tradeoff. Dispatching dynamic methods is slightly slower than dispatching virtual methods. In turn, dynamic methods are more space-efficient.

To specify the dispatch mechanism used for a method, the **virtual** or **dynamic** directives are used. Once a dispatch mechanism is specified for a method, it cannot be changed in a descendant class. Also, it is perfectly acceptable to use both virtual and dynamic methods in a single class declaration.

Since Kylix uses two different directives to specify how a method is dispatched, how does this affect methods declared in descendant classes? Specifically, how does one override a virtual or dynamic method? To override the functionality of an ancestor method, the **override** directive is used. Override is used regardless of how the ancestor's method was declared. Consider the following program code:

```
program VirtualMethods;

type
  TBase = class
    procedure VirtualProc; virtual;
    procedure DynamicProc; dynamic;
    procedure BrokenChain; virtual;
  end;

  TDesc = class( TBase )
    procedure VirtualProc; override;
    procedure DynamicProc; override;
    procedure BrokenChain; virtual;
  end;

  procedure TBase.VirtualProc;
  begin
    Writeln( 'TBase.VirtualProc' );
  end;

  procedure TBase.BrokenChain;
  begin
    Writeln( 'TBase.BrokenChain' );
  end;


  procedure TDesc.VirtualProc;
  begin
    inherited VirtualProc;
    Writeln( 'TDesc.VirtualProc' );
  end;

  procedure TDesc.BrokenChain;
  begin
    inherited BrokenChain;
    Writeln( 'This method will not be ' +
             'called by TestBroken' );
  end;

  procedure TestVirtual( P: TBase );
  begin
    P.VirtualProc;    // Test virtual method chain
  end;

  procedure TestBroken( P: TBase );
  begin
    // Test what happens when chain is broken
    P.BrokenChain;
  end;

var
  DescObj: TDesc;
begin
  DescObj := TDesc.Create;

  // Displays 'TBase.VirtualProc',
  // then 'TDesc.VirtualProc'
  Writeln( '{==== Testing Virtual Chain ====}' );
  TestVirtual( DescObj );

  // Displays 'TBase.BrokenChain' ONLY
  Writeln( '{==== Testing Broken Chain ====}' );
  TestBroken( DescObj );
  DescObj.Free;
end.
```

The override directive is not used simply for convenience: it is necessary to maintain the polymorphic hierarchy established between the two classes. For example, since the BrokenChain method is re-declared as virtual in the TDesc class, the chain is broken, and the call to TestBroken displays only "TBase.BrokenChain."

## *Abstract methods*

Another feature related to virtual methods involves the first method in a virtual method hierarchy: the declaration of abstract methods. Kylix takes a more formal approach to abstract methods by utilizing the **abstract** directive, which is placed after the method's declaration. This formal designation makes it unnecessary—and illegal—to actually define the method's implementation for this class. For example, the following code declares a class with an abstract method:

```
type
  TSample = class
    . . .
    procedure SomeMethod; virtual; abstract;
  end;
```

Even though SomeMethod is abstract, objects of TSample still can be created (although you will receive a compiler warning if you have warnings enabled). If SomeMethod is called, the application will terminate. The abstract directive is valid only in the class where the method is first declared, and the method must be either virtual or dynamic.

## *Method overloading*

Next on the list is method overloading, which allows class declarations to have more than one method with the same name. Several rules must be followed when overloading methods. For instance, each method must have a different type signature. That is, the types of parameters and return types must be unique to each method. Overloaded methods are identified by the **overload** keyword. The following sample shows how two Update methods can be declared for the same class:

```
type
  TSample = class
  public
```

```
    procedure Update( S: string ); overload;
    procedure Update( I: Integer ); overload;
  end;
```

Note that global functions and procedures can also be overloaded in Kylix.

### *Default parameters*

In addition to method overloading, Kylix also supports default parameters for procedures and functions. Default parameters allow developers to call the procedure or function without specifying values for every single parameter. A common example of a default parameter can be found in the standard Inc routine, which is used to increment an integer value. Assuming X is declared as an integer, the following two statements are valid:

```
Inc( X );      { Increment X by 1 }
Inc( X, 5 );   { Increment X by 5 }
```

To specify default parameters in your own procedures and functions, the parameter must have the form: Name: Type = Value. Therefore, to implement our own custom increment routine, we would write:

```
procedure Increment( var X: Integer;
                     I: Integer = 1);
begin
  X := X + I;
end;
```

When using default parameters with method overloading, it is possible to introduce ambiguous method signatures. Consider the following two methods:

```
procedure P( I: Integer;
             J: Integer = 0 ); overload;

procedure P( Size: Integer ); overload;
```

The compiler will not allow both of these declarations to exist because it is impossible to determine which method should be used in a call such as P( 5 ).

## Method pointers

Method pointers are similar to procedure pointers of function pointers. However, instead of pointing to standalone procedures, method pointers must point to methods of a class. Method pointers are extremely useful when used between two classes. That is, one class contains a method pointer, which is linked to a method of another class. This ability provides the basis for extending an object by *delegation* rather than by deriving a new object and overriding its methods.

This process of delegation is the way Kylix supports component events (e.g., OnClick). Consider the following code fragments, which show the pieces necessary for a scroll bar to control the number of columns used in a list box. The TNotifyEvent type is a method type declaration. (It looks very similar to a normal procedural type declaration except that the *of object* clause is placed at the end.) The TScrollBar class defines a method pointer, FOnChange, of this same type. During design mode, Kylix assigns FOnChange to point to the TForm1.ScrollBar1Change method. Therefore, when the user clicks the scroll bar, the scroll bar component interprets the mouse events and eventually calls the Change method. The Change method checks to see if the FOnChange method pointer actually points to something. If it does, the associated method is executed. By using method pointers, a TScrollBar descendant class does not have to be created to provide this additional functionality.

```
// Extracted from Classes.pas
type
  TNotifyEvent = procedure ( Sender: TObject )
                 of object;
```

```
// Extracted from QStdCtrls.pas
type
  TScrollBar = class( TWidgetControl )
  private
    FOnChange: TNotifyEvent;
  protected
    procedure Change; dynamic;
  end;

procedure TScrollBar.Change;
begin
  if Assigned( FOnChange ) then
    FOnChange( Self );
end;

// Unit1.pas Sample
type
  TForm1 = class( TForm )
    ScrollBar1: TScrollBar;
    ListBox1: TListBox;
    procedure ScrollBar1Change( Sender: TObject );
  private
    { Private declarations }
  public
    { Public declarations }
  end;

procedure TForm1.ScrollBar1Change(Sender: TObject);
begin
  ListBox1.Columns := ScrollBar1.Position;
end;
```

## Class references and virtual constructors

Let's continue our discussion of methods by focusing on class constructors. In Kylix, constructors can be virtual. But, virtual constructors by themselves are not very useful. In order to utilize virtual constructors, Object Pascal supports the notion of class reference types (sometimes called metaclasses).

From a class reference type, a class reference variable can be instantiated and assigned to any class that is assignment-compatible with the referenced class. In the code that follows, the ClassRef variable is a class reference for the TBase class. Therefore, any of the other classes can be assigned to ClassRef because all of those classes are derived from TBase. As the first line after the "begin" shows, ClassRef is legally assigned to the TSample class. And since the constructors for this hierarchy are all virtual, an object instance of TSample is created on the next line.

```
type
  TBase = class
    constructor Create; virtual;
    . . .
  end;

  TDescendant = class( TBase )
    constructor Create; override;
    . . .
  end;

  TSample = class( TBase )
    constructor Create; override;
    . . .
  end;

  // Class Reference Type
  TBaseClass = class of TBase;

var
  ClassRef: TBaseClass;  // Class Reference Variable
  BaseObj: TBase;        // Object Instance Variable

begin
  // Point ClassRef to TSample Class
  ClassRef := TSample;

  // Polymorphically Create TSample Object
  BaseObj := ClassRef.Create;
end;
```

Virtual constructors are very powerful indeed—especially when used with class references. Actually, it is through the use of class references and virtual constructors that the Kylix form designer is able to create components that are dropped onto forms.

## Class methods

Next on our list of object model features is class methods, which are like regular methods except that they can be executed via a class reference. Although class methods also can be called from an object instance, the implementation of class methods cannot reference any instance data, fields, or normal methods of the class type. Constructors and other class methods may be referenced, however. As a result, class methods usually modify global data or return information about the class.

Class methods combine the benefits of belonging to a class with the accessibility of a normal procedure or function. Even though these methods can be called without creating an object instance, class methods are still bound by the access rights specified in the class declaration. Specifically, if a class method is declared in the private section, that method can be called only from within the unit that defines the class.

The following program declares a simple class that contains a class method. First, the class method is invoked without creating a TSample object, and the returned string is displayed. The string displayed is the same one that will be displayed when GetClassName is called through the Obj object.

```
program ClassMethods;

{$APPTYPE CONSOLE}

type
  TSample = class
    ID: Integer;
    class function GetClassName: string;
  end;

class function TSample.GetClassName : string;
begin
  // Not Dependent on Any Class Data
  Result := 'The Sample Class';
end;

var
  Obj: TSample;
  S: string;

begin
  // Invoke the Class Method
  S := TSample.GetClassName;

  Writeln( S );   // Displays 'The Sample Class'

  // Create object instance and
  // call the method the normal way
  Obj := TSample.Create;
  S := Obj.GetClassName;

  Writeln( S );   // Displays 'The Sample Class'
  Obj.Free;
end.
```

### *Class variables*

Note that Object Pascal currently does not support class variables. It is possible, however, to accomplish the same effect by declaring a variable for this purpose in the implementation section of the unit that defines the associated class. The class methods will have access to the unit variable, but the outside world will not.

## Runtime type information

When dealing with polymorphism and hierarchies of classes, there are many times when it is necessary to determine the type of object to which an object reference is referring. Kylix provides an easy way to accomplish this through runtime type information (RTTI). More precisely, the **is** operator provides access to an object's RTTI to determine if an object's type is that of a particular class or one of its descendants. The **is** operator is a Boolean operator that takes as arguments an object instance and a class type. For example, the following method can be used to implement a Copy toolbar button:

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  if ActiveControl is TCustomEdit then
    TCustomEdit( ActiveControl ).CopyToClipboard;
end;
```

The **is** operator is used to check if the currently active control is a descendant of TCustomEdit. If ActiveControl is compatible with TCustomEdit (e.g., TEdit, TMemo, TMaskEdit), then the CopyToClipboard method is called.

Kylix also provides the **as** operator, which uses RTTI to ensure safe typecasting. The statement ActiveControl as TCustomEdit is roughly equivalent to TCustomEdit(ActiveControl). However, the **as** operator goes an extra step to ensure that the typecast is valid. If the typecast cannot be made, an EInvalidCast exception is raised. The **as** operator is used quite extensively in the CLX code, especially in with..do blocks. For example,

```
with ActiveControl as TCustomEdit do . . .
```

could be used instead of

```
if ActiveControl is TCustomEdit then
  with TCustomEdit( ActiveControl ) do . . .
```

## Properties

The most significant feature of the Kylix object model is the formal specification for properties. The concept of properties is not a new one. In fact, properties are a natural aspect of real-world objects. For example, differences between two chairs can be described by differences in their properties such as color, height, and style. In other object-oriented languages (such as C++ and Java), properties are represented in code as data fields in your class definition with corresponding methods to access the data. But since properties are such integral parts of describing objects, Borland has formally introduced properties into the Kylix object model. This formal implementation allows for more complete data privacy and flexible data access. Consider the following class declaration:

```
type
  TChair = class
  private
    FColor: TColor;
    . . .
  protected
    function GetColor: TColor; virtual;
    procedure SetColor( Value: TColor ); virtual;
  public
    property Color: TColor
      read GetColor
      write SetColor;
  end;
```

This partial declaration defines a simple chair class. The reserved word **property** is used to specify a property of the chair, in this case, color. The declaration of the Color property specifies that its type is TColor. The read and write clauses specify the access methods of the property. That is, whenever the value of the property is required, the method that is listed in the read clause is called. Likewise, the method listed in the write clause is used whenever the value of the property is changed.

The read access method is always a function whose return type is the same as the property. By convention, read access methods start with “Get” followed by the name of the property.
The write access method, on the other hand, is always a procedure that takes a single parameter of the same type as the property. A programmer never makes a direct call to an access method. Instead, properties are used like variables, and the compiler takes care of using the correct access method when needed.

The FColor private data field represents the internal data storage for the Color property. In this example, the data types of the data storage and the property are the same. This is not a requirement, nor is it necessary that a property even have an internal data field. In fact, a property may represent a conceptual value that is calculated from other sources. Like access methods, storage fields also have a naming convention: an F is prefixed to the property name.

It should be noted that a property does not need access methods. More precisely, the read and write clauses may specify the internal data field directly. For example, the above property definition could be written as:

```
property Color: TColor
  read FColor
  write SetColor;
```

In this case the GetColor method is not needed. Instead, the Color property gets its value directly from FColor. Although you could also replace the SetColor access method with the FColor field, you would give up quite a bit of power and flexibility in doing so. One of the most powerful features of the implementation of properties of Kylix is that side effects can occur via the access methods. For example, if the user changes the color of a chair, the SetColor method not only stores the new color value in FColor, but also repaints the chair with the new color.

Taking this a step further, if you completely omit the write clause from a property declaration, you create a read-only property. By definition, read-only properties are available only at runtime. Likewise, removing the read clause would create a write-only property, but these instances are rare.

A property can be any type that can be returned from a function. Note that instances of a class also can be returned from a function. Thus, a property could be an object instance of a class. For example, the common font property is actually a class, which has its own properties.

Take another look at the class declaration for TChair. Specifically, notice the usage of the three access directives. Since the internal data fields are part of the implementation of the property, they appear in the private section. How does this affect descendant components? Since these data fields are implementation-dependent, there is no reason to have these fields visible to descendants. Instead, a descendant should access the property itself and not the internal representation. This allows the original component class to change the internal representation without affecting descendant classes.

Access methods for properties can be placed in either the private or protected sections of a class. The advantage of placing them in the protected section and making them virtual is that descendant class easily can modify the side effects of a property without having to redefine the property.

Since access methods by their very nature should not be accessible to an object instance, they should not be placed in the public nor published sections. The property itself should be declared as public (or published) so that the user can access it.

It is also possible to have a single property behave as if it were an array. Array properties have multiple values of the same type, with an individual value being referenced by an index. However, you cannot reference an array property as a whole, like you can with a normal array. Array properties are declared like this:

```
property Colors[ Index: Integer ]: string
  read GetColors
  write SetColors;
```

The access methods look slightly different for an array property. That is, the read access method takes a single parameter that represents the index of the property item to retrieve.

```
function TChair.GetColors( Index: Integer ): string;
```

Likewise, the write access method takes as its first parameter an index, and the second parameter is the new value for the property. Here is an example of a write access method:

```
procedure TChair.SetColors( Index: Integer;
                            Value: TColor );
```

Array properties are quite powerful because the elements in the array can be indexed any way you choose. That is, the index value does not have to be an ordinal data value: the index could be a string type, for example.

## Summary

Fortunately, you do not have to be proficient in the Kylix object model in order to develop fully functional applications. Knowing the enhancements that have been made, however, should give you an appreciation for the power and flexibility of the Kylix compiler. It should also give you a better understanding of how Kylix works, since Kylix and CLX are written in Object Pascal.

Borland

100 Enterprise Way
Scotts Valley, CA 95066-3249
www.borland.com | 831-431-1000