

ZBasic Object Extensions

draft

Version 1.4

Copyright © 2009 Elba Corp. All rights Reserved.

Publication History

February 2009 – First release

Disclaimer

Elba Corp. makes no warranty regarding the accuracy of or the fitness for any particular purpose of the information in this document or the techniques described herein. The reader assumes the entire responsibility for the evaluation of and use of the information presented. The Company reserves the right to change the information described herein at any time without notice and does not make any commitment to update the information contained herein. No license to use proprietary information belonging to the Company or other parties is expressed or implied.

Trademarks

ZBasic, ZX-24, ZX-24a, ZX-24p, ZX-24n, ZX-40, ZX-40a, ZX-40p, ZX-40n, ZX-44, ZX-44a, ZX-44p, ZX-44n, ZX-328n, ZX-1280, ZX-1280n, ZX-1281 and ZX-1281n are trademarks of Elba Corp. Other brand and product names are trademarks or registered trademarks of their respective owners.

Table of Contents

Introduction.....	1
Enabling Object Extensions.....	1
Defining a Class	1
Defining Class Methods	3
Object Creation Issues	4
Object Destruction Issues.....	5
Object Assignment Issues	6
Object Self-reference and Parent Reference.....	6
Explicit Class and Default Namespace Reference	6
Class Sections	7
Static Class Members.....	7
Constants, Enumeration and Structures within a Class.....	9
Inheritance.....	9
Abstract Classes, Abstract Methods	11
Final Classes.....	12
Using Mixins.....	12
Using the Const Attribute for Methods.....	13
Based Objects.....	14

This page is intentionally blank.

ZBasic Object Extensions

Introduction

The purpose of the object-oriented extensions for ZBasic is to bring the benefits of object-oriented programming - inheritance, polymorphism, and encapsulation - to ZBasic users. Although ZBasic historically has been derived from Microsoft Visual Basic (VB6), the decision was made not to implement the VB object-oriented features in ZBasic largely due to its weak implementation of the inheritance model. Since inheritance is one of the primary benefits of object oriented programming, implementing a weak version of it would do a great disservice to ZBasic users, especially those who are getting their first exposure to object-oriented programming. Instead, the realization of ZBasic object-oriented features takes ideas from other, more popular, object-oriented languages (e.g. C++ and Java) and implements them in ways that fit well in the ZBasic language structure.

Enabling Object Extensions

In order to use the object-oriented extensions described in this document, you must enable them. This strategy was chosen so that the additional keywords needed by the extensions do not cause existing programs to fail to compile. Enabling the extensions can be achieved by using the directive `Option Objects` in the first module compiled. Alternately, the compiler option `--objects` has the same effect; it must occur on the command line or in the `.pjt` file before the first `.bas` file processed.

One side effect of enabling object extensions is that it also allows overloaded subroutines and functions to be defined outside of a class. Overloading is the computer science term for defining two or more procedures having the same name but having different formal parameter lists. The compiler selects which of the overloads to invoke based on the actual parameters specified in a particular call.

Defining a Class

The characteristics of an object are described by defining a `Class`. In many ways, defining a class is very similar to defining a structure in ZBasic. However, instead only of having members that are data items, a class generally will also have members that are subroutines or functions. As with a structure, the members of a class may have either `Public` or `Private` visibility attributes. A third visibility attribute, `Protected`, affords an additional dimension of access control (described later) that is useful when one class inherits characteristics from another class.

Similar to defining a `Structure`, a `Class` is defined using the `Class...End Class` sequence. You may define as many classes in a module as you wish. A private class cannot be used by code in other modules while a public class can be.

```
[Public / Private] Class <name>
    <member-definition>
    ...
End Class
```

The ellipsis in the syntax above connotes that there may be zero or more additional member definitions. As with structures, if neither `Public` nor `Private` is specified, the class will be public, meaning that it will be visible outside of the module in which it is defined. After a class is defined, its name may be used as the type in a variable definition. Consider this simple example.

```
Public Class myClass
    Dim i as Integer
    Dim s as String
End Class
```

A `<member-definition>` that describes a data member has the same syntax as that used to define a variable. As with an ordinary variable, a member may be a single data element or it may be an array. The syntax for a member definition is given by the two descriptions below – the first being for a non-array member and the second being for a member that is an array.

```
{Public / Private / Dim} <name> As <type>
```

or

```
{Public / Private / Dim} <name>(<dim-spec-list>) As <type>
```

As with ordinary variables, `Dim` has exactly the same effect as `Private`. The names of the members of each class may be any legal identifier but a particular name may be used only once in each class. The use of a name as a member in one class does not preclude it from also being used as a member name in a different class or structure or as a variable, constant, parameter, etc.

The `<type>` specified for a member may be any of the pre-defined types like `Integer`, `Byte`, `String`, etc. or it may be a user-defined type like an enumeration, a structure or another class. It is important to note that recursive class definitions, with members that are or contain (directly or indirectly) an object of the class being defined, are not allowed. Defining a class that contains one or more members that are of different class types, known as *composition*, is an important aspect of object-oriented programming.

Examples

```
Class MyDate
    Public year as UnsignedInteger
    Public month as Byte
    Public day as Byte
End Class
```

```
Class MyTime
    Public hour as Byte
    Public minute as Byte
    Public seconds as Single
End Class
```

```
Class MyTimeStamp
    Public tdate as MyDate
    Public ttime as MyTime
    Private isCurrent as Boolean
End Class
```

```
Sub Main()
    Dim ts as MyTimeStamp

    Call GetTimeStamp(ts.tdate.year, ts.tdate.month, ts.tdate.day, _
        ts.ttime.hour, ts.ttime.minute, ts.ttime.seconds)
End Sub
```

The example above emphasizes the similarities between structures and classes; except for the different keyword used, the syntax is identical. As with structures, instances of classes (called objects) can be passed to subroutines and functions either by reference or by value. If an object is passed by value, it will be read-only within the receiving procedure.

A variable that is an instance of a class may be assigned to another variable that is an instance of the same class using the standard assignment operator. If the class has members that are an allocated string type or other objects, special, automatically generated “copy routines” are invoked to perform the copying. This achieves the desired result in most cases, but there are certain special cases where the compiler will not produce a logically correct copy. For such cases, there is a way to specify how the object should be copied. See the discussion about assignment constructors below.

Two variables that are the same type of object may be compared for equality or inequality using the standard comparison operators, `=` and `<>`. As with structures, the equality/inequality test is implemented using a byte-by-byte comparison of the content of the two objects. If one or more members of the object are the `BoundedString` type, the byte-by-byte comparison may result in a `False` value even though the strings are identical. This is because the currently-unused portion of the string store may contain byte values that are different between the

two instances being compared. Similarly, comparison of objects containing allocated strings, while allowed, is not recommended because of the likelihood of producing false negative results.

Defining Class Methods

So far, the discussion of classes has been limited to the similarities between structures and classes. However, classes, and instances of classes (objects), are much more powerful than structures. Most of the additional capability results from the ability to associate subroutines and functions with the class. In the parlance of object-oriented programming, a subroutine or function that is part of a class definition is known as a method. We will generally refer to the class' subroutines and functions as methods unless the context requires a clear distinction between a method that is a subroutine and one that happens to be a function.

A method is defined as being part of a class simply by including the desired subroutine/function definition within the class definition, right along with all of the class data members. Consider this simple example.

```
Class MyTime
    Private hour as Byte
    Private minute as Byte
    Private seconds as Single

    Public Sub SetTime(ByVal h as Byte, ByVal m as Byte, ByVal s as Single)
        hour = h
        minute = m
        Seconds = s
    End Sub
End Class
```

This simple class has three data members and one one method, whose ostensible purpose is to set the data members of the object to specific values. A class method is invoked the same way as any "normal" subroutine or function except that the name of the object to be acted upon must also be specified. The example below, which utilizes the class defined above, illustrates how this is done.

```
Dim t as MyTime
Sub Main()
    Call t.SetTime(12, 0, 0.0)
End Sub
```

In this example, the variable `t` is an instance of the class `MyTime`. In the parlance of object-oriented programming, `t` is said to be an object of type `MyTime`.

So far, the example class isn't particularly useful. It has a method for setting the data members of the object but, because the data members are defined to be `Private`, they can't be accessed by procedures other than class methods. Making data members private is an important part of the concepts of information hiding and encapsulation that are central to object-oriented programming. The general idea is that you make most (preferably, all) data members private and then provide *accessor* methods to return data values from the object to the caller and also provide *mutator* methods to allow a caller to modify an object's data members indirectly.

The `SetTime()` method above is a mutator method that happens to provide the ability to modify all of the data members in a single call. Depending on the nature and complexity of the classes that you define, it may be preferable to define mutator methods that modify individual data members, subsets of the class' data members, or all data members. One of the advantages of providing mutator methods for setting data member values is that the mutator method can enforce limits on values of the data members or relationships between values of multiple data members. Such control would be difficult or impossible to enforce if outside entities were allowed to directly modify the data members. Moreover, requiring the use of mutator methods to change data members also allows the freedom to later change the way that the data members are stored without affecting any of the object's users. An example of enforcing data value ranges is shown below in the re-written `SetTime()` method. The class definition has also been augmented with accessor methods for the three data members.

```
Class MyTime
    Private hour as Byte
    Private minute as Byte
    Private seconds as Single
```

```

Public Sub SetTime(ByVal h as Byte, ByVal m as Byte, ByVal s as Single)
    hour = Min(h, 23)
    minute = Min(m, 59)
    Seconds = Min(s, 59.999)
End Sub

Public Function GetHour() as Byte
    GetHour = hour
End Function

Public Function GetMinute() as Byte
    GetMinute = minute
End Function

Public Function GetSeconds() as Single
    GetSeconds = seconds
End Function
End Class

```

Object Creation Issues

When an object is created, the data members have initial values that depend on where the object is defined. This is exactly the same situation as with non-object variables. For example, if an `Integer` variable is defined at the module level, its value is guaranteed to be initially zero. In contrast, if an `Integer` variable is defined within a subroutine or function (without the `Static` attribute), its initial value is indeterminate. `String` variables (both the allocated type and `BoundedString` type) have slightly different rules in that they are guaranteed initially to have zero-length string values no matter where the variable is defined. Similarly, fixed-length strings are guaranteed to have an initial value of a space-filled string with the specified length. These rules apply to the data members of objects (just as they do with structures), again depending on where the object is defined.

Beyond these fundamental initialization guarantees, when you define a class you can also define optional *constructor* methods that initialize the values of some or all of the data members. A constructor method is a subroutine that has the special name `_Create`. You may define several different constructor methods, all having the name `_Create`, but each of the constructor overloads must have different formal parameter lists so that the compiler can distinguish between them.

The expanded `MyTime` class definition below includes several constructor methods and the additional code shows how a particular constructor can be specified when an object is instantiated. It is important to note that when a particular constructor executes, the fundamental initialization (described above) will already have been performed.

```

Class MyTime
    Private hour as Byte
    Private minute as Byte
    Private seconds as Single

    Public Sub _Create()
        Call SetTime()
    End Sub

    Public Sub _Create(ByVal h as Byte)
        Call SetTime(h)
    End Sub

    Public Sub _Create(ByVal h as Byte, ByVal m as Byte)
        Call SetTime(h, m)
    End Sub

    Public Sub _Create(ByVal h as Byte, ByVal m as Byte, ByVal s as Single)
        Call SetTime(h, m, s)
    End Sub

```

```

Public Sub SetTime(ByVal h as Byte = 0, ByVal m as Byte = 0, _
    ByVal s as Single = 0.0)
    hour = Min(h, 23)
    minute = Min(m, 59)
    Seconds = Min(s, 59.999)
End Sub

Public Function GetHour() as Byte
    GetHour = hour
End Function

Public Function GetMinute() as Byte
    GetMinute = minute
End Function

Public Function GetSeconds() as Single
    GetSeconds = seconds
End Function
End Class

Dim t as MyTime = _Create(12)

Sub Main()
    Debug.Print t.GetHour()
End Sub

```

Of course in this particular case, the four separate constructors could be replaced by a single constructor that uses default parameter values. (Just as with regular ZBasic procedures, the compiler automatically inserts the specified default value if you omit one or more of the right-most parameters on a particular invocation.) If that were done, the constructor would have the form shown below.

```

Public Sub _Create(ByVal h as Byte = 0, ByVal m as Byte = 0, _
    ByVal s as Single = 0.0)
    Call SetTime(h, m, s)
End Sub

```

When you define an array of objects, you may specify different constructors for each element of the array as shown in the example below. In this particular case, explicit constructor invocations were specified only for three of the ten elements of the array of objects. The remaining elements are initialized using the *default constructor* – one that can be invoked with no parameters. You may also specify a constructor list for multi-dimension arrays. In that case, the constructors are applied sequentially to the elements of the array with the leftmost index varying the fastest. This corresponds with the way that arrays are laid out in memory.

```

Dim t2(1 to 10) as MyTime = { _Create(12), _Create(), _Create(1, 2) }

```

It should be noted that, although frequently done, it is not necessary to define a default constructor. If you don't define a default constructor, the compiler will automatically supply one for you that performs only the fundamental initialization. A constructor that has one or more parameters, all of which have default values specified, is effectively a default constructor.

Object Destruction Issues

When an object reaches the end of its lifetime, i.e. it goes out of scope, it may need to be “cleaned up”. For example, if one or more data members are allocated string types, the memory allocated for the string store (if any) needs to be freed. The ZBasic compiler generates code that correctly deallocates string members of objects just as it does for individual string variables and structure members that are allocated string types.

Beyond that automatic cleanup activity, you may need to perform additional “cleanup” operations at the end of an object's life. For example, if your object acquires other resources that need to be released, e.g. a semaphore or memory allocated using `System.Alloc()`, you need to handle these operations yourself by writing an optional class method called a *destructor*. A class may have at most one destructor which is a parameterless

subroutine with the special name `_Destroy`. In the code for your destructor you only need to handle the data members that aren't automatically handled by the compiler. For example, you needn't be concerned with string deallocation because that is handled automatically. The automatically generated cleanup code is executed after the code in your destructor executes.

Object Assignment Issues

A third optional method, known as an *assignment constructor*, may be defined to ensure that the assignment of one object to another (necessarily of the same type) is carried out properly. An assignment constructor is a subroutine with the special name `_Assign()` that takes a single `ByVal` parameter with the same type as the class. For most classes, it isn't necessary to define an assignment constructor because the ZBasic compiler correctly handles object assignment. For data members that are allocated strings, the compiler generates code that frees of the destination object member's existing string and then creates a copy of the source object member's string. For data members that are objects, the compiler generates code that invokes the object's assignment constructor or, if it has none, the default assignment process for that object. The same strategy is employed for data members that are structures having allocated string members. However, all other data members the compiler generates code that performs a simple byte-wise copy.

If your class has data members that have special requirements, e.g. a data member that is an address of a block of memory allocated with `System.Alloc()`, you'll need to write your own specialized assignment constructor. It is important to note that if you do provide an assignment constructor, you are responsible for handling *every* data member of the class, not just those that require special treatment. An example of an assignment constructor for the `MyTime` class is shown below. In this case, the explicit assignment constructor is superfluous since it does no more than the default assignment process does. If an assignment constructor is not needed, it is advisable not to create one because it introduces a possible source of problems if new data members are added to the class and the assignment constructor is not updated to handle the new members.

```
Public Sub _Assign(ByVal src as MyTime)
    hour = src.hour
    minute = src.minute
    seconds = src.seconds
End Sub
```

Object Self-reference and Parent Reference

Occasionally, it is necessary or expedient to be able to refer to the particular object instance in a method. For such purposes, the special identifier `_this` is recognized. This identifier can be used as a qualifier on a member reference (e.g. `_this.hour`) or as a prefix to the `DataAddress` qualifier: `_this.DataAddress`.

Similarly, it is sometimes useful to be able to refer to a method of the parent class of an object. For this purpose, the special identifier `_parent` can be used as a prefix to a class method, e.g. `Call _parent.DoSomething()`. This is only necessary, of course, if the class in which the reference occurs has a method with the same signature (name plus parameter types), e.g. it is often useful for invoking a parent's constructor. If needed, the `_parent` prefix can be used multiple times to refer to a particular ancestor class, e.g. `_parent._parent._parent.DoSomething()`.

Explicit Class and Default Namespace Reference

In some cases, it may be necessary to refer to a method or data member of a specific class, overriding the method or member that would otherwise be matched by a particular name. You can refer to the namespace of a specific class by adding a class name prefix to the identifier. The class name prefix has the form of the class' name followed by two colons, e.g. `MyTime::hour`. The class name prefix is useful inside of class methods as well as in non-class subroutines and functions and for referring to static methods of a class.

Occasionally, it may be necessary to refer to an identifier that is outside of a class when the class contains a method or data member having the same name. Adding the default namespace prefix (two colons) to the identifier allows you to refer to the namespace outside of the class hierarchy, e.g. `::foo`.

Class Sections

When defining a class, the various members have default visibility attributes identical to those that would apply if the same type of member were being defined outside of a class. For example, if a data member is defined using `Dim`, it will be private. Similarly the default visibility for a method is public just as it is for subroutines and functions outside of a class.

The differing default visibility attributes for different types of member can lead to confusion about the visibility of a particular element of a class. To mitigate this issue, one could adopt the practice of specifying the `Public` or `Private` attribute for every member. An alternative strategy is to use section labels in the class to override the default visibility of the members in the section. The `MyTime` class definition is rewritten below using section labels.

```
Class MyTime
Public:
    Sub _Create(ByVal h as Byte = 0, ByVal m as Byte = 0, _
                ByVal s as Single = 0.0)
        Call SetTime(h, m, s)
    End Sub

    Sub SetTime(ByVal h as Byte = 0, ByVal m as Byte = 0, _
                ByVal s as Single = 0.0)
        hour = Min(h, 23)
        minute = Min(m, 59)
        Seconds = Min(s, 59.999)
    End Sub

    Function GetHour() as Byte
        GetHour = hour
    End Function

    Function GetMinute() as Byte
        GetMinute = minute
    End Function

    Function GetSeconds() as Single
        GetSeconds = seconds
    End Function

Private:
    Dim hour as Byte
    Dim minute as Byte
    Dim seconds as Single
End Class
```

Section labels may be used in any order and may appear multiple times. The order used in the example above is suggested but any suitable arrangement may be chosen. It is important to remember that private data members and methods are only visible to methods of the class. This is in contrast to the situation with structures where private members are visible to all procedures within the same module but not visible outside of the module in which the structure is defined.

Static Class Members

So far, all of the methods and data members defined in the examples have been object-specific. That means that invoking a method or accessing a data member had to be realized via an object reference, e.g. `t.SetTime()`. Within the methods themselves, the methods and data members of the class can be accessed without an explicit object reference because the object reference is implicit.

It is possible to define methods and data members that can be accessed without any object reference, explicit or implicit. To do so, simply add the `Static` keyword to the definitions as illustrated in the example below.

```

Class MyTime
Public:
    Sub _Create(ByVal h as Byte = 0, ByVal m as Byte = 0, _
                ByVal s as Single = 0.0)
        Call SetTime(h, m, s)
    End Sub

    Sub SetTime(ByVal h as Byte = 0, ByVal m as Byte = 0, _
                ByVal s as Single = 0.0)
        hour = Min(h, 23)
        minute = Min(m, 59)
        Seconds = Min(s, 59.999)
    End Sub

    Function GetHour() as Byte
        GetHour = hour
    End Function

    Function GetMinute() as Byte
        GetMinute = minute
    End Function

    Function GetSeconds() as Single
        GetSeconds = seconds
    End Function

    Static Function Fmt(ByVal h as Byte, ByVal m as Byte, _
                        ByVal s as Single) as String
        Fmt = CStr(h) & ":" & CStr(m) & ":" & ::Fmt(s, 2)
    End Function

Private:
    Dim hour as Byte
    Dim minute as Byte
    Dim seconds as Single
End Class

Sub Main()
    Debug.Print MyTime::Fmt(3, 10, 15.7)
End Sub

```

In this example, a static method named `Fmt()` is defined that produces a formatted time string. There are several important aspects of the example to note. Firstly, you may recall that there is a ZBasic System Library function called `Fmt()`. This example illustrates how you can create an overload for a System Library routine. Secondly, within the new `Fmt()` method, it was convenient to be able to use the System Library function of the same name. The example code shows how you can use the default namespace prefix to specify that you want to invoke the System Library routine rather than the `Fmt()` method of the `MyTime` class. Lastly, the invocation of `Fmt()` in the `Main()` subroutine shows how the classname prefix is used to specify that you want to invoke the `Fmt()` method of the `MyTime` class.

A static method may invoke only other static methods and procedures defined outside of the class. Also, it may access only static data members and other data items defined outside of the class. It cannot access non-static data members because those are specific to an instance.

A static data member can be similarly defined for a class. Access to the data member is achieved using the classname prefix just as with the static method. It is important to note that there is only one instance of each static data member in the application no matter how many objects of the class containing the static data member have been instantiated. The effect is the same as if a module-level variable were defined outside of the class; the advantage of making it part of the class is that it associates the static data item with the class to a greater degree than would be the case if a module-level variable were defined.

It is important to realize that a static method generally must be public for it to be of any value. If it were private (or protected), it would only be accessible to class (and subclass) methods. For the most part, the same is true for static data members although there may be special cases where private (or protected) static data items would be useful.

Constants, Enumeration and Structures within a Class

It is possible to define constants, enumerations and structures within a class definition. Such definitions may be private if it is desirable that they are only available to class methods or they may be public if that is useful.

Inheritance

The example classes described thus far have derived all of their characteristics and capabilities from the methods and data members explicitly defined as part of the class. One of the great benefits of object-oriented programming comes from the ability to define a class that is based on the functionality of an existing class. Essentially, the idea is to take an existing class and extend it by adding some new capability without affecting the users of the existing class. Consider the example below.

```
Class A
Public:
    Enum Color
        Red
        Green
        Blue
    End Enum

    Sub _Create(ByVal s as Byte = 0, ByVal c as Color = Red)
        m_size = s
        m_color = c
    End Sub

    Sub Identify()
        Debug.Print "I'm an A, size="; m_size
    End Sub
Private:
    Dim m_size as Byte
    Dim m_color as Color
End Class

' Define a new class derived from an existing class.
Class B Extends A
Public:
    ' Define a constructor invoking a base class constructor.
    Sub _Create(ByVal s as Byte = 0, ByVal c as Color = Red, _
        ByVal w as Byte = 1)
        Call _parent._Create(s, c)
        m_weight = w
    End Sub

    Sub ShowWeight()
        Debug.Print "weight = "; m_weight
    End Sub

Private:
    Dim m_weight as Byte
End Class

Dim a1 as A = _Create(5, A::Green)
Dim b1 as B = _Create(10, B::Red, 100)

Sub Main()
    Call a1.Identify()
```

```

    Call b1.Identify()
    Call b1.ShowWeight()
End Sub

```

This example illustrates how the derived class B inherits some of its functionality from its base class A. The object `b1` has a method named `Identify()` even though it is not explicitly present in the definition of class B; it inherits that method from its base class. This example also shows how to invoke a particular base class constructor in the body of the constructor of a derived class - the `_parent` prefix is used to refer to a constructor in the parent class. The base class constructor invocation, if present, must be the first statement in the constructor. If you don't explicitly invoke a base class constructor, the compiler will automatically include code that invokes the default constructor of the base class. You may also invoke a constructor of particular ancestor class instead of a constructor of the base class.

It is important to be aware that the private methods and data members of the base class are not accessible to the derived class. Because it is often useful for derived classes and base classes to share some method and data members that are not publically available, ZBasic supports a third visibility attribute called `Protected`. Base class methods and data members that have the `Protected` visibility attribute are not accessible to code outside of the class methods but they are accessible to code in the methods of a derived class. You can use the `Protected` keyword within a class definition anywhere that you can use the `Public` and `Private` keywords, including in section labels.

If you compile and run the example code above, you'll note that the invocation of `b1.Identify()` displays the same output as when `a1.Identify()` is invoked. It is likely that this behavior is not really what is wanted. Rather, it is more likely that the objects would identify themselves differently. This can be corrected in several ways. Perhaps the most obvious way is to add a protected data member that contains all of or part of the identity string or some value to otherwise identify the object. While this will work, it is wasteful of space because every instance of the object will contain that data member. A better way to solve this problem is to add an `Identify()` method to the B class. A re-written version of the previous example appears below which has an `Identify()` method defined in both classes. The `m_size` data member was also made `Protected` so that it can be accessed by subclasses such as class B.

```

Class A
Public:
    Enum Color
        Red
        Green
        Blue
    End Enum

    Sub _Create(ByVal s as Byte = 0, ByVal c as Color = Red)
        m_size = s
        m_color = c
    End Sub

    Sub Identify()
        Debug.Print "I'm an A, size="; m_size
    End Sub
Protected:
    Dim m_size as Byte
Private:
    Dim m_color as Color
End Class

' Define a new class derived from an existing class.
Class B : A
Public:
    ' Define a constructor using a base class constructor.
    Sub _Create(ByVal s as Byte = 0, ByVal c as Color = Red, _
        ByVal w as Byte = 1) : _Create(s, c)
        m_weight = w
    End Sub

```

```

Sub Identify()
    Debug.Print "I'm a B, size="; m_size; ", ";
    Call ShowWeight()
End Sub

Sub ShowWeight()
    Debug.Print "weight = "; m_weight
End Sub

Private:
    Dim m_weight as Byte
End Class

Dim a1 as A = _Create(5, A::Green)
Dim b1 as B = _Create(10, B::Red, 100)

Sub Main()
    Call a1.Identify()
    Call b1.Identify()
End Sub

```

It is important to understand that when a class is derived from another class, the compiler generates code to ensure that the base class object is fully initialized before the constructor for the derived class executes. Similarly, for assignment constructors the base class assignment constructor is executed before that of the derived class. This order guarantees that the derived class object may safely access the methods and members of the base class in its constructors.

The situation with destructors is similar but the order is reversed. In the case of destructors the derived class destructor executes before the base class destructor. This order guarantees that the derived class may rely on the integrity of the base class data members if necessary.

One aspect of inheritance that must be carefully considered is that since the base class constructor executes before a derived class constructor, base class constructors must avoid invoking methods that would result in the derived class method executing before derived class' constructor has executed.

Abstract Classes, Abstract Methods

When designing an object-oriented solution, it is often useful to define a class that embodies some essential characteristics and functions that can then be inherited by several other classes. In some situations, the base class is "incomplete" in the sense that its only purpose is to serve as the base class for other classes. You can prevent such an incomplete class from being instantiated (i.e. defining an object of that type) by defining the base class to be *abstract*. This is done by adding the `Abstract` attribute to the class definition.

```

Abstract Class MyObject
Public:
Protected:
    Dim m_size as Integer
    Dim m_weight as Integer
Private:
End Class

```

If you try to create an object of the class `MyObject`, the compiler will produce an error message indicating that you cannot instantiate an abstract class. This is to remind you that the purpose of the class is as a building block for other classes.

You can also indicate that a class is abstract by defining the class with at least one abstract method as in the class definition below.

```

Class MyObject
Public:
    Abstract Sub Identify()

```

```

        End Sub
Protected:
    Dim m_size as Integer
    Dim m_weight as Integer
Private:
End Class

```

The result of both of the preceding class definitions is an abstract class. The difference is that when you define abstract methods, in addition to making the class abstract it also establishes a requirement that all derived classes must define method that has the same signature, i.e. the same name and same number and types of parameters.

If you wish, you may include code in the body of the abstract method. One reason for doing so would be to define some essential functionality that many or most derived classes could use. The derived class method can invoke the abstract base class method using the class namespace prefix or the parent class prefix. For example, class A could be derived from `MyObject` and the `Identify()` methods of the `MyObject` and `A` classes could be rewritten as shown below, respectively. You should compile and run such an example to observe the result.

```

Abstract Sub Identify()
    Debug.Print "Howdy, ";
End Sub

Sub Identify()
    Call MyObject::Identify()
    Debug.Print "I'm an A, size="; m_size
End Sub

```

Final Classes

The classes described thus far have the characteristic that they may be (or, in the case of `Abstract` classes, must be) extended. It is also possible to define a class that may not be extended. This is done by using the `Final` attribute preceding the `Class` keyword.

```

Class A
Public:
    Sub Identify()
    End Sub
Protected:
    Dim m_size as Integer
    Dim m_weight as Integer
Private:
End Class

Final Class B Extends A
Public:
    Sub Identify()
    End Sub
Protected:
Private:
End Class

```

Using Mixins

Earlier in this document, the concept of inheritance (defining a class that inherits functionality from another class) was discussed. Also, the concept of composition (defining a class containing one or more data members that are other objects) was discussed.

To review, if class B is defined as inheriting from class A, then class B is said to satisfy the “*is a*” relationship (sometimes written as ISA) with respect to class A. This means that an object of type class B can be treated as

if it is, in fact, an object of type class A. The public and protected methods and data members of class A are available to all methods of class B as if they were directly part of the definition of class B, etc.

In contrast, if class B is defined with a data member of type class A (i.e. using composition), then class B would be said to satisfy the “*has a*” relationship (sometimes written as HASA) with respect to class A. Only the public methods and data members of class A would be available to the methods of class B and the access would need to be qualified using the name of the data member.

The concept of a *mixin* combines aspects of both inheritance and composition. If class B includes class A as a mixin, then the public methods and data members of class A are accessible to the methods of class B as if they were directly part of the definition of class B (as with inheritance) but class B satisfies the HASA relationship with respect to class A (as with composition). Because you can specify multiple mixin classes, doing so provides some of the benefits of multiple inheritance but without having the difficult issues that arise with traditional multiple inheritance.

Including one or more other classes as mixins of a given class is accomplished using the `Includes` keyword as shown in the example below. If more than one mixin class is specified, the mixin class names are comma-separated.

```
Class A
Public:
    Sub DoSomething()
    End Sub
Protected:
Private:
End Class

Class B Includes A
Public:
    Sub Identify()
        Call DoSomething()
    End Sub
End Class

Dim MyB as B

Sub Main()
    Call MyB.DoSomething()
End Sub
```

One restriction on using mixins is that the public and protected methods and data members of each mixin must be unique among all of the mixins included and must not duplicate any method or data member names of the including class. This restriction is necessary because all public and protected methods and data members need to reside in the namespace of the including class and therefore must be unique. It is possible, however, to have methods of the same name in multiple mixins and/or the including class as long as they each have unique signatures, i.e. they must have different number and/or types of parameters. Note, however, that the including class may contain methods that overload methods of mixin classes. It is important to be aware that a mixin class method that is marked as `Abstract` *must* be overloaded by the including class.

Finally, if a class is defined using both inheritance and mixins, the `Extends <class name>` specification must appear before the `Includes <class name list>` specification, e.g.

```
Class D Extends B Includes A, C
Public:
    Sub DoSomething()
    End Sub
End Class
```

Using the Const Attribute for Methods

In earlier discussion, it was stated that an object may be passed to a subroutine or function `ByRef` or `ByVal`. If it is passed `ByVal`, the object instance is considered to be “read-only” within the receiving procedure. As such,

the procedure has read access to the public data members of the object but it is not allowed to modify those data members. Also, the receiving procedure may not pass the object `ByRef` to any other procedure or method. Additionally, in order to invoke a method of the object, that method must be declared to be “constant”, meaning that it isn’t allowed to modify the object or pass the object by reference to another method or procedure.

To define a class method as constant, add the keyword `Const` following the closing parenthesis of the parameter list. Optionally, for methods that are functions, you may instead place the `Const` keyword after the function’s return type. An example is shown below using a previously defined method.

```
Sub Identify() Const
    Call MyObject::Identify()
    Debug.Print "I'm an A, size="; m_size
End Sub
```

Of course, since the method above invokes the base class method of the same name, the latter method must also have the `Const` attribute. In summary, a `Const` method may invoke only other `Const` methods and may not modify any data members or any data members of the base class.

Based Objects

It may be useful in some situations to be able to define an object at a specific memory address. For example, you may want to allocate some memory using `System.Alloc()` to hold an object. Or, you may wish to define an object that occupies previously allocated space, such as a pre-defined buffer. In either event, the desired effect can be achieved by defining an object based at a given address. Consider the example below that uses the class `B` defined earlier.

```
Dim buf(1 to 20) as Byte

Sub Main()
    Dim b1 as B Based buf.DataAddress
    Call b1._Create(5, B::Blue, 25)
    Call b1.ShowWeight()
    Call b1._Destroy()
End Sub
```

There are several important aspects of this example that should be clearly understood. Firstly, when you create a based object you are responsible for all aspects of its management. You must ensure that sufficient space for the object exists at the address at which you base the object. The example above is poorly coded because the buffer may, in fact, be too small for the object. The example could be improved by replacing the upper bound of the buffer with `SizeOf(B)`. Secondly, you must explicitly invoke the constructor for the object before using any of its methods or data members because, prior to the constructor invocation, the object content is completely undefined. Note that explicit constructor invocation is disallowed for non-based objects but is (usually) required for based objects. Also, you are also responsible for invoking the object destructor (if necessary). Failure to do so may lead to memory “leaks”, i.e. allocated memory blocks that are not properly freed.

A second example, below, illustrates using a based object with allocated memory. Other than the means by which the space for the object is acquired, the issues are identical to those of the previous example.

```
Sub Main()
    Dim addr as UnsignedInteger
    addr = System.Alloc(SizeOf(B))
    Dim b1 as B Based addr
    Call b1._Create(5, B::Blue, 25)
    Call b1.ShowWeight()
    Call b1._Destroy()
    Call System.Free(addr)
End Sub
```

Another use for a based object is to be able to treat a base class object as if it were a derived object. Here again, it is your responsibility to ensure that such treatment is appropriate. The compiler will not issue any error messages or warnings about incompatible classes. As an example, consider the subroutine below.

```
Sub IdentifyObject(ByVal obj as A)
    Call obj.Identify()

    ' Define an object based on the object passed
    Dim objB as B Based obj.DataAddress
    Call objB.ShowWeight()
End Sub
```

If we knew, in certain cases, that the passed object were actually of class B (derived from class A), we could use a based object definition to be able to access the unique methods and data members of the B class. This capability is rarely needed but it is good to keep it in mind for those cases where it is. It is important to distinguish this particular use of a based object from the use case given in the first example in this section. In this case the object is assumed to be validly constructed and, therefore, the constructor/destructor should not be invoked on the object as it is in the first example where the based object was defined as overlaying memory of undefined content.

One final note, you may optionally include the `Const` keyword between the class name and the `Based` keyword to define a read-only based object. With such a read-only based object definition, access is limited to `Const` methods and, moreover, data members may not be modified nor passed by reference to other procedures.

BasicX Compatibility Note

The object-oriented features are not supported in BasicX compatibility mode.