

# The REALbasic Language Revisited

*Two Years in the Evolution of Programming Parlance*

BY MATT NEUBURG



The *basic* in REALbasic's name is somewhat misleading. REALbasic is a modern object-oriented garbage-collected language. It rivals respected languages such as Java in elegance and power—particularly in light of its many changes since the publication of the second edition of my book, *REALbasic: The Definitive Guide* (O'Reilly, 2001). That book documented REALbasic 3.5; now, two years later, version 5.5 is in beta. This article describes some of the changes to REALbasic.

Note that I'm talking here just about the language. Naturally, the development environment, the repertory of built-in classes and interface widgets, and so forth have also undergone many improvements.

## Ground of Being

All REALbasic code resides in a method of some *class*, or of a global entity called a *module*. A class or module can also define *properties* (instance variables) and *constants*. The ability to define constants

in a class is new; previously, you had to define constants globally in a module or locally in a method.

In code, you call a method, or refer to a property, by means of dot notation appended to a reference to an instance of that class. Here, `MyClass` has a string property, `itsString`, and a method, `itsStringMaker`, that returns a string:

```
dim c as MyClass
c = new MyClass
c.itsString = c.itsStringMaker()
```

References to instances are pointers, but REALbasic hides this fact from the programmer. Garbage collection is by means of reference counting; the object to which the automatic variable `c` in the above code points, if there are no other references to it, is destroyed spontaneously when `c` goes out of scope.

A class is a subclass of some other class (with the object as the ultimate superclass), and inherits its superclass's

methods and properties. A subclass may add properties but cannot refuse an inherited property. Any inherited methods may be overridden, with polymorphism implemented in the usual way (the search for a method starts with the class of the instance to which a message was originally sent).

There are also class interfaces, which are essentially collections of method declarations. A class that implements a class interface must implement all of its methods. A class may implement any number of class interfaces.

Two methods of the same class may have the same name, provided the number or types of their parameters differ; this is called *overloading*.

A class may have a constructor and/or a destructor. The constructor may declare parameters; in that case, the new call will supply arguments, as shown here:

```
// MyClass:
Sub Constructor()
    self.itsString = "howdy"
End Sub
Sub Constructor(s as string)
    self.itsString = s
End Sub
Sub Constructor(c as MyClass)
    self.itsString = c.itsString
End Sub
// elsewhere:
dim c as new MyClass
dim c2 as new MyClass("hello")
dim c3 as new MyClass(c2)
c2.itsString = "bonjour"
msgbox c.itsString // howdy
msgbox c2.itsString // bonjour
msgbox c3.itsString // hello
```

This code also illustrates two features that have arrived since the publication of my book. Constructors can now be

named `Constructor` instead of having the same name as the class (convenient if you change a class's name). And the first three lines of the `elsewhere` code show declaration and initialization in the same line.

### Parameter Passing

Previously, a method's parameters could not be optional. You could work around this limitation in various ways—through overloading, for instance, or by passing a collection or dictionary object. Now a method can make parameters optional by declaring default values for them; the supplied arguments are paired with the declared parameters in left-to-right order:

```
Sub myMethod(s1 as string = "hey",
    s2 as string,
    s3 as string = "ho")
    msgbox s1 + s2 + s3
End Sub
// elsewhere in the same class:
self.myMethod("ha") // heyhaho
self.myMethod("ha", "hi")
// hahiho
```

Also, a `ParamArray` parameter mops up subsequent arguments into an array:

```
Sub myMethod2(paramArray args as variant)
    dim arg as string
    dim argStrings() as string
    for each arg in args
        argStrings.append arg
        // make string array for join()
    next
    msgbox join(argStrings)
End Sub
// elsewhere in the same class:
self.myMethod2("hey", 1, "ho")
// hey 1 ho
```

The preceding example takes advantage of the fact that a variant can be anything, even a scalar, and that it is coerced automatically on assignment. The `for..each` construct has also been added since the release of my book (as has the `join` function). Unfortunately, `for..each` works by assignment, not by aliasing, so you can't use it to alter scalar items of an array in place, as in Perl.

### Privacy, Getters, and Setters

In an earlier example, I said:

```
c.itsString = c.itsStringMaker()
```

What if you don't want just any old code to be able to access `MyClass`'s `itsString` property this way? In the past, you could mark a member of a class as *private*, which actually meant what most folks would call *protected*; code in the same class or its subclasses could access a private member.

This terminology is now rationalized; a member of a class can be *public*, *protected*, or *private* (where *private* now means that only the defining class can access it).

A member of a module can also have any of three privacy levels. A private member can be accessed only by code within the module. A public member and a global member differ in that the public member, though available everywhere, requires use of the module's name; this is much nicer than the rather messy global namespace that comprised all modules in earlier versions of REALbasic.

Now comes the cool part: getter and setter methods can give the *illusion* of accessing a property. For example, suppose `MyClass` has no `itsString` property, but rather has a protected property, `itsProtectedString`. Then I can define public `MyClass` methods as follows:

```
// MyClass:
Sub itsString(assigns s as string)
    self.itsProtectedString = s
End Sub
Function itsString() As string
    return self.itsProtectedString
End Function
```

The effect is that my earlier code accessing a `MyClass` property, `itsString`, still works. I can still say this, for example:

```
dim c1 as new MyClass("hey")
dim c2 as new MyClass("ho")
c1.itsString = c2.itsString
```

This syntax gives the caller the illusion of getting and setting a property when in fact it's calling methods. Of course, you could do much more in these methods than get and set a property.

### Custom Conversion Operators

Another new linguistic elegance is the ability to define what should happen when a value of one data type appears where a different data type is expected. You do this through `operator_convert` methods. A class can define a *from* converter and/or a *to* converter. You can implement multiple `operator_convert` methods for a class, describing what to do for different data types.

For example, suppose I define `MyClass` methods as follows:

```
Function operator_convert() As string
    return self.itsProtectedString
End Function
Sub operator_convert(s as string)
    self.itsProtectedString = s
End Sub
```

Now I can use a `MyClass` instance where a string is expected, and vice versa; the value will be coerced implicitly, according to the methods I've defined. So I could rewrite an earlier example thus:

```
dim c as new MyClass
dim c2 as new MyClass("hello")
dim c3 as new MyClass(c2)
c2 = "bonjour"
msgbox c // howdy
msgbox c2 // bonjour
msgbox c3 // hello
```

Actually, I can go even further, replacing the second line with this:

```
dim c2 as MyClass = "hello"
```

Refer back to my earlier `myMethod2` example; this will now work:

```
self.myMethod2(new MyClass("hey"),
new MyClass("ho"))
```

That's because the assignment from a variant to a string in `myMethod2` now knows what to do when the variant holds a `MyClass` instance.

### Custom Arithmetic and Comparison Operators

You can define arithmetic operators and comparison operators for classes. Each such operator has two possible definitions, depending on which side of the operator `self` appears on. So, for example, I might implement addition for two `MyClass` instances and for a `MyClass` and a string:

```
Function operator_add(s as string) As MyClass
    return new MyClass(self.itsProtectedString + s)
End Function
Function operator_add(c as MyClass) As MyClass
    return new MyClass(self.itsProtectedString +
        c.itsProtectedString)
End Function
Function operator_addRight(s as string) As MyClass
    return new MyClass(s + self.itsProtectedString)
End Function
// elsewhere:
    dim c1 as new MyClass("ho")
    dim c2 as new MyClass("ha")
    dim s as string = "hi"
    msgbox c1 + c2 // hoha
    msgbox c2 + c1 // haho
    msgbox s + c1 // hiho
    msgbox c1 + s // hohi
```

In the past, equality comparison of instances reported the identity of the instances (the two references point to the same thing). If you define comparison operators, that won't work, so a new operator, `is`, reports the identity of the instances:

```
Function operator_compare(c as MyClass) As integer
    return strcmp(self.itsProtectedString,
        c.itsProtectedString, 1)
End Function
// elsewhere:
    dim c1 as new MyClass("hi")
    dim c2 as new MyClass("hi")
    if c1 = c2 then msgbox "they are equal"
    if not (c1 is c2) then msgbox
        "they are different instances"
```

(The single-line `if..then` construct is new as well.)

### Class Extensions

You may define a method on a class from outside that class. The point of this feature is to add methods to a class belonging to REALbasic rather than to the programmer; this is similar to Objective-C's categories, and works even for scalar types. The method needs to be global (that is, defined in a module).

For example, here's a workaround for REALbasic's lack of an arithmetic assignment operator:

```
Sub inc(byref extends i as integer, j as integer = 1)
    i = i + j
End Sub
// elsewhere:
dim i as integer
msgbox str(i) // 0
i.inc
msgbox str(i) // 1
i.inc(10)
msgbox str(i) // 11
```

REALbasic itself takes advantage of this feature to reduce the prevalence of global functions. For example, you can now say `s.uppercase` instead of the inelegant `uppercase(s)`. Curiously, the migration away from global functions has not been thoroughly carried out; you must still use `str(sqrt(i))` to say `i.sqrt.str` is illegal. Of course, you can make it legal by implementing `sqrt` and `str` as class extensions.

### Other Miscellaneous Improvements

In the past, you could handle exceptions only at the method level. A method could end with an exception handler, which would catch exceptions raised within the method. If you wanted to apply exception handling to a smaller region of code, you had to

factor out that code into a separate method, which was often inconvenient or not even feasible. Now, there's a `try..catch..end` block structure (with an optional `finally` block), similar to the one in Java.

Return types are now covariant. This means that given a `MyClass` class and its `MySubclass` subclass, if you define `myMethod` somewhere as returning a `MyClass`, it's OK to override `myMethod` with a method defined as returning a `MySubclass`. (The lack of this feature is the keystone of a forceful critique of Java by Dragos Manolescu and Adrian Kunzle; see <http://micro-workflow.com/PDF/JavaIsNotAFrameworkLanguage.pdf>.)

Arrays in REALbasic have always suffered from an identity crisis: they are not mere scalars, but they aren't full-fledged objects either. Consequently, arrays have been subject to many special restrictions. Arrays are still neither fish nor fowl, but the latest version of REALbasic has largely lifted or rationalized these restrictions; for example, you can now assign an array to a variable or return it from a function, and you can pass multidimensional arrays as parameters.

This version of REALbasic removes a number of small annoyances. For example, you can now wrap long lines using a line-continuation character; the variable in a `for` loop doesn't have to be an integer; in a `select` block, case statements have a much more flexible syntax; and `dim` statements (local variable declarations) no longer have to precede all other statements in a method.

### Conclusions

The REALbasic language does have its failings. Some of these are minor inconveniences and inelegancies. I sometimes wish for REALPerl, REALJavaScript, or whatever, if only to have a unary increment operator, arithmetic assignment operators, and so forth. But such dissatisfaction occurs with any language.

The one major linguistic feature whose lack I feel keenly is class methods. You cannot, for example, call a `MyClass` method without first generating a `MyClass` instance. This makes it impossible to implement a Singleton properly—or rather you can implement it, but you can't enforce it, since no one can create an instance unless its constructor is public, in which case anyone can create one. An easy workaround involves instantiating

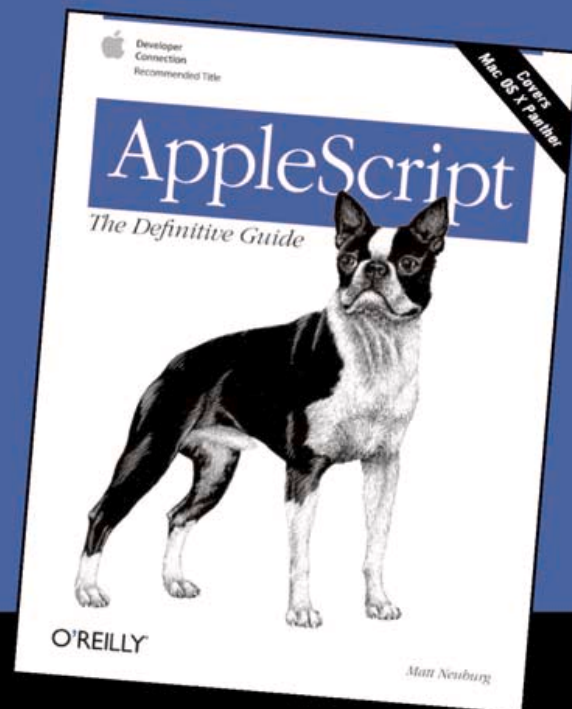
all Singletons at startup; you can then enforce the singularity by raising an exception in the constructor. But this isn't really a true Singleton.

The pertinent question, however, is whether REALbasic is adequate for serious programming, which means object-oriented programming, since REALbasic is an object-oriented language. The answer, on the whole, is clearly yes. REALbasic is a good cross-platform application framework. But even more crucial, it uses a true object-oriented language, in which you can fully implement most of the important design patterns, possessing some powerful elegances of expression. ☺

**Matt Neuburg** ([matt@tidbits.com](mailto:matt@tidbits.com)) is a freelance writer and programmer, and a TidBits contributing editor. He is a former editor of *MacTech* magazine and is the author of the O'Reilly *Definitive Guide* books on Frontier, REALbasic, and AppleScript.

“...a masterful guide to the language... it is certainly the best book on AppleScript I have seen.”

—TONY WILLIAMS  
SLASHDOT, JANUARY 2004



O'REILLY®