Chapter 1: A quick-and-dirty example

In which we develop a surprisingly simple canvas application

I really enjoy attending the annual REAL World conference, where I've talked about OOP for the last few years.

I always encourage folks at the conference who are having design issues to bring them to me to talk about.

In the course of discussing one such problem, I threw together a really simple example that will be a great way to kick things off here.

The problem I was brought was how to make a graphical print layout easier to write and maintain. And since writing graphical user interfaces is pretty much *the* example of using OOP, the way forward was really obvious, and throwing together a little illustration took literally about twenty minutes.

So, obviously, it's nothing fancy. But it does illustrate the overall structure you'll want to use with any kind of drawing program or similar Canvas project. Here is what the running program looks like:

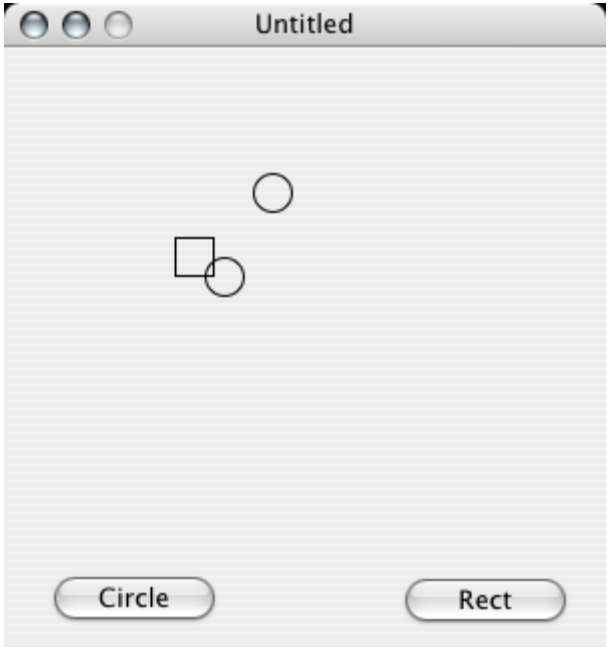
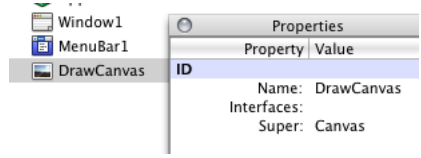


Figure 1: A quick and dirty drawing program example

There is a canvas in the window. When you click on the *Circle* button, a circle is added to the canvas. Similarly for the *Rect* button. You can add any number of circles and rectangles, and you can drag them around.

You can download the project from the Relevant Logic website, but you might like to walk through building it now instead.

1. Fire up REALbasic. Add a *Canvas* subclass called *DrawCanvas*. Open its code editor tab.



2. Add the following method:

```
Private Sub Add(d As DrawObject)
    Me.TheDrawObjects.Append d
```

3. Add the following private properties:

4. Add the following Event Handlers:

```
Function MouseDown(X As Integer, Y As Integer) As Boolean
  For Each d As DrawObject In Me.TheDrawObjects
    If d.IsWithin(X, Y) Then
      d.MouseDown x, y
      Me.TheClickedObject = d
      Return True
    End If
  Next
  Me.TheClickedObject = Nil
  Return True
```

```
Sub MouseDrag(X As Integer, Y As Integer)
  If Me.TheClickedObject <> Nil Then
    Me.TheClickedObject.MouseDrag X, Y
  End If
  Me.Refresh
```

```
Sub MouseUp(X As Integer, Y As Integer)
  Me.TheClickedObject = Nil
```

```
Sub Paint(g As Graphics)
For Each d As DrawObject In Me.TheDrawObjects
  d.Draw
Next
```

This is typical object oriented design: *DrawCanvas* is given responsibility for maintaining the state and making the decisions that are always the same when we interact with the canvas. The things that must be done differently depending on which circle or square we clicked on are handled by a separate object.

Note that those separate objects are all of the same *type*. Types are central to object oriented design, and we will be discussing types throughout this book.

By the way: you may wonder why I say "types" and not "classes". This is because classes are not the only — or even the most important — way to define a type in REALbasic. For the moment, consider "type" to more or less just mean "class" and pretend I'm waving my hands a lot whenever I talk about types. We'll get more precise shortly.

Moving on.

As we think about types, we think about the relationship between them. These relationships will be of two kinds:

- The public methods and properties of the type; and
- The other types that it employs, the methods the class calls on those types, and the public properties accessed from those types

We will refer to these as the *inward* and *outward* relationships, respectively.

Object oriented design pretty much comes down to:

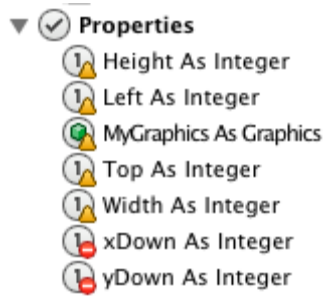
- Defining some types; and
- Allocating among the types responsibility for each of the parts of the state and behavior of the program

Our goal as we do this is to simplify the design of the individual types and the relationships between them.

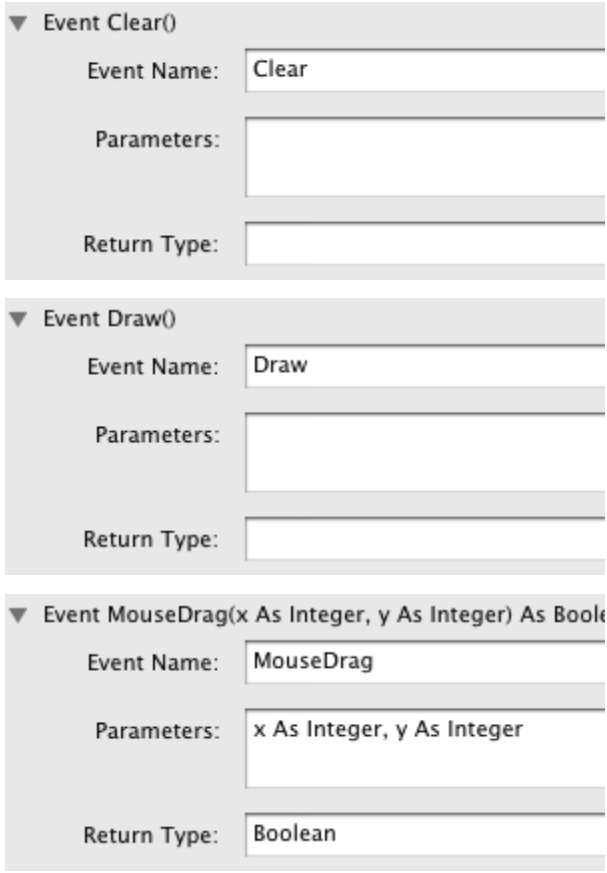
So bear this idea in mind throughout this book: good OOP has *the simplest possible types in the simplest possible relationships*.

Moving on.

1. Create a *DrawObject* class
2. Give it these properties:



3. Create the following Event Definitions:



4. Implement the methods that *DrawCanvas* expects:

```
Sub Constructor(c As DrawCanvas, x As Integer, y As Integer, w As Integer, h As Integer)
    Me.MyGraphics = c.Graphics
    Me.MyGraphics.ForeColor = FrameColor
    Me.Left = x
    Me.Top = y
    Me.Width = w
    Me.Height = h
```

```
Sub Draw()
    RaiseEvent Draw
```

```
Function IsWithin(x As Integer, y As Integer) As Boolean
    Return x >= Me.Left And _
        x <= Me.Left + Me.Width And _
        y >= Me.Top And _
        y <= Me.Top + Me.Height
```

```
Sub MouseDown(x As Integer, y As Integer)
    Me.xDown = x - Me.Left
    Me.yDown = y - Me.Top
```

```
Sub MouseDrag(X As Integer, Y As Integer)
    If Me.TheClickedObject <> Nil Then
        Me.TheClickedObject.MouseDrag X, Y
    End If
    Me.Refresh
```

DrawObject will be part of a class hierarchy: it will have subclasses that draw the circle and the square. Again, we design this class using the same principles we used with *DrawCanvas*: we put the state and behavior here that will always be the same, no matter what we're drawing, and we delegate the parts of the job that are different to the subclasses. We do that using *Events*, which are a way for a superclass to call a method on its subclasses. We will discuss Events in detail shortly.

If you've done much REALbasic programming, most of the code here should be straightforward, but I want to make sure the following is clear:

- Our class has a constructor (simply a method named *Constructor*), which is always called as an object is created; and
- *RaiseEvent* is how a class can call on code in the event handler of its subclass.

Classes and instances

The other thing we need to make clear is this business of classes and instances. There are a few things going on all together that are usually confusing at first.

In broad terms, a class is a blueprint, from which we make objects. The objects made from a given class are called *instances* of that class.

An object is a collection of the properties defined in a class, along with a tag that tells the program where to find the code for that class. When you call a method on an object, the program consults the list of methods to work out which actual code to execute. The part of the program that's calling a method asks an object to do something, but that can be any of a range of things, depending on which object is being asked.

Because this isn't confusing enough, a class also has an *inheritance chain*. Each class has a superclass (if you don't define one in the class's properties, the superclass is assumed to be *Object*). That superclass can itself have a superclass, and so on, up to object. So a given class has a sequence of superclasses which is its inheritance chain.

As well as the inheritance chain of a *class*, we can also speak of an inheritance chain of an *object*, which is just the inheritance chain of that object's class.

If you look at things the other way around, a given class can have many *subclasses*, forming a *tree*, with *Object* at the root. Like this:

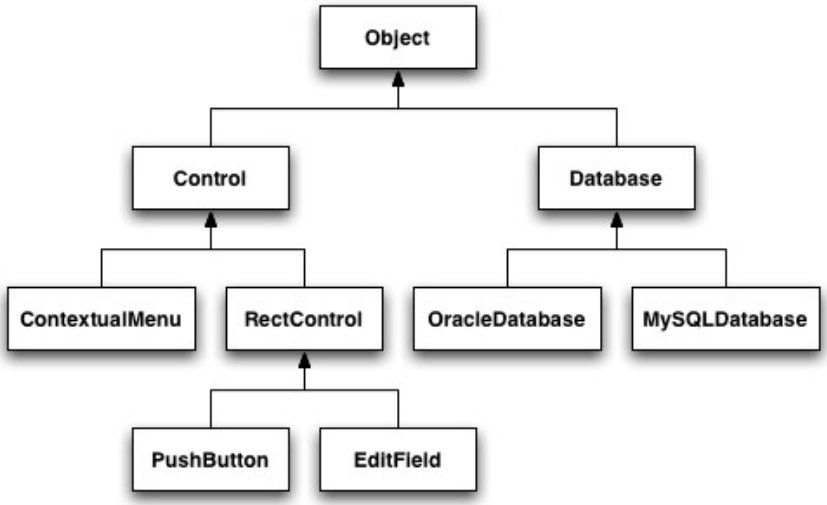


Illustration 1: A REALbasic class hierarchy

This tree is intended to be similar to other tree structures we use to organize all kinds of things. In biology, we put organisms into a tree with clades, kingdoms, species and so on. Like this:

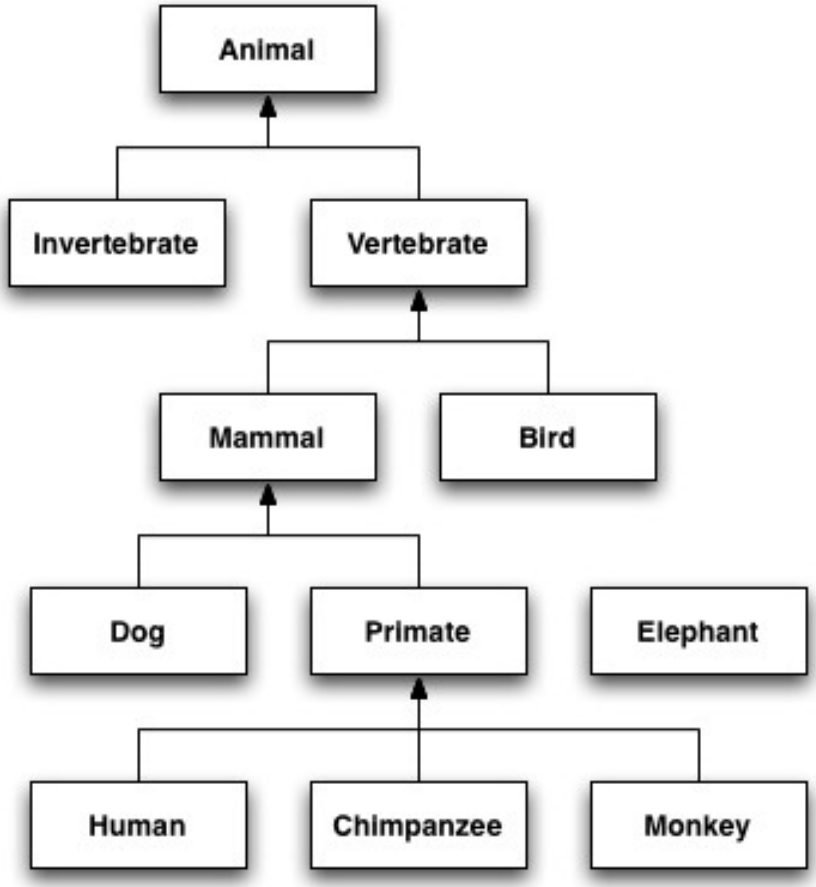


Illustration 2: A biological class hierarchy

If we mix up our terminology a bit here, humans are a subclass of primates, which are a subclass of mammals and so on. In biology as in OOP, much of what we know about a class comes from what we know about its superclass — humans have all the properties of primates, along with other properties specific to humans. Similarly, a PushButton in REALbasic has all the properties of a RectControl (*Left*, *Top*, *Width*, *Height*, and so on), plus some others of its own (such as *Caption*).

So a class has all the properties defined directly on the class itself, and those defined on all of its superclasses.

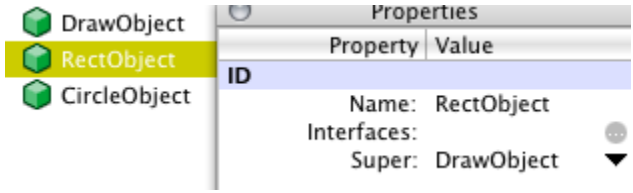
The behavior of instances of a class is more complicated. It will have all the methods defined on itself and all of its superclasses, but the relationships between all of the methods and event handlers might be such that some of that code can never be executed, or it might be executed in a different way.

One of the harder things to wrap your mind around when learning OOP (and sometimes even if you're a seasoned OOP guru) is writing the code that's in the middle of a class hierarchy somewhere. Because that code will have different subclasses that it interacts with, depending on the inheritance chain of the object it is being called on.

The same thinking we just mentioned — the simplest relationships between the simplest types — will also be our guide in designing the types that go into our class hierarchies.

Let's finish our little drawing program, so we can see how the whole picture fits together.

1. Create *RectObject*, a subclass of *DrawObject*:



2. Open the code editor for *RectObject*, and give it the following Event Handlers:

```
Sub Clear()
    Me.MyGraphics.ClearRect Me.Left, _
        Me.Top, _
        Me.Width, _
        Me.Height
```

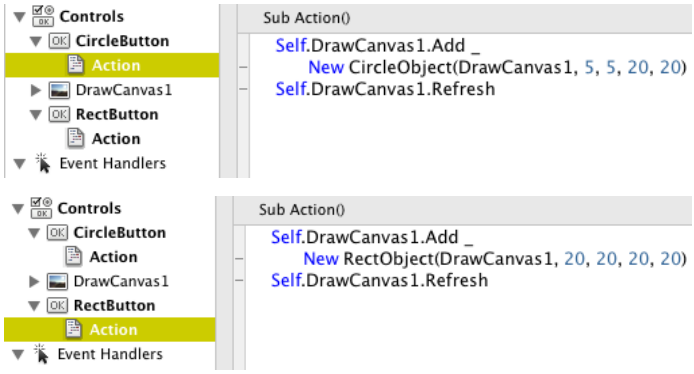
```
Sub Draw()
    Me.MyGraphics.DrawRect Me.Left, _
        Me.Top, _
        Me.Width, _
        Me.Height
```

3. Create *CircleObject*, another subclass of *DrawObject*. The code for the Event Handlers for *CircleObject* is left as an exercise for the reader[1].

Note that there is no code in the *MouseDrag* Event Handler.

[1]Remember: you can download the code from this book from my website, http://relevantlogic.com/oopbook

4. Open *Window1*, and lay it out with a *DrawCanvas* and *RectButton* and *CircleButton* pushbuttons. Open its code editor, and give it the following Event Handlers:



That's it. Not a whole lot of code for a nice little example. Look through the code and try stepping through the code as you drag an object around and so on.