3

Classes and Objects

The Class Hierarchy

A former colleague used to say that he had "developed this new technology" whenever somebody in his department wrote a few lines of VBScript for the website. He is not the first person to use overly fanciful language to describe programming exploits. It happens all the time. Object-oriented programming is a particularly jargon-rich field of study, laden with ill-formed metaphors and polysyllabic titles. There was a time when object-oriented programming (OOP) advocates would say that OOP was easier to learn than more traditional styles of programming. I don't know if that is true; I suppose the fact that I have deferred the discussion of the object-oriented features until the third chapter of this book is some indication of how I feel about this subject.

One of the biggest challenges facing a novice programmer is the reality that knowing how to program and knowing how to program well are two different things. Learning a programming language is cut and dried (albeit a bit tedious at times). Learn the syntax and you've learned the language. It's very much a science. Doing it well, however, is much more like an art—something that is learned over time and something that you get a "feel for" that is often difficult to explain simply and clearly. In this chapter, I hope to provide an overview of object-oriented concepts and how they are implemented in REALbasic, and give some insight into strategies for executing your object-oriented programming effectively.

Cities are full of buildings, and each building has a particular street address, which can tell you how to find the building if you are looking for it. Computers have addresses, too, but instead of identifying buildings, these addresses point to locations in memory. Variables, properties, functions, and subroutines all reside at a particular address, too, when they are being used by your program.

When you declare a variable, you assign a type to the variable. To date, most of the variables in the examples have been one of REALbasic's intrinsic data types. When a

variable is used in this way—pointing to an intrinsic data type—it's said to be a scalar variable. The variable represents the value that's stored at a particular location in memory.

Variables can also be used to represent objects. When they do, they aren't scalar variables anymore because an object doesn't represent a value in the way that an intrinsic data type does. The variable is said to be a reference to the object; it points to the location in memory where this object resides. An object is a collection of different things and can be made up of properties and various methods. In this respect it is like a module, but it differs from a module in some very important ways.

Suppose we have an object with two different properties, one called FirstName and the other called SecondName, both of which are Strings. Let's also say that we have a variable called Person that points to this object. Referring to the object itself doesn't refer to any particular value, because the object has multiple properties. This means that to access a particular property of an object, you need to refer to the object itself, plus the property. You do this using the familiar dot notation, like so:

```
Person.FirstName
```

This should look familiar because it's the way you refer to protected properties and methods of modules. The biggest difference is that you always have to refer to an object's members this way, even if the property is public.

Now knowing that an object is a collection of values is only a small part of the story. What makes objects powerful is that they are members of classes. To create an object, you must first create a class.

To explain what classes are conceptually, I'll return to the building and street address analogy. If you pull up to any address, you'll find a building. If you were to come visit me, you could drive until you reached my address; pull in the driveway and you'd be at my house. My house is a building, but it's a different kind of building than you might find down the street a bit where there is a grocery store and a gas station. Furthermore, my house is just one example of a particular type of house. Houses come in all styles—some are ranches and others are Cape Cod or Tudor styles.

Whether you are talking about houses or grocery stores or gas stations, every building has some things in common—a roof and some walls, for example. I live in a single-family residential neighborhood, so every building on my street is a house. The fact that it is a house means that in addition to a roof and some walls (which all buildings have), it also has at least one bedroom, a bathroom, and a kitchen, because buildings that happen to be houses all have those features.

If I were to try to systemize my understanding of buildings, I might start by classifying buildings into the different types of buildings and showing how they are related. A good way to do this is by organizing buildings into a conceptual hierarchy, like the one outlined in Figure 3.1.

Every node on this tree represents a type of building or, to use object-oriented terminology, a class of building. At the top of the hierarchy is the base class, and it has two subclasses—residential and commercial. You can say that the building class is the superclass of the residential and commercial classes. In addition to the super/subclass terminology, you also hear people refer informally to them as parent classes and child classes.

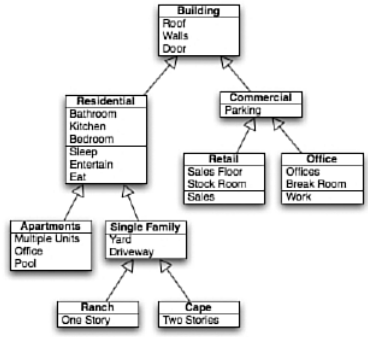

Figure 3.1 Different styles of houses can be organized in a hierarchy.

One way that we make these classifications is based on physical attributes of the buildings themselves—things such as walls and doors. At the same time, we also consider the kind of activity that takes place in them.

This isn't the only way to classify buildings. You might come up with a different hierarchy that's equally valid. For example, you might decide that the most important differentiating factor isn't whether a building is residential or commercial, but how many stories the building has. You might start off with building, and then branch to one-story buildings and multistory buildings, and go on from there.

Even the buildings themselves can be subdivided into smaller units. A house doesn't just sit empty. People live in them and do things. The house is organized into rooms and these rooms are designed for certain activities. In one room you cook, in another you watch TV, and so on. Some rooms have storage areas, such as closets and cupboards, and others have tools, such as stoves and refrigerators and washing machines.

The hierarchy shows classes of buildings, not particular buildings themselves. A description of a house is not any particular house; it's a description of what houses that are members of that class have in common. One way to think of classes is to think of them as a blueprint, or a design, for a building. You can build many houses using the same blueprint. If a class is analogous to a blueprint, an object is analogous to a particular house.

After you've driven to my house and you're standing in the driveway looking at it, you are looking at one particular instance of a house. There are other houses in the neighborhood that were built from the same plan, but they are filled with different people, doing different things.

One thing you may have noticed when I classified buildings and organized them into a hierarchy is that I didn't list all the features of each class of building at each level. At the top level, I said buildings have roofs, walls, and doors. At the second level of the hierarchy, I didn't repeat that they have roofs, walls, and doors because I wrote down only what was new and unique about the next group. The assumption is that the subclass has all the attributes of the superclass, plus some additional attributes that differentiate it from the superclass. To put it another way, the subclass inherits the features of the superclass.

Inheritance is a big deal, and it's just about the coolest part of object-oriented programming. In practical terms, this means that when you write your program, you organize your code into classes. The advantage of this kind of inheritance from a programmer's perspective is that it keeps you from having to rewrite as much code. The subclass inherits all the functionality of the superclass, so that means you don't have to write new methods in the subclass that replicate the basic functionality of the parent.

And this leads to the challenging part. It's up to you to decide how to organize your classes. You can do it any way you like, but you need to do it in a way that makes sense and in a way that maximizes code reuse (from inheritance). Knowing how to write the code that creates a class is only half the battle. Knowing what goes into the class is the hard part. There is no single solution; the answer is really dependent on the kind of program you are going to write. You have to figure out what makes the most sense for your application.

Again, I'll return to the house analogy. Inside my house, the downstairs is divided into six rooms: a living room, a dining room, a kitchen, a bedroom, a bathroom, and a laundry room. In the kitchen are cupboards. Some are full of food and others are filled with plates and dishes. There is a refrigerator, stove, and so on. Likewise, in the laundry room you'll find laundry detergent, and then there is the usual stuff that goes into a living and a dining room.

This makes sense—the reason I put the laundry detergent in the laundry room with the washing machine is because it's convenient and makes sense. Likewise, I put my clothes in my closet so that I can find them in the morning and get dressed.

That's exactly how object-oriented programming works, too. A class in object-oriented programming groups together information and instructions for what to do with the information. It's a place to store information of a particular kind, and a set of instructions for what to do with it. When you are creating classes, you have to decide what kind of information makes sense to be associated with what kind of tasks.

Although objects are capable of having properties, constants, and methods, they are not required to have them. A chair, for example, is an object. By definition, a chair doesn't do anything. It can only have things done to it. It can be sat upon. It can be held in place by gravity. It can be broken and it can be painted red. But it can't scoot itself up to the table, or teeter precariously on two legs, unless a child is in the chair to do the teetering. An object from the programming perspective can be dull and boring like a chair. It can be a thing, a set of values, or, as they are called in REALbasic and other languages, it can have properties.

An object can be nothing more than a set of properties. In this respect, an object is very similar to a struct or User Defined Type familiar to Visual Basic programmers. It can also have methods, constants, events, and menu handlers, all of which will be covered in due time.

There are some basic concepts to discuss first, and the best way to discuss them is to illustrate them by creating a class or two and looking at what we are doing. In particular, I want to discuss some important OOP concepts: inheritance, encapsulation, and polymorphism. Much like calling a new script a new technology, these words create a certain

mysterious aura that make everything sound much more complicated than it is. Hopefully, after you've read this chapter you'll have enough mastery of REALbasic's flavor of object-oriented programming to use it with confidence.

Creating a New Class

A new class is created just like a new module. Open up a new project in REALbasic and click the **Add Class** button and a new class will be added, as shown in Figure 3.2. However, **App**, **Window1**, and **Menubar1** are classes, too. They are provided by REALbasic automatically, but everything you can do with this class we are about to create, you can do with them as well. These three classes will be covered in much more detail in the next two chapters.

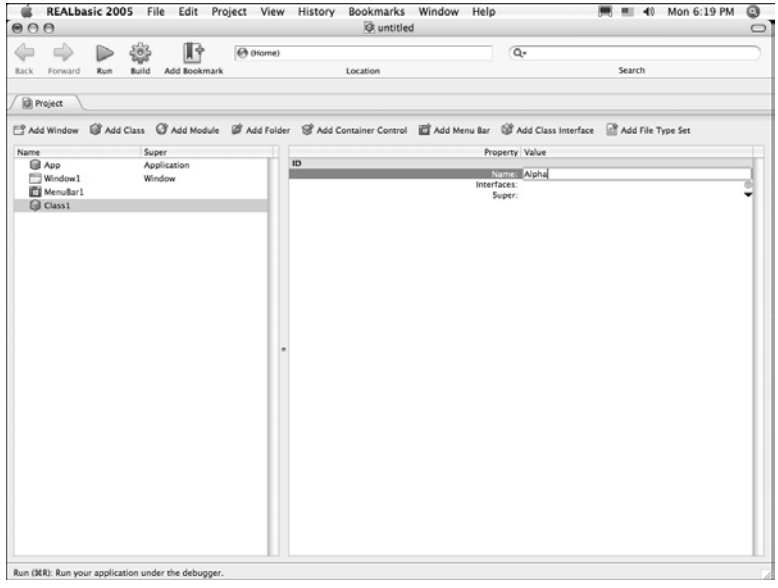

Figure 3.2 Creating a new class in the REALbasic IDE.

After you create this new class, select it, and change the name to **`Alpha`** in the **Properties pane** on the right side of the window. Double-click the **Alpha** class in the Project Editor to bring up the Alpha tab, as shown in Figure 3.3.

When you've done this, you'll see that you have additional buttons on the Code Editor toolbar. In addition to the familiar **Add Method**, **Add Property**, and **Add Constant**, you'll see **Add Menu Handler** and **Add Event Definition**. **Menu Handlers** and **Event Definitions** are special kinds of methods that I'll cover later.

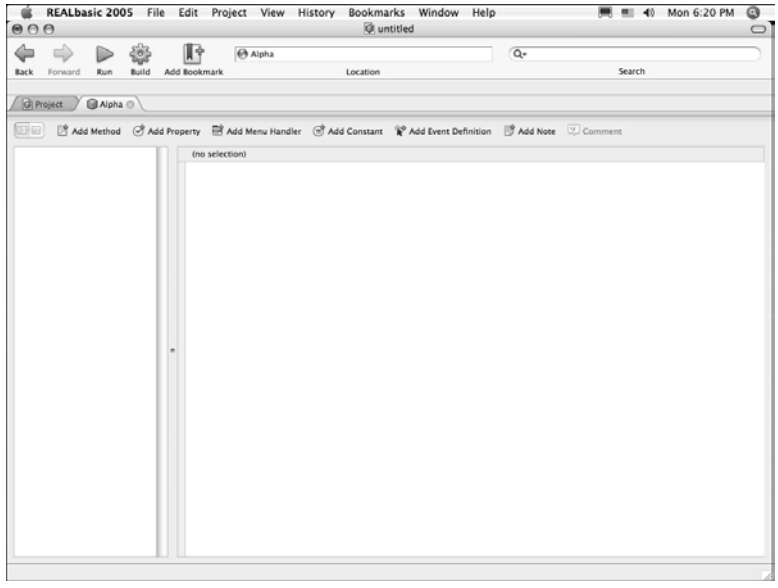

Figure 3.3 Editing code for the Alpha class.

The **Alpha** class won't do anything really useful. The purpose is to implement some methods and properties in such a way that you can see the different way that classes implement and use them. The first method to add is the `getName` method:

```
Function getName() as String
  Return "Alpha"
End Function
```

As you can see, this method returns the string `"Alpha"` when it is called.

Declaration and Instantiation

When you build a house, you start with the blueprints, but it's not a house until you build the walls, roof, and so on, and eventually people move in. I said earlier that a class is like a blueprint. Because the `Alpha` class is a blueprint, it needs to be built in order to be used. This is called *instantiation,* or creating an instance of the `Alpha` class. An instance of a class is called an *object*.

This is one area where classes differ from modules. You do not have to instantiate a module to have access to the methods and properties of the module. Instantiating an object is a two-part process. First, the object must be declared, just like variables are declared. The only difference is that the type of this variable is going to be the class name rather than one of REALbasic's intrinsic data types. The second step is the actual instantiation, the part where the "house is built." When you build a house, you need

boards and nails. When you declare an object, you are setting aside space for the constants, properties, and methods of the class, and when you instantiate an object, you are filling up the space set aside with the constants, properties, and methods of the class. Again, it's like building a house—first the rooms are made, and then people move in to live in them.

```
Dim a as Alpha
Dim s as String
a = New Alpha()
s = a.getName() // s equals "Alpha"
```

The variable is declared with a `Dim` statement, and it is instantiated using the `New` operator. After the class is instantiated and assigned to the variable `a`, you can use dot notation to access the members of the class, much like you can access the members of a module. In this example, the method `getName()` is called and the value returned is assigned to the string `s`.

Always eager to provide help to those like me who have an aversion to extra typing, the engineers at REALbasic have provided a handy shortcut that combines these two steps into one (this works with data types, too).

```
Dim a as New Alpha()
Dim s as String
s = a.getName() //s equals "Alpha"
```

Constructors and Destructors

When operators were introduced, I said that they were basically functions. They are given input in the form of operands; the operator causes a calculation to be performed, and the results are returned. Because New is an operator, that must mean that it's a function, too. As you can see from this example, the value returned by the `New` operator is an instance of `Alpha`, which is assigned to the variable `a`. The reason I followed `Alpha` with parentheses when instantiating `a` is so that it will be more clear that it is acting as a function. You are, in fact, invoking a function when you use the `New` operator. Your ability to refer to the class name alone, without explicitly typing out the method, is a matter of convenience.

REALbasic (and other object-oriented languages) gives this method a unique name: `Constructor`. It is the `Constructor` method that is being called when you use the `New` operator. In many cases, the default `Constructor` is all you need—all it does is instantiate the object, setting aside space for properties and things like that, but not setting their value. You can add to this by implementing your own `Constructor`.

Right now, our `Alpha` class returns the string `"Alpha"` when we call the function `getName()`. Suppose that we want to be able to decide what value this function will return when we instantiate the object. The way to do this is to create a property, which

will hold the value that the `getName()` function will return, and we will add a new `Constructor` method that will allow us to pass the string we want returned when we first instantiate the object.

The first step is to create the property, like you did when creating a module. Set the access scope of this property to Private and give it the name "Name", and a data type of string.

Next, add the method called "`Constructor`" (`Constructors` are always `public`). Do not specify a return value. Implement it as a subroutine (it knows automatically to return an instance of the class it's a member of).

```
Sub Constructor(aName as String)
  me.Name = aName
End
```

Finally, we need to update our `getName()` method.

```
Function getName() as String
   return me.Name
End
```

You have now implemented a new `Constructor` that takes a string as a parameter and assigns the value of that string to the property `Name`. Now, if you try to instantiate this class like you did before, you'll get an error. For you to instantiate it, you need to pass a string to the `Constructor`.

```
Dim a as Alpha
Dim s as String
a = New Alpha("Blue")
s = a.getName() // s = "Blue"
```

New in REALbasic 2005, you can set the values of properties implicitly when the class is instantiated, rather than explicitly as I did in this example. This is done by filling in the values in the Code Editor as shown in Figure 3.4.

In earlier versions of REALbasic, the `Constructor` method was named after the name of the class itself. So in this case, you can name the constructor "Alpha" and it would be called when the class is instantiated.

There is also a `Destructor` method that's called when the object is being purged from memory. It can be declared by calling it `Destructor` or by using the object name preceded by the tilde (~) character. For the **Alpha** class, you would name it "~Alpha". The default is to use the terms `Constructor` and **Destructor**, and I would recommend sticking with that—I only share the other method in case you encounter it in preexisting code.

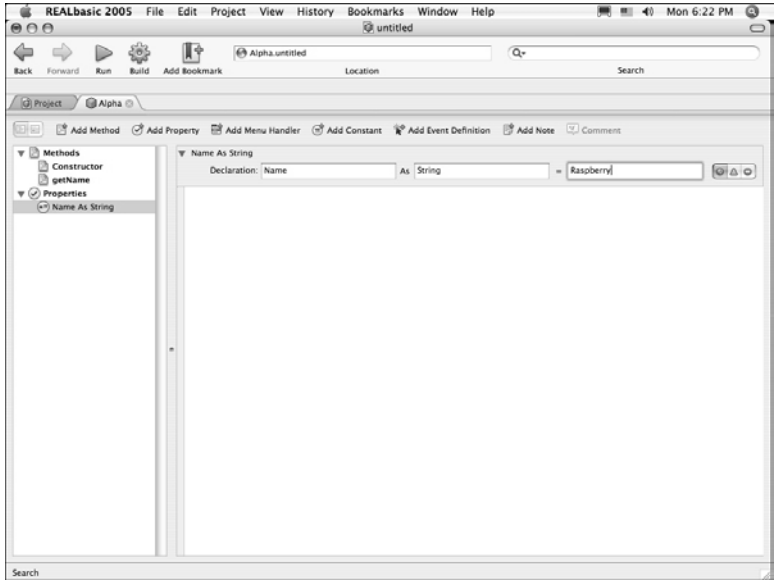

Figure 3.4 Set values for class properties in the Code Editor.

Garbage Collection

One thing you may have noticed is that I haven't had to say anything about how to manage the computer's memory in REALbasic. In programming languages such as C, much of the work you do centers on allocating memory for data and making sure that memory is freed up when you are done with it. In REALbasic, you don't have to worry about this very much because REALbasic uses a process called *garbage collection*, which means that it automatically purges objects from memory when appropriate. The operative phrase here is "when appropriate."

Garbage collection uses a process called *reference counting* to keep track of objects. Variables that refer to objects don't behave like scalar variables do. Although you may have two scalar variables whose value is "5", it does not mean that these two scalar values are references to the same location in memory. Objects, on the other hand, can easily have more than one variable serve as a reference to their location.

Reference counting refers to keeping track of how many variables (local variables and properties) reference a particular object. As long as at least one reference to an object exists, the object stays resident in memory. After the last reference goes away, the object is dumped. This is when the **Destructor** method is called.

In most cases you won't have to worry about memory, but there are some situations where references inadvertently do not go away. This can sometimes result in a situation where new objects are being instantiated while old objects are still hanging around in memory, which means that your program will consume larger and larger amounts of memory as the application runs. This is called a *memory leak*, and memory leaks are bad.

The trouble usually starts when you have two objects that each refer to each other. Because object **A** has a reference to object **B** and object **B** has a reference to object **A**, neither object is ever going to be garbage collected. The way around this is to implement a method for object **A** that explicitly sets the reference to object **B** to `Nil`. Do the same thing with object **B**. Now when object **A** is destroyed, the reference to object **B** is safely removed (or, more accurately, the reference count to object **B** is decremented).

Inheritance

Now you have a class that implements one method, and you know how to instantiate that class. One of the most powerful features of object-oriented programming is that you organize your classes into conceptual hierarchies, just like you can organize buildings into a conceptual hierarchy. One class can subclass another and when it does this, it inherits the members of the class from which it does the inheriting.

Although you are no doubt familiar with the idea of inheritance, object-oriented inheritance works a little differently than genetic inheritance does. First of all, people have two parents. I'm a mix of my dad and my mom; 50% of my gene pool comes from mom and 50% from dad. (I've often said that I have the distinct pleasure of having inherited all of my parents' worst qualities. In a remarkable reversal of fortune, my daughter has managed to inherit only the finest traits from her parents, mostly from her mom). REALbasic takes a simpler view of inheritance. There's only one parent, which is the superclass. In REALbasic, at the top of the family tree is the root class; **Object** and all other classes are subclasses of it.

In the object-oriented world, the child is called a subclass and the parent is referred to as the superclass. When you create a subclass in REALbasic, it automatically inherits all the methods and properties of the parent class. There is absolutely no difference other than their name.

Creating a subclass that doesn't do anything differently from the superclass is more or less a pointless exercise. The reason for creating a subclass is that you want to reuse the code you've written for the superclass, and then either add new capabilities or change the way certain things are done.

A child inherits all the features of the parent, but it can also add some new ones. In practice, this means that the subclass has all the methods, constants, and properties of the parent class (with a few exceptions), but it can also implement its own methods, constants, and properties. In this respect, at least, object-oriented inheritance isn't all that much different from the way parent-child relationships work with humans. That's why I always had to set the time on the VCR when I was growing up. My parents didn't know

how to do it, but I did. In addition to the methods passed along by the parent, the child can have its own set of methods.

To create a subclass of **Alpha** in REALbasic, create a class as if it is a new class. Let's name it **Beta**, which can be done by typing the name into the **Properties pane**. There are two other properties listed in the pane that we have not discussed yet. After `Name` comes `Interfaces`, which I cover later, followed by `Super`. Setting the value of `Super` is how you designate this object's superclass, as shown in Figure 3.5. Type in **`Alpha`**, and you now have an official subclass of **Alpha**. It is identical in every way, except that it's called **Beta** instead of **Alpha**.

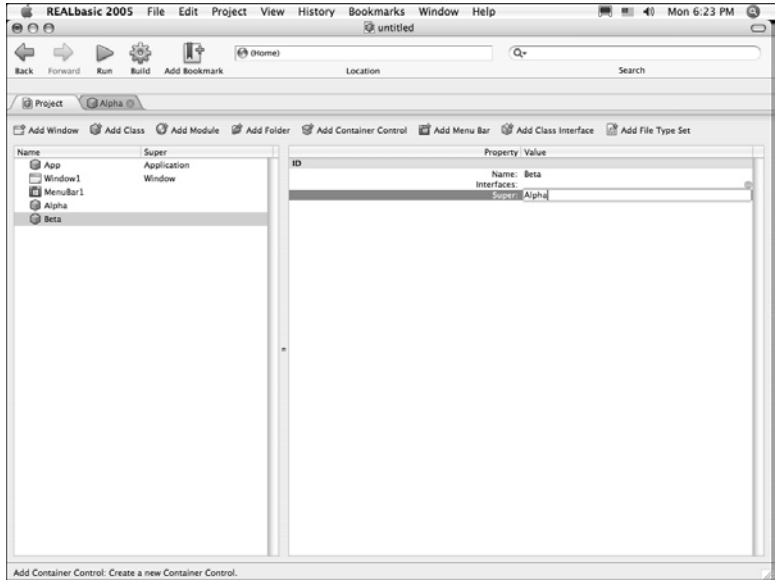

Figure 3.5 Define an object's superclass in the Properties pane.

```
Dim a as Alpha
Dim b as Alpha
a = New Alpha("Woohoo!")
b = New Beta("Woohoo!")
```

At this point, the only difference between `a` and `b` is their type (or class).

Object Operators

Now it's time to introduce some more operators that are used with classes. I've already introduced `New`. Two others are `Isa` and `Is`.

`Isa` is used to determine whether an object belongs to a class:

```
Dim t,u as Boolean
t = (a Isa Alpha) // t is True
u = (b Isa Alpha) // u is True
t = (a Isa Beta) // t is False
u = (b Isa Beta) // u is True
```

This is an example of the most basic feature of inheritance. A subclass is a member of the superclass, but the superclass is not a member of the subclass.

The related `Is` operator tests to see if one object is the same object as another one.

```
Dim t as Boolean
t = (a Is b) // t is False
```

Note that even though `a` and `b` both have the same values, they are different objects because they were both instantiated independently. Here's an example of when this test would return `True`:

```
Dim a,b as Alpha
Dim t as Boolean
a = new Alpha("Woohoo!")
b = a
t = (a is b) // t is True
```

Because `b` is a reference to `a`, it is considered the same object. In other words, both `a` and `b` point to the same location in memory. They do not simply share an equivalent value; they are, in fact, the very same object.

Adding and Overriding Methods

So far, I've created **Beta**, a subclass of **Alpha**, but that is all. Except for the type, there's really no difference between them. If you wanted to, you could add new methods to **Beta** that would be available only to **Beta** and not **Alpha**. This is easy enough to do—just add methods to **Beta** as you would to any class or module.

A more interesting thing you may consider doing is overriding an existing method.

Again, there is a shade of resemblance between object-oriented inheritance and human inheritance. Once in a while (as children are wont to do), the child decides to do something that the parent already does, but in a different way. For example, my father and I both go to church, but he goes to a Baptist church and I go to a Catholic one. In object-oriented circles, this is called overriding.

The way to override one of **Alpha's** methods in **Beta** is to implement a method with the exact (and I mean *exact*) signature as a method that is found in **Alpha**. Because **Alpha** implements only one method, `getName()` (well, two if you count the `Constructor`, which can also be overridden), we'll override `getName()` in **Beta**.

```
Function getName() as String
  Return "Beta says: " + me.Name
End Function
```

Now let's see how this works in practice.

```
Dim a as Alpha
Dim b as Beta
Dim s,t as String
a = New Alpha("Golly!")
b = New Beta("Golly!")
s = a.getName() // s = "Golly!"
t = b.getName() // b = "Beta says: Golly!"
```

Calling the Overridden Method

Normally, when you override a method, you want the new method to replace the overridden method, but there are also times when you want the new method to merely add a few steps to the parent method. You can do this by calling "`super`" on the method.

When deciding which method to call, REALbasic starts at the lowest level of the class hierarchy and looks to see whether the method is implemented there. If it is not, REALbasic checks the parent class, and so on, until it finds an implementation of the method. Because of this approach, as soon as it finds an implementation of the method, it stops looking and never touches any implementation further up the hierarchy.

Overloading

Overloading and overriding are two words that look alike, sound alike, and refer to similar things with markedly different behavior. That's a recipe for confusion if ever there was one. When you are creating classes and subclassing them, you will be making use of both overloading and overriding.

Overloading is associated with the idea of Polymorphism (who comes up with these words?). All that Polymorphism means is that you can have multiple versions of the same method that are distinguished according to the parameters that are passed to them.

Note that REALbasic does not pay any attention to the return value offered by a method when overloading methods. This means you can't do something like this:

```
Function myFunction(aString as String) as Integer
Function myFunction(aString as String) as String
```

and expect to be able to do this:

```
Dim i as Integer
Dim s as String
i = myFunction("Hello")
s = myFunction("Hello")
```

The only thing that counts are the arguments.

Now, when I instantiate the **Alpha** or **Beta** classes, I pass a string to the `Constructor`, and this is the value that the `getName()` method returns. Let's say for the moment that there are times when I want `getName()` to return a different value. One way to do it is to change the value of the property, but because that property is `Private`, it requires some extra steps, so we'll hold off on that approach for now.

Another way to do it would be to overload the `getName()` method, which means implementing another version of `getName()` with a different signature. Remember, a method signature refers to the collection of parameters that it takes. Because the original `getName()` doesn't take any parameters, we will implement a new version of `getName()` that takes a string as a parameter. For this example, implement the new method in **Beta**.

```
Function getName(aNewName as String) as String
   Return aNewName
End Function
```

Now let's see this overloaded method in practice.

```
Dim a as Alpha
Dim b as Beta
Dim s,t, u as String
a = New Alpha("Good morning.")
b = New Beta("Good night.")
s = b.getName() // s = "Good night."
t = b.getName("Hello.") // t = "Hello."
u = a.getName("Goodbye.") // Error!
// Won't compile because Alpha doesn't implement this method.
```

As you can see, REALbasic knows which version of `getName()` to call for the **Beta** class based solely on the parameters passed. In the first example, because nothing is passed, it returns the value of the property `Name`. When a string is passed, it returns the value of the string. Out of curiosity, I tried to pass a string to the **Alpha** implementation of `getName()`, and the compiler fell into all kinds of histrionics because **Alpha** doesn't implement a method called `getName()` that expects a string as a parameter. Remember, **Beta** is a subclass of **Alpha**. **Alpha's** methods are available to **Beta**, but **Beta's** aren't available to **Alpha**.

Casting

I alluded to the fact that a subclass is a member of the parent class, but the parent class is not a member of the subclass. This makes perfect sense, but it can at times make for some confusion, especially when dealing with overridden and overloaded methods.

I want to return to the **Alpha** and **Beta** classes from the previous examples and add a method to each one; this will illustrate the potential confusion (at least, it was confusing to me).

Alpha will add the following:

```
Sub TestMethod(a as Alpha)
```

Beta will add the following:

```
Sub TestMethod(b as Beta)
```

First, take a look at the following example:

```
Dim a as Alpha
Dim b as Beta
Dim s as String
a = New Beta("What's this?")
```

What is `a`? Is it an **Alpha** or a **Beta**? One way to tell would be to invoke the `getName()` method and see what is returned. If you recall, I overloaded the `getName()` method so that it accepted a string as a parameter.

```
s = a.getName("My String") // Error!
```

If you try to call the **Beta** method you'll get an error. That's because as far as REALbasic is concerned, `a` represents an **Alpha**. However, because you instantiated it as a **Beta**, you do have access to **Beta's** methods. You can do this by casting the `a` as a **Beta**.

```
s = Beta(a).getName("My String") // s equals "My String"
```

Reference the class to which you are casting the object, followed by the object surrounded in parentheses, and now you will get access to the methods of **Beta**.

Note that the following will not work:

```
Dim a as Alpha
Dim b as Beta
Dim s as String
a = New Alpha("What's this?")
s = Beta(a).getName("A String") // Error
```

The reason is that `a` is an **Alpha**, not a **Beta**. Recall that the superclass is not a member of the subclass.

```
Dim a as Alpha
Dim bool as Boolean
a = New Alpha("What's this?")
bool = (a Isa Beta) // bool is False
```

Oddities

When you subclass a class that has overloaded methods, if the subclass overloads methods, the superclass doesn't know about them. So if you instantiate the subclass but assign

it to a variable as the superclass, the overloaded method can't be called from it (this contrasts with how overridden methods are handled, because you can call an overridden method in the same way and get the subclass implementation of it).

I'll provide an example of one way that really confused me for a few days until I realized what I was doing wrong.

An overloaded method is one that takes different arguments (but has the same name). An overridden method has the same signature (takes the same arguments), but is implemented differently. This applies to inheritance between a class and a subclass. An overloaded method is overloaded in the same class, and overridden method is overridden in a subclass.

Start with two classes, **Parent** and **Child**. Set the parent of the **Child** class to **Parent**. For both classes, implement a method called `WhoAmI` and one called `GetName()` as follows:

Parent:

```
Sub WhoAmI(aParent as Parent)
  MsgBox "Parent.WhoAmI(aParent as Parent) /" + _  "Parameter:
➥" + aParent.getName()
End Sub
Function getName() as String
  Return "Parent"
End Function
```

Child:

```
Sub WhoAmI(aChild as Child)
  MsgBox "Child.WhoAmI(aChild as Child) /" + _ "Parameter: " + aChild.getName()
End Sub
Function getName() as String
  Return "Child"
End Function
```

The `MsgBox` subroutine will cause a small **Window** to appear, displaying the string passed to it.

Next, take a look at the following code. The text that would appear in the `MsgBox` is listed in a comment next to the method.

```
  Dim p as Parent
  Dim c as Child
  Dim o as Parent

  p = New Parent
  c = New Child
  o = New Child

  p.WhoAmI p // "Parent.WhoAmI : Parent.getName"
  p.WhoAmI c // "Parent.WhoAmI : Child.getName"

  c.WhoAmI p // "Parent.WhoAmI : Parent.getName"
  c.WhoAmI c // "Child.WhoAmI : Child.getName"
```

```
o.WhoAmI p //  "Parent.WhoAmI : Parent.getName"
o.WhoAmI c // "Parent.WhoAmI : Child.getName"
o.WhoAmI o // "Parent.WhoAmI : Child.getName"

Child(o).WhoAmI p //  "Parent.WhoAmI : Parent.getName"
Child(o).WhoAmI c // "Child.WhoAmI : Child.getName"
Child(o).WhoAmI o // "Parent.WhoAmI : Child.getName"
```

The left side of the response shows which class implemented the `WhoAmI` method that was just called, and the right side of the response shows which class implemented the `getName()` that was called on the object passed as an argument to the `WhoAmI` method.

The first two examples show the **Parent** object `p` calling the `WhoAmI` method. In the first example, a **Parent** object is passed as the argument for `WhoAmI` and in the second example, a **Child** instance is passed as the argument. Both examples act as you would expect them to. The `p` object always uses the methods implemented in the **Parent** class and the `c` object uses a method implemented in the **Child** class.

The next group of examples are the same as the first two, with one important difference: A **Child** instance is calling `WhoAmI` instead of a **Parent** instance. There's something strange about the results, however:

```
c.WhoAmI p // "Parent.WhoAmI : Parent.getName"
c.WhoAmI c // "Child.WhoAmI : Child.getName"
```

If `c` is a **Child** object, why do the results show that `c` called the **Parent** class implementation of the `WhoAmI` method? Why does it call the **Child** class implementation of `WhoAmI` in the second example?

The answer is that `Child.WhoAmI()` does not override `Parent.WhoAmI()`. It overloads it. Remember that to override a method, the signatures have to be the same. When **Child** implements `WhoAmI`, the parameter is defined as a **Child** object, but when **Parent** implements it, the parameter is defined as a **Parent**. REALbasic decides which version of the overloaded method to call based on the signature. What is tricky here is that it is easy to forget that `WhoAmI` is overloaded only in the **Child** class, not in the **Parent** class, so when a **Parent** object is passed as an argument, REALbasic uses the **Parent** class implementation of `WhoAmI`. However, the `c` object has access to the methods of the **Parent** class because it is a subclass of the **Parent** class.

The rest of the examples work on the object `o`, which is in a unique situation. It was declared to be a variable in the **Parent** class, but when it was instantiated, it was instantiated as a **Child** and this makes for some interesting results. The first three sets of responses all show that `o` is calling the **Parent** class version of the `WhoAmI` method:

```
o.WhoAmI p // "Parent.WhoAmI : Parent.getName"
o.WhoAmI c // "Parent.WhoAmI : Child.getName"
o.WhoAmI o // "Parent.WhoAmI : Child.getName"
```

The `getName()` method is where things get interesting. As expected, when the argument is an instance of the **Parent** class, the `Parent.getName()` method is executed; likewise, if it is a member of the **Child** class. But when you pass `o` as the argument to

WhoAmI, o calls the **Child** implementation of getName(), rather than the **Parent** implementation.

This seems odd because when o calls WhoAmI, it calls the **Parent** class method, but when it calls getName(), it calls the **Child** class method. Because o was declared as a member of the **Parent** class, it is cast as a member of the **Parent** class, even though you used the **Child** class Constructor. In fact, when you cast o as a **Child**, you get an entirely different set of results:

```
Child(o).WhoAmI p //  "Parent.WhoAmI : Parent.getName"
Child(o).WhoAmI c // "Child.WhoAmI : Child.getName"
Child(o).WhoAmI o // "Parent.WhoAmI : Child.getName"
Child(o).WhoAmI child(o)  // "Child.WhoAmI : Child.getName"
```

In the first of this quartet, you get the expected answer because p is a **Parent** instance, and that means that regardless of whether o is a **Child** or a **Parent**, REALbasic will call the **Parent** class implementation of WhoAmI. In the second example of this group, after o is cast as a **Child**, it behaves as expected, calling the **Child** class version of WhoAmI.

However, when o is passed as the argument to WhoAmI, it doesn't matter that o has been cast as a **Child**, it again calls the Parent.WhoAmI method. On the other hand, if you also cast the o object passed in the argument as a **Child** as well, things again are as they should be. So the question is why does this happen:

```
Child(o).WhoAmI o // "Parent.WhoAmI : Child.getName"
```

The implementation of WhoAmI that is used is determined by the type of the object passed in the parameter. The implementation of getName() is not determined by any parameter because it doesn't require a parameter. You can also do the following and get the same result:

```
Child(o).WhoAmI parent(o) // "Parent.WhoAmI : Child.getName"
```

The getName method is overridden, not overloaded. Because o was instantiated as a **Child** object, it will always call the **Child** implementation of getName. WhoAmI is overloaded, not overridden, and it is overloaded only in the **Child** class and not the **Parent** class.

Encapsulation

The idea of encapsulation is that an object does a lot of things privately that no one else needs to know about.

The **Public** methods of an object provide an interface to the outside world. It exposes the methods that can be called by other objects in your program. These typically aren't all the methods that compose the object because the object itself has lots of little details to attend to.

It's kind of like the drive-up window at a fast-food restaurant. When I pull up, the first thing I encounter is the menu that tells me what I can order; then there's the staticky speaker that I talk into to place my order. After ordering, I proceed to the first window to pay, then go to the second window to pick up my food, and then drive off.

A fast-food restaurant is encapsulated. It provides a means for me to place an order and pick it up, but I don't really have any idea what's going on inside, except that I know that when I place an order, it triggers a sequence of events inside the restaurant that results in food being handed over to me at the second window.

There's a fast-food restaurant near my house that I go to all the time. I do the same thing every time I go, and even order the same food (I'm a creature of habit). Even though my experience with the restaurant doesn't change, for all I know, they could have hired a consultant last week who showed them a new way to make hamburgers that was more cost efficient and would save them lots of money. I don't know about it and don't really need to know about it. As long as the steps I take to order the food are the same and the hamburger I order tastes the same, I really don't care.

Sometimes encapsulation is described in a slightly different way. These developers say that encapsulation means that an object should know how to do all the important things required to do its job. So if you create a class that represents a square, the class should also be able to display that square in a window or print that square on a sheet of paper. Although it's true that the class should group together related properties and methods that function as a unit, you shouldn't infer that the class needs to know about and be able to do everything related to it. If you follow that logic to the bitter end, all your programs will just be one gigantic class, and that defeats the purpose.

In practice, people often organize their classes quite differently. Some objects you use in your program will have only properties and no methods at all, which means that they don't know anything about themselves other than the data they represent. That's not a violation of the principle of encapsulation.

The fast-food restaurant I frequent doesn't slaughter its own cattle, grind it up in the back room, and cook my patties all the in same place. They hire someone to provide the ground-up meat. They "outsource." Classes and objects outsource, too. Often, a class is just a collection of other classes.

We'll dispense with the concepts for a moment. There's a practical side to object-oriented programming, too. Your goal is to create a program that maximizes code reuse, so you don't spend your life doing the same thing over and over. You also want a program that is easy to change in the future so that if something goes wrong, it's easy to figure out what it is that goes wrong. These factors are as important, and maybe more so, as the conceptual purity of your object model.

For example, I have a program that I use to write books and create websites. In fact, I'm using it right now. The "documents" I create will end up in print and sometimes online in different formats. Some of the documents are retrieved off of the file systems, others are pulled from Subversion, others are opened up using an HTTP request over the web.

If I took the encapsulation advice offered by some, I would have methods in my document class for opening files from the file system, from Subversion or through the web within the document class. However, because I know that I may be adding additional sources of data in the future, in different formats, which means I would have to go back and continue to add to this class, making it bigger and more complex over time, I created a group of classes. One group represents the data in the documents, another group handles getting the data from different sources, and a third group handles writing the documents out into different formats. If I have a new data source, I just subclass my `Provider` class and override the `open` method. I don't have to change the document class.

Access Scope: Public, Private, Protected

In practice, encapsulation is practiced by setting the scope of methods and properties. You've already encountered the idea of scope in the section on modules, but it takes on much greater importance when you're working with classes.

The whole point of encapsulation is that you want your objects to have a little modesty; not everything has to be hanging out in public for the whole world to see. The reason this is a good idea when programming is the same reason you don't want to do it personally. When things are exposed, other people have access to things they should not have access to and can thereby cause mischief. You might get sunburned or pregnant. Encapsulation, like modesty, is a virtue.

You only want to expose those parts of you that others have any business dealing with. Keep everything else safely tucked away.

When it comes to deciding how much of yourself you expose to others, or what others are allowed to do to or with you, it really depends on who that other person is.

Have you ever noticed that it's okay for you to make fun of your mom or dad or sister, but it's not okay if one of your friends does? What's considered acceptable behavior changes according to whether you're one of the family or not one of the family. Being a member of the family is a privileged position.

There are also some things that only certain members of the family are able to do, like drink beer out of the refrigerator. The kids and their friends don't get to do it because that's dad's beer and nobody is going to touch it.

This is how encapsulation works. If a method is `protected`, it's a method that you're keeping "in the family." Subclasses can call the method, but unrelated classes cannot. A method can also be designated as `private`, which is like Dad's beer stash. Only dad and no one else, not even his firstborn, can drink his beer. It's dad's private beer supply. A `private` method can be called only by the method that implements it, and no other classes, superclasses, or subclasses can touch it.

Recall that we have dealt with access scope before, when working with modules. The way that access scope is handled in modules is slightly different, with an emphasis on avoiding namespace collisions.

With classes, these terms have the following meaning:

- Public—Any other object or module can access this class member.
- Protected—Only members of this class or subclasses of this class can access this member.
- Private—Only this class can access this member; subclass members cannot.

Setting Properties with Methods

One thing that advocates of pure object-oriented programming often recommend is that you should avoid setting properties directly. Instead, you should use a method (sometimes called getter and setter methods).

It should be noted that there is a slight performance hit from using getter and setter methods because there's an extra step involved. I doubt that it is enough to make a substantial difference in most applications, but if you really want to convince yourself that it's okay to set your properties directly, I suppose that's about as good a reason as I can find.

The rationale behind this is our friend encapsulation. Being able to set properties directly is sort of like walking into the kitchen of the fast-food restaurant and making your own hamburger. It's not really safe to do that. In programming terms it's because the underlying mechanism that establishes the value for the property might change, or a subclass might want to set the value of the property in a different way. You obviously can't override the setting of properties.

Generally speaking, I think it's good advice to avoid directly accessing properties. That doesn't mean that it's bad to do otherwise. It's just that I've found, in practice, that the classes that use getter and setter methods tend to be easier to maintain over time. Your mileage may vary, as they say.

Default and Optional Parameters

Optional parameters are really just shortcuts to overloading methods. There are two ways to indicate that a parameter is optional. The obvious way is to use the `Optional` keyword, like so:

```
aMethod(aString as String, Optional anInteger as Integer)
```

The second way is to establish a default value for one of the parameters. Another way to accomplish what I just did in the previous example is this:

```
aMethod(aString as String, anInteger as Integer = 100)
```

Really, the `Optional` keyword just sets a default value of "0" to the `anInteger` variable, so the only real difference is that you get to set an arbitrary default value when using the second approach.

Declaring Variables Static and Const

When declaring variables within a method, they are treated by default like local variables that come in scope when the method starts and go out of scope when the method has completed executing. There are two variants of local variables that are also available.

Const

If you declare a variable within a method using `Const` instead of `Dim`, you are declaring a constant. The difference between this kind of constant and the other kind that I have written about is that these are local constants and so are active only for the duration of the method:

```
Const a = 100
```

This constant is local to the method, but it can be declared anywhere within the method.

Static

`Static` variables are new to REALbasic 2005. As such I haven't used them extensively in real-life, but they are an interesting addition. A static variable is like a local variable that's declared in a method except for one thing: It retains its value between invocations of that method.

```
Static a as Integer
```

Revisiting the StringParser Module

In the previous chapter, I created a `StringParser` module that did two things: it would take a string and split it up into sentences and it would take a string and split it up into words. Although a module works just fine, I think it would be worthwhile to take that same code and rethink it in terms of classes and objects.

Turning the module into a class is fairly straightforward. You can follow the same original steps to get started; the only difference is that you should click **New Class** rather than **New Module** in the editor and name it **Parser**. The differences come into play with the methods. Because one of the goals of object-oriented design is to maximize code reuse, it's instructive to think about this problem in terms of code reuse. You also want to think about it in terms of making it easy to work with. With the module, we had two different methods we could call, and they both called the same private method.

Right away, this creates an opportunity because the two methods are already sharing code—the `private Split` method.

Because I want to maximize code reuse, I'll dispense with the constant and use properties instead. The first step is to create a string property that will hold the string that will be used to split up the other string. There's no need for any other object to access

this property directly, so I'll set its scope to `Protected`. If I set the scope to `Private`, subclasses of the base class will not be able to access it, and I want them to be able to do that.

```
Parser.Delimiter as String
```

In the module, the two primary functions returned an `Array`. I could do the same with this class, but that's actually a little more awkward because when I call those functions, I have to declare an `Array` to assign the value that's returned from them. If I were to do the same with a class, I would have to declare the `Array` plus the class instance, which strikes me as extra work. Instead, I can create the `Array` as a property of the class. You create an `Array` property just like any other data type, except that you follow the name with two parentheses. I'll call this `Array Tokens`.

```
Parser.Tokens() as String
```

Next I will implement the protected `Split` method, but with a few differences. The biggest is that I do not need to return the `Array` in a function. Instead, I am going to use the `Tokens() Array`, which is a property of the class. First, this means that the function will now be a subroutine and instead of declaring a `word_buffer() Array`, I'll refer to the `Tokens() Array`.

```
Protected Sub Split(aString as String)
  Dim chars(-1) as String
  Dim char_buffer(-1) as String
  Dim x,y as Integer
  Dim pos as Integer
  Dim prev as Integer
  Dim tmp as String

  // Use the complete name for the global Split function
  // to avoid naming conflicts
  chars = REALbasic.Split(aString, "")
  y = ubound(chars)

  prev = 0
  for x = 0 to y
    pos = me.Delimiter.inStr(chars(x))
    // If inStr returns a value greater than 0,
    // then this character is a whitespace
    if pos > 0 then
      Me.Tokens.append(join(char_buffer,""))
      prev = x+1
      reDim char_buffer(-1)
    else
      char_buffer.append(chars(x))
    end if
  next
```

```
  // get the final word
  Me.Tokens.append(join(char_buffer,""))

End Sub
```

One thing you may have noticed is the use of the word `Me` when referring to `Tokens`. That's because `Tokens` is a property of the **Parser** class and, as I said before, you refer to an object by the object's name and the name of the property or method you are wanting to access. REALbasic uses the word `Me` to refer to the parent object where this method is running, and this helps to distinguish `Tokens()` as a property rather than as a local variable. REALbasic 2005 is much more finicky about this than previous versions (and rightly so, I think). Before, `Me` was mostly optional, but now it is required. We'll revisit `Me` when talking about **Windows** and **Controls**, because there is a companion to `Me`, called `Self`, that comes into play in some situations.

I also pass only the string to be parsed as a parameter, because the delimiter list value will come from the `Delimiter` property.

Now, the final two questions are how to set the value for the `Delimiter` property and how to replicate what was done previously with two different methods. I'm going to do this by subclassing Parse, rather than implementing them directly.

Create a new class and call it **SentenceParser** and set the `Super` to be **Parser**. Create a new `Constructor` that doesn't take any parameters, like so:

```
Sub Constructor()
    me.Delimiter = ".,?!()[]"
End Sub
```

Create another method called `Parse`:

```
Sub Parse(aString as String)
    me.Split(aString)
End Sub
```

Now if I want to parse a string into sentences, I do this:

```
Dim sp as SentenceParser
Dim s as String
sp = new SentenceParser()
sp.Parse("This is the first sentence. This is the second.")
s = sp.Tokens(0) // s= "This is the first sentence"
```

If I want to add the functionality of splitting the string into words, I need to create a new class, called **WordParser**, set the `Super` to **SentenceParser**, and override the `Constructor`:

```
Sub Constructor()
     me.Delimiter = " " + chr(13) + chr(10)
End Sub
```

And I can use this in this manner:

```
Dim wp as WordParser
Dim s as String
wp = new WordParser()
wp.Parse("This is the first sentence. This is the second.")
s = wp.Tokens(0) // s= "This"
s = wp.Tokens(Ubound(wp.Tokens)) // s = "second."
```

In this particular example, I think a strong argument could be made that it would have been just as easy to have stuck with the module. That's probably true, because this is a simple class with simple functionality. You do have the advantage, however, of having an easy way to add functionality by subclassing these classes, and each one uses the same basic interface, so it's relatively easy to remember how to use it.

REALbasic provides a large number of predefined classes. Most of the rest of this book will be examining them. In the next section, I'll take a look at one in particular, the **Dictionary** class, and then I'll show a more realistic (and hopefully useful) example of how you can subclass the **Dictionary** class to parse a file of a different format.

The Dictionary Class

Although you will be creating your own classes, REALbasic also provides you with a large library of classes that you will use in your applications. You've been exposed to some of them already—**Windows** and **Controls**, for example.

The **Dictionary** class is a class provided by REALbasic that you'll use a lot, and it will serve as the superclass of our new **Properties** class.

A **Dictionary** is a class that contains a series of key/value pairs. It's called a **Dictionary** because it works a lot like a print dictionary works. If you're going to look up the meaning of a word in a dictionary, you first have to find the word itself. After you've found the word you can read the definition that's associated with it.

If you were to start from scratch to write your own program to do the same thing, you might be tempted to use an `Array`. To implement an `Array` that associated a key with a value, you would need to create a two-dimensional `Array` something like this:

```
Dim aDictArray(10,1) as String
aDictArray(0,0) = "My First Key"
aDictArray(0,1) = "My First Value"
aDictArray(1,0) = "My Second Key"
aDictArray(1,1) = "My Second Value"
aDictArray(2,0) = "My Third Key"
aDictArray(2,1) = "My Third Value"
// etc…
```

If you wanted to find the value associated with the key "My Second Key", you would need to loop through the `AArray` until you found the matching key.

```
For x = 0 to 10
  If aDictArray(x, 0) = "My Second Key" Then
```

```
    // The value is aDictArray(x,1)
    Exit
  End If
Next
```

Using this technique, you have to check every key until you find the matching key before you can find the associated value. This is okay if you have a small list, but it can get a little time consuming for larger lists of values. One feature of a printed dictionary that makes the search handy is that the words are all alphabetized. That means that when you go to look up a word, you don't have to start from the first page and read every word until you find the right one. If the keys in the `Array` are alphabetized, the search can be sped up, too.

Using the current `Array`, the best way to do this would be with a binary search, which isn't a whole lot different from the way you naturally find a word in the dictionary. For example, if the word you want to look up starts with an "m," you might turn to the middle of the dictionary and start your search for the word there. A binary search is actually a little more primitive than that, but it's a similar approach. To execute a binary search for "My Second Key", a binary search starts by getting the upper bound of the `Array` ("10", in this case) and cutting that in half. This is the position in the `Array` that is checked first .

```
result = StrComp(aDictArray(5,0), "My Second Key")
```

If the result of `StrComp` is `0`, you've found the key. However, if the value returned is `-1`, then "My Second Key" is less than the value found at `aDictArray(5,0)`. Because the key of `aDictArray(5,0)` would be "My Sixth Key", `-1` is the answer we would get. So the next step is to take the value `5` and cut it in half, as well (I'll drop the remainder). This gives us `2`. Now you test against `aDictArray(2,0)` and you still get `-1`, so you cut `2` in half, which gives you `1`. When you test `aDictArray(1,0)`, you find that it matches "My Second Key", so you can now find the value associated with it.

This is all well and good if you happen to have a sorted two-dimensional `Array` consisting of one of REALbasic's intrinsic data types. If the `Array` isn't sorted (and you have to jump through some hoops to get a sorted two-dimensional `Array`), things begin to get complicated. What happens if the key is not an intrinsic data type?

These are really the two reasons for using **Dictionaries**—you don't have to have a sorted list of keys to search on, and your keys do not have to be limited to intrinsic data types. It's still easier (and possibly more efficient) to sequentially search through an `Array` if the list of keys is relatively short. As the list grows larger, the more beneficial the use of a dictionary becomes.

The Properties File Format

For our new subclass, we are going to be parsing a properties file. Java developers often use properties files in their applications. The file format used by properties files is very simple, so I will use this format in my example.

The format is a list of properties and an associated value separated (or delimited) by an equal sign (=). In practice, the property names are usually written using dot notation, even though that's not necessary. Here's an example:

```
property.color=blue
property.size=1
```

The format doesn't distinguish between integers and strings or other data types, so this unknown is something that the module will need to be prepared to deal with. Because the properties file has a series of key/value pairs, a **Dictionary** is well suited to be the base class. The **Dictionary** already has members suited for dealing with data that comes in a key/value pair format, so our subclass will be able to use those properties and methods, while adding only a few to deal with the mechanics of turning a string in the properties file format into keys and values in the **Dictionary**.

Dictionary Properties

Dictionaries use hash tables to make searches for keys faster. Without getting into too much detail, you can adjust the following property at times in order to optimize performance:

```
Dictionary.BinCount as Integer
```

In most cases you'll never need to adjust this because the class automatically adjusts it as necessary, but there are times when you can improve performance. This is especially true if you know you are going to have a particularly large dictionary with lots of key/value pairs.

The following property returns the number of key/value pairs in the dictionary:

```
Dictionary.Count as Integer
```

Dictionary Methods

The following method removes all the key/value pairs from the **Dictionary**:

```
Sub Dictionary.Clear
```

This removes a particular entry, based on the key:

```
Sub Dictionary.Remove(aKey as Variant)
```

When using a **Dictionary**, you almost always access the value by way of the key. That's more or less the whole point of a **Dictionary**. You do not always know if any given key is available in the **Dictionary**, so you need to find out if the key exists first before you use it to get the associated value. Use the following method to do so:

```
Function Dictionary.HasKey(aKey as Variant) as Boolean
```

The values in a **Dictionary** are accessed through the `Value` function.

```
Function Dictionary.Value(aKey as Variant) as Variant
```

Here's an example of how to use the `Value` function:

```
Dim d as Dictionary
Dim s as String
d = new Dictionary
d.value("My key") = "My value"
If d.HasKey("My Key") Then
  s = d.value("My key") // s equals "My value"
End If
```

The **Dictionary** class also provides you with a way to get access to the keys by their index, which can be helpful in some circumstances, using the following method:

```
Function Dictionary.Key(anIndex as Integer) as Variant
```

There's no guarantee that the keys are in any particular order, but this can be used in circumstances where you want to get all the key/value pairs. Assume `d` is a **Dictionary** with 100 key/value pairs:

```
Dim d as Dictionary
Dim x as Integer
Dim ThisKey, ThisValue as Variant
Dim s as String

// Assign 100 haves to d…
For x = 0 to 99 //
  ThisKey = d.Key(x)
  ThisValue = d.Value(ThisKey)
  s = s  + ThisKey.StringValue + ":" + ThisValue.StringValue + Chr(13)
Next
```

This example takes all the keys and values in the **Dictionary** and places each key/value pair on a single line, separated by a colon. The ASCII value for a newline character is 13, which explains the `Chr(13)`.

The `Lookup` function is new for REALbasic 2005:

```
Function Dictionary.Lookup(aKey as Variant, defaultValue as Variant) as Variant
```

It works like the `Value` function, but instead of passing only the key, you also pass a default value to be used if the key doesn't exist in the dictionary. This saves you the nearly unbearable hassle of having to use the `HasKey()` function every time you try to get at a value in the **Dictionary**.

Here's how it works:

```
Dim d as Dictionary
Dim s as String
d = New Dictionary()
d.Value("Alpha") = "a"
d.Value("Beta") = "b"
s = d.Lookup("Delta", "d") // s equals "d"
```

In this example, the key "Delta" does not exist, so `d` returns the default value "d" and this value gets assigned to `s`.

This next example can be used in replacement of this older idiom, which is the way you had to do it before the `Lookup` method was added:

```
Dim d as Dictionary
Dim s as String
d = New Dictionary()
d.Value("Alpha") = "a"
d.Value("Beta") = "b"
If d.HasKey("Delta") Then
  s = d.Value("Delta")
Else
  s = "d"
End If
```

Example: Creating a Properties Class

In the following pages, I will create a new class called **Properties**, which will be a subclass of the **Dictionary** class.

Properties.Constructor

There are three potential scenarios to consider for the `Constructor`. You may get the data to be parsed for the **Properties** class from a file or a string, and you also may want to be able to instantiate the class without any data so that you can add it later, either by individually setting the key/value pairs or by some other method.

To accomplish this, I'll overload the `Constructor` method with three versions. If I did nothing, the default `Constructor` would be used, which takes no parameters. However—and this is a big "however"—if I overload the `Constructor` class with any other `Constructor`, the default `Constructor` goes away and is not accessible. The reason is that it allows you to require that the class accept a parameter in the `Constructor` because you do not always want the default `Constructor` available.

This means that you have to reimplement the default `Constructor` to retain the capability to instantiate the class with no parameters, so that is what I do first:

```
Sub Constructor()
  // Do nothing
End Sub
```

The second implementation will accept a string in the parameter. Whenever this `Constructor` is called, the class automatically parses the file as part of the `Constructor`. This means that after the class is instantiated, the key/value pairs are accessible, with no further steps required.

```
Sub Constructor(myPropertyString as String)
```

```
  If myPropertyString <> "" Then
    Me.parsePropertyFile(myPropertyString)
  Else
    // An error as occurred
  End If

End Sub
```

The following implementation accepts a `FolderItem`. It tests to make sure the `FolderItem` is readable and writeable, both of which are functions of the `FolderItem` class. If the FolderItem instance is readable and writeable, it sends the file to the overloaded `parsePropertyFile` function.

```
Sub Constructor(myPropertyFile as FolderItem)
  If myPropertyFile.exists Then

    If myPropertyFile.IsReadable and myPropertyFile.IsWriteable Then
      Me.parsePropertyFile(myPropertyFile)
    Else
      // An error has occurred
    End If

  Else
    // An error has occurred
  End If

End Sub
```

Properties.get

```
Function get(aKey as string, aDefault as string) As string
  Dim s as String
  s = Me.Lookup(aKey, aDefault)
End Function
```

The `get()` function is also overloaded to take only one string as a parameter. When this occurs, I need to test first to see whether the **Dictionary** has the key passed in the parameter, and if it doesn't, respond appropriately.

```
Function get(aKey as string) As string
  Dim s as String

  If Me.HasKey(aKey) = True Then
    s = Me.Value(aKey)
    Return
  Else
    // Handle the Error
```

```
    Return ""
  End If
End Function
```

You can accomplish the equivalent by using the **Dictionary**'s `Lookup()` function. Remember that nothing is stopping you from calling the `Lookup()` function directly from other parts of your program because it is a `public` method of **Dictionary**.

```
Function get(aKey as string) As string
  Dim s as String

  s = Me.Lookup(aKey, "")
End Function
```

I can also replicate the same functionality by calling the first `get()` function from the second:

```
Function get(aKey as string) As string
  Dim s as String

  s = Me.get(aKey, "")
End Function
```

Of the three options I have given for implementing function `get(aKey) as String`, the last option is preferable, in my opinion. You may be wondering why you should bother implementing the `get()` method at all, because you can use the native **Dictionary** methods anyway. One reason is because `get()` provides a simpler syntax. **Dictionaries** can use `Variants` for keys and values, but I really care only about strings, so `get()` makes the assumption that it's a string. More importantly, because of the two versions of `get()` that I am using, I can change the implementation in the future of how the default value is generated when no default value is given, or add tests to the code to make sure that the default value that is passed is a valid default value.

There is yet another approach to implementing `get()` that accomplishes the same goals, more or less: instead of overloading `get()`, you can override and overload `Lookup()` to achieve the same effect.

I'll override `Lookup()` first. Remember that to override a method, the signature has to be the same, which means the parameters must accept `Variants`. Also, the reason I said I liked using `get()` was that it gave me an opportunity to test for legal values first. You can do the same here, by calling the parent class's version of `Lookup` after you've tested the values:

```
Function Lookup(aKey as Variant, aDefaultValue as Variant) as String
  Dim key,default as String
  key = aKey.StringValue
  default = aDefaultValue.StringValue
  If default <> "" Then
    Return Dictionary.Lookup(key, default).StringValue()
  End If
End Function
```

I can also keep things simple by using an overloaded version of `Lookup()` that accepts only a key in the parameter and handles the default value automatically:

```
Function Lookup(aKey as Variant) as String
  Dim key,default as String
  key = aKey.StringValue
  default = aDefaultValue.StringValue
  If default <> "" Then
    Return Dictionary.Lookup(key, default).StringValue()
  End If
End Function
```

Note that in both cases, I returned a string rather than a `Variant`. REALbasic doesn't pay attention to the return value when determining whether a method is overloaded. This allows me to force the return of a string, even though the original `Lookup()` implementation returns a `Variant`.

Properties.set

In addition to `get()`, I also implemented a `set()` subroutine. Again, this is for simplicity's sake because I could also just call the `Dictionary.Value()` function directly. It also means I can test the values before assigning them, which always helps. I also use the `Assigns` keyword, which alters the way that the method can be called in your program. An example of how it is used follows this example:

```
Sub set(aKey as string, assigns aProperty as string)

If aKey <> "" and aProperty <> "" Then
  Me.Value(aKey) = aProperty
Else
// An error occurred
End if
End Sub
```

By using the `Assigns` keyword in the parameter, you can set a key/value pair using the following syntax, which mirrors the syntax used for the `Value()` function of **Dictionary**:

```
Dim prop as Properties
prop = New Properties()
prop.set("FirstKey") = "FirstValue"
```

Properties.parsePropertyFile

The first step to parsing the file is to split the string into individual lines. I have two functions for this—one that accepts a string and another that accepts a **FolderItem**. The string version splits the string on a newline character (ASCII value of 13) and then cycles through each line, parsing the lines individually.

```
Protected Sub parsePropertyFile(myFile as string)
  Dim my_array(-1) as String
  Dim line as String

  my_array = myFile.split(Chr(13))

  For Each line In my_array
    Me.parseLine(line)
  Next
End Sub
```

A second version of `parsePropertyFile` accepts a **FolderItem** as a parameter. **FolderItems** are discussed at length later in the book, but the code in this example is sufficiently self-explanatory that you get the general idea of what is happening. When you open a **FolderItem** as a text file, it returns a **TextInputStream**. A **TextInputStream** allows you to read it line by line by calling the `ReadLine()` function.

```
Protected Sub parsePropertyFile(myPropertyFile as folderItem)
  Dim textInput as TextInputStream
  Dim astr as string

  If myPropertyFile Is Nil Then
       // An error has occurred
    Return
  End If

  If myPropertyFile.exists Then
    Me.file = myPropertyFile
    textInput = me.file.OpenAsTextFile
    Do
      astr=textInput.ReadLine()
      Me.parseLine(astr)
    Loop Until textInput.EOF

    textInput.Close
  Else
    // An error has occurred
  End If

End Sub
```

Properties.parseLine:

Each line is parsed individually.

```
Protected Sub parseLine(my_line as string)
  Dim aKey, aValue as string
```

```
  // Split line at "=", and ignore comments
  If Left(my_line, 1) <> "#" Then
    If CountFields(my_line, "=")= 2 Then

      aValue = Trim(NthField(my_line, "=", 2))
      aKey = Trim(NthField(my_line, "=", 1))

      Me.set(aKey) = aValue

    End If

  End If // starts with "#"

End Sub
```

The first thing I test for is to see whether the line starts with a # character. In property files, a # at the beginning of a line indicates a comment, and because comments are only for human consumption, I can safely ignore them in my program.

The next step is to split the line into two values, one for the property and the other for the value. Although I could use the `Split()` function, I have chosen to use the `CountFields`/`NthField` combination primarily because it makes it a little easier to count how many fields there are. Because each property is separated from its value by an equal sign, I will parse the line only if there are two fields in the line (which is the same thing as saying that there is only one "=" sign on the line).

The property name, or key, is in the first field, and the value is in the second field. When extracting the values using `NthField`, I trim the results, which removes whitespace on either side of the string. This accounts for situations where there are extra spaces between the "=" sign and the values themselves. As a result, both of these formats will be parsed correctly:

```
aProperty=aValue
aProperty = aValue
```

Finally, I use the `set()` method to add the key/value pair to the **Dictionary**.

Now that all the members of the class are implemented, here is an example of how you can use the class:

```
Dim prop as Properties
Dim s as String
Dim propStr as String
propStr = "First=FirstValue" + Chr(13) + "Second=SecondValue"
prop = New Properties(propStr)
s = prop.get("First") // s equals "FirstValue"
prop.set("Third") = "ThirdValue"
s = prop.get("Third") // s equals "ThirdValue"
```

Data-Oriented Classes and Visual Basic Data Types

In the previous chapter, I said that some data types in Visual Basic don't exist in REALbasic as data types. The reason is that REALbasic often has an object-oriented alternative. For example, an intrinsic data type in Visual Basic is a **Date**, but in REALbasic it's a class. In other situations, I said that similar functionality could be implemented with the **MemoryBlock** class. Now that you know about classes, I can review these two classes in more detail and complete the discussion of data types and how REALbasic compares with Visual Basic. I will also review the **Collection** class, because it is very similar to the **Dictionary** class. REALbasic recommends sticking to the **Dictionary** class for new projects and retains the **Collection** class only for compatibility with Visual Basic and earlier versions of REALbasic.

Date Class

Date is an intrinsic data type in Visual Basic, but it's a class in REALbasic. It measures time in seconds since 12:00 a.m., January 1, 1904. The total number of seconds since that time is how a particular date and time are identified, and it is stored in the `TotalSeconds` property of the **Date** class as a double.

The `Variant` type treats **Dates** in a special way to help maintain compatibility with Visual Basic. The `VarType()` function returns a 7 as the data type for dates, rather than 9, which is what is typically done with objects. Basically, all the `Variant` is doing is storing the `TotalSeconds` property as a double.

ParseDate Function

There is a `ParseDate` helper function that accepts a date formatted as a string and parses it into a `Date` object. It returns a Boolean representing success or failure in parsing the string that was passed to it. If it returns a Boolean, how then do you pass it a string and get a **Date** object back? The function signature looks like this:

```
REALbasic.ParseDate(aDate as String, aParsedDate as Date) as Boolean
```

The answer is that objects, like **Date**, are passed by reference, rather than by their value. I wrote earlier that variables that were one of REALbasic's intrinsic data types were called scalar variables. These variables refer to the value held in memory, whereas variables that referred to objects were references to the object, like a pointer to a particular section of memory. When you pass the **Date** object to the `ParseDate` function, the `ParseDate` function does its work on the **Date** object passed to it so that when it changes a value of the **Date** object, that change is reflected in the original variable that was passed to the function.

```
Dim d as Date
Dim s,t as String
Dim b as Boolean
```

```
s = "12/24/2004"
b = ParseDate(s, d) // b is True
t = d.ShortDate // t = "12/24/2004"
```

Note that all you have to do is `Declare d` as a **Date**; you do not need to instantiate it prior to passing it to the `ParseDate` function.

You can also do the same trick with scalar variables if you use the `ByRef` keyword in the function signature. There are actually two keywords here—`ByRef` and `ByVal`—and scalar variables are, by default, passed `ByVal`.

The acceptable formats of the string are the following:

```
1/1/2006
1-1-2006
1.1.2006
1Jan2006
1.Jan.2006
1 Jan. 2006
```

Date Properties

`TotalSeconds` reflects how the date and time are actually stored in a `Date` object. When you instantiate a new `Date` object, it automatically gets set to the current date and time.

```
Date.TotalSeconds as Double
```

You can also get or set different components of the time with the following properties. Changes made to these will be reflected in `TotalSeconds`:

```
Date.Second as Integer
Date.Minute as Integer
Date.Hour as Integer
Date.Day as Integer
Date.Month as Integer
Date.Year as Integer
```

The following two methods set the date or date and time using the standard SQL format used in databases. This comes in handy when dealing with databases.

```
Date.SQLDate as String
```

You can get or set the date using the format `YYYY-MM-DD`.

```
Date.SQLDateTime as String
```

You can get or set the date using the format `YYYY-MM-DD HH:MM:SS`.

These properties are all read-only and return the date or time in different formats, according to your wishes:

```
Date.LongDate as String
Date.LongTime as String
Date.ShortDate as String
Date.ShortTime as String
Date.AbbreviatedDate as String
```

The following example shows how they are used.

```
Dim d as new Date()
Dim dates() as String

d.TotalSeconds = 3204269185

dates.append "TotalSeconds: " + format(d.TotalSeconds, "#")
dates.append "Year: " + str(d.Year)
dates.append "Month: " + str(d.Month)
dates.append "Day: " + str(d.Day)
dates.append "Hour: " + str(d.Hour)
dates.append "Minute: " + str(d.Minute)
dates.append "Second: " + str(d.Second)
dates.append "--"
dates.append "SQLDateTime: " + d.SQLDateTime
dates.append "SQLDate: " + d.SQLDate
dates.append "--"
dates.append "DayOfWeek: " + str(d.DayOfWeek)
dates.append "DayOfYear: " + str(d.DayOfYear)
dates.append "WeekOfYear: " + str(d.WeekOfYear)
dates.append "--"
dates.append "LongDate: "  + d.LongDate
dates.append "LongTime: " + d.LongTime
dates.append "ShortDate: " + d.ShortDate
dates.append "ShortTime: " + d.ShortTime
dates.append "AbbreviatedDate: " + d.AbbreviatedDate

EditField1.text = join(dates, EndOfLine)
```

I executed this action using three different locale settings (which I changed on my Macintosh in System Preferences, International). The first setting was the standard (for me) setting of United States, using the Gregorian calendar. I followed that by setting my locale to France, and then Pakistan, using the Islamic civil calendar. The only properties that change according to your locale are the `LongDate`, `LongTime`, `ShortDate`, `ShortTime`, and `AbbreviatedDate`.

United States, Gregorian Calendar:

```
TotalSeconds: 3204269185
Year: 2005
Month: 7
Day: 15
Hour: 10
Minute: 46
Second: 25
--
SQLDateTime: 2005-07-15 10:46:25
SQLDate: 2005-07-15
--
```

```
DayOfWeek: 6
DayOfYear: 196
WeekOfYear: 29
--
LongDate: Friday, July 15, 2005
LongTime: 10:46:25 AM
ShortDate: 7/15/05
ShortTime: 10:46 AM
AbbreviatedDate: Jul 15, 2005
```

France, Gregorian Calendar:

```
LongDate: vendredi 15 juillet 2005
LongTime: 10:46:25
ShortDate: 15/07/05
ShortTime: 10:46
AbbreviatedDate: 15 juil. 05
```

Pakistan Islamic Civil Calendar:

```
LongDate: Friday 8 Jumada II 1426
LongTime: 10:46:25 AM
ShortDate: 08/06/26
ShortTime: 10:46 AM
AbbreviatedDate: 08-Jumada II-26
```

MemoryBlock Class

The **MemoryBlock** class is a catch-all class that lets you do a lot of interesting things with REALbasic. **MemoryBlocks** are generally helpful when dealing with any binary data and can replace some of the missing data types from Visual Basic. They are commonly used with `Declares`, which are statements in REALbasic that allow you to access functions in shared libraries.

A **MemoryBlock** is exactly what is says it is: it's a stretch of memory of a certain byte length that you can access and manipulate.

The NewMemoryBlock Function

```
REALbasic.NewMemoryBlock(size as Integer) as MemoryBlock
```

There are two ways to instantiate **MemoryBlocks** in REALbasic. The first is the old-fashioned way, using the `New` operator, and the second uses the global `NewMemoryBlock` function.

```
Dim mb as MemoryBlock
Dim mb2 as MemoryBlock
mb = New MemoryBlock(4)
mb2 = NewMemoryBlock(4)
```

You may have to look closely to see the difference between the two techniques. The second one is missing a space between `New` and **MemoryBlock** because it's a global function and not a `Constructor` for the **MemoryBlock** class. Whether you use the traditional way of instantiating an object or the `NewMemoryBlock` function, you need to pass an integer indicating how many bytes the **MemoryBlock** should be. The integer can be 0, but some positive number has to be passed to it (a negative number causes it to raise an **UnsupportedFormat** exception).

In this example, both **MemoryBlocks** are 4 bytes long. The fact that a **MemoryBlock** can be of an arbitrary length is what makes it so flexible. It can be anywhere from 0 bytes to however much memory your computer has available (the `NewMemoryBlock` function returns Nil if there is not enough memory to create the **MemoryBlock**).

Visual Basic has a Byte data type that takes up 1 byte. This can be replicated with **MemoryBlock** quite easily:

```
Dim bytetype as MemoryBlock
Dim i as Integer
bytetype = NewMemoryBlock(1)
bytetype.byte(0) = 1
i = bytetype.byte(0) // i equals 1
```

MemoryBlock provides a slew of functions to let you get at the individual bytes it contains. I'll review those shortly.

MemoryBlock Properties

This property refers to the order in which bytes are sequenced when representing a number:

```
MemoryBlock.LittleEndian as Boolean
```

I know that's kind of an obtuse sentence, but that's about the best I can do at the moment. As is often the case, an example will be more illuminating. Recall that an integer is made up of 4 bytes. When the computer stores or processes those 4 bytes, it expects them to be in a particular order. This makes perfect sense. The number 41 is not the same as 14, for instance. The problem is that not all operating systems use the same standard. Intel x86 platforms use one type and Macintosh uses another. (With the recent announcement that Macs are moving to Intel chips, this may be less of a problem.) Linux runs on Intel, so you can expect those applications to be the same as those compiled for Windows.

In any event, the example is this. Let's use a really big number, like 10 million. If you look at that number expressed in hexadecimal format on a Macintosh, it looks like this:

```
00 98 96 80
```

If you do it on an Intel machine, which is LittleEndian, it looks like this:

```
80 96 98 00
```

The terms *big-endian* and *little-endian* refer to where the "most significant byte" is in position. In big-endian format, the most significant byte comes first (which also happens to be the way our decimal number system works. In little-endian systems, the least significant byte comes first. According to WikiPedia, this particular nuance can also be referred to as *byte order*.

You can both read and write to this property. If you are writing a cross-platform application, you need to be aware of this if you will be sharing binary data across systems.

```
MemoryBlock.Size as Integer
```

You can read and write to this property. It refers to the size, in bytes, of this **MemoryBlock**. If you modify the `Size`, the **MemoryBlock** will be resized. Bear in mind that if you make it smaller than it currently is, you might lose data.

MemoryBlock Methods

All five of the following functions return a value of the given type, at the location specified by the `Offset` parameter:

```
Function MemoryBlock.BooleanValue(Offset as Integer) as Boolean
Function MemoryBlock.ColorValue(Offset as Integer) as Color
Function MemoryBlock.DoubleValue(Offset as Integer) as Double
Function MemoryBlock.SingleValue(Offset as Integer) as Single
Function MemoryBlock.StringValue(Offset as Integer, [Length as Integer]) as String
```

Because each of these data types has a predefined size (except for string), all you do is pass the `Offset` to it. If you call `MemoryBlock.DoubleValue(0)`, the **MemoryBlock** will return a double based on the first 8 bytes of the **MemoryBlock**.

When using `StringValue`, you may specify the length of the string, but you do not have to. If you don't, it will just return a string from the offset position to the end of the **MemoryBlock**. The string can contain nulls and other non-normal characters, so some caution may be necessary when doing this, unless you know for certain what values you will come across.

These three functions work very much like their global function cousins `LeftB`, `MidB`, and `RightB`:

```
Function MemoryBlock.LeftB(Length as Integer) as MemoryBlock
Function MemoryBlock.MidB(Offset as Integer, [Length as Integer]) as MemoryBlock
Function MemoryBlock.RightB(Length as Integer) as MemoryBlock
```

They do the same thing except that instead of returning strings, they return **MemoryBlocks**.

Get or set a 1-byte integer:

```
Function MemoryBlock.Byte(Offset as Integer) as Integer
```

```
 Get or set a signed, 2-byte integer:Function MemoryBlock.Short(Offset as Integer)
as Integer
```

```
 Get or set an unsigned 2-byte integer.:Function MemoryBlock.UShort(Offset as
Integer) as Integer
```

Get or set a 4-byte integer, just like REALbasic's native type.

```
Function MemoryBlock.Long(Offset as Integer) as Integer
```

These four functions, `Byte`, `Short`, `Ushort`, and `Long`, return integers based on their offset position. They return integers even though they often refer to data that requires fewer bytes to represent, because an integer is really the only thing they can return—it's the smallest numeric data type. The primary use of this is when handling `Declares`, because it allows you to pass values and access values using a broader selection of data types than those offered by REALbasic.

Also, note that when I say that one of these functions "gets or sets a 2-byte integer," I really mean an integer within the range that can be expressed by 2 bytes. An integer is an integer, and it takes up 4 bytes of memory when REALbasic gets hold of it, even if it can be stored in a **MemoryBlock** with fewer bytes.

MemoryBlock.Cstring, MemoryBlock.Pstring, MemoryBlock.Wstring, MemoryBlock.Ptr:

```
Function CString(Offset as Integer) as String
```

Returns a String terminated.

```
Function PString(Offset as Integer) as String
Function WString(Offset as Integer, [Length as Integer]) as String
Function Ptr(Offset as Integer) as MemoryBlock
```

A pointer to an address in memory. Used by Declares.

Example: Mimicking a Structure with a MemoryBlock

I'd be getting ahead of myself if I gave an example of how to use a **MemoryBlock** with a `Declare`, so I've come up with another example. One advantage to using a **MemoryBlock** (in some cases) is that you can store data in it more efficiently and can often access it faster.

To illustrate this, I've developed two classes, both of which emulate a structure. The first is a class that has only properties, and the second is a subclass of **MemoryBlock**. These two structurelike classes will hold data representing the two-character United States state abbreviation, the five-digit ZIP code, a three-digit area code, and a seven-digit phone number.

The first class is called **Struct**, and it has the following properties:

```
Struct.State as String
Struct.Zip as Integer
Struct.AreaCode as Integer
Struct.PhoneNumber as Integer
```

If I were to save these items as strings, I would need 17 bytes to represent it—one for each character. The **Struct** class already reduces that to 14 bytes because it uses Integers

for the ZIP code, the area code, and the phone number, all of which can be expressed as Integers.

Implementing the **MemoryBlock** structure will be a little different because to create the **MemoryBlock**, I first have to know how many bytes to allocate for it. I could allocate 14 bytes, just like the class version, but I would be allocating more bytes than I need. For example, an area code is three digits, meaning it can be anywhere from 0 to 999. Although an Integer can represent that number, an integer takes 4 bytes and you don't need 4 bytes to represent 999; all you need are 2 bytes. The same is true for the ZIP code. The only number that requires an Integer is the phone number. This means we can shave off 4 bytes and create a **MemoryBlock** of only 10 bytes. That's a fairly significant size difference.

The next question is how do we best implement the **MemoryBlock**. I could use the `NewMemoryBlock` function and pass it a value of 10 to create an appropriate **MemoryBlock**, but that means I'd have to keep passing 10 to the function each time I wanted to create one and that seems to me to be a bit of a hassle; because I'm a little lazy by nature, I'd like to avoid that if at all possible.

The next option would be to instantiate a `New` **MemoryBlock**, but that, too, requires that I pass a parameter to the `Constructor` and I don't want to do that either. All I want is a class that I can instantiate without having to think about it. The answer is to subclass **MemoryBlock** and overload the `Constructor` with a new `Constructor` that does not take any integers as a parameter. Then, while in the `Constructor`, call the `Constructor` of the superclass. Create a subclass of **MemoryBlock** and call it **MemBlockStruct**. Create the following `Constructor`:

```
Sub MemBlockStruct.Constructor()
  MemoryBlock.Constructor(10)
End Sub
```

Now you can do this to create a 10-byte **MemoryBlock**:

```
Dim mb as MemBlockStruct
Dim sz as Integer
mb = New MemBlockStruct
sz = mb.Size // sz = 10
```

Because this is a subclass of **MemoryBlock**, you can still call the original constructor and pass any value you want to it, like so:

```
Dim mb as MemBlockStruct
Dim sz as Integer
mb = New MemBlockStruct(25)
sz = mb.Size // sz = 25
```

If you don't want that to happen, you need to override that `Constructor` and ignore the value that is passed to it:

```
Sub MemBlockStruct.Constructor(aSize as Integer)
 Me.Constructor()
End Sub
```

Note that I could have called the superclass `MemoryBlock.Constructor(10)`, like I did in the other `Constructor`, but that would mean that if at a later time I decided I wanted to add 1 or more bytes to the **MemBlockStruct** class, I would have to go back and change the value from 10 to something else in both `Constructors`. This way, should I decide to do that, I need to change only the `MemBlockStruct.Constructor()` method and no other.

I now have done enough to have two workable classes. Here is how they can be used in practice:

```
Dim ms as MemBlockStruct
Dim ss as Struct
Dim a,b,c as Integer
ss = new Struct()
ss.State = "NH"
ss.Zip = 3063 // the first character is "0", which is dropped
ss.AreaCode = 603
ss.PhoneNumber = 5551212
ms = new MemBlockStruct()
ms.StringValue(0,2) = "NH"
ms.Short(2) = 3063
ms.Short(4) = 603
ms.Long(6) = 5551212
a = ms.Short(2) // a = 3063
b = ms.Short(4) // b = 603
c = ms.Long(6) // c = 5551212
```

Accessing the values of the **MemBlockStruct** class by the offset position is a little clunky, so I can optionally take the step of creating methods for this class to make it easier. Here is an example for how the methods would look for setting the state abbreviation:

```
Function MemBlockStruct.setState(aState as String) as String
  If LenB(aState) = 2 Then
    me.StringValue(0,2) = aState
  End If
End Function

Function MemBlockStruct.getState() as String
   Return me.StringValue(0,2)
End Function
```

Collection Class

The **Collection** class is another class, similar to the **Dictionary** class. The documentation for REALbasic recommends that you use the **Dictionary** class instead, but that they have retained the collection class for reasons of compatibility with Visual Basic.

The big differences between a **Collection** and a **Dictionary** are these:

- **Collections** can only have strings as their keys.
- A **Collection** does not use a hash table like a **Dictionary** does. This means that the time required searching for a value is determined by how many key/value pairs are in the collection. The larger the collection, the longer, on average, it will take to find your value.

Collection Methods

Collection uses a slightly different terminology than **Dictionaries** do. The `Add` method is roughly equivalent to `Value()` for **Dictionaries**, in the sense that you use it to add items to the **Collection**.

```
Sub Collection.Add(Value as Variant, Key as String)
```

Much like it does with a **Dictionary**, the `Count` property returns the number of items in the **Collection**:

```
Function Collection.Count As Integer
```

As you would expect, you can get and remove values using the key:

```
Function Collection.Index(Key as String) as Variant
Function Collection.Remove(Key as String) as Variant
```

You can also get or remove an item by referring to it's indexed position:

```
Function Collection.Index(Index as Integer) as Variant
Function Collection.Remove(Index as Integer) as Variant
```

When getting or removing an item in a collection using the index, remember that **Collection** indexes are 1 based, not 0 based like `Array`s and **Dictionaries**.

Advanced Techniques

In this section I will review some relatively more advanced programming techniques you can use to extend the functionality of your REALbasic classes.

Interfaces and Component-Oriented Programming

Interfaces are an alternative to subclassing. With subclassing, one class can be a subclass of another. Interfaces do not deal in subclasses. When talking about Interfaces, you say that a class implements a particular Interface.

Interfaces are another means by which you encapsulate your program. This is also one of those areas where the term is used differently in different programming languages. In REALbasic, an interface is used much in the same way that the term is used with Java. Other languages, such as Objective-C, refer to a similar concept as a protocol.

Nevertheless, when one class is a subclass of another, it shares all the same members with the parent class. Although the subclass may optionally reimplement or override a method from the parent class, it doesn't have to reimplement any method unless it wants to modify what that method does. The benefit to this, so the story goes, is that it enables code reuse. Simply put, you don't have to write any new methods for the subclass to get the functionality resident in the superclass.

There are some cases when you have two objects (or classes) that are related and that perform the same tasks, but they both do it in such a fundamentally different way that establishing the class/subclass relationship doesn't really buy you anything. At the same time, it's convenient to formally establish this relationship so that the different objects can be used in similar contexts. If two class share a common set of methods, it is possible to link these two classes together by saying they share a common interface. This task involves defining the interface itself and then modifying to classes to indicate that they both implement this common interface.

When this is the case, no code is shared. In the immediate sense, at least, code reuse isn't the object of this activity, but as you will see, it does make it possible to streamline the code that interacts with objects that share the same interface, which is why it can be so helpful.

When I talked about encapsulation, I used the fast-food restaurant drive-up window as an example. The fast-food restaurant I was thinking about when I wrote that example had three points of interaction for me: the first was the menu and speaker where I placed my order, the second was the first window, where I paid, and the third was the second window where I picked up my order. I said that the fast-food restaurant was encapsulated because I only had three touch points where I interacted with it, even though a whole lot happened between the time I placed my order and picked up my order.

Fast-food restaurants aren't the only places that use the drive-up window "interface." Banks use them, too. Even liquor stores in Texas do. At least they did when I lived in Texas—at that time the drinking age was 19, and as a 19-year-old living in Texas, I thought that drive-up liquor stores were a very clever idea. I know better than that now. But I digress.

The point is that fast-food restaurants, banks, and liquor stores really don't have all that much in common, other than the drive-up window. As a consumer of hamburgers, liquor, and money (for hamburgers and liquor), I go to each place for entirely different reasons, but there is an advantage to my knowing that all three locations have drive-up windows. From a practical perspective, it means that I can drive to all three locations and never have to get out of my car. As long as I am in my car, I can interact with things that have drive-up windows. I might drive to the bank for a little cash, then swing by the fast-food restaurant for a hamburger and cola, followed by a quick jaunt to the liquor store for an aperitif. Same car, same guy at all three locations.

In my own humble opinion, I think interfaces are underrated and underused in REALbasic. Whenever you read about object-oriented programming, you almost invariably read about classes and objects and inheritance and things like that. Based on my own experience, I used to try to squeeze everything into a class/subclass relationship,

regardless of whether it made sense to do so, when I could more easily have implemented an interface and be left with a more flexible program. So my advice to you (painfully learned, I might add) is to give due consideration to interfaces when you are thinking about how to organize your program.

Interfaces seem to have higher visibility in the Java programming world. I really began to see and understand their usefulness when I worked with the Apache Cocoon content-management framework. At the heart of Cocoon is the Cocoon pipeline, and the pipeline represents a series of steps that a piece of content must take to be displayed in a web page, or written to a PDF file. Basically, there are three stages:

1. The content must first be generated. Perhaps it resides as a file on the local file system, or it could live somewhere out on the Web, or it could even be stored in a database or some other dynamic repository.
2. The content must then be converted or transformed into the format required for the particular task at hand. If you are using Cocoon to manage your website, you might take the source data and transform it into HTML.
3. Finally, the content must be sent to the destination—either to a web browser at some distance location or into a file of a particular format to be stored on your computer.

In all three cases, the basic task is the same, but the specific steps taken within each stage are different, depending on the situation.

Cocoon uses a component-based approach; a *component* is basically an object that implements a particular interface. In the Cocoon world of components there are Generators, Transformers, and Serializers. These are not classes—they are interface definitions that say that any object that is a Generator is expected to behave in a certain way. There is a component manager that manages these components and shepherds the content through the pipeline as it goes from component to component.

The task or activity that a Generator must do is simple. It has to accept a request for content in the form of a URL, and it returns XML. That's it. Regardless of where the content comes from—a local file, a remote file, or a database, the same basic activity gets done. All the component manager knows about the Generator is that it can ask for a file at a given URL, and it can expect a file in return in XML format.

Now, if you are a programmer and you need to supply Cocoon with content, all you have to do is write an object that implements the interfaces specified by the Generator component and plug it into the pipeline. It's as simple as that. That's the beauty of components—you can mix and match them, plug them in, and take them out.

If you were to try the same thing using classes, you'd find your life to be extremely more complicated. You'd have to start with some abstract "Generator" class that had methods for requesting data and returning data as XML. Then you'd have to write a subclass for each kind of content you might want to retrieve and then override the associated methods. But what happens if you want to use a class as a generator that doesn't subclass your "Generator" class? Because REALbasic supports only single inheritance, each class can have only one superclass, and this imposes some limits on what you can do.

Interfaces, on the other, do not have the same limitations. A class can implement any number of interfaces, as long as it has the right methods.

Interfaces in REALbasic

To demonstrate how interfaces can be put to use in REALbasic, I'll define an Interface and then modify the **Properties** class to work with this interface. The interface will also be used extensively in the following section when I talk about how to create your own customized operator overloaders in REALbasic.

Before I get started, I should say that REALbasic uses interfaces extensively with Controls in a process called Control Binding. In the chapters that follow, I will revisit interfaces and show you how REALbasic uses them to make it easier to program your user interface.

If you recall, the **Properties** class implemented earlier in the chapter was designed to be able to accept both a string or a **FolderItem** in the `Constructor`. This is good because it makes it convenient to use, but it gets a little ugly when you get inside to look at the guts of the program. The reason I don't like it is that there are also two `ParsePropertyFile` methods, one for strings and the other for FolderItems, and each one is implemented differently. I don't like this because if I want to make a change in the future, I need to make sure that the change doesn't break either method. I'd rather have to check only one.

I will repeat them here to refresh your memory:

```
Protected Sub parsePropertyFile(myFile as string)
  Dim my_array(-1) as String
  Dim line as String

  my_array = myFile.split(Chr(13))

  For Each line In my_array
    Me.parseLine(line)
  Next
End Sub
```

This first version accepts a string in the parameter and then does two things. First, it splits the string into an `Array` and then it cycles through that `Array` to parse each individual line. Next, take a look at the **FolderItem** version:

```
Protected Sub parsePropertyFile(myPropertyFile as folderItem)
  Dim textInput as TextInputStream
  Dim astr as string

  If myPropertyFile Is Nil Then
       // An error has occurred
    Return
  End If
```

```
  If myPropertyFile.exists Then
    Me.file = myPropertyFile
    textInput = me.file.OpenAsTextFile
    Do
      astr=textInput.ReadLine()
      Me.parseLine(astr)
    Loop Until textInput.EOF

    textInput.Close
  Else
    // An error has occurred
  End If

End Sub
```

This, too, can be divided into two basic steps. First, the file is opened and a **TextInputStream** is returned, and second, a loop cycles through the **TextInputStream** parsing each line in the file. A much better design would be to have the "looping" take place in one method, and the other prep work take place elsewhere. To do that, we need to find a common way to loop through these two sources of data so that they can be handled by one method.

The obvious and easiest-to-implement solution would be to get a string from the **TextInputStream** and then pass that string to the `parsePropertyFile` method that accepts strings. Unfortunately, that doesn't help me explain how to use interfaces, so I'm not going to do it that way. Instead, I'm going to implement an interface called **ReadableByLine** and then pass objects that implement the **ReadableByLine** interface to a new, unified `parsePropertyFile` method.

There are advantages to doing it this way—some of which are immediate, whereas others are advantageous later on when you want to do something else with this class. In fact, this interface will come in even handier in the next section when I am writing custom operators for the **Properties** class.

To implement the class interface, you first have to decide which methods will compose the interface. To make it the most useful, you want to use ones that will be applicable to the most situations. In the previous implementation of the `parsePropertyFile` methods, the loop that iterated over the **TextInputStream** is most promising. (Because `Arrays` are not really classes, the techniques used to iterate over the `Array` aren't applicable.) The methods are used by the **TextInputStream** class, and the code in question is this:

```
Do
  astr=textInput.ReadLine()
  Me.parseLine(astr)
Loop Until textInput.EOF
```

If you look up **TextInputStream** in the language reference, you'll find that there are two implementations of `Readline`, and that `EOF` is a method. Their signatures follow:

```
TextInputStream.ReadLine() as String
TextInputStream.ReadLine(Enc as TextEncoding) as String
TextInputStream.EOF() as Boolean
```

There are other methods used by **TextInputStream**, but these are the ones I used previously. The language reference also says that the **TextInputStream** implements the **Readable** class interface. (The current language reference is in a state of flux as I write this, so I can only assume it still says this, but it's possible that it may not.) If REALbasic has already defined an interface, that's worth looking into. **Readable** is also implemented by **BinaryStream**, and the interface has two implementations of a `Read` method:

```
.Read(byteCount as Integer) as String
.Read(byteCount as Integer, Enc as Encoding) as String
```

Rather than reading text line by line, it reads a certain number of bytes of data, which isn't what we need. Therefore, inspired by the **Readable** title, I'll create a **ReadableByLine** interface that can be implemented by any class that wants you to be able to read through it line by line.

You create a class interface just like you create a class or a module. In the same project that contains the **Properties** class, click the **Add Class Interface** button on the **Project toolbar**, and you'll be presented with an screen much like the one used for modules and classes. The primary difference is that all your options are grayed out except **Add Method**; that's because interfaces contain only methods.

All you do when creating an interface is define the signature of the method—you do not implement it in any way. To implement the **ReadableByLine** interface, we'll implement the `ReadLine()` and `EOF()` methods like those used by **TextInputStream**. I'll also add another called `GetString()`, the usefulness of which will become evident later.

```
.ReadLine() as String
.ReadLine(Enc as TextEncoding) as String
.EOF() as Boolean
.GetString() as String
```

The next step is to go back to the **Properties** class and make the following modifications.

Implement the following `parsePropertiesFile` method:

```
Protected Sub parsePropertyFile(readable as ReadableByLine)
Dim aString as String

  If Not (Readable Is Nil) Then
    Me.readableSource = readable
    Do
      aString=readable.readLine()
      me.parseLine(aString)
```

```
    Loop Until readable.EOF
  End if
End Sub
```

Note that in the parameter, we refer to **ReadableByLine** exactly as if it were a class and readable was an instance of the class. Any variable that is declared an "instance" of **ReadableByLine** only has access to the methods of the interface, regardless of what the underlying class of the object is (which can be anything as long as it implements the **ReadableByLine** methods).

Next, the `Constructors` need to get updated. Instead of supplying a **FolderItem** or a string to the `parsePropertyFile` method, we want to send objects that implement the **ReadableByLine** interface.

The **TextInputStream** implements those methods. After all, that's where we got them from, but this also presents a problem. Because **TextInputStream** is a built-in class, I just can't go in and say that **TextInputStream** implements the **ReadableByLine** interface. REALbasic doesn't provide a mechanism for that.

The next possibility is to create a custom subclass of the **TextInputStream** and then say that it implements the interface, but that creates a problem, too. **TextInputStream** is one of a handful of built-in classes that can't be subclassed. Every time you use **TextInputStream**, you generate it from a **FolderItem** as a consequence of calling the `OpenAsTextFile` method. You can instantiate it yourself, or subclass it, but you can't assign any text to it (as far as I can tell), so it's only worthwhile when you get it from a **FolderItem**.

There's a third approach, which does mean you have to create a new class, but it's a good solution for these kinds of problems. All you do is create a class that's a wrapper of the **TextInputStream** class and say that it implements the **ReadableByLine** interface. This is a common tactic and it's sometimes called the "façade design pattern" in object-oriented circles because its one object provide a false front to another.

When you create the class, name it **ReadableTextInputStream** and add **ReadableByLine** to the interfaces label in the **Properties pane**.

ReadableTextInputStream Methods

The `Constructor` accepts a **TextInputStream** object and assigns it to the protected `InputStream` property.

```
Sub Constructor(tis as TextInputStream)
  InputStream = tis
End Sub
```

The other methods call the associated method from the **TextInputStream** class. They also append the value to the `SourceLines() Array`, which is primarily a convenience. It also provides a way to get at the data after you have iterated through all the "ReadLines" of **TextInputStream**.

```
Function ReadLine() As String
  Dim s as String
  s = InputStream.ReadLine
  Me.SourceLines.Append(s)
  Return s
End Function
Function ReadLine(Enc as TextEncoding) As String
  Dim s as String
  s = InputStream.ReadLine(Enc)
  Me.SourceLines.Append(s)
  Return s
End Function
Function EOF() As Boolean
  Return InputStream.EOF
End Function
Function getString() As String
  Return Join(SourceLines, EndOfLine.UNIX)
End Function
```

The final `getString()` method provides a way to get a string version of the **TextInputStream**.

ReadableTextInputStream Properties

```
Protected InputStream As TextInputStream
SourceLines(-1) As String
```

Now that the **TextInputStream** problem is solved, something similar needs to be done for strings.

ReadableString Methods

The string to be read is passed to the `Constructor`. Because I will be reading the string line by line, the first step is to "normalize" the character used to signify the end of a line. The `ReplaceLineEndings` function does that for me, and I have opted to standardize on Unix line endings, which are the ASCII character 13, otherwise known as the newline character.

Then I use the familiar `Split` function to turn the string into an `Array`. Finally, I will need to be able to track my position in the `Array` when using `ReadLine` so that I will know when I come to the end. I initialize those values in the `Constructor`. `LastPosition` refers to the last item in the `Array`. The `ReadPosition` is the current item in the `Array` being accessed, and it starts at zero. It will be incremented with each call to `ReadLine` until all lines have been read.

```
Sub Constructor(aSource as String)
  Source = ReplaceLineEndings(aSource, EndOfLine.UNIX)
  SourceLines = Split(Source, EndOfLine.UNIX)
  LastPosition = Ubound(SourceLines)
  ReadPosition = 0
End Sub
```

If you recall, the `Do…Loop` we used to iterate through the `ReadLines()` of the **TextInputStream** tested to see if it had reached the end of the file after each loop. The `EOF()` function provides for this functionality and will return true if the `ReadPosition` is larger than the `LastPosition`.

```
Function EOF() As Boolean
  If ReadPosition >  LastPosition Then
    Return True
  Else
    Return False
  End if
End Function
```

In the `ReadLine()` methods, the line is accessed using the `ReadPosition` prior to incrementing the `ReadPosition`. The line text is assigned to the `s` variable. After that, `ReadPosition` is incremented, then `s` is returned. `ReadPosition` is incremented after accessing the `Array` because the EOF test comes at the end of the loop. This is also the way we check to see if `ReadPosition` is greater than `LastPosition`, because after you have read the last line and incremented `ReadPosition`, it will be equal to `LastPosition + 1`.

```
Function ReadLine() As String
  Dim s as String
  s = SourceLines(ReadPosition)
  ReadPosition = ReadPosition + 1
  Return s
End Function
Function ReadLine(Enc as TextEncoding) As String
  // Not implemented;
  Return ReadLine()
End Function
```

Because I start with a string, the `getString()` method is only a matter of returning it. The `Source` property is assigned the passed string in the `Constructor`.

```
Function getString() As String
  Return Source
End Function
```

ReadableString Properties

```
ReadPosition As Integer
Encoding As TextEncoding
Source As String
SourceLines(-1) As String
LastPosition As Integer
```

Finally, the two constructors for the **Properties** class need to be modified as follows:

```
Sub Properties.Constructor(aPropertyFile as FolderItem)
  Dim readable as ReadableTextInputStream
  readable = new _ ReadableTextInputStream(myPropertyFile.OpenAsTextFile)
  parsePropertyFile readable
End Sub


Sub Properties.Constructor(myPropertyString as String)
  Dim readable As ReadableString

  readable = New ReadableString(myPropertyString)
  parsePropertyFile readable

End Sub
```

For good measure, I create a `Constructor` that accepts a **ReadableByLine** implementing object directly.

```
Sub Properties.Constructor(readable as ReadableByLine)
  parsePropertyFile readable

End Sub
```

This may seem like a lot of work to implement an interface, but it shows you a realistic example of how one might be used. Now that this groundwork is laid, it is easy to come up with additional sources of "properties." For example, you could store them in a database. A database cursor would adapt to the **ReadableByLine** interface easily, and it would take very little work to add that as a new source for key/value pairs.

In the next section it will come up again, and at that point you will be able to see how the interface provides a lot more flexibility as the class you are working on grows in functionality.

Custom Operator Overloading

The fact that you can use the + with integers and strings is an example of operator overloading that you have already encountered. Customizable operator overloading is a relatively new feature in REALbasic, and I think it's extremely powerful and fun to use. It creates a lot of flexibility for the developer, and it is a good thing. A very good thing.

To refresh your memory, an operator works like a function except that instead of arguments being passed to it, operators work with operands. There's usually an operand to the left and the right of the operator, and REALbasic determines which overloaded version of the operator to use, based upon the operand types. That's how REALbasic knows to treat 1+1 different from "One" + "Two".

Customized operator overloading is accomplished by implementing one or more of the following methods in your class. I'm going to use the **Properties** class as our operator overloading guinea pig and add a lot of new features that will allow you to do all kinds of lovely things with the class, including adding together two **Properties** objects, testing for equality between two different instances, coercing a **Properties** object into a string and so on.

The operators will also work with the **ReadableByLine** interface so that in addition to being able to add two **Properties** objects together, you can add an object that implements **ReadableByLine** to a **Properties** object, and so on.

Special Operators

Function Operator_Lookup(aMemberName as String) as Variant

`Operator_Lookup()` counts among my favorite operators to overload. I don't know why. I just think it's fun because it effectively allows you to define objects in a much more dynamic way by adding members to the object at runtime. You're not really adding members, but it gives the appearance of doing so. The `Operator_Lookup()` function overloads the "." operator, which is used when accessing a method or property of an object.

Here is an example from the **Properties** class:

```
Function Operator_Lookup(aKey as String) as String
 Return me.get(aKey)
End Function
```

By implementing this function, I can now access any key in the **Properties** class as if it were a public property or method of the **Properties** class. This is how it would work in practice:

```
Dim prop as Properties
Dim s as String
prop = New Properties()
prop.set("FirstKey") = "FirstValue"
prop.set("SecondKey") = "SecondValue"
s = prop.FirstKey // s equals "FirstValue"
```

As you can see in this example, I am able to access the "FirstKey" key using dot (".") notation. When you try to access a member that REALbasic doesn't recognize, it first turns to see if an `Operator_Lookup()` method has been implemented. If one has been implemented, it passes the member name to the `Operator_Lookup()` method and lets that function handle it any way it likes.

In the **Properties** example, it calls the `get()` function, which returns a value for the key or, if the key is not in the **Properties** object, returns an empty string. Basically, I have a **Dictionary** whose keys are accessible as if they were members of the object. If you are familiar with Python, this may sound familiar because Python objects *are* Dictionaries, and you can access all Python objects both through Python's **Dictionary**

interface or as members of the Python class. I should say, however, that I would not recommend using a **Dictionary** subclass with the `Operator_Lookup()` overloaded method as a general replacement for custom objects because there is a lot of overhead when instantiating **Dictionaries**. It's a perfect solution for situations like the **Properties** class, which needs to be a **Dictionary** subclass for a lot of reasons.

Function Operator_Compare(rightSideOperand as Variant) as Double

The `Operator_Compare()` function overloads the "=" operator. It tests to see if one operand is equal to the other one. The developer gets to decide what constitutes "equal to."

This is illustrated in the following example, which was implemented in the **Properties** class. The **Properties** object is the operand on the left side of the expression, and the right side of the expression is the argument passed using the "readable as **ReadableByLine**" parameter.

One thing you should notice right away is that the right side operand isn't of the same class as the left side operand; that's okay because you get to decide how this works. I find this approach useful because I may want to compare a file with an instantiated **Properties** object to see if there are any differences between the two. It's an extra step if I have to open the file, instantiate a **Properties** object, and then compare it with the original. This approach lets me compare the values represented by two distinct but related objects.

In practice, you can return any numeric data type—an integer, a short, or a double, but the answer is evaluated much like the `StrComp()` function. In other words, the number 0 means that both operands are equal to each other. A number greater than 0 means the left-side operand is greater than the right-side operand, and a number less than 0 means the left operand is less than the right.

```
Function Operator_Compare(readable as ReadableByLine) as Integer
  // mode = lexicographic
  dim newReadable as ReadableString

  Return StrComp(me.getString, readable.getString, 1 )
Function
```

This example actually uses `StrComp()` to make the comparison between the two related objects. It compares the string provided by the **Properties** object with that provided by the **ReadableByLine** argument. The function returns the results of the `StrComp()` function.

Function Operator_Convert() as Variant

There is an `Operator_Convert()` function and an `Operator_Convert` subroutine, both of which work with the assignment operator, but in slightly different ways. Consider the following implementation:

```
Function Operator_Convert() as String
 Return Me.getString()
End Function
```

This is how it is used:

```
Dim prop as Properties
Dim s as String
prop = New Properties("a=first" + chr(13) + "b=second")
s = prop // s equals the string '"a=first" + chr(13) + "b=second"'
```

In this example, the `prop` object is assigned to a variable with a string data type. The `Operator_Convert()` function coerces the `prop` object into a string. This is functionally equivalent to called the `prop.getString()` method.

Sub Operator_Convert(rightSideOperand as Variant)

The subroutine version handles assignment as well. The difference is that whereas the function assigns the value of the object executing the method to some other variable, the subroutine assigns the value of some other variable to the object executing the method. The argument passed in the "`readable as ReadableByLine`" parameter is the object whose value is being assigned to the calling object.

```
Sub Operator_Convert(readable as ReadableByLine)
  Dim tmpReadable as ReadableByLine

  If Not (me.readableSource is Nil) Then

    Me.Clear
    tmpReadable = Me.readableSource
    Me.readableSource = Nil
    Try
      Me.parsePropertyFile(readable)
    Catch
      // If new file fails, restore old version
      me.readableSource = tmpReadable
      me.parsePropertyFile(me.readableSource)
    End

  Else
    // No pre-existing data; so read new data
    Me.parsePropertyFile(readable)
  End If

End Function
```

In this example, I assign the value of a **ReadableByLine** object to that of a **Properties** object. This means that I can use the assignment operator ("=") to instantiate a new **Properties** object.

```
Dim prop1, prop2 as Properties
Dim s1, s2 as String
Dim r as ReadableString
s1 = "a=first" + chr(13) + "b=second"
s2 = "c=third" + chr(13) + "d=fourth"
prop1 = New Properties(s1)
```

```
r = New ReadableString(s2)
prop2=r
```

In this example, `prop1` is instantiated using the `New` operator, whereas `prop2` is instantiated using simple assignment. In a sense, this functionality gives the **Properties** class something of the flavor of a data type.

Addition and Subtraction

You can also overload the "+" and "-" operators, enabling you to add objects to each other or subtract objects from each other. You have already seen an overloaded version of "+" that is used with strings. When used with strings, the "+" operator concatenates the strings. Because our **Properties** class deals with strings, you can create similar functionality for your objects. In this case, you can add two **Properties** objects together and this will concatenate the strings associated with each object and return a new object generated from the concatenated string.

Listing 3.1 **Function Operator_Add(rightSideOperand as Variant) as Variant**

```
Function Operator_Add(readable as ReadableByLine) as Properties
  // Since parsePropertyFile does not reset
  //the data when called, this adds more values
  //to the Property object
  Dim s as String
  Dim newProp as Properties

  s = Self.getString + EndOfLine.UNIX + readable.getString

  newProp = New Properties(s)

  return newProp
End Function
```

When you implement the `Operator_AddRight()` version, you do the same thing, but switch the order in which the strings are concatenated. In the following example, note that the string concatenation happens in the opposite order from the previous example.

Listing 3.2 **Operator_AddRight(leftSideOperand as Variant) as Variant**

```
Function Operator_AddRight(readable as ReadableByLine) as Properties
  Dim prop as Properties
  Dim s as String

  s = readable.getString+ EndOfLine.UNIX +  me.getString
  prop = New Properties(s)

  Return prop
End Function
```

Operator_Subtract(rightSideOperand as Variant) as Variant, Operator_SubtractRight (leftSideOperand as Variant) as Variant

The subtraction operators work in a predictable way, just like the add operators, except that the values are being subtracted. Although I did not implement this method in the **Properties** class, I can imagine it working something like this: Any key/value pair that exists in the left-side operand is deleted from that object if it also exists in the right-side operand. If I wanted to delete a group of key/value pairs from a **Properties** object, I could use a **Properties** object that contained these key/value pairs as one of the operands.

Boolean

Operator_And(rightSideOperand as Variant) as Boolean, Operator_AndRight (leftSideOperand as Variant) as Boolean, Operator_Or(rightSideOperand as Variant) as Boolean, Operator_OrRight(leftSideOperand as Variant) as Boolean

For the `And` operator to return `True`, both operands must evaluate to `True` as well. Here's a scenario where this might make sense: the **Properties** class (and many other classes, too) can be instantiated, but not populated by any values. If you tested to see if the operands were equal to `Nil`, you would be told that they are not `Nil`. However, it might be useful to be able to distinguish whether there are any values at all in the object. You can use the `And` operator to do this. Within the `Operator_And` method, test to see if each operand has a value. If both do, return `True`. If one or more does not, return `False`.

More Operator Overloading

There are several more operators that can be overloaded, but they do not have an application with the **Properties** class. They work much like the ones I have covered here, except that they overload negation, the Not operator, and the multiplication operators such as *, /, `Mod`, and so on. They are

```
Operator_Negate() as Variant
Operator_Not() as Boolean
 Operator_Modulo(rightSideOperand as Variant) as Variant
Operator_ModuloRight(leftSideOperand as Variant) as Variant
Operator_Multiply(rightSideOperand as Variant) as Variant
Operator_MultiplyRight(leftSideOperand as Variant) as Variant
Operator_Power(rightSideOperand as Variant) as Variant
Operator_PowerRight(leftSideOperand as Variant) as Variant
Operator_Divide(rightSideOperand as Variant) as Variant
Operator_DivideRight(leftSideOperand as Variant) as Variant
Operator_IntegerDivide(rightSideOperand as Variant) as Variant
Operator_IntegerDivideRight(leftSideOperand as Variant) as Variant
```

Extends

Finally, there is an additional way of extending a class that does not involve subclassing or interfaces. REALbasic allows you to implement methods in Modules and use those methods to extend or add functionality to other classes. Some classes, like the **TextInputStream** class I discussed earlier, can't really be effectively subclassed in REALbasic, and this is where using the `Extends` keyword can come in handy because it gives you an opportunity to add methods to **TextInputStream** without subclassing it.

I'll start with an example, which should make things clearer.

Unix Epoch

Unix uses a different epoch to measure time than REALbasic does. Instead of January 1, 1904 GMT, Unix measures time since January 1, 1970 GMT. The difference between these two values is 2,082,848,400 seconds. One useful addition to the Date object would be a method that converted REALbasic's measurement of time to the way that Unix measures time.

It's possible to subclass **Date**, but that only provides a partial solution because it doesn't help you with all the other instances of data objects that you encounter while programming. For example, the `FolderItem` class has **Date** properties for the file creation date and the date the file was last modified. A subclass won't help you there.

It is for this kind of problem in particular that `Extends` is made. `Extends` allows you to *extend* a class with a new method without having to subclass it. The method that will be used to extend the class must be implemented in a module, and the first parameter must be preceded by the `Extends` keyword, followed by a parameter of the class that is being extended.

The following example adds a method `UnixEpoch` that can be called as a member of the `Date` class. It returns the date as the number of seconds since January 1, 1970, rather than January 1, 1904.

```
Protected Function UnixEpoch(Extends d as Date) As Double
   Return d.TotalSeconds - 2082848400
End Function
```

Conclusion

In addition to being an object-oriented programming language, one of the most important distinguishing characteristics of REALbasic is that it is a cross-platform application development environment. You can compile applications for Windows, Macintosh, and Linux. In my experience, REALbasic is the easiest language to use for this kind of development, but it is not without its own idiosyncrasies and, despite their similarities, the three platforms vary in fundamental ways that a developer needs to be aware of. In the next chapter, I write about the cross-platform features of REALbasic and how the differences among the platforms will impact your development.

This page intentionally left blank