In Chapter 4, you learned that the .NET Framework solves the circular reference problem by keeping track of all the objects used by your application and using garbage collection to free any objects no longer referenced.

In Chapter 5, you learned that every piece of data down to the simplest integer variable is an object.

This means that every item of data you use in your application is subject to the memory management rules of .NET.

Almost....

Value Objects and Reference Objects

The designers of the .NET Framework knew that it would be terribly wasteful to store every item of data down to numeric variables on the heap. Even though the .NET heap is remarkably efficient, it still demands an allocation step that is slower than a simple stack allocation. And there is also the cost of tracking the object and collecting it after it is freed.

Nevertheless, they wanted the ease of use and consistency that comes from deriving every variable from a base Object type.

To resolve this conflict, they ended up defining two types of objects: Reference objects and Value objects. ¹

A Reference object is similar to the COM objects you are familiar with from VB6. When you assign one Reference object to another, you do not make a copy of the data—instead, you simply obtain a new reference to the existing data. Reference objects are always allocated on the heap. Reference objects can inherit from other objects and other objects can, in turn, inherit from them.

¹ Just to be clear, the terms Reference type and Reference object mean the same thing, as does Value type and Value object in this context.
Value Objects

Value objects are allocated in the application's data segment or on the stack—not on the heap. Value types in VB .NET are created in much the same way as you created user-defined types in VB6 except that you use the new Structure keyword. Here's an example of a simple structure defined in the ValueType sample program:

```vbnet
Public Structure mystruct
    Public AString As String
    Public Sub SetString(ByVal newstring As String)
        AString = newstring
    End Sub
    Public Sub New(ByVal InitialString As String)
        AString = InitialString
    End Sub
End Structure
```

There are some interesting points to consider about this declaration.

First, the structure contains a string. Under VB6, you could define fixed-length strings that would be stored within the structure. Under VB .NET, the String data type is a Reference type. So, here you have a Value type that contains a single variable, which is a Reference type. What then, is the benefit of using a Value type in this case?

There is none. It's the wrong approach.

Value types are best used to create small objects that are mostly based on numeric data types, for example, complex numbers or location coordinates.

Unlike VB6 user-defined types, VB .NET structures can have methods and properties. They can also have constructors: methods used to initialize the structure. There's just one catch, however; constructors you define must have at least one parameter. The default constructor is used by the CLR to initialize the structure to all zeros.

The Button1_Click method in the ValueType sample project appears as follows:

```vbnet
Private Sub Button1_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles button1.Click
    Dim a As mystruct
    a.SetString("Hello")
    Dim b As mystruct = a
```

2. You'll read a lot more about strings in Chapters 8 and 9—they work very differently from VB6. For now, you just need to know that a string inside a structure is stored as a pointer to a location on the heap. The string data is not stored in the structure itself.