Inheritance

Often, types in a program share the same characteristics. For example, a program may contain types that represent a customer and an employee.

```vbnet
Class Customer
    Public Name As String
    Public Address As String
    Public City As String
    Public State As String
    Public ZIP As String
    Public CustomerID As Integer
End Class

Class Employee
    Public Name As String
    Public Address As String
    Public City As String
    Public State As String
    Public ZIP As String
    Public Salary As Integer
End Class
```

In this situation, both the Customer and Employee classes contain a number of identical fields. This is because the two classes each describe a person, and a person has certain characteristics, such as a name and address, that exist independent of whether or not they are a customer or an employee.
This commonality between the Customer type and the Employee type can be expressed through inheritance. Instead of repeating the same information in both types, you can create a class called Person that contains the common characteristics of a person.

```vbnet
Class Person
    Public Name As String
    Public Address As String
    Public City As String
    Public State As String
    Public ZIP As String
End Class
```

The class Person represents all the characteristics of a person that exist independent of whether the person is a customer or an employee. Once the Person class is defined, the Customer and Employee classes can inherit all the members of the Person class. This means that the classes have to define only the members that are unique to each class.

```vbnet
Class Customer
    Inherits Person
    Public CustomerID As Integer
End Class

Class Employee
    Inherits Person
    Public Salary As Integer
End Class
```

When one class inherits members from another class, the inheriting class derives from the other type. The type being derived from is called the base type. A type inherits all the members that the base type defines, including methods and events. So the Employee and Customer classes still have fields named Name, Address, City, State, ZIP, and Phone, even though they don’t explicitly declare them, because they inherit them from Person. For example, the classes can be used as follows.

```vbnet
Module Test
    Sub Main()
        Dim c As Customer = New Customer()
        c.Name = "John Smith"
    End Sub
End Module
```
A class that derives from another class can in turn be derived from by another class. For example, Employee can be further specialized by classes such as Manager and Programmer.

```vbnet
Class Programmer
    Inherits Employee

    Public Project As String
End Class

Class Manager
    Inherits Employee

    Public Programmers() As Programmer
End Class
```

In this example, the Programmer class contains the members defined in its immediate base class, Employee, as well the members defined in Employee's base class, Person. Related types can be viewed as a hierarchy with a tree structure, as in Figure 13-1.

Obviously, a type cannot directly or indirectly inherit from itself. Also, notice that the type Object is at the top of the inheritance hierarchy. If a class does not explicitly inherit from another class, it inherits from Object by default. Thus, Object is always the common root of all inheritance hierarchies. Also notice that in this type hierarchy, the most general types are at the top of the tree. As you move down the hierarchy, the classes at each level become more specialized and specific. Inheritance is a very powerful way of expressing the relationships between types.

Advanced

Visual Basic .NET supports only single inheritance, which means that a class can derive from only one base type.
An important thing to keep in mind about inheritance and accessibility is that a derived class does not have access to its base classes’ Private members. Private members can be accessed only by the immediate type in which they’re defined. Protected members, however, can be accessed within an inheritance hierarchy. The Protected access level restricts access to a member to only the class itself, but it extends access to all derived classes as well. For example:

```vbnet
Class User
    Private SSN As String
    Protected Password As String
End Class

Class Guest
    Inherits User

    Sub New()
        ' Error: SSN is private to User
        SSN = "123-45-7890"
    End Sub

    ' OK: Password is protected and can be accessed
    Password = "password"
End Class
```

The class Guest can access the Password field inherited from its base class because it is Protected. However, it cannot access the SSN field, because it is Private.
When a class accesses a Protected member, the access must take place through an instance of that class or a more derived class. It cannot take place through a base class. For example, the following code is incorrect.

```vbnet
Class User
    Protected Name As String
    Private SSN As String
    Protected Password As String
End Class

Class Guest
    Inherits User
    Shared Sub ChangeName(ByVal u As User, ByVal Name As String)
        ' Error: Access to Name in User cannot go through base class User
        u.Name = Name
    End Sub
End Class
```

This rule may seem strange, but it is necessary to prevent unexpected access to Protected members. Without the rule, it would be possible to gain access to a Protected member of another type simply by deriving from a common base class.

```vbnet
Class User
    Protected Name As String
    Private SSN As String
    Protected Password As String
End Class

Class Administrator
    Inherits User
End Class

Class Guest
    Inherits User
    Public Sub PrintAdministratorPassword(ByVal u As User)
        ' Error: Access to Password in User cannot go through base class User
        Console.WriteLine(u.Password)
    End Sub
End Class
```
In this example, Guest cannot access Administrator’s protected field Password—it can only access the Password field of instances of Guest.

**NOTE**

Protected and Friend access levels can also be combined—the Protected Friend access level is the union of the two access levels.

### Conversions

When a class derives from another class, it automatically inherits all the members of the base class. As a result, a derived class can always be safely converted to one of its base classes. For example:

```vbs
Class Person
    Public Name As String
    Public Address As String
    Public City As String
    Public State As String
    Public ZIP As String
End Class

Class Employee
    Inherits Person

    Public Salary As Integer
End Class

Module Test
    Sub Main()
        Dim p As Person = New Employee()
        p.Name = "John Doe"
    End Sub
End Module
```

In this example, the Framework can allow an instance of Customer to be assigned to a variable of type Person because it knows that a Customer is also a Person. Thus, a Customer can be treated like a Person, and the fields that Customer inherits from Person can be changed. If the preceding example had tried to access fields that were specific to Customer or Employee, however, an error would be given because when an instance is viewed as as a Person, only the members defined by Person can be used, as the following example illustrates.
Module Test
Sub Main()
    Dim p As Person
    p = New Customer()
    ' Error: CustomerID is not a member of Person
    p.CustomerID = 10
    p = New Employee()
    ' Error: Salary is not a member of Person
    p.Salary = 34923.23
End Sub
End Module

Conversely, an instance of a base class can be converted to a derived class, but the conversion is not always guaranteed to succeed. A variable typed as `Person` could contain an instance of the `Employee` class, but it could also contain an instance of some other type.

Module Test
Sub Main()
    Dim p As Person = New Employee()
    ' Error: Can't convert Employee to Customer
    Dim c As Customer = CType(p, Customer)
End Sub
End Module

The `Customer` class also inherits from `Person`, so it can be converted to `Person`. However, the instance stored in the variable is still a `Customer`; as such, it cannot be treated as an instance of the `Employee` class. In this case, the Framework will throw a `System.InvalidCastException` exception at runtime when the conversion is executed.

The important principle to keep in mind is that when you create an instance of a class, it always stays that type, no matter what it is converted to. A `Customer` class converted to `Person` is still a `Customer`, even if the additional fields that `Customer` adds to the `Person` class are not visible. The power of inheritance is that it allows code to be written that works on the most general type in a hierarchy, which means that code can be written very broadly. For example, a method that takes a `Person` and prints the name and address of that `Person` can take a `Customer` or an `Employee`, instead of having to write separate methods for `Customer` and `Employee`. 
Module Test
Sub PrintAddress(ByVal p As Person)
    Console.WriteLine(p.Name)
    Console.WriteLine(p.Address)
End Sub

Sub Main()
    Dim c As Customer
    Dim e As Employee

    PrintAddress(c)
    PrintAddress(e)
End Sub
End Module

Array Covariance
Inheritance conversions extend to arrays as well. In general, an array of a particular type cannot be converted to any other type, because the array storage is allocated based on the type of the array. For example, it is not possible to covert a one-dimensional array of Integer to a one-dimensional array of Long, because Integer and Long do not have the same size. Thus, an array of ten Integer values could not hold ten Long values within the same space. However, because classes are reference types, the size of an array that holds ten Customer instances is the same size as an array that holds ten Employee instances. Thus, an array of a reference type may be converted to an array of another reference type, provided that the element types themselves convert to one another. For example:

Module Test
Sub Main()
    Dim Customers(9) As Customer
    Dim People() As Person

    For Index As Integer = 0 To 9
        Customers(Index) = New Customer()
    Next Index

    People = Customers

    For Index As Integer = 0 To 9
        People(Index).Name = "John Doe"
    Next Index
End Sub
End Module
The one-dimensional array of Customer can be converted to a one-dimensional array of Person because a Customer can be converted to a Person. This conversion behavior of arrays is called covariance. Covariance is useful in the same way that inheritance is.

Module Test

    Sub SetCityState(ByVal People() As Person, ByVal City As String, ByVal State As String)
        For Each Person As Person In People
            Person.City = City
            Person.State = State
            Next Person
        End Sub

    Sub Main()
        Dim Employees(9) As Employee
        ...

        SetCityState(Employees, "Akron", "OH")
    End Sub
    End Module

In this example, the base class Person provides a method that will set the City and State fields for an array of Person instances. The Main method can pass an array of Employee instances to the method because an array of Employee can be converted to an array of Person. One important thing to note, though, is that even though the array has been converted to an array of Person, it still can only hold Employee instances. Thus, an attempt to assign any other type into the array will cause a System.InvalidCastException exception. For example:

Module Test

    Sub FillArray(ByVal People() As Person)
        For Index As Integer = 0 To People.Length -1
            People(Index) = New Person()
        Next Index
    End Sub

    Sub Main()
        Dim Employees(9) As Employee
        FillArray(Employees)
    End Sub
    End Module
This example will throw an exception because the array being passed in to FillArray is an array of Employee, not Person, so only Employee instances can be stored in the array.

The .NET Framework Type Hierarchy

As previously discussed, if a class does not have an explicitly stated base class, its base class is Object. This means that all classes ultimately derive from Object. Indeed, all types in the Framework type system—even the fundamental types, structures, enumerations, delegates, and arrays—derive from Object through special base classes that cannot otherwise be inherited from (see Figure 13-2). Structures and the predefined types derive from the type System.ValueType. Enumerations derive from the type System.Enum. Delegates derive from the type System.Delegate. Arrays derive from the type System.Array. And all these types inherit from Object.

What this means is that any type in the type system can be converted to Object. This makes Object a universal type. A method that takes Object can accept any type, while a field typed as Object can store any type.

Compatibility

The Object type combines the capabilities that used to be split between the Object type and the Variant type in previous versions of Visual Basic.

One interesting aspect of this design is that Object is a reference type. This raises the question: How can structures and fundamental types like Integer and Double, all of which are value types, inherit from a reference type? More specifically, how can a value type like Integer be converted to its base class, Object, when Object is a reference type? The Framework solves this conundrum through a process called boxing. When a value type is converted to Object, the Framework copies the value stored in the value type to the heap and returns a reference to the value. This process is called boxing the value type (see Figure 13-3). The reference can then be used to access the boxed value on the heap.
When a reference to a boxed value type is converted back to the value type, the Framework copies the value stored on the heap back into the variable. This process is called *unboxing* a boxed value type (see Figure 13-4).

The following code shows an example of boxing and unboxing an `Integer`.

```vbnet
Module Test
    Sub Main()
        Dim o As Object
        Dim i As Integer

        i = 5
        o = i ' Copies the value to the heap
    End Sub
End Module
```
In general, a boxed value type can only be unboxed back to its specific type. For example, the following code will throw an exception because a boxed value of structure X cannot be unboxed into a variable typed as structure Y.

```
Structure X
    Public Value As Integer
End Structure
```

**DirectCast**

In general, a boxed value type can only be unboxed back to its specific type. For example, the following code will throw an exception because a boxed value of structure X cannot be unboxed into a variable typed as structure Y.

```
Dim i As Integer = 5
Dim o As Object
o = i

Console.WriteLine(o)
i = CInt(o)  ' Copies the value back from the heap
Console.WriteLine(i)
End Sub
End Module
```
Structure Y
   Public Value As Integer
End Structure

Module Test
   Sub Main()
      Dim o As Object
      Dim x As X
      Dim y As Y

      ' Box the value of x
      o = x

      ' Error: Cannot unbox a value of type X into a variable of type Y
      y = CType(o, Y)
   End Sub
End Module

---

Figure 13-4: Unboxing an Integer Value
The exceptions to this rule are the fundamental types: It is possible to unbox a boxed fundamental type into any other fundamental type that it has a conversion to. For example, the following code is valid because the Integer value in \( x \) can be unboxed into the Long variable \( y \).

```vbnet
Module Test
    Sub Main()
        Dim o As Object
        Dim x As Integer = 5
        Dim y As Long

        ' Box the value of x
        o = x

        ' OK: Can unbox the Integer value into a Long variable
        y = CLong(o)
    End Sub
End Module
```

This can be very useful, but it comes at a price. When any value is converted from Object, the program must check at runtime to see whether the value is a boxed fundamental type so that it can apply the special unboxing behavior described in the previous paragraph. These checks add a little bit of overhead to the conversion, which normally is not significant. However, if it is known ahead of time that the conversion type exactly matches the boxed type, there may be some advantage to avoiding the overhead. For example, when lots of conversions are being performed, the overhead could become significant.

The `DirectCast` operator works just like the `CType` operator, except that it does not allow unboxing a boxed value type into anything but its original type—even if the boxed value type is a fundamental type. The advantage, though, is that the overhead of checking for the fundamental types is avoided. For example, in the following code the second conversion will be more efficient than the first.

```vbnet
Module Test
    Sub Main()
        Dim o As Object
        Dim x As Integer = 5
        Dim y, z As Integer
        Dim a As Long
```
' Box the value of x
  o = x

' Normal conversion
  y = CInt(o)

' More efficient conversion
  z = DirectCast(o, Integer)

' Error: Types do not match
  a = DirectCast(o, Long)
End Sub
End Module

The last conversion emphasizes the fact that DirectCast can only unbox boxed values to their original type. So, unlike in the previous example, o cannot be unboxed into a Long variable.

**Style**

Unless code is particularly performance sensitive and doing a lot of unboxing, CType is more general than DirectCast and is preferred.

**Overriding**

Defining an inheritance hierarchy is all about defining the types in a system from most general to most specific. With inheritance, however, a derived type can only add new members to those it inherits from its base type. Sometimes, though, a derived type may want to change the behavior of members that it inherits from a base type. For example, the base class Person may define a Print method that prints information about the class.

Class Person
  Public Name As String
  Public Address As String
  Public City As String
  Public State As String
  Public ZIP As String

  Sub Print()
    Console.WriteLine(Name)
In this example, though, calling the method Employee.Print will only print the name and address of an employee, not the employee’s salary. There is no way, using inheritance, to change the inherited implementation of Person.Print.

Changing the implementation of derived methods is possible, however, through overriding. A base class can declare that a particular method or methods are Overridable, which means that a derived class can replace the implementation that the base class provides. For example:

Class Person
    Public Name As String
    Public Address As String
    Public City As String
    Public State As String
    Public ZIP As String

    Overridable Sub Print()
        Console.WriteLine(Name)
        Console.WriteLine(Address)
        Console.WriteLine(City & " , " & State & " " & ZIP)
    End Sub
End Class

Class Employee
    Inherits Person

    Public Salary As Integer
End Class

In this example, though, calling the method Employee.Print will only print the name and address of an employee, not the employee’s salary. There is no way, using inheritance, to change the inherited implementation of Person.Print.

Changing the implementation of derived methods is possible, however, through overriding. A base class can declare that a particular method or methods are Overridable, which means that a derived class can replace the implementation that the base class provides. For example:

Class Person
    Public Name As String
    Public Address As String
    Public City As String
    Public State As String
    Public ZIP As String

    Overridable Sub Print()
        Console.WriteLine(Name)
        Console.WriteLine(Address)
        Console.WriteLine(City & " , " & State & " " & ZIP)
    End Sub
End Class

Class Employee
    Inherits Person

    Overrides Sub Print()
        Console.WriteLine(Name)
        Console.WriteLine(Address)
        Console.WriteLine(City & " , " & State & " " & ZIP)
        Console.WriteLine("Salary = " & Salary)
    End Sub

    Public Salary As Integer
End Class
In this example, the Employee class overrides Person’s implementation of the Print method with its own version of the Print method that prints the salary as well as the employee’s name and address.

One interesting thing to note is that when a type overrides a base type’s member, that override applies to all instances of the type, no matter what their stated type is. In other words, Employee’s implementation of Print is the one that will be called on an Employee instance, even if it is typed as a Person. The following example:

```vbnet
Module Test
    Sub Main()
        Dim e As Employee = New Employee()
        Dim p As Person
        e.Name = "John Doe"
        e.Address = "123 Main St."
        e.City = "Toledo"
        e.State = "OH"
        e.ZIP = "48312"
        e.Salary = 43912
        p = e
        p.Print()
    End Sub
End Module
```

will print this:

John Doe
123 Main St.
Toledo, OH 48312
Salary = 43912

even though the type of the variable that Print is being called on is Person instead of Employee.

Properties can also be overridden. The following is an example of overriding a property.

```vbnet
Class Order
    Public Cost As Double
    Public Quantity As Integer

    Public Overridable ReadOnly Property Total() As Double
        Get
            Return Cost * Quantity
        End Get
End Class
```
When you are overriding a read-write property, both the Get and the Set accessors must be provided, even if you wish to override only one of them. Only methods and properties can be overridden, and they can be overridden only if they specify the Overrides keyword in their declaration. This is to prevent programmers from accidentally letting derived classes override methods that they did not intend to be overridden.

Only accessible members from a base type can be overridden. Thus, a Friend Overridable method cannot be overridden outside the assembly it is declared in. (It is not valid to declare a method Private Overridable, because no derived type could override such a method.) When you are overriding a method, the access of the overriding method must be the same as the method being overridden.

NOTE
When you are overriding a Protected Friend method from a derived class that is not in the same assembly as the class, the overriding method specifies just Protected instead of Protected Friend.

Sometimes it is desirable to override a method but prevent any further derived classes from overriding the method. Adding the NotOverridable keyword to a method that is overriding another method prevents any further derived classes from overriding the method.
MyBase and MyClass

In the example in the previous section, Employee.Print had to supply the entire implementation of Person.Print so that the name and address would still be printed—if it hadn’t done that, only the salary would have been printed. In this situation, the keyword MyBase can be used to call the methods of the base class, allowing Employee.Print to call Person.Print. Calling methods off of MyBase calls the base class’s implementation of a method, even if the derived class has overridden it. So the example could be rewritten as follows.

```csharp
Class Person
    Public Name As String
    Public Address As String
    Public City As String
    Public State As String
    Public ZIP As String

    Overridable Sub Print()
        Console.WriteLine(Name)
        Console.WriteLine(Address)
        Console.WriteLine(City & " , " & State & " " & ZIP)
    End Sub
End Class

Class Employee
    Inherits Person

    Overrides Sub Print()
        MyBase.Print()
        Console.WriteLine("Salary = " & Salary)
    End Sub

    Public Salary As Integer
End Class
```

The result would be the same: First, Person.Print would be called to print the name and address, and then Employee.Print would print the salary.

Sometimes it is desirable to call the particular implementation of a method that your class provides, regardless of whether the instance might be of a type that overrides it. Qualifying the method call with the keyword
MyClass will always call the containing class’s implementation of a method, ignoring any further implementation. The following example:

```vbnet
Class Person
    Public Name As String
    Public Address As String
    Public City As String
    Public State As String
    Public ZIP As String
    Sub CallPrint()
        Print()
    End Sub
    Sub CallMyClassPrint()
        MyClass.Print()
    End Sub
    Overridable Sub Print()
        Console.WriteLine(Name)
        Console.WriteLine(Address)
        Console.WriteLine(City & ", " & State & " " & ZIP)
    End Sub
End Class

Class Employee
    Inherits Person
    Overrides Sub Print()
        Console.WriteLine(Name)
        Console.WriteLine(Address)
        Console.WriteLine(City & ", " & State & " " & ZIP)
        Console.WriteLine("Salary = " & Salary)
    End Sub
    Public Salary As Integer
End Class

Module Test
    Sub Main()
        Dim e As Employee = New Employee()
        Dim p As Person

        e.Name = "John Doe"
        e.Address = "123 Main St."
        e.City = "Toledo"
        e.State = "OH"
        e.ZIP = "48312"
        e.Salary = 43912

        p = e
    End Sub
```

INHERITANCE
When the method `Person.CallPrint` calls the overridable `Print` method, what `Print` method ends up getting called depends on the actual type instance at runtime. Since the instance in this case is actually an `Employee`, `Person.CallPrint` ends up calling `Employee.Print`. However, because `CallMyClassPrint` qualifies the call to `Print` with `MyClass`, it always calls `Person.Print`, even if the instance is a more derived class.

**Abstract Classes and Methods**

In the examples we’ve been using so far in this chapter, `Person`, `Employee`, and `Customer` have all been classes that can be created using the `New` operator. However, there may be situations where a base class should never be created—perhaps there should only be instances of the `Employee` type and `Customer` type and never an instance of the `Person` type. It’s possible just to add a comment saying `Person` should never be created, or `Person` might have a `Private` constructor to make it impossible to create. However, `Person` can also be designated as an *abstract* type. An abstract type is the same as a regular (or *concrete*) type in all respects except for one: An abstract
Abstract classes are special in that they can also define *abstract methods*. Abstract methods are overridable methods that are declared with the MustOverride keyword and provide no implementation. A class that inherits from a class with abstract methods must provide an implementation for the abstract methods or must be abstract itself. For example, the
Person class could define an abstract PrintName method that each derived class has to implement to display the person’s name correctly.

```
MustInherit Class Person
    Public Name As String
    Public Address As String
    Public City As String
    Public State As String
    Public ZIP As String

    MustOverride Sub PrintName()

    Sub Print()
        PrintName()
        Console.WriteLine(Address)
        Console.WriteLine(City & "", " State & " & ZIP)
    End Sub
End Class

Class Customer
    Inherits Person

    Overrides Sub PrintName()
        Console.Write("Customer ")
        Console.WriteLine(Name)
    End Sub

    Public CustomerID As Integer
End Class

Class Employee
    Inherits Person

    Overrides Sub PrintName()
        Console.Write("Employee ")
        Console.WriteLine(Name)
    End Sub

    Public Salary As Integer
End Class
```

In this example, Person.Print can call the PrintName method, even though Person supplies no implementation for the method, because it is guaranteed that any derived class that can be instanced must provide an implementation.
Conclusion

Inheritance is a powerful way of expressing the relationships between types and reusing code across multiple types. The .NET Framework class libraries make extensive use of inheritance, and understanding inheritance is essential to understanding those libraries. Overridable methods and abstract methods provide a way for derived classes to specialize the behavior of their base classes. In the next chapter, we will discuss another way of reusing code across types: interfaces.

Here are some style points to keep in mind.

- The Object type combines the capabilities that used to be split between the Object type and the Variant type in previous versions of Visual Basic.
- Unless code is particularly performance sensitive and doing a lot of unboxing, CType is more general than DirectCast and is preferred.