Overview of Visual Basic 9.0

Erik **Meijer**

Amanda **Silver**

Paul **Vick**

Abstract

"Visual Basic code-named Orcas" (Visual Basic 9.0) introduces several language extensions that build on "Visual Basic code-named Whidbey" (Visual Basic 8.0) to support data-intensive programming --creating, updating, and querying relational databases, XML documents, and object graphs--in a unified way. In addition, Visual Basic 9.0 introduces several new language features to enhance Visual Basic's unique facility for static typing where possible, and dynamic typing where necessary. These new features are: *Implicitly typed local variables*, *Query comprehensions*, *Object initializers*, *Anonymous types*, *Full integration with the Linq framework*, *Deep XML support*, *Relaxed delegates*, *Nullable types*, *Dynamic interfaces*, and *Dynamic identifiers*.

This document is an informal overview of these new features. More information, including updates to the Visual Basic language definition and compiler previews, is available on the Visual Basic Developer Center (http://msdn.microsoft.com/vbasic/default.aspx).

Table of Contents

1. Getting Started With Visual Basic 9.0

To see the power of these language features at work, let's start with a real world example --the CIA World Factbook database http://www.odci.gov/cia/publications/factbook/. The database contains a variety of geographic, economic, social, and political information about the worlds countries. For the sake of our example, we begin with a schema for the name of each country and its capital, total area, and population. We represent this schema in Visual Basic 9.0 using the following class:

```
Class Country
  Public Property Name As String
  Public Property Area As Float
  Public Property Population As Integer
End Class
```

Here is a small subset of the country database that we will use as our running example:

```
Dim Countries = _
{ new Country{ _
    .Name = "Palau", .Area = 458, .Population = 16952 }, _
  new Country{ _
    .Name = "Monaco", .Area = 1.9, .Population = 31719 }, _
  new Country{ _
    .Name = "Belize", .Area = 22960, .Population = 219296 }, _
  new Country{ _
    .Name = "Madagascar", .Area = 587040, .Population = 13670507 } _
}
```

Given this list, we can query for all countries whose population is less than one million by using the following query comprehension:

```
Dim SmallCountries = _
  Select Country _
  From Country In Countries _
  Where Country.Population < 1000000

For Each Country As Country In SmallCountries
  Console.WriteLine(Country.Name)
Next
```

Because only Madagascar has more than one million inhabitants, the above program would print the following list of country names when compiled and run:

```
Palau
Monaco
```

```
Belize
```

Let's examine the program to understand the features of Visual Basic 9.0 that made this so simple to write. First of all, the declaration of the `Countries` variable

```
Dim Countries = _
  { new County { .Name = "Palau", .Area = 458, .Population = 16952 }, _
  ... _
  }
```

uses the new *object-initializer* syntax `new Country {..., .Area = 458, ...}` to create a complex object instance through a concise, expression-based syntax similar to the existing `With` statement.

The declaration also illustrates *implicitly typed local-variable* declarations, where the compiler infers the type of the local variable `Countries` from the initializer expression on the right-hand side of the declaration. The declaration above is precisely equivalent to an explicitly typed local-variable declaration of type `Country()`.

```
Dim Countries As Country() = {...}
```

To repeat, this is still a strongly typed declaration; the compiler has automatically inferred the type of the right-hand side of the local declaration, and there is no need for the programmer to manually enter that type into the program.

The local-variable declaration `SmallCountries` is initialized with a SQL-style query comprehension to filter out all countries that have fewer than one million inhabitants. The resemblance to SQL is intentional, enabling programmers who already know SQL to get started with Visual Basic query syntax all the more quickly.

```
Dim SmallCountries = _
    Select Country _
    From Country In Countries _
    Where Country.Population < 1000000
```

Note that we have another application of implicit typing: the compiler infers the type of `SmallCountries` as `IEnumerable(Of Country)`. The compiler translates the query comprehension itself into standard query operators. In this case, the translation could be as simple as the following:

```
Function F(Country As Country) As Boolean
  Return Country.Population < 1000000
End Function

Dim SmallCountries As IEnumerable(Of Country) = _
  Countries.Where(AddressOf F)
```

The expanded syntax passes the compiler-generated local function as a delegate `AddressOf F` to the extension function `Where`, which is defined in the standard query operator library as an extension of the `IEnumerable(Of T)` interface.

Now that we have seen a few of the new features in Visual Basic 9, let us drill down into a more detailed overview.

2. Implicitly Typed Local Variables

In an implicitly typed local-variable declaration, the type of the local variable is inferred from the initializer expression on the right-hand side of a local declaration statement. For example, the compiler infers the types of all the following variable declarations:

```
Dim Population = 31719
Dim Name = "Belize"
Dim Area = 1.9
Dim Country = New Country{ .Name = "Palau", ...}
```

Hence they are precisely equivalent to the following, explicitly typed declarations:

```
Dim Population As Integer = 31719
Dim Name As String = "Belize"
Dim Area As Float = 1.9
Dim Country As Country = New Country{ .Name = "Palau", ...}
```

Because types of local-variable declarations are inferred by default, no matter what the setting of `Option Strict` is, access to such variables is always early-bound. The programmer must explicitly specify late binding in Visual Basic 9.0, by explicitly declaring variables as of type `Object`, as follows:

```
Dim Country As Object = new Country{ .Name = "Palau", ... }
```

Requiring explicit late binding prevents accidentally using late binding and, more importantly, it allows powerful extensions of late binding to new data types such as XML, as we will see below. There will be an optional project-level switch to toggle the existing behaviour.

The loop-control variable in a `For...Next` or a `For Each...Next` statement can also be an implicitly typed variable. When the loop-control variable is specified, as in `For Dim I = 0 To Count`, or as in `For Each Dim C In SmallCountries`, the identifier defines a new, implicitly typed local variable, whose type is inferred from the initializer or collection expression and is scoped to the entire loop. This use of `Dim` to the right of `For` is new to Visual Basic 9.0, as are implicitly typed loop-control variables.

With this application of type inference, we can rewrite the loop that prints all small countries as follows:

```
For Each Dim Country In SmallCountries
  Console.WriteLine(Country.Name)
Next
```

The type of `Country` is inferred to be `Country`, the element type of `SmallCountries`.

3. Object and Collection Initializers

In Visual Basic, the `With` statement simplifies access to multiple members of an aggregate value without specifying the target expression multiple times. Inside the `With` statement block, a member-access expression starting with a period is evaluated as if the period were preceded by the target expression of the `With` statement. For example, the following statements initialize a new `Country` instance and subsequently initializes its fields to the required values:

```
Dim Palau = New Country()
With Palau
  .Name = "Palau"
  .Area = 458
  .Population = 16952
End With
```

The new Visual Basic 9.0 *object initializers* are an expression-based form of `With` for creating complex object instances concisely. Using object initializers, we can capture the above two statements into a single (implicitly typed) local declaration, as follows:

```
Dim Palau = New Country { _
  .Name = "Palau", _
  .Area = 458, _
  .Population = 16952
}
```

This style of object initialization from expressions is important for queries. Typically, a query looks like an object declaration initialized by a `Select` clause on the right-hand side of the equals sign. Because the `Select` clause returns an expression, we must be able to initialize the entire object with a single expression.

As we have seen, object initializers are also convenient for creating collections of complex objects. Any collection that supports an `Add` method can be initialized using a *collection initializer* expression. For instance, given the declaration for cities as the partial class,

```
Partial Class City
  Public Property Name As String
  Public Property Country As String
  Public Property Longitude As Float
  Public Property Latitude As Float
End Class
```

we can create a `List(Of City)` of capital cities of our example countries as follows:

```
Dim Capitals = New List(Of City){ _
  { .Name = "Antanarivo", _
    .Country = "Madagascar", _
    .Longitude = 47.4, _
    .Lattitude = -18.6 }, _
  { .Name = "Belmopan", _
    .Country = "Belize", _
```

```
    .Longitude = -88.5, _
    .Latitude = 17.1 }, _
  { .Name = "Monaco", _
    .Country = "Monaco", _
    .Longtitude = 7.2, _
    .Latitude = 43.7 }, _
  { .Country = "Palau",
    .Name = "Koror", _
    .Longitude = 135, _
    .Latitude = 8 } _
}
```

This example also uses nested object initialers, where the constructors of the nested initializers are inferred from the context. In this case, each nested initializer is precisely equivalent to the full form `New City{...}`.

4. Anonymous Types

Often, we just want to remove, or to *project* out, certain members of a type as the result of a query. For example, we might want to know just the `Name` and `Country` of all tropical capital cities, using the `Latitude` and `Longitude` columns in the source data to identify the tropics, but projecting away those columns in the result. In Visual Basic 9.0, we do this by creating a new object instance --without naming the type --for each city `C` whose latitude is in between the tropic of Cancer and the tropic of Capricorn:

```
Const TropicOfCancer = 23.5
Const TropicOfCapricorn = -23.5

Dim Tropical = Select New{ .Name = City.Name, .Country = City.Country } _
               From City In Capitals _
               Where TropicOfCancer =< City.Latitude _
               AndAlso City.Latitude <= TropicOfCapricorn
```

The inferred type of the local variable `Tropical` is a collection of instances of an anonymous type, that is, `IEnumerable(Of { Name As String, Country As String })`. The Visual Basic compiler will create a new, system-generated class, for example, `_Name_As_String_Country_As_String_`, whose member names and types are inferred from the object initializer, as follows:

```
Class _Name_As_String_Country_As_String_
    Public Property Name As String
    Public Property Country As String
    Public Default Property Item(Index As Integer) As Object
    ...
End Class
```

Within the same program, the compiler will merge identical anonymous types. Two anonymous object initializers that specify a sequence of properties of the same names and types in the same order will produce instances of the same anonymous type. Externally, Visual Basic-generated anonymous types are erased to `Object`, which allows the compiler to uniformly pass anonymous types as arguments and results of functions. For use within Visual Basic code, the compiler decorates the generated class with special custom attributes to remember that the type

`_Name_As_String_Country_As_String_` actually represents the anonymous type `{ Name As String, Country As String }`.

Because anonymous types are typically used to project members of an existing type, Visual Basic 9.0 allows the shorthand projection notation `New { City.Name, City.Country }` to abbreviate the long-form `New { .Name = City.Name, .Country = City.Country }`. When used in the result expression of a query comprehension, we can abbreviate projection initializers even further, as follows:

```
Dim Tropical = Select City.Name, City.Country _
               From City In Capitals _
               Where TropicOfCancer =< City.Latitude _
               AndAlso City.Latitude >= TropicOfCapricorn
```

Note that both of these abbreviated forms are identical in meaning to the long form above.

5. Deep XML Support

XLinq is a new, in-memory XML programming API designed specifically to leverage the latest .NET Framework capabilities such as the Language-Integrated Query framework. Just as query comprehensions add familiar, convenient syntax over the underlying standard .NET Framework query operators, Visual Basic 9.0 provides deep support for XLinq through *XML literals* and *late binding over XML*.

To illustrate XML literals, let us query over the essentially flat relational data sources `Countries` and `Capitals` to construct a hierarchical XML model that nests the capital of each country as a child element and calculates the population density as an attribute.

To find the capital for a given country, we do a *join* on the name-member of each country with the country-member of each city. Given a country and its capital, we can then easily construct the XML fragment by filling in embedded expression *holes* with computed values. We would write a "hole" for a name attribute with parentheses, as in `Name=(Country.Name)`, and a "hole" for a child element with special angle-bracket syntax borrowed from ASP.NET, as in `<Name><%= City.Name %></Name>`. Here is our query that combines XML literals and query comprehensions:

```
Dim CountriesWithCapital As XElement = _
  <Countries>
    <%= Select <Country Name=(Country.Name)
                        Density=(Country.Population/Country.Area)>
                 <Capital>
                   <Name><%= City.Name %></Name>
                   <Longitude><%= City.Longitude %></Longtitude>
                   <Latitude><%= City.Latitude %></Latitude>
                 </Capital>
               </Country> _
        From Country In Countries, City In Capitals _
        Where Country.Name = City.Country %>
  </Countries>
```

Note that the type `XElement` could be omitted from the declaration, in which case it will be inferred, just like any other local declaration. We leave the explicit type in for this example, to make a point below.

In this declaration, we want the result of the `Select` query to be substituted inside the `<Countries>` element. Thus, the `Select` query is the content of the first "hole," demarcated by the familiar ASP.NET style tags `<%=` and `%>` inside `<Countries>`. Because the result of a `Select` query is an expression, and XML literals are expressions, it is natural, then, to nest another XML literal in the `Select` itself. This nested literal itself contains nested attribute "holes" for `Country.Name` and the computed population density ratio `Country.Population/Country.Area`, and nested element "holes" for the name and coordinates of the capital city.

When compiled and run, the above query will return the following XML document (reformatted slightly to save space):

```
<Countries>
 <Country Name="Palau" Density="0.037117903930131008">
   <Capital>
     <Name>Koror</Name><Longitude>135</Longitude><Latitude>8</Latitude>
   </Capital>
 </Country>
 <Country Name="Monaco" Density="16694.21052631579">
   <Capital>
     <Name>Monaco</Name><Longitude>7.2</Longitude><Latitude>3.7</Latitude>
   </Capital>
 </Country>
 <Country Name="Belize" Density="9.5512195121951216">
   <Capital>
    <Name>Belmopan</Name><Longitude>-88.5</Longitude><Latitude>17.1</Latitude>

   </Capital>
 </Country>
 <Country Name="Madagascar" Density="23.287181452711909">
   <Capital>
     <Name>Antananarivo</Name>
     <Longitude>47.4</Longitude><Latitude>-18.6</Latitude>
   </Capital>
  </Country>
</Countries>
```

Visual Basic 9.0 compiles XML literals into normal `System.Xml.XLinq` objects, ensuring full interoperability between Visual Basic and other languages that use XLinq. For our example query, the code produced by the compiler (if we could see it) would be:

```
Dim CountriesWithCapital As XElement = _
  New XElement("Countries", _
    Select New XElement("Country", _
             New XAttribute("Name", Country.Name), _
             New XAttribute("Density", Country.Population/Country.Area), _
             New XElement("Capital", _
               New XElement("Name", City.Name), _
               New XElement("Longitude", City.Longitude), _
               New XElement("Latitude", City.Latitude)))
    From Country In Countries, City In Capitals _
    Where Country.Name = City.Country)
```

Besides constructing XML, Visual Basic 9.0 also simplifies *accessing* XML structures via late binding over XML; that is, identifiers in Visual Basic code are bound at run time to corresponding XML attributes and elements. For example, we can print out the population density of all example countries as follows:

- Use the *child axis* expression `CountriesWithCapital.Country` to get all `"Country"` elements from the `CountriesWithCapital` XML structure;
- Use the *attribute axis* expression `Country.@Density` to get the `"Density"` attribute of the `Country` element;
- Use the *descendants axis* expression `Country...Latitude` --written literally as three dots in the source code-- to get all `"Latitude"` children of the `Country` element, no matter how deeply in the hierarchy they occur; and
- Use the *extension indexer* on `IEnumerable(Of T)` to select the first element of the resulting sequence.

If we put this all together, the code looks like this:

```
For Each Dim Country In CountriesWithCapital.Country
  Console.WriteLine("Density = "+ Country.@Density)
  Console.WriteLine("Latitude = "+ Country...Latitude(0))
Next
```

The compiler knows to use late binding over normal objects when the target expression of a declaration, assignment, or initialization is of type `Object` rather than of some more specific type. Likewise, the compiler knows to use late binding over XML when the target expression is of type, or collection of, `XElement`, `XDocument`, or `XAttribute`.

As a result of late binding over XML, the compiler translates as follows:

- The child-axis expression `CountriesWithCapital.Country` translates into the raw XLinq call `CountriesWithCapital.Elements("Country")`, which returns the collection of all child elements named `"Country"` of the `Country` element;
- The attribute axis expression `Country.@Density` translates into `Country.Attribute("Density")`, which returns the single child attribute named `"Density"` of `Country`; and
- The descendant axis expression `Country...Latitude(0)` translates into a combination of `ElementAt(Country.Descendants(Latitude),0)`, which returns the collection of all elements named at any depth below `Country`.

6. Query Comprehensions

Query comprehensions provide a language integrated syntax for queries that is very similar to SQL, but adopted to fit well with the look and feel of Visual Basic on the one hand, and on the other hand to integrate smoothly with the new .NET language integrated query framework.

Those familiar with the implementation of SQL will recognize, in the underlying .NET Framework sequence operators, many of the compositional relational-algebra operators such as projection, selection, cross-product, grouping, and sorting that represent query plans inside the query processor.

Because the semantics of query comprehensions are defined by translating them into sequence operators, the underlying operators are bound to whatever sequence operators are in scope. This implies that by importing a particular implementation, the query-comprehension syntax can effectively be re-bound by the user. In particular, query comprehensions can be re-bound to a sequence-operators implementation that uses the DLinq infrastructure, or a local query optimizer

that attempts to distribute the execution of the query over several local or remote data sources. This rebinding of the underlying sequence operators is similar in spirit to the classic COM provider model, whereby different implementations of the same interface can support a great variety of operational and deployment choices without modifying the overlying application code.

The basic `Select...From...Where...` comprehension filters out all values that satisfy the predicate in the `Where` clause. One of our very first examples showed how to find all countries with fewer than a million inhabitants:

```
Dim SmallCountries = _
  Select Country _
  From Country In Countries _
  Where Country.Population < 1000000
```

Inside a sequence operator, the identifier `It` is bound to the current "row". Like `Me`, members of `It` are automatically in-scope. The notion of `It` corresponds to XQuery's context item "." and it can be used like "*" in SQL. For example, we can return the collection of all countries with their capitals by using the following query:

```
Dim CountriesWithCapital = _
  Select It _
  From Country In Countries, City In Capitals _
  Where Country.Name = City.Country
```

The inferred type for this local declaration is `IEnumerable(Of { Country As Country, City As City })`.

Using the `Order By` clause, we can sort the results of queries according to any number of sort keys. For example, the following query returns a list of the names of all countries, sorted by their longtitude in ascending order and by their population in descending order:

```
Dim Sorted = _
  Select Country.Name _
  From Country In Countries, City In Capitals _
  Where Country.Name = City.Country
  Order By City.Longtitude Asc, Country.Population Desc
```

Aggregate operators such as `Min`, `Max`, `Count`, `Avg`, `Sum` ... operate on collections and "aggregate" them to single values. We can count the number of small countries using the following query:

```
Dim N As Integer = _
  Select Count(Country) _
  From Country In Countries _
  Where Country.Population < 1000000
```

Like SQL, we provide special syntax for aggregates, which is extremely convenient for "tupling" multiple aggregate operations. For example, to count the number of small countries and compute their average density with one statement, we can write

```
Dim R As { Total As Integer, Density As Double } = _
  Select New { .Total = Count(Country), _
               .Density = Avg(Country.Population/Country.Area) } _
  From Country In Countries _
  Where Country.Population < 1000000
```

This form of aggregation is shorthand for applying a compiler-generated aggregate function over the result of the normal result set without any aggregation.

Aggregate functions appear most often in combination with partitioning the source collection. For example, we can group all countries by whether they are tropical and *then* aggregate the count of each group. To do so, we introduce the `Group By` clause. The helper function `IsTropical` encapsulates the test whether a `Country` has a tropical climate:

```
Partial Class Country
Function IsTropical() As Boolean
  Return TropicOfCancer =< Me.Latitude _
    AndAlso Me.Latitude >= TropicOfCapricorn
  End Function
End Class
```

Given this helper function, we use exactly the same aggregation as above, but first partition the input collection of `Country` and `Capital` pairs into groups for which `Country.IsTropical` is the same. In this case there are two such groups: one that contains the tropical countries Palau, Belize, and Madagascar; and another that contains the nontropical country Monaco.

Key	Country	City
`Country.IsTropical() = True`	Palau	Koror
`Country.IsTropical() = True`	Belize	Belmopan
`Country.IsTropical() = True`	Madagascar	Antanarivo
`Country.IsTropical() = False`	Monaco	Monaco

Then, we aggregate the values in these groups by computing the total count and average density. The result type is now a collection of pairs of `Total As Integer` and `Density As Double`:

```
Dim CountriesByClimate _
  As IEnumerable(Of Total As Integer, Density As Double }) =
    Select New { .Total = Count(Country), _
           .Density = Avg(Country.Population/Country.Area) } _
    From Country In Countries, City In Capitals _
    Where Country.Name = City.Country
    Group By Country.IsTropical()
```

The above query hides considerable complexity inasmuch as the result of the `Group By` clause is actually a collection of grouping values of type `IEnumerable(Of Grouping(Of { Boolean, { Country As Country, City As City }))`, much like the table above. Each such `Grouping` item contains a `Key` member derived from the key-extraction expression `Country.IsTropical()` and a `Group` that contains the unique collection of countries and cities for which the key extraction expression has the same value. The Visual Basic compiler synthesizes the user-defined aggregate function, that given such a grouping, calculates the required result by aggregating over each partition.

Note that in the previous example each `Group` contains both the `Country` and `Capital`, whereas we only need the `Country` to compute the final result of the query. The `Group By` clause allows for a preselection of the groups. For example, we can partition the names of all countries by their hemisphere using the following comprehension:

```
Dim ByHemisphere As IEnumerable(Of Grouping(Of Boolean, String)) = _
  Select It _
  From Country In Countries, City In Capitals _
  Where Country.Name = City.Country
  Group Country.Name By City.Latitude >= 0
```

This would return the collection `{ New Grouping { .Key = False, .Group = { "Madagascar", "Belize" }}, New Grouping { .Key = True, .Group = { "Palau" }}`.

Query comprehensions in Visual Basic 9.0 are fully compositional, meaning that query comprehensions can be arbitrarily nested, restricted only by the static-typing rules. Compositionality makes it easy to understand a large query by simply understanding each individual subexpression in isolation. Compositionality also makes it easy to define the semantics and typing rules of the language clearly. Compositionality, as a design principle, is rather different from the principles that underlie the design of SQL. The SQL language is not fully compositional, and rather has an ad-hoc design with many special cases that grew over time as experience with databases accumulated in the community. Due to the lack of full compositionality, however, it is not possible, in general, to understand a complex SQL query by understanding the individual pieces.

One of the reasons that SQL lacks compositionality is that the underlying relational data model is itself not compositional. For instance, tables may not contain subtables; in other words, all tables must be flat. As a result, instead of breaking up complex expressions into smaller units, SQL programmers write monolithic expressions whose results are flat tables, fitting to the SQL data model. To quote Jim Gray, "anything in computer science that is not recursive is no good." Because Visual Basic is based on the CLR type system, there are no restrictions on what types can appear as components of other types. Aside from static typing rules, there are no restrictions on the kind of expressions that can appear as components of other expressions. As a result, not only rows, objects, and XML, but also active directory, files, registry entries, and so on, are all first-class citizens in query sources and query results.

7. Extension Methods

Much of the underlying power of the .NET Framework standard query infrastructure comes from *extension methods*. In fact the compiler translates all query comprehensions directly into the standard query operator extension methods defined by the namespace that is in scope. Extension methods are shared methods marked with custom attributes that allow them to be invoked through instance-method syntax. In effect, extension methods extend existing types and constructed types with additional methods.

Because extension methods are intended mostly for library designers, Visual Basic does not offer direct language syntax support for declaring them. Instead, authors directly attach the required custom attributes on modules and members to mark them as extension methods. The following example defines an extension method `Count` on arbitrary collections:

```
<System.Runtime.CompilerServices.Extension> _
Module MyExtensions
  <System.Runtime.CompilerServices.Extension> _
  Function Count(Of T)([Me] As IEnumerable(Of T)) As Integer
    For Each Dim It In [Me]
      Count += 1
    Next
  End Function
End Module
```

Recall that the square-bracket syntax is a keyword escape, permitting `Me` to be used as the name of an ordinary variable. Because the extension method is a shared method that will *simulate* an instance method, it is convenient to use the identifier `Me` as the name of the input, as we would in an actual instance method, but it must be escaped with brackets since it is a keyword, and therefore not really allowed in a shared method.

Extension methods are just regular shared methods, hence we can invoke the `Count` function as we would invoke any other shared function in Visual Basic, by just supplying explicitly the instance collection on which to operate:

```
Dim TotalSmallCountries = _
  MyExtensions.Count(Select Country _
        From Country In Countries _
        Where Country.Population < 1000000)
```

Extension methods come into scope through the normal `Imports` statement. These extension methods will then appear as additional methods on the types given by their first parameter.

```
Imports MyExtensions

Dim TotalSmallCountries = _
  (Select Country _
   From Country In Countries _
   Where Country.Population < 1000000).Count()
```

Extension methods have lower precedence than regular instance methods; if the normal processing of an invocation expression finds no applicable instance methods, the compiler tries to interpret the invocation as an extension-method invocation.

The most natural way to write this query, however, is to use aggregate syntax, as we have seen before:

```
Dim TotalSmallCountries = _
   Select Count(Country) _
   From Country In Countries _
   Where Country.Population < 1000000
```

8. Nested Functions

Many of the standard query operators such as `Where`, `Select`, `SelectMany`, etc. are defined as extension methods that take delegates of type `Func(Of S,T)` as arguments. In order for the compiler to translate comprehensions into

the underlying query operators, or in order for Visual Basic programmers to call query operators directly, there is a need to create delegates easily. In particular, we need to be able to create so-called *closures*, delegates that capture their surrounding context. The Visual Basic mechanism for creating closures is through nested local function and subroutine declarations.

To show the use of nested function, we will call into the raw underlying query operators as defined in the `System.Query` namespace. One of the extension methods is the `TakeWhile` function that yields elements from a sequence while a test is true and then skips the remainder of the sequence.

```
<Extension> _
Shared Function TakeWhile(Of T) _
  (source As IEnumerable(Of T), Predicate As Func(Of T, Boolean)) _
    As IEnumerable(Of T)
```

The `OrderByDescending` operator sorts its argument collection in descending order according to the proved sort key:

```
<Extension> _
Shared Function OrderByDescending (T, K As IComparable(Of K)) _
  (Source As IEnumerable(Of T), KeySelector As Func(Of T, K)) _
     As OrderedSequence(Of T)
```

An alternative way of finding all small countries is by first sorting them by population, and then using `TakeWhile` to pick out all the countries that have less than a million inhabitants.

```
Function Population(Country As Country) As Integer
  Return Country.Population
End Function

Function LessThanAMillion(Country As Country) As Boolean
  Return Country.Population < 1000000
End Function

Dim SmallCountries = _
  Countries.OrderBy(AddresOf Population) _
           .TakeWhile(AddresOf LessThanAMillion)
```

Though it is not required for query comprehensions, Visual Basic may support direct syntax for anonymous functions and subroutines (so called "lambda expressions"), which would be translated by the compiler to local function declarations.

9. Nullable Types

Relational databases present semantics for nullable values that are often inconsistent with ordinary programming languages and often unfamiliar to programmers. In data-intensive applications, it is critical for programs to handle these semantics clearly and correctly. Recognizing this necessity, in "Whidbey" the CLR has added run-time support for nullability using the generic type `Nullable(Of T As Struct)`. Using this type we can declare nullable versions of value types such as `Integer`, `Boolean`, `Date`, etc. For reasons that will become apparent, the Visual Basic syntax for nullable types is `T?`.

For example, because not all countries are independent, we can add a new member to the class `Country` that represents their independence date, if applicable:

```
Partial Class Country
  Public Property Independence As Date?
End Class
```

Just as with array types, we can also affix the nullable modifier on the property name, as in the following declaration:

```
Partial Class Country
  Public Property Independence? As Date
End Class
```

The independence date for Palau is `#10/1/1994#`, but the British Virgin Islands are a dependent territory of the United Kingdom, and hence its independence date is `Nothing`.

```
Dim Palau = _
  New Country { _
    .Name = "Palau", _
    .Area = 458, _
    .Population = 16952, _
    .Independence = #10/1/1994# }

Dim VirginIslands = _
  New Country { _
    .Name = "Virgin Islands", _
    .Area = 150, _
    .Population= 13195, _
    .Independence = Nothing }
```

Visual Basic 9.0 will support three-valued logic and null propagation arithmetic on nullable values, which means that if one of the operands of an arithmetic, comparison, logical or bitwise, shift, string, or type operation is `Nothing`, the result will be `Nothing`. If both operands are proper values, the operation is performed on the underlying values of the operands and the result is converted to nullable.

Because both `Palau.Independence` and `VirginIslands.Independence` have type `Date?`, the compiler will use null-propagating arithmetic for the substractions below, and hence the inferred type for the local declaration `PLength` and `VILength` will both be `TimeSpan?`.

```
Dim PLength = #8/24/2005# - Palau.Independence          REM 3980.00:00:00
```

The value of `PLength` is `3980.00:00:00` because neither of the operands is `Nothing`. On the other hand, because the value of `VirginIslands.Independence` is `Nothing`, the result is again of type `TimeSpan?`, but the value of `VILength` will be `Nothing` because of null-propagation.

```
Dim VILength = #8/24/2005# - VirginIslands.Independence REM Nothing
```

As in SQL, comparison operators will do null propagation, and logical operators will use three-valued logic. In `If` and `While` statements, `Nothing` is interpreted as `False`; hence in the following code snippet, the `Else` branch is taken:

```
If VILength &lt; TimeSpan.FromDays(10000)
  ...
Else
  ...
End If
```

Note that under three-valued logic, the equality checks `X = Nothing`, and `Nothing = X` always evaluates to `Nothing`; in order to check if `X` is `Nothing`, we should use the two-valued logic comparison `X Is Nothing` or `Nothing Is X`.

The run time treats nullable values specially when boxing and unboxing to and from `Object`. When boxing a nullable value that represents `Nothing` (that is, the `HasValue` property is `False`), that value is boxed into a null reference. When boxing a proper value (that is, the `HasValue` property is `True`), the underlying value is first unwrapped and then boxed. Because of this, no object on the heap has dynamic type `Nullable(Of T)`; all such apparent types are rather just `T`. Dually, we can unbox values from `Object` into either `T`, or into `Nullable(Of T)`. However, the consequence of this is that late-binding cannot dynamically decide wether to use two-valued or three-valued logic. For example, when we do an early-bound addition of two numbers, one of which is `Nothing`, null propagation is used, and the result is `Nothing`:

```
Dim A As Integer? = Nothing
Dim B As Integer? = 4711
Dim C As Integer? = A+B REM C = Nothing
```

However, when using late-bound addition on the same two values, the result will be `4711`, because the late binding will use two-valued logic based on the fact that the dynamic type of both `A` and `B` is `Integer`, not `Integer?`. Hence `Nothing` is interpreted as `0`:

```
Dim X As Object = A
Dim Y As Object = B
Dim Z As Object = X+Y REM Z = 4711
```

In order to ensure the correct semantics, we need to direct the compiler to use the null-propagating overload

```
Operator +(x As Object?, y As Object?) As Object?
```

by converting either of the operands to a nullable type using the ? operator:

```
Dim X As Object = A
Dim Y As Object = B
Dim Z As Object? = X?+Y REM Z = Nothing
```

Note that this implies that we must be able to create `T?` for any type `T`. The underlying CLR `Nullable(Of T As Struct)` type constrains the argument type to non-nullable structures only. The Visual Basic compiler erases `T?` to

`T` where `T` is not a non-nullable value type, and to `Nullable(Of T)` when T is a non-nullable value type. The compiler keeps around enough internal metadata to remember that within the Visual Basic program, the static type in both cases is `T?`.

10. Relaxed Delegates

When creating a delegate using `AddressOf` or `Handles` in Visual Basic 8.0, one of the methods targeted for binding to the delegate identifier must exactly match the signature of the delegate's type. In the example below, the signature of the `OnClick` subroutine must exactly match the signature of the event handler delegate `Delegate Sub EventHandler(sender As Object, e As EventArgs)`, which is declared behind the scenes in the `Button` type:

```
Dim WithEvents B As New Button()

Sub OnClick(sender As Object, e As EventArgs) Handles B.Click
  MessageBox.Show("Hello World from" + B.Text)
End Sub
```

However, when invoking *non-delegate* functions and subroutines, Visual Basic does not require the actual arguments to exactly match one of the methods we are trying to invoke. As the following fragment shows, we can actually invoke the `OnClick` subroutine using an actual argument of type `Button` and of type `MouseEventArgs`, which are subtypes of the formal parameters `Object` and `EventArgs`, respectively:

```
Dim M As New MouseEventArgs(MouseButtons.Left, 2, 47, 11,0)
OnClick(B, M)
```

Conversely, suppose that we could define a subroutine `RelaxedOnClick` that takes two `Object` parameters, and then we are allowed to call it with actual arguments of type `Object` and `EventArgs:`

```
Sub RelaxedOnClick(sender As Object, e As Object) Handles B.Click
  MessageBox.Show("Hello World from" + B.Text))
End Sub
Dim E As EventArgs = M
Dim S As Object = B
RelaxedOnClick(B,E)
```

In Visual Basic 9.0, binding to delegates is relaxed to be consistent with method invocation. That is, if it is possible to *invoke* a function or subroutine with actual arguments that exactly match the formal-parameter and return types of a delegate, we can bind that function or subroutine to the delegate. In other words, delegate binding and definition will follow the same overload-resolution logic that method invocation follows.

This implies that in Visual Basic 9.0 we *can* now bind a subroutine `RelaxedOnClick` that takes two `Object` parameters to the `Click` event of a `Button`:

```
Sub RelaxedOnClick(sender As Object, e As Object) Handles B.Click
  MessageBox.Show(("Hello World from" + B.Text)
End Sub
```

The two arguments to the event handler, `sender` and `EventArgs`, very rarely matter. Instead, the handler accesses the state of the control on which the event is registered directly and ignore its two arguments. To support this common case, delegates can be relaxed to take no arguments, if no ambiguities result. In other words, we can simply write the following:

```
Sub RelaxedOnClick Handles B.Click
  MessageBox.Show("Hello World from" + B.Text)
End Sub
```

It is understood that delegate relaxation also applies when constructing delegates using an `AddressOf` or delegate creation expression, even when the method group is a late-bound call:

```
Dim F As EventHandler = AddressOf RelaxedOnClick
Dim G As New EventHandler(AddressOf B.Click)
```

11. Dynamic Interfaces (or Strong "Duck Typing")

In purely statically typed languages such as C# or Java or Visual Basic (with `Option Strict On`), members must exist at compile time on the type of the target expression. For example, the second assignment below causes a compile-time error because class `Country` does not have an `Inflation` member:

```
Dim Palau As Country = Countries(0)
Dim Inflation = Country.Inflation
```

However, in many situations, it is necessary to access a member even though the type of the target type is unknown at compile-time; this is a common scenario extension fields customized during application deployment. With `Option Strict Off`, Visual Basic allows late-bound member access on targets of type `Object`. While powerful and extremely flexible, late-binding comes with a cost. In particular, the user does not benefit from Intellisense, type inference, and compile-time checking and needs casts or explicit types to move back to the early-bound world.

Even when making late-bound call, it is common to assume that the value adheres to a certain "interface." As long as the object satisfies that interface, the call will succeed. The dynamic-language community calls this "Duck Typing": if it walks like a duck and talks like a duck, then it is a duck. To illustrate the idea of Duck Typing, the example below returns contact information from a `School` or a `Citizen`, both of which have a `Name` property of type `String`, and a `Phone` property of type `Integer`.

```
Function ContactInfo(country As Country, Address As String) As Object
  For Each Dim school In Country.Schools
    If school.Address = addr Then
      Return New { Name := school,PrincipalName, school.Phone }
    End If
  Next

  For Each Dim citizen in Country.Citizens
    If citizen.Address = addr Then
      Return citizen
    End If
```

```
  Next
End Function
```

When attempting to access the `Name` property of the result type using late-binding, there is a static assumption that the value returned by `ContactInfo` has a `Name` member of type `String`. Using the new feature of dynamic interfaces, we this assumption can be made explicit. A target whose static type is a dynamic interface is always accessed using late-binding, but the member access is statically typed. This means that the user benefits from full Intellisense and type inference, and do not have to do any casting or explicit typing:

```
Dynamic Interface Contact
  Property Name As String
  Property Address As Integer
End Interface

Dim Contact As Contact = ContactInfo(country, "123 Main Street")
Dim Name = contact.Name                REM Inferred As String.
```

12. Dynamic Identifiers

Late binding allows programmers to call methods on receiver objects whose type is unknown at compile-time. Dynamic interfaces leverage the fact that programmers assume that statically they know the name and signatures of the members they expect in a late-bound invocation. However, in certain truly dynamic scenarios, we might not even know the type of the receiver nor the name of the member we want to invoke. Dynamic identifiers allow for extremely late-bound calls where the identifier and the argument list of an invocation expression or the type and the argument list of a constructor call are computed dynamically.

An example of using dynamic identifiers is in test-driven development where you specify test information in an XML file and execute each test dynamically. Suppose we want to test the `IsTropical` function that we defined earlier:

```
Partial Class Country
  Function IsTropical() As Boolean
    Return TropicOfCancer =< Me.Latitude _
           AndAlso Me.Latitude >= TropicOfCapricorn
  End Function
End Class
```

To do so, we first create an XML file that contains a number of tests that specify the method to be called and the expected result, the constructor for the receiver and the arguments for the actual call:

```
<tests>
  <test method="IsTropical" result="False">
    <receiver classname="Country">
      <argument>Monaco</argument>
    </receiver>
  </test>
  <test method="IsTropical" result="True">
    <receiver classname="Country">
      <argument>Belize</argument>
```

```
    </receiver>
  </test>
</tests>
```

In other words, the XML file above encodes the following two tests

```
Debug.Assert(New Country("Monaco").IsTropical() = False)
Debug.Assert(New Country("Belize").IsTropical() = True)
```

Our task is to interpret the XML file such that it runs this code. Using type inference, extension methods, query comprehensions, dynamic identifiers and late-binding over XML it is very easy to write what is essentially a Visual Basic interpreter in Visual Basic:

```
    Sub RunTests (Tests As XElement)
      For Each Dim Test In Tests
        REM dynamically create receiver object
        Dim ConstructorType = Type.GetType(Test.receiver.@classname)
        Dim ConstructorArgs = _
              (Select a.Value() From a In Test.receiver.argument).ToArray()
        Dim Instance = New(ConstructorType)(ConstructorArgs)
        REM dynamically call member
        Dim MethodName = CStr(test.@method)
        Dim MethodArgs = _
              (Select a.Value() From a In test.receiver.argument).ToArray()
        Dim Result = Receiver.(Method)(MethodArgs)
        REM check for expected result
        Debug.Assert(Result = test.@result)
      Next
    End Sub
```

The dynamic constructor call expression `New(ConstructorType)(ConstructorArgs)` dynamically computes calls the constructor for the type `ConstructorType` computed from the class attribute as specified in the receiver element of the test and the actual arguments ConstructorArgs as given by the argument child elements of the receiver element of the test. Under the covers, it calls the `Activator.CreateInstance(Type, Object())` method. Similarly, the dynamic invocation expression `Instance.(MethodName)(MethodArgs)` dynamically calls the method `MethodName` on the receiver `Instance`, passing `MethodArgs` as the actual arguments. In this case the method name is taken from the method attribute of the test and the actual arguments are taken from the argument children of the test. Under the covers, as in any late-bound situation, the normal `NewLateBinding.LateCall` is used. Finally, the computed result is compared to the expected result as specified by the result attribute of the test.

The corresponding code in a language such as C# that does not support dynamism at all is at least an order of magnitude larger and requires a lot of mind-numbing reflection plumbing code. However, even in many dynamic languages such as Python, PHP, or VB 8, it is not as easy to call constructor and methods where the type repectively the method name is computed at runtime. For example, in VB 8 the equivalent code would have been about five times as large and be something incomprehensible like:

```
Sub RunTests (Tests As XElement)
  For Each Test As XElement In Tests.Elements("test")
    REM dynamically create receiver object
```

```
    Dim ConstructorType As System.Type = _

            Type.GetType(Test.Element("receiver").Attribute("receiver"))
    Dim ConstructorArgsList As New List(Of Object)
    For Each Parameter As XElement In _
        XElementSequence.Elements(Test.Element("receiver"), "parameter")
      ConstructorArgsList.Add(Parameter.Value())
    Next
    Dim ConstructorArgs = ConstructorArgsList.ToArray()
    Dim Instance = Activator.CreateInstance _
                       (ConstructorType, ConstructorArgs)
    REM dynamically call member
    Dim MethodName As String = Test.Attribute("receiver")
    Dim MethodArgsList As New List(Of Object)
    For Each MethodArg As XElement In _
        test.Elements("parameter")
      MethodArgsList.Add(MethodArg.Value())
    Next
    Dim MethodArgs As Object() = MethodArgsList.ToArray()
    REM cannot directly use late binding
    Dim Result = NewLateBinding.LateCall _
          (Instance, Nothing, MethodName, MethodArgs, _
           Nothing, False, False)
    REM check for expected result
    Debug.Assert(Result = test.@result)
  Next
End Sub
```

It is remarkable how lifting the arbitrary restriction on computing types and method names unleashes the full power of reflexive metaprogramming that is typically only found in languages such as SmallTalk, directly to the Visual Basic user. This makes Visual Basic 9.0 the language of choice for modern Agile and test-driven development methodologies.

13. Conclusion

Visual Basic 9.0 introduces a variety of new features. In this document, we have presented these features in a series of linked examples, but the underlying themes deserve emphasis as well:

- *Relational, object, and XML data*. Visual Basic 9.0 unifies access to data independently of its source in relational databases, XML documents, or arbitrary object graphs, however persisted or stored in memory. The unification consists in styles, techniques, tools, and programming patterns. The especially flexible syntax of Visual Basic makes it easy to add extensions like XML literals and SQL-like query comprehensions deeply into the language. This greatly reduces the "surface area" of the new .NET Language Integrated Query APIs, increases the discoverability of data-access features through IntelliSense and Smart Tags, and vastly improves debugging by lifting foreign syntaxes out of string data into the host language. In the future, we intend to increase the consistency of XML data even further by leveraging XSD schemas.

- *Increased dynamism with all the benefits of static typing*. The benefits of static typing are well known: identifying bugs at compile time rather than run time, high performance through early-bound access, clarity through explicitness in source code, and so on. However, sometimes, dynamic typing makes code shorter, clearer, and more flexible. If a language does not directly support dynamic typing, when programmers need it they must implement bits and pieces of dynamic structure through reflection, dictionaries, dispatch tables, and other techniques. This opens up opportunities for bugs and raises maintenance costs. By supporting static typing where possible, and dynamic typing where needed, Visual Basic delivers the best of both worlds to programmers.

- *Reduced cognitive load on programmers*. Features such as type inference, object initializers, and relaxed delegates greatly reduce code redundancy and the number of exceptions to the rules that programmers need to learn and remember or look up, with no impact on performance. Features such as dynamic interfaces support IntelliSense even in the case of late-binding, greatly improving discoverability over advanced features.

Although it may seem that the Visual Basic 9.0 list of new features is long, we hope the above themes will convince you that it is coherent, timely, and dedicated to making Visual Basic the world's finest programming system. We hope your imagination will be stimulated, too, and that you will join us in realizing that this is really just the beginning of even greater things to come.

Biography

Erik **Meijer**
Architect
Microsoft Corporation [http://www.microsoft.com]
Redmond
Washington
United States of America
http://www.research.microsoft.com/~emeijer/ [http://www.research.microsoft.com/~emeijer]

Erik Meijer is an architect in the WebData XML group at Microsoft where he works with the C# and Visual Basic teams on language and type-systems for data integration in programming languages. Prior to joining Microsoft he was an associate professor at Utrecht University and adjunct professor at the Oregon Graduate Institute. Erik is one of the designers of the Mondrian scripting language, standard functional programming language Haskell98, and Comega.

Amanda **Silver**
Program Manager
Microsoft Corporation [http://www.microsoft.com]
Redmond
Washington
United States of America

Amanda Silver is the program manager for the Visual Basic compiler.

Paul **Vick**
Technical Lead
Microsoft Corporation [http://www.microsoft.com]
Redmond
Washington
United States of America

Paul Vick is the technical lead for the Visual Basic language.