# Iterator and Async Methods in Visual Basic

The “Microsoft Visual Studio Async CTP” introduces *async methods* and *iterator methods* to Visual Basic, along with their four new contextual keywords: Async, Await, Iterator, Yield.

Async methods are used for long-running computations. This language specification does not aim to teach asynchrony – the best place for that is through watching [Anders Hejlsberg’s PDC 2010 talk](http://player.microsoftpdc.com/Session/1b127a7d-300e-4385-af8e-ac747fee677a), and then by working through the walkthrough and samples. Unfortunately there is not yet a large body of further training material: VB’s form of asynchrony was largely inspired by F# Async Workflows in Visual Studio 2010 and had not been widely used before then. I want to stress just one point:

**Asynchrony is NOT the same as “running on a background thread”**. In computing, as the word’s etymology suggests, asynchronous merely means “not [a-] at the same time [-synchronous]”. In other words you can invoke a function, and promptly get back a *placeholder* for the result, but the *actual* result of the function will be given to you not immediately but at some other later time. Background threads are one way to achieve asynchrony, but they’re not the only way, and are often not even the best way.

What we are delivering in this Async CTP is a *compositional* (easy) way to make asynchronous calls in the Visual Basic language.

This Async CTP also introduces iterator methods to VB. Iterator methods are deeply related to async methods. Because we are introducing both at the same time, we have been able to reflect their underlying similarity by giving them similar syntax and semantics. VB Iterator methods are largely the same as those that already exist in C#. The differences that there are, are all motivated for reasons of similarity with async methods:

1. VB iterator methods require an explicit Iterator modifier;
2. VB iterator methods use the Yield keyword instead of yield return, and they use VB’s existing Return and Exit Function statements instead of yield break;
3. the VB Yield statement is allowed in the Try body of an exception handler with a Catch clause; and
4. VB allows anonymous iterator methods (lambdas) as well as top-level iterator methods.

This specification for iterator and async methods is written as a series of drop-in-replacements for sections in the existing Visual Basic 10 Language Specification, normally found in  
C:\Program Files\Microsoft Visual Studio 10.0\VB\Specifications

*-- Lucian Wischik, VB Spec Lead, April 2010.*

# 6. Source Files and Namespaces

## 6.1 Program Startup and Termination

The program entry point method, which must always be named Main, cannot be an async method.

# 9. Type Members

## 9.2 Methods

### 9.2.1 Regular, Async and Iterator Method Declarations

There are two types of methods: subroutines, which do not return values, and functions, which do. The body and End construct of a method may only be omitted if the method is defined in an interface or has the MustOverride modifier. If no return type is specified on a function and strict semantics are being used, a compile-time error occurs; otherwise the type is implicitly Object or the type of the method's type character. The accessibility domain of the return type and parameter types of a method must be the same as or a superset of the accessibility domain of the method itself.

A **regular method** is one with neither Async nor Iterator modifiers. It may be a subroutine or a function. Section [10.1.1] details what happens when a regular method is invoked.

An **iterator method** is one with the Iterator modifier and no Async modifier. It must be a function, and its return type must be IEnumerator, or IEnumerable, or IEnumerator(Of T) for some T, or IEnumerable(Of T), and it must have no ByRef parameters. Section [10.1.3] details what happens when an iterator method is invoked.

An **async method** is one with the Async modifier and no Iterator modifier. It must either be a subroutine, or a function with return type Task or Task(Of T) for some T, and must have no ByRef parameters. Section [10.1.4] details what happens when an async method is invoked.

It is a compile-time error if a method is not one of these three kinds of method.

**Note.** Async and Iterator are not reserved words.

MethodDeclaration ::=  
 SubDeclaration |  
 MustOverrideSubDeclaration |  
 FunctionDeclaration |  
 MustOverrideFunctionDeclaration

ProcedureModifier ::=  
 AccessModifier |  
 Shadows |  
 Shared |  
 Overridable |  
 NotOverridable |  
 Overrides |  
 Overloads |  
 Partial |  
 Iterator |  
 Async

## 9.7 Properties

PropertyModifier ::=  
 AutoPropertyModifier |  
 Default |  
 ReadOnly |  
 WriteOnly |  
 Iterator

### 9.7.5 Iterator Properties

An *iterator property* is a property with the Iterator modifier. It must have an explicit Get accessor, and its type must be IEnumerator, or IEnumerable, or IEnumerator(Of T) for some T, or IEnumerable(Of T).

The Get accessor of an iterator property is interpreted as an iterator method.

ReadOnly Iterator Property Primes As IEnumerable(Of Integer)  
 Get  
 Yield 1  
 Yield 2  
 Yield 3  
 Yield 5  
 End Get  
End Property

# 10 Statements

## 10.1 Control flow

*Control flow* is the sequence in which statements and expressions are executed. The order of operations depends on the particular statement or expression.

For example, when evaluating an addition operation [Section 11.3], first the left operand is evaluated, then the right operand, and then the operator itself. Blocks are executed [Section 10.1.2] by first executing their first immediate substatement, and then proceeding one by one through the statements of the block.

Implicit in this ordering is the concept of a *control point*, which is the next operation to be executed. When a method is invoked (or “called”), we say it creates an *instance* of the method. Each method instance has its own copy of the method’s parameters and local variables, and its own control point.

### 10.1.1 Regular methods

Here is an example of a regular method.

Function Test() As Integer  
 Console.WriteLine("hello")  
 Return 1  
End Sub

Dim x = Test() ' invokes the function, prints "hello", assigns 1 to x

When a regular method is invoked,

1. It first creates an instance of the method specific to that invocation. This instance includes a copy of all parameters and local variables of the method.
2. It then initializes all of its parameters to the supplied values, and initializes all of its local variables to the default values of their types.
3. In the case of a Function, it also initializes an implicit local variable called the *function return variable* whose name is the function’s name, whose type is the return type of the function and whose initial value is the default of its type.
4. It then sets the method instance’s control point at the first statement of the method body, and immediately starts to execute the method body from there [Section 10.1.2].

When control flow exits the method body normally - through reaching the End Function or End Sub that mark its end, or through an explicit Return or Exit statement - control flow returns to the caller of the method instance. If there is a function return variable, the result of the invocation is the final value of this variable.

When control flow exits the method body through an unhandled exception, that exception is propagated to the caller.

After control flow has exited, there are no longer any live references to the method instance or its copy of local variables or parameters, and so they may be garbage collected.

### 10.1.4 Iterator Methods

Iterator methods are used as a convenient way to generate a sequence, one which can be consumed by the For Each statement. Iterator methods use the Yield statement [Section 10.15] to provide elements of the sequence. For example:

Iterator Function Test() As IEnumerable(Of Integer)  
 Console.WriteLine("hello")  
 Yield 1  
 Yield 2  
End Function

Dim en = Test()  
For Each x In en ' prints "hello" before the first x  
 Console.WriteLine(x) ' prints "1" and then "2"  
Next

When an iterator method is invoked whose return type is IEnumerator(Of T),

1. It first creates an instance of the iterator method specific to that invocation. This instance includes a copy of all parameters and local variables of the method.
2. It then initializes all of its parameters to the supplied values, and initializes all of its local variables to the default values of their types.
3. It also initializes an implicit unnamed local variable called the *iterator current variable* whose type is T and whose initial value is the default of its type.
4. It then sets the method instance’s control point at the first statement of the method body.
5. It then creates an *iterator object* associated with this method instance. The iterator object implements the declared return type and has behavior as described below.
6. Control flow is then resumed *immediately* in the caller, and the result of the call is the iterator object. Note that this transfer is done without exiting the iterator method instance, and does not cause finally handlers to execute. The method instance is still referenced by the iterator object, and will not be garbage collected so long as there exists a live reference to the iterator object.

When the iterator object’s Current property is accessed, it returns the *current variable* of the invocation.

When the iterator object’s MoveNext method is invoked, it does not create a new method instance. Instead it uses the existing method instance (and its control point and local variables and parameters) that was created when the iterator method was first invoked. Control flow resumes execution at the control point of that method instance, and proceeds through the body of the iterator method as normal.

When the iterator object’s Dispose method is invoked, it again re-uses the existing method instance that was created when the iterator method was first invoked. Control flow resumes at the control point of that method instance, but then immediately behaves as if an Exit Function statement were the next operation.

The above descriptions of behavior for invocation of MoveNext or Dispose on an iterator object only apply if all previous invocations of MoveNext or Dispose on that iterator object have already returned to their callers.

When control flow exits the iterator method body normally - through reaching the End Function that mark its end, or through an explicit Return or Exit statement – it must have done so in the context of an invocation of MoveNext or Dispose function on an iterator object to resume the iterator method instance, and it will have been using the method instance that was created when the iterator method was first invoked. The control point of that instance is left at the End Function statement, and control flow resumes in the caller; and if it had been resumed by a call to MoveNext then the value False is returned to the caller.

When control flow exits the iterator method body through an unhandled exception, then the exception is propagated to the caller.

As for the other possible return types of an iterator function,

1. When an iterator method is invoked whose return type is IEnumerable(Of T) for some T, it first creates an instance – specific to that invocation of the iterator method – of all parameters in the method, and initializes them with the supplied values. It then returns an object which implements the return type. Should this object’s GetEnumerator method be called, it creates an instance – specific to that invocation of GetEnumerator – of all parameters and local variables in the method. It initializes the parameters to the values already saved, and proceeds as above.
2. When an iterator method is invoked whose return type is the non-generic interface IEnumerator, it behaves exactly as for IEnumerator(Of Object).
3. When an iterator method is invoked whose return type is the non-generic interface IEnumerable, it behaves exactly as for IEnumerable(Of Object).

### 10.1.5 Async methods

Async methods are a convenient way to do long-running work without e.g. blocking the UI of an application. Async is short for *Asynchronous* - it means that the caller of the async method will resume execution promptly, but the eventual completion of the async method’s instance may happen at some later time in the future. By convention async methods are named with the suffix “Async”.

Async Function TestAsync() As Task(Of String)  
 Console.WriteLine("hello")  
 Await Task.Delay(100)  
 Return "world"  
End Function

Dim t = TestAsync() ' prints "hello"  
Console.WriteLine(Await t) ' prints "world"

When an async method is invoked

1. It first creates an instance of the async method specific to that invocation. This instance includes a copy of all parameters and local variables of the method.
2. It then initializes all of its parameters to the supplied values, and initializes all of its local variables to the default values of their types.
3. In the case of an async method with return type Task(Of T) for some T, it also initializes an implicit unnamed local variable called the *task return variable* whose type is T and whose initial value is the default of T.
4. If the async method is a Function with return type Task or Task(Of T) for some T, it then implicitly creates an object of that type associated with the current invocation. This is called an *async object* and represents the future work that will be done by executing the instance of the async method. When control resumes in the caller of this async method instance, the caller will receive this async object as the result of its invocation.
5. If instead the async method is a Sub, and if SynchronizationContext.Current is not Nothing, then it calls OperationStarted() on that synchronization context.
6. It then sets the invocation’s control point at the entry of the async method body, and immediately starts to execute the method body from there [Section 10.1.2].

As detailed in [Section 11.25], execution of an Await expression has the ability to cause the method instance’s control point to be suspended at the Await expression while control flow is resumed elsewhere; and control flow can later resume at the same instance’s control point through invocation of a *resumption delegate*. Note that this suspension is done without exiting the async method, and does not cause finally handlers to execute. The method instance is still referenced by the resumption delegate and the Task result (if any), and will not be garbage collected so long as there exists a live reference to either.

Let the *current caller* of the method instance be either the original invocation, or invocation of the resumption delegate, whichever is more recent. Either way, when an async method instance is not suspended, it is executing within the method instance created by the original invocation of the async method.

Annotation

It is helpful to imagine the statement “Dim x = Await WorkAsync()” approximately as syntactic shorthand for the following:

Dim $temp = WorkAsync().GetAwaiter()  
 If Not $temp.IsCompleted Then  
 $temp.OnCompleted(resumptionDelegate)  
 Return [task]  
 CONT: ' invocation of *resumptionDelegate* will resume here  
 End If  
 Dim x = $temp.GetResult()

When control flow exits the async method body - through reaching the End Sub or End Function that mark its end, or through an explicit Return or Exit statement, or through an unhandled exception - the instance’s control point is set to the end of the method. Behavior then depends on the return type of the async method.

1. In the case where the return type is Task:
   1. If control flow exits through an unhandled exception, then the async object’s status is set to TaskStatus.Faulted, and its Exception.InnerException property is set to the exception. Control flow resumes in the current caller.
   2. Otherwise, the async object’s status is set to TaskStatus.Completed. Control flow resumes in the current caller.
2. In the case where the return type is Task(Of T) for some T: the behavior is as above, except that in non-exception cases the async object’s Result property is also set to the final value of the task return variable.
3. In the case where the async method is a Sub: If the original synchronization context was not Nothing then it calls OperationCompleted on that synchronization context. Next, control flow resumes in the current caller. If control flow exited through an unhandled exception, this exception is propagated to the current caller.

Annotation

Mutable structures in general are considered bad practice, and they are not supported by async or iterator methods. In particular, each invocation of an async or iterator method in a structure will implicitly operate on a *copy* of that structure that is copied at its moment of invocation. Thus, for example,

Structure S  
 Dim x As Integer  
 Async Sub Mutate()  
 x = 2  
 End Sub  
 End Structure

Dim s As New S With {.x = 1}  
 s.Mutate()  
 Console.WriteLine(s.x) ’ prints “1”

## 10.10.1 Structured Exception Handling

## 10.10.1.1 Finally Blocks

A Finally block is always executed when execution leaves any part of the Try statement. No explicit action is required to execute the Finally block; when execution leaves the Try statement, the system will automatically execute the Finally block and then transfer execution to its intended destination. The Finally block is executed regardless of how execution leaves the Try statement: through the end of the Try block, through the end of a Catch block, through an Exit Try statement, through a GoTo statement, or by not handling a thrown exception.

Note that the Await expression in an async method, and the Yield statement in an iterator method, can cause flow of control to suspend in the async or iterator method instance and resume in some other method instance. However, this is merely a suspension of execution and does not involve exiting the respective async method or iterator method instance, and so does not cause Finally blocks to be executed.

## 10.10.1.2 Catch Blocks

Annotation

The filters in When clauses are normally evaluated prior to the exception being thrown. For instance, the following code will print “Filter, Finally, Catch”.

Sub Main()  
 Try  
 Foo()  
 Catch ex As Exception When F()  
 Console.WriteLine("Catch")  
 End Try  
 End Sub

Sub Foo()  
 Try  
 Throw New Exception  
 Finally  
 Console.WriteLine("Finally")  
 End Try  
 End Sub

Function F() As Boolean  
 Console.WriteLine("Filter")  
 Return True  
 End Function

However, Async and Iterator methods cause all finally blocks inside them to be executed prior to any filters outside. For instance, if the above code had “Async Sub Foo()”, then the output would be “Finally, Filter, Catch”.

## 10.10.2 Unstructured Exception-Handling Statements

Unstructured error handling statements are not allowed in iterator or async methods.

## 10.11 Branch statements.

Branch statements modify the flow of execution in a method. There are six branch statements:

1. A GoTo statement... [as before]
2. An Exit statement causes the control point to jump to the next statement after the end of the immediately containing block statement of the specified kind. If the Exit statement is not contained within the kind of block specified in the statement, it is a compile-time error. If the block is the method block itself, then control flow exits the method as described in Sections [10.1.1, 10.1.3, 10.1.4].
3. A Continue statement... [as before]
4. A Stop statement... [as before]
5. An End statement... [as before]
6. A Return statement with no expression is equivalent to an Exit Sub or Exit Function statement. A Return statement with an expression is only allowed in a regular method that is a function, or in an async method that is a function with return type Task(Of T) for some T. Its expression must be classified as a value and its type must be implicitly convertible to the *function return variable* (in the case of regular methods) or to the *task return variable* (in the case of async methods). Its behavior is to evaluate its expression, then store it in the return variable, then execute an implicit Exit Function statement.

## 10.14 Await Statements

AwaitStatement ::= AwaitOperatorExpression StatementTerminator

An await statement has the same syntax as an await expression, is allowed only in methods that also allow await expressions, and has the same behavior as an await expression.

However, it may be classified as either a value or void. Any value resulting from evaluation of the await expression is discarded.

## 10.15 Yield Statements

YieldStatement ::= Yield *Expression* StatementTerminator

Yield is a reserved word if the immediately enclosing method or lambda expression in which it appears has an Iterator modifier, and if the Yield appears after that Iterator modifier; it is unreserved elsewhere. It is also unreserved in preprocessor directives. The Yield statement is only allowed in the body of a method or lambda expressions where it is a reserved word. Furthermore, within the method or lambda a yield statement may not occur inside the body of a Catch or Finally block, nor inside the body of a SyncLock statement.

*[In the Async CTP, Yield is a reserved word everywhere]*

The yield statement takes a single expression which must be classified as a value and whose type is implicitly convertible to the type of the *iterator current variable* of its enclosing iterator method.

Control flow only ever reaches a Yield statement when the MoveNext method is invoked on an iterator object. (This is because an iterator method instance only ever executes its statements due to the MoveNext or Dispose methods being called on an iterator object; and the Dispose method will only ever execute code in Finally blocks,where Yield is not allowed).

When a Yield statement is executed, its expression is evaluated and stored in the *iterator current variable* of the iterator method instance associated with that iterator object. The value True is returned to the invoker of MoveNext, and the control point of this instance stops advancing until the next invocation of MoveNext on the iterator object.

# 11 Expressions

## 11.1 Expression Classification

* Void. This occurs when an expression is an invocation of a subroutine, or an await expression with no result. An expression classified as void is only valid in the context of an invocation statement or an await statement.

### Expression Reclassification

4.3. An async or iterator lambda method may only be interpreted as the argument to a delegate-construction expression, if the delegate has no ByRef parameters.

## 11.8 Invocation Expressions

### 11.8.1 Overloaded Method Resolution

*[This has not yet been implemented in the Async CTP. However, it will have the following characteristics...]*

|  |  |
| --- | --- |
| ' Given these two overloads:  Sub f(a As Func(Of Task(Of Integer))) Sub f(a As Func(Of Task(Of Double)))  ' this will bind to the first:  f(Async Function()  Return 1  End Function) | *Overload resolution currently looks at the “delegate relaxation level” of return statements in a lambda, as compared to the target delegate’s return type.*  *With async function lambdas, it requires a target delegate of type Task or Task(Of T) for some T. It will look at the relaxation level of return statements as compared to that T.*  *With iterator function lambdas, it will require a target delegate of type IEnumerable or IEnumerator or IEnumerable(Of T) for some T or IEnumerator(Of T). It will look at the relaxation level of yield statements as compared to that T.* |
| ' Given these two overloads:  Sub g(i As Func(Of IEnumerable(Of Integer))) Sub g(f As Func(Of IEnumerable(Of Double)))  ' this will bind to the first:  g(Iterator Function()  Yield 1  End Function) |

## 11.12 Operator Expressions

### 11.12.1 Operator Precedence and Associativity

*[In the Async CTP, Await has lowest precedence rather than highest. Use parentheses to force different order. However, not that the Async CTP does not preserver order of execution, and so use of Await in compound expressions is discouraged.]*

|  |  |
| --- | --- |
| Category | Operators |
| Primary | All non-operator expressions |
| Exponentiation | ^ |
| Unary negation | +, - |
| Await | Await |
| Multiplicative | \*, / |
| Integer division | \ |
| Modulus | Mod |
| Additive | +, - |
| Concatenation | & |
| Shift | <<, >> |
| Relational | =, <>, <, >, <=, >=, Like, Is, IsNot |
| Logical NOT | Not |
| Logical AND | And, AndAlso |
| Logical OR | Or, OrElse |
| Logical XOR | Xor |

### 11.8.5 Type Argument Inference

*[This has not yet been implemented in the Async CTP. However, it will have the following characteristics…]*

|  |  |
| --- | --- |
| ' Given this method: Sub f(Of T)(f As Func(Of Task(Of T)))  ' this call will infer T=Integer: f(Async Function()  Return 1  End Function) | *Generic type inference currently, if asked to match a regular lambda expression argument to a delegate parameter type with return type “T”, will infer a hint for T based on the return type of the lambda expression.*  *With async lambda expressions, when they are matched to a delegate parameter type with return type “Task(Of T)”, they will infer a hint for T based on the return type of the lambda expression.*  *Iterator lambda expressions will work similarly with IEnumerable(Of T) and IEnumerator(Of T)..* |
| ' Given this method: Sub g(Of T)(f As Func(Of IEnumerable(Of T)))  ' this call will infer T=Integer: g(Iterator Function()  Yield 1  End Function) |

## 11.20 Lambda Expressions

LambdaExpression ::=  
 SingleLineLambda |  
 MultiLineLambda

SingleLineLambda ::=  
 [ LambdaModifier + ] Function [ OpenParenthesis [ ParametertList ] CloseParenthesis ] Expression |  
 [ LambdaModifier + ] Sub [ OpenParenthesis [ ParametertList ] CloseParenthesis ] Statement

MultiLineLambda ::=  
 MultiLineFunctionLambda |  
 MultiLineSubLambda

MultiLineFunctionLambda ::=  
 [ LambdaModifier+ ] Function [ OpenParenthesis [ ParametertList ] CloseParenthesis ] [ As TypeName ] LineTerminator  
 Block  
 End Function

MultiLineSubLambda ::=  
 [ LambdaModifier+ ] Sub [ OpenParenthesis [ ParametertList ] CloseParenthesis ] LineTerminator  
 Block  
 End Sub

LambdaModifier ::=  
 Async |  
 Iterator

A lambda expression begins with the optional modifiers Async or Iterator, followed by the keyword Function or Sub and a parameter list.

A **regular lambda expression** is one with neither Async nor Iterator modifiers.

An **iterator lambda expression** is one with the Iterator modifier and no Async modifier. It must be function. When it is reclassified to a value, it can only be reclassified to a value of delegate type whose return type is IEnumerator, or IEnumerable, or IEnumerator(Of T) for some T, or IEnumerable(Of T), and which has no ByRef parameters.

An **async lambda expression** is one with the Async modifier and no Iterator modifier. An async sub lambda may only be reclassified value of sub delegate type with no ByRef parameters. An async function lambda may only be reclassified to a value of function delegate type whose return type is Task or Task(Of T) for some T, and which has no ByRef parameters.

*[ In the Async CTP, single-line async function lambdas are not yet implemented. ]*

Multi-line Function lambda expressions can declare a return type but cannot put attributes on it. If a multi-line Function lambda expression does not declare a return type but the return type can be inferred from the context in which the lambda expression is used (see Section 11.1.1 on Expression Reclassification), then that return type is used. Otherwise the return type of the function is calculated as follows.

1. In a regular lambda expression, the return type is the dominant type of the expressions in all the Return statements in the statement block.
2. In an iterator lambda expression, the return type is IEnumerable(Of T) where T is the dominant type of the expressions in all the Yield statements in the statement block.
3. In an async lambda expression, the return type is Task(Of T) where T is the dominant type of the expressions in all the Return statements in the statement block.

For example:

Function f(min As Integer, max As Integer) As IEnumerable(Of Integer)  
 If min > max Then Throw New ArgumentException()  
 Dim x = Iterator Function()  
 For i = min To max  
 Yield i  
 Next  
 End Function

' infer x to be a delegate with return type IEnumerable(Of Integer)  
 Return x()  
End Function

In all cases, if there are no Return (respectively Yield) statements, or if there is no dominant type among them, and strict semantics are being used, a compile-time error occurs; otherwise the dominant type is implicitly Object. Note that the return type is calculated from all Return (respectively Yield) statements, even if they are not reachable.

## 11.25 Await Operator

AwaitOperatorExpression ::= Await *Expression*

Await is a reserved word if the immediately enclosing method or lambda expression in which it appears has an Async modifier, and if the Await appears after that Async modifier; it is unreserved elsewhere. It is also unreserved in preprocessor directives. The await operator is only allowed in the body of a method or lambda expressions where it is a reserved word. Furthermore, within the method or lambda, an await expression may not occur inside the body of a Catch or Finally block, nor inside the body of a SyncLock statement, nor inside a query expression.

*[In the Async CTP, Await is a reserved word everywhere]*

The await operator takes a single expression which must be classified as a value and whose type must be an *awaitable* type, or Object. If its type is Object then all processing is deferred until run-time. A type C is said to be awaitable if all of the following are true:

1. C contains an accessible instance or extension method named GetAwaiter which has no arguments and which returns some type E;
2. E contains a readable instance property named IsCompleted which takes no arguments;
3. E contains an accessible instance Sub named OnCompleted which takes a single ByVal argument of type System.Action and which has return type Boolean;
4. E contains an accessible instance or extension method named GetResult which takes no arguments.

If GetResult was a Sub, then the await operator is classified as void. Otherwise, the await operator is classified as a value and its type is the return type of the GetResult method.

Following is an example of a class that can be awaited:

Class Task(Of T)  
 Function GetAwaiter() As TaskAwaiter(Of T)  
 Return New TaskAwaiter With {.m\_Task = Me}  
 End Function  
End Class

Structure TaskAwaiter(Of T)  
 Friend m\_Task As Task(Of T)

ReadOnly Property IsCompleted As Boolean  
 Get  
 Return m\_Task.IsCompleted  
 End Get  
 End Property

Sub OnCompleted(r As Action)  
 Dim sc = SynchronizationContext.Current  
 If sc Is Nothing Then  
 m\_Task.ContinueWith(Sub() r())  
 Else  
 m\_Task.ContinueWith(Sub() sc.Post(Sub() r(), Nothing))  
 End If  
 End Sub

Function GetResult() As T  
 If m\_Task.IsCanceled Then Throw New TaskCanceledException(m\_Task)  
 If m\_Task.IsFaulted Then Throw m\_Task.Exception.InnerException  
 Return m\_Task.Result  
 End Function  
End Structure

Annotation

Library authors are recommended to follow the pattern that they invoke the continuation delegate on the same SynchronizationContext as their OnCompleted was itself invoked on.

When control flow reaches an Await operator, it calls GetAwaiter on its operand. It then calls BeginAwait on the result, passing a *resumption delegate* (see Section [10.1.3]) associated with the current instance of the async method.

If BeginAwait returned False, then it immediately calls EndAwait. If EndAwait was a function, then the value of the await expression is the return value of this function.

If BeginAwait returned True, then the control point of the current async method instance remains suspended at the Await expression. However, control flow resumes in the *current caller* [defined in Section 10.1.5]. If later the resumption delegate is invoked, control flow will resume at the control point of the same async method instance where it will call EndAwait and continue as above.