VB.NET for Java developers

Erstellt und zuletzt geändert von Frederik Zipp vor Kurzem

Java	**VB.NET**
Naming conventions	
• Classes, interfaces, enums: *CamelCase* • Methods, local variables, fields: *mixedCamelCase* • Constants, enum values: *UPPER_CASE*	• Classes, structures, namespaces: *CamelCase* • Interfaces: *ICamelCase* • Private methods, local variables, fields: *mixedCamelCase* • public methods, properties: *CamelCase* • Constants, enum values: *CamelCase*
Identifiers in Java are *case-sensitive*.	Identifiers in VB.NET are *case-**in**sensitive*. It's advised to follow the conventions to ensure interoperability with C# and other CLI-based languages.
Code organization	
**.java*	**.vb*
Statement separator: `semicolon;`	No statement separator. To continue a line over a linebreak, append an underscore ('_'), Multiple statements in one line are separated by a colon (':')

`import java.io.*;`
 	
`package org.foo.bar;`
 	
`// ...`
	`Imports System.IO`
	
	`Namespace Foo.Bar`
	`    ' ...`
	`End Namespace`

It is not possible to access the default package from within another package.	To access the root namespace from within another namespace, prepend `Global.`
Comments	
`// ...`	`' ...`
`/* ... */`	*(n/a)*

`/**`
` * ...`
` */`
	`''' <summary>`
	`''' ...`
	`''' </summary>`

Basic data types	

`int`
`short`
`long`
`boolean`
`char`
`byte`
`double`
`float`
`(n/a)`
`java.math.BigInteger`
	`Integer`
	`Short`
	`Long`
	`Boolean`
	`Char`
	`SByte`
	`Double`
	`Single`
	`Decimal (128 bit)`
	`System.Numerics.BigInteger`

	Additional unsigned types: `UInteger`, `UShort`, `ULong`, `Byte`

`String`
`Object`
`Class`
	`String`
	`Object`
	`Type`

Literals	

`null`
`true, false`
`"abc"`
`'D'`
`0xFF`
`2.9f`
`3.14159265`
`123456L`
	`Nothing`
	`True, False`
	`"abc"`
	`"D"c`
	`&HFF`
	`2.9!`
	`3.14159265`
	`123456L`

`"\t\r\n"`	No escape sequences in strings. Concatenate the predefined constants `vbCr`, `vbLf`, `vbNewLine`, `vbTab`, ... to your strings.
Variable declaration	

`Foo foo;`
`Foo foo = new Foo();`
 	
`int i = 42;`
	`Dim foo As Foo`
	`Dim foo As Foo = new Foo()`
	`Dim foo As new Foo() ' slightly shorter`
	`' with type inference per "Option Infer On", Object otherwise`
	`Dim foo = new Foo()`
	
	`' without type inference`
	`Dim i As Integer = 42`
	`' with type inference per "Option Infer On", Object otherwise`
	`Dim i = 42`

Constants	
`final` `static final`	`ReadOnly        ' Can be set in the constructor` `Const`
Arrays	

`int[] numbers;`
 	
`int[][] numbers;`
 	
`int[][][] numbers;`
 	
`int[] numbers = new int[6];`
 	
`numbers[0]`
`numbers[5]`
 	
`new int[] {1, 2, 4, 8}`
	`Dim numbers() As Integer`
	`' "jagged" multi-dimensional array ("array of arrays")`
	`Dim numbers()() As Integer`
	`' "jagged" multi-dimensional array ("array of arrays of arrays")`
	`Dim numbers()()() As Integer`
	`' "rectangular" multi-dimensional array (a coherent box)`
	`Dim numbers(,) As Integer`
	`' "rectangular" multi-dimensional array (a coherent cube)`
	`Dim numbers(,,) As Integer`
	
	`' ATTENTION: 5 specifies the last valid array index,`
	`' not the number of elements`
	`Dim numbers(5) As Integer`
	
	`numbers(0)`
	`numbers(5)`
	
	`New Integer() {1, 2, 4, 8}`
	
	`' with type inference (results in an array of Doubles,`
	`' because Double is the dominant type):`
	`{1.5, 2, 9.9, 18}`

Operators	
Arithmetic	

`+, -, *`
`/ (float)`
`/ (int)`
`%`
	`+, -, *`
	`/`
	`\`
	`Mod`

Java	VB.NET
`Math.pow(x, y)`	`x ^ y`
Assignment	

`=`
`+=, -=, *=`
`/= (float)`
`/= (int)`
`(n/a)`
`++, --`
	`=`
	`+=, -=, *=`
	`/=`
	`\=`
	`<<=, >>=`
	`(n/a)`

String concatenation	
`+` `+=`	`&` `&=`
Logical	

`&&`
`||`
`!`
	`AndAlso (short-circuit evaluation, like Java), And`
	`OrElse (short-circuit evaluation, like Java), Or`
	`Not`

Bitwise	

`&, |`
`~`
`^`
`<<, >>`
`>>>`
	`And, Or`
	`Not`
	`Xor`
	`<<, >>`
	`(n/a)`

Comparison	

`>, <`
`==`
`!=`
`a.equals(b)`
`!a.equals(b)`
`==`
`!=`
	`>, <`
	`=  ' Values`
	`<>  ' Values`
	`=  ' Object instances`
	`<>  ' Object instances`
	`Is  ' Referencial equality of objects`
	`IsNot  ' Referencial inequality of objects`

Conditional	
`condition ? a : b`	`If(condition, a, b)`
Instantiation	
`new` `o instanceof Foo`	`New` `TypeOf o Is Foo`
Method references	

`ContainingClass::staticMethodName`
`containingObject::instanceMethodName`
`this::instanceMethodName`
	`AddressOf ContainingClass.StaticMethodName`
	`AddressOf containingObject.InstanceMethodName`
	`AddressOf InstanceMethodName`

Casting and conversion	

`(Foo) bar`
`bar instanceof Foo ? (Foo) bar : null`
`*.valueOf()`
	`DirectCast(bar, Foo)`
	`TryCast(bar, Foo)  ' falls back to "Nothing"`
	`CType(bar, Foo)  ' conversion`
	
	`' Additional convenience functions of "CType" for`
	`' standard types:`
	`CBool(bar), CByte(bar), CChar(bar), CDate(bar), CDbl(bar),`
	`CDec(bar), CInt(bar), CLng(bar), CObj(bar), CSByte(bar),`
	`CShort(bar), CSng(bar), CStr(bar), CUInt(bar), CULng(bar),`
	`CUShort(bar)`

Control structures	
Loops	

`for (Foo foo : bar) {`
`}`
 	
`for (int i = 1; i <= n; i++) {`
`}`
 	
`for (int i = n; i >= 0; i -= 2) {`
`}`
 	
`while (condition) {`
`}`
 	
`do {`
`} while (condition);`
 	
`do {`
`} while (!condition);`
 	
`continue`
`break`
	`For Each foo In bar`
	`Next`
	
	`For i As Integer = 1 To n`
	`Next`
	
	`For i As Integer = n To 0 Step -2`
	`Next`
	
	`While condition`
	`End While`
	
	`Do`
	`Loop While condition`
	
	`Do`
	`Loop Until condition`
	
	`Continue For, Continue Do, Continue While, ...`
	`Exit For, Exit Do, Exit While, ...`

Conditional statements	

`if (condition) {`
`} else if (condition) {`
`} else {`
`}`
	`If condition Then  ' Then is optional in a multi-line If`
	`ElseIf condition Then`
	`Else`
	`End If`

Case differentiation	

`switch (number) {`
`case 1:`
`  // ...`
`  break;`
`default:`
`  // ...`
`}`
	`Select Case number`
	`  Case 1 To 5`
	`  Debug.WriteLine("Between 1 and 5, inclusive")`
	`  Case 6, 7, 8`
	`  Debug.WriteLine("Between 6 and 8, inclusive")`
	`  Case 9 To 10`
	`  Debug.WriteLine("Equal to 9 or 10")`
	`  Case Else`
	`  Debug.WriteLine("Not between 1 and 10, inclusive")`
	`End Select`
	(no "fallthrough")

Exception handling	
Throwing	
`throw new Exception("")`	`Throw New Exception("")`
Catching	

`try {`
`} catch (Exception e) {`
`} finally {`
`}`
	`Try`
	`Catch e As Exception`
	`Finally`
	`End Try`

Popular exception types	

Java	VB.NET

`IllegalArgumentException`
`NullPointerException`
`UnsupportedOperationException`
`IOException`
	`ArgumentException, ArgumentNullException, ArgumentOutOfRangeException`
	`NullReferenceException`
	`NotSupportedException, NotImplementedException`
	`IOException`

Resource management	
`try (Resource resource = new Resource()) {` `}`	`Using resource As New Resource()` `End Using`
Resource has to implement the interface *AutoCloseable*.	Resource has to implement the interface *IDisposable*.
Assertions	
`assert`	`Debug.Assert(),Trace.Assert()`
Type definitions	
`class`	`Class ... End Class`
`interface`	`Interface ... End Interface`
`extends`	`Inherits` (has to be in the next line or separated by : (colon))
`implements`	`Implements` (has to be in the next line or separated by : (colon))
`enum`	`Enum ... End Enum` (no constructors or methods)
	`Module ... End Module` (like a class with only static methods)
	`Structure ... End Structure` (value type: copy-on-assignment, no inheritance)
`final`	`NotInheritable`
`abstract (class)`	`MustInherit`
	`Partial` (a class spanning multiple files)
`this`	`Me`
`super`	`MyBase`
`Foo.class`	`GetType(Foo)`
`foo.getClass()`	`foo.GetType()`
Type parameters	
`Foo<T>` `Foo<K, V>`	`Foo(Of T)` `Foo(Of K, V)`
Covariance und Contravariance:	

use-site variance
`Foo<? extends Bar>`
`Foo<? super Bar>`
	declaration-site variance
	`Foo(Of Out Bar)`
	`Foo(Of In Bar)`

Constructors	

`public Foo() {`
`  super();`
`}`
 	
`public Foo() {`
`  this(42);`
`}`
	`Public Sub New()`
	`  MyBase.New()`
	`End Sub`
	
	`Public Sub New()`
	`  Me.New(42)`
	`End Sub`

Methods	
Visibility	

`public`
`private`
`protected`
`(default)`
	`Public`
	`Private`
	`Protected`
	`Friend`

Modifiers	

`abstract`
`static`
`final`
`(default)`
 	
`@Override`
	`MustOverride`
	`Shared`
	`NotOverridable (default)`
	`Overridable`
	
	`Overrides`

Methods with return value	

`public int name(double a, String b) {`
`  return 1;`
`}`
	`Public Function Name(ByVal a As Double, ByVal b As String) As Integer`
	`  Return 1`
	`End Function`

	ByVal correlates to the parameter semantics of Java, out-parameters (that don't exist in Java) can be declared by using `ByRef`. If nothing is explicitly specified, `ByVal` is the default since VB.NET. In VB6, the default was `ByRef`. It's best practice to explicitly specify both keywords, according to Microsoft.
Methods without return value ("procedures")	
`public void bla() {` `}`	`Public Sub Bla()` `End Sub`
Calling a method or a constructor without parameters	
`foo.bar()` `new Foo()`	`foo.Bar()  or shorter:  foo.Bar` `New Foo()  or shorter:  New Foo`
Varargs	

`...`
 	
`public double calcSum(double... args) {`
`}`
	`ParamArray`
	
	`Public Function CalcSum(ByVal ParamArray args() As Double) As Double`
	`End Function`

Optional parameters with default values	
(n/a)	`Public Function MyFun(ByVal s As String, Optional ByVal b As Boolean = False) As Integer` `End Function`
Lambdas	

`(x) -> x + 1`
 	
`(x) -> {`
`  return x + 2;`
`}`
 	
`Function<T, R>`
	`Function(x) x + 1`
	
	`Function(x)`
	`  Return x + 2`
	`End Function`
	
	`Func(Of T, TResult)`

Java	VB.NET

`Supplier<T>`
`Consumer<T>`
`Predicate<T>`
`(n/a)`
	`Func(Of TResult)`
	`Action(Of T)`
	`Predicate(Of T)` alternatively: `Func(Of T, Boolean)`
	`Func(Of T1, T2, T3, TResult)`

Properties (getter and setter methods)	
Reading and writing	

`public class Foo {`
`  private int bar;`
 	
`  public int getBar() {`
`  return this.bar;`
`  }`
 	
`  public void setBar(int bar) {`
`  this.bar = bar;`
`  }`
`}`
	`' shortened form (implemented automatically):`
	
	`Public Class Foo`
	`  Public Property Bar As Integer`
	`End Class`
	
	`' long form (allows for custom getter and setter):`
	
	`Public Class Foo`
	`  Private bar As Integer`
	
	`  Public Property Bar() As Integer`
	`  Get`
	`  Return bar`
	`  End Get`
	`  Set(ByVal value As Integer)`
	`  bar = value`
	`  End Set`
	`  End Property`
	`End Class`

Read-only	

`public class Foo {`
`  private int bar;`
 	
`  public int getBar() {`
`  return this.bar;`
`  }`
`}`
	`Public Class Foo`
	`  Private bar As Integer`
	
	`  Public ReadOnly Property Bar() As Integer`
	`  Get`
	`  Return bar`
	`  End Get`
	`  End Property`
	`End Class`

Write-only	

`public class Foo {`
`  private int bar;`
 	
`  public void setBar(int value) {`
`  this.bar = value;`
`  }`
`}`
	`Public Class Foo`
	`  Private bar As Integer`
	
	`  Public WriteOnly Property Bar() As Integer`
	`  Set(ByVal value As Integer)`
	`  bar = value`
	`  End Set`
	`  End Property`
	`End Class`

Anonymous types	

	`Dim bob = New With {.Name = "Uncle Bob", .Age = 42}`
	`Dim bob = New With {Key .Name = "Uncle Bob", .Age = 42}`
	`  ' Key properties are regarded in Equals`

Object initialisation	

`Person bob = new Person();`
`bob.setAge(42);`
`bob.setName("Bob");`
	`Dim bob As New Person {.Age = 42, .Name = "Bob"}`
	
	`Dim bob As New Person`
	`With bob`
	`  .Age = 42`
	`  .Name = "Bob"`
	`End With`

Object methods	

`.hashCode()`
`.equals(o)`
`.toString()`
	`.GetHashCode()`
	`.Equals(o)`
	`.ToString()`
	Guidelines for Implementing Equals and the Equality Operator

Common interfaces	

`Comparable`
`Comparator`
`Closeable`
`Serializable`
	`IComparable`
	`IComparer`
	`IDisposable`
	`ISerializable`

Collections	

`Iterable<T>`
`Iterator<T>`
`.iterator()`
`Collection<T>`
`List<T>`
`ArrayList<T>`
`LinkedList<T>`
`Set<T>`
`HashSet<T>`
`HashMap<K, V>`
	`IEnumerable(Of T)`
	`IEnumerator(Of T)`
	`.GetEnumerator()`
	`ICollection(Of T)`
	`IList(Of T)`
	`List(Of T)`
	`LinkedList(Of T)`
	`ISet(Of T)`
	`HashSet(Of T)`
	`Dictionary(Of K, V)`

Collection initialisation	
	`New Dictionary(Of Integer, String) From {{0, "Sunday"}, {1, "Monday"}}` `New List(Of String) From {"Sunday", "Monday"}`
Higher-order functions and functional operations	

`// On Stream<T>`
`.anyMatch(it -> condition)`
`.allMatch(it -> condition)`
`.map(x -> result)`
`.flatMap(x -> result)`
`.filter(it -> condition)`
`.reduce(a, b -> result)`
`// ...`
	`' On IEnumerable(Of T)`
	`.Any(Function(it) condition)`
	`.All(Function(it) condition)`
	`.Select(Function(x) result)`
	`.SelectMany(Function(x) result)`
	`.Where(Function(it) condition)`
	`.Aggregate(Function(a, b) result)`
	`' ...`

	Additional LINQ support: `Dim customersForRegion = From cust In customers Where cust.Region = region`
Console output	
`System.out.println()`	`System.Console.WriteLine()`
Threads	
java.lang.Thread	*System.Threading.Thread*

`Thread thread = new Thread(new Runnable() {`
`  @Override`
`  public void run() {`
`  // do something`
	`Private Shared Sub DoWork()`
	`  ' do something`
	`End Sub`

Java	VB.NET

`    }`
`});`
`thread.start();`
	`Dim thread As New Thread(AddressOf DoWork)`
	
	`thread.Start()`

Synchronisation	
`synchronized (obj) {` `}`	`SyncLock obj` `End SyncLock`
`volatile`	volatile equivalent in VB.NET
Additional popular types	

`java.lang.StringBuilder`
`java.util.Date`
`java.io.File`
`java.io.InputStream, OutputStream`
	`System.Text.StringBuilder`
	`System.DateTime`
	`System.IO.File (statische Methoden)`
	`System.IO.Stream`

Listeners	**Events**
Declaration and Firing	

`public class EventSource {`
`    private ListenerHandler<LogonListener> listeners;`
 	
`    public EventSource() {`
`        super();`
`        this.listeners = new ListListenerHandler<LogonListener>();`
`    }`
 	
`    public void addLogonListener(LogonListener listener) {`
`        this.listeners.add(listener);`
`    }`
 	
`    public void removeLogonListener(LogonListener listener) {`
`        this.listeners.remove(listener);`
`    }`
 	
`    public void causeEvent() {`
`        this.listeners.notifyAll(new Notifier<LogonListener>() {`
`            @Override`
`            public void performNotification(LogonListener listener) {`
`                listener.logonCompleted("e");`
`            }`
`        });`
`    }`
 	
`    public interface LogonListener {`
`        public void logonCompleted(String userName);`
`    }`
`}`
	`Public Class EventSource`
	`    Public Event LogonCompleted(ByVal userName As String)`
	
	`    Public Sub CauseEvent()`
	`        RaiseEvent LogonCompleted("e")`
	`    End Sub`
	`End Class`

Registration, deregistration and handling	

`void testEvents() {`
`    EventSource obj = new EventSource();`
`    LogonHandler eventHandler = new LogonHandler();`
`    obj.addLogonListener(eventHandler);`
`    obj.causeEvent();`
`    obj.removeLogonListener(eventHandler);`
`    obj.causeEvent();`
`}`
 	
`private class EventHandler implements LogonListener {`
`    @Override`
`    public void logonCompleted(String userName) {`
`        System.out.println("User logon: " + userName);`
`    }`
`}`
	`Sub TestEvents()`
	`    Dim obj As New EventSource()`
	`    AddHandler obj.LogonCompleted, AddressOf EventHandler`
	`    obj.CauseEvent()`
	`    RemoveHandler obj.LogonCompleted, AddressOf EventHandler`
	`    obj.CauseEvent()`
	`End Sub`
	
	`Sub EventHandler(ByVal userName As String)`
	`    Console.WriteLine("User logon: " & userName)`
	`End Sub`

	Alternatively, automatic connection with `WithEvents` and `Handles`:
	
	`Public Class EventDemo`
	`    WithEvents obj As New EventSource()`
	
	`    Public Sub TestEvents()`
	`        obj.CauseEvent()`
	`    End Sub`
	
	`    Sub EventHandler(ByVal userName As String) Handles obj.LogonCompleted`
	`        Console.WriteLine("User logon: " & userName)`
	`    End Sub`
	`End Class`

Annotations	**Attributes**
`@Foo(true)`	`<Foo(True)>`
Extension methods	

(n/a)	The example extends the String class (type of the first parameter) with the method *Print()*:
	
	`Imports System.Runtime.CompilerServices`
	
	`<Extension()>`
	`Public Shared Sub Print(ByVal aString As String)`
	`    Console.WriteLine(aString)`
	`End Sub`

Operator overloading	

(n/a)	`Public Shared Operator +(ByVal a As Foo, ByVal b As Foo) As Foo`
	`    Return '...`
	`End Operator`
	Have to be `Shared` and return a value. Parameter type and return type have to be the same as the enclosing class.

Integration with native code	

`native`	`Declare`
	
	`Declare Function getUserName Lib "advapi32.dll" Alias "GetUserNameA" ( _`
	`    ByVal lpBuffer As String, ByRef nSize As Integer) As Integer`

Miscellaneous	
Hides fields with the same name in supertypes, not an Override, therefore not polymorphic	`Shadows`
A namespace with simplified objects and methods for typical tasks, designed to be used by novice programmers (and lazy ones)	`My`
Value of a local variable isn't lost after the method finishes. Similar to `static` variables of functions in C.	`Static`
Compares a string to a pattern. Evaluates to Boolean. The pattern isn't a regular expression, it's more like wildcard operators.	`Like` Example: `"FRL16mxy" Like "F?L*"    ' => True`
Prohibits dangerous implicit conversions. Only *"widening"* conversions are allowed. Has to be declared before any other code.	`Option Strict On`

Java	VB.NET
Antiquated	
	`GoTo`, `On Error ...`, `ReDim`, `Erase`, `Wend`, `REM`, `GoSub`, `Call`

Keine Stichwörter