Vom Java-Enwickler zum VB.NET-Experten in 21 Minuten

Java

VB.NET

Namenskonventionen

- Klassen, Interfaces, Enums: *CamelCase*
- Methoden, lokale Variablen, Felder: *mixedCamelCase*
- Konstanten, Enum-Werte: *UPPER_CASE*

- Klassen, Strukturen, Namensräume: *CamelCase*
- Interfaces: *ICamelCase*
- Private Methoden, lokale Variablen, Felder: *mixedCamelCase*
- Öffentliche Methoden, Properties: *CamelCase*
- Konstanten, Enum-Werte: *CamelCase*

Java-Bezeichner sind *case-sensitive*.

VB-Bezeichner sind *case-**in**sensitive*. Dennoch sollten die Konventionen eingehalten werden, um Interoperabilität mit C# und anderen CLI-Sprachen zu gewährleisten.

Code-Organisierung

**.java*

Statement-Separator: `;`

```
import java.io.*;

package org.foo.bar;

// ...
```

**.vb*

Kein Statement-Separator, Zeile fortführen mit Underscore ('_'), mehrere Statements in eine Zeile jeweils durch Doppelpunkt (':') getrennt

```
Imports System.IO

Namespace Foo.Bar
    ' ...
End Namespace
```

In Java ist es nicht möglich, aus einem Package heraus auf das Default-Package zuzugreifen. In VB.NET ist es jedoch möglich, durch Voranstellen von `Global.` aus einem Namensraum heraus auf den Wurzel-Namensraum zuzugreifen.

Kommentare

```
// ...
/* ... */

/**
 * ...
 */
```

```
' ...
(n/a)

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

Basistypen

```
int
short
long
boolean
char
byte
double
float
(n/a)
java.math.BigInteger


String
Object
Class
```

```
Integer
Short
Long
Boolean
Char
SByte
Double
Single
Decimal (128 bit)
System.Numerics.BigInteger
```

Zusätzlich vorzeichenlose Typen: `UInteger`, `UShort`, `ULong`, `Byte`

```
String
Object
Type
```

Literale

```
null
true, false
"abc"
'D'
0xFF
2.9f
3.14159265
123456L

"\t\r\n"
```

```
Nothing
True, False
"abc"
"D"c
&HFF
2.9!
3.14159265
123456L
```

Keine Escape-Sequenzen in Strings, stattdessen Strings mit den Konstanten `vbCr`, `vbLf` `vbNewLine`, `vbTab`, ... konkatenieren

Variablendeklarationen

```
Foo foo;
Foo foo = new Foo();


int i = 42;
```

```
Dim foo As Foo
Dim foo As Foo = new Foo()
Dim foo As new Foo()        ' etwas kürzer
Dim foo = new Foo()         ' mit Typinferenz per "Option Infer On", sonst Typ Object
Dim i As Integer = 42       ' ohne Typinferenz
Dim i = 42                  ' mit Typinferenz per "Option Infer On", sonst Typ Object
```

Konstanten

```
final
static final
```

```
ReadOnly        ' Darf im Konstruktor gesetzt werden
Const
```

Arrays

```
int[] numbers;
int[][] numbers;
int[][][] numbers;



int[] numbers = new int[6];

numbers[0]
numbers[5]
```

```
Dim numbers() As Integer
Dim numbers()() As Integer      ' "Ausgefranstes" mehrdimensionales Array
                                ' ("Array von Arrays")
Dim numbers()()() As Integer    ' "Ausgefranstes" mehrdimensionales Array
                                ' ("Array von Arrays von Arrays")
Dim numbers(,) As Integer       ' "Rechteckiges" mehrdimensionales Array
                                ' (Ein zusammenhängender Block)
Dim numbers(,,) As Integer      ' "Rechteckiges" mehrdimensionales Array
                                ' (Ein zusammenhängender Block)
```

Java

```
new int[] {1, 2, 4, 8}
```

VB.NET

```
Dim numbers(5) As Integer       ' ACHTUNG: Zahl gibt letzten Index an,
                                           nicht Anzahl der Elemente

numbers(0)
numbers(5)

New Integer() {1, 2, 4, 8}
{1.5, 2, 9.9, 18}               ' Mit Typinferenz (hier: Double-Array,
                                ' da Double der dominante Typ ist)
```

Operatoren

Arithmetisch

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

Math.pow(x, y)
```

```
+, -, *
/
\
Mod

x ^ y
```

Zuweisung

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

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

String-Konkatenation

```
+
+=
```

```
&
&=
```

Weshalb man & statt + verwenden sollte

Logisch

```
&&
||
!
```

```
AndAlso (short-circuit evaluation, wie in Java), And
OrElse (short-circuit evaluation, wie in Java), Or
Not
```

Bitweise

```
&, |
~
^
<<, >>
>>>
```

```
And, Or
Not
Xor
<<, >>
(n/a)
```

Vergleich

```
>, <
==
!=
a.equals(b)
!a.equals(b)
==
!=
```

```
>, <
=       ' Werte
<>      ' Werte
=       ' Objekte
<>      ' Objekte
Is      ' Referenz-Gleichheit von Objekten
IsNot   ' Referenz-Ungleichheit von Objekten
```

Der `IsNot` Operator ist von Microsoft patentiert ...

Konditional

```
condition ? a : b
```

```
If(condition, a, b)
```

Instanz

```
new
o instanceof Foo
```

```
New
TypeOf o Is Foo
```

Funktionsreferenz

(n/a)

`AddressOf` (Referenz auf Methode, zur Verwendung als First-Class-Funktion)

Casting

```
(Foo) bar
bar instanceof Foo ? (Foo) bar : null
*.valueOf()
```

```
DirectCast(bar, Foo)
TryCast(bar, Foo)       ' "Nothing" als Fallback
CType(bar, Foo)         ' mit Konversion

' Zusätzlich Kurzformen von "CType" für Standardtypen:
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)
```

Kontrollstrukturen

Schleifen

```
for (Foo foo : bar) {
}

for (int i = 1; i <= n; i++) {
}

for (int i = n; i >= 0; i -= 2) {
}

while (condition) {
}

do {
} while (condition);
```

```
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
```

Java	VB.NET
`do {` `} while (!condition);`	`Do` `Loop Until condition`
`continue` `break`	`Continue For, Continue Do, Continue While, ...` `Exit For, Exit Do, Exit While, ...`
Bedingungen	

`if (condition) {`
`} else if (condition) {`
`} else {`
`}`
	`If condition Then  ' Then ist optional bei mehrzeiligem If`
	`ElseIf condition Then`
	`Else`
	`End If`

Fallunterscheidung	

`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`
	
	(kein "Durchfallen")

Ausnahmebehandlung	
Werfen	
`throw new Exception("")`	`Throw New Exception("")`
Fangen	

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

Beliebte Exception-Typen	

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

Resource-Management
Ab Java 7:
`try (Resource resource = new Resource()) {`
`}`
	`Using resource As New Resource()`
	`End Using`

Ressource muss *AutoCloseable* implementieren.	Ressource muss *IDisposable* implementieren.
Assertions	
`assert`	`Debug.Assert(), Trace.Assert()`
Typdefinitionen	

`class`
`interface`
`extends`
`implements`
`enum`
	`Class       ... End Class`
	`Interface   ... End Interface`
	`Inherits                  (muss in nächste Zeile oder mit : getrennt werden)`
	`Implements                (muss in nächste Zeile oder mit : getrennt werden)`
	`Enum        ... End Enum       (keine Konstruktoren/Methoden)`
	`Module      ... End Module     (entspricht Klasse mit nur statischen Methoden)`
	`Structure   ... End Structure  (Value Type: copy-on-assignment, keine Vererbung)`

`final`
`abstract (Klasse)`
	`NotInheritable`
	`MustInherit`
	`Partial         (Klasse, die auf mehrere Dateien verteilt ist)`

`this` `super`	`Me` `MyBase`
`Foo.class` `foo.getClass()`	`GetType(Foo)` `foo.GetType()`
Typparameter	
`Foo<T>` `Foo<K, V>`	`Foo(Of T)` `Foo(Of K, V)`
Kovarianz und Kontravarianz:	
`Foo<? extends Bar>` `Foo<? super Bar>`	`Foo(Of Out Bar)` `Foo(Of In Bar)`
Konstruktoren	

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

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

Methoden	
Sichtbarkeit	

Java	VB.NET

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

Modifizierer	

`abstract`
`static`
`final`
`(default)`
	`MustOverride`
	`Shared`
	`NotOverridable (default)`
	`Overridable`

`@Override`	`Overrides`
Methoden mit Rückgabewert	

`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` entspricht der Parameter-Semantik von Java, mit `ByRef` lassen sich Out-Parameter deklarieren, die es in Java nicht gibt. Wenn nichts angegeben, ist seit VB.NET der Standard, bei VB6 war noch `ByRef` der Standard. Es gilt laut Microsoft als gute Praxis, beide immer explizit anzugeben.

Methoden ohne Rückgabewert ("Prozeduren")	
`public void bla() {` `}`	`Public Sub Bla()` `End Sub`
Aufruf von Methoden/Konstruktoren ohne Argumente	
`foo.bar()` `new Foo()`	`foo.Bar()  oder kürzer:  foo.Bar` `New Foo()  oder kürzer:  New Foo`
Varargs	
`...`	`ParamArray`
`public double calcSum(double... args) {` `}`	`Public Function CalcSum(ByVal ParamArray args() As Double) As Double` `End Function`
Optionale Parameter mit Default-Werten	
	`Public Function MyFun(ByVal s As String, Optional ByVal b As Boolean = False) As Integer` `End Function`

Closures

Groovy:

`{ x -> x + 1 }`	`Function(x) x + 1`

`{ x ->`
`  return x + 2`
`}`
	`Function(x)`
	`  Return x + 2`
	`End Function`

`Closure`	`Func(Of T, TResult)`
	`Func(Of T1, T2, T3, TResult)`
	`Func(Of Integer, Boolean)`

`&methodName`	`AddressOf MethodName`

Properties (Getter und Setter)

Lesen und Schreiben

```
public class Foo {
    private int bar;

    public int getBar() {
        return this.bar;
    }

    public void setBar(int bar) {
        this.bar = bar;
    }
}
```

```
' Kurzform (automatisch implementiert):

Public Class Foo
    Public Property Bar As Integer
End Class

' Langform (erlaubt custom Getter und 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
```

Nur-Lesen

```
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
```

Nur-Schreiben

```
public class Foo {
    private int bar;

    public void setBar(int value) {
```

```
Public Class Foo
    Private bar As Integer

    Public WriteOnly Property Bar() As Integer
```

Java

```
        this.bar = value;
    }
}
```

VB.NET

```
        Set(ByVal value As Integer)
            bar = value
        End Set
    End Property
End Class
```

Anonyme Typen

```
Dim bob = New With {.Name = "Uncle Bob", .Age = 42}
Dim bob = New With {Key .Name = "Uncle Bob", .Age = 42}
                    'Key Properties werden für Equals mit einbezogen
```

Objekt-Initialisierer

```
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

```
.hashCode()
.equals(o)
.toString()
```

```
.GetHashCode()
.Equals(o)
.ToString()
```

Guidelines for Implementing Equals and the Equality Operator

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-Initialisierung

(ab VB.NET 2010)

```
New Dictionary(Of Integer, String) From {{0, "Sunday"}, {1, "Monday"}}
New List(Of String) From {"Sunday", "Monday"}
```

Collection-Funktionen und Queries

Groovy:

```
.any {}
.every {}
.collect {}
.findAll {}
```

```
.Any(Function(it) condition)    ' ergibt Boolean
.All(Function(it) condition)    ' ergibt Boolean
.Select(Function(x) result)     ' ergibt neue Collection
.Where(Function(it) condition)  ' ergibt neue Collection
```

Zusätzlich LINQ-Unterstützung, z.B. für Collections:

```
Dim customersForRegion = From cust In customers Where cust.Region = region
```

Ausgabe

```
System.out.println()
```

```
System.Console.WriteLine()
```

Threads

java.lang.Thread

```
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        // do something
    }
});
thread.start();
```

System.Threading.Thread

```
Private Shared Sub DoWork()
    ' do something
End Sub

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

Synchronisierung

```
synchronized (obj) {
}
```

```
SyncLock obj
End SyncLock
```

```
volatile
```

volatile equivalent in VB.NET

Sonstige Typen

```
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
```

Java

VB.NET

Listeners (Events)

Deklarieren und Feuern

```
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
```

Verbinden/Trennen und Handlen

```
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
```

Alternativ automatische Verbindung mit `WithEvents` und `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
```

Annotationen (Attribute)

```
@Foo(true)
```

```
<Foo(True)>
```

Erweiterungs-Methoden

(n/a)

Beispiel erweitert String-Klasse (Typ des ersten Parameters) um Methode *Print()*:

```
Imports System.Runtime.CompilerServices

<Extension()>
Public Shared Sub Print(ByVal aString As String)
    Console.WriteLine(aString)
End Sub
```

Operatoren Überladen

(n/a)

```
Public Shared Operator +(ByVal a As Foo, ByVal b As Foo) As Foo
    Return '...
End Operator
```

Müssen immer `Shared` sein und einen Wert zurückliefern; Parametertypen und Rückgabetyp müssen die selben sein wie die umgebende Klasse.

Anbinden von Native-Code

```
native
```

```
Declare

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

Sonstiges

Verdeckt in Obertyp definierte Elemente gleichen Namens, ist kein Override, also nicht polymorph

```
Shadows
```

Ein Namensraum mit vereinfachten Objekten und Methoden für häufige Aufgaben, richtet sich u.a. an Programmiereinsteiger

```
My
```

Java

Variablenwert einer lokalen Variable bleibt nach Beendigung einer Methode erhalten. Vergleichbar mit `static` Variable in einer C-Funktion.

Vergleicht einen String mit einem Muster (kein regulärer Ausdruck – mehr wie Wildcards). Wertet zu Boolean aus.

Verhindert gefährliche implizite Konversionen, erlaubt nur erweiternde (*"widening"*) Konversionen. Muss vor allem anderen Code stehen.

Veraltetes

VB.NET

```
Static
```

```
Like
```

Beispiel: `"FRL16mxy" Like "F?L*"    ' => True`

```
Option Strict On
```

```
GoTo, On Error ..., ReDim, Erase, Wend, REM, GoSub, Call
```