From Java developer to VisualBasic.NET expert in roughly 21 minutes

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*

Statement separator: `semicolon;`

```
import java.io.*;

package org.foo.bar;

// ...
```

It is not possible to access the root package from a package.

**.vb*

No statement separator. To continue a line over a linebreak, append an underscore ('_'), Multiple statements in one line are separated by a colon (':')

```
Imports System.IO

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

To access the root namespace, prepend `Global.` in a namespace.

Comments

```
// ...
```

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

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

```
' ...
```

(n/a)

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

Basic datatypes

```
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
```

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

```
String
Object
Type
```

Literals

```
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
```

No escape sequences in strings. Concat the predefined constants `vbCr`, `vbLf`, `vbNewLine` 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 less code
Dim foo = new Foo()           ' with type inference per "Option Infer On", Object otherwise
Dim i As Integer = 42         ' without type inference
Dim i = 42                    ' with type inference per "Option Infer On", Object otherwise
```

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]
```

```
Dim numbers() As Integer
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)
```

Java	VB.NET

	`Dim numbers(5) As Integer       ' ATTENTION: 5 specifies the last valid array index,`
	`                                               not the number of elements`

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

	`numbers(0)`
	`numbers(5)`

	`New Integer() {1, 2, 4, 8}`

	`{1.5, 2, 9.9, 18}               ' with type inference (results in an array of Doubles,`
	`                                ' because Double is the dominant type)`

Operators

Arithmetic

`+, -, *`	`+, -, *`
`/ (float)`	`/`
`/ (int)`	`\`
`%`	`Mod`
`Math.pow(x, y)`	`x ^ y`

Assignment

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

String concatenation

`+`	`&`
`+=`	`&=`
	Why you should use & instead of +

Logical

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

Bitwise

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

Comparison

`>, <`	`>, <`
`==`	`=       ' Values`
`!=`	`<>      ' Values`
`a.equals(b)`	`=       ' Objects`
`!a.equals(b)`	`<>      ' Objects`
`==`	`Is      ' Referencial equality of objects`
`!=`	`IsNot   ' Referencial inequality of objects`
	The `IsNot` operator is patented by Microsoft ...

Conditional

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

Instantiation

`new`	`New`
`o instanceof Foo`	`TypeOf o Is Foo`

Function pointer

(n/a) (damn!)	`AddressOf` (reference to method, to be used as a first class function)

Casting

`(Foo) bar`	`DirectCast(bar, Foo)`
`bar instanceof Foo ? (Foo) bar : null`	`TryCast(bar, Foo)       ' falls back to "Nothing"`
`*.valueOf()`	`CType(bar, Foo)         ' with 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 Each foo In bar`
	`Next`

`for (int i = 1; i <= n; i++) {`
`}`
	`For i As Integer = 1 To n`
	`Next`

`for (int i = n; i >= 0; i -= 2) {`
`}`
	`For i As Integer = n To 0 Step -2`
	`Next`

`while (condition) {`
`}`
	`While condition`
	`End While`

`do {`
`} while (condition);`
	`Do`
	`Loop While condition`

Java

```
do {
} while (!condition);

continue
break
```

VB.NET

```
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 discrimination

```
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

```
IllegalArgumentException
NullPointerException
UnsupportedOperationException
IOException
```

```
ArgumentException, ArgumentNullException, ArgumentOutOfRangeException
NullReferenceException
NotSupportedException, NotImplementedException
IOException
```

Resource management

Since Java 7:

```
try (Resource resource = new Resource()) {
}
```

```
Using resource As New Resource()
End Using
```

Resource has to implement *AutoCloseable*.

Resource has to implement *IDisposable*.

Assertions

```
assert
```

```
Debug.Assert(),Trace.Assert()
```

Type definitions

```
class
interface
extends
implements
enum



final
abstract (Klasse)


this
super

Foo.class
foo.getClass()
```

```
Class       ... End Class
Interface   ... End Interface
Inherits                        (has to be in the next line or separated by : (colon))
Implements                      (has to be in the next line or separated by : (colon))
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)

NotInheritable
MustInherit
Partial         (a class spanning multiple files)

Me
MyBase

GetType(Foo)
foo.GetType()
```

Type parameters

```
Foo<T>
Foo<K, V>
```

```
Foo(Of T)
Foo(Of K, V)
```

Covariance und Contravariance:

```
Foo<? extends Bar>
Foo<? super Bar>
```

```
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

```
Java                                            VB.NET

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

Modifiers

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

@Override                                       Overrides
```

Methods with return value

```
public int name(double a, String b) {           Public Function Name(ByVal a As Double, ByVal b As String) As Integer
    return 1;                                       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()                                       foo.Bar()  or shorter:  foo.Bar
new Foo()                                       New Foo()  or shorter:  New Foo
```

Varargs

```
...                                             ParamArray

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

Optional parameters with default values

```
                                                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 ->                                          Function(x)
    return x + 2                                    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 and setter methods)

Reading and writing

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

Readonly

```
public class Foo {                              Public Class Foo
    private int bar;                                Private bar As Integer

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

Writeonly

```
public class Foo {                              Public Class Foo
    private int bar;                                Private bar As Integer

    public void setBar(int value) {                 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
```

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

```
.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 initialisation
(from VB.NET 2010)

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

Collection functions and queries
Groovy:

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

```
.Any(Function(it) condition)    ' results in Boolean
.All(Function(it) condition)    ' results in Boolean
.Select(Function(x) result)     ' results in a new Collection
.Where(Function(it) condition)  ' results in a new Collection
```

Additional LINQ support for collections:

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

Output

```
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()
```

Synchronisation

```
synchronized (obj) {
}

volatile
```

```
SyncLock obj
End SyncLock
```

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
```

Java

VB.NET

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) (sigh!)

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`

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