C# for Java Programmers

Thomas G. Schuessler

Thomas G. Schuessler is the founder of ARAsoft, a company offering products, consulting, custom development, and training to customers worldwide, specializing in integration between SAP and non-SAP components and applications. Thomas is the author of SAP's BIT525 and BIT526 classes. Prior to founding ARAsoft in 1993, he worked with SAP AG and SAP America for seven years.

(complete bio appears on page 28)

old beliefs die hard even when demonstrably false.
Edward O. Wilson, Consilience

Java has become the standard programming language in many companies. Java 2 Enterprise Edition (J2EE) is a standard for writing enterprise applications. SAP provides its own J2EE server and has developed several applications in Java. The SAP Java Connector (JCo) is a very successful middleware component that allows Java applications to access SAP applications written in ABAP. So why look at Microsoft's new programming language C#[1]? The answer is that in my opinion two approaches will survive as common development platforms, Sun's J2EE and Microsoft's .NET. And C# is (un)arguably the best Microsoft programming language for .NET.

Recently, SAP has released the SAP .NET Connector, allowing developers to build the same kinds of applications for .NET that they could already build with Java using JCo.

Microsoft developed .NET as a full-blown competitor for J2EE, which is understandably viewed by Microsoft as one of the two major threats to its market position (the other one of course being Linux). Who will win, Microsoft or Sun? I frankly do not know, but it seems to be a reasonable assumption that both .NET and J2EE will have significant market share for quite some time.

[1] Pronounced "see sharp", as in the musical note. My assumption is that Microsoft was looking for a name that reflected the C++ heritage of the language and simply combined the two plus signs to form the # character.

Listing 1: A Java or a C# Class?

```
public class SampleClass {

  private int anInteger;

  public SampleClass(int number) {
    anInteger = number;
  }

  public int getNumber() {
    return anInteger;
  }
  public void setNumber(int number) {
    anInteger = number;
  }

  public String getText() {
    String result;
    switch (anInteger) {
      case 0:
        result = "Zero";
        break;
      case 1:
        result = "One";
        break;
      default:
        result = "Not zero or one";
```

Developers will have better employment chances if they are proficient in both Java and C#. This article is an introduction to C# for Java developers.

I will cover the major differences between the two languages without you having to read several hundred pages of a C# book. These books are rather voluminous since they teach the reader lots of concepts that a Java programmer is already familiar with. The goal of this article is to enable you to get started writing C# code quickly. And actually writing some sample applications is the only way to really make a meaningful decision between J2EE/Java and .NET/C#. To speed up your learning process, the article contains lots of suitable code snippets as well as the complete source code for two classes in Appendices A and B.

Why C#?

One of the advantages (or disadvantages, as some would claim) of .NET is that it allows you to choose from (and mix-and-match) multiple programming languages. The main languages offered by Microsoft for .NET are Visual C#, Visual C++ with Managed Extensions, and Visual Basic .NET. Other languages like Eiffel, Perl, Python, and COBOL (no typo!) are available from third parties. I have selected C# for two reasons:

- C# is closer to Java than C++ and Visual Basic.
- C# is — personal opinion! — a better language than C++ and Visual Basic.

I will not cover Visual J# .NET in this article.


```
        break;
      }
      return result;
    }

    public static int[] get10AmicableNumberPairs() {
      int[] pairs = new int[2*10];
      int pairsCount = 0;
      for (int i = 2; ; i++) {
        int sum = getSumOfAliquotParts(i);
        if (i < sum && i == getSumOfAliquotParts(sum)) {
          pairs[pairsCount * 2] = i;
          pairs[pairsCount * 2 + 1] = sum;
          pairsCount++;
        }
        if (pairsCount == 10) break;
      }
      return pairs;
    }

    public static int getSumOfAliquotParts(int number) {
      int sum = 0;
      for (int i = 1; i < number; i++) {
        if (number % i == 0) sum += i;
      }
      return sum;
    }
  }
```

If you want to use your existing Java libraries in .NET, consider converting them to C# using the conversion tool from **http://msdn.microsoft.com/vstudio/downloads/jca/**.

How Different Are C# and Java?

Look at the code in **Listing 1** and **Listing 2**. Is this C# or Java? The answer is: both. Some source code is identical in both languages. This is not that surprising given that Java inherited from C and C++, while C# inherited from C, C++, and Java.

Don't get your hopes up too high, though. There are lots of — sometimes subtle — differences between C# and Java. But let us start with the commonalities.

Listing 2: Using SampleClass in C# and Java

```
SampleClass mySampleClass = new SampleClass(1);
String text = mySampleClass.getText();
int sum = SampleClass.getSumOfAliquotParts(220);
int[] pairs = SampleClass.get10AmicableNumberPairs();
```

✓ ***Note!***

The code in Listings 1 and 2 does not do anything too interesting because the syntax of C# and Java is only partially identical, which precluded me from writing a really useful class that would work in both languages. A C# developer would implement the class in Listing 1 in a different way, taking advantage of certain features of C# that I will introduce in this article.

In case you are interested: Aliquot parts of an integer number are those integers by which the number can be divided without remainder, starting with 1, but excluding the number itself. This is used in the definition of amicable numbers. Two numbers are amicable if the sum of the aliquot parts of the first number is the second number and vice versa. The first pair of amicable numbers is 220 and 284. Pythagoras already knew that. The famous composer and violin player Paganini found the pair 1184 and 1210 that had been missed by all serious mathematicians before. See ***http://amicable.homepage.dk/knwnap.htm*** *for more information.*

And for those interested in performance: Running the — not very cleverly implemented — get10AmicableNumberPairs() method in Java (Sun's JDK 1.3) took only slightly longer than in C#. Java's performance has really improved. Of course, real applications do not consist solely of integer manipulations, therefore be careful not to generalize this result.

Commonalities Between C# and Java

Both languages are object-oriented. They have classes, supporting single inheritance, and interfaces. Classes are abstract or concrete. All classes are ultimately derived from a common base class, *Object*. Classes can have members, which include fields and methods. You can overload and override methods. Fields and methods can be static. Access modifiers like `private` control the ability of other classes to use these members. Methods have a return value, which can be `void`. A class can have multiple constructors with different argument types.

Memory management is automatic, with a Garbage Collector cleaning up after you.

There is a runtime environment and a standard class library. In C#, the runtime environment is called the Common Language Runtime, and the standard class library is known as the .NET Framework Class Library (both together constitute the .NET Framework).

Both languages support exceptions in a similar fashion, using the `try`, `catch`, `finally`, and `throw` keywords.[2]

Both languages contain the following keywords for program flow control: `if/else`, `switch/case/default`, `for`, `while`, `do/while`, `break`, `continue`, `return`.

Both languages have the following operators in common:

+, -, *, /, %, =, +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=, <, <=, >, >=, ==, !=, &&, ||, <<, >>, [], (), ., ++, —, !, ~, &, |, ^, ?:

Both languages allow you to use arrays and have built-in support for multi-threading.

Primitive Data Types in C#

After this rather impressive list of commonalities, now comes the time to take a deep breath and delve into the differences. Java contains the following

[2] There are lots of differences concerning exception handling, though. They will be discussed at large later.

primitive data types: `boolean`, `char`, `long`, `int`, `short`, `byte`, `double`, `float`. Which primitive data types exist in C#? Answer: *none*. (Let us all take an incredulous moment here, courtesy of John Cage — not the composer, the Ally McBeal character.)

What about the code in Listing 1? Did it not use variables of type `int`? Yes, but they are objects! Everything is an object in C#. The C# code in **Listing 3** will compile and execute nicely.

Listing 3: Everything Is an Object

```
int i = 42;
if (7.Equals(i)) {
  Console.Out.WriteLine("Liar");
}
String s = 7.GetType().FullName;
Console.Out.WriteLine(s);
```

The output will be `System.Int32`. This is the class behind the alias `int`. **Figure 1** contains all the aliases available in C#.

A few words about the column titled "CLS-Compliant". CLS stands for the .NET Common Language Specification that defines the rules that any .NET-compliant language needs to adhere to if it wants to interoperate with other .NET languages. A developer interested in language interoperability should make sure that any non-compliant features used are not visible to other classes, e.g., by using

Figure 1 *C# Type Aliases*

C# Type Alias	.NET Class	CLS-Compliant	Description
bool	System.Boolean	Yes	Boolean value (true/false)
char	System.Char	Yes	Unicode character
string	System.String	Yes	String
byte	System.Byte	Yes	8-bit unsigned integer
sbyte	System.SByte	No	8-bit signed integer
short	System.Int16	Yes	16-bit signed integer
ushort	System.UInt16	No	16-bit unsigned integer
int	System.Int32	Yes	32-bit signed integer
uint	System.UInt32	No	32-bit unsigned integer
long	System.Int64	Yes	64-bit signed integer
ulong	System.UInt64	No	64-bit unsigned integer
decimal	System.Decimal	Yes	128-bit decimal number (28 significant digits)
double	System.Double	Yes	64-bit floating point number
float	System.Single	Yes	32-bit floating point number
object	System.Object	Yes	An object

unsigned 32-bit integers (`uint`) only as `private` variables.[3]

Back to less arcane topics like performance. If everything is an object in C#, will adding two integers not be slow, because it involves a method call? No, because all the types in Figure 1 — with the exception of `object` and `string` — are value types (as opposed to reference types). For a variable that is a value type, the variable does not contain a reference to an object, but the actual data. Hence operations like adding two integers are as fast as if C# had real primitive types.

Why did the C# designers come up with this approach? Is there anything wrong with Java's primitive types? Look at the Java code in **Listing 4**. I am trying to put a string and an integer into a hash table. This will not compile since the *put()* method expects an *Object* and an integer is a primitive type. Also, casting an *Object* to an `int` in the last line is not allowed.

Listing 5 contains a working version of the same idea. The developer has to wrap the primitive type into an object before adding it to the hash table and unwrap the object again when assigning it back to the primitive type.

In C# (see **Listing 6**), the code looks much cleaner. The wrapping and unwrapping (called boxing and unboxing) takes place automatically as needed. Note that — as opposed to the earlier C# code samples — I have defined the string variable using the alias in lower case according to the table in Figure 1. "String" and "string" can be used interchangeably in C#. Also note the square brackets to access items in the hash table. This rather intuitive syntax makes use of an indexer, a very neat feature of C# to be introduced later.

[3] Putting the attribute `[assembly: CLSCompliantAttribute(true)]` into at least one class of an assembly causes the C# compiler to issue warnings if a class in the assembly violates the CLS rules. Attributes and assemblies will be covered later in this article.

Listing 4: Illegal Java Hash Table Code

```
Hashtable ht = new Hashtable();
String s = "Java";
ht.put("A string", s);
int i = 42;
ht.put("An integer", i);
s = (String) ht.get("A string");
i = (int) ht.get("An integer");
```

Listing 5: Legal Java Hash Table Code

```
Hashtable ht = new Hashtable();
String s = "Java";
ht.put("A string", s);
int i = 42;
ht.put("An integer", new Integer(i));
s = (String) ht.get("A string");
i = ((Integer)
  ht.get("An integer")).intValue();
```

Listing 6: A Hash Table in C#

```
Hashtable ht = new Hashtable();
string s = "C#";
ht["A string"] = s;
int i = 42;
ht["An integer"] = i;
s = (String) ht["A string"];
i = (int) ht["An integer"];
```

Constants

Java does not have a special syntax for constants, but allows you to define fields as `static final`. C# has the `const` keyword for constants that can be evaluated at compile-time (see **Listing 7**). If the value for the constant cannot be determined at compile-time,

Listing 7: Constants in C#

```
public const double PI = 3.14159265358;
public static readonly Uri ARAsoft = new Uri("http://www.arasoft.de");
```

e.g., if a constructor of a class needs to be invoked, then a constant field can be defined as `static readonly` (see the second definition in Listing 7).

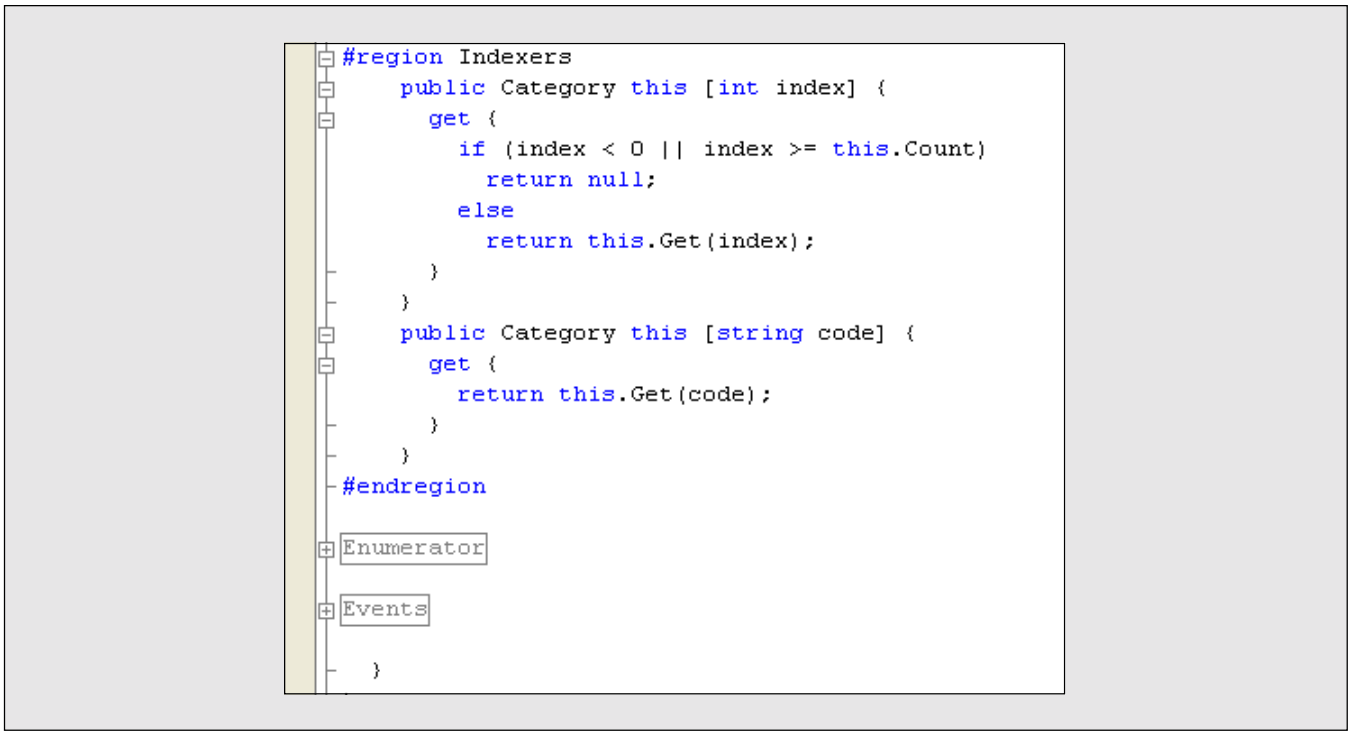
Managing Source Code

In Java, the source code for different classes must reside in different files. The only exception are inner classes, the names of which have to be specified including the containing class, e.g., `TopLevelClass.NestedClass`. In C#, you can combine arbitrary classes in one source file. This gives the developer greater flexibility in organizing the source code of a project according to personal preferences.

Java does not have preprocessor directives, C# does. The `#if`, `#else`, `#elif`, `#endif`, `#define`, and `#undef` directives allow you to control the inclusion and exclusion of certain parts of your source code under specific conditions.

The `#region` and `#endregion` directives allow you to show/hide certain parts of your source code within the development environment. In **Figure 2**, the `Indexers` region is shown, whereas the `Enumerator` and `Events` regions are hidden.

Figure 2 The Directives #region and #endregion



Using this feature, it becomes much easier to work with large source files. Be aware, though, that when you are searching for a string in the IDE, the closed regions will not be searched.

Managing Components

In Java, the issue of name space collisions is avoided by using packages. Each package can contain multiple classes and interfaces. Usually, the compiled versions of the classes and interfaces of one or more packages are packaged into a `zip` file.

In C#, the equivalent of a package is a name space. Components are delivered as assemblies. There are two types of assemblies: libraries (usually `dll` files) and applications (`exe` files). Applications can be executed directly, whereas libraries are used by (other) libraries and applications. **Listing 8** shows a class defined in a name space. The `namespace` keyword is the equivalent of the Java `package` statement. While there can only be one `package` statement per Java source file, a source file in C# can contain multiple `namespace` blocks, each delimited by a set of curly brackets.

When you want to use classes or interfaces from a different package in Java, you normally employ `import` statements. The C# equivalent is the `using` statement (see Listing 8). There is a big difference, though: Java allows you to import individual classes and interfaces[4], whereas the `using` statement in C# only applies to a complete name space.

Listing 8: A Name Space in C#

```
using System;
using System.Collections.Specialized;
namespace De.ARAsoft.Categories
{
  public class Category : IComparable {
    public Category() {}
  }
}
```

Class Access Modifiers

Classes in Java and C# can be defined as `public` or `private` with the same effect in both languages. Specifying neither `public` nor `private` in Java defines the visibility of the class as *package*, i.e., the class can be used within its package, but not outside of it. The C# equivalent is `internal`, which is the default access modifier and defines the visibility of the class as *assembly*. This has a slightly different effect than package visibility in Java, since an assembly can contain more than one name space.

Member Access Modifiers

Fields and methods in Java and C# can be defined as `public`, `private`, or `protected` with the same effect in both languages. Omitting the access modifier in Java yields package scope. The C# equivalent again is `internal`, but this is not the default access modifier for a field or method. Omitting the access modifier in C# yields private visibility, instead.[5] Access modifiers `protected` and `internal` can be used together for the same member, all other combinations are illegal.

Base Classes and Interfaces

If a class in Java extends a class other than *Object*, you have to indicate this using the `extends`

[4] Specifying individual classes and interfaces in Java import statements is recommended since the reader of your source code can see immediately which classes and interfaces from other packages you are using.

[5] A better default, if you ask me.

Listing 9: Base Class and Interfaces in C#

```
public class Categories :
             System.Collections.Specialized.NameObjectCollectionBase,
             IEnumerable {
```

keyword. If a Java class implements one or more interfaces, they are specified following the `implements` keyword. In C#, there is no explicit syntactical distinction between these two concepts: the base class as well as the implemented interfaces are listed after the name of the class following a colon — with the base class listed first (see **Listing 9**). Class *Categories* extends class *System.Collections.Specialized.NameObjectCollectionBase* and implements interface *IEnumerable*. A class that extends *Object* directly (like class *Category* in Listing 8) can omit the base class and directly list the implemented interfaces. In other words, the only way in C# to find out whether the first name after the colon identifies the base class or just an implemented interface is to know whether this first name identifies a class or an interface. This is one of the few areas where I actually like Java better. In order to make your C# code easier to read, I would advise that you follow Microsoft's recommendation and have the names of your interfaces all start with the letter "I".

Inheritance and Polymorphism

Using inheritance and polymorphism in C# works slightly differently than in Java. Take a look at **Listing 10**. Class *BaseClass* defines one constructor, which takes an integer argument. The only public

Listing 10: A Base Class and a Derived Class

```
public class BaseClass {
  protected int number;
  public BaseClass(int number) {
    this.number = number;
  }
  public virtual int GetNumber() {
    return number;
  }
  protected void SetNumber(int number) {
    this.number = number;
  }
}
public class DerivedClass : BaseClass {
  string name;
  public DerivedClass() : base(0) {}
```

(continued on next page)

(continued from previous page)

```
    public DerivedClass(int number) : base(number) {
    }
    public DerivedClass(int number, string name) : base(number) {
      this.name = name;
    }
    public sealed override int GetNumber() {
      return ++number;
    }
    new public void SetNumber(int number) {
      base.SetNumber(number);
    }
  }
```

method of *BaseClass* is *GetNumber()*. This method is defined as `virtual`, which means that it can be overridden by a derived class. Methods not defined as `virtual` or `abstract` cannot be overridden.

Class *DerivedClass* has three constructors, one is parameterless, one takes an integer, one an integer and a string. All constructors of *DerivedClass* invoke the constructor of *BaseClass*. This is accomplished by using the `base` keyword after the colon.

DerivedClass overrides the *GetNumber()* method. This requires the use of the `override` keyword. *GetNumber()* is also defined as `sealed`, which means that it cannot be overridden by derived classes. The `sealed` keyword can also be used for a complete class, like the `final` keyword in Java.

DerivedClass wants to have a public *SetNumber()* method. This would seem to be impossible since this method was not defined as virtual in *BaseClass*. Adding the `new` keyword takes care of this issue. *SetNumber()* in *DerivedClass* now hides the method by the same name in *BaseClass*.

This feature is especially valuable when you extend a class that is part of a component that you do not have the source code for. In Java, the following could happen: You extend class *A* from package `com.company.stuff`. You write a method *x()* for your derived class and declare it as `public`. All is well so far. Now the vendor of `com.company.stuff` ships a new version where class *A* has a `public final` method *x()*. From now on, your derived class will no longer compile. The only solution would be to rename the method, which might require changes in many other classes. In .NET, you would simply add the `new` keyword to your method *x()* and the clients of your derived class will not have to do anything.

Exception Handling

C# and Java share the following features concerning exception handling:

- You use the `try`, `catch`, and `finally` keywords to handle exceptions in your code.
- There is a base class, *Exception*, from which you can derive your own exception classes.
- You can throw exceptions in your code using the `throw` keyword.

Listing 11: Exception Handling

```
Categories cats = new Categories();
try {
  Category cat1 = new Category();
  cat1.Code = null;
}
catch (ArgumentNullException) {
  Console.Out.WriteLine("Passed null argument.");
  throw;
}
catch (Category.InvalidChangeException ex) {
  throw new InvalidOperationException
            ("An invalid change was made.", ex);
}
```

Let us now look at how C# differs from Java as regards exception handling:

- There is no `throws` clause in method declarations in C#. In Java, a method must either catch all exceptions (unless they are of type *RuntimeException*) or declare the exceptions that might be propagated to the caller in the method signature using the `throws` clause. Although some Java developers consider it inconvenient, this requirement guarantees that a developer calling a method is aware of the exceptions that might be thrown and can react appropriately.

  A C# developer is not forced syntactically to deal with exceptions, but of course you should still deal with exceptions responsibly.

- If you want to rethrow the exception caught in a catch block, you can simply execute a `throw` statement without any arguments (see the first catch block in **Listing 11**).

- If you do not need the exception object that was thrown, then you do not have to specify an exception variable in your catch block (see the first catch block in Listing 11).

- One of the constructors of class *System.Exception*[6] allows you to pass an exception object (known as the *inner exception*). This enables you to throw an exception of your choice without losing the information contained in the original exception to which you are reacting (see the second catch block in Listing 11[7]). The original exception can be accessed using the *GetBaseException()* method of class *Exception*.

- The sequence of your catch blocks is important in C#. As opposed to Java, catch blocks are evaluated sequentially. In other words, a more generic catch (like for *System.Exception*) would prevent a more specific exception from being handled in its proper catch block. Fortunately the compiler

[6] Class *Exception* is part of the *System* namespace, so that its fully qualified name is *System.Exception*.

[7] Class *Category.InvalidChangeException* is an inner class.

Listing 12: Invalid Exception Handling

```
Categories cats = new Categories();
try {
  Category cat1 = new Category();
  cat1.Code = null;
}
catch (Exception ex) {
  Console.Out.WriteLine("This catch would handle all exceptions!!!");
}
catch (ArgumentNullException) {
  Console.Out.WriteLine("Passed null argument.");
  throw;
}
catch (Category.InvalidChangeException ex) {
  throw new InvalidOperationException
            ("An invalid change was made.", ex);
}
```

Listing 13: Invalid Scope Problem

```
Categories cats = new Categories();
try {
  Category cat1 = new Category();
  cat1.Code = null;
}
catch (ArgumentNullException) {
  Console.Out.WriteLine("Passed null argument.");
  throw;
}
catch (Category.InvalidChangeException ex) {
  throw new InvalidOperationException
            ("An invalid change was made.", ex);
}
Category cat1 = new Category();
```

catches (no pun intended) this, so that the code shown in **Listing 12** will generate compiler error messages.

- Variables declared in `try/catch/finally` blocks cannot have the same names as variables that are in scope outside of these blocks. The code in **Listing 13** will not compile because `cat1` is defined in the `try` block and in the enclosing scope. The exception variables in multiple catch blocks can have the same name, though, so the code in **Listing 14** will compile fine. Note that both exception variables are called `ex`.

Listing 14: Multiple Exception Variables Have the Same Name

```
Categories cats = new Categories();
try {
  Category cat1 = new Category();
  cat1.Code = null;
}
catch (ArgumentNullException ex) {
  Console.Out.WriteLine("Passed null argument.");
  throw;
}
catch (Category.InvalidChangeException ex) {
  throw new InvalidOperationException
            ("An invalid change was made.", ex);
}
```

Listing 15: Checked and Unchecked Arithmetic

```
int i = 123456789;
int j = 987654321;
checked {
  int m = unchecked(i * j);
  int n = i * j;  // Exception is thrown
}
unchecked {
  int m = checked(i * j); // Exception is thrown
  int n = i * j;
}
```

- Class *Exception* has a read-only property *StackTrace*, which returns a string containing the stack trace. For other useful properties and methods of this class, please refer to the documentation of Visual Studio .NET.

- If an arithmetic overflow or underflow occurs, e.g., when the result of an integer computation is too large for the result variable, an exception of type *OverFlowException* will only be thrown if the computation is executed in a *checked* context. Otherwise, the result will be truncated and execution will continue as if nothing untoward had happened! Doesn't that sound scary? Fortunately, there are ways to control whether a specific instruction is executed checked or unchecked. You can define this globally as a property for your project, include code in `checked` or `unchecked` blocks, and even use the `checked` and `unchecked` operators. See **Listing 15** for examples.

Listing 16: A Method with ref and out Arguments

```
public class Reference {
  public static void CallByReference(ref string name, out int length) {
    name = name.ToUpper();
    length = name.Length;
  }
}
```

Listing 17: A Call to a Method with ref and out Arguments

```
string test = "test";
int length;
Reference.CallByReference(ref test, out length);
```

This has been a long and — hopefully not too — tiring section of the article. While Java and C# share the general concept of exception handling, they deal with exceptions differently and you should exercise extreme caution in this area when switching between the two languages.

Calling by Reference

We will now start to examine features of C# for which there is no Java equivalent.

In Java, methods are always called by value. When you pass an object to a method, the method can manipulate the object by calling its methods, but the object reference itself can never be modified. When you pass a primitive data type, its value can never be modified by the called method. In most cases, this is what you want, but Java gives you no alternative. C# does. You can define parameters as `ref` or `out` (see **Listing 16**). In both cases, the parameter is passed by reference, i.e., the content can be modified by the called method. The difference between `ref` and `out` is that in the latter case the passed variable does not have to be initialized by the caller and the called method will assign a value to it. It is obviously very important for the caller of a method to know which arguments are passed by value and which are passed by reference. Hence the C# syntax requires that `ref` and `out` parameters are identified as such by the caller (see **Listing 17**).

Properties

Allowing a client of a class to directly manipulate its fields (also known as instance variables) by making them `public` is frowned upon for good reasons. It would be a violation of the principles of object-oriented programming. In Java, we therefore usually make our fields `private` and provide get and set methods that control access to them. The same can be done in C#, of course, as exemplified by the

Listing 18: Access to Fields

```
// Accessing a public field
int i = x.field;
x.field = 7;
// Accessing a field via get/set methods
int j = x.getField();
x.setField(7);
```

Listing 19: A Property in C#

```
private string code;
public string Code {
  get {
    return code;
  }
  set {
    if (value == null)
      throw new ArgumentNullException();
    if (code == null)
      code = value;
    else
      throw new InvalidChangeException("Only changeable once.");
  }
}
```

GetNumber() and *SetNumber()* methods in Listing 10. Obviously, the syntax used to access a public field is much more straightforward than having to call get and set methods (see **Listing 18**). It would be nice if we could combine the syntactical ease of direct access to fields with the control we obtain by writing our own access methods. C# provides a solution in the form of properties, also known as *smart fields*.

Listing 19 contains the source code for a property called `Code`. There is a private field, `code`, that holds the value for the property. Then follows the definition of the property itself, with get and set accessor functions. In the set function, the value assigned to the property is available in special variable `value`, which is a keyword in C#. Note that the set function checks the contents of `value` and `throws` an exception if it is not appropriate. Client code using a class with the `Code` property is shown in **Listing 20**.

Listing 20: Using a Property in C#

```
Category category = new Category();
category.Code = "THA";
string s = category.Code;
```

Listing 21: Using an Indexer in C#

```
Categories cats = new Categories();
Category cat = new Category("ASI", "Asian", true);
cats.Add(cat);
cats.Add(new Category("THA", "Thai", true));
cat.AddSubCategory(cats["THA"]);
```

You can make a property read-only or write-only by omitting the set or get accessor functions, respectively. But you cannot have different access modifiers for get and set, which is very unfortunate, since very often we would like to allow read access to a property to everybody, but limit write access to `private`, `protected`, or `internal`. This is a surprising decision by the C# designers, since Visual Basic, where properties have been common for a long time, allowed you to have different access modifiers for get and set. Visual Basic .NET does not support this anymore, though.

Indexers

As stated earlier, both C# and Java support arrays, allowing access to array elements using the `[]` syntax. Many objects that are not technically arrays, though, are conceptually like arrays. A collection, for example, contains elements in a fashion similar to an array. Syntactically, most languages, including Java, require us to access these elements in a totally different way. In recognition of this, the designers of C# have introduced *indexers*, which allow you to write classes that can be used as if they were arrays. Look at the code in **Listing 21**. In the first statement, an object of type *Categories* is created. In the second and third statement a *Category* object is created and added to the collection. The fourth statement creates and adds yet another category. Finally, in the last statement an indexer is used to access a category, identified by its code (THA), in the collection. This syntax is very intuitive and allows you to use proper arrays and array-like objects in the same fashion. Note that the index does not even have to be numeric. C# allows us to create indexers for various data types, as shown in **Listing 22**, the source code for the indexers provided by class *Categories*.

The foreach Statement

Iterating through, or enumerating, an array or array-like object is a common requirement in a program. Both C# and Java allow the use of `while`, `do/while`, and `for` statements to deal with this requirement. In each case, you have to be careful to ensure that all elements of the object are accessed and no attempt is made to access non-existing elements. Making mistakes in this area is a very common programming issue. To facilitate the correct handling of enumeration, C# has inherited (i.e., stolen) from Visual Basic's `For Each` statement, spelled `foreach` in C#. The `foreach` statement can be used on arrays and collection objects that follow certain rules. Using `foreach` is easy, as can be gleaned from the source code in **Listing 23**. Making your own collection classes support `foreach` is a little more effort (see **Listing 24**). Some of this code would not have been necessary if we just wanted to support C#'s `foreach`, but remember that .NET allows you to mix and match languages, specifically

Listing 22: Providing Indexers in a Class

```
public Category this [int index] {
  get {
    if (index < 0 || index >= this.Count)
      return null;
    else
      return this.Get(index);
  }
}
public Category this [string code] {
  get {
    return this.Get(code);
  }
}
```

Listing 23: Using foreach in C#

```
foreach (string key in newCats.GetAllRootCategoryKeys()) {
  Category catX = newCats[key];
  TreeNode node = new TreeNode(catX.Description);
  node.Tag = catX;
  treeView1.Nodes[0].Nodes.Add(node);
  AddSubNodes(node, catX, newCats);
}
```

Listing 24: Support for foreach in a Class

```
new public Enumerator GetEnumerator() {
  return new Enumerator(this, version);
}
IEnumerator IEnumerable.GetEnumerator() {
  return GetEnumerator();
}

public class Enumerator : IEnumerator {
  private int index;
```

(continued on next page)

(continued from previous page)

```
    private Categories collection;
    private string[] keys;
    private int version;
    public Enumerator(Categories categories, int version) {
      collection = categories;
      keys = categories.GetSortedKeys();
      this.version = version;
      Reset();
    }
    public bool MoveNext() {
      index++;
      return (index < collection.Count);
    }
    public void Reset() {
      index = -1;
    }
    public Category Current {
      get {
        if (index == -1) throw new InvalidOperationException
                                    ("Use MoveNext() first.");
        return collection[keys[index]];
      }
    }
    object IEnumerator.Current {
      get {
        return Current;
      }
    }
  }
```

to use classes developed in one language in another language. To enable a program written in Visual Basic, for instance, to use our collection class, we have to implement the *IEnumerable* interface in addition to providing an *Enumerator* for C#. **Listing 25** is actual Visual Basic code using two classes written in C#. If that is not cool, I do not know what is.

Operator Overloading

Java allows you to concatenate two strings using the + operator, but does not support operator overloading elsewhere. C#, due to its C++ heritage, enables us to overload most operators. Obviously, you want to use this feature with caution, so that a programmer who uses one of your classes has an intuitive understanding of the semantics of an overloaded operator. **Listing 26** shows two ways of adding an element to a collection. The first (in the second statement) uses the *Add()* method, the second one (in the last statement) uses the overloaded + operator. Providing both a method and an overloaded operator for a function is recommended since most other .NET languages do not support operator overloading. **Listing 27** contains the source code for the overloaded operator. As you can see, the *Add()* method is called to provide the required functionality.

Listing 25: Using C# Classes in Visual Basic

```
Dim cats As Categories = New Categories()
Dim cat As Category = New Category("THA", "Thai", True)
cats.Add(cat)
cats.Add(New Category("JAP", "Japanese", True))
For Each cat In cats
  MessageBox.Show(cat.Description)
Next
```

Listing 26: Using an Overloaded Operator

```
Category cat = new Category("ASI", "Asian", true);
cats.Add(cat);
cat = new Category("CHI", "Chinese", true);
cats += cat;
```

Listing 27: The Overloaded Operator

```
public static Categories operator +
              (Categories categories, Category category) {
  categories.Add(category);
  return categories;
}
```

Value Types

Earlier in this article I mentioned that while everything in C# is an object, certain performance advantages can be gained by using value types. There are two data types in C#, reference types and value types. Normal classes are reference types. Value types are further subdivided into enumerations (as opposed to Java, C# has an `enum` keyword to define C-style enumerations) and `struct` types. Most of the built-in simple types in C# are `struct` types (`int`, `bool`, etc.). Programmers can develop their own `struct` types for lightweight objects that do not require inheritance or their own parameterless

Listing 28: Using a struct Type

```
HelpvaluesItem country = new HelpvaluesItem("FRA", "France");
MessageBox.Show(country.ToString());

HelpvaluesItem currency = new HelpvaluesItem();
MessageBox.Show(currency.ToString());

HelpvaluesItem companyCode;
companyCode.code = "0007";
companyCode.description = "ARAsoft";
MessageBox.Show(companyCode.ToString());
```

constructors, which are prohibited for `struct` types. **Listing 28** shows three different usages of a `struct` type.

- In the first case (the first two statements), the constructor with two parameters is called, thus initializing the fields of the object.
- In the second case (the middle two statements) the parameterless constructor is called (C# generates this constructor automatically and does not allow us to provide our own version). All fields are initialized to default values.
- In the third case (the bottom four statements), a variable of type *HelpvaluesItem* is created without using `new`. This is unique to `struct` types and requires the developer to assign values to all fields of the object.

Listing 29 provides the source code for type *HelpvaluesItem*.

When should you use `struct` types in your projects? Whenever an object mainly consists of a — limited — number of data fields and you can live with the limitations explained above. Otherwise, stick to a regular class.

User-Defined Conversions

Java and C# support implicit and explicit type conversions. Implicit conversions take place without any effort by the developer. If you assign an `int` to a `long`, no special action is required. Explicit conversions require casting by the developer. An object of a certain class can always be cast to any type higher up in its class hierarchy.

C# goes one step further and allows you to define your own implicit and explicit conversions. You can support conversions between a class and other types, including simple types like `int` and classes from which yours is not derived. This is obviously another feature that needs to be used judiciously, but can make your code look much more straightforward. **Listing 30** contains the source code for implicit conversions between an `int` and an object of type *Operation*.

Events

Defining, creating, and consuming events is very nicely supported in C#. **Listing 31** contains the important code snippets to define and create an event

Listing 29: A struct Type

```
public struct HelpvaluesItem {
  public string code, description;
  public HelpvaluesItem(string code, string description) {
    this.code = code;
    this.description = description;
  }
  public override string ToString() {
    return "Code: " + code + ", Description: " + description;
  }
}
```

Listing 30: A User-Defined Implicit Conversion

```
public static implicit operator int(Operation op) {
  for (int i = 0; i < ops.Length; i++) {
    if (ops[i] == op)
      return i;
  }
  return -1;  // Cannot ever happen
}
public static implicit operator Operation(int i) {
  if (i >= 0 && i < ops.Length)
    return ops[i];
  else
    throw new OverflowException("Invalid integer value");
}
```

Listing 31: Defining and Creating an Event

```
// Defining the event
public delegate void ChangedEventHandler(object sender, CategoriesEventArgs e);
public event ChangedEventHandler Changed;
protected virtual void OnChanged(CategoriesEventArgs e) {
  if (Changed != null)
    Changed(this, e);
}
// Create an actual event
OnChanged(new CategoriesEventArgs(category, Operation.Added));
```

Listing 32: Registering an Event Handler and Processing an Event

```
// Register the event handler
cats.Changed += new Categories.ChangedEventHandler(Changed);
// Process the event
private void Changed(object sender, Categories.CategoriesEventArgs e) {
  Console.Out.WriteLine("Event raised: " + e.Operation +
                        " " + e.Category.Code);
}
```

(classes *Operation* and *CategoriesEventArgs* are not listed here, but are part of Appendix B). **Listing 32** shows the complete code of a client application to register an event handler and process the event. For more details on event handling in C#, please see the Visual Studio .NET documentation.

Attributes

Most programming languages are static and cannot be extended. C# is different. There are standard attributes that can be associated with assemblies, types, methods, properties, and so on. You can define your own attributes to store arbitrary information related to your code, e.g., the field name in a database where a property should be stored. And, most importantly, you can retrieve the attributes at runtime using *Reflection*. I have mentioned one attribute earlier in this article, when I explained how to specify that all classes in an assembly should be checked for CLS-compliance by the compiler. There are many more standard attributes, and you can easily create and use your own.

The Sample Project

Learning a new programming language in my experience works best when you set yourself a non-trivial task — difficult enough so that you can try out most of the important features of the new language — and solve it. The task that I picked was to develop a component that could manage categories to be used for classification purposes. Categories should be allowed to form a hierarchy and the same category should be able to exist in multiple places in the hierarchy. Why do I need something like this? I always wanted to have a restaurant database, and obviously the cuisine(s) offered by a restaurant are one of the most important classification criteria. One restaurant can offer multiple cuisines, and there is obviously a hierarchical relationship between them, e.g., Szechuan food is a subcategory of Chinese food. On the other hand, I would like to be able to set up a category containing all spicy cuisines, which would also have to contain Szechuan food. When I actually want to search for a restaurant, I want to be capable of specifying a category higher up in the hierarchy and still find all restaurants classified with any cuisine contained in this category. In other words, when searching for spicy food, I should find all Szechuan restaurants even if they have not been explicitly classified as spicy, but just as Szechuan. This approach, allowing multiple cuisines to be assigned to a restaurant and including subcategories in searches, provides maximum convenience and

flexibility. Compare this with the nuisance involved in finding a suitable restaurant in most databases available on the Internet today!

The usefulness of this classification approach is not limited to managing a restaurant database, though. Similar requirements apply to many other classification scenarios.

I decided to have two classes in my component, one that encapsulates a single category (called *Category*) and one that encapsulates a collection of categories (called *Categories*). A category has code and description properties as well as a property that determines whether it is a root category, i.e., visible at the root level of the categories tree. And a category must keep track of its subcategories. The complete source code for classes *Category* and *Categories* is available in Appendices A and B. Bits and pieces of it have been used earlier in the article to exemplify various aspects of C#.

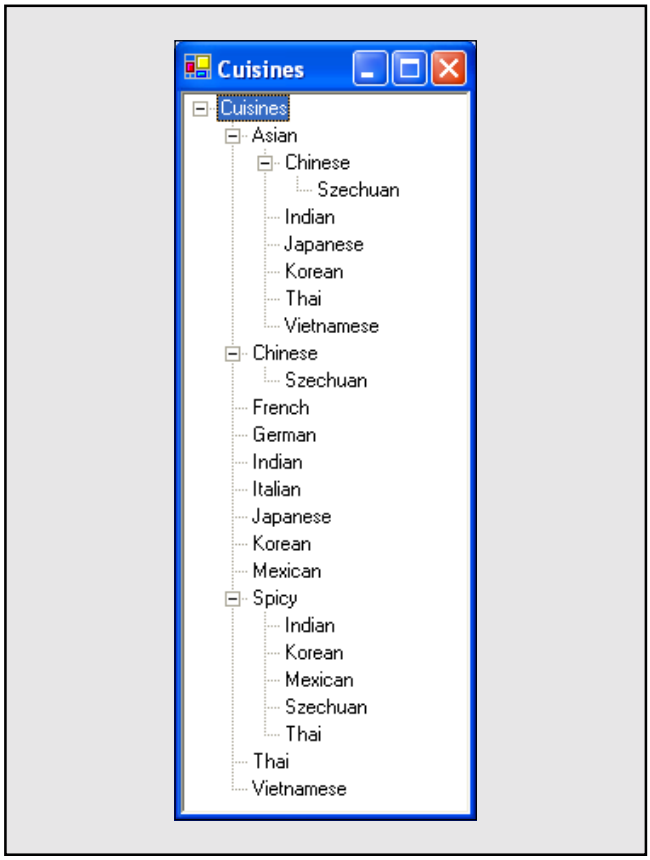
Displaying the Categories

In order to manipulate the categories in a GUI, I decided to use the *TreeView* control available in .NET. Dealing with trees always requires some sort of recursive programming, so I wrote the method *AddSubNodes()*, which can be called recursively to populate the tree with the complete categories hierarchy. **Listing 33** contains this

Listing 33: Code to Display the Cuisine Categories

```
private void Form1_Load(object sender, System.EventArgs e) {
  treeView1.BeginUpdate();
  treeView1.Nodes.Add(new TreeNode("Cuisines"));
  foreach (string key in newCats.GetAllRootCategoryKeys()) {
    Category catX = newCats[key];
    TreeNode node = new TreeNode(catX.Description);
    node.Tag = catX;
    treeView1.Nodes[0].Nodes.Add(node);
    AddSubNodes(node, catX, newCats);
  }
  treeView1.Nodes[0].ExpandAll();
  treeView1.EndUpdate();
}
private void AddSubNodes
             (TreeNode node, Category cat, Categories cats) {
  foreach (string key in cat.SubCategoryKeys) {
    Category subCat = cats[key];
    TreeNode newNode = new TreeNode(subCat.Description);
    newNode.Tag = subCat;
    node.Nodes.Add(newNode);
    AddSubNodes(newNode, subCat, cats);
  }
}
```

Figure 3 Displaying the Cuisines Tree



method and the code that gets executed when the appropriate form of the GUI application is loaded.

Figure 3 is a screenshot that shows you the program in action.

Serialization Using XML

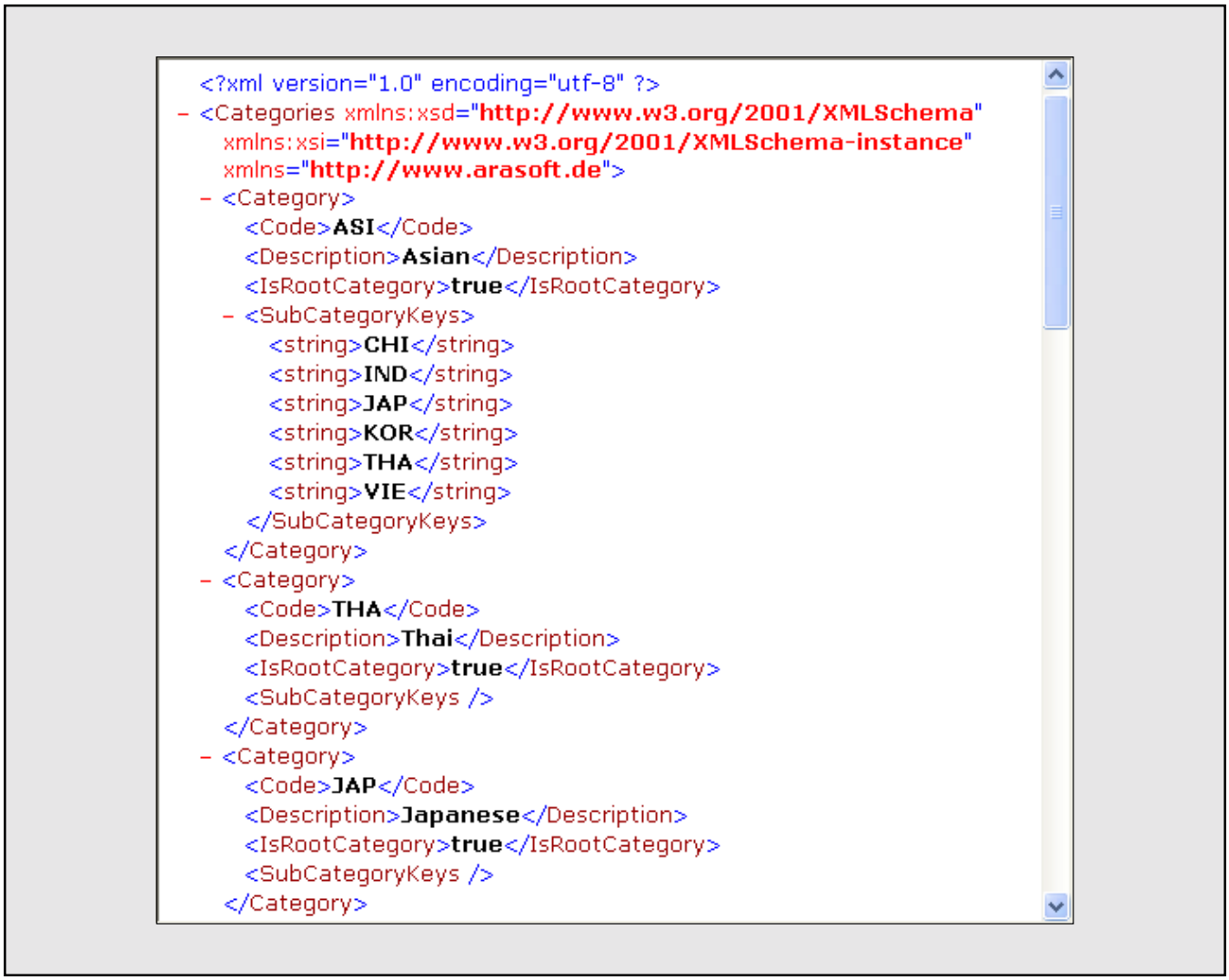
C# offers excellent support for using XML in your applications. For my sample project, I decided to store the categories in XML format to be able to take advantage of XSLT and other XML-based technologies. Using XML as a serialization format was very easy to accomplish, as you can see in **Listing 34**. Without any special code in the *Category* and *Categories* classes, it is possible to serialize and deserialize the categories.

Figure 4 shows a portion of the XML file displayed in Internet Explorer.

Listing 34: XML Serialization and Deserialization

```
// Serialize categories to XML
XmlRootAttribute root = new XmlRootAttribute("Categories");
root.Namespace = "http://www.arasoft.de";
XmlSerializer xs = new XmlSerializer(typeof (Categories), root);
const string fileName = @"c:\categories.xml";
TextWriter writer = new StreamWriter(fileName);
xs.Serialize(writer, cats);
writer.Close();
// Deserialize categories from XML
FileStream fs = new FileStream(fileName, FileMode.Open);
Categories newCats = (Categories) xs.Deserialize(fs);
fs.Close();
```


Figure 4 *XML Representation in Internet Explorer*



Conclusion

In this article, I have introduced what I consider to be the most important aspects of C#. Obviously, I have not covered everything there is to know about C#. If you are now — as I hope you to be — itching to learn more, I recommend the Visual Studio .NET documentation, which includes several useful tutorials. If you insist on me recommending a book — as many of you probably do — the best I have seen so far is:

Tom Archer; Andrew Whitechapel:
Inside C#, Second Edition,
Microsoft Press,
ISBN 0-7356-1648-5

Yes, it is a rather voluminous tome (873 pages), but it discusses some really advanced topics, while most C# books I have seen that are allegedly for advanced readers spend most of their pages and your time on topics that any decent Java programmer is already familiar with.

While there are many other factors that determine which development platform and programming language should be used, the amount of fun a developer has using the programming languages does and should play a role. In my opinion, C# is definitely more fun than Java, and I intend to use both languages in the future, C# for projects where Windows is the only required platform and Java for platform-independent projects.

Thomas G. Schuessler is the founder of ARAsoft (www.arasoft.de), a company offering products, consulting, custom development, and training to a worldwide base of customers. The company specializes in integration between SAP and non-SAP components and applications. ARAsoft offers various products for BAPI-enabled programs on the Windows and Java platforms. These products facilitate the development of desktop and Internet applications that communicate with R/3. Thomas is the author of SAP's BIT525 "Developing BAPI-enabled Web Applications with Visual Basic" and BIT526 "Developing BAPI-enabled Web Applications with Java" classes, which he teaches in Germany and in English-speaking countries. Thomas is a regularly featured speaker at SAP TechEd and SAPPHIRE conferences. Prior to founding ARAsoft in 1993, he worked with SAP AG and SAP America for seven years. Thomas can be contacted at thomas.schuessler@sap.com or at tgs@arasoft.de.

Appendix A: Class Category

```
using System;
using System.Collections.Specialized;

namespace De.ARAsoft.Categories
{
  public class Category : IComparable  {

    private string code;
    private string description;
    private StringCollection cats = new StringCollection();
    private bool isRootCategory;
    internal Categories parent;

    public Category()  {}

    public Category(string code, string description,
                    bool isRootCategory)  {
      this.code = code;
      this.description = description;
      this.isRootCategory = isRootCategory;
    }

    public string Code {
      get {
        return code;
      }
      set {
        if (value == null)
          throw new ArgumentNullException();
        if (code == null)
          code = value;
        else
          throw new InvalidChangeException
            ("Field 'Code' can only be set once.");
      }
    }

    public string Description {
```

```
    get {
      return description;
    }
    set {
      description = value;
    }
  }

  public bool IsRootCategory {
    get {
      return isRootCategory;
    }
    set {
      isRootCategory = value;
    }
  }

  public string[] SubCategoryKeys {
    get {
      string[] array = new string[cats.Count];
      cats.CopyTo(array, 0);
      Array.Sort(array);
      return array;
    }
    set {
      cats = new StringCollection();
      cats.AddRange(value);
    }
  }

  public bool AddSubCategory(Category cat) {
    if (cat == null ) return false;
    if (cats.Contains(cat.Code)) return false;
    if (IsCyclical(cat))
      throw new InvalidOperationException
        ("Cyclical references are not allowed.");
    cats.Add(cat.Code);
    return true;
  }
  public static Category operator + (Category me, Category sub) {
    me.AddSubCategory(sub);
    return me;
  }

  public bool RemoveSubCategory(Category cat) {
    if (cat == null ) return false;
    if ( ! cats.Contains(cat.Code) ) return false;
```



```
      cats.Remove(cat.Code);
      return true;
    }
    public static Category operator - (Category me, Category sub) {
      me.RemoveSubCategory(sub);
      return me;
    }

    public void RemoveAllSubCategories() {
      cats = new StringCollection();
    }

    private bool IsCyclical(Category cat) {
      Categories cats = cat.parent;
      StringCollection subKeys = new StringCollection();
      GetAllSubCategoryKeysRecursive(cat, subKeys);
      return subKeys.Contains(this.Code) ? true : false;
    }
    private void GetAllSubCategoryKeysRecursive
                (Category cat, StringCollection keys) {
      string[] subKeys = cat.SubCategoryKeys;
      for (int i = 0; i < subKeys.Length; i++) {
        string key = subKeys[i];
        if ( ! keys.Contains(key) )
          keys.Add(key);
        GetAllSubCategoryKeysRecursive(cat.parent[key], keys);
      }
    }

    public override string ToString() {
      return "Code: " + code + ", Description: " + description;
    }

#region IComparable
    public int CompareTo(Object o) {
      if ( ! (o is Category) )
        throw new ArgumentException();
      return String.Compare(this.Code, ((Category)o).Code);
    }
#endregion

#region Exceptions
    public class InvalidChangeException : ApplicationException {
      public InvalidChangeException() {
      }
      public InvalidChangeException(string message)
        : base(message) {
```

```
      }
      public InvalidChangeException(string message, Exception inner)
        : base(message, inner) {
      }
    }
#endregion
  }
}
```


Appendix B: Class Categories

```
using System;
using System.Collections;
using System.Collections.Specialized;

[assembly: CLSCompliantAttribute(true)]

namespace De.ARAsoft.Categories
{
  public class Categories :
              System.Collections.Specialized.NameObjectCollectionBase,
              IEnumerable {

    int version = 0;

    public Categories()  {
    }

    public Category Add(Category category) {
      if (this.BaseGet(category.Code) == null) {
        this.BaseAdd(category.Code, category);
        version ++;
        if (category.parent != null)
          throw new InvalidOperationException
          ("A category cannot belong to multiple Categories objects.");
        category.parent = this;
        OnChanged(new CategoriesEventArgs(category, Operation.Added));
      }
      return category;
    }
    public static Categories operator +
                (Categories categories, Category category) {
      categories.Add(category);
      return categories;
    }

    public Category Remove(Category category) {
      if (this.BaseGet(category.Code) != null) {
        this.BaseRemove(category.Code);
        version ++;
```

```
      category.parent = null;
      OnChanged(new CategoriesEventArgs
                  (category, Operation.Removed));
    }
    return category;
  }
  public static Categories operator –
              (Categories categories, Category category) {
    categories.Remove(category);
    return categories;
  }

  public Category Get(int index) {
    return (Category)this.BaseGet(index);
  }
  public Category Get(string code) {
    return (Category)this.BaseGet(code);
  }

  public string[] GetAllKeys() {
    return this.BaseGetAllKeys();
  }
  public string[] GetSortedKeys() {
    string[] keys = this.BaseGetAllKeys();
    Array.Sort(keys);
    return keys;
  }

  public string[] GetAllRootCategoryKeys() {
    StringCollection keys = new StringCollection();
    foreach (string key in this.GetSortedKeys()) {
      Category cat = this[key];
      if (cat.IsRootCategory) {
        keys.Add(key);
      }
    }
    string[] array = new String[keys.Count];
    keys.CopyTo(array, 0);
    return array;
  }

  public override string ToString() {
    string result = "Categories:\r\n";
    string[] keys = this.GetSortedKeys();
    foreach (string key in keys) {
      Category cat = this.Get(key);
      result += "\t" + cat.ToString() + "\r\n";
    }
    return result;
```



```
    }

#region Indexers
    public Category this [int index] {
      get {
        if (index < 0 || index >= this.Count)
          return null;
        else
          return this.Get(index);
      }
    }
    public Category this [string code] {
      get {
         return this.Get(code);
      }
    }
#endregion

#region Enumerator
    new public Enumerator GetEnumerator() {
      return new Enumerator(this, version);
    }
    IEnumerator IEnumerable.GetEnumerator() {
      return GetEnumerator();
    }

    public class Enumerator : IEnumerator {

      private int index;
      private Categories collection;
      private string[] keys;
      private int version;

      public Enumerator(Categories categories, int version) {
        collection = categories;
        keys = categories.GetSortedKeys();
        this.version = version;
        Reset();
      }

      public bool MoveNext() {
        CheckVersion();
        index++;
        return (index < collection.Count);
      }

      public void Reset() {
        CheckVersion();
        index = -1;
```

```
      }

      public Category Current {
        get {
          CheckVersion();
          if (index == -1) throw new InvalidOperationException
                                      ("Use MoveNext() first.");
          return collection[keys[index]];
        }
      }
      object IEnumerator.Current {
        get {
          CheckVersion();
          return Current;
        }
      }

      void CheckVersion() {
        if (version != collection.version)
          throw new InvalidOperationException
                    ("Collection changed during enumeration.");
      }
    }
#endregion

#region Events
    public struct Operation {

      private static Operation[] ops = {
        new Operation("Added"),
        new Operation("Removed"),
      };

      private string type;

      private Operation(string type) {
        this.type = type;
      }

      public static Operation Added {
        get {
          return ops[0];
        }
      }
      public static Operation Removed {
        get {
          return ops[1];
        }
      }
```



```
  public static bool operator ==(Operation x, Operation y) {
    return x.Equals(y);
  }
  public static bool operator !=(Operation x, Operation y) {
    return !(x.Equals(y));
  }
  public override bool Equals(Object obj) {
    return obj is Operation && base.Equals(obj);
  }
  public override int GetHashCode() {
    return type.GetHashCode();
  }

  public static implicit operator int(Operation op) {
    for (int i = 0; i < ops.Length; i++) {
      if (ops[i] == op)
        return i;
    }
    return -1;  // Cannot ever happen
  }
  public static implicit operator Operation(int i) {
    if (i >= 0 && i < ops.Length)
      return ops[i];
    else
      throw new OverflowException("Invalid integer value");
  }

  public override string ToString() {
    return type;
  }
}

public class CategoriesEventArgs : EventArgs {

  private Category category;
  private Operation operation;

  public CategoriesEventArgs
         (Category category, Operation operation) {
    this.category = category;
    this.operation = operation;
  }

  public Category Category {
    get {
      return category;
    }
  }
```

```
      public Operation Operation {
        get {
          return operation;
        }
      }
    }

    public delegate void ChangedEventHandler
                         (object sender, CategoriesEventArgs e);
    public event ChangedEventHandler Changed;
    protected virtual void OnChanged(CategoriesEventArgs e) {
      if (Changed != null)
        Changed(this, e);
    }
#endregion
  }
}
```

