C# Versus Java

Dr. Dobb's Journal February 2001

Do we really need another language?

By Marc Eaddy

Marc is a project leader developing real-time stock market applications at ILX Systems. He can be contacted at me133@ columbia.edu.

Microsoft describes C# ("C sharp") as a "simple, modern, object-oriented, and type-safe programming language derived from C and C++." That statement would apply equally well to Java. In fact, after comparing the two languages, it's obvious that prerelease descriptions of C# resemble Java more than C++. As Example 1 illustrates, the language features and syntax are similar. Example 1(a) is the canonical "Hello World" program in Java, while Example 1(b) is the program in C#.

(a)
```
class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, World!");
   }
}
```

(b)
```
class HelloWorld {
   public static void Main(string[] args) {
      System.Console.WriteLine("Hello, World!");
   }
}
```

Example 1: Hello World. (a) in Java; (b) in C#.

But the resemblance goes beyond syntax, keywords, and delimiters. It also includes features that Java and C# hold in common, such as:

- Automatic garbage collection.
- Reflection for type information discovery.
- Source code is compiled to an intermediate bytecode.
- Just-in-Time (JIT) compilation compiles bytecode into native code.
- Everything must be in a class — no global functions or data.
- No multiple inheritance, although you can implement multiple interfaces.
- All classes derive from *Object*.

- Security for restricting access to resources.
- Exceptions for error handling.
- Packages/namespaces for preventing type collision.
- Code comments as documentation.
- Arrays are bounds checked.
- GUI, networking, and threading support.
- No uninitialized variables.
- No pointers.
- No header files.

C#: The Evolution of Visual J++

Why does Microsoft think we need another language? When Microsoft introduced Visual J++ in October 1996, it threw lots of resources into the project. Their efforts produced the fastest JVM on the market and the Windows Foundation Classes (WFC), a set of Java classes that wrapped the Win32 API. Not coincidentally, Anders Hejlsberg, the project leader for WFC (and most famous as the author of Turbo Pascal), is the chief architect for C#.

Microsoft decided to make changes to Java to integrate it more closely with Windows. Some of the changes — interfacing seamlessly with COM, refusing to support RMI and JNI, and adding delegates — caused it to break compliance with the Java Standard. Consequently, Sun Microsystems sued Microsoft in October 1997 for violating its Java license agreement. This doomed Microsoft's future development of Java and Visual J++. However, Microsoft decided to take its advances in the Java language, Java compiler, and JVM and morph them into an even more ambitious project — Microsoft .NET.

Microsoft .NET

The term "Microsoft .NET" is similar to "Windows DNA" in that it refers to many things: a business strategy, development model, marketing pitch, development platform, and language run time. But whereas DNA is more of a "best practice" approach to development, at the heart of .NET is the Common Language Runtime (CLR), a set of operating-system services and an execution engine that provides run-time support for .NET programs. You can think of the Common Language Runtime as fulfilling the same role as the Java virtual machine.

Programs written in C# are compiled into an intermediate language called "MSIL," the equivalent to Java bytecode or Visual Basic p-code. Any language that can be compiled to MSIL can take advantage of the CLR features such as garbage collection, reflection, metadata, versioning, events, and security, to name a few. In addition, a class written in one language can actually inherit from a class written in another language and override its methods.

Although this article is about C#, keep in mind that the class libraries and the CLR features are usable by any language that has an MSIL compiler. Initially, Microsoft will provide MSIL compilers for C#, Visual Basic, JScript, and Managed C++. Third-party vendors have also developed .NET compilers for a number of languages, including: Java (Rational), Eiffel (Interactive Software Engineering and Monash University), Perl (ActiveState), Python (ActiveState), Scheme (Northwestern University), Smalltalk (Quasar Knowledge Systems), Cobol (Fujitsu), Component Pascal (Queensland University of Technology), APL (Dyalog), Standard ML (Microsoft Research-Cambridge), Mercury (University of Melbourne), and Oberon (ETH Zentrum).

It's interesting that Microsoft is pushing cross-language development while Sun/Java pushes cross-

platform development.

However, both approaches have their share of problems. Writing components in multiple languages always entails some interoperability problems. Moreover, there is always the problem of what to do when your Scheme programmer moves on to greener pastures. Cross-platform development has also never been flawless, as Java programmers well know, especially in the areas of GUIs and threading.

C# Similarities to Java

In addition to sharing a number of features, most of the keywords in Java have their C# counterpart. Some keywords are identical; for example, *new*, *bool*, *this*, *break*, *static*, *class*, *throw*, *virtual*, and *null*. This is expected since these keywords are derived from C++. Interestingly, many keywords in Java that do not have direct C++ equivalents such as *super*, *import*, *package*, *synchronized*, and *final*, have different names in C# (*base*, *using*, *namespace*, *lock*, and *sealed*, respectively). Table 1 summarizes some of the keywords and features that exist in both languages, but look slightly different in C#.

Keyword/Feature	Java	C#

Inheritance	class Foo
	extends Bar
	implements IFooBar {
	}
		class Foo : Bar, IFooBar {
		}

Referring to the base class	super.hashCode();	base.GetHashCode();
Importing types	import java.util;	using System.Net;

Namespaces	package MyStuff;
	
	class MyClass {
	...
	}
		namespace MyStuff {
		class MyClass {
		...
		}
		}

Prevent inheritance	final class Foo { }	sealed class Foo { }
Constants	final static int MAX = 50;	const int MAX = 50;
Code comments	/** * Convert string to * uppercase. * * @param s The string to * convert * * @return The uppercase * string * */	/// <summary> /// Convert string to /// uppercase. /// </summary> /// <param name="s"> /// The string to convert /// </param> /// <returns> /// The uppercase string /// </returns>
Deprecated classes or methods	@deprecated Don't use!	[obsolete("Don't use!")]

Synchronization	synchronized(this) {
	++refs;
	}
		lock(this) {
		++refs;
		}

Table 1: Cosmetic differences between Java and C#.

The *Object* Class

Another good example of cosmetic differences is the *System.Object* class in C#, which has the exact same methods as the *java.lang.Object* class in Java except they are spelled differently. The *clone* method in Java is called *MemberwiseClone* in C#, Java *equals* is *Equals* in C#, *finalize* is *Finalize*, *getClass* is *getType*, *hashCode* is *GetHashCode*, and *toString* is *ToString*.

Mere coincidence? Well, according to Anders Hejlsberg in a recent interview, "C# is not a Java clone" ("Deep Inside C#: An Interview with Microsoft Chief Architect Anders Hejlsberg," by John Osborn, http://windows.oreilly.com/news/hejlsberg_0800.html). Right, it's a *MemberwiseClone*.

Access Modifiers

C# specifies access modifiers inline as part of the member definition just like Java does, instead of in a block like C++. The modifiers *public* and *private* have the exact same meanings in all three languages. However, "protected access" in Java is called "protected internal" in C#, and "package access" in Java is called "internal" in C#. C#'s protected modifier gives access to any subclass, even if it is not in the same program. Another difference is that package access is the default for Java while private is the default for C#.

Exceptions

Similar to Java, *try* blocks in C# support the *finally* clause. There is no *throw*s clause in C#, so effectively, all exceptions are unchecked. You aren't forced to handle any exceptions. The opinion at Microsoft, according to one employee, is that forcing developers to handle exceptions does more harm than good. This leads to many *catch(Exception e)* exception handlers that don't do anything useful.

This is an unfortunate decision because the *throw*s clause makes it obvious which exceptions can be thrown by a method and are part of the method's contract. Otherwise, you are forced to read the called method's source code to know which exceptions can be thrown.

C# Improvements Over Java

Since C# and Java look and act alike in many ways, why bother using C# at all? As you might expect from a successor language, C# makes improvements over Java in some areas. In addition to adding innovative features, C# has a simplified syntax for things such as iteration, events, and treating primitive types like objects, which reduces the amount of code you need to write.

Reflection, Metadata, And Custom Attributes

Java and C# compilers both emit metadata with the class bytecodes to support reflection. Reflection provides the ability to obtain type information dynamically and makes it possible for the run-time system to automatically provide implementations for run-time type identification, dynamic method invocation, serialization, marshaling, and scripting support.

Microsoft has extended the notion of metadata with Custom Attributes, which let you markup a class, method, method parameter, and just about anything else, with extra information that can be accessed at run time. When compiled, the attributes are combined with the EXE/DLL itself, so it is not possible for them to be out of sync with the code. Attributes eliminate the need to maintain separate IDL files and

type libraries. The C# compiler supports attributes for interoperating with legacy COM objects (Internet time indeed!) and the Win32 APIs, object serialization, conditional code execution, and deprecating program entities.

An example of where attributes are useful is specifying the XML schema to use when serializing a class (see Listing One). As you can see, the class is marked-up directly with bracketed attributes to indicate what XML node and attribute names to use when serializing or deserializing an *Album* object. Listing Two is a program that creates an *Album* and serializes it. Listing Three is the resulting XML file. (The command line for getting the XML serialization example to compile using the prerelease C# compiler is *csc.exe /r:System.Xml.dll /r:System.dll /r:System.Xml .Serialization.dll Albums.cs.*)

Versioning

Versioning is a feature that is much needed in the Windows and Java development space. Windows developers are accustomed to "DLL Hell," where mismatched versions of DLLs can cause all kinds of application problems. Java developers are also familiar with deprecated APIs and incompatible versions of serialized objects.

.NET promises to fix these problems by letting you specify version dependencies between components and by supporting side-by-side execution of multiple versions of a component. This is similar to the versioning capabilities specified in the Java Product Versioning Specification, except that compile-time, installation time, and run-time enforcement is built-in.

Assertions

Assertions are useful for sanity checking and enforcing the preconditions, postconditions, and invariants of Design By Contract. They often catch bugs introduced by faulty design/logic, incorrect assumptions, integration, and code maintenance. Assertions are not supported in Java, although the Java Community Process is working on it. Listing Four demonstrates assertions in C#, and Figure 1 shows the assertion dialog that appears. It lists the filename and line number where the assertion occurred, the general and detailed error messages, and the stack trace.

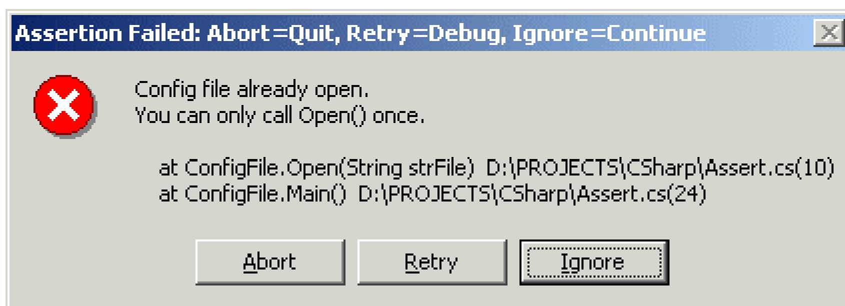

Figure 1: The assertion dialog.

ref and *out* Parameters

You can pass parameters by reference or specify that they are output parameters by using the *ref* or *out* modifiers. Since Java only allows pass-by-value, you have to perform silly tricks to get the same effect, such as using return values, passing in a one-element array, or putting objects inside wrapper classes. An example of pass-by-reference is: *void swap(ref long n1, ref long n2);*.

Virtual Methods

By default, all methods in Java are virtual and can be overridden by a derived class. In C#, as in C++, methods must be explicitly declared virtual. A common error in Java and C++ when overriding methods is when someone inadvertently modifies the signature of the base class method. Because the signatures don't match, the derived class method hides instead of overriding the base class method. C# turns this into a compile-time error by requiring that the derived class use the *override* keyword to override a virtual method. In addition, a compile-time warning is given if a method in the derived class hides a method in the base class. In this case, you can use the *new* keyword to remove the warning.

enums

Java does not implement C/C++-style enums because its designers claim that *enums* are not object oriented. Face it, *enums* are a lot more typesafe than Java static final *int* constants. *enums* are not only present in C#, they are typesafe, can have the ++, -, <, and > operators applied to them, and can be converted to and from a string.

decimal Data Types

C# decimal data types are 128 bits and have a greater precision and smaller range than floating-point types. They are particularly useful for financial applications.

switch Statements

By default, case labels in a C# *switch* statement do not fall through to the next case label. In addition to being able to specify an integer in the *switch* and *case* statements, you can also specify a string.

C# Syntactic Sugar

A method is just a function in which the first argument is a pointer to the object. Instead of calling *foo (object,x,y)*, C++/Java/C# allow you to write *o*bject.foo(x,y). This is an example of how syntactic sugar makes object-oriented development easier. C# provides some sugar of its own to make component-oriented development easier.

Delegates and Events

Delegates are a significant innovation for C#. They are object-oriented function pointers that can reference *static* or *instance* methods. They provide a type-safe mechanism for implementing callback functions and events. C# supports single-cast, multicast, synchronous, and asynchronous delegates.

Java provides for events by using the JavaBeans event model and adapter classes. Event handling in C# is simpler and only requires you to implement individual methods instead of entire interfaces. Delegates were first introduced as a nonstandard feature of Visual J++, much to the dismay of Sun Microsystems, which derided delegates as being "not object oriented." For Sun's delegates argument, see http://www.javasoft.com/docs/white/delegates.html, and http://msdn.microsoft.com/visualj/technical/articles/delegates/truth.asp for Microsoft's rebuttal. (Some even conjecture that the delegates issue was the turning point that lead to the Java lawsuit.)

The *event* keyword introduces a delegate into your class that lets you fire events. Listing Five is a

sample C# program that uses an event to notify a *StockTracker* when the price of a stock changes.

Value Types

Value types in C# (*long*, *int*, *char*) can be treated just like reference types without requiring special wrapper classes as in Java (*java.lang.Long*, for example). The compiler implicitly converts value types into objects (and vice versa) on demand through a process called "boxing and unboxing." However, this incurs no overhead if the value type is never treated like an object. This allows C# developers to view the world as a unified type system in which all data types derive from *object*. Example 2 shows examples of how each language lets you treat primitive types as objects.

(a)
```
int i = 10;
System.out.println((new Integer(i)).hashCode()); // Use wrapper class
```

(b)
```
int i = 10;
System.Console.WriteLine(i.GetHashCode()); // Conversion is implicit
```

Example 2: Value types. (a) in Java; (b) in C#.

Properties

For our purposes, properties and fields refer to two different concepts. A field is a class data member (*long size,* for example), whereas a property is a pair of *getter* and *setter* access methods that provide access to a field or calculate the needed values. Properties in C# are similar to JavaBeans properties in that access to them is controlled through access methods. However, in C#, properties are directly supported by the language instead of relying on reflection and naming conventions, as in JavaBeans. This leads to a clean and concise syntax that makes data hiding a breeze. Example 3 illustrates how the *Name* property is implemented using the *name* field in both Java and C#.

(a)
```
String name;
public String getName() { return name; }
public void setName(String value) { name = value; }

Ex:
obj.setName("Marc");
if (obj.getName() != "Marc")
    throw new Exception("That ain't me!");
```

(b)
```
string name;
public string Name {
    get { return name; }
    set { name = value; } // 'value' is the new value
}

Ex:
obj.Name = "Marc";
if (obj.Name != "Marc")
    throw new System.Exception("That ain't me!");
```

Example 3: Implementing the Name property using the name field: (a) in Java; (b) in C#.

The advantages of the C# property style is that the *getter/setter* methods are located in the same block of code and have a more intuitive syntax because properties are accessed exactly like normal fields. JavaBeans properties also look like normal fields when accessed from scripts or in a visual design tool, but not when accessing them from Java code.

foreach-Style Iteration Syntax

The *foreach* statement in C# (borrowed from Visual Basic) lets you easily enumerate over classes that support the Enumerable interface, which includes arrays and collections. This eliminates the need to write *for(int i=0; i < ary.length; ++i)* and the *getIterator/hasNext/next* triad as you have to in Java and C++. Listing Six demonstrates iteration in Java, while Listing Seven demonstrates *foreach*-style iteration in C#.

String Formatting

A great example of C# syntactic sugar is string formatting. Java provides the *MessageFormat* class to allow *printf*-style formatting. The syntax in C# is much cleaner because you can pass a variable number of parameters to the function. Example 4 shows off the string formatting features and variable parameters of C#. Both examples output *Error: File not found. (Code 2).*

```
(a)
// Create an object array
Object args[] = { "File not found.", new Integer(2) };
// Format the message
System.out.println(
   java.text.MessageFormat.format("Error: {0} (Code {1})",args));

(b)
System.Console.WriteLine("Error: {0} (Code {1})", "File not found.", 2);
```

Example 4: String formatting. (a) in Java; (b) in C#.

Operator Overloading and Cast Operators

Operator overloading allows you to redefine the semantics of operators (+, -, +=, ==, and so on) for a given type. When used properly, operator overloading can provide an intuitive syntax for treating user-defined types like primitive types. C# provides the *operator* keyword for this purpose.

In addition to operator overloading, C# lets you define cast operators. For example, a class called *Fraction* can provide a cast operator that converts a *Fraction* into a *double*.

C# Performance Improvements

Several C# features result in performance improvements over Java. These include:

- Never interpreted. The advent of Just-in-Time (JIT) compilers means that Java bytecodes are usually compiled directly into native code by the JVM instead of being interpreted. One difference with the Common Language Runtime is that the bytecode is never interpreted, it is always JITed. Microsoft has a proven track record for writing the fastest JVM, so you can expect their MSIL JITer to be fast.
- Native platform support for Win32 and COM is an improvement over the performance of J/Direct

and the Java Native Interface.

- The *stackalloc* operation is similar to *new*, except that it allocates memory for an array on the stack instead of the heap. Stack allocation is much faster than heap allocation and is not subject to garbage collection.
- A *struct* data type is exactly like an object, except that *struct*s are value types instead of reference types and do not support inheritance. They can be allocated on the stack just like primitive types and do not require the extra *vtable* overhead. Use *struct*s to represent simple types, such as *Point*, *Rect*, *Fraction*, and so on. They lose their effectiveness when their size goes above 16 bytes.
- Code in C# can be marked *unsafe* to allow it to have direct memory access and prevent the garbage collector from interfering.

What's Not to Like?

Neither Java nor C# support generic types (templates). However, a proposal has been submitted to the Java Community Process, and Microsoft Research at Cambridge is said to be developing a solution for .NET.

Like J/Direct, the P/Invoke class library provides the ability for C# to talk to native Win32 APIs and DLLs. While this is a boon for interoperability, it requires writing C# function prototypes that have the same signatures as the API functions. For example, handles (HWND, for example) are approximated using a C# *int* and pointers are approximated with *ref* parameters. This technique is prone to signature mismatches that can cause unpredictable behavior and crashes.

Similar to P/Invoke, COM Interop provides wrapper classes for allowing C# to use COM objects and vice versa. Unfortunately, certain types of COM interfaces are not 100-percent compatible with wrappers, which causes an impedance mismatch.

Conclusion

C# is Java with some nifty features, innovations, syntactic sugar, and performance enhancements thrown in. C#'s elegance, simplicity, and power promises to deliver Windows C++ developers from the pits of boilerplate COM code and memory violations to the land of RAD. For Java programmers, C# picks up where Java leaves off by providing a high-performance, component-oriented language that integrates tightly with Windows. Let's hope C# is as much fun as Java is to work with.

DDJ

Listing One

```
using System.NewXml;
using System.Xml.Serialization;
[XmlRoot("album", Namespace="music")]
public class Album {
    [XmlElement("artist")]
    public string artist;
    [XmlElement("title")]
    public string title;
    [XmlArray("songs"), XmlArrayItem("song")]
    public string[] songs;
}
```

Back to Article

Listing Two

```
using System.Xml.Serialization;
using System.IO;
class TestAlbum
{
    public static void Main() {
        Album album = new Album();
        album.artist   = "Sasha";
        album.title    = "Xpander";
        album.songs    = new string[5];
        album.songs[0] = "Xpander Edit";
        album.songs[1] = "Xpander";
        album.songs[2] = "Belfunk";
        album.songs[3] = "Rabbitweed";
        album.songs[4] = "Baja";
        // Serialize the object to a file
        FileStream fs = new FileStream("Album.xml", FileMode.Create);
        XmlSerializer serializer = new XmlSerializer(typeof(Album));
        serializer.Serialize(fs, album);
    }
}
```

Back to Article

Listing Three

```
<?xml version="1.0"?>
<album xmlns:xsi=http://www.w3.org/1999/XMLSchema-instance
xmlns="music">
  <artist>Sasha</artist>
  <title>Xpander</title>
  <songs>
    <song>Xpander Edit</song>
    <song>Xpander</song>
    <song>Belfunk</song>
    <song>Rabbitweed</song>
    <song>Baja</song>
  </songs>
</album>
```

Back to Article

Listing Four

```
using System.Diagnostics;
class ConfigFile {
    bool isFileOpen;
    public void Open(string strFile) {
        // Pre-conditions
        Debug.Assert(!isFileOpen, "Config file already open.",
            "You can only call Open() once.");
        Debug.Assert(strFile.Length > 0);
        isFileOpen = true;
        // ...
    }
    public static void Main() {
```

```
        ConfigFile file = new ConfigFile();
        file.Open("Joe.xml");
        file.Open("Joe.xml"); // Causes an assertion!
    }
}
```

Back to Article

Listing Five

```
using System;
delegate void PriceDecreasedDelegate(string name, long newPrice);
    // Called when the price drops
delegate void PriceIncreasedDelegate(string name, long newPrice);
    // Called when the price increases
class Stock {
    // Holds the price of a stock
    public Stock(string stockName, long startPrice) {
        name = stockName;
        price = startPrice;
    }
    public long Price {
        get { return price; }
        set {
         if (value > price &&
           Fire_OnPriceIncreased != null) {
           Fire_OnPriceIncreased(name, value); // Inform listeners
         } else if (value < price &&
           Fire_OnPriceDecreased != null) {
           Fire_OnPriceDecreased(name, value); // Inform listeners
         }
         price = value;  // Update the price
        }
    }
    // DATA
    string name;
    long price;
    public event PriceDecreasedDelegate Fire_OnPriceDecreased = null;
    public event PriceIncreasedDelegate Fire_OnPriceIncreased = null;
}
class StockTracker {
    // Outputs a message when the stock price changes
    public StockTracker(Stock stock) {
        // Connect the Stock events to our event handlers
        stock.Fire_OnPriceDecreased +=
            new PriceDecreasedDelegate(OnPriceDecreased);
        stock.Fire_OnPriceIncreased +=
            new PriceIncreasedDelegate(OnPriceIncreased);
    }

    // Signature of event handler must match the
    // PriceDecreasedDelegate delegate declaration
    private void OnPriceDecreased(string name, long val) {
     Console.WriteLine("Price of {0} dropped to {1}!", name, val);
    }
    private void OnPriceIncreased(string name, long val) {
        Console.WriteLine("Price of {0} rose to {1}!", name, val);
    }
}
```

```
class StockTester {
    // Test the events
    public static void Main() {
        Stock ibm = new Stock("IBM", 100);
        StockTracker tracker = new StockTracker(ibm);

        // Update the stock price
        ibm.Price = 125; // Outputs "Price of IBM rose to 125!"
        ibm.Price = 90;  // Outputs "Price of IBM dropped to 90!"
    }
}
```

Back to Article

Listing Six

```
import java.util.*;
class Iterate {
    public static void main(String[] args) {

        // Enumerate command-line args array
        for(int i = 0; i < args.length; ++i)
            System.out.println(args[i]);

        // Create a linked list
        LinkedList list = new LinkedList();
        list.add("Cube Farm");
        list.add("Sasha & Digweed");

        // Enumerate the list
        ListIterator it = list.listIterator(0);
        while (it.hasNext())
            System.out.println(it.next());
    }
}
```

Back to Article

Listing Seven

```
using System.Collections;
class Iterate {
    public static void Main(string[] args) {

        // Enumerate command-line args array
        foreach (string arg in args)
            System.Console.WriteLine(arg);

        // Create a linked list
        ObjectList list = new ObjectList();
        list.Add("Cube Farm");
        list.Add("Sasha & Digweed");

        // Enumerate the list
        foreach (string str in list)
            System.Console.WriteLine(str);
    }
}
```

Back to Article

Copyright © 2004 CMP Media LLC, *Dr. Dobb's Journal's* Privacy Policy, Terms of Service,
Comments: webmaster@ddj.com
SDMG Websites: BYTE.com, C/C++ Users Journal, Dr. Dobb's Journal, MSDN Magazine, Sys Admin, SD Expo, SD Magazine, Unixreview, Windows Developer Network, New Architect