

J#.NET - Advantages for the Java Developer



By [Narayana Rao Surapaneni](#)

Date: Jan 11, 2002

Article is provided courtesy of [InformIT](#).

[Return to the article](#)

This article describes Microsoft's soon-to-be-released J#.NET tool for compiling Java-like code. Several examples are provided to illustrate how a Java developer could leverage J#.NET with existing skills.

October, 2001 marked the beginning of a new era for Java developers. In its efforts to bring Java developers into the fold of the .NET framework, Microsoft released a new tool named J#.NET (pronounced "J sharp dot net"). The tool compiles Java-like code into Intermediate Language, which targets the CLR (Common Language Runtime Environment) of the Microsoft .NET platform. The .NET Technology Group of Patni Computer Systems has evaluated this tool from many perspectives and feels that this tool places Java developers in a very comfortable position, irrespective of the outcome of the technology feud between two software giants.

Introduction

The Java language has revolutionized the way we program applications for the Internet. Two great ideas make this language so promising:

- Applications written using Java run on multiple different platforms.
- Automatic memory management or garbage collection (GC) comes as a great relief for developers.

Other than these two issues, most of the remaining Java features can be implemented using the majority of other languages. Since its first public release in 1995, the Java language has been maturing and consolidating its position in the market as an increasing number of organizations realize its built-in strengths.

In the beginning, there were efforts to dilute Java's increasing popularity and to prove it to be just like any other language. Various sources attempted to prove that Java applications don't perform alike on all platforms, especially with respect to speed. But Java stood the test of time and emerged even stronger. Ironically, Microsoft stood isolated, even after having produced the world's fastest and best Java compiler and JVM implementation. (Microsoft's Java compiler can compile 10,000 lines of code per second.) Much of the Java API competes directly with Microsoft's API: JDBC with ODBC, JTAPI with TAPI, JDO with ADO, JSP with ASP, Bean with COM, and so on.

Though Java has had huge success, it has also left some problems unattended. Sun initially stated that with Java, "write once run anywhere" is possible; after introducing J2EE, however, Sun admitted that the "one size fits all" plan doesn't work. Java also failed to capture the desktop application market. When developing OS-specific applications, Java was not chosen over other languages (on Windows, VB and VC++; on UNIX, C, C++, and so on) because many developers consider performance to be a key issue; compilers specific to an operating system would run much faster than the Java compiler. Further, the way in which Java is implemented by different vendors is not uniform across the industry.

Many companies have joined hands in strengthening Java by bringing out various application servers, despite Microsoft producing their first application server back in 1996 (Microsoft Transaction Server, abbreviated MTS). All of these events may have contributed to a rethinking at Microsoft, to deliver a much better technology based on open standards. (Despite Java's open-mindedness toward the Open Source community, the Java owners never seemed to give serious thought to getting Java standardized by the prevalent standardizing committees. Of course, presently there are efforts to get it standardized by ISO and other standardizing committees. Only Java's original implementers might know the reasons.)

To combat emerging technologies by other organizations, retain its leadership in desktop computing, and consolidate the server market, Microsoft unveiled its vision a year ago: the .NET initiative. The .NET platform comprises the tools you need to create and run XML web services. It has three main components:

- **The .NET Framework and Visual Studio.NET**

These are the developer tools to build XML web services. The .NET Framework is the set of programming interfaces at the heart of the Microsoft .NET platform; Visual Studio .NET is a multi-language suite of programming tools.

- **Server Infrastructure**

The server infrastructure for .NET, including Windows and the .NET Enterprise Servers, is a suite of infrastructure applications for building, deploying, and operating XML web services. Key technologies include support for XML, scale-out, and business process orchestration across applications and services.

- **.NET My Services**

Until very recently called *Hailstorm*, this is a set of out-of-the box web services that Microsoft has in mind to ship with the final release of .NET. Among the more popular services, Microsoft Passport is currently being used by hundreds of web sites for uniform, single-sign-on secure authentication.

These are some of the languages supported by .NET:

- APL
- C#
- C++
- Caml
- COBOL
- Haskell
- JScript
- Mercury
- Oberon
- OZ
- Pascal
- Perl
- Python
- Scheme
- Smalltalk
- VB

Before we explore J#, let's look briefly at the architectures of .NET (CLR) and Java (JVM).

The following section introduces the .NET Framework architecture in brief and then moves to different cross-language implementations using various .NET-compliant languages.

.NET Framework Architecture (CLR)

Microsoft's .NET Framework provides an amazing aspect of development, called *cross-language development*, which includes cross-language inheritance, cross-language debugging, and cross-language exception handling. More generally, we can visualize cross-language development as follows. Code written using any .NET-compliant language (such as C#) should be usable in any other .NET-compliant language (such as VB). Later on, these code modules should be usable in yet another language (such as JScript, J#, and so on). To make this possible, you need a common runtime environment that can understand all these languages.

One of the main design goals of .NET is to encourage cross-language development. The advantage is that the developer can choose the language that best suits for delivering a given module/unit (each language has its own strengths) and still be able to integrate into a single application. The end result is that the language becomes equal. Even employers feel more comfortable with cross-language development, as they have more resources and options at hand.

The .NET Framework also eliminates "DLL hell" and allows for side-by-side deployment of components because registration information and state data are no longer stored in the Registry, where this information can be difficult to establish and maintain.

Next, let's explore the cross-language capabilities of CLR (the Common Language Runtime Environment) and see how IL (Intermediate Language) becomes the core of all .NET-compliant languages. In particular, I'll discuss how to program for the CLR and show some examples that demonstrate the cross-language capabilities of the .NET Framework.

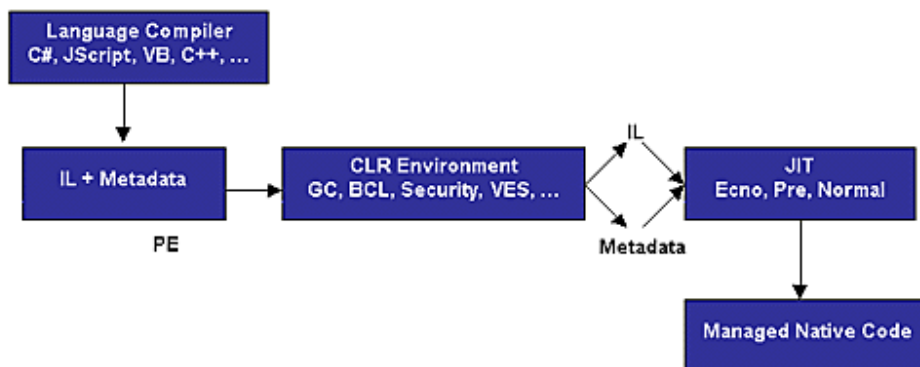
Common Language Runtime

The Common Language Runtime (CLR) Environment provides a rich set of features for cross-language development and deployment. CLR supports both object-oriented languages and procedural languages. CLR manages the execution of code and provides various services such as security, garbage collection, cross-language exception handling, cross-language inheritance, support for the Base Class Library (BCL), and so on. These are the main constituents of the CLR:

- The **Common Type System (CTS)** supports object-oriented programming languages as well as procedural languages. Basically CTS provides a rich type system that's intended to support a wide range of languages.
- The **Common Language Specification (CLS)** is a subset of the Common Type System, to which all language compilers targeting CLR must adhere.
- All compilers under .NET will generate a uniform, common language called **Intermediate Language (IL)**, no matter what language is used to develop the application. In fact, CLR will not be aware of the language used to develop an application. For this reason, IL can be considered the language of CLR—a platform for cross-language development.
- The **Just in Time (JIT) compiler** converts the Intermediate Language code back to a platform/device-specific code. In .NET you have three types of JIT compilers:
 - **Pre-JIT** (compiles entire code into native code at one stretch)
 - **Econo-JIT** (compiles code part by part, freeing when required)
 - **Normal-JIT** (compiles only that part of the code when called, and places it in the cache)
- Type safety is ensured in this phase. In all, the role of a JIT compiler is to bring higher performance by placing the once-compiled code in cache, so that when the next call is made to the same method/procedure, it's executed at a faster speed.

- The **Virtual Execution System (VES)** implements the Common Type System. VES loads links and runs **Portable Executable (PE)** files. VES also ensures loading of the information contained in metadata.
- **Metadata** describes and references the datatypes defined by the VOS type system, lays out instances of classes in memory, resolves method invocation, and solves versioning problems (DLL hell).

[Figure 1](#) depicts the .NET architecture.



[Figure 1](#) .NET architecture.

The Windows operating systems run on the Intel family of microprocessor chips. To generate platform-neutral code, two things should happen:

- Elimination of hardware dependencies such as microprocessor instruction sets, etc.
- Elimination of software dependencies such as operating-system-naïve API, etc.

Once the code is compiled using any of the .NET-compliant language compilers, it gets converted to IL (Intermediate Language), as shown in Figure 1. This code is not compiled to machine-native code, but to an intermediate form that doesn't contain any specific information about hardware or software dependencies. When you run this code, since it's not in machine-specific form, it fails to execute. A runtime environment is required that understands and converts the IL code to machine-specific code. This is the role played by CLR (along with the many other functions discussed earlier). I won't talk about these issues in detail here, but will instead concentrate on implementation of cross-language capabilities of the .NET Framework.

Common Language Features

- All languages use the same library, the **Base Class Library (BCL)**. However, the syntax used by these languages remains the same as that of the original language.
- No language under .NET has its own library.
- Garbage collection is the responsibility of the CLR environment and not a language.

Java Architecture (JVM)

The architecture of JVM doesn't differ dramatically from that of CLR. [Figure 2](#) shows an architectural diagram of JVM; Table 1 demonstrates some of the differences between the two architectures.

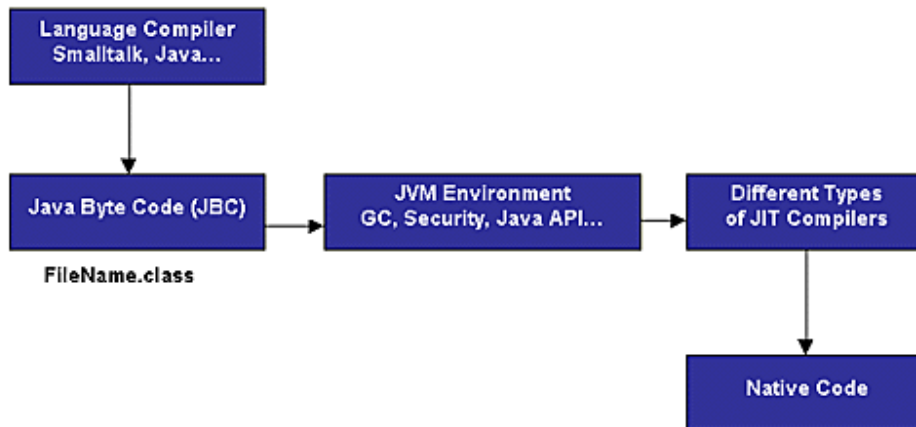


Figure 2 JVM architecture.

Table 1 Java Architecture Versus .NET Architecture

.NET Architecture	Java Architecture
Designed to support multiple different programming languages. Currently, 30 languages support the .NET architecture.	Though other languages' code can be converted to run under JVM, they don't acquire true cross-language capabilities.
Compiles the source code to Intermediate Language (IL), which is itself a language.	Compiles the source code to Java bytecode, which by itself is not a language.
CLR implements a contiguous memory allocation algorithm.	JVM implements a noncontiguous memory allocation algorithm.
Compiles the source code twice during the process of converting to native code. Compiling works faster than interpreting.	Compiles and interprets the source code once during the process of converting it to native code.

J#.NET

Many Java programmers across the globe prefer to use Java in its original form. Having put their efforts into learning Java APIs, they don't want to learn yet another huge API in order to bring cross-language capabilities to the applications they develop. J# is a tool to give more power to the applications they develop under the .NET Framework, using more or less the same language.

J# can be seen as the only language under the .NET Framework that directly supports the syntax and API (to a certain extent) of any existing language. This also can be seen as a step toward making JUMP (Java User Migration Path) a reality.

The following section shows many examples of how this tool makes life easier for the Java developer, irrespective of the framework used to develop applications.

J# Versus Java, Example by Example

Let's examine some programs using different languages and test whether the same CLR would support them. In other words, if the same environment can understand multiple languages at runtime, it supports cross-language development.

I've chosen four categories of examples to demonstrate how elegantly this new tool from Microsoft works:

- Command-line examples
- Cross-language examples
- Windows forms examples
- Web services examples

I'll demonstrate all the examples based on the following approach:

- How the program looks in Java
- How the program looks in J#
- How the program looks in .NET

NOTE

To maintain uniformity across all programs written using different languages, I've used `import System.*;` in Java programs, though it's essentially not required.

Command-Line Examples

Take a look at the following simple Java program, which outputs a simple string to the command prompt. There is nothing special in this program.

Java Example

```
//To Compile - javac welCome.java//To Run - java welComeimport System.*;class welCome{public static void main(String[] args){System.out.println("Welcome To .NET Technology Group - PCS");}}
```

Now examine the same program written using J#. This program looks much like the Java program except for the extension and the compiler; here we use JC instead of JAVAC.

J# Example

```
//To Compile - jc welCome.jsl//To Run - welComeimport System.*;public class welCome{public static void main(System.String[] args){System.out.println("Welcome To .NET Technology Group - PCS");}}
```

Now look at the following version. I've changed the file extension to `.java` but still use the JC compiler instead of JAVAC. The code we compile gets converted to Intermediate Language rather than to Java bytecode. If you're programming for the first time in the .NET world, this is a big surprise. However, if you've worked on other .NET compilers, you might have observed that you needn't bother about the extensions of the filenames; instead, you just look at the code for a given file. That's the reason we can compile with both extensions—`.java` as well as `.jsl`. In fact, you can save your files with any extension.

```
//To Compile - jc welCome.java//To Run - welCome
import System.*;public
class welCome{public static void main(System.String[]
args){System.out.println("Welcome To .NET Technology Group - PCS");}}
```

In the next version, notice that I've changed the output function—from `System.out.println()` to `System.Console.WriteLine()`. As mentioned earlier, under the .NET Framework all languages use a common library, the Base Class Library (BCL). Since this compiler targets CLR, it's expected to acquire the same capability.

.NET Example

```
//To Compile - jc welCome.jsl//To Run - welCome
import System.*;public
class welCome{public static void main(System.String[]
args){System.Console.WriteLine("Welcome To .NET Technology Group -
PCS");}}
```

The following table compares the programs so far.

	Java Program	J#/.NET Program
File extension	<code>.java</code>	Any extension; <code>.jsl</code> by default
Compiler	JAVAC	JC
Library functions	Java API	Java API and .NET Framework library
Intermediate form	Java bytecode	Intermediate Language (IL)
Runtime environment	JVM (Java Virtual Machine)	CLR (Common Language Runtime)
Platform	Can run on multiple platforms	Designed to run on multiple platforms
Interoperability with other languages	Lacks cross language capabilities to a great extent	Sound cross-language capabilities
Output	Filename <code>.class</code>	Filename <code>.exe</code> or Filename <code>.dll</code>

	Java Program	J#/.NET Program
Documentation	HTML	XML

Cross-Language Examples

This particular feature is unique to CLR. By *cross-language capability*, I mean that a method written in one language can be overridden in another language and so on. Exception handling across different languages could be achieved.

Though Java has limited capability to interoperate with certain languages like Smalltalk, it has never attained full-blown cross-language capabilities. The reason could be simply that Java was never designed to attain such a capability.

Java to C#

The following examples implement a `getMessage()` method in the J# program, inherit it in the C# program, and override it with another implementation. The following program generates a DLL, namely `welcome.dll`.

```
//jc /t:library welcome.jslimport System.*;public class welcome{ public
void getMessage() { System.Console.WriteLine("Welcome To .NET
Technology Group - PCS"); }}
```

The following program uses the `welcome.dll` created in the program above, inherits the `getMessage()` method, and overrides this program with new implementation. It works amazingly well to enable Java programmers to inherit the code written in other languages and use it in their programs.

```
//csc /r:welcome.dll, BCLIB.dll demoWelcome.txtusing System;class
demoWelcome:welcome{ public static void Main() { demoWelcome t =
new demoWelcome(); welcome w = new welcome(); t.getMessage();
w.getMessage(); } public override void getMessage() {
Console.WriteLine("Cross-Language Capabilities of .NET"); }}
```

Java to Java

The following code is written in Java with the `getMessage()` method. This feature wouldn't work in Java.

```
//To Compile - Javac welcomeimport System.*;public class welcome{
public void getMessage() { System.out.println("Welcome To .NET
Technology Group"); }}
```

In the following example, we will try to inherit the `welcome` class and override the `getMessage()` of the `welcome` class with a new implementation. Of course, in this case, we will be able to accomplish this only within a single language.

```
// To Compile - Javac demoWelcome // To Run - Java demoWelcomeimport
System.*;class demoWelcome extends welcome{ public static void
```

```
main(String[] args) { demoWelCome t = new demoWelCome(); welCome w
= new welCome(); t.getMessage(); w.getMessage(); } public void
getMessage() { System.out.println("Cross-Language Capabilities of
.NET - PCS"); }}
```

C# to Java

The earlier examples showed you how to take a method written in Java and override it in C# to create a new implementation. But there might be situations where we would like to reverse this process. In the following program, the `getMessage()` method is implemented using C# and then overridden in J#/Java.

```
//To Compile - csc /t:library welCome using System; public class welCome{
public void getMessage() { System.out.println("Welcome To .NET
Technology Group"); }}
```

The following example overrides the `getMessage()` and implements it as I wish. This probably surprises many Java programmers, thanks to such a nice cross-language environment.

```
//jc /r:welCome.dll demoWelCome.jsl import System.*; class demoWelCome
extends welCome{ public static void main(System.String[] args) {
demoWelCome t = new demoWelCome(); welCome w = new welCome();
t.getMessage(); w.getMessage(); } public void getMessage() {
System.out.println("Cross-Language Capabilities of .NET"); }}
```

The .NET Way

The following program shows you how this is implemented in a strict .NET Framework.

```
//To Compile - csc /t:library welCome using System; public class welCome{
public void getMessage() { System.Console.WriteLine("Welcome To
.NET Technology Group"); }}
```

In the following example, we will try to inherit the `welCome` class within J# itself. Later, we will also try to override the `getMessage()` method.

```
//jc /r:welCome.dll demoWelCome.jsl import System.*; class demoWelCome
extends welCome{ public static void main(System.String[] args) {
demoWelCome t = new demoWelCome(); welCome w = new welCome();
t.getMessage(); w.getMessage(); } public override void getMessage()
{ Console.WriteLine("Cross-Language Capabilities of .NET"); }}
```

The following table compares the programs.

	Java Program	J#/.NET Program
Override Keyword	Not necessary	Optional
Library Function	<code>System.out.println</code>	<code>System.Console.WriteLine</code>

	Java Program	J#/.NET Program
Overriding	Java compiler gives no warning	J# compiler gives a warning

Windows Forms

Now let's move ahead to build some GUI applications. Here we'll examine the examples written using pure J#, Java AWT, and Java Swing.

The following example is written using AWT of Java-like code but targets for CLR rather than JVM. The program looks exactly like a pure Java AWT program. But if you observe the code closely you find that I've saved the file with a .jsl extension. It doesn't matter even if I save it with a .class extension. I compiled without any errors and can run it successfully.

Windows Forms

```
//To Compile - jc AWTTest.jsl// To Run - AWTTestimport
java.awt.*;import java.awt.event.*;public class AWTTest extends Frame
implements WindowListener{ public AWTTest() { this.setTitle("Welcome
To .NET Technology Group"); this.setSize(400,300);
this.addWindowListener(this); this.setBackground(Color.white);
this.setVisible(true); } public void windowClosing(WindowEvent we){
System.exit(0); } public void windowOpened(WindowEvent we){ } public
void windowClosed(WindowEvent we){} public void
windowIconified(WindowEvent we){} public void
windowDeiconified(WindowEvent we){} public void
windowActivated(WindowEvent we){} public void
windowDeactivated(WindowEvent we){} public static void main(String[]
args){ AWTTest obj = new AWTTest(); }}
```

Java AWT

Here's a simple Java AWT program that displays a window.

```
//To Compile - javac AWTTest.java//To Run - java AWTTestimport
java.awt.*;import java.awt.event.*;public class AWTTest extends Frame
implements WindowListener{ public AWTTest() { this.setTitle("Welcome
To .NET Technology Group"); this.setSize(400,300);
this.addWindowListener(this); this.setBackground(Color.white);
this.setVisible(true); } public void windowClosing(WindowEvent we){
System.exit(0); } public void windowOpened(WindowEvent we){ } public
void windowClosed(WindowEvent we){} public void
windowIconified(WindowEvent we){} public void
windowDeiconified(WindowEvent we){} public void
windowActivated(WindowEvent we){} public void
windowDeactivated(WindowEvent we){} public static void main(String[]
args){ AWTTest obj = new AWTTest(); }}
```

Java Swing

Here I've used Swing rather than AWT, just to show how you can implement the same program using Swing.

```
//To Compile - javac SwingTest//To Run - java SwingTestimport
javax.swing.*;import java.awt.event.*;import java.awt.*;public class
SwingTest extends JFrame implements WindowListener{ Container cnt =
null; public SwingTest() { cnt = this.getContentPane();
this.setTitle("Welcome To .NET Technology Group");
this.setSize(400,300); this.addWindowListener(this);
cnt.setBackground(Color.white); this.setVisible(true); } public
void windowClosing(WindowEvent we){ System.exit(0); } public void
windowOpened(WindowEvent we){} public void windowClosed(WindowEvent
we){} public void windowIconified(WindowEvent we){} public void
windowDeiconified(WindowEvent we){} public void
windowActivated(WindowEvent we){} public void
windowDeactivated(WindowEvent we){} public static void main(String[]
args){ SwingTest obj = new SwingTest(); }}
```

The .NET Way - Pure J#

The following program uses the .NET Framework Base Class Library and Java-like syntax. This feature is common to all .NET languages, no matter what language you use for writing applications.

```
//jc /r:System.Windows.Forms.dll,System.Drawing.dll welCome.jslimport
System.Drawing.*;import System.ComponentModel.*;import
System.Windows.Forms.*;public class welCome extends
System.Windows.Forms.Form{ private System.ComponentModel.Container
components = null; public welCome() { InitializeComponent();
} protected void Dispose(boolean disposing) { if (disposing) {
if (components != null) { components.Dispose(); } }
super.Dispose(disposing); } private void InitializeComponent() {
this.components = new System.ComponentModel.Container();
this.set_Size(new System.Drawing.Size(300,300));
this.set_Text("Welcome To .NET Technology Group"); } public static
void main(String[] args) { Application.Run(new welCome()); }}
```

The following table compares the programs.

	Java Program	J#/.NET Program
AWT Support	Full support	Limited support
Swing	Full support	No support

Web Services

Web services are relatively new to most Java programmers. Designing webservices using .NET is fairly easy and simple. Though it requires manual generation of web services proxy class, in beta 1 of J# you can write both service and client very easily.

The following code shows how simple it is to write a web service using J#, using the `welcome.aspx` file of ASP.NET applications.

```
import System.Collections.*;import System.ComponentModel.*;import
System.Data.*;import System.Diagnostics.*;import System.Web.*;import
System.Web.Services.*;public class welcome extends
System.Web.Services.WebService{ public welcome() {
InitializeComponent(); } private void InitializeComponent() { }
public void Dispose() { } public System.String getMessage(String
helloMsg) { return helloMsg; }}
```

Advantages and Disadvantages

The following table summarizes the differences between Java under the Java framework and the .NET Framework.

Feature	Java Framework	.NET Framework
Code reusability	No true cross-language code reusability	Advocates true code reusability
Support for web services	Late starter to this area, but it does support this feature	Strong built-in support for web services
Platform independence	Great support	Generates platform-neutral code
Standards	Great support for XML and SOAP	Great built-in support for XML and SOAP

J# offers the following features:

- Generates platform-neutral code in the form of MSIL
- Support for class libraries of VJ++, JDK 1.1.4
- Interoperates with COM
- `jbimp.exe` converts the JBC (Java bytecode) to MSIL
- Brings cross-language capabilities into Java language
- Simultaneously supports both JDK 1.1.4 and BCL (Base Class Library) of .NET Framework
- Accesses platform-native resources
- Generates XML documentation
- Writes ASP.NET applications
- Writes web services
- Retains the majority of Java-like syntax and features

These are J#'s disadvantages:

- Programs can't run under the umbrella of JVM
- No support for the Java Native Interface or Remote MethodInvocation
- No operator overloading
- Minimal support to convert binaries making J/Direct calls
- Automatic generation of proxy classes for web services
- Cannot call `WebService` methods using `Enumtypes`
- Cannot add reference to `.exe` files

Conclusion

Overall, the J#.NET tool looks promising and provides the smallest learningcurve for Java developers among the available .NET compilers. Java developerswill gain an additional advantage, as they still remain the preferredprogramming community either in the .NET Framework environment or the Javaenvironment. This also could pose an indirect threat to Microsoft's newlyreleased language, C#. Though many more features need to be perfected in J#, myinitial evaluation lands in favor of this tool for the Java developer. Happyprogramming!

Acknowledgments

I would like to thank every member of the .NET Technology Group at Patni Computer Systems Limited for their invaluable support and encouragement. I also would like to thank T.Venkatachalam and Karthik Iyer for their patience in reviewing this document. Finally, I extend my special thanks to Milind Pande, manager of the .NET Technology Group, for his continued support for creative and innovative thinking.