Chapter 2
The Runtime Environment

The Java and .NET platforms are similar in that they both require a runtime environment. This environment typically sits on top of the operating system and performs critical tasks such as class loading, code interpretation, dynamic compilation and execution, code verification to ensure safety, and access control to ensure security. A properly written and compiled Java or .NET application is capable of running on any platform to which its runtime environment has been ported. The portability of Java and .NET is, in large part, measured by the availability of its runtime environment on diverse hardware and software platforms.

In this chapter, we will examine the similarities and differences between the Java and .NET runtime environments. As you will see, there is an equivalent .NET component for nearly every component of the Java platform and vice versa. In the upcoming pages, we will compare the following formats and technologies.

- Bytecode vs. Intermediate Language
- Java Virtual Machine vs. Common Language Runtime
- Packages vs. Namespaces
- Java API vs. .NET Framework Class Library
- Classpath vs. Reference
- JAR Files vs. Assemblies

Following these comparisons, we will discuss the current level of portability realized by each platform. That is, we will evaluate the hardware and software platforms on which each runtime environment is currently supported or will support in the near future. Last, we will examine the versions of the Java and .NET platforms that are available and where to get them.

Bytecode vs. Intermediate Language

In many respects, Java bytecode and .NET intermediate language (IL) are very similar. Both formats employ an intermediate form that lends itself to either interpretation (supported by Java) or dynamic compilation (supported by Java and .NET). Note that, unlike Java bytecode, IL was not designed for interpretation and interpretative execution is not typically supported by the .NET platform. Later in this section we will examine the design differences that contribute to making Java bytecode easier to interpret than .NET IL.

The bytecode and IL specifications are analogous to a processor's native instruction set. These specifications define the entire set of commands supported by their respective platforms. Each language defines approximately 256 instructions (including some "no-op" instructions saved for future use). Limiting the number of instructions, or "opcodes", to 256 allows each instruction to be represented by a single byte (hence the term "bytecode").

In order to visualize the differences between Java bytecode and .NET IL, let's take a look at the actual instructions that are generated when a simple program is compiled with the Java and

C# compilers. To begin, examine the following statements. Note that these statements are valid in both Java and C#.

```
int x, y, z;
x = 1;
x++;
y = 2;
y--;
z = 10;
z = (x + y) * z;
```

Now assume that these statements are added to a method within a Java and C# class and compiled. After compilation, it is possible to examine the bytecode and IL produced by the compiler through the use of special utilities included with the Java and .NET platforms. The tools used to disassemble Java and C# compiled code are called `javap` and `ildasm`, respectively, and are invoked from the command line as follows:

```
C:\> javap –c MyCompiledClass
C:\> ildasm MyCompiledClass.exe
```

As promised, let's now take a look at the code generated by the Java and C# compilers after the previous statements are compiled. First, let's examine the Java version. The Java compiler (`javac`) generates the following bytecode instructions for the given statements. Note that the compiler generates only the information shown under the **INSTRUCTION** column. The **BYTE**, **COMMENT**, and **JAVA STATEMENT** columns have been added by the author for documentation purposes.

```
BYTE INSTRUCTION COMMENT                                JAVA STATEMENT
---- ----------- -------------------------------------- ----------------
 1   iconst_1    push integer value 1 onto stack        x = 1;
 2   istore_1    pop integer and store in variable 1    x = 1;
 3   iinc 1 1    increment integer variable 1 by 1      x++;
 6   iconst_2    push integer value 2 onto stack        y = 2;
 7   istore_2    pop integer and store in variable 2    y = 2;
 8   iinc 2 -1   decrement integer variable 2 by 1      y--;
11   bipush 10   push integer value 10 onto stack       z = 10;
13   istore_3    pop integer and store in variable 3    z = 10;
14   iload_1     push integer variable 1 onto stack     z = (x + y) * z;
15   iload_2     push integer variable 2 onto stack     z = (x + y) * z;
16   iadd        add top two integers on stack          z = (x + y) * z;
17   iload_3     push integer variable 3 onto stack     z = (x + y) * z;
18   imul        multiply top two integers on stack     z = (x + y) * z;
19   istore_3    pop result and store in variable 3     z = (x + y) * z;
```

Similarly, when added to a method within a C# class and compiled, the C# compiler (`csc`) generates the following IL code for these statements.

```
BYTE INSTRUCTION COMMENT                                C# STATEMENT
---- ----------- -------------------------------------- ----------------
 1   ldc.i4.1    push integer value of 1 onto stack     x = 1;
 2   stloc.0     pop value and store in variable 0      x = 1;
 3   ldloc.0     push variable 0 onto stack             x++;
 4   ldc.i4.1    push integer value of 1 onto stack     x++;
 5   add         add top two integers on stack          x++;
 6   stloc.0     pop result and store in variable 0     x++;
 7   ldc.i4.2    push integer value of 2 onto stack     y = 2;
 8   stloc.1     pop value and store in variable 1      y = 2;
 9   ldloc.1     push variable 1 onto stack             y--;
10   ldc.i4.1    push integer value 1 onto stack        y--;
11   sub         subtract top two integers on stack     y--;
12   stloc.1     pop result and store in variable 1     y--;
13   ldc.i4.s 10 push integer value 10 onto stack       z = 10;
15   stloc.2     pop value and store in variable 2      z = 10;
16   ldloc.0     push variable 0 onto stack             z = (x + y) * z;
17   ldloc.1     push variable 1 onto stack             z = (x + y) * z;
18   add         add top two values on stack            z = (x + y) * z;
19   ldloc.2     push variable 2 onto stack             z = (x + y) * z;
20   mul         multiply top two values on stack       z = (x + y) * z;
21   stloc.2     pop result and store in variable 2     z = (x + y) * z;
```

Let's begin our investigation of Java bytecode and .NET IL by examining their similarities. First, it is apparent that both intermediate languages are fashioned after a typical assembly language format. That is, the compiled code is comprised of simple statements with each statement consisting of a basic instruction followed by zero or more arguments. For example, the following bytecode and IL statements push a decimal 10 onto the stack. Each statement consists of a basic instruction (`bipush` and `ldc.i4.s`) and a numeric argument (`10`).

```
bipush 10    //Java bytecode
ldc.i4.s 10  //.NET IL
```

According to the **BYTE** column shown in the bytecode and IL code listings, both of these commands consume two bytes. This is due to the fact that the instruction requires one byte and the integer argument requires a second.

Another similarity between these languages lies in their use of “shortcut” commands. A “shortcut” command is a specialized instruction designed to perform a common operation in fewer bytes than the operation would typically require. For instance, pushing an integer onto the stack normally requires a two-byte instruction (one byte for the command and one byte for the integer argument). However, the following bytecode and IL “shortcut” commands perform this operation using a specialized single-byte instruction (pushing a 0 or 1 onto the stack is considered a common operation and is supported by “shortcut” commands on both platforms).

```
iconst_1  //Java bytecode
ldc.i4.1  //.NET IL
```

“Shortcut” commands contribute to denser code and faster execution. Absent these commands, the previous operation (push a 1 onto the stack) would have required two bytes as follows:

```
bipush 1    //Java bytecode
ldc.i4.s 1  //.NET IL
```

The differences between these two languages are nearly as pronounced as the similarities. First, there are some syntactic differences between the languages. Specifically, IL uses a period to separate words while bytecode prefers an underscore. Less quantitatively, IL syntax appears slightly more complex than bytecode.

Second, the bytecode example is a bit more compact due to support for additional “shortcut” commands. For example, notice the number of commands required for each language to perform the `x++` increment and `y--` decrement operations. Java performs each of these operations using a single “shortcut” command that consumes 3 bytes. In contrast, .NET requires 4 bytes for the same operation. This accounts for the 2 byte difference in the overall size of each program (19 bytes for Java versus 21 bytes for C#). Though the bytecode was more compact in this example, this is not always the case. To the contrary, some applications may make more use of specialized .NET "shortcut" commands that do not exist in Java, thus, making some .NET applications more compact than their Java counter parts.

Finally, the most prominent difference between bytecode and IL is the manner in which each language conveys type information. You may have noticed from the previous code listings that the Java bytecode instructions were prefixed with an "`i`" indicating that these instructions operate on integers (e.g., the "`iadd`" instruction adds two integers). In contrast, .NET IL instructions do not explicitly specify the data types upon which they operate. Rather, IL utilizes generic instructions and requires the CLR to determine the appropriate type of operation based on the data types of the stack arguments (e.g., the generic "`add`" instruction adds the top two values on the stack, regardless of type). Though it may seem trivial, it is this subtle difference that impairs .NET's ability to efficiently interpret IL code. Requiring the interpreter to evaluate stack variables before performing an operation is very expensive and would most likely incur a significant performance penalty. Of course, the .NET designers were well aware of this limitation and established early on that the .NET platform would not support interpretative execution.

Java Virtual Machine vs. Common Language Runtime

The Java Virtual Machine (JVM) and the Common Language Runtime (CLR) serve similar purposes. These components each provide a layer of abstraction that isolates applications from the underlying hardware and operating system. Rather than compiling applications to a specific hardware or OS platform, applications are compiled to an intermediate language that is native to the abstraction layer. In this manner, applications are capable of executing on any platform to which the abstraction layer has been ported. This abstraction layer, often called an "abstract machine", is typically implemented in one of two ways. One implementation incorporates an interpreter that strictly emulates the abstract machine. This emulation process allows diverse machines to appear identical to an application. The second implementation employs a platform-specific compiler that compiles the abstract machine code (i.e., intermediate language) to the machine language natively supported by the host system. While .NET solely supports the latter implementation (i.e., native compilation), the Java platform employs a combination of both techniques (i.e., interpretation/emulation and native compilation).

In addition to their differing execution preferences (i.e., interpretation vs. native compilation), the JVM and CLR diverge in other significant ways. Two of their most salient differences regard support for multiple languages and application domains. First, unlike the JVM which was designed to execute programs written in the Java programming language, the CLR was explicitly designed to support multiple languages through a common object model known as the Virtual Object System (VOS). Implemented by each .NET language, the VOS allows for full object-oriented interaction between classes written in different languages. For example, it is possible for a C# class to extend a class developed with Visual Basic.NET and invoke or override all inherited functionality. Second, the CLR supports a method of application partitioning known as application domains. Application domains provide a level of isolation between programs running within the same process. Since they are much lighter weight than processes, application domains increase scalability by allowing multiple applications to share a single process without conflict. The JVM does not support this level of application isolation.

Despite their many differences, the JVM and CLR also share many common characteristics. For example, each runtime environment supports polymorphism through interfaces and a singleinheritance, single-rooted object model. Additionally, each platform provides automatic garbage collection, code verification, and security. A careful review of their architectures reveals that, apart from a few subtle differences, the JVM and CLR are very similar in functionality and comparable in performance.

Packages vs. Namespaces

Java packages and .NET namespaces are used to organize classes within a class library. These structures group classes in a hierarchical fashion in order to avoid naming collisions and allow classes to be easily browsed and quickly located when needed. The primary difference between these two structures rests in the fact that Java packages are grouped both logically and physically while .NET namespaces represent a logical grouping only. The following paragraphs highlight the differences between packages and namespaces.

Classes within the Java API are logically grouped into packages and physically grouped into file system directories. All compiled classes belonging to the same package must reside within the same directory and the directory structure in which these classes are stored must be isomorphic to the package name. In other words, the directory structure must directly map to the package name (more about this later). Packages are declared in the first line of code within a Java source file. All classes and interfaces defined within a single Java source file must belong to the same package and a single source file can contain only one public class. There is no limit to the length of package names or how deeply they may be nested. A typical package declaration is as follows:

```
package java.util;

public class Date
{
  //Class java.util.Date implementation here...
}
```

Packages provide a logical grouping by allowing classes to be uniquely referenced according to their package. For instance, assume that a class named `Date` exists in both the `java.util` and the `java.sql` packages. Naming conflicts can be avoided by explicitly referencing the appropriate `Date` class using the fully qualified package name `java.util.Date` or `java.sql.Date`. In Chapter 3, we will learn how to abbreviate these references through the use of Java's `import` statement.

In addition to their logical grouping, Java packages are physically grouped within a well-defined directory structure. As mentioned previously, the directory structure in which a class is stored must strictly follow the class's package name. For instance, the `java.util.Date` and `java.sql.Date` compiled classes (i.e., `.java` and `.class` files) must reside within the *java/util/* and *java/sql/* directories, respectively.

Note that all standard java classes reside under the `java` or `javax` packages (e.g., `java.lang.String`, `javax.swing.JButton`). User-defined classes cannot reside within either of these packages. The standard Java package naming convention indicates that organizations should base package names on their unique Internet domain name. This ensures that code created by one organization will not experience naming conflicts when combined with code from another party. The Java package naming convention states that the top-level domain should be listed first followed by the organization's domain name (e.g., the domain `sun.com` would be reversed to read `com.sun` at the beginning of a package name). After the reverse domain name, the package may be arbitrarily organized in whatever manner the author feels is most intuitive. Here are a few examples:

```
package com.sourcestream.util;   //example for sourcestream.com domain
package org.jboss.util.zip;      //example for jboss.org domain
package edu.byu.enroll.student;  //example for byu.edu domain
```

.NET namespaces are similar to Java packages with a few exceptions. First, namespaces represent a logical grouping only. Unlike packages, namespaces do not require classes to reside in any particular directory. But like packages, namespaces prevent naming conflicts by allowing classes having the same name to be referenced according to their fully qualified namespace. Namespaces are declared within a source file using the `namespace` keyword. A typical namespace declaration in C# looks like this:

```
namespace SourceStream.File
{
  public class Directory
  {
    //Class SourceStream.File.Directory implementation here...
  }
}
```

Notice that the previous namespace is prefaced with the name of the author's company: SourceStream. The naming convention for namespaces is slightly different than that for Java packages. Rather than prefacing all user-defined namespaces with an organization's reverse domain name, .NET namespaces simply begin with a company name. Though not guaranteed to be unique, prefacing namespaces with a company name makes naming collisions highly unlikely.

In addition to not being tied to a directory structure, namespaces are more flexible than Java packages in other ways. For instance, unlike packages that must be declared on a single line, namespaces can be declared on a single line or nested as follows:

```
namespace SourceStream
{
  namespace File
  {
    public class Directory
    {
      //Class SourceStream.File.Directory implementation here...
    }
  }
}
```

Furthermore, unlike Java packages, a single source file can contain multiple public classes within different namespaces as shown here:

```
namespace SourceStream
{
  public class Generic
  {
    //Class SourceStream.Generic implementation here...
  }

  namespace File
  {
    public class Directory
    {
      //Class SourceStream.File.Directory implementation here...
    }
  }

  namespace Phone
  {
    public class Directory
    {
      //Class SourceStream.Phone.Directory implementation here...
    }
  }
}
```

Similar to the `java` and `javax` packages, all standard .NET namespaces begin with `System`. Therefore, user-defined classes should never begin with this namespace. If you remember to always begin your custom namespaces with your company name, you shouldn't encounter any conflicts.

Like packages, namespaces prevent naming conflicts. For example, the previous example contains two classes named `Directory`. This situation has the potential to generate naming collisions. However, since these classes are in separate namespaces, they can each be uniquely identified by fully qualifying references to them using `SourceStream.File.Directory` and `SourceStream.Phone.Directory`. In Chapter 3, we will examine how to shorten these references by utilizing the C# language's `using` keyword.

> **NOTE**
> Neither Java nor .NET requires that all programs be explicitly assigned to a package or namespace. If a source file does not specify a package or namespace, the compiled program will be automatically assigned to a nameless package or namespace. In this case, the resulting class file could be accessed directly by name rather than having to specify a particular package or namespace.

Java API vs. .NET Framework Class Library

The Java API and .NET Framework Class Library are very similar in many respects. They are both comprehensive object-oriented class libraries that provide easy access to critical system resources and services (e.g., file systems, networks, threads, graphical user interfaces, etc.). The most significant difference between these two libraries is the manner in which they are organized. Though both libraries are very well designed, the names and locations of classes vary. To illustrate, Table 2.1 presents several common Java packages and their comparable .NET namespaces. Detailed differences between the Java API and the .NET Framework Class Library along with concise code examples will be presented in the following chapters.

Table 2.1 ***Common Java Packages and Comparable .NET Namespaces***

	Java Package(s)	**.NET Namespace**
Database Access	`java.sql` `javax.sql`	`System.Data`
File IO	`java.io` `java.nio`	`System.IO`
Network Access	`java.net`	`System.Net`
Collections	`java.util`	`System.Collections`
UI Development	`java.awt` `javax.swing`	`System.Windows.Forms`
XML Processing	`javax.xml`	`System.Xml`

Classpath vs. Reference

Both Java and .NET require a mechanism to tell the compiler and runtime environment where required classes can be found. In Java, this mechanism is called a *classpath*. In .NET, it's known as a *reference*. A strong understanding of these mechanisms is essential since problems encountered when compiling or executing Java and .NET applications can often be traced to an invalid classpath or reference setting.

A Java classpath is simply a list of files and/or directories that contain classes required to compile and/or execute an application. A class is "required" by an application if the application references the class or any of its dependent classes. In other words, all of an application's dependencies must be locatable within the classpath.

A classpath can be set in two ways. First, an environment variable named `CLASSPATH` can be set containing a delimited list of files and/or directories. This list is delimited by semi-colons on Windows and colons on UNIX and Linux. For example, the following command-line instructions set the `CLASSPATH` environment variable on Windows and UNIX, respectively.

```
C:\> set CLASSPATH=c:\jdk1.4\jre\lib\rt.jar;c:\classes\
shell> export CLASSPATH=/usr/jdk1.4/jre/lib/rt.jar:/usr/classes/
```

In addition to using an environment variable, the classpath can be set at compile or execution time using the `–classpath` command-line switch. The following instructions use the command-line switch to tell the java compiler and interpreter, respectively, where to find the classes that are required by the given application. See Chapter 6 for detailed instructions regarding the proper use of the java compiler and interpreter.

```
C:\> javac -classpath c:\jdk1.4\jre\lib\rt.jar;c:\classes\ MyApp.java
C:\> java -classpath c:\jdk1.4\jre\lib\rt.jar;c:\classes\ MyApp
```

The `–classpath` command-line switch takes precedence over the `CLASSPATH` environment variable. Therefore, if the command-line switch is specified, its value overrides any `CLASSPATH` environment variable. On the other hand, if not specified, the Java compiler and runtime environment will use the value stored in the `CLASSPATH` environment variable by default.

Similar to Java's classpath, the reference setting in .NET serves the same purpose but works a bit differently. For instance, .NET references are never set through an environment variable. Rather, reference settings are issued on the command-line at compile time using the `/reference` switch (or `/r` for short). The value for a reference must consist of a semi-colon delimited list of files. Unlike the Java classpath, directory names are not supported by the `/reference` switch. For example, the following command compiles an executable program called `MyApp.cs` that requires class libraries named `Library.dll` and `Util.dll`. See Chapter 6 for more detailed information regarding the C# compiler.

```
C:\> csc /reference:Library.dll;Util.dll MyApp.cs
```

In response to the previous command, the C# compiler will generate a file called `MyApp.exe`. In contrast to Java, you need not execute an interpreter when running a compiled .NET application. Instead, you can simply type the executable file's name on the command line or double-click the file within the graphical file explorer. Upon recognizing the file as a portable executable (i.e., compiled to intermediate language), the operating system will automatically invoke the .NET runtime in order to execute the program. Additionally, the .NET runtime will search the application's directory and a global cache for all required class libraries. In this way, reference information is not required at runtime.

JAR Files vs. Assemblies

Java JAR files are similar to .NET assemblies in some ways but very different in others. At first glance, these two constructs appear to be somewhat equivalent. However, upon closer inspection, we can see that they are actually quite different.

When an application is compiled, the Java compiler generates one or more class files (i.e., files ending with the `.class` extension). The class file is Java's basic unit of execution. All Java applications consist of one or more class files combined with any dependencies such as resource files, configuration files, or images. As long as all required class files are locatable on the classpath (and reside within the proper directory structure), a Java application can be executed by the interpreter.

Though class files are simple to use, they are often not very convenient. For example, distributing large Java applications consisting of hundreds or thousands of class files can be unwieldy. For this reason, the engineers at Sun developed the concept of a Java archive or JAR file. A JAR file is a collection of classes and other resource files combined into a single file using the industry standard ZIP format. JAR files are useful for packaging entire applications into a single file or for partitioning common functionality into separate files. For example, an entire application can be combined into a single `MyApp.jar` file or reusable portions of the application may be partitioned into separate JAR files such as `FileUtil.jar`, `DBUtil.jar`, and `Network.jar`. In this manner, reusable libraries can be easily distributed with multiple applications by simply including the appropriate JAR file in the distribution.

You might be wondering how a JAR is different from a normal ZIP file. Good question. A JAR is basically just a standard ZIP file with one exception. In addition to classes and resource files, every JAR contains a special file called the *manifest*. The manifest contains metadata that describes the contents of the JAR file such as information regarding the version and author of the file. JAR files are created using Java's `jar` utility as shown here. See Chapter 4 for more information on the `jar` utility.

```
C:\> jar –cvf MyApp.jar *.class
```

Java JAR files can serve as more than just portable class libraries. They can also be made "executable" by specifying the main class to run when the JAR file is executed. The main class must be declared within the manifest. Executable JAR files can be run from the command line as follows:

```
C:\> java –jar MyApp.jar
```

Now let's take a look at .NET assemblies. In some ways, assemblies resemble Java class files more closely than they do JAR files. Like Java class files, assemblies are .NET's basic unit of execution. All .NET compilers generate assemblies from source code. However, unlike class files, an assembly may assume several different forms. For example, depending on the compilation parameters, an assembly may be an executable (`.exe` extension), a class library (`.dll` extension), or a module (`.netmodule` extension). An executable assembly can be run from the command line, a class library can be referenced from other assemblies, and a module can be compiled into other assemblies.

So, in what ways are assemblies like JAR files? Similar to JAR files, assemblies may contain numerous classes. In .NET, a source file may define any number of classes all of which are compiled into a single assembly file. Similarly, multiple source files can be compiled into a single assembly. This is in stark contrast to Java classes where each Java class maps to a single compiled class file. Also similar to JAR files, .NET assemblies include a manifest file containing metadata about the contents of the assembly. The following statements demonstrate how to compile two source files into one class library and then reference the library when compiling an executable. See Chapter 6 for more information regarding the C# compiler.

```
C:\> csc /target:library /out:MyLibrary.dll FileUtil.cs NetworkUtil.cs
C:\> csc /target:exe /reference:MyLibrary.dll MyApp.cs
```

In summary, .NET assemblies are similar to Java class files in that they both represent their respective platforms' basic unit of execution. On the other hand, assemblies are similar to JAR files in that they contain a manifest file describing their contents and may include any number of classes. Like JAR files, assemblies can simplify distribution by providing the means whereby an entire application can be compiled into a single file.

Cross-Platform Support

Both Java and .NET enjoy some level of cross-platform support. However, Java is currently far more ubiquitous across heterogeneous hardware and software platforms. This probably stems from the fact that Java has been around a lot longer and, from inception, Java was designed to be portable across diverse hardware architectures and operating systems. Java is currently supported on many hardware platforms such as Intel, SPARC, Motorola, StrongARM, MIPS, PowerPC, and Alpha as well as numerous software platforms including:

- Windows
- Solaris
- MacOS
- AIX
- OS/2, OS/390, OS/400
- HP-UX
- SCO UnixWare
- Linux

- Netware
- FreeBSD
- Tru64
- OpenVMS
- NextStep
- PocketPC
- PalmOS
- EPOC

Arriving somewhat late to the multi-platform party, .NET is attempting to gain acceptance as a cross-platform alternative to Java. Though .NET is primarily available and supported on Windows, Microsoft has ported the Common Language Infrastructure (CLI) to the Intel x86 version of the FreeBSD operating system under the auspices of a shared source project called *Rotor* (see *http://msdn.microsoft.com/net/sscli/*). The CLI is comprised of the CLR and the .NET base class library. Though this port is a step in the right direction, it should be noted that full .NET functionality will not be supported on the FreeBSD platform. Since the ADO.NET, ASP.NET, and Windows Forms APIs are not part of the standardized CLI, they will not be supported on FreeBSD. In fact, there is some question whether Microsoft will allow these technologies to be ported to other platforms at all.

One notable exception, however, is the Mono project currently being conducted by Ximian, Inc. and numerous open source developers (see *http://go-mono.com*). After receiving tacit approval from Microsoft, the Mono project is attempting to port the entire .NET architecture (including ADO.NET, ASP.NET, and Windows Forms) to Linux. Since ADO.NET, ASP.NET, and Windows Forms are not standardized portions of .NET, the legal ramifications of this effort are unclear. Regardless, progress is being made toward fully enabling .NET on the Linux platform.

Though the cross-platform prospects for .NET may seem dim (given the lack of critical APIs like ADO.NET and ASP.NET), .NET does have one strong advantage over Java in this area. In contrast to Java which remains under Sun's control, the .NET CLI and C# language specifications have been submitted to the ECMA standards body (see *http://www.ecma.ch*) and have been approved as international standards. You may review Microsoft's submissions to the ECMA at *http://msdn.microsoft.com/net/ecma/*. Given the standardization of the CLI and C# language, it is legal for any entity to implement these specifications without payment to Microsoft. In contrast, Sun charges license fees to companies that wish to implement a Java Virtual Machine using Sun's reference implementation or display a Java compatible logo on their product. Of course, Microsoft has to make money as well. Therefore, Microsoft has implemented non-standard commercial APIs on top of the .NET CLI for which they can charge. As mentioned previously, these APIs include many of .NET's most popular technologies such as ADO.NET, ASP.NET, and Windows Forms.

Where to Get Java and .NET

The Java and .NET platforms are readily available on the Internet free of charge. Each platform is offered in two versions: basic runtime configuration and software development kit (SDK). The runtime versions, known as the Java Runtime Environment (JRE) and the .NET Framework, contain only the runtime environment required for executing Java or .NET applications. On the other hand, the software development kit versions, known as the Java SDK and .NET Framework SDK, contain the runtime environment plus development tools such as compilers and debuggers.

In order to compile and run the code examples in this book, you will need the SDK versions of both platforms. The Java SDK can be downloaded from the following URL:

```
http://java.sun.com/
```

The .NET Framework SDK is available at:

```
http://microsoft.com/net/
```

Summary

This chapter demonstrated the differences and similarities between several comparable Java and .NET technologies. We learned that Java and .NET compilers generate bytecode and intermediate language code, respectively. These intermediate formats are optimized for quick interpretation or compilation at runtime. It was noted that bytecode is better suited for interpretation than IL. However, .NET engineers did not design IL for interpretation. Rather, unlike Java, .NET managed code is always natively compiled before execution.

Next, we learned the differences between the Java Virtual Machine and the Common Language Runtime. The primary difference was the CLR's innate support for multiple languages. In contrast, the JVM was specifically designed and optimized to execute classes developed using the Java language.

While examining Java packages and .NET namespaces, we discovered that these mechanisms are an efficient method of organizing classes and avoiding naming conflicts. We learned that the principal difference between packages and namespaces was that packages require classes to reside within a well-defined directory structure corresponding to the package name. Namespaces have no such restriction.

The Java API and .NET Framework Class Library both provide a comprehensive object library for accessing system resources. These APIs provide extensive pre-tested functionality that forms the foundation upon which programs can be built. We then discussed the differences between classpaths and references. Both of these mechanisms convey the location of required classes to the compiler. In Java, appropriate classpath settings are necessary for compilation and execution while in .NET, a reference setting is only required for compilation. The .NET runtime automatically checks the application's directory and the global assembly cache for any required classes.

We discussed how .NET assemblies are, in some ways, similar to both JAR files and Java class files. We learned that assemblies, like Java class files, are the platform's basic unit of execution. We also learned that, like JAR files, an assembly can contain any number of classes and includes a manifest file that describes its contents.

We examined the availability of Java and .NET across various hardware architectures and operating systems. It was determined that the Java runtime was available on a far wider array of hardware and software platforms. However, .NET was making strides towards porting its runtime to a few select platforms.

Finally, we discussed the versions of the Java and .NET platforms that are available and where to get them. We learned that both platforms are readily available on the Internet free of charge.

Though their runtime environments are similar in many ways, these platforms vary greatly in their support for multiple languages. In the next chapter, we will learn more about this as we examine the programming languages supported by the Java and .NET platforms.