The Programmer's Guide to Java and .NET

By Dustin R. Callaway

Copyright © 2003 Dustin R. Callaway. All rights reserved.
http://www.sourcestream.com

Chapter 1
Introduction to the Java and .NET Platforms

The Java and .NET platforms have garnered unprecedented support, excitement, and hype within the software industry. Large portions of corporate budgets have been allocated to delivering next generation technologies based on one or both of these platforms. A solid understanding of what they are—and what they are not—is critical to anyone involved in setting an organization's technical direction or implementing software-based solutions.

The purpose of this chapter is to provide a basic understanding of the nature and origin of the Java and .NET platforms. What are they? Why do we need them? These are two of the fundamental questions that will be addressed. Additionally, this chapter will discuss the history of each platform and the design goals that guided their creation. After all, in order to fully understand where a technology is going, it is helpful to know where it has been. And lastly, we'll take a brief glimpse into the future of the software industry's two brightest stars: Java and .NET.

What Are They?

The definition of the terms Java and .NET can vary widely based on the context in which they are used. Each of these names has been "overloaded" to the point where their exact meanings can often be difficult to determine. For example, the term Java may refer to a computer programming language, a class library, or an environment for building and running cross-platform applications. Likewise, depending on whom you talk to, .NET may mean a collection of similarly branded Microsoft products, a Web services deployment technology, a multi-language development environment, or a managed runtime environment. Though all of these definitions are accurate in one sense or another, this book primarily focuses on the more narrow definition associated with the terms "Java platform" and ".NET platform."

Put simply, the Java and .NET platforms are the foundations upon which programs can be built and executed. Commonly referred to as *software platforms*, these technologies implement common services that assist application development and provide an environment in which applications run. In other words, a software platform forms the context within which a program is executed.

Software platforms greatly simplify and accelerate the software development process by exposing extensive pre-built functionality and insulating the programmer from the details and complexities of the underlying operating system and hardware. In addition to making software faster and easier to write, the Java and .NET platforms improve software quality, security, performance, and maintainability. These benefits are discussed in the next section.

Let's begin by evaluating the central pieces of the Java and .NET platforms. At its core, each platform consists of a comprehensive class library and an execution engine. Combined, these two components are referred to as the *runtime environment*. The class library exposes common functionality that can be invoked by applications and the execution engine executes programs

through interpretation and/or native compilation (more about this later). Figure 1.1 illustrates where the class library and execution engine fit into the software hierarchy.

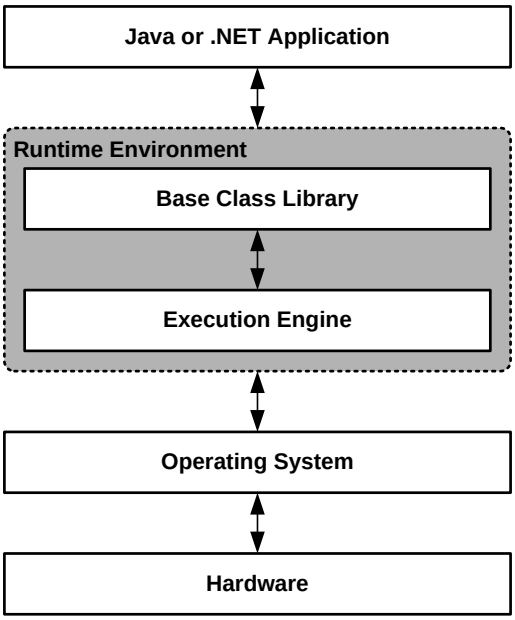

Figure 1.1 ***The Runtime Environment Sits Between the Application and Operating System***

As you might expect, the Java and .NET worlds have chosen different names to describe their various architectural components. Table 1.1 documents the names by which the Java and .NET platforms refer to their respective class libraries, execution engines, and runtime environments. Figure 1.2 shows the key components of the Java and .NET architectures.

Table 1.1 ***Java and .NET Names for Key Architectural Components***

	Java	***.NET***
Class Libraries	Java API	.NET Framework Class Library
Execution Engine	Java Virtual Machine (JVM)	Common Language Runtime (CLR)
Runtime Environment	Java Runtime Environment (JRE)	.NET Framework

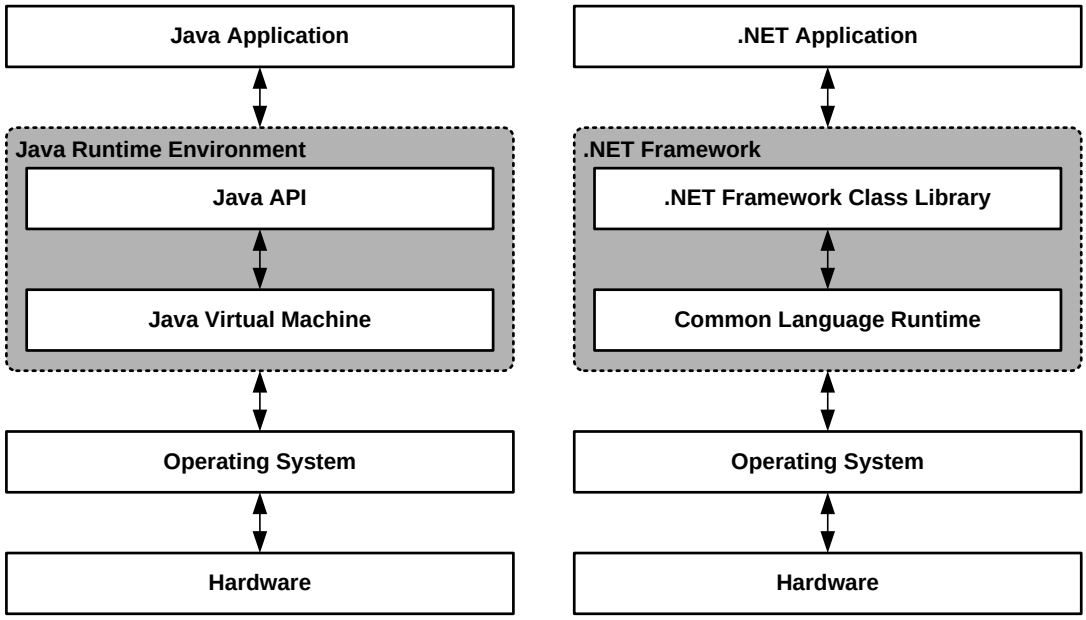

Figure 1.2 ***Key Components of the Java and .NET Architectures***

In addition to the execution engine and class library, the compiler is a critical piece of the Java and .NET platforms. Though not part of the runtime environment, a specialized compiler is necessary in order to build applications that are hosted by either platform. Similar to other compilers you may have used, Java and .NET compilers read text-based source code and generate compiled binary code. However, unlike many other compilers, Java and .NET compilers do not produce native code that is tightly coupled to the underlying hardware and operating system. Rather, Java and .NET compilers produce a processor-independent binary format that is optimized for quick interpretation or native compilation at runtime. This intermediate binary format enables the runtime environment to tailor execution to the current hardware and operating system at runtime. In Java, this binary format is referred to as *bytecode*. In .NET, it's called *intermediate language* (IL).

> **NOTE**
> Though it is usually used in reference to .NET compiled code, this book uses the term *managed code* generically when referring to either Java bytecode or .NET intermediate language. Additionally, the term *managed application* refers to an application compiled to bytecode or IL for execution by the Java or .NET runtime environment, respectively.

In addition to decoupling the code from a specific hardware configuration and operating system, managed code provides numerous other benefits. These benefits are discussed in detail in the following section.

Why Do We Need Them?

In the early days of programming, operational instructions were given by altering a calculating machine's physical configuration. Ranging from Blaise Pascal's mechanical adding machine in 1642 to Charles Babbage's difference engine in 1822, programming involved configuring a series of mechanical gears, switches, or dials in order to alter the physical motion of the machine. These physical motions were then translated into numerical calculations. With the introduction of the ENIAC in 1942, physical motion was replaced by electrical signals as a means of performing calculations. Unfortunately, each program had to be "hard-wired" into the computer's circuitry. In 1945, John Von Neumann suggested that a computer should have a simple, general purpose design that allows it to perform any computation without altering the physical structure of the machine. This concept gained momentum as punch cards were introduced as a means of providing computational instruction and data without changing the physical hardware.

Though general purpose computers were a huge step forward, programming was still painfully tedious. Initially, programs were written in the computer's native binary language (also known as "machine language") and were contained on binary-encoded punch cards. The ones and zeros encoded on these cards provided the machine with the operating instructions necessary to perform computations.

Over time, computer programming has been abstracted further and further above the computer's native language. Assembly language simplified binary programming by replacing ones and zeros with simple instructions that were understood by the computer's central processing unit (CPU). Since it designated the exact instruction set by which it could be programmed, the CPU was essentially the platform upon which assembly language programs were developed. The C language (and its predecessors) simplified programming even further by providing a high level language that could be compiled into native machine code. Though C was a huge improvement over assembly language, it still required the programmer to write complex machine-specific code to perform many common tasks such as threading or networking. Clearly, computer programming could be further simplified by additional abstraction.

The next level of abstraction came in the form of a software platform. As discussed in the previous section, a software platform is the foundation that provides services upon which applications can be built and the environment in which these applications are run. The introduction of the operating system (OS), such as UNIX and Windows, provided a software platform upon which programmers could build. The UNIX system libraries and the Win32 API (application programming interface) provided functionality that helped programmers accomplish common tasks ranging from network communication to building graphical user interfaces. These operating systems could be considered the "platforms" upon which native UNIX and Windows applications were written.

Although OS-specific function libraries tremendously simplified programming, they still left much to be desired. For instance, due to their tight coupling with the hardware and operating system, most programs were not the least bit portable. That is, if a program was to be ported to a different OS, the process usually involved a complete rewrite using the new operating system's native API. Additionally, the compilers for these operating systems generated machine-specific native code that would not run on disparate processors. Though native code executes quickly, it is plagued by numerous problems including glaring deficiencies in the areas of portability, safety, security, speed and simplicity of development, dynamic programming, distributed computing, and maintainability.

So, at last, we arrive at the Java and .NET platforms. As previously discussed, these technologies are software platforms that sit on top of an operating system further insulating the programmer from the underlying hardware and OS. Both of these platforms provide a

comprehensive API that greatly assists application development as well as a runtime environment within which compiled code is run. Let's examine how the Java and .NET platforms address the deficiencies inherent in natively compiled alternatives.

Portability

Java and .NET compiled code is far more portable than natively compiled applications. This is due to the fact that Java and .NET applications are compiled into an architecture-independent intermediate form rather than hardware-specific machine code. The Java bytecode and .NET IL formats are highly optimized for fast interpretation or native compilation on the host platform at runtime. Furthermore, Java and .NET avoid common portability pitfalls by explicitly defining architecture-dependent characteristics such as the size of primitive data types, arithmetic behavior, and byte ordering. In this way, it is possible to execute compiled Java and .NET code on any platform for which the appropriate runtime environment is available.

Though Java and .NET programs are highly portable, there is a caveat. It is entirely possible to limit portability through the use of native method calls or "unsafe" code. These issues are discussed in detail in Chapter 2. Lastly, as presented in the upcoming section on design goals, it should be noted that Java places greater emphasis on portability than does .NET and may enjoy the benefits of portability to a higher degree.

Safety

Before any code is executed on the Java or .NET platform, it must first be verified safe by the execution engine. This verification process ensures that the compiled code is properly formed and does not perform any illegal operations (e.g., forged pointers, illegal object casts, buffer overflows). Though legitimate compilers generate only valid code, an attacker may maliciously alter code after compilation or compiled code may inadvertently become corrupted, thereby making it unsafe.

In addition to code verification, the Java and .NET platforms are safer than native code because they do not allow applications direct access to memory. Though .NET applications can circumvent this safety precaution by marking code as unsafe, .NET's default security model does not permit the execution of unsafe code unless it comes from a fully trusted source. By restricting direct memory access, errors that commonly result from memory access violations (e.g., errant pointers in C) are completely eliminated. Similarly, all array and string accesses are bounds checked at runtime. Checking array and string bounds at runtime prevents programs from performing illegal operations such as writing data beyond the end of an array or reading memory from outside the bounds of a string.

The Java and .NET platforms' automatic garbage collection is another important safety feature. Unlike Java and .NET, native code solutions require disciplined programming in order to ensure that obsolete objects do not consume memory resources indefinitely. To accomplish this, native code programmers must manually release memory by explicitly destroying objects that pass out of scope or are no longer in use. Memory management is a tedious and error-prone task. If not managed properly, an application may continually consume memory until it is exhausted. This type of error, known as a *memory leak*, is one of the most common problems that afflict native code applications.

The Java and .NET platforms avoid memory management issues through automatic garbage collection. *Automatic garbage collection* is the process of automatically reclaiming memory that an application has dynamically allocated but is no longer using. In Java and .NET, the garbage collector is implemented within the execution engine (i.e., Java Virtual Machine and Common Language Runtime, respectively). The garbage collector manages memory by keeping track of all

references to each object. When no more references exist, the object is considered obsolete and is marked for garbage collection. The garbage collector, which typically executes periodically on a low priority thread, then reclaims the memory consumed by all obsolete objects. In short, the Java and .NET garbage collectors track references to all objects and automatically release memory allocated to objects for which no references exist (e.g., the object has passed out of scope or has been removed from a collection). In this way, the programmer is freed from the mundane tasks of memory management and the possibility of memory leaks is greatly reduced.

Lastly, the Java and .NET platforms provide robust exception handling that ensures predictable program behavior even when unexpected conditions are encountered. An *exception* is an error condition that alters normal program execution. Though native code applications may gracefully handle expected exceptions (e.g., file not found exception), they may subsequently crash on unexpected exceptions (e.g., out of memory error). The execution engine insulates the Java and .NET platforms from this problem. If not caught by a Java or .NET application, an exception will propagate up the calling chain until it is handled by the execution engine itself. In this manner, no exception will go uncaught and the system will not crash if a programmer fails to handle an unexpected condition. Though the execution engine will handle all uncaught exceptions, it is usually better to handle these conditions within the application. The Java and .NET exception handling features make it very easy for the programmer to capture and recover from almost any type of exception that can be anticipated.

Security

Java and .NET security is largely derived from the safety features incorporated into each platform. For example, the code verification process guarantees that maliciously constructed code will not be executed. Likewise, since both platforms restrict direct memory access, programs cannot interfere with other system processes or read from memory beyond that which they have been granted access. As you can see, the safety features built into each of these platforms significantly contribute to their security.

In addition to their stringent code safety requirements, the Java and .NET platforms include a complex security sub-system that oversees the execution of all managed applications. The security sub-system grants privileges to applications based on their origin (i.e., from where or whom they came). For instance, either platform may establish a security policy that grants code residing on the local machine or local network full access to the system while significantly restricting access granted to code downloaded from the Internet. For example, allowing an untrusted remote application access to your local file system is extremely insecure. If full file system access is granted, an untrusted application may steal information from local files, delete files or directories, or even format your hard drive. On the other hand, you may wish to grant an application installed on your local machine full access to the file system so that it can scan for viruses, archive files on your hard disk, or perform other important tasks. Both Java and .NET permit privileges to be granted based on the origin of managed code.

> **NOTE**
> The restricted environment within which untrusted code is executed is typically referred to as a *sandbox* (Java) or *protected environment* (.NET).

In addition to granting privileges according to the code's origin, the Java and .NET security facilities allow rights to be granted based on the author. If you are able to positively establish the identity of the code's creator, you can determine whether or not you trust the code to behave in an expected manner (e.g., not steal information or damage your system). Basically, trusted code works on this principle—if you trust the creator, you can trust the code. Both Java and .NET establish trust through the use of digital signatures. *Digital signatures* are used to

cryptographically sign code in a manner that allows the recipient to positively identify the code's author and determine whether the code has been altered. Based on this information, privileges may then be granted regardless of the location from which the code was retrieved. In this way, code downloaded from remote locations can be granted additional privileges if created by a trusted party.

As you can see, security was designed into the Java and .NET platforms from the start. Though no system is completely secure, Java and .NET come far closer to this ideal than natively compiled alternatives. For more information on security, see Chapter 10.

Speed and Simplicity of Development

The Java and .NET platforms make the development process far faster and simpler than traditional programming. These productivity gains are realized largely due to three critical Java and .NET programming features—automatic garbage collection, simple object-oriented languages, and a comprehensive class library.

As discussed earlier, automatic garbage collection frees the programmer from mundane memory management responsibilities. Java and .NET allow the developer to focus on implementing business logic rather than scrutinizing code for possible memory leaks. Since complex memory allocation and de-allocation is handled for you, garbage collection simplifies and accelerates the programming process.

The simple object-oriented languages supported by both platforms are another source of programmer productivity improvements. Though the .NET platform supports numerous programming languages, C# (pronounced "see-sharp") is the .NET language presented throughout this book. C# was chosen because it is a simple, pure, object-oriented language that maps perfectly to the capabilities of the .NET platform. The fact that C# fits the .NET environment so well is no accident. Unlike other .NET-supported languages, C# was specifically designed as a .NET language from the start. Additionally, as a brand new language, C# was able to adopt the best ideas from the most popular existing languages including Java, C++, and Visual Basic. Very similar to C#, Java is also a simple, pure, object-oriented language that maps perfectly to the Java platform. This might be expected considering the fact that the Java programming language is the only language officially supported on the Java platform (though bytecode compilers do exist for other languages).

> **NOTE**
> In addition to many other languages, the .NET platform supports the Java programming language. In fact, Microsoft has shipped a Java IDE (integrated development environment) for .NET called Visual J#. This product serves as a replacement for the outdated Visual J++ IDE. Though it incorporates the Java language, Visual J# does not produce Java bytecode and is not compatible with the Java platform. Rather, it is a tool for building .NET applications using the Java language and the .NET class library. Though the Java programming language is supported by .NET, the author chose to use the language that most appropriately matched the given platform—C# for .NET and Java for the Java platform.

These two languages—C# and Java—improve the speed and simplicity of development by eliminating much of the complexities and baggage inherent in traditional languages. For instance, C# and Java do not support many of the most complex and error-prone features of C++ such as

multiple inheritance, templates, header files, and pointers (C# supports pointers only if the code is marked "unsafe"). Additionally, unlike natively compiled C++, C# and Java have built-in support for threading, networking, polymorphism through interfaces, and complex data types (such as strings). The removal of complex and outdated aspects of older languages coupled with the addition of new productivity features contribute to make Java and .NET programming much faster and simpler than traditional programming.

The fact that the Java and .NET platforms provide a consistent API across all supported platforms greatly improves developer productivity. Rather than having to learn a different API for each platform, programmers can focus on learning a single comprehensive class library and leverage that knowledge across many platforms and, in the case of .NET, across many languages. For instance, prior to .NET, developers built Windows applications by invoking COM objects or issuing DLL calls using the C-based Win32 API. In contrast, with .NET (as well as Java), programmers can learn a single consistent API that is applicable to all supported platforms ranging from cellular phones to high-end servers.

Finally, Java and .NET make programming faster and simpler through the enormous amount of functionality exposed by their class libraries. These well-tested and performance-optimized libraries provide functionality to perform most common tasks including database access, XML parsing, networking, object serialization, collection management, user interface development, string manipulation, and remote object calls. These libraries simplify and accelerate programming by allowing developers to utilize pre-built classes for common functions rather than having to "re-invent the wheel."

Dynamic Programming

The Java and .NET platforms provide strong support for dynamic programming. *Dynamic programming* refers to the ability to locate, load, and instantiate classes (local or remote) at runtime as well as obtain information about a class at runtime and dynamically invoke its methods.

The dynamic nature of Java and .NET offers the programmer great power and flexibility. For example, Java and .NET make it is possible for an application to instantiate a class whose name was not known at compile time. Rather, the class's name could have been read from a configuration file on disk or received over the network at runtime. Similarly, through a process known as *reflection*, programs can dynamically retrieve information about a class and its members including the signatures of methods and constructors, data types of fields, events supported, and interfaces implemented. Through reflection, it is possible to determine the type of any object at runtime as well as dynamically access public fields or invoke public methods whose names were not known at compile time. This sort of dynamic programming is not supported by traditional "compile, link, and run" types of languages.

In addition to runtime class loading, the dynamic features of Java and .NET allow classes to be altered and recompiled without affecting their clients or subclasses. Unlike traditional languages, Java and .NET do not convert named variable and method references to numeric values at compile time. Rather, symbolic references within managed code are resolved at runtime. Upon first encounter, these references are replaced by numeric offsets in order to improve performance on subsequent calls. Likewise, the process of mapping objects to memory is postponed until runtime. Resolving named references and laying out objects in memory at runtime rather than compile time allow managed code classes to change without requiring clients and subclasses to be recompiled.

Distributed Computing

The Java and .NET platforms were designed to support distributed computing. *Distributed computing* allows an application's various components to be logically and physically partitioned. That is, these components may execute within different processes on the same computer or reside on separate computers that are connected by a network. For example, a Web browser is a type of distributed application. Even though it runs on a user's local computer, it invokes services and retrieves information from other computers connected to the Internet. In this way, the processing of requests is distributed across multiple machines.

There are three important areas in which the Java and .NET platforms support distributed computing. First, each platform supports dynamic class loading from remote locations. This allows a local application to load and execute code retrieved from another machine. Second, both platforms provide sophisticated features for implementing remote procedure calls (known as remote method calls in object-oriented programming). These features simplify distributed programming by allowing applications to invoke methods on remote objects as if they were local. Though they may appear local to the application, remote method calls are executed on another computer and the results are returned to the caller. Last, Java and .NET provide strong support for Web services. The term *Web services* refers to a platform-independent technology that facilitates remote method calls between disparate systems using industry standard protocols. As discussed in Chapter 9, Web services are a primary method of integrating Java- and .NET-based systems.

Maintainability

Code maintenance is a major concern during and after system development. Fortunately, the Java and .NET platforms excel in this area. *Maintainability* refers to how simple it is to fix, alter, or enhance a system. Java and .NET simplify code maintenance through their common class libraries and services as well as the application of simple, object-oriented languages and integrated code documentation features.

First, since all Java and .NET applications are built upon their respective class libraries, any programmer familiar with these libraries can contribute to the code maintenance effort without additional training. Many natively compiled alternatives utilize custom libraries for performing common tasks. These libraries often require a significant amount of time and training before a developer is proficient enough to maintain the code. Additionally, other common services provided by each platform, such as automatic garbage collection, simplified threading, well-defined data types, strong type checking, and security, contribute to easier maintenance.

Second, the pure object-oriented nature of C# and Java facilitate good programming practices such as object reuse, inheritance, polymorphism, and encapsulation. A properly constructed object-oriented system is far easier to understand and maintain than traditional applications. Though C++ is also an object-oriented language, it is far more complex and difficult to learn than C# or Java.

Lastly, the Java and .NET platforms provide innovative tools and techniques for documenting code. Both platforms encourage developers to add specially formatted documentation to their source code as it is written. Once the code is complete, an external tool can be used to extract the documentation into HTML or XML files for indexing and presentation. This stream-lined documentation process removes many of the hurdles that have discouraged programmers in the past from properly documenting their code. And naturally, clearly documented code is essential to code maintenance.

History

Though they are very similar in many respects, the Java and .NET platforms traveled very different paths on their way to becoming the industry's premier enterprise development platforms. For example, Java was originally positioned as an operating environment for consumer electronics devices and interactive television. On the other hand, .NET was initially envisioned as a wholesale replacement for the complex and outdated Windows API. Though their origins differed greatly, we will see that these platforms ended up solving many of the same problems. However, before we further investigate the present and future of Java and .NET, let's first take a look at their past.

Java Platform

Development of the Java platform began in 1990 at Sun Microsystems as a research project known as the "Green Project." With a staff of 13 people, the goal of this project was to produce software that intelligently connected all types of networked devices. The original business plan involved licensing the technology to numerous manufacturers within the consumer electronics industry. Given the fact that consumer devices were powered by a wide array of processors and hardware architectures, it was decided early on that existing programming languages were inadequate for this type of project. A portable language and operating environment was required if the Green Project was to succeed.

The new programming language was created by "Green Team" member James Gosling. Gosling realized that if a programming language was to be embedded in thousands of networked devices, it must be secure, reliable, distributed, and portable. With these goals in mind, he set about creating a new language platform originally called Oak—named after a tree outside of his office window. Loosely based on C++, Objective C, Eiffel, and SmallTalk, Oak was new and innovative in a number of significant ways. First, the Oak compiler did not produce native code. Rather, Oak programs compiled to an intermediate form, known as bytecode, that could be efficiently interpreted at runtime. The interpreter provided security and reliability through automatic garbage collection, bytecode verification, and array bounds checking. Additionally, Gosling decided that many of the more complex and error-prone features of C++, such as pointers and multiple inheritance, only contributed to bugs and security vulnerabilities. By leaving these features out, he eliminated a whole host of security and reliability issues. The Oak language was distributed in the sense that networking facilities were built in and the language allowed code to be downloaded from another machine and executed locally within a secure environment. Finally, due to its interpreted nature, Oak was architecture independent and portable to any platform for which its interpreter was available.

In the summer of 1992, the Green Team completed a working prototype of a networked remote control device for home entertainment equipment. Named after the code used by Green Team members to answer their phones from another extension, this device was officially dubbed *7 (pronounced "star seven"). After a demonstration to Sun executives, *7 was an instant hit. Sun's CEO, Scott McNealy, believed that the Green Team had developed something truly unique. Within a few months of the demonstration, a new company was formed for the purpose of developing and marketing the technologies demonstrated by the Green Project. The new company, a wholly owned subsidiary of Sun Microsystems, was called FirstPerson, Inc.

Though the initial response was positive, FirstPerson struggled to find customers and a suitable market. After failed attempts at licensing software for cellular phones and home automation, FirstPerson focused their attention on the emerging interactive television market. Though the strengths of the Oak language platform seemed ideally suited for set-top boxes, it turned out that the interactive television market was far too immature and unprofitable. In 1994,

after failing to convince Sun executives that it could produce a viable business plan for marketing the Oak technology, FirstPerson was absorbed back into Sun Microsystems.

After lying dormant for months, Bill Joy, a Sun-cofounder and distinguished engineer, resurrected Oak as the World Wide Web began to emerge. With the introduction of the revolutionary Mosaic browser in 1993, interest in the World Wide Web began to grow at a phenomenal rate. Joy realized that the Web was just the kind of distributed, networked environment for which Oak was originally designed. After garnering the support of key Sun executives, the Oak project was revived, newly named the "Liveoak Project," and James Gosling was tasked with adapting his language for the Internet. About this time, a patent search revealed that the Oak name was not available. After much brainstorming, patent searching, and a trip to the local coffee shop, Java emerged as the official name of Gosling's creation.

Meanwhile, a brash young Sun engineer (and an original member of the Green Team) by the name of Patrick Naughton was given the assignment of creating a compelling application to demonstrate the power of the Java language. Naughton responded by creating a Java-based browser originally called WebRunner and later renamed HotJava. This browser proved to be far more compelling than anything available at the time. Not only could it display text and images like Mosaic, HotJava could also dynamically download and execute Java applets. Applets were small Java applications that ran in a secure "sandbox" within a browser. Java's security and portability were ideally suited to the heterogeneous nature of the Web. HotJava and Java applets were greeted with an acceptance and enthusiasm unmatched in the industry. In 1995, the hype and excitement around Java reached a fever pitch when Mark Andreesen, co-founder of Netscape Communications, announced at the SunWorld conference that Netscape would license Java for use within its incredibly popular Navigator browser. After Andreesen's endorsement, one company after another announced support for Java and Sun could hardly keep up as they rapidly licensed the technology to a growing field of interested parties. Java had suddenly aroused the entire industry. As it breathed new life into thousands of static HTML pages, Java had finally found a home as the ultimate enabling technology of the World Wide Web.

Though Java initially gained fame through client-side applets, with the introduction of servlets in 1997, it was quickly discovered that Java's real strength was on the server. Somewhat analogous to server-side applets, servlets were portable applications that executed on the server (unlike applets that executed on the client). Unlike applets, servlets did not present a user interface to the client. Rather, they were primarily used to dynamically generate HTML in response to browser requests. Far more efficient than CGI, servlets were rapidly adopted by Internet programmers for building Web sites that were secure, reliable, and portable.

Shortly after the introduction of servlets, Sun announced a new technology for implementing portable business logic on the server. This technology, named Enterprise JavaBeans, provided developers with a standard and much simplified way of creating robust and scalable server-side business objects that could be invoked over the network. Servlets and Enterprise JavaBeans quickly became a powerful combination for creating enterprise-class software solutions.

Today, Java is one of the most widely used languages in the world and the Java platform has evolved into a mature and robust foundation for building enterprise applications. Java can be found everywhere from college campuses to scientific labs. From its humble beginnings as an operating environment for consumer devices, Java has become a de facto standard and defining technology in the software industry.

.NET Platform

In 1997, Microsoft executives were well aware that they had arrived late to the Internet party. Though their software controlled more than 95% of the world's personal computers, Bill Gates and company realized that the Internet presented the first real threat to their monumentally

successful Windows monopoly. After all, IBM had held an equally envious position during the 1960s and 70s until the industry's rapid transition from mainframes to personal computers caught the company off guard. Exploiting this vulnerability, Microsoft was able to wrest control of the software industry away from IBM. Realizing that the Internet represented a paradigm shift of equal magnitude to the transition that doomed IBM, Microsoft executives began developing a plan for maintaining Window's dominance in an increasingly Internet-centric world. In 1997, this plan took the form of an ambitious initiative known as Project 42.

Project 42 was an internal effort aimed at developing tools for building distributed, highly-available, Web-based systems out of potentially unreliable components. The project got its name from the science-fiction novel *The Hitchhiker's Guide to the Galaxy* where central characters search for the answer to the question: What's the meaning of the universe? The answer, they discover, is 42. Given his success at leading their charge into the browser market, Microsoft tapped James "J" Allard to oversee the project. Though it began with lofty goals, Project 42 proved too ambitious and the organization was dissolved in May of 1999. However, much of the work initiated under this project, including XML support, the C# language, and the CLR, eventually evolved into the underpinnings of the .NET platform.

Though Project 42 did not achieve all of its goals, Microsoft knew that they must embrace the Internet if they were to remain relevant in a wired world. This time Microsoft turned to veteran technical leader Paul Maritz to lead the charge. Having overseen the Windows product line for much of his career, Maritz was tasked with developing an innovative strategy that incorporates the Internet while leveraging the Windows operating system. Maritz immediately went to work. To jumpstart the effort and provide additional motivation, he immediately scheduled the date that the new plan would be publicly unveiled. The deadline was set at September 13, 1999.

Steve Ballmer, Microsoft's current CEO, agreed to make the presentation. After initial disappointment, Ballmer caught the Maritz's vision for marrying Windows and the Web. The idea was to migrate away from the traditional shrink-wrapped software business and towards selling software as a service over the Internet. At Microsoft's annual employee meeting in September of 1999, Ballmer enthusiastically presented Microsoft's new strategy for leveraging the Web. The Internet would play a key role in this new plan but Windows would remain central.

Once the general strategy was approved, Maritz's team was responsible for filling in the details. After countless brainstorming sessions, the team focused their attention on providing a means whereby a Windows application could interact with distributed components on the Web. This concept, known as Web services, seemed to fit their goals perfectly by providing a means of weaving Windows and the Web into a single fabric. Windows applications would no longer be isolated to the PC. With Web services, they could seamlessly interact with remote systems over the Internet. From purchasing a book to reserving a flight, Windows and Web services seemed like a combination that could complete just about any task while providing a richer and friendlier user experience than basic Web browsing could afford on its own. This melding of Windows and the Web became the driving force behind the creation of the .NET platform and the keystone of Microsoft's Internet strategy.

Development of the .NET platform progressed through many different projects and was known under many different code names. Following Project 42, key .NET technologies were successively known as COM Object Runtime (COR), Lightning, COM+2.0, Universal Runtime (URT), COM#, and Next Generation Web Services (NGWS) before being formally unveiled in July 2000 as the Common Language Runtime (CLR) and the .NET Framework.

Today, .NET is a robust platform upon which scalable, distributed, enterprise applications can be built and hosted. Though Microsoft marketing primarily targets enterprise customers, .NET provides general purpose functionality capable of building all types of applications ranging from Windows applications to Web sites to Web services. Though Microsoft arrived late to the

Internet party, with .NET, they made quite an impressive entrance. With its popularity growing rapidly, it is apparent that .NET will represent the greatest challenge and most viable alternative to the Java platform.

Design Goals

The Java and .NET platforms share many design goals. Design goals are the guiding principles that direct the creation of a product. Whenever a tradeoff is required, every effort is made to see that the tradeoff favors a primary design goal over competing considerations. Typical tradeoffs might include choosing security over performance, portability over functionality, simplicity over flexibility, or reliability over memory efficiency.

Before we discuss how Java and .NET design goals differ, let's first examine where they are similar. The following design goals guided the creation of both platforms.

- Simple

  Platform class libraries and programming languages used by the platform should be easy to learn and simple to use.

  Java and C# are very simple to learn and apply. This also holds true in regard to the Java API and .NET Framework class library.

- Familiar

  Programming languages used by the platform should be familiar to the average programmer.

  Java and C# are syntactically similar to C and C++.

- Object-Oriented

  Programming languages used by the platform should adhere to modern object-oriented principles in order to improve code reuse, maintenance, distributed processing, and other programming best practices such as encapsulation, data hiding, inheritance, and polymorphism.

  Java and C# are pure object-oriented languages that were designed from the ground up according to object-oriented guidelines.

- Strongly Typed

  Programming languages used by the platform should be strongly typed in order to catch as many errors as possible at compile time.

  Java and C# are strongly typed and require casts and method declarations to be made explicitly as opposed to C-style implicit casts and declarations.

- Secure

  The platform should execute code within a safe environment that limits the accessibility of untrusted code. The environment should validate code before execution to ensure that it does not attempt to perform operations that violate any of the platform's security policies.

  Java and .NET run applications within a protected environment (or "sandbox") in order to restrict the functionality of untrusted code. Both platforms verify that managed code has not been maliciously constructed or altered before it is executed.

- Reliable

The platform should facilitate reliability through automatic garbage collection and robust exception handling.

Java and .NET relieve the developer of error-prone memory management tasks by providing automatic garbage collection. Additionally, both platforms provide a robust exception handling mechanism for capturing and recovering from errors.

- Dynamic

  The platform should be dynamic. Classes should be loadable, instantiable, and callable at runtime. Programs should be able to determine the class type of any object at runtime. The dynamic nature of the class should allow classes to be altered and recompiled without requiring the recompilation of clients or subclasses.

  Java and .NET are highly dynamic platforms that support all of these features.

- Distributed

  The platform should have strong support for distributed computing. Distributed computing allows an application's components to reside on various machines connected over a network.

  Java and .NET support distributed computing through support for remote method calls and robust networking facilities as well as the ability to download remote classes and instantiate them locally.

- Scalable

  The platform must support a wide range of hardware configurations ranging from resource limited cellular phones and handheld computers to resource rich workstations and servers. Additionally, as a means of scaling economically and efficiently, the platform must be capable of supporting multiple processors and distributing load across physical machines.

  Java and .NET are scalable across a wide array of hardware configurations and distributed architectures.

- Multithreaded

  The platform and all applicable languages should innately support multithreaded programming. Modern applications are required to perform multiple tasks concurrently. A multithreaded platform is required to ensure that these applications perform responsively.

  Java and .NET provide integrated multithreading support at both the platform and language levels.

- Architecture Neutral

  Code compiled to an intermediate form should be architecture independent. Arithmetic behavior, byte ordering, and the size of all data types should be well-defined and consistent across all supported architectures.

  Java and .NET strictly define data types and behaviors in order to insure architecture neutrality.

Now that we have discussed the design goals that Java and .NET have in common, let's examine some of the areas where they diverge.

Java Platform

The Java design team established early on that Java should be interpreted, portable, and safe. Each of these features were deemed critical in light of the Java team's mission to develop a simple and secure operating environment for connected devices that is both architecture and operating system independent. In addition, the Java creators decided that standardizing on a single common language and developing specifications that could be implemented by any number of vendors was beneficial to the platform.

- Interpreted

  Java was originally designed as an interpreted language for several reasons. First, being interpreted, a Java application could run wherever an interpreter was available regardless of the host's hardware and operating system. Given the appropriate interpreter, binary files compiled as bytecode could be executed on any platform without recompilation. Second, Java's interpreted nature enabled powerful dynamic capabilities such as the ability to load classes at runtime or download remote classes from across the network. Lastly, interpreting code is, for the most part, less processor and memory intensive than native compilation. Since Java was originally designed for consumer electronics and other simple networked devices, interpretation was deemed necessary in order to ensure that devices with very limited resources, such as cellular phones and set-top boxes, could efficiently execute Java bytecode.

- Portable

  Java was designed from the ground up to be portable across architectures and operating systems. Java's ability to target a wide range of hardware and software platforms was considered critical to its success. Therefore, when confronted with making a choice between higher performance or more functionality and portability, the Java designers almost always decided in favor of portability.

- Safe

  First and foremost, Java was designed to be safe and secure. Removing pointers from the Java language was a critical design decision that contributed to this goal. By not allowing programmers direct access to memory, a whole host of safety and security concerns were eliminated. Though the lack of pointers sometimes made it difficult to achieve high performance for certain operations, safety and reliability were deemed more important.

- Single Language

  The Java platform was originally devised and constructed to efficiently compile and execute Java source code. Though it is possible to compile other languages to Java bytecode (and many such compilers exist), multiple language support was not a driving design principle. Rather, it was believed that the Java platform was better served by focusing on a single language around which the Java community could coalesce. The arguments in favor of this decision included the opinions that projects developed in multiple languages are much more difficult to maintain and enhance, fracturing the community into various "language camps" would slow the adoption and evolution of the Java platform, and focusing on a single language simplified developer training and support costs.

- Multi-Vendor

  Early on, Sun Microsystems decided to capitalize on its new language platform through licensing Java technology and trademarks to companies that wished to compete in the Java market. That is, rather than focusing primarily on building and marketing Java products, Sun attempted to foster a vibrant and competitive software community by

providing common specifications upon which all industry participants could build compatible products. The products that demonstrated superior quality, standards compliance, performance, flexibility, and scalability are the ones that would thrive. To further drive Java as a de facto industry standard, Sun established the Java Community Process (JCP). The JCP is an open organization that gives all interested parties a vote regarding the evolution of the Java platform (see *http://jcp.org*). Though it is headed by Sun, the JCP allows anyone to innovate on the Java platform and submit their innovations as proposed standards. Given sufficient community support, any organization's contributions can become official Java standards. This concept of producing public specifications upon which many companies compete has driven Java adoption throughout the industry.

.NET Platform

The .NET design team diverged from the Java platform's primary design goals in a few areas. In particular, the .NET designers decided that features like native compilation at runtime, full operating system functionality, and maximum performance should receive top priority. It was recognized and accepted that these priorities would come at a small cost to safety and portability. Additionally, support for numerous and diverse programming languages was considered critical to the platform's success. Finally, initial design goals also included versioning and ease of deployment features. And, as discussed below, we will see that Microsoft is comfortable positioning itself as the only vendor of the complete .NET offering.

- Natively Compiled at Runtime

  Though .NET applications are compiled to IL, they are never interpreted. Rather, all .NET applications are natively compiled before execution. Typically, this compilation is dynamically performed by the CLR at runtime and the natively compiled code is cached for future use. Dynamic compilation at runtime is commonly referred to as "just-in-time", or "JIT", compilation. Both the Java and .NET platforms utilize JIT compilers extensively. Though Java uses JIT compilation to enhance performance, it also supports interpretation for limited resource environments. In contrast, to maximize performance, .NET requires native compilation and does not support interpretation. This tradeoff gives .NET an edge in raw performance but limits portability on less functional platforms.

- Full Functionality

  Some have argued that in some areas Java only supports "lowest common denominator" functionality in order to maximize portability. In other words, if a feature is not implemented by a large majority of supported platforms, it would not be supported by Java. The .NET platform designers took another approach and decided to fully support platform-specific functionality (typically referring to operating system functionality) at some cost to portability.

  Platform-specific functionality is invoked using .NET's Platform Invocation Services, or PInvoke, interface to issue direct calls to the underlying operating system. In many ways, PInvoke is similar to the Java Native Interface (JNI) API. Both features allow a managed application to make direct calls into unmanaged code. However, because JNI limits Java's portability, its use is strongly discouraged. Sun Microsystems even introduced a "100% Pure Java" branding program that encourages the development of completely platform independent applications and discourages platform-specific JNI calls. On the other hand, .NET encourages the use of platform-specific functions whenever additional performance or functionality is desirable. Of course, since both JNI and PInvoke make

calls into unmanaged code, any additional performance or functionality realized by these invocations comes at a cost to safety as well as portability.

- Maximum Performance

  In addition to the PInvoke functionality discussed previously, the .NET platform supports the execution of "unsafe" code in order to maximize performance. The term "unsafe" comes from the C# keyword of the same name. By flagging a block of code as unsafe, the .NET platform extends C# (or other .NET languages) the ability to use pointers and directly manipulate memory. Though this type of execution is inherently unsafe, the .NET designers felt that the platform should give the developer every option to tune a "managed" application for maximum performance. Of course, as soon as a managed application implements unsafe code, it is no longer fully managed which results in additional safety, reliability, and portability concerns. Fortunately, being a highly secure platform, .NET strictly limits execution of unsafe code to fully trusted applications only.

- Multiple Languages

  Largely due to the fact that the Microsoft development community spanned numerous languages ranging from Visual Basic to C++, multiple language support in .NET was deemed critical if the company was to successfully drive adoption of an entirely new programming model. In addition to spurring migration of its existing developer base to .NET, Microsoft argued that multiple language support would allow programmers to choose the right tool for the job rather than having their choice dictated to them. It was also reasoned that training and migration costs could be greatly reduced if aspiring .NET programmers were only required to learn a new class library and not an entirely new language. Though some have argued that multiple language support is merely syntactic and somewhat superficial since all .NET languages have essentially the same capabilities, at the very least it does allow developers to choose a language structure and syntax with which they are comfortable.

- Versioning

  Traditional Microsoft applications were plagued by a versioning issue often referred to as "DLL hell." This issue revolved around the fact that when a shared DLL was updated by a new application, older applications that utilized that DLL often broke. The .NET designers went to great lengths to solve this problem. Through the use of embedded version information within all shared and private components coupled with language keywords that strictly regulate method overriding and inheritance, the .NET platform ensures that new versions of shared components and class libraries will not break applications that depend on older versions.

- Ease of Deployment

  Traditionally, installation of a Windows application was complicated and burdensome. It required placing numerous files in various directories and properly updating complex registry settings. In order to simplify the deployment process, it was decided that all .NET components should be self-describing thereby eliminating the traditional requirement that components be registered in the registry. This design goal mandated that .NET application deployment should require nothing more than simply copying program files to the destination directory. By incorporating metadata within all applications, .NET has eliminated its reliance on the registry and greatly simplified the deployment process.

- Single Vendor

  The .NET platform is largely a single vendor product. That is, Microsoft is the only company that provides a full implementation of the .NET platform and technologies. Though the C# language and the Common Language Infrastructure (i.e., the CLR plus

the .NET base class library) have been accepted as official ECMA standards, the standardized portions of .NET represent only a small subset of its complete functionality. For instance, critical .NET technologies such as ASP.NET, ADO.NET and Windows Forms have not been standardized. Further more, since Microsoft does not produce public specifications regarding these technologies, it is unlikely that many, if any, other companies will compete with Microsoft in the implementation of these products. Though a single vendor technology solution must be evaluated with a critical eye, the advantage of this approach is that there are no compatibility concerns between competing vendors' products. For example, given that Microsoft is the only vendor, if your .NET solution runs on Microsoft servers, it is guaranteed to run everywhere that .NET is supported.

The Future

The battle for developer mindshare and corporate dollars in the enterprise space has essentially been reduced to a two-horse race—a heated competition between the Java and .NET platforms. Looking forward, it is the author's opinion that the vast majority of enterprise applications will be built upon one or both of these dominant platforms. Consequently, learning Java and .NET is time well spent for any forward-looking developer. At this point in time, betting your programming career on Java and/or .NET appears to be a very safe wager.

Summary

This chapter introduced the Java and .NET platforms. It began by discussing what they are and why we need them. Through this discussion, we learned that these platforms provide a solid foundation upon which programmers can build secure and robust applications. Core components of Java and .NET include their respective class libraries, execution engines, and compilers. These platforms excel over traditional programming methods in many areas including portability, safety, security, speed and simplicity of development, dynamic programming, distributed computing, and maintainability. We also learned that the design goals that shaped the Java and .NET platforms were very similar; diverging only in regard to the prioritization of some key features such as safety, portability, functionality, and performance. Finally, a brief history of each platform was presented and a prediction regarding the future of Java and .NET was offered.

Now that we have a basic understanding of the Java and .NET platforms, it's time to dig a little deeper. In the next chapter, we will examine the similarities and differences between the Java and .NET runtime environments.