Part I

Introduction to Java

Chapter 1
Introduction

1.1 What is Java?

The term Java refers to more than just a computer language like C or Pascal. Java encompasses several distinct components:

- **A high-level language** – Java is an object-oriented language whose source code at a glance looks very similar to C and C++ but is unique in many ways.
- **Java bytecode** – A compiler transforms the Java language source code to files of binary instructions and data called bytecode that run in the Java Virtual Machine.
- **Java Virtual Machine (JVM)** – A JVM program takes bytecode as input and interprets the instructions just as if it were a physical processor executing machine code. (We discuss actual hardware implementations of the Java interpreter in Chapter 24.)

Sun Microsystems owns the Java trademark (see the next section on the history of Java) and provides a set of programming tools and class libraries in bundles called Java Software Development Kits (SDKs). The tools include `javac`, which compiles Java source code into bytecode, and `java`, the executable program that creates a JVM that executes the bytecode. Sun provides SDKs for Windows, Linux, and Solaris. Other vendors provide SDKs for their own platforms (IBM AIX and Apple Mac OS X, for example). Sun also provides a runtime bundle with just the JVM and a few tools for users who want to run Java programs on their machines but have no intention of creating Java programs. This runtime bundle is called the Java Runtime Environment (JRE).

In hope of making Java a widely used standard, Sun placed minimal restrictions on Java and gave substantial control of the development of the language over to a broadly based Java community organization (see Section 1.4 "Java: open or closed?"). So as long as other implementations obey the official Java specifications, any or all of the Java components can be replaced by non-Sun components. For example, just as compilers for different languages can create machine code for the same processor, there are programs for compiling source code written in other languages, such as Pascal and C, into Java bytecode. There are even Java bytecode assembler programs. Many JVMs have been written by independent sources.

Java might be said more accurately to refer to a set of programming and computing specifications. However, in this book and the Web Course, unless otherwise indicated, we follow the common use of the term Java to refer to the high-level language that follows the official specifications and the virtual machine platform on which the compiled language runs. The usage is normally clear from the context.

Finally, many people know Java only from the applets that run in their web browsers. Java programs, however, can also run as standalone programs just like any other language. Such standalone programs are referred to as "applications" to distinguish them from applets.

1.2 History of Java

During 1990, James Gosling, Bill Joy and others at Sun Microsystems began developing a language called Oak. They primarily intended it as a language for microprocessors embedded in consumer devices such as cable set-top boxes, VCRs, and handheld computers (now known as personal data assistants or PDAs).

To serve these goals, Oak needed the following features:

- platform independence, since it must run on devices from multiple manufacturers
- extreme reliability (can't expect consumers to reboot their VCRs!)
- compactness, since embedded processors typically have limited memory

They also wanted a next-generation language that built on the strengths and avoided the weaknesses of earlier languages. Such features would help the new language provide more rapid software development and faster debugging.

By 1993 the interactive TV and PDA markets had failed to take off, but internet and web activity began its upward zoom. So Sun shifted the target market to internet applications and changed the name of the project to Java.

The portability of Java made it ideal for the Web, and in 1995 Sun's *HotJava* browser appeared. Written in Java in only a few months, it illustrated the power of applets – programs that run within a browser – and the ability of Java to accelerate program development.

Riding atop the tidal wave of interest and publicity in the Internet, Java quickly gained widespread recognition (some would say *hype*), and expectations grew for it to become the dominant software for browsers and perhaps even for desktop programs. However, the early versions of Java did not possess the breadth and depth of capabilities needed for desktop applications. For example, the graphics in Java 1.0 appeared crude and clumsy compared with the graphics features available in software written in C and other languages and targeted at a single operating system.

Applets did in fact become popular and remain a common component of web page design. However, they do not dominate interactive or multimedia displays in the browser as expected. Many other "plug-in" programs also run within the browser environment.

Though Java's capabilities grew enormously with the release of several expanded versions (see Section 1.3), Java has not yet found wide success in desktop client applications. Instead Java gained widespread acceptance at the two opposite ends of the platform spectrum: large business systems and very small systems.

Java is used extensively in business to develop enterprise, or middleware, applications such as on-line stores, transactions processing, dynamic web page generation, and database interactions. Java has also returned to its Oak roots and become very common on small platforms such as smart cards, cell phones, and PDAs. For example, as of mid-2004 there are over 350 different Java-enabled mobile phone handsets available across the world, and over 600 million Java Cards have been distributed

1.3 Versions of Java

Since its introduction, Sun has released new versions of the Java core language with significant enhancements about every two years or so. Until recently, Sun denoted the versions with a 1.x number, where x reached up to 4. (Less drastic releases with bug fixes were indicated with a third number as in 1.4.2.) The next version, however, will be called Java 5.0. Furthermore, Sun has split its development kits into so-called editions, each aimed towards a platform with different capabilities. Here we try to clarify all of this.

1.3.1 Standard Edition

Below is a time line for the different versions of the *Standard Edition* (SE) of Java, which offers the core language libraries (called *packages* in Java) and is aimed at desktop platforms. We include a sampling of the new features that came with each release.

- **1995** – Version 1.0. The Java Development Kit (JDK) included:
  - 8 packages with 212 classes.
  - Netscape 2.0–4.0 included Java 1.0.
  - Microsoft and other companies licensed Java.
- **1997** – Version 1.1:
  - 23 packages, 504 classes.
  - Improvements included better event handling, inner classes, improved VM.

  - Microsoft developed its own 1.1-compatible Java Virtual Machine for the Internet Explorer.
  - Many browsers in use are still compatible only with 1.1.
  - Swing packages with greatly improved graphics became available during this time but were not included with the core language.
- **1999** – Version 1.2. Sun began referring to the 1.2 and above versions as the Java 2 Platform. The Software Development Kit (SDK) included:
  - 59 packages, 1520 classes.
  - Java Foundation Classes (JFC), based on Swing, for improved graphics and user interfaces, now included with the core language.
  - Collections Framework API included support for various lists, sets, and hash maps.
- **2000** – Version 1.3:
  - 76 packages, 1842 classes.
  - Performance enhancements including the Hotspot virtual machine.
- **2002** – Version 1.4:
  - 135 packages, 2991 classes.
  - Improved IO, XML support, etc.
- **2004** – Version 5.0 (previously known as 1.5). This version was available only in beta release as this book went to press. See Section 1.9 for an overview of what is one of the most extensive updates of Java since version 1.0. It includes a number of tools and additions to the language to enhance program development, such as:
  - faster startup and smaller memory footprint
  - metadata
  - formatted output
  - generics
  - improved multithreading features
  - 165 packages, over 3000 classes

During the early years, versions for Windows platforms and Solaris (Sun's version of Unix) typically became available before Linux and the Apple Macintosh. Over the last few years, Sun has fully supported Linux and has released Linux, Solaris, and Windows versions of Java simultaneously. Apple Computer releases its own version for the Mac OS X operating system, typically a few months after the official Sun release of a new version. In addition, in the year 1999, Sun split off two separate editions of Java 2 software under the general categories of *Micro* and *Enterprise* editions, which we discuss next.

1.3.2 Other editions

Embedded processor systems, such as cell phones and PDAs, typically offer very limited resources as compared to desktop PCs. This means small amounts of RAM and very little disk space or non-volatile memory. It also usually means

small, low-resolution displays, if any at all. So Sun offers slimmed-down versions of Java for such applications. Until recently this involved three separate bundles of Java 1.1-based packages, organized according to the size of the platform. *Java Card* is intended for extremely limited Java for systems with only 16 KB non-volatile memory and 512 bytes volatile. The *EmbeddedJava* and *PersonalJava* bundles are intended for systems with memory resources in the 512 KB and 2 MB ranges, respectively.

To provide a more unified structure to programming for small platforms, Sun has replaced EmbeddedJava and PersonalJava (but not JavaCard) with the Java 2 Micro Edition (J2ME). The developer chooses from different subsets of the packages to suit the capacity of a given system. (We briefly review J2ME in Chapter 24.)

At the other extreme are high-end platforms, often involving multiple processors, that carry out large-scale computing tasks such as online stores, interactions with massive databases, etc. With the Java 2 Platform came a separate set of libraries called the Java 2 *Enterprise Edition* (J2EE) with enhanced resources targeted at these types of applications. Built around the same core as Standard Edition packages, it provides an additional array of tools for building these so-called middleware products.

1.3.3 Naming conventions

We note that the naming and version numbering scheme in Java can be rather confusing. As we see from the time line above, the original Java release included the Java Development Kit (JDK) and was referred to as either Java 1.0 or JDK 1.0. Then came JDK 1.1 with a number of significant changes. The name Java 2 first appeared with what would have been JDK 1.2. At that time the JDK moniker was dropped in favor of SDK (for Software Development Kit). Thus the official name of the first Java 2 development kit was something like Java 2 Platform Standard Edition (J2SE) SDK Version 1.2. Versions 1.3 and 1.4 continued the same naming/numbering scheme.

Meanwhile many people continue to use the JDK terminology – thus JDK 1.4 when referring to J2SE SDK Version 1.4. Another common usage is the simpler but less specific Java Version 1.x, or even just Java 1.x to mean J2SE Version 1.x. Both of these usages are imprecise because there is also a Java 2 Enterprise Edition (J2EE) Version 1.4. To make it clear what you mean, you should probably either use J2SE or J2EE rather than just Java when mentioning a version number unless the meaning is clear from context. This book is not about J2EE, though we do touch on Java Servlet technology in Chapters 14 and 21 and on web services in general in Chapter 21. Since we never need to refer to the Enterprise Edition, we use the terms Java 1.x, SDK 1.x, and J2SE 1.x interchangeably.

By the time this book is in your hands, Sun will have released the Java 2 Standard Edition Version 5.0. The version number 5.0 replaces what would have been version number 1.5. Undoubtedly many people will continue to use the 1.5 terminology. In fact, the Beta 2 release of J2SE 5.0 (the latest available at the time of this writing) continues to use the value 1.5 in some places. You may also come across the code name Tiger for the 5.0 release; however, we expect that usage to fade away just like previous code names Kestrel and Merlin have all but disappeared from the scene. This book uses the notation J2SE 5.0 or Release 5.0 or Version 5.0 or Java 5.0 or sometimes just 5.0 when referring to this very significant new release of Java.

We provide a brief overview of Java 5.0 in Section 1.9 and examine a number of 5.0 topics in some detail throughout the book.

1.4 Java – open or closed?

Java is not a true open language but not quite a proprietary one either. All the core language components – compiler, virtual machines, core language class packages, and many other tools – are free from Sun. Furthermore, Sun makes detailed specifications and source code openly available for the core language. Another company can legally create a so-called *clean room* compiler and/or a Java Virtual Machine as long as it follows the detailed publicly available specifications and agrees to the trademark and licensing terms. Microsoft, for example, created its own version 1.1 JVM for the Internet Explorer browser. See the Web Course for a listing of other independent Java compilers and virtual machines.

Sun, other companies, and independent programmers participate in the Java Community Process (JCP) organization whose charter is "to develop and revise Java technology specifications, reference implementations, and technology compatibility kits." Proposals for new APIs, classes, and other changes to the language now follow a formal process in the JCP to achieve acceptance.

Sun, however, does assert final say on the specifications and maintains various restrictions and trademarks (such as the *Java* name). For example, Microsoft's JVM differed in some significant details from the specifications and Sun filed a lawsuit (later settled out of court) that claimed Microsoft attempted to weaken Java's "Write Once, Run Anywhere" capabilities.

1.5 Java features and benefits

Before we examine how Java can benefit technical applications, we look at the features that make Java a powerful and popular general programming language. These features include:

- **Compiler/interpreter combination**
  - Code is compiled to bytecode, which is interpreted by a Java Virtual Machine (JVM).

  - This provides portability to any base operating system platform for which a virtual machine has been written.
  - The two-step procedure of compilation and interpretation allows for extensive code checking and improved security.
- **Object-oriented**
  - Object-oriented programming (OOP) throughout – no coding outside of class definitions.
  - The bytecode retains an object structure.
  - An extensive class library available in the core language packages.
- **Automatic memory management**
  - A *garbage collector* in the JVM takes care of allocating and reclaiming memory.
- **Several drawbacks of C and C++ eliminated**
  - No accessible memory pointers.
  - No preprocessor.
  - Array limits automatically checked.
- **Robust**
  - Exception handling built-in, strong type checking (that is, all variables must be assigned an explicit data type), local variables must be initialized.
- **Platform independence**
  - The bytecode runs on any platform with a compatible JVM.
  - The "Write Once Run Anywhere" ideal has not been achieved (tuning for different platforms usually required), but is closer than with other languages.
- **Security**
  - The elimination of direct memory pointers and automatic array limit checking prevents rogue programs from reaching into sections of memory where they shouldn't.
  - Untrusted programs are restricted to run inside the virtual machine sandbox. Access to the platform can be strictly controlled by a Security Manager.
  - Code is checked for pathologies by a class loader and a bytecode verifier.
  - Core language includes many security related tools, classes, etc.
- **Dynamic binding**
  - Classes, methods, and variables are linked at runtime.
- **Good performance**
  - Interpretation of bytecodes slowed performance in early versions, but advanced virtual machines with adaptive and just-in-time compilation and other techniques now typically provide performance up to 50% to 100% the speed of C++ programs.
- **Threading**
  - Lightweight processes, called threads, can easily be spun off to perform multiprocessing.
- **Built-in networking**
  - Java was designed with networking in mind. The core language packages come with many classes to program Internet communications.
  - The Enterprise Edition provides an extensive set of tools for building middleware systems for advanced network applications.

These features provide a number of benefits compared to program development in other languages. For example, C/C++ programs are beset by bugs resulting from direct memory pointers, which are eliminated in Java. Similarly, the array limit checking prevents another common source of bugs. The garbage collector relieves the programmer of the big job of memory management. It's often said that these features lead to a significant speedup in program development and debugging compared to working with C/C++.

1.5.1 Java features and benefits for technical programming

The above features benefit all types of programming. For science and engineering specifically, Java provides a number of advantages:

- **Platform independence** – Engineers and scientists, particularly experimentalists, probably use more types of computers and operating systems that any other group. Code that can run on different machines without rewrites and recompilation saves time and effort.
- **Object-oriented** – Besides the usual benefits from OOP, scientific programming can often benefit from thinking in terms of objects. For example, atomic particles in a scattering simulation are naturally self-contained objects.
- **Threading** – Multiprocessing is very useful for many scientific tasks such as simulations of phenomena where many processes occur simultaneously. This can be quite useful in the conceptual design of a program even when it will run on a single-processor machine. However, Java Virtual Machines on multiprocessor platforms also can distribute threads to the different processors to obtain true parallel performance.
- **Simulation tools** – The extensive graphics resources and multithreading in the core Java language provide for depicting and animating engineering and scientific devices and phenomena.
- **Networking** – Java comes with many networking capabilities that allow one to build distributed systems. Such capabilities can be applied, for example, to data collection from remote sensors.
- **Interfacing and enhancing legacy code** – Java's strong graphics and networking capabilities can be applied to existing C and Fortran programs. A Java graphical user interface (GUI) can bring enhanced ease of use to a Fortran or C program, which then acts as a computational engine behind the GUI.

1.5.2 Java shortcomings for technical programming

Several features of Java that make it a powerful and highly secure language, such as array limit checking and the absence of direct memory pointers, can also slow it down, especially for large-scale intensive mathematical calculations. Furthermore, the interpretation of bytecode that makes Java programs so easily portable can cause a big reduction in performance as compared to running a

program in local machine code. This was a particular problem in the early days of Java, and its slow performance led many who experimented with Java to drop it.

Fortunately, more sophisticated JVMs have greatly improved Java performance. So called *Just-in-Time* compilers, for example, convert bytecode to local machine code on the fly. This is especially effective for repetitive sections of code. During the first pass through a loop the code is interpreted and converted to native code so that in subsequent passes the code will run at full native speeds.

Another approach involves adaptive interpretation, such as in Sun's Hotspot, in which the JVM dynamically determines where performance in a program is slow and then optimizes that section in local machine code. Such an approach can actually lead to faster performance than C/C++ programs in some cases.

Here are some other problems in applying Java to technical programming:

- **No rectangular arrays** – Java 2D arrays are actually 1D arrays of references to other 1D arrays. For example, a 4×4 sized 2D array in Java behaves internally like a single 1D array whose elements point to four other 1D arrays:

```
A[0] ==> B[0] B[1] B[2] B[3] B[4]
A[1] ==> C[0] C[1] C[2] C[3] C[4]
A[2] ==> D[0] D[1] D[2] D[3] D[4]
A[3] ==> E[0] E[1] E[2] E[3] E[4]
```

  The B, C, D, and E arrays could be in widely separated locations in memory. This differs from Fortran or C in which 2D array elements are contiguous in memory as in this 4×4 array:

```
A(0,0) A(0,1) A(0,2) A(0,3) A(0,4)
A(1,0) A(1,1) A(1,2) A(1,3) A(1,4)
A(2,0) A(2,1) A(2,2) A(2,3) A(2,4)
A(3,0) A(3,1) A(3,2) A(3,3) A(3,4)
```

  Therefore, with the Java arrays, moving from one element to the next requires extra memory operations as compared to simply incrementing a pointer as in C/C++. This slows the processing when the calculations require multiple operations on large arrays.
- **No complex primitive type** – Numerical and scientific calculations often require imaginary numbers but Java does not include a complex primitive data type (we discuss the primitive data types in Chapter 2). You can easily create a complex number class, but the processing is slower than if a primitive type had been available.
- **No operator overloading** – For coding of mathematical equations and algorithms it would be very convenient to have the option of redefining (or "overloading") the definition of operators such as "+" and "−". In C++, for example, the addition of two complex number objects could use `c1 + c2`, where you define the + operator to properly add such objects. In Java, you must use method invocations instead. This lengthens the code