Software

You probably think you already know what software is. It's that CD a friend gives you with the latest game to run on your computer. Were life only so simple (and copyright laws so liberal) that you could live happily ever after with such beliefs.

Life, alas, is not so simple, and neither is software. The disc isn't software. Software is nothing you can hold in your hand. Rather, it's the hardware that stores the software. The real software is made from the same stuff as dreams, as evanescent as an inspiration, as elusive as the meaning of the current copyright laws. Software is nothing but a set of ideas, ideas that (one hopes) express a way to do something. Those ideas are written in a code the same way our words and sentences code our thoughts. The codes take the form of symbols, which may appear on paper or be represented as pulses in electrical circuits. No matter. The code is only the representation of the ideas, and the ideas are really the software.

So much for the philosophy of software. Not many people think about software that way. Most approach it more practically. Software is something your computer needs. You might say your computer needs software to run its programs, but that's like saying you need food to eat dinner. The food is dinner. The software is the programs.

Ouch. No matter how bad that sounds, that's the way most people think. As a practical matter, they're getting close to the truth, but software is more than programs even at a practical level. Software is more than the box you buy and little silver disc that comes inside it. Software is not a singular thing. It is an entire technology that embraces not only what
you see when you run your computer but also a multitude of invisible happenings hiding beneath the surface. A modern computer runs several software programs simultaneously, even when you think you’re using just one, or even when you don’t think anything is running at all. These programs operate at different levels, each one taking care of its own specific job, invisibly linking to the others to give you the illusion you’re working with a single, smooth-running machine.

What shows on your monitor is only the most obvious of all this software—the applications, the programs such as Microsoft Office and Grand Theft Auto 3 that you actually buy and load onto your computer, the ones that boldly emblazon their names on your screen every time you launch them. But there’s more. Related to applications are the utilities you use to keep your computer in top running order, protect yourself from disasters, and automate repetitive chores. Down deeper is the operating system, which links your applications and utilities together to the actual hardware of your computer. At or below the operating system level, you use programming languages to tell your computer what to do. You can write applications, utilities, or even your own operating system with the right programming language.

**Definitions**

Software earns its name for what it is not. It is not hardware. Whatever is not hard is soft, and thus the derivation of the name. Hardware came first—simply because the hard reality of machines and tools existed long before anyone thought of computers or of the concept of programming. Hardware happily resides on page 551 of the 1965 dictionary I keep in my office—you know, nuts, bolts, and stuff like that. Software is nowhere to be found. (Why I keep a 1965 dictionary in my office is another matter entirely.)

Software comprises abstract ideas. In computers, the term embraces not only the application programs you buy and the other kinds of software listed earlier but also the information or data used by those programs.

Programs are the more useful parts of software because programs do the actual work. The program tells your computer what to do—how to act, how to use the data it has, how to react to your commands, how to be a computer at all. Data just slow things down.

Although important, a program is actually a simple thing. Broken down to its constituent parts, a computer program is nothing but a list of commands to tell your hardware what to do. Like a recipe, the program is a step-by-step procedure that, to the uninitiated, seems to be written in a secret code. In fact, the program is written in a code called the programming language, and the resultant list of instructions is usually called code by its programmers.
Applications

The programs you buy in the box off your dealer’s shelves, in person or through the Web, the ones you run to do actual work on your computer, are its applications. The word is actually short for application software. These are programs with a purpose, programs you apply to get something done. They are the dominant beasts of computing, the top of the food chain, the colorful boxes in the store, the software you actually pay for. Everything else in your computer system, hardware and software alike, exists merely to make your applications work. Your applications determine what you need in your computer simply because they won’t run—or run well—if you don’t supply them with what they want.

Today’s typical application comes on one or more CDs and is comprised of megabytes, possibly hundreds of megabytes, of digital stuff that you dutifully copy to your hard disk during the installation process. Hidden inside these megabytes is the actual function of the program, the part of the code that does what you buy the software for—be it to translate keystrokes into documents, calculate your bank balance, brighten your photographs, or turn MP3 files into music. The part of the program that actually works on the data you want to process is called an algorithm, the mathematical formula of the task converted into program code. An algorithm is just a way for doing something written down as instructions so you can do it again.

The hard-core computing work performed by major applications—the work of the algorithms inside them—is typically both simple and repetitive. For example, a tough statistical analysis may involve only a few lines of calculations, although the simple calculations will often be repeated again and again. Changing the color of a photo is no more than a simple algorithm executed over and over for each dot in the image.

That’s why computers exist at all. They are simply good at repeatedly carrying out the simple mathematical operations of the algorithms without complaining.

If you were to tear apart a program to see how it works—what computer scientists call dis-assemblying the program—you’d make a shocking discovery. The algorithm makes up little of the code of a program. Most of the multimegabyte bulk you buy is meant to hide the algorithm from you, like the fillers and flavoring added to some potent but noxious medicine.

Before the days of graphical operating systems, as exemplified by Microsoft’s ubiquitous Windows family, the bulk of the code of most software applications was devoted to making the rigorous requirements of the computer hardware more palatable to your human ideas, aspirations, and whims. The part of the software that serves as the bridge between your human understanding and the computer’s needs is called the user interface. It can be anything from a typewritten question mark that demands you type some response to a multicolor graphic menu luring your mouse to point and click.
Windows simplifies the programmers’ task by providing most of the user interface functions from your applications. Now, most of the bulk of an application is devoted to mating not with your computer but with Windows. The effect is the same. It just takes more megabytes to get there.

No matter whether your application must build its own user interface or rely on the one provided by Windows, the most important job of most modern software is simply translation. The program converts your commands, instructions, and desires into a form digestible by your operating system and computer. In particular, the user interface translates the words you type and the motion of your arm pointing your mouse into computer code.

This translation function, like the Windows user interface, is consistent across most applications. All programs work with the same kind of human input and produce the same kind of computer codes. The big differences between modern applications are the algorithms central to the tasks to be carried out. Application software often is divided into several broad classes based on these tasks. Table 3.1 lists the traditional division of functions or major classifications of computer application software.

<table>
<thead>
<tr>
<th>Class of Software</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web browsers</td>
<td>Entering and interacting with Web sites and servers to buy, sell, research your favorite interests, and just while away evenings, afternoons, and the rest of your life</td>
</tr>
<tr>
<td>E-mail programs</td>
<td>Sending and receiving instant messages from friends, associates, and solicitors in your neighborhood and around the world</td>
</tr>
<tr>
<td>Word processors</td>
<td>Getting your words right for print or electronic publishing</td>
</tr>
<tr>
<td>Spreadsheets</td>
<td>The accountant’s ledger made automatic to calculate arrays of numbers</td>
</tr>
<tr>
<td>Databases</td>
<td>A filing system with instant access and the ability to automatically sort itself</td>
</tr>
<tr>
<td>Drawing and painting programs</td>
<td>Creating and editing images such as blueprints and cartoon cells that can be filed and edited with electronic ease</td>
</tr>
<tr>
<td>Multimedia software</td>
<td>Playing MP3 and WAV files for music, showing AVI or MOV video files, playing games, or displaying images and sound like a movie theatre under the control of an absolute dictator (you)</td>
</tr>
</tbody>
</table>
The lines between many of these applications are blurry. For example, many people find that spreadsheets serve all their database needs, and most spreadsheets now incorporate their own graphics for charting results.

**Suites**

Several software publishers completely confound the distinctions by combining most of these application functions into a single package that includes database, graphics, spreadsheet, and word processing functionalities. These combinations are termed *application suites*. Ideally, they offer several advantages. Because many functions (and particularly the user interface) are shared between applications, large portions of code need not be duplicated, as would be the case with standalone applications. Because the programs work together, they better know and understand one another’s resource requirements, which means you should encounter fewer conflicts and memory shortfalls. Because they are all packaged together, you stand to get a better price from the publisher.

Today there’s another name for the application suite: *Microsoft Office*. Although at one time at least three major suites competed for space on your desktop, the other offerings have, for the most part, faded away. You can still buy Corel WordPerfect Office or Lotus SmartSuite, although almost no one does. (In fact, most copies of Office are sold to businesses. Most individuals get it bundled along with a new computer.) Office has become popular because it offers a single-box solution that fills the needs of most people, handling more tasks with more depth than they ordinarily need. The current version, Office XP, dominates the market because Microsoft sells it to computer makers at a favorable price (and, some say, with more than a little coercion).

**Utilities**

Even when you’re working toward a specific goal, you often have to make some side trips. Although they seem unrelated to where you’re going, they are as much a necessary part of the journey as any other. You may run a billion-dollar pickle-packing empire from your office, but you might never get your business negotiations done were it not for the regular housekeeping that keeps the place clean enough for visiting dignitaries to walk around without slipping on pickle juice on the floor.

The situation is the same with software. Although you need applications to get your work done, you need to take care of basic housekeeping functions to keep your system running in top condition and working most efficiently. The programs that handle the necessary auxiliary functions are called *utility software*.

From the name alone you know that utilities do something useful, which in itself sets them apart from much of the software on today’s market. Of course, the usefulness of any tool depends on the job you have to do—a pastry chef has little need for the hammer that...
so well serves the carpenter or computer technician—and most utilities are crafted for some similar, specific need. For example, common computer utilities keep your disk organized and running at top speed, prevent disasters by detecting disk problems and viruses, and save your sanity should you accidentally erase a file.

The most important of these functions are included with today’s computer operating systems, either integrated into the operating system itself or as individual programs that are part of the operating system package. Others you buy separately, at least until Microsoft buys out the company that offers them.

Common utilities include backup, disk defragmenting, font management, file compression, and scheduling—all of which were once individual programs from different publishers but now come packaged in Windows. Antivirus and version-tracking programs are utilities available from separately from Windows.

Modern utilities are essentially individual programs that load like ordinary applications when you call on them. The only difference between them and other applications is what they do. Utilities are meant to maintain your system rather than come up with answers or generate output.

**Applets**

An *applet* is a small software application that’s usually dedicated to a single simple purpose. It may function as a standalone program you launch like an ordinary application, or it may run from within another application. Typically the applet performs a housekeeping function much like a utility, but the function of an applet is devoted to supporting an overriding application rather than your computer in general. That said, some system utilities may take the form of applets, too. Applets are mostly distinguished from other software by their size and the scope of their functions.

When you download a page from the Web that does something exciting on the screen (which means it shows you something other than text that just lays there), it’s likely you’ve invisibly downloaded an applet from the Internet. Applets let you or distant programmers add animation and screen effects to Web pages.

Other applets are often included with or as part of application software packages (for example, the “wizard” that leads you through the installation process). Some are included with operating systems such as Windows. In fact, the chief distinction between an applet and a full application may be little more than the fact that you don’t buy applets separately and never have.
Operating Systems

The basic level of software with which you will work on your computer is the operating system. It's what you see when you don’t have an application or utility program running. But an operating system is much more than what you see on the screen.

As the name implies, the operating system tells your computer how to operate, how to carry on its most basic functions. Early operating systems were designed simply to control how you read from and wrote to files on disks and were hence termed disk operating systems (which is why the original computer operating system was called DOS). Today’s operating systems add a wealth of functions for controlling every possible computer peripheral from keyboard (and mouse) to monitor screen.

The operating system in today’s computers has evolved from simply providing a means of controlling disk storage into a complex web of interacting programs that perform several functions. The most important of these is linking the various elements of your computer system together. These linked elements include your computer hardware, your programs, and you. In computer language, the operating system is said to provide a common hardware interface, a common programming interface, and a common user interface.

An interface, by the way, is the point where two things connect together—for example, the human interface is where you, the human being, interact with your computer. The hardware interface is where your computer hardware links to its software. The programming interface is where programs link to the operating system. And the user interface is where you, as the user, link to the operating system. Interfaces can combine and blend together. For example, the user interface of your operating system is part of the human interface of your computer.

Of the operating system’s many interfaces only one, the user interface, is visible to you. The user interface is the place where you interact with your computer at its most basic level. Sometimes this part of the operating system is called the user shell. In today’s operating systems, the shell is simply another program, and you can substitute one shell for another. Although with Windows most people stick with the shell that Microsoft gives them, you don’t have to. People who use Unix or Linux often pick their own favorite shell.

In the way they change the appearance of your operating system, shells are like the skins used by some applications—for example, the skins that give your MP3 player software the look of an old-fashioned Wurlitzer jukebox.

In effect, the shell is a starting point to get your applications running, and it’s the home base that you return to between applications. The shell is the program that paints the desktop on the screen and lets you choose the applications you want to run.
Behind the shell, the Application Program Interface (or API) of the operating system gives programmers a uniform set of calls, key words that instruct the operating system to execute a built-in program routine that carries out some predefined function. For example, the API of Windows enables programmers to link their applications to the operating system to take advantage of its user interface. A program can call a routine from the operating system that draws a menu box on the screen.

Using the API offers programmers the benefit of having the complicated aspects of common program procedures already written and ready to go. Programmers don’t have to waste their time on the minutiae of moving every bit on your monitor screen or other common operations. The use of a common base of code also eliminates duplication, which makes today’s overweight applications a bit more svelte. Moreover, because all applications use basically the same code, they have a consistent look and work in a consistent manner. This prevents your computer from looking like the accidental amalgamation of the late-night work of thousands of slightly aberrant engineers that it is.

At the other side of the API, the operating system links your applications to the underlying computer hardware through the hardware interface. Once we take a look at what that hardware might be, we’ll take a look how the operating system makes the connection in the following section, titled “Hardware Interfaces.”

Outside of the shell of the user interface, you see and directly interact with little of an operating system. The bulk of the operating system program code works invisibly (and continuously). And that’s the way it’s designed to be.

Operating systems must match your hardware to work on a given computer. The software instructions must be able to operate your computer. Consequently, Apple computers and Intel-based computers use different operating systems. Similarly, programs must match the underlying operating system, so you must be sure that any given program you select is compatible with your operating system.

Today, four families of operating system are popular with personal computers.

Windows 9X

The Windows 9X family includes Windows 95, Windows 98, and Windows Millennium Edition (Me). These operating systems are built on a core of the original 16-bit DOS (dating back to 1981) and are meant to run only on Intel microprocessors and chips that are completely compatible with Intel architecture.

Microsoft announced Windows at a news conference in New York on November 10, 1983. But development of the graphic environment (as it was then classified) proved more troublesome than Microsoft anticipated, and the first version of the “16-bit version” of Windows did not reach store shelves as Windows 1.0 until November 20, 1985. The last
major release of 16-bit Windows was Windows Me, which was first put on sale on September 14, 2000.

Because DOS was originally written in a time before running several programs at once was common, Microsoft’s programmers had to add on this ability and provide the power to isolate each program from one another. In fact, they kludged together DOS, graphics, and multitasking control into what most programmers regard as a gnarled mess of code. Little wonder, then, that these 16-bit versions of Windows have a reputation for unreliability and are most suited to home computer users. You wouldn’t want your life to depend on them.

Windows NT

The 32-bit Windows family includes Windows NT, Windows 2000, and Windows XP, in all its versions. When the core of this operating systems was originally conceived as Window NT, Microsoft’s programmers decided to scrap all of the original Windows and start over. Instead of DOS they used OS/2, an operating system jointly developed by Microsoft and IBM, as its core. Microsoft called the first release of the new operating system Windows NT 3.1, cleverly skipping over early version numbers to make the new NT seem like a continuation of the then-current 16-bit Windows version, 3.1. According to Microsoft, Windows NT 3.1 was officially released to manufacturing on July 27, 1993. The same 32-bit core, although extensively refined, is still in use in current versions of Windows XP.

Instead of recycling old 16-bit code, Microsoft’s programmers started off with the 32-bit code native to the 386 microprocessor that was current at the time (and is still used by today’s latest Pentium 4 microprocessors). Although Microsoft attempted to craft versions of Windows NT to run on processors other than those using Intel architecture, no such adaptation has proven successful, and the operating system runs exclusively on Intel-architecture microprocessors.

From the very beginning, Microsoft conceived Windows NT as a multitasking operating system, so its engineers built in features that ensure the isolation of each application it runs. The result was a robust operating system that had but one problem—it lacked complete compatibility with old, DOS-based programs. Over the years, this compatibility became both less necessary (as fewer people ran DOS applications) and better, as Microsoft’s engineers refined their code. Most professionals regard Windows XP as a robust operating system on par with any other, suitable for running critical applications.

Unix

Unix is not a single operating system but several incompatible families that share a common command structure. Originally written at Bell Labs to run on a 16-bit Digital
Equipment Company PDP-7 computer, Unix has been successfully adapted to nearly every microprocessor family and includes 8-, 16-, 32-, and 64-bit versions under various names.

Unix traces its roots to 1969, but according to Dennis M. Ritchie, one of its principal developers (along with Ken Thompson), its development was not announced to the world until 1974. From the beginning, Unix was designed to be a time-sharing, multiuser system for mainframe-style computers. It proved so elegant and robust that it has been adapted to nearly every microprocessor platform, and it runs on everything from desktop computers to huge clusters of servers.

Nearly every major hardware platform has a proprietary version of Unix available to it. Some of these include AIX for IBM computers, Solaris for Sun Microsystems machines, and even OS X, the latest Apple Macintosh operating system. Note that programs written for one version of Unix will likely not run on other hardware than for which it was intended. It has become a sort of universal language among many computer professionals.

**Linux**

Unlike other operating systems, Linux can be traced back to a single individual creator, Linus Torvalds, who wrote its kernel, the core of the operating system that integrates its most basic functions, in 1991 while a student at the University of Helsinki. He sent early versions around for testing and comment as early as September of that year, but a workable version (numbered 0.11) didn’t become available until December. Version 1.0, regarded as the first truly stable (and therefore usable) version of Linux came in March, 1994. Torvalds himself recommends pronouncing the name of the operating system with a short-i sound (say Lynn-ucks).

The most notable aspect of Linux is that it is “open source.” That is, Torvalds has shared the source code—the lines of program code he actually wrote—of his operating system with the world, whereas most software publishers keep their source code secret. This open-source model, as it is called, lets other programmers help refine and develop the operating system. It also results in what is essentially a free, although copyrighted, operating system.

Contrary to popular opinion, Linux is not a variety of Unix. It looks and works like Unix—the commands and structure are the same—but its kernel is entirely different. Programs written for Unix will not run under Linux (although if you have the source code, these programs can be relatively easily converted between the two operating systems). Programmers have adapted Linux to many hardware platforms, including those based on Intel microprocessors.

The open-source status of Linux, along with its reputation for reliability and robustness (inherited without justification from Unix), have made it popular. In other words, it's like
Unix and it’s free, so it has become the choice for many Web servers and small business systems. On the other hand, the strength it gains from its similarity to Unix is also a weakness. It is difficult for most people to install, and it lacks the wide software support of Windows.

Programming Languages

A computer program—whether it’s an applet, utility, application, or operating system—is nothing more than a list of instructions for the brain inside your computer, the microprocessor, to carry out. A microprocessor instruction, in turn, is a specific pattern of bits, a digital code. Your computer sends the list of instructions making up a program to its microprocessor one at a time. Upon receiving each instruction, the microprocessor looks up what function the code says to do, then it carries out the appropriate action.

Microprocessors by themselves only react to patterns of electrical signals. Reduced to its purest form, the computer program is information that finds its final representation as the ever-changing pattern of signals applied to the pins of the microprocessor. That electrical pattern is difficult for most people to think about, so the ideas in the program are traditionly represented in a form more meaningful to human beings. That representation of instructions in human-recognizable form is called a programming language.

As with a human language, a programming language is a set of symbols and the syntax for putting them together. Instead of human words or letters, the symbols of the programming language correspond to patterns of bits that signal a microprocessor exactly as letters of the alphabet represent sounds that you might speak. Of course, with the same back-to-the-real-basics reasoning, an orange is a collection of quarks squatting together with reasonable stability in the center of your fruit bowl.

The metaphor is apt. The primary constituents of an orange—whether you consider them quarks, atoms, or molecules—are essentially interchangeable, even indistinguishable. By itself, every one is meaningless. Only when they are taken together do they make something worthwhile (at least from a human perspective): the orange. The overall pattern, not the individual pieces, is what’s important.

Letters and words work the same way. A box full of vowels wouldn’t mean anything to anyone not engaged in a heated game of Wheel of Fortune. Match the vowels with consonants and arrange them properly, and you might make words of irreplaceable value to humanity: the works of Shakespeare, Einstein’s expression of general relativity, or the formula for Coca-Cola. The meaning is not in the pieces but their patterns.

The same holds true for computer programs. The individual commands are not as important as the pattern they make when they are put together. Only the pattern is truly meaningful.
You make the pattern of a computer program by writing a list of commands for a microprocessor to carry out. At this level, programming is like writing reminder notes for a not-too-bright person—first socks, then shoes.

This step-by-step command system is perfect for control freaks but otherwise is more than most people want to tangle with. Even simple computer operations require dozens of microprocessor operations, so writing complete lists of commands in this form can be more than many programmers—let alone normal human beings—want to deal with. To make life and writing programs more understandable, engineers developed higher-level programming languages.

A higher-level language uses a vocabulary that’s more familiar to people than patterns of bits, often commands that look something like ordinary words. A special program translates each higher-level command into a sequence of bit-patterns that tells the microprocessor what to do.

**Machine Language**

Every microprocessor understands its own repertoire of instructions, just as a dog might understands a few spoken commands. Whereas your pooch might sit down and roll over when you ask it to, your processor can add, subtract, and move bit-patterns around as well as change them. Every family of microprocessor has a set of instructions that it can recognize and carry out: the necessary understanding designed into the internal circuitry of each microprocessor chip.

The entire group of commands that a given model of microprocessor understands and can react to is called that microprocessor's instruction set or its command set. Different microprocessor families recognize different instruction sets, so the commands meant for one chip family would be gibberish to another. For example, the Intel family of microprocessors understands one command set; the IBM/Motorola PowerPC family of chips recognizes an entirely different command set. That’s the basic reason why programs written for the Apple Macintosh (which is based on PowerPC microprocessors) won’t work on computers that use Intel microprocessors.

That native language a microprocessor understands, including the instruction set and the rules for using it, is called machine language. The bit-patterns of electrical signals in machine language can be expressed directly as a series of ones and zeros, such as 0010110. Note that this pattern directly corresponds to a binary (or base-two) number. As with any binary number, the machine language code of an instruction can be translated into other numerical systems as well. Most commonly, machine language instructions are expressed in hexadecimal form (base-16 number system). For example, the 0010110 subtraction instruction becomes 16(hex). (The “(hex)” indicates the number is in hexadecimal notation, that is, base 16.)
Assembly Language

Machine language is great if you're a machine. People, however, don't usually think in terms of bit-patterns or pure numbers. Although some otherwise normal human beings can and do program in machine language, the rigors of dealing with the obscure codes takes more than a little getting used to. After weeks, months, or years of machine language programming, you begin to learn which numbers do what. That's great if you want to dedicate your life to talking to machines, but not so good if you have better things to do with your time.

For human beings, a better representation of machine language codes involves mnemonics rather than strictly numerical codes. Descriptive word fragments can be assigned to each machine language code so that 16(hex) might translate into SUB (for subtraction). Assembly language takes this additional step, enabling programmers to write in more memorable symbols.

Once a program is written in assembly language, it must be converted into the machine language code understood by the microprocessor. A special program, called an assembler, handles the necessary conversion. Most assemblers do even more to make the programmer's life more manageable. For example, they enable blocks of instructions to be linked together into a block called a subroutine, which can later be called into action by using its name instead of repeating the same block of instructions again and again.

Most of assembly language involves directly operating the microprocessor using the mnemonic equivalents of its machine language instructions. Consequently, programmers must be able to think in the same step-by-step manner as the microprocessor. Every action that the microprocessor does must be handled in its lowest terms. Assembly language is consequently known as a low-level language because programmers write at the most basic level.

High-Level Languages

Just as an assembler can convert the mnemonics and subroutines of assembly language into machine language, a computer program can go one step further by translating more human-like instructions into the multiple machine language instructions needed to carry them out. In effect, each language instruction becomes a subroutine in itself.

The breaking of the one-to-one correspondence between language instruction and machine language code puts this kind of programming one level of abstraction farther from the microprocessor. That's the job of the high-level languages. Instead of dealing with each movement of a byte of information, high-level languages enable the programmer to deal with problems as decimal numbers, words, or graphic elements. The language program takes each of these high-level instructions and converts it into a long series of digital code microprocessor commands in machine language.
High-level languages can be classified into two types: interpreted and compiled. Batch languages are a special kind of interpreted language.

**Interpreted Languages**

An interpreted language is translated from human to machine form each time it is run by a program called an **interpreter**. People who need immediate gratification like interpreted programs because they can be run immediately, without intervening steps. If the computer encounters a programming error, it can be fixed, and the program can be tested again immediately. On the other hand, the computer must make its interpretation each time the program is run, performing the same act again and again. This repetition wastes the computer’s time. More importantly, because the computer is doing two things at once—both executing the program and interpreting it at the same time—it runs more slowly.

Today the most important interpreted computer language is **Java**, the tongue of the Web created by Sun Microsystems. Your computer downloads a list of Java commands and converts them into executable form inside your computer. Your computer then runs the Java code to make some obnoxious advertisement dance and flash across your screen.

The interpreted design of Java helps make it universal. The Java code contains instructions that any computer can carry out, regardless of its operating system. The Java interpreter inside your computer converts the universal code into the specific machine language instructions your computer and its operating system understand.

Before Java, the most popular interpreted language was BASIC, an acronym for the Beginner’s All-purpose Symbolic Instruction Set. BASIC was the first language for personal computers and was the foundation upon which the Microsoft Corporation was built.

In classic form, using an interpreted language involved two steps. First, you would start the language interpreter program, which gave you a new environment to work in, complete with its own system of commands and prompts. Once in that environment, you then executed your program, typically starting it with a “Run” instruction. More modern interpreted systems such as Java hide the actual interpreter from you. The Java program appears to run automatically by itself, although in reality the interpreter is hidden in your Internet browser or operating system. Microsoft’s Visual Basic gets its interpreter support from a runtime module, which must be available to your computer’s operating system for Visual Basic programs to run.

**Compiled Languages**

Compiled languages execute like a program written in assembler, but the code is written in a more human-like form. A program written with a compiled language gets translated from high-level symbols into machine language just once. The resultant machine language is then stored and called into action each time you run the program. The act of converting the program from the English-like compiled language into machine language is called
compiling the program. To do this you use a language program called a *compiler*. The original, English-like version of the program, the words and symbols actually written by the programmer, is called the *source code*. The resultant machine language makes up the program's *object code*.

Compiling a complex program can be a long operation, taking minutes, even hours. Once the program is compiled, however, it runs quickly because the computer needs only to run the resultant machine language instructions instead of having to run a program interpreter at the same time. Most of the time, you run a compiled program directly from the DOS prompt or by clicking an icon. The operating system loads and executes the program without further ado. Examples of compiled languages include today’s most popular computer programming language, C++, as well as other tongues left over from earlier days of programming—COBOL, Fortran, and Pascal.

Object-oriented languages are special compiled languages designed so that programmers can write complex programs as separate modules termed *objects*. A programmer writes an object for a specific, common task and gives it a name. To carry out the function assigned to an object, the programmer need only put its name in the program without reiterating all the object’s code. A program may use the same object in many places and at many different times. Moreover, a programmer can put a copy of an object into different programs without the need to rewrite and test the basic code, which speeds up the creation of complex programs. C++ is object oriented.

*Optimizing compilers* do the same thing as ordinary compilers, but do it better. By adding an extra step (or more) to the program compiling process, the optimizing compiler checks to ensure that program instructions are arranged in the most efficient order possible to take advantage of all the capabilities of the computer’s processor. In effect, the optimizing compiler does the work that would otherwise require the concentration of an assembly language programmer.

**Libraries**

Inventing the wheel was difficult and probably took human beings something like a million years—a long time to have your car sitting up on blocks. Reinventing the wheel is easier because you can steal your design from a pattern you already know. But it's far, far easier to simply go out and buy a wheel.

Writing program code for a specific but common task often is equivalent to reinventing your own wheel. You're stuck with stringing together a long list of program commands, just like all the other people writing programs have to do. Your biggest consolation is that you need to do it only once. You can then reuse the same set of instructions the next time you have to write a program that needs the same function.
For really common functions, you don’t have to do that. Rather than reinventing the wheel, you can buy one. Today’s programming languages include collections of common functions called libraries so that you don’t have to bother with reinventing anything. You only need pick the prepackaged code you want from the library and incorporate it into your program. The language compiler links the appropriate libraries to your program so that you only need to refer to a function by a code name. The functions in the library become an extension to the language, called a meta-language.

**Development Environments**
Even when using libraries, you’re still stuck with writing a program in the old-fashioned way—a list of instructions. That’s an effective but tedious way of building a program. The computer’s strength is taking over tedious tasks, so you’d think you could use some of that power to help you write programs more easily. In fact, you might expect someone to write a program to help you write programs.

A *development environment* is exactly that kind of program, one that lets you drag and drop items from menus to build the interfaces for your own applications. The environment includes not only all the routines of the library, but also its own, easy-to-use interface for putting the code in those libraries to work. Most environments let you create programs by interactively choosing the features you want—drop-down menus, dialog boxes, and even entire animated screens—and match them with operations. The environment watches what you do, remembers everything, and then kicks in a *code generator*, which creates the series of programming language commands that achieves the same end. One example of a development environment is Microsoft Visual Studio.

Working with a development environment is a breeze compared to traditional programming. For example, instead of writing all the commands to pop a dialog box on the screen, you click a menu and choose the kind of box you want. After the box obediently pops on the screen, you can choose the elements you want inside it from another menu, using your mouse to drag buttons and labels around inside the box. When you’re happy with the results, the program grinds out the code. In a few minutes you can accomplish what it might have taken you days to write by hand.

**Batch Languages**
A *batch language* allows you to submit a program directly to your operating system for execution. That is, the batch language is a set of operating system commands that your computer executes sequentially as a program. The resultant batch program works like an interpreted language in that each step gets evaluated and executed only as it appears in the program.

Applications often include their own batch languages. These, too, are merely lists of commands for the application to carry out in the order you’ve listed them to perform some
common, everyday function. Communications programs use this type of programming to automatically log in to the service of your choice and even retrieve files. Databases use their own sort of programming to automatically generate reports that you regularly need. The process of transcribing your list of commands is usually termed scripting. The commands that you can put in your program scripts are sometimes called the scripting language.

Scripting actually is programming. The only difference is the language. Because you use commands that are second nature to you (at least after you’ve learned to use the program) and follow the syntax that you’ve already learned running the program, the process seems more natural than writing in a programming language. That means if you’ve ever written a script to log on to the Internet or have modified an existing script, you’re a programmer already. Give yourself a gold star.

Hardware Interfaces

Linking hardware to software has always been one of the biggest challenges facing those charged with designing computer systems—not just individual computers but whole families of computers, those able to run the same programs. The solution has almost invariably been to wed the two together by layering on software, so much so that the fastest processors struggle when confronted by relatively simple tasks. An almost unbelievable amount of computer power gets devoted to moving bytes from software to hardware through a maze of program code, all in the name of making the widest variety of hardware imaginable work with a universal (at least if Microsoft has its way) operating system.

The underlying problem is the same as in any mating ritual. Software is from Venus, and hardware is from Mars (or, to ruin the allusion for sake of accuracy, Vulcan). Software is the programmer's labor of love, an ephemeral spirit that can only be represented. Hardware is the physical reality, the stuff pounded out in Vulcan's forge—enduring, unchanging, and often priced like gold. (And, yes, for all you Trekkers out there, it is always logical.) Bringing the two together is a challenge that even self-help books would find hard to manage. Yet every computer not only faces that formidable task but also tackles it with aplomb, although maybe not as fast as you’d like.

Here’s the challenge: In the basic computer, every instruction in a program gets targeted on the microprocessor. Consequently, the instructions can control only the microprocessor and don’t themselves reach beyond. The circuitry of the rest of the computer and the peripherals connected to it all must get their commands and data relayed to them by the microprocessor. Somehow the microprocessor must be able to send signals to these devices. Today the pathway of command is rarely direct. Rather, the chain of command is a hierarchy, one that often takes on aspects of a bureaucracy.
Control Hierarchy

Perhaps the best way to get to know how software controls hardware is to look at how your system executes a simple command. Let’s start with a common situation.

Because your patience is so sorely tested by reading the electronic version of this book using your Web browser, Internet Scapegoat, you decide to quit and go on to do something really useful, such as playing FreeCell. Your hand wraps around your mouse, you scoot it up to the big black × at the upper-right corner of the screen, and you click your left mouse button. Your hardware has made a link with software.

Your click is a physical act, one that closes the contacts of a switch inside the mouse, squirting a brief pulse of electricity through its circuitry. The mouse hardware reacts by sending a message out the mouse wire to your computer. The mouse port of your computer detects the message and warns your computer by sending a special attention signal called a hardware interrupt squarely at your microprocessor.

At the same time, the mouse driver has been counting pulses sent out by your mouse that indicate its motion. The mouse driver counts each pulse and puts the results into the memory of your computer. It uses these values to find where to put the mouse pointer on the screen.

The interrupt causes your computer to run a software interrupt routine contained in the mouse driver software. The driver, in turn, signals to Windows that you’ve clicked the button. Windows checks in memory for the value the mouse driver has stored there for the position of the mouse pointer. This value tells Windows where the mouse pointer is to determine how to react to your button press. When it discovers you’ve targeted the ×, Windows sends a message to the program associated with it—in this case, your browser.

The browsers reacts, muttering to itself the digital equivalent of “He must be insane,” and it immediately decides to pop up a dialog box that asks whether you really, really want to quit such a quality program as the browser. The dialog box routine is part of the program itself, but it builds the box and its contents from graphics subroutines that are part of the Windows operating system. The browser activates the subroutines through the Windows application interface.

Windows itself does not draw the box. Rather it acts as a translator, converting the box request into a series of commands to draw it. It then sends these commands to its graphics driver. The driver determines what commands to use so that the video board will understand what to do. The driver then passes those commands to another driver, the one associated with the video board.

The video board’s driver routes the command to the proper hardware ports through which the video board accepts instructions. The driver sends a series of commands that
causes the processor on the video board (a graphics accelerator) to compute where it must change pixels to make the lines constituting the box. Once the graphic accelerator finishes the computations, it changes the bytes that correspond to the areas on the screen where the line will appear in a special memory area on the video board called the frame buffer.

Another part of the video board, the rasterizer, scans through the frame buffer and sends the data it finds there to the port leading to your monitor, converting it into a serial data stream for the journey. Using synchronizing signals sent from the video board as a guide, the monitor takes the data from the data stream and illuminates the proper pixels on the screen to form the pattern of the dialog box.

When you awake from your boredom-inspired daze, you finally see the warning box pop up on the screen and react.

The journey is tortuous, but when all goes right, it takes a smaller fraction of a second than it does for you to become aware of what you’ve done. And when all doesn’t go right, another tortuous chain of events will likely result, often one involving flying computers, picture windows, and shards of glass on the front lawn.

Keys to making this chain of command function are the application program interface, the driver software, and the device interfaces of the hardware itself. Sometimes a special part of your computer, the BIOS, also gets involved. Let’s take a deeper look at each of these links.

**Application Program Interface**

A quick recap: An interface is where two distinct entities come together. The most important of the software interfaces in the Windows environment is the application program interface or API.

Rather than a physical thing, the API is a standard set of rules for exchanging commands and data. The Windows API comprises a set of word-like commands termed program calls. Each of these causes Windows to take a particular action. For example, the command DrawBox could tell Windows to draw a box on the screen, as in the preceding example.

To pass along data associated with the command—in the example, how large the box should be and where to put it on the screen—many calls require your program to send along parameters. Each parameter is ordinary data that is strictly formatted to meet the expectation of Windows. That is, the order of the parameters is predefined, and the range permitted for the data is similarly constrained.

Each software interface in the Windows system uses a similar system of calls and parameter passing. Gaining familiarity with the full repertory of the API is one of the biggest challenges facing programmers.
Device Drivers

Device drivers are matchmakers. A device driver takes a set of standardized commands from the operating system and matches them to the capabilities of the device that the driver controls. Typically the device that gets controlled is a piece of hardware, but, as our example shows, one driver may control another driver that controls the hardware.

Just about every class of peripheral has some special function shared with no other device. Printers need to switch ribbon colors; graphics boards need to put dots onscreen at high resolution; sound boards need to blast fortissimo arpeggios; video capture boards must grab frames; and mice have to do whatever mice do. Different manufacturers often have widely different ideas about the best way to handle even the most fundamental functions. No programmer or even collaborative program can ever hope to know all the possibilities. It’s even unlikely that you could fit all the possibilities into an operating system written in code that would fit onto a stack of disks you could carry. There are just too many possibilities.

Drivers make the connection, translating generalized commands made for any hardware into those used by the specific device in your computer. Instead of packing every control or command you might potentially need, the driver implements only those that are appropriate for a specific type, brand, and model of product that you actually connect up to your computer. Without the driver, your operating system could not communicate with your computer.

Device drivers give you a further advantage. You can change them almost as often as you change your mind. If you discover a bug in one driver—say sending an uppercase F to your printer causes it to form-feed through a full ream of paper before coming to a panting stop—you can slide in an updated driver that fixes the problem. You don’t have to replace the device or alter your software. In some cases, new drivers extend the features of your existing peripherals because programmer didn’t have enough time or inspiration to add everything to the initial release.

The way you and your system handles drivers depends on your operating system. Older operating systems (such as DOS and old versions of Windows) load all their drivers when they start and stick with them all while you use your computer. Windows 95 and newer versions treat drivers dynamically, loading them only when they are needed. Not only does this design save memory, because you only need to load the drivers that are actually in use, it also lets you add and remove devices while you’re using your computer. For example, when you plug in a USB scanner, Windows can determine what make and model of scanner you have and then load the driver appropriate for it.

Although today drivers load invisibly and automatically, things were not always so easy. The driver needs to know what system resources your hardware uses for its communications. These resources are the values you set through the Windows Add Hardware
Wizard (click the Resources tab in Device Manager to see them). With old-fashioned drivers, hardware, and operating systems, you had to physically adjust settings on the hardware to assign resources and then configure the driver to match what you configured. It wasn’t pretty, often wasn’t easy, and was the most common reason people could not get hardware to work—usually because they created the problem themselves. Now Windows creates the problems automatically.

**BIOS**

The *Basic Input/Output System*, mentioned earlier as giving a personality to your computer, has many other functions. One of the original design intentions of the BIOS was to help match your computer’s hardware to its software. To that end, the BIOS was meant to act as a special driver software included with your computer so that it can boot up right after you take it out of the box.

Part of the BIOS is program code for drivers that’s permanently or semi-permanently recorded in special memory chips. The code acts like the hardware interface of an operating system but at a lower level—it is a hardware interface that’s independent of the operating system.

Programs or operating systems send commands to the BIOS, and the BIOS sends out the instructions to the hardware using the proper resource values. It lies waiting in your computer, ready for use.

The original goal of the BIOS was to make computer software “platform independent,” meaning that programs don’t care what kind of computer they’re running on. Although that seems a trivial concern in these days of dynamic-loading drivers, it was only a dream decades ago when computers were invented. In those dark-old days, programmers had to write commands aimed specifically at the hardware of the computer. Change the hardware—plug in a different printer—and the software wouldn’t work. The BIOS, like today’s driver software, was meant to wallpaper over the difference in hardware.

The idea, although a good one, didn’t work. Programmers avoided the BIOS in older computers because it added an extra software layer that slowed things down. But the BIOS really fell from favor when Microsoft introduced modern Windows. The old hardware-based BIOS couldn’t keep up with changes in technology. Moreover, using software drivers allowed hardware engineers to use more memory for the needed program code.

The BIOS persists in modern computers as a common means of accessing hardware before the operating system loads. The BIOS code of every computer today still includes the equivalent of driver software to handle accessing floppy disk drives, the keyboard, printers, video, and parallel and serial port operation.
Device Interfaces
Regardless of whether software uses device drivers, looks through the BIOS, or accesses hardware directly, the final link to hardware may be made in one of two ways, set by the hardware design: input/output mapping and memory mapping. **Input/output mapping** relies on sending instructions and data through ports. **Memory mapping** requires passing data through memory addresses. Ports and addresses are similar in concept but different in operation.

**Input/Output Mapping**
A **port** is an address but not a physical location. The port is a logical construct that operates as an addressing system separate from the address bus of the microprocessor, even though it uses the same address lines. If you imagine normal memory addresses as a set of pigeon holes for holding bytes, input/output ports act like a second set of pigeon holes on the other side of the room. To distinguish which set of holes to use, the microprocessor controls a flag signal on its bus called **memory I/O**. In one condition, it tells the rest of the computer that the signals on the address bus indicate a memory location; in its other state, the signals indicate an input/output port.

The microprocessor’s internal mechanism for sending data to a port also differs from memory access. One instruction, **move**, allows the microprocessor to move bytes from any of its registers to any memory location. Some microprocessor operations can even be performed in immediate mode, directly on the values stored at memory locations.

Ports, however, use a pair of instructions: **In** to read from a port, and **Out** to write to a port. The values read can only be transferred into one specific register of the microprocessor (called the **accumulator**) and can only be written from that register. The accumulator has other functions as well. Immediate operations on values held at port locations is impossible, which means a value stored in a port cannot be changed by the microprocessor. It must load the port value into the accumulator, alter it, and then reload the new value back into the port.

**Memory Mapping**
The essence of memory mapping is **sharing**. The microprocessor and the hardware device it controls share access to a specific range of memory addresses. To send data to the device, your microprocessor simply moves the information into the memory locations exactly as if it were storing something for later recall. The hardware device can then read those same locations to obtain the data.

Memory-mapped devices, of course, need direct access to your computer’s memory bus. Through this connection, they can gain speed and operate as fast as the memory system
and its bus connection allow. In addition, the microprocessor can directly manipulate the
data at the memory location used by the connection, thus eliminating the multistep
load/change/reload process required by I/O mapping.

The most familiar memory-mapped device is your computer’s display. Most graphic sys-
tems allow the microprocessor to directly address the frame buffer that holds the image
that appears on your monitor screen. This design allows the video system to operate at
the highest possible speed.

The addresses used for memory mapping must be off limits to the range in which the
operating system loads your programs. If a program should transgress on the area used for
the hardware connection, it can inadvertently change the data there—nearly always with
bad results. Moreover, the addresses used by the interface cannot serve any other function,
so they take away from the maximum memory addressable by a computer. Although such
deductions are insignificant with today’s computers, it was a significant shortcoming for
old systems that were limited to a maximum of 1 to 16 megabytes. Because these short-
comings have diminished over the years, memory mapping has gained popularity in
interfacing.

**Addressing**

To the microprocessor, the difference between ports and memory is one of perception:
Memory is a direct extension of the chip. Ports are the external world. Writing to I/O
ports is consequently more cumbersome and usually requires more time and microproces-
sor cycles.

I/O ports give the microprocessor and computer designer greater flexibility, and they give
you a headache when you want to install multimedia accessories.

Implicit in the concept of addressing, whether memory or port addresses, is proper deliv-
er. You expect a letter carrier to bring your mail to your address and not deliver it to
someone else’s mailbox. Similarly, computers and their software assume that deliveries of
data and instructions will always go where they are supposed to. To ensure proper deliv-
er, addresses must be correct and unambiguous. If someone types a wrong digit on a
mailing label, it will likely get lost in the postal system.

In order to use port or memory addresses properly, your software needs to know the
proper addresses used by your peripherals. Many hardware functions have fixed or stan-
ardized addresses that are the same in every computer. For example, the memory
addresses used by video boards are standardized (at least in basic operating modes), and
the ports used by most hard disk drives are similarly standardized. Programmers can write
the addresses used by this fixed-address hardware into their programs and not worry
whether their data will get where it’s going.
The layered BIOS approach was originally designed to eliminate the need for writing explicit hardware addresses in programs. Drivers accomplish a similar function. They are written with the necessary hardware addresses built in.

**Resource Allocation**
The basic hardware devices got assigned addresses and memory ranges early in the history of the computer and for compatibility reasons have never changed. These fixed values include those of serial and parallel ports, keyboards, disk drives, and the frame buffer that stores the monitor image. Add-in devices and more recent enhancements to the traditional devices require their own assignments of system resources. Unfortunately, beyond the original hardware assignments there are no standards for the rest of the resources. Manufacturers consequently pick values of their own choices for new products. More often than you’d like, several products may use the same address values.

Manufacturers attempt to avoid conflicts by allowing a number of options for the addresses used by their equipment. In days gone by, you had to select among the choices offered by manufacturers using switches or jumpers. Modern expansion products still require resource allocation but use a software-controlled scheme called *Plug-and-Play* to set the values, usually without your intervention or knowledge.

Although a useful computer requires all three—software, hardware, and the glue to hold them together—hardware plays the enabling role. Without it, the computer doesn’t exist. It *is* the computer. The rest of this book will examine the individual components of that hardware as well as related technologies that enable your computer to be the powerful tool it is.