

Chapter 1

Overview: The Mental Landscape

WHEN YOU BEGIN a journey, it's a good idea to have a mental map of the terrain you'll be passing through. The same is true for an intellectual journey, such as learning to write computer programs. In this case, you'll need to know the basics of what computers are and how they work. You'll want to have some idea of what a computer program is and how one is created. Since you will be writing programs in the Java programming language, you'll want to know something about that language in particular and about the modern computing environment for which Java is designed.

As you read this chapter, don't worry if you can't understand everything in detail. (In fact, it would be impossible for you to learn all the details from the brief expositions in this chapter.) Concentrate on learning enough about the big ideas to orient yourself, in preparation for the rest of the book. Most of what is covered in this chapter will be covered in much greater detail later in the book.

1.1 The Fetch and Execute Cycle: Machine Language

A COMPUTER IS A COMPLEX SYSTEM consisting of many different components. But at the heart—or the brain, if you want—of the computer is a single component that does the actual computing. This is the *Central Processing Unit*, or CPU. In a modern desktop computer, the CPU is a single “chip” on the order of one square inch in size. The job of the CPU is to execute programs.

A *program* is simply a list of unambiguous instructions meant to be followed mechanically by a computer. A computer is built to carry out instructions that are written in a very simple type of language called *machine language*. Each type of computer has its own machine language, and the computer can directly execute a program only if the program is expressed in that language. (It can execute programs written in other languages if they are first translated into machine language.)

When the CPU executes a program, that program is stored in the computer's *main memory* (also called the RAM or random access memory). In addition to the program, memory can also hold data that is being used or processed by the program. Main memory consists of a sequence of *locations*. These locations are numbered, and the sequence number of a location is called its *address*. An address provides a way of picking out one particular piece of information from among the millions stored in memory. When the CPU needs to access the program instruction or data in a particular location, it sends the address of that information as a signal to the memory; the memory responds by sending back the value contained in the

specified location. The CPU can also store information in memory by specifying the information to be stored and the address of the location where it is to be stored.

On the level of machine language, the operation of the CPU is fairly straightforward (although it is very complicated in detail). The CPU executes a program that is stored as a sequence of machine language instructions in main memory. It does this by repeatedly reading, or *fetching*, an instruction from memory and then carrying out, or *executing*, that instruction. This process—fetch an instruction, execute it, fetch another instruction, execute it, and so on forever—is called the *fetch-and-execute cycle*. With one exception, which will be covered in the next section, this is all that the CPU ever does. (This is all really somewhat more complicated in modern computers. A typical processing chip these days contains several CPU “cores,” which allows it to execute several instructions simultaneously. And access to main memory is speeded up by memory “caches,” which can be more quickly accessed than main memory and which are meant to hold data and instructions that the CPU is likely to need soon. However, these complications don’t change the basic operation.)

A CPU contains an *Arithmetic Logic Unit*, or ALU, which is the part of the processor that carries out operations such as addition and subtraction. It also holds a small number of *registers*, which are small memory units capable of holding a single number. A typical CPU might have 16 or 32 “general purpose” registers, which hold data values that are immediately accessible for processing, and many machine language instructions refer to these registers. For example, there might be an instruction that takes two numbers from two specified registers, adds those numbers (using the ALU), and stores the result back into a register. And there might be instructions for copying a data value from main memory into a register, or from a register into main memory.

The CPU also includes special purpose registers. The most important of these is the *program counter*, or PC. The CPU uses the PC to keep track of where it is in the program it is executing. The PC simply stores the memory address of the next instruction that the CPU should execute. At the beginning of each fetch-and-execute cycle, the CPU checks the PC to see which instruction it should fetch. During the course of the fetch-and-execute cycle, the number in the PC is updated to indicate the instruction that is to be executed in the next cycle. Usually, but not always, this is just the instruction that sequentially follows the current instruction in the program. Some machine language instructions modify the value that is stored in the PC. This makes it possible for the computer to “jump” from one point in the program to another point, which is essential for implementing the program features known as loops and branches that are discussed in [Section 1.4](#).

* * *

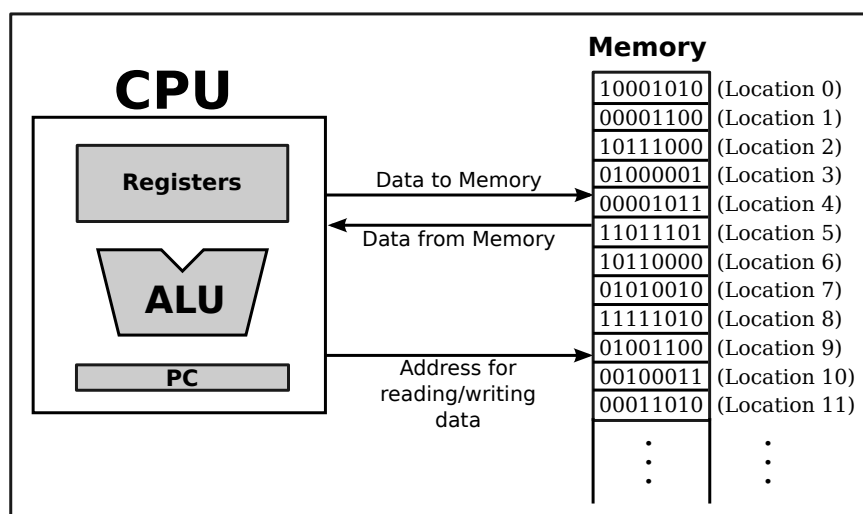
A computer executes machine language programs mechanically—that is without understanding them or thinking about them—simply because of the way it is physically put together. This is not an easy concept. A computer is a machine built of millions of tiny switches called *transistors*, which have the property that they can be wired together in such a way that an output from one switch can turn another switch on or off. As a computer computes, these switches turn each other on or off in a pattern determined both by the way they are wired together and by the program that the computer is executing.

Machine language instructions are expressed as binary numbers. A binary number is made up of just two possible digits, zero and one. Each zero or one is called a *bit*. So, a machine language instruction is just a sequence of zeros and ones. Each particular sequence encodes some particular instruction. The data that the computer manipulates is also encoded as binary numbers. In modern computers, each memory location holds a *byte*, which is a sequence of

eight bits. A machine language instruction or a piece of data generally consists of several bytes, stored in consecutive memory locations. For example, when a CPU reads an instruction from memory, it might actually read four or eight bytes from four or eight memory locations; the memory address of the instruction is the address of the first of those bytes.

A computer can work directly with binary numbers because switches can readily represent such numbers: Turn the switch on to represent a one; turn it off to represent a zero. Machine language instructions are stored in memory as patterns of switches turned on or off. When a machine language instruction is loaded into the CPU, all that happens is that certain switches are turned on or off in the pattern that encodes that instruction. The CPU is built to respond to this pattern by executing the instruction it encodes; it does this simply because of the way all the other switches in the CPU are wired together.

So, you should understand this much about how computers work: Main memory holds machine language programs and data. These are encoded as binary numbers. The CPU fetches machine language instructions from memory one after another and executes them. Each instruction makes the CPU perform some very small task, such as adding two numbers or moving data to or from memory. The CPU does all this mechanically, without thinking about or understanding what it does—and therefore the program it executes must be perfect, complete in all details, and unambiguous because the CPU can do nothing but execute it exactly as written. Here is a schematic view of this first-stage understanding of the computer:



1.2 Asynchronous Events: Polling Loops and Interrupts

THE CPU SPENDS ALMOST ALL of its time fetching instructions from memory and executing them. However, the CPU and main memory are only two out of many components in a real computer system. A complete system contains other devices such as:

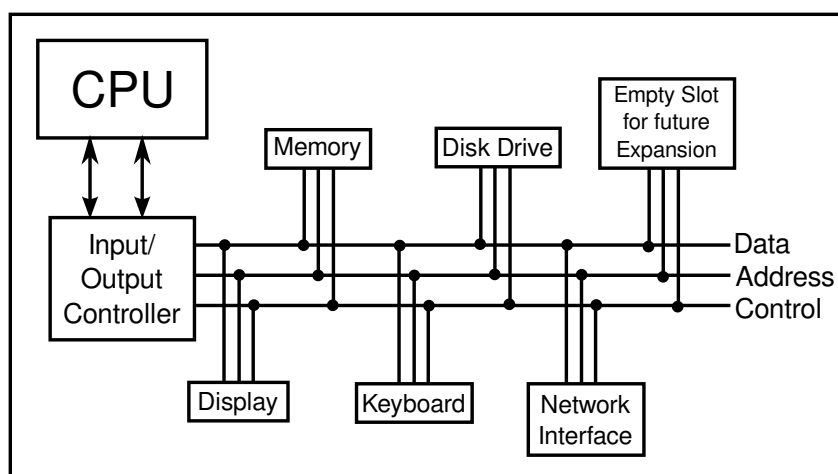
- A *hard disk* or *solid state drive* for storing programs and data files. (Note that main memory holds only a comparatively small amount of information, and holds it only as long as the power is turned on. A hard disk or solid state drive is used for permanent storage of larger amounts of information, but programs have to be loaded from there into main memory before they can actually be executed. A hard disk stores data on a spinning magnetic disk, while a solid state drive is a purely electronic device with no moving parts.)

- A **keyboard** and **mouse** for user input.
- A **monitor** and **printer** which can be used to display the computer's output.
- An **audio output device** that allows the computer to play sounds.
- A **network interface** that allows the computer to communicate with other computers that are connected to it on a network, either wirelessly or by wire.
- A **scanner** that converts images into coded binary numbers that can be stored and manipulated on the computer.

The list of devices is entirely open ended, and computer systems are built so that they can easily be expanded by adding new devices. Somehow the CPU has to communicate with and control all these devices. The CPU can only do this by executing machine language instructions (which is all it can do, period). The way this works is that for each device in a system, there is a **device driver**, which consists of software that the CPU executes when it has to deal with the device. Installing a new device on a system generally has two steps: plugging the device physically into the computer, and installing the device driver software. Without the device driver, the actual physical device would be useless, since the CPU would not be able to communicate with it.

* * *

A computer system consisting of many devices is typically organized by connecting those devices to one or more **busses**. A bus is a set of wires that carry various sorts of information between the devices connected to those wires. The wires carry data, addresses, and control signals. An address directs the data to a particular device and perhaps to a particular register or location within that device. Control signals can be used, for example, by one device to alert another that data is available for it on the data bus. A fairly simple computer system might be organized like this:



Now, devices such as keyboard, mouse, and network interface can produce input that needs to be processed by the CPU. How does the CPU know that the data is there? One simple idea, which turns out to be not very satisfactory, is for the CPU to keep checking for incoming data over and over. Whenever it finds data, it processes it. This method is called **polling**, since the CPU polls the input devices continually to see whether they have any input data to report. Unfortunately, although polling is very simple, it is also very inefficient. The CPU can waste an awful lot of time just waiting for input.

To avoid this inefficiency, *interrupts* are generally used instead of polling. An interrupt is a signal sent by another device to the CPU. The CPU responds to an interrupt signal by putting aside whatever it is doing in order to respond to the interrupt. Once it has handled the interrupt, it returns to what it was doing before the interrupt occurred. For example, when you press a key on your computer keyboard, a keyboard interrupt is sent to the CPU. The CPU responds to this signal by interrupting what it is doing, reading the key that you pressed, processing it, and then returning to the task it was performing before you pressed the key.

Again, you should understand that this is a purely mechanical process: A device signals an interrupt simply by turning on a wire. The CPU is built so that when that wire is turned on, the CPU saves enough information about what it is currently doing so that it can return to the same state later. This information consists of the contents of important internal registers such as the program counter. Then the CPU jumps to some predetermined memory location and begins executing the instructions stored there. Those instructions make up an *interrupt handler* that does the processing necessary to respond to the interrupt. (This interrupt handler is part of the device driver software for the device that signaled the interrupt.) At the end of the interrupt handler is an instruction that tells the CPU to jump back to what it was doing; it does that by restoring its previously saved state.

Interrupts allow the CPU to deal with *asynchronous events*. In the regular fetch-and-execute cycle, things happen in a predetermined order; everything that happens is “synchronized” with everything else. Interrupts make it possible for the CPU to deal efficiently with events that happen “asynchronously,” that is, at unpredictable times.

As another example of how interrupts are used, consider what happens when the CPU needs to access data that is stored on a hard disk. The CPU can access data directly only if it is in main memory. Data on the disk has to be copied into memory before it can be accessed. Unfortunately, on the scale of speed at which the CPU operates, the disk drive is extremely slow. When the CPU needs data from the disk, it sends a signal to the disk drive telling it to locate the data and get it ready. (This signal is sent synchronously, under the control of a regular program.) Then, instead of just waiting the long and unpredictable amount of time that the disk drive will take to do this, the CPU goes on with some other task. When the disk drive has the data ready, it sends an interrupt signal to the CPU. The interrupt handler can then read the requested data.

* * *

Now, you might have noticed that all this only makes sense if the CPU actually has several tasks to perform. If it has nothing better to do, it might as well spend its time polling for input or waiting for disk drive operations to complete. All modern computers use *multitasking* to perform several tasks at once. Some computers can be used by several people at once. Since the CPU is so fast, it can quickly switch its attention from one user to another, devoting a fraction of a second to each user in turn. This application of multitasking is called *timesharing*. But a modern personal computer with just a single user also uses multitasking. For example, the user might be typing a paper while a clock is continuously displaying the time and a file is being downloaded over the network.

Each of the individual tasks that the CPU is working on is called a *thread*. (Or a *process*; there are technical differences between threads and processes, but they are not important here, since it is threads that are used in Java.) Many CPUs can literally execute more than one thread simultaneously—such CPUs contain multiple “cores,” each of which can run a thread—but there is always a limit on the number of threads that can be executed at the same time. Since there are often more threads than can be executed simultaneously, the computer has to be

able switch its attention from one thread to another, just as a timesharing computer switches its attention from one user to another. In general, a thread that is being executed will continue to run until one of several things happens:

- The thread might voluntarily *yield* control, to give other threads a chance to run.
- The thread might have to wait for some asynchronous event to occur. For example, the thread might request some data from the disk drive, or it might wait for the user to press a key. While it is waiting, the thread is said to be *blocked*, and other threads, if any, have a chance to run. When the event occurs, an interrupt will “wake up” the thread so that it can continue running.
- The thread might use up its allotted slice of time and be suspended to allow other threads to run. Most computers can “forcibly” suspend a thread in this way; computers that can do that are said to use *preemptive multitasking*. To do preemptive multitasking, a computer needs a special timer device that generates an interrupt at regular intervals, such as 100 times per second. When a timer interrupt occurs, the CPU has a chance to switch from one thread to another, whether the thread that is currently running likes it or not. All modern desktop and laptop computers, and even typical smartphones and tablets, use preemptive multitasking.

Ordinary users, and indeed ordinary programmers, have no need to deal with interrupts and interrupt handlers. They can concentrate on the different tasks that they want the computer to perform; the details of how the computer manages to get all those tasks done are not important to them. In fact, most users, and many programmers, can ignore threads and multitasking altogether. However, threads have become increasingly important as computers have become more powerful and as they have begun to make more use of multitasking and multiprocessing. In fact, the ability to work with threads is fast becoming an essential job skill for programmers. Fortunately, Java has good support for threads, which are built into the Java programming language as a fundamental programming concept. Programming with threads will be covered in [Chapter 12](#).

Just as important in Java and in modern programming in general is the basic concept of asynchronous events. While programmers don’t actually deal with interrupts directly, they do often find themselves writing *event handlers*, which, like interrupt handlers, are called asynchronously when specific events occur. Such “event-driven programming” has a very different feel from the more traditional straight-through, synchronous programming. We will begin with the more traditional type of programming, which is still used for programming individual tasks, but we will return to threads and events later in the text, starting in [Chapter 6](#)

* * *

By the way, the software that does all the interrupt handling, handles communication with the user and with hardware devices, and controls which thread is allowed to run is called the *operating system*. The operating system is the basic, essential software without which a computer would not be able to function. Other programs, such as word processors and Web browsers, are dependent upon the operating system. Common desktop operating systems include Linux, various versions of Windows, and MacOS. Operating systems for smartphones and tablets include Android and iOS.

1.3 The Java Virtual Machine

MACHINE LANGUAGE CONSISTS of very simple instructions that can be executed directly by the CPU of a computer. Almost all programs, though, are written in *high-level programming languages* such as Java, Python, or C++. A program written in a high-level language cannot be run directly on any computer. First, it has to be translated into machine language. This translation can be done by a program called a *compiler*. A compiler takes a high-level-language program and translates it into an executable machine-language program. Once the translation is done, the machine-language program can be run any number of times, but of course it can only be run on one type of computer, since each type of computer has its own individual machine language. (In fact, Java also depends on the particular operating system under which it is running, since it must work with the operating system to perform certain tasks such as accessing the computer's hardware. But let's ignore that complication here.) If the program is to run on another type of computer it has to be re-translated, using a different compiler, into the appropriate machine language.

There is an alternative to compiling a high-level language program. Instead of using a compiler, which translates the program all at once, you can use an *interpreter*, which translates it instruction-by-instruction, as necessary. An interpreter is a program that acts much like a CPU, with a kind of fetch-and-execute cycle. In order to execute a program, the interpreter runs in a loop in which it repeatedly reads one instruction from the program, decides what is necessary to carry out that instruction, and then performs the appropriate machine-language commands to do so.

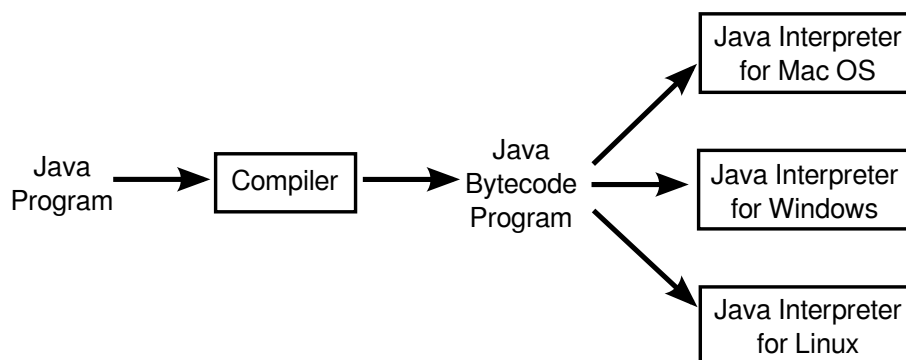
(A compiler is like a human translator who translates an entire book from one language to another, producing a new book in the second language. An interpreter is more like a human interpreter who translates a speech at the United Nations from one language to another at the same time that the speech is being given.)

One use of interpreters is to execute high-level language programs. For example, the programming language Lisp is usually executed by an interpreter rather than a compiler. However, interpreters have another purpose: They can let you use a machine-language program meant for one type of computer on a completely different type of computer. For example, one of the original home computers was the Commodore 64 or "C64". While you might not find an actual C64, you can find programs that run on other computers—or even in a web browser—that "emulate" one. Such an emulator can run C64 programs by acting as an interpreter for the C64 machine language.

* * *

The designers of Java chose to use a combination of compiling and interpreting. Programs written in Java are compiled into machine language, but it is a machine language for a computer that doesn't really exist. This so-called "virtual" computer is known as the *Java Virtual Machine*, or JVM. The machine language for the Java Virtual Machine is called *Java bytecode*. There is no reason why Java bytecode couldn't be used as the machine language of a real computer, rather than a virtual computer. But in fact the use of a virtual machine makes possible one of the main selling points of Java: the fact that it can actually be used on **any** computer. All that the computer needs is an interpreter for Java bytecode. Such an interpreter simulates the JVM in the same way that a C64 emulator simulates a Commodore 64 computer. (The term JVM is also used for the Java bytecode interpreter program that does the simulation, so we say that a computer needs a JVM in order to run Java programs. Technically, it would be more correct to say that the interpreter *implements* the JVM than to say that it *is* a JVM.)

Of course, a different Java bytecode interpreter is needed for each type of computer, but once a computer has a Java bytecode interpreter, it can run any Java bytecode program, and the same program can be run on any computer that has such an interpreter. This is one of the essential features of Java: the same compiled program can be run on many different types of computers.



Why, you might wonder, use the intermediate Java bytecode at all? Why not just distribute the original Java program and let each person compile it into the machine language of whatever computer they want to run it on? There are several reasons. First of all, a compiler has to understand Java, a complex high-level language. The compiler is itself a complex program. A Java bytecode interpreter, on the other hand, is a relatively small, simple program. This makes it easy to write a bytecode interpreter for a new type of computer; once that is done, that computer can run any compiled Java program. It would be much harder to write a Java compiler for the same computer.

Furthermore, Java was created with the idea that some programs would be downloaded over a network. This leads to obvious security concerns: you don't want to download and run a program that will damage your computer or your files. The bytecode interpreter acts as a buffer between you and the program you download. You are really running the interpreter, which runs the downloaded program indirectly. The interpreter can protect you from potentially dangerous actions on the part of that program.

When Java was still a new language, it was criticized for being slow: Since Java bytecode was executed by an interpreter, it seemed that Java bytecode programs could never run as quickly as programs compiled into native machine language (that is, the actual machine language of the computer on which the program is running). However, this problem has been largely overcome by the use of *just-in-time compilers* for executing Java bytecode. A just-in-time compiler translates Java bytecode into native machine language. It does this while it is executing the program. Just as for a normal interpreter, the input to a just-in-time compiler is a Java bytecode program, and its task is to execute that program. But as it is executing the program, it also translates parts of it into the native machine language. The translated parts of the program can then be executed much more quickly than they could be interpreted. Since a given part of a program is often executed many times as the program runs, a just-in-time compiler can significantly speed up the overall execution time.

I should note that there is no necessary connection between Java and Java bytecode. A program written in Java could certainly be compiled into the machine language of a real computer. And programs written in other languages can be compiled into Java bytecode. However, the combination of Java and Java bytecode is platform-independent, secure, and

network-compatible while allowing you to program in a modern high-level object-oriented language.

There are even some other programming languages that compile into Java bytecode. The compiled bytecode programs can then be executed by a standard JVM. New languages that have been developed specifically for programming the JVM include Scala, Groovy, Clojure, and Processing. Jython and JRuby are versions of older languages, Python and Ruby, that target the JVM. These languages make it possible to enjoy many of the advantages of the JVM while avoiding some of the technicalities of the Java language. In fact, the use of other languages with the JVM has become important enough that several new features have been added to the JVM specifically to add better support for some of those languages. And this improvement to the JVM has in turn made possible some new features in Java.

* * *

I should also note that the really hard part of platform-independence is providing a “Graphical User Interface”—with windows, buttons, etc.—that will work on all the platforms that support Java. You’ll see more about this problem in [Section 1.6](#).

1.4 Fundamental Building Blocks of Programs

THERE ARE TWO BASIC ASPECTS of programming: data and instructions. To work with data, you need to understand *variables* and *types*; to work with instructions, you need to understand *control structures* and *subroutines*. You’ll spend a large part of the course becoming familiar with these concepts.

A *variable* is just a memory location (or several consecutive locations treated as a unit) that has been given a name so that it can be easily referred to and used in a program. The programmer only has to worry about the name; it is the compiler’s responsibility to keep track of the memory location. As a programmer, you just need to keep in mind that the name refers to a kind of “box” in memory that can hold data, even though you don’t have to know where in memory that box is located.

In Java and in many other programming languages, a variable has a *type* that indicates what sort of data it can hold. One type of variable might hold integers—whole numbers such as 3, -7, and 0—while another holds floating point numbers—numbers with decimal points such as 3.14, -2.7, or 17.0. (Yes, the computer does make a distinction between the integer 17 and the floating-point number 17.0; they actually look quite different inside the computer.) There could also be types for individual characters (`'A'`, `';`, etc.), strings (“Hello”, “A string can include many characters”, etc.), and less common types such as dates, colors, sounds, or any other kind of data that a program might need to store.

Programming languages always have commands for getting data into and out of variables and for doing computations with data. For example, the following “assignment statement,” which might appear in a Java program, tells the computer to take the number stored in the variable named “principal”, multiply that number by 0.07, and then store the result in the variable named “interest”:

```
interest = principal * 0.07;
```

There are also “input commands” for getting data from the user or from files on the computer’s disks, and there are “output commands” for sending data in the other direction.

These basic commands—for moving data from place to place and for performing computations—are the building blocks for all programs. These building blocks are combined

into complex programs using control structures and subroutines.

* * *

A program is a sequence of instructions. In the ordinary “flow of control,” the computer executes the instructions in the sequence in which they occur in the program, one after the other. However, this is obviously very limited: the computer would soon run out of instructions to execute. **Control structures** are special instructions that can change the flow of control. There are two basic types of control structure: **loops**, which allow a sequence of instructions to be repeated over and over, and **branches**, which allow the computer to decide between two or more different courses of action by testing conditions that occur as the program is running.

For example, it might be that if the value of the variable “principal” is greater than 10000, then the “interest” should be computed by multiplying the principal by 0.05; if not, then the interest should be computed by multiplying the principal by 0.04. A program needs some way of expressing this type of decision. In Java, it could be expressed using the following “if statement”:

```
if (principal > 10000)
    interest = principal * 0.05;
else
    interest = principal * 0.04;
```

(Don’t worry about the details for now. Just remember that the computer can test a condition and decide what to do next on the basis of that test.)

Loops are used when the same task has to be performed more than once. For example, if you want to print out a mailing label for each name on a mailing list, you might say, “Get the first name and address and print the label; get the second name and address and print the label; get the third name and address and print the label...” But this quickly becomes ridiculous—and might not work at all if you don’t know in advance how many names there are. What you would like to say is something like “While there are more names to process, get the next name and address, and print the label.” A loop can be used in a program to express such repetition.

* * *

Large programs are so complex that it would be almost impossible to write them if there were not some way to break them up into manageable “chunks.” Subroutines provide one way to do this. A **subroutine** consists of the instructions for performing some task, grouped together as a unit and given a name. That name can then be used as a substitute for the whole set of instructions. For example, suppose that one of the tasks that your program needs to perform is to draw a house on the screen. You can take the necessary instructions, make them into a subroutine, and give that subroutine some appropriate name—say, “drawHouse()”. Then anyplace in your program where you need to draw a house, you can do so with the single command:

```
drawHouse();
```

This will have the same effect as repeating all the house-drawing instructions in each place.

The advantage here is not just that you save typing. Organizing your program into subroutines also helps you organize your thinking and your program design effort. While writing the house-drawing subroutine, you can concentrate on the problem of drawing a house without worrying for the moment about the rest of the program. And once the subroutine is written, you can forget about the details of drawing houses—that problem is solved, since you have a

subroutine to do it for you. A subroutine becomes just like a built-in part of the language which you can use without thinking about the details of what goes on “inside” the subroutine.

* * *

Variables, types, loops, branches, and subroutines are the basis of what might be called “traditional programming.” However, as programs become larger, additional structure is needed to help deal with their complexity. One of the most effective tools that has been found is object-oriented programming, which is discussed in the next section.

1.5 Objects and Object-oriented Programming

PROGRAMS MUST BE DESIGNED. No one can just sit down at the computer and compose a program of any complexity. The discipline called *software engineering* is concerned with the construction of correct, working, well-written programs. The software engineer tries to use accepted and proven methods for analyzing the problem to be solved and for designing a program to solve that problem.

During the 1970s and into the 80s, the primary software engineering methodology was *structured programming*. The structured programming approach to program design was based on the following advice: To solve a large problem, break the problem into several pieces and work on each piece separately; to solve each piece, treat it as a new problem which can itself be broken down into smaller problems; eventually, you will work your way down to problems that can be solved directly, without further decomposition. This approach is called *top-down programming*.

There is nothing wrong with top-down programming. It is a valuable and often-used approach to problem-solving. However, it is incomplete. For one thing, it deals almost entirely with producing the **instructions** necessary to solve a problem. But as time went on, people realized that the design of the **data structures** for a program was at least as important as the design of subroutines and control structures. Top-down programming doesn’t give adequate consideration to the data that the program manipulates.

Another problem with strict top-down programming is that it makes it difficult to reuse work done for other projects. By starting with a particular problem and subdividing it into convenient pieces, top-down programming tends to produce a design that is unique to that problem. It is unlikely that you will be able to take a large chunk of programming from another program and fit it into your project, at least not without extensive modification. Producing high-quality programs is difficult and expensive, so programmers and the people who employ them are always eager to reuse past work.

* * *

So, in practice, top-down design is often combined with *bottom-up design*. In bottom-up design, the approach is to start “at the bottom,” with problems that you already know how to solve (and for which you might already have a reusable software component at hand). From there, you can work upwards towards a solution to the overall problem.

The reusable components should be as “modular” as possible. A *module* is a component of a larger system that interacts with the rest of the system in a simple, well-defined, straightforward manner. The idea is that a module can be “plugged into” a system. The details of what goes on inside the module are not important to the system as a whole, as long as the module fulfills its assigned role correctly. This is called *information hiding*, and it is one of the most important principles of software engineering.

One common format for software modules is to contain some data, along with some subroutines for manipulating that data. For example, a mailing-list module might contain a list of names and addresses along with a subroutine for adding a new name, a subroutine for printing mailing labels, and so forth. In such modules, the data itself is often hidden inside the module; a program that uses the module can then manipulate the data only indirectly, by calling the subroutines provided by the module. This protects the data, since it can only be manipulated in known, well-defined ways. And it makes it easier for programs to use the module, since they don't have to worry about the details of how the data is represented. Information about the representation of the data is hidden.

Modules that could support this kind of information-hiding became common in programming languages in the early 1980s. Since then, a more advanced form of the same idea has more or less taken over software engineering. This latest approach is called ***object-oriented programming***, often abbreviated as OOP.

The central concept of object-oriented programming is the ***object***, which is a kind of module containing data and subroutines. The point-of-view in OOP is that an object is a kind of self-sufficient entity that has an internal ***state*** (the data it contains) and that can respond to ***messages*** (calls to its subroutines). A mailing list object, for example, has a state consisting of a list of names and addresses. If you send it a message telling it to add a name, it will respond by modifying its state to reflect the change. If you send it a message telling it to print itself, it will respond by printing out its list of names and addresses.

The OOP approach to software engineering is to start by identifying the objects involved in a problem and the messages that those objects should respond to. The program that results is a collection of objects, each with its own data and its own set of responsibilities. The objects interact by sending messages to each other. There is not much “top-down” in the large-scale design of such a program, and people used to more traditional programs can have a hard time getting used to OOP. However, people who use OOP would claim that object-oriented programs tend to be better models of the way the world itself works, and that they are therefore easier to write, easier to understand, and more likely to be correct.

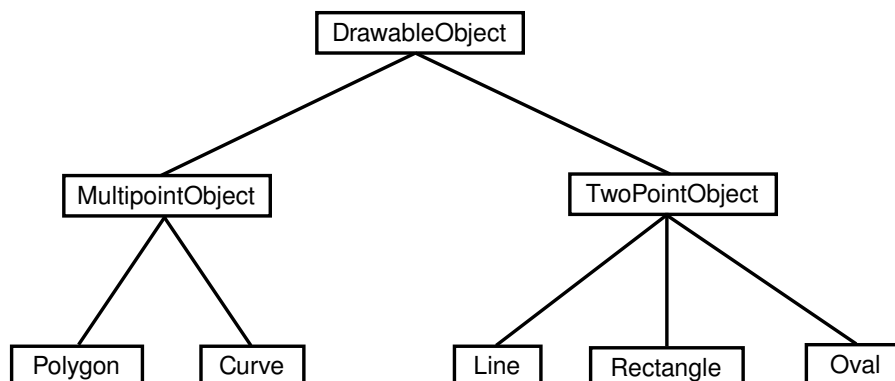
* * *

You should think of objects as “knowing” how to respond to certain messages. Different objects might respond to the same message in different ways. For example, a “print” message would produce very different results, depending on the object it is sent to. This property of objects—that different objects can respond to the same message in different ways—is called ***polymorphism***.

It is common for objects to bear a kind of “family resemblance” to one another. Objects that contain the same type of data and that respond to the same messages in the same way belong to the same ***class***. (In actual programming, the class is primary; that is, a class is created and then one or more objects are created using that class as a template.) But objects can be similar without being in exactly the same class.

For example, consider a drawing program that lets the user draw lines, rectangles, ovals, polygons, and curves on the screen. In the program, each visible object on the screen could be represented by a software object in the program. There would be five classes of objects in the program, one for each type of visible object that can be drawn. All the lines would belong to one class, all the rectangles to another class, and so on. These classes are obviously related; all of them represent “drawable objects.” They would, for example, all presumably be able to respond to a “draw yourself” message. Another level of grouping, based on the data needed to represent each type of object, is less obvious, but would be very useful in a program: We can

group polygons and curves together as “multipoint objects,” while lines, rectangles, and ovals are “two-point objects.” (A line is determined by its two endpoints, a rectangle by two of its corners, and an oval by two corners of the rectangle that contains it. The rectangles that I am talking about here have sides that are vertical and horizontal, so that they can be specified by just two points; this is the common meaning of “rectangle” in drawing programs.) We could diagram these relationships as follows:



DrawableObject, MultipointObject, and TwoPointObject would be classes in the program. MultipointObject and TwoPointObject would be *subclasses* of DrawableObject. The class Line would be a subclass of TwoPointObject and (indirectly) of DrawableObject. A subclass of a class is said to *inherit* the properties of that class. The subclass can add to its inheritance and it can even “override” part of that inheritance (by defining a different response to some message). Nevertheless, lines, rectangles, and so on **are** drawable objects, and the class DrawableObject expresses this relationship.

Inheritance is a powerful means for organizing a program. It is also related to the problem of reusing software components. A class is the ultimate reusable component. Not only can it be reused directly if it fits exactly into a program you are trying to write, but if it just almost fits, you can still reuse it by defining a subclass and making only the small changes necessary to adapt it exactly to your needs.

So, OOP is meant to be both a superior program-development tool and a partial solution to the software reuse problem. Objects, classes, and object-oriented programming will be important themes throughout the rest of this text. You will start using objects that are built into the Java language in the next chapter, and in [Chapter 5](#) you will begin creating your own classes and objects.

1.6 The Modern User Interface

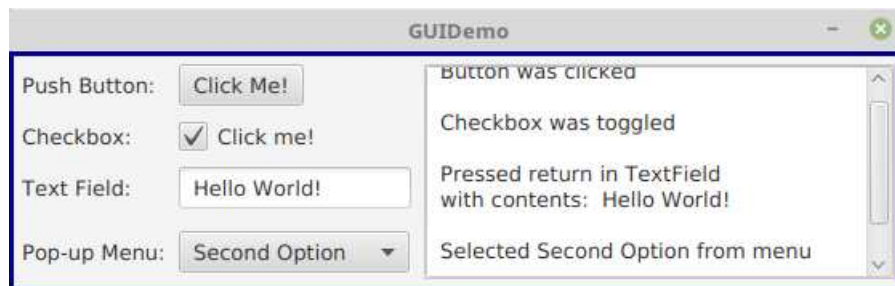
WHEN COMPUTERS WERE FIRST INTRODUCED, ordinary people—including most programmers—couldn’t get near them. They were locked up in rooms with white-coated attendants who would take your programs and data, feed them to the computer, and return the computer’s response some time later. When timesharing—where the computer switches its attention rapidly from one person to another—was invented in the 1960s, it became possible for several people to interact directly with the computer at the same time. On a timesharing system, users sit at “terminals” where they type commands to the computer, and the computer types back its response. Early personal computers also used typed commands and responses, except that there

was only one person involved at a time. This type of interaction between a user and a computer is called a *command-line interface*.

Today, of course, most people interact with computers in a completely different way. They use a *Graphical User Interface*, or GUI. The computer draws interface *components* on the screen. The components include things like windows, scroll bars, menus, buttons, and icons. Usually, a *mouse* is used to manipulate such components or, on “touchscreens,” your fingers. Assuming that you have not just been teleported in from the 1970s, you are no doubt already familiar with the basics of graphical user interfaces!

A lot of GUI interface components have become fairly standard. That is, they have similar appearance and behavior on many different computer platforms including MacOS, Windows, and Linux. Java programs, which are supposed to run on many different platforms without modification to the program, can use all the standard GUI components. They might vary a little in appearance from platform to platform, but their functionality should be identical on any computer on which the program runs.

Shown below is an image of a very simple Java program that demonstrates a few standard GUI interface components. When the program is run, a window similar to the picture shown here will open on the computer screen. There are four components in the window with which the user can interact: a button, a checkbox, a text field, and a pop-up menu. These components are labeled. There are a few other components in the window. The labels themselves are components (even though you can’t interact with them). The right half of the window is a text area component, which can display multiple lines of text. A scrollbar component appears alongside the text area when the number of lines of text becomes larger than will fit in the text area. And in fact, the whole window can itself be considered to be a “component.”



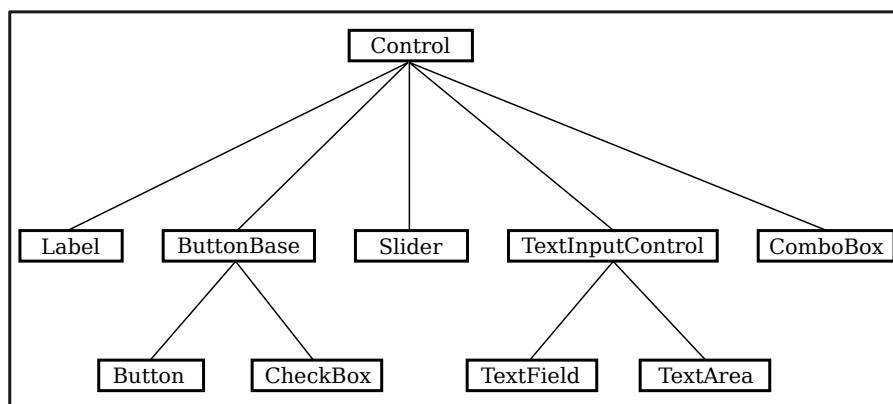
(If you would like to run this program, the source code, [GUIDemo.java](#), is available on line. For more information on using this and other examples from this textbook, see [Section 2.6](#).)

In fact, there are three complete sets of GUI components that can be used with Java. One of these, the AWT or *Abstract Windowing Toolkit*, was part of the original version of Java. The second, which is known as *Swing*, builds on the AWT; it was introduced in Java version 1.2, and was the standard GUI toolkit for many years. The third GUI toolkit, JavaFX, briefly became a standard part of Java in Version 8 but is now distributed separately. JavaFX is meant as a more modern way to write GUI applications, but using it is complicated by the fact that it has to be downloaded and installed separately. This textbook covers JavaFX exclusively, but an alternative version of the textbook is available that covers Swing instead. Either version of the textbook can be a reasonable choice.

When a user interacts with GUI components, “events” are generated. For example, clicking a push button generates an event, and pressing a key on the keyboard generates an event. Each time an event is generated, a message is sent to the program telling it that the event has occurred, and the program responds according to its program. In fact, a typical GUI program

consists largely of “event handlers” that tell the program how to respond to various types of events. In the above example, the program has been programmed to respond to each event by displaying a message in the text area. In a more realistic example, the event handlers would have more to do.

The use of the term “message” here is deliberate. Messages, as you saw in the previous section, are sent to objects. In fact, Java GUI components are implemented as objects. Java includes many predefined classes that represent various types of GUI components. Some of these classes are subclasses of others. Here is a diagram showing just a few of the JavaFX GUI classes and their relationships:



Don’t worry about the details for now, but try to get some feel about how object-oriented programming and inheritance are used here. Note that all the GUI classes shown here are subclasses, directly or indirectly, of a class called *Control*, which represents general properties that are shared by many JavaFX components. In the diagram, two of the direct subclasses of *Control* themselves have subclasses. The classes *TextField* and *TextArea*, which have certain behaviors in common, are grouped together as subclasses of *TextInputControl*. Similarly *Button* and *CheckBox* are subclasses of *ButtonBase*, which represents properties common to both buttons and checkboxes. (*ComboBox*, by the way, is the class that represents pop-up menus.)

Just from this brief discussion, perhaps you can see how GUI programming can make effective use of object-oriented design. In fact, GUIs, with their “visible objects,” are probably a major factor contributing to the popularity of OOP.

Programming with GUI components and events is one of the most interesting aspects of Java. However, we will spend several chapters on the basics before returning to this topic in [Chapter 6](#).

1.7 The Internet and Beyond

COMPUTERS CAN BE CONNECTED together on *networks*. A computer on a network can communicate with other computers on the same network by exchanging data and files or by sending and receiving messages. Computers on a network can even work together on a large computation.

Today, millions of computers throughout the world are connected to a single huge network that we know as the *Internet*. New computers are being connected to the Internet every day, both by wireless communication and by physical connection using technologies such as DSL, cable modems, and Ethernet.

There are elaborate *protocols* for communication over the Internet. A protocol is simply a detailed specification of how communication is to proceed. For two computers to communicate at all, they must both be using the same protocols. The most basic protocols on the Internet are the *Internet Protocol* (IP), which specifies how data is to be physically transmitted from one computer to another, and the *Transmission Control Protocol* (TCP), which ensures that data sent using IP is received in its entirety and without error. These two protocols, which are referred to collectively as TCP/IP, provide a foundation for communication. Other protocols use TCP/IP to send specific types of information such as web pages, electronic mail, and data files.

All communication over the Internet is in the form of *packets*. A packet consists of some data being sent from one computer to another, along with addressing information that indicates where on the Internet that data is supposed to go. Think of a packet as an envelope with an address on the outside and a message on the inside. (The message is the data.) The packet also includes a “return address,” that is, the address of the sender. A packet can hold only a limited amount of data; longer messages must be divided among several packets, which are then sent individually over the Net and reassembled at their destination.

Every computer on the Internet has an *IP address*, a number that identifies it uniquely among all the computers on the Net. (Actually, the claim about uniqueness is not quite true, but the basic idea is valid, and the full truth is complicated.) The IP address is used for addressing packets. A computer can only send data to another computer on the Internet if it knows that computer’s IP address. Since people prefer to use names rather than numbers, most computers are also identified by names, called *domain names*. For example, the main computer of the Mathematics Department at Hobart and William Smith Colleges has the domain name math.hws.edu. (Domain names are just for convenience; your computer still needs to know IP addresses before it can communicate. There are computers on the Internet whose job it is to translate domain names to IP addresses. When you use a domain name, your computer sends a message to a domain name server to find out the corresponding IP address. Then, your computer uses the IP address, rather than the domain name, to communicate with the other computer.)

The Internet provides a number of services to the computers connected to it (and, of course, to the users of those computers). These services use TCP/IP to send various types of data over the Net. Among the most popular services are instant messaging, file sharing, electronic mail, and the World-Wide Web. Each service has its own protocols, which are used to control transmission of data over the network. Each service also has some sort of user interface, which allows the user to view, send, and receive data through the service.

For example, the email service uses a protocol known as *SMTP* (Simple Mail Transfer Protocol) to transfer email messages from one computer to another. Other protocols, such as POP and IMAP, are used to fetch messages from an email account so that the recipient can read them. A person who uses email, however, doesn’t need to understand or even know about these protocols. Instead, they are used behind the scenes by computer programs to send and receive email messages. These programs provide the user with an easy-to-use user interface to the underlying network protocols.

The World-Wide Web is perhaps the most exciting of network services. The World-Wide Web allows you to request *pages* of information that are stored on computers all over the Internet. A Web page can contain *links* to other pages on the same computer from which it was obtained or to other computers anywhere in the world. A computer that stores such pages of information is called a *web server*. The user interface to the Web is the type of program

known as a *web browser*. Common web browsers include Microsoft Edge, Firefox, Chrome, and Safari. You use a Web browser to request a page of information. The browser sends a request for that page to the computer on which the page is stored, and when a response is received from that computer, the web browser displays it to you in a neatly formatted form. A web browser is just a user interface to the Web. Behind the scenes, the web browser uses a protocol called *HTTP* (HyperText Transfer Protocol) to send each page request and to receive the response from the web server.

* * *

Now just what, you might be thinking, does all this have to do with Java? In fact, Java is intimately associated with the Internet and the World-Wide Web. When Java was first introduced, one of its big attractions was the ability to write *applets*. An applet was a small program that is transmitted over the Internet and that runs on a web page. Applets made it possible for a web page to perform complex tasks and have complex interactions with the user. Alas, applets suffered from a variety of problems, and they are no longer used. There are now other options for running programs on Web pages.

But applets were only one aspect of Java's relationship with the Internet. Java can be used to write complex, stand-alone applications that do not depend on a Web browser. Many of these programs are network-related. For example many of the largest and most complex web sites use web server software that is written in Java. Java includes excellent support for network protocols, and its platform independence makes it possible to write network programs that work on many different types of computer. You will learn about Java's network support in [Chapter 11](#).

Its support for networking is not Java's only advantage. But many good programming languages have been invented only to be soon forgotten. Java had the good luck to ride on the coattails of the Internet's immense and increasing popularity.

* * *

As Java has matured, its applications have reached far beyond the Net. The standard version of Java already comes with support for many technologies, such as cryptography, data compression, sound processing, and graphics. And programmers have written Java libraries to provide additional capabilities. Complex, high-performance systems can be developed in Java. For example, Hadoop, a system for large scale data processing, is written in Java. Hadoop has been used by Yahoo, Facebook, and other Web sites to process the huge amounts of data generated by their users.

Furthermore, Java is not restricted to use on traditional computers. For example, Java can be used to write programs for Android smartphones (though not for the iPhone). (Android uses Google's own version of Java and does not use the same graphical user interface components as standard Java.)

At this time, Java certainly ranks as one of the most widely used programming languages. It is a good choice for almost any programming project that is meant to run on more than one type of computing device, and is a reasonable choice even for many programs that will run on only one device. It is probably still the most widely taught language at Colleges and Universities. It is similar enough to other popular languages, such as C++, JavaScript, and Python, that knowing it will give you a good start on learning those languages as well. Overall, learning Java is a great starting point on the road to becoming an expert programmer. I hope you enjoy the journey!