SESSION 1

Programming Languages for Objects

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 data 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.

The details of the fetch-and-execute cycle are not terribly important, but there are a few basic things you should know. The CPU contains a few internal registers, which are small memory units capable of holding a single number or machine language instruction. The CPU uses one of these registers -- the program counter, or 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.)

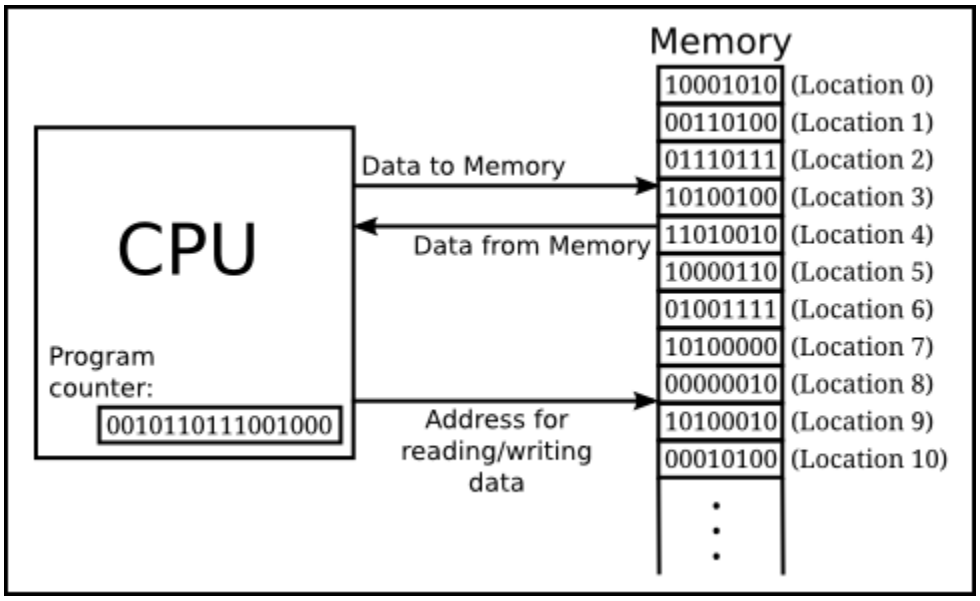
A computer executes machine language programs mechanically -- that is without understanding them or thinking about them -- simply because of the way it is physically put together. This is not an easy concept. A computer is a machine built of millions of tiny switches called transistors, which have the 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.)

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. It does 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:



Asynchronous Events: Polling Loops and Interrupts

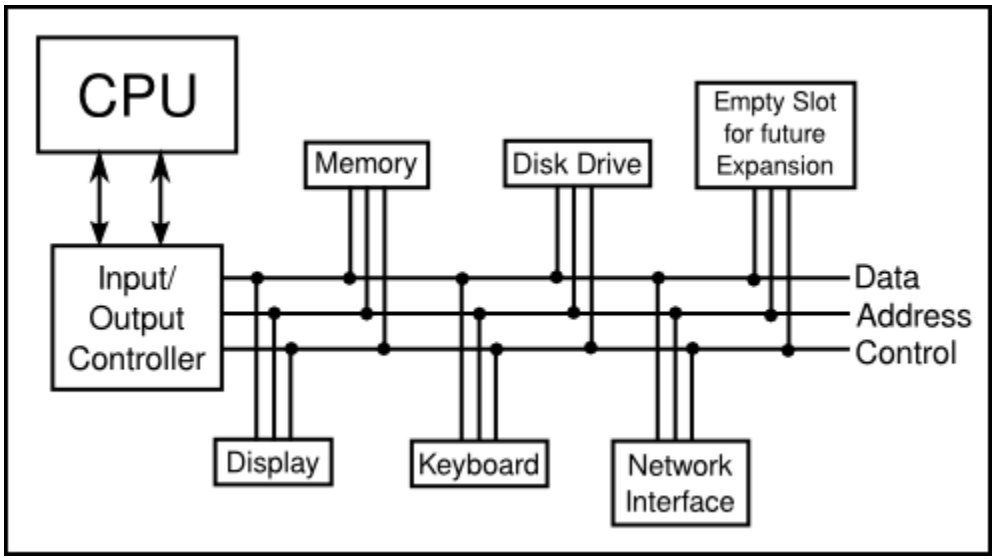
THE CPU SPENDS ALMOST ALL of its time fetching instructions from memory and executing them. However, the CPU and main memory are only two out of many components in a real computer system. A complete system contains other devices such as:

- 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. Not all computers can "forcibly" suspend a thread in this way; those that can 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 or threads 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 operating systems include Linux, various versions of Windows, and Mac

The Java Virtual Machine

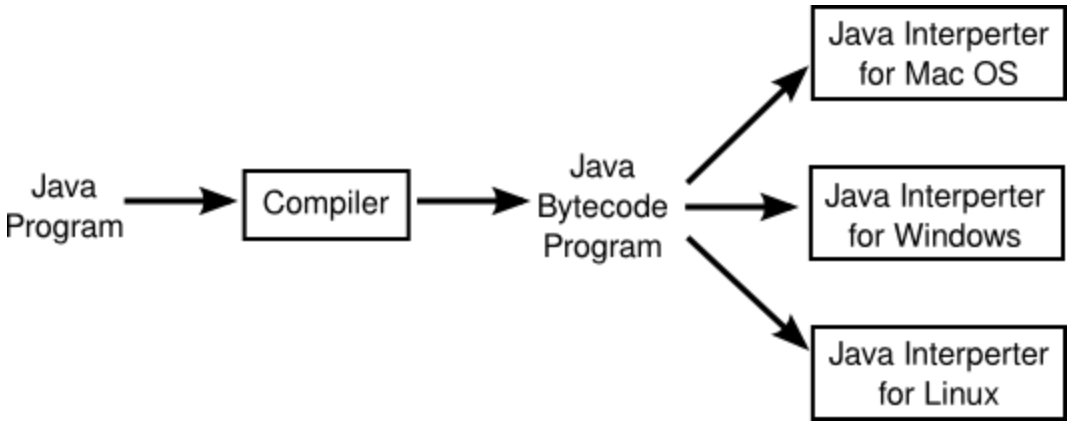
MACHINE LANGUAGE CONSISTS of very simple instructions that can be executed directly by the CPU of a computer. Almost all programs, though, are written in high-level programming languages such as Java, Fortran, or C++. A program written in a high-level language cannot be run directly on any computer. 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). 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.

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 compilation 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, some Java programs are meant to 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 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.

(In the past few years, it has become fairly common to create new programming languages, or versions of old 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 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 of the new features in Java 7 and Java 8.)

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 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.

A Brief History of Object-Oriented Programming

SIMULA was the first object language. As its name suggests it was used to create simulations. Alan Kay, who was at the University of Utah at the time, liked what he saw in the SIMULA language. He had a vision of a personal computer that would provide graphics-oriented applications and he felt that a language like SIMULA would provide a good way for non-specialists to create these applications. He sold his vision to Xerox Parc and in the early 1970s, a team headed by Alan Kay at Xerox Parc created the first personal computer called the Dynabook. Smalltalk was the object-oriented language developed for programming the Dynabook. It was a simulation and graphics-oriented programming language. Smalltalk exists to this day although it is not widely used commercially.

The idea of object-oriented programming gained momentum in the 1970s and in the early 1980s Bjorn Stroustrup integrated object-oriented programming into the C language. The resulting language was called C++ and it became the first object-oriented language to be widely used commercially.

In the early 1990s a group at Sun led by James Gosling developed a simpler version of C++ called Java that was meant to be a programming language for video-on-demand applications. This project was going nowhere until the group re-oriented its focus and marketed Java as a language for programming Internet applications. The language has gained widespread popularity as the Internet has boomed, although its market penetration has been limited by its inefficiency.

Objects

Object-oriented programming is first and foremost about objects. Initially object-oriented languages were geared toward modeling real world objects so the objects in a program corresponded to real world objects. Examples might include:

1. Simulations of a factory floor--objects represent machines and raw materials
2. Simulations of a planetary system--objects represent celestial bodies such as planets, stars, asteroids, and gas clouds
3. A PC desktop--objects represent windows, documents, programs, and folders
4. An operating system--objects represent system resources such as the CPU, memory, disks, tapes, mice, and other I/O devices

The idea with an object is that it advertises the types of data that it will store and the types of operations that it allow to manipulate that data. However, it hides its implementation from the user. For a real world analogy, think of a radio. The purpose of a radio is to play the program content of radio stations (actually translate broadcast signals into sounds that humans can understand). A radio has various dials that allow you to control functions such as the station you are tuned to, the volume, the tone, the bass, the power, and so on. These dials represent the operations that you can use to manipulate the radio. The implementation of the radio is hidden from you. It could be implemented using vacuum tubes or solid state transistors, or some other technology. The point is you do not need to know. The fact that the implementation is hidden from you allows radio manufacturers to upgrade the technology within radios without requiring you to relearn how to use a radio.

The idea is the same with objects. An object advertises the set of functions it will perform for you but does not reveal its implementation. Think of it as a black box. For example, think of the objects on a desktop. For simplicity think of two types of objects, a document and a program. A document contains data and a program contains executable statements. The desktop needs some functions to manipulate these objects. For example, it needs to be able to copy, cut, and paste these objects. It does not care how these objects are implemented. It just needs them to perform those three functions. Consequently both documents and programs provide functions that allow them to be copied, cut, and pasted.

As another example, consider a stack. A stack provides a set of operations such as push, pop, isempty, and top. If the stack is implemented as an object, its implementation will be hidden from the program. It may be implemented as an array, a queue, or some other data structure. The program does not need to know how the array is implemented. Its only concern is that the stack provide the specified operations and that the operations provide the desired behavior.

The set of operations provided by an object is called its **interface**. The interface defines both the names of the operations and the behavior of these operations. In essence the interface is a contract between the object and the program that uses it. The object guarantees that it will provide the advertised set of operations and that they will behave in a specified fashion. Any object that adheres to this contract can be used interchangeably by the program. Hence the implementation of an object can be changed without affecting the behavior of a program. For example, we can replace a stack object that is implemented as an array with a stack object that is implemented as a queue without affecting the behavior of the program.

Classes

An object is not much good if each one must be custom crafted. For example, radios would not be nearly as prevalent if each one was handcrafted. What is needed is a way to provide a blueprint for an object and a way for a "factory" to use this blueprint to mass produce objects. Classes provide this mechanism in object-oriented programming. A **class** is a factory that is able to mass produce objects. The programmer provides a class with a blueprint of the desired type of object. A "blueprint" is actually composed of:

1. A declaration of a set of variables that the object will possess,
2. A declaration of the set of operations that the object will provide, and
3. A set of function definitions that implements each of these operations.

The set of variables possessed by each object are called **instance variables**. The set of operations that the object provides are called **methods**. For most practical purposes, a method is like a function.

When a program wants a new instance of an object, it asks the appropriate class to create a new object for it. The class allocates memory to hold the object's instance variables and returns the object to the program. Each object knows which class created it so that when an operation is requested for that object, it can look up in the class the function that implements that operation and call that function.

Inheritance

To motivate inheritance, think of a radio alarm clock. A radio alarm clock has all of the functions of a radio plus additional functions to handle the alarm clock. If we adopt the radio's interface for the radio alarm clock, then someone who knows how to operate a radio will also know how to operate the radio portion of the radio alarm clock. Hence, rather than designing the radio alarm clock from scratch, we can extend or inherit the interface defined by the radio. Of course, we can also use the existing implementation for a radio and extend it to handle the alarm clock functions.

In object-oriented programming, **Inheritance** means the inheritance of another object's interface, and possibly its implementation as well. Inheritance is accomplished by stating that a new class

is a **subclass** of an existing class. The class that is inherited from is called the **superclass**. The subclass always inherits the superclass's complete interface. It can extend the interface but it cannot delete any operations from the interface. The subclass also inherits the superclass's implementation, or in other words, the functions that implement the superclass's operations. However, the subclass is free to define new functions for these operations. This is called **overriding** the superclass's implementation. The subclass can selectively pick and choose which functions it overrides. Any functions that are not overridden are inherited.

There is a great deal of debate about how to use inheritance. In particular, the debate swirls about whether inheritance should be used when you want to inherit an interface or whether it should be used when you want to inherit implementation. For example, suppose that you want to define a search object that stores (key, value) pairs and allows values to be looked up by providing their keys. More precisely, let us say that the search object supports the following operations:

- insert: Adds a (key, value) pair to the object.
- delete: Deletes a (key, value) pair from the object.
- lookup: Given a key retrieves the value associated with that key from the object.

Later we decide that we want a new object that allows us to traverse the (key, value) pairs in sorted order. The new object should support the above operations plus two additional operations, `rewind` that puts us back to the beginning, and `next` that returns the next (key, value) pair. Since the new object supports all of the operations of the original search object, we can make the new object inherit the original object's interface. This is an example of interface inheritance.

To give an example of implementation inheritance, suppose that we want to implement the original search object using a binary search tree. The binary search tree probably already has an implementation for these three operations but it may not use these names for the operations. If we wanted to inherit the binary tree's implementation, we would make the search object be a subclass of the binary tree. We could then make the insert, delete, and lookup operations call the appropriate binary tree operations. Of course, we could also scrap our proposed interface and use the names of the binary tree interface instead.

Does something seem wrong with this picture? Well, remember that an object is supposed to hide its implementation and that it should be interchangeable with other objects that implement the same interface. We can't very well scrap the interface and use the binary tree's interface because that would tie the interface to the binary tree's interface. So we should hold firm on our originally proposed interface. However, there's another problem. By making the search object inherit from the binary tree, we've also made its implementation dependent on the binary tree.

Hopefully the above example shows why implementation inheritance may not be a good idea. In general I've found that inheritance should be used only when you want to inherit an interface. If it so happens that the implementation you get can be also be used, well and good. There are other ways to re-use implementation and we will discuss those later in the course.

Abstract Data Types

Objects provide an ideal mechanism for implementing abstract data types. An abstract data type is:

1. A type of data, and
2. A set of operations for manipulating that data.

Examples of abstract data types include stacks, trees, and hash tables. The reason for the word "abstract" is that an abstract data type defines only the set of operations it supports (i.e., its interface). It does not define an implementation. In order to make the data type concrete one must provide an implementation.

An object is a nice implementation vehicle for abstract data types since the data stored by the object can represent the abstract data type's data and the object's interface can represents the abstract data type's set of operations.

Execution Model

One of the original purposes for object languages was to model applications that have multiple objects that may be operating simultaneously. For example, the machines on a factory floor operate simultaneously. In an operating system the various input devices, such as disks, the keyboard, and the mouse, may be operating simultaneously. In a computer game various players, either human or computer-generated, may be operating simultaneously.

These applications all have something in a common--they do not have a single thread of control. Instead control is distributed throughout the application. At any given moment multiple objects may want to perform some action. Conventional imperative languages like C have trouble modeling this type of application because they have a single thread of control. Object languages solved this problem by making everything an object and having control reside within each object. That is, at any given moment multiple objects could be executing an operation (at least this is the conceptual model--a computer with a sequential processor might simulate this model by interleaving the execution of the operations). Objects would communicate with one another by passing messages. A message is simply an invocation of an operation in another object. For example, an operating system might queue input events arriving from various input devices. Each input device might be represented as an object and the input event queue might be represented as an object. Each time an input device receives an input event, it would invoke the operation on the event queue that adds an input event to the queue.

Smalltalk and Java are two well known examples of this "everything is an object" concept. They are also well suited for modeling distributed responsibility. Unfortunately, not every application needs to have distributed responsibility and that is where these object-oriented languages run into problems. People try to force this model onto applications that are more naturally modeled using a single thread of control. When this happens, programmers, especially programmers learning object-oriented programming, get confused.

In this course we will use C++ which has a different execution model. It retains the notion of a central thread of control. Objects provide services to the central thread of control. For example, a stack object provides a stack service. We will primarily use objects to implement abstract data types. Using objects in this way will give us the advantages of data encapsulation and code re-use. Data encapsulation means that the implementation of an object is hidden and hence we can interchange objects with different implementations but the same interface. Code re-use means that we can use the same object in different applications, so we don't have to write the code twice.

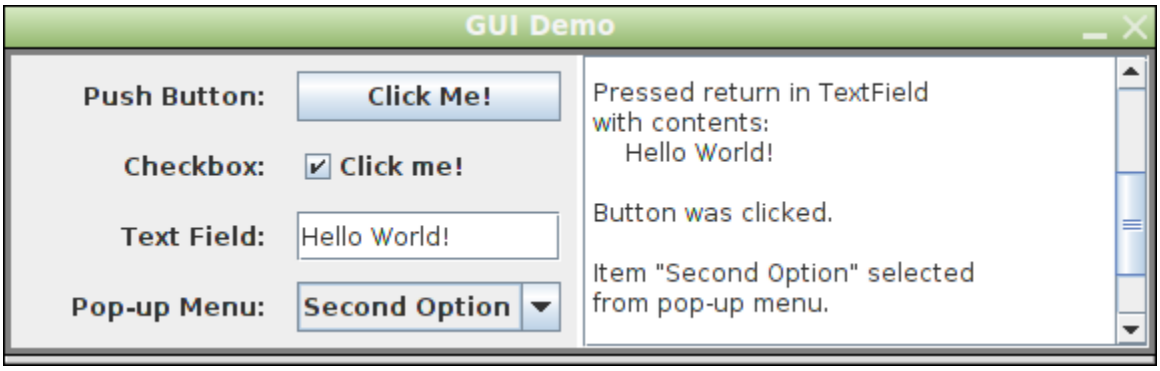
The Modern User Interface

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 Mac OS, 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, in Java terminology, the whole window is itself considered to be a "component."

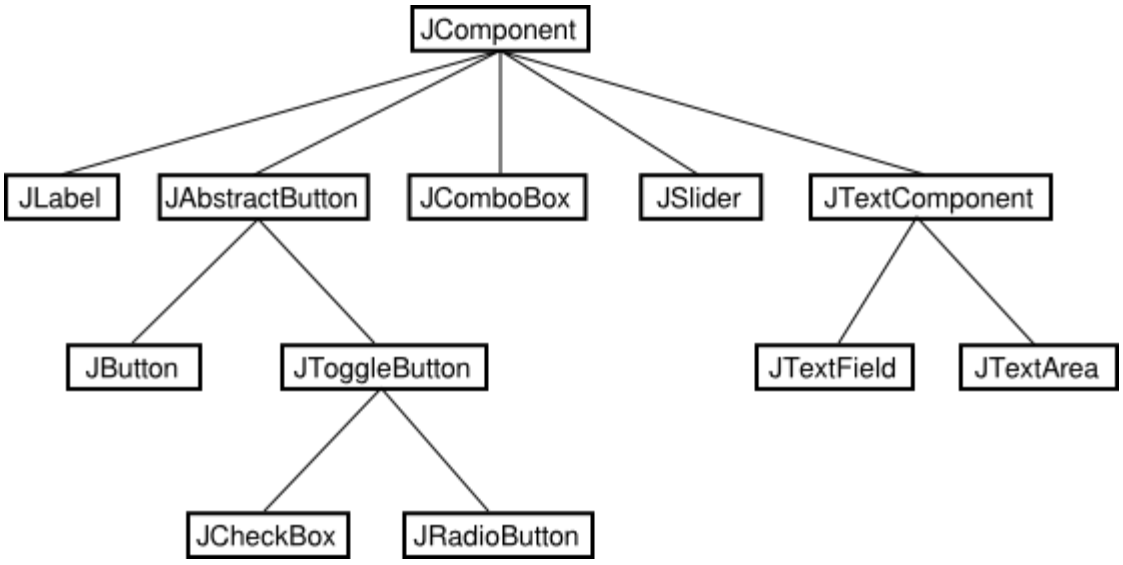


(If you would like to run this program, the source code, *GUIDemo.java*, as well as a compiled program, *GUIDemo.jar*, are available on line. For more information on using this and other examples from this textbook, see Section 2.6.)

Now, Java actually has two complete sets of GUI components. One of these, the AWT or Abstract Windowing Toolkit, was available in the original version of Java. The other, which is known as Swing, was introduced in Java version 1.2, and is used in preference to the AWT in most modern Java programs. The program that is shown above uses components that are part of Swing.

When a user interacts with GUI components, "events" are generated. For example, clicking a push button generates an event, and pressing return while typing in a text field 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 this 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 Swing's 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 are subclasses, directly or indirectly, of a class called *JComponent*, which represents general properties that are shared by all Swing components. In the diagram, two of the direct subclasses of *JComponent* themselves have subclasses. The classes *JTextArea* and *JTextField*, which have certain behaviors in common, are grouped together as subclasses of *JTextComponent*. Similarly *JButton* and *JToggleButton* are subclasses of *JAbstractButton*, which represents properties common to both buttons and checkboxes. (*JComboBox*, by the way, is the Swing 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.

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 called 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 Internet Explorer, 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 is a small program that is transmitted over the Internet and that runs on a web page. Applets make it possible for a web page to perform complex tasks and have complex interactions with the user. Alas, applets have suffered from a variety of security problems, and fixing those problems has made them more difficult to use. Applets have become much less common on the Web, and in any case, there are other options for running programs on Web pages.

But applets are 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 has 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 and data compression. Free extensions are available to support many other technologies such as advanced sound processing and three-dimensional graphics. Complex, high-performance systems can be developed in Java. For example, Hadoop, a system for large scale data processing, is written in Java. Hadoop is 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. Java can be used to write programs for many smartphones (though not for the iPhone). It is the primary development language for Android-based devices. (Some mobile devices use a version of Java called Java ME ("Mobile Edition"), but Android uses Google's own version of Java and does not use the same

graphical user interface components as standard Java.) Java is also the programming language for the Amazon Kindle eBook reader and for interactive features on Blu-Ray video disks.

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, C++, 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!