Programming:
The Fundamental Concepts of Computer Coding

Harry H. Porter III[*]
Portland State University

March 19, 2003

Abstract

This paper surveys the most basic concepts of programming and is intended for someone without any programming experience. Simple examples in the Java language are used to illustrate these core ideas.

How do computers work? Programmers create **programs** and these programs tell the computer how to behave. The act of writing programs is called **coding** and the programs, when taken together, are called **code**.

Each program is written in some **programming language**. There are hundreds of programming languages in use today, but the most widely known are Java, C++, VisualBasic and Perl. In the past, programming languages like Fortran, Basic, C, Pascal, Smalltalk and Lisp had more prominent roles and many programs written in these languages are still in widespread use.

Today, many languages are designed for specialized applications. HTML, which is used in formatting web pages, does not qualify as a true programming language, since it is too specialized and lacks important features available in the other languages.

When a program is created, it is placed in a **file**. A file is simply a sequence of bytes which is given a name and stored on a disk. Recall that a **bit** is the smallest unit of data storage. A **byte**, which is a sequence of 8 bits, is another, slightly larger, measure of data storage. The size of

[*] Author's Web Page: `www.cs.pdx.edu/~harry`
Author's E-mail: `harry@cs.pdx.edu`
This Paper: `www.cs.pdx.edu/~harry/musings/Programming.pdf`
`www.cs.pdx.edu/~harry/musings/Programming.html`

disks is measured in bytes or, more likely, millions or billions of bytes. The rate of data transmission can be measured in bytes per second.

A byte is 8 bits. Here are two different bytes of data:

a byte:	0 0 1 0 1 1 0 1
another byte:	1 1 1 0 1 0 0 1

How many different combinations of 8 bits are there? There are 256 different combinations, so a byte can hold any one of 256 different values. These different values are numbered from 0 through 255.

Possible byte values
0
1
2
...
255

So, in one byte of memory, we can store a single number, as long as it is between 0 and 255. If we set up a correspondence between characters and numbers, then we could interpret this number as a character. Here is such a correspondence:

Byte value	Character
0	`A`
1	`B`
2	`C`
...	...
25	`Z`
26	`a`
27	`b`
28	`c`
...	...
51	`z`
52	`0`
53	`1`
54	`2`
...	...
61	`9`
62	(space character)
63	`(`
64	`)`
65	`-`
...	...

Using a table like this, we can store a single character in each byte. We can store 10 characters, such as "`Hello there`", in 10 bytes.

The **ASCII code** is exactly such a correspondence between byte values and characters as we showed above. The ASCII code is a little different than our code, but the idea is the same. The ASCII code defines a fixed set of characters that are available for use. It includes most of the characters on the keyboard, but it does not include characters like é or □ or □ , and it does not include any information about font size, italics, etc. Files that include only ASCII characters are called **plain-text** files. There is one byte per character and their interpretation as characters is quite standardized.

In addition to characters, the ASCII code includes several **control characters**. These are not characters at all, but are used to convey other information. The most important control characters are **newline** and **tab**. We use the abbreviation **\n** for newline and **\t** for tab. In the ASCII code, newline happens to be number 10 and tab happens to be number 9. There are other control characters, which are given names like control-A, control-B, control-C. The newline is also called control-J and tab is control-I. Hitting the enter key usually sends a control-J (newline) and hitting the tab key sends a control-I (tab) to the computer.

There are many different programs for creating files. Word processing programs, such as MS Word, put a lot more into the file than just the ASCII character codes. The extra material is used to represent information about font, spacing, and so on. Programmers, on the other hand, use simpler **text editors** to create the files containing their programs. A text editor (as opposed to a word processor) is used to create and manipulate plain-text files. Programs are placed in plain-text files, containing only ASCII characters.

A program is a file containing a sequence of **statements**. Each statement tells the computer what to do. Here is an example program:

```
print ("Hello.")
print ("I am a computer.")
print ("Have a nice day!")
```

The term "statement" might be misleading, since these are imperative statements (i.e., commands), not statements of fact. They say "do this" or "do that." Statements in English have the quality of being true or false, while each statement in a program instructs the computer to take some action.

Once a program is created, it can be **run**. The program tells the computer what to do, but the instructions are not followed until the program is run. Often we say the program is **executed** (or individual instructions in a program are executed); this means that the commands are followed and actions occur.

When a program contains a sequence of statements, they are executed one after the other when the program is run. **Sequential execution** is the norm: when one statement follows another, the second statement will be executed immediately after the first statement.

When we run the program above, it will produce the following **output**, which will be displayed somewhere:

```
Hello.
I am a computer.
Have a nice day!
```

When a program is run, it will interact with the outside world. It may display material on the screen; it may accept input from the keyboard and mouse; it may modify files or cause the printer to print something. It may also communicate with other programs over the internet or control various devices (like motors and lights) connected to the computer. Collectively, this interaction is called **input/output** and it occupies a huge part of many programs, since the interactions and communications can be quite complex.

In the simplest programs, a program will produce simple (ASCII) characters as output and will accept characters typed at keyboard as input.

Programs can be usefully compared to sequences of instructions from other areas of life. A cooking recipe contains a sequence of commands:

Beat 6 eggs in a bowl.
Add 3 cups of sugar.
Set the oven to 375 °F.
...

In a recipe, we assume the thing executing the program is a human and we can be rather vague in our instructions. We can say things like:

Make a white sauce.
Add vanilla to taste.
Bake until done.

With computers, all instructions are precise and unambiguous. Programming languages have been designed in such a way that every instruction means exactly one thing. The instruction will be executed exactly as it was written. Generally, the programmer writes the instruction correctly, so the computer does the desired thing.

Sometimes, the programmer makes a mistake and then the program contains a **bug**. For example, if our program had been:

```
print ("Helllo")
```

then the computer would have printed "Helllo" complete with the misspelling.

The ability of a computer to execute instructions precisely and exactly as they are written is phenomenal. All of the problems with computer bugs originate from mistakes in programs, mistakes made by programmers. The cases of computers producing incorrect output due to (say) extreme heat or radiation are so rare as to be ignored.

On the other hand, human fallibility is a major problem in programming. As programs become larger, the likelihood of programs containing bugs has increased. As computers control more and more life-critical processes, the consequences of bugs are also increasing. There is much ongoing effort in trying to make programs more reliable, but there seems to be no clear solution, beyond using brains and good **software engineering** practices.

When a programmer writes a program, he or she will type it into a file called a **source code** file. This file contains plain-text character data. Unfortunately, the computer cannot execute the program in this form. The program must be translated into an **executable** file (also called a **code file**).

A piece of software called a **compiler** is used to perform this translation of a program from source code to executable file. The compiler is itself a program which reads—as input—a source code file and which produces—as output—an executable file. The compiler is one piece of software that every program developer has on his or her computer, but which typical users would never see on their computers.

Each programming language has its own compiler. Thus, there is a Java compiler, a C++ compiler, and so on. Actually, each different sort of computer will have a different compiler as well. For example, there is one version of the C++ compiler for the Intel/PC computers and another C++ compiler for the Macintosh computers. Each compiler produces executable **target code** for a specific machine, such as the Pentium microprocessor (in the PC) or the G4 microprocessor (in the Mac).

A compiler is essentially a translator: it translates a program written in a **high-level language** like Java into a program written in a low-level language called **machine code**, which is the language of the microprocessor. Machine language is used to express instructions that the computer hardware can execute directly. A microprocessor can only execute very simple instructions, such as add two numbers together or move a byte of data from one place in memory to another. A single complex instruction in the high-level program might be translated into hundreds of machine code instructions.

Machine instructions are quite technical and, although the processor can execute billions of them every second, they are difficult for humans to understand. Today, almost no one writes machine code directly. Instead, everyone writes programs in a high-level language like Java since it is much easier to use. The compiler performs the detailed and tricky task of translating the program into machine code, which can then be executed on the computer.

So after creating the source code file, the programmer will run the compiler to produce the executable. First, the compiler will check the program to make sure it is legal—that is, to make sure it contains statements that conform to the requirements of the programming language—and, if there are problems, the compiler will display messages telling the programmer which statements are in error. If the program passes the compiler checks, then the compiler will create an executable.

The next step involves running the program and testing it on various combinations of input, to assure that it functions correctly. This is called **debugging**. The programmer should try his or her program repeatedly until there is high confidence that it is correct. Debugging is a critical step in program development and often takes more time than writing the program itself.

We can summarize the **program development process** as follows:

```
1.   Determine what the program should do and how it will do it.
2.   Type the program into the source code file.
3.   Run the compiler.
4.   If the compiler finds errors...
5.       Edit the source code file to fix the error.
6.       Go back to step 3.
7.   Else, if are no compiler errors...
8.       Run the executable on some input and see what happens.
9.       If the output is incorrect...
10.          Edit the source code file to fix the bug.
11.          Go back to step 3.
12.      Else, if the output is correct...
13.          If we run have not run enough tests...
14.              Go back to step 8.
15.          Else, we've tested enough...
16.              Quit.
```

The above process might be viewed as a program of sorts. In this analogy, the programmer is the computer which executes the program. However, this is not a program because it is not written in any programming language and is not precise enough to be executed by a computer. There are many details that are missing. (What should we call the file in step 2? What test input should we use in step 8? In step 10, how should we identify and fix a bug?)

The above is an **algorithm**, not a program. An algorithm tells us what to do and provides a general plan of how to do it. An algorithm may be expressed in any language. We often use a clipped form of English, where every statement is an imperative (like above), but we can also express an algorithm directly in a programming language. Technically, every program is an algorithm, but the idea behind an algorithm is that it captures the abstract idea of how a program will perform its task.

As an example of an algorithm, we can describe the process of how to sort a list of numbers. (There are several algorithms for sorting. We might try building up the list by inserting each new number is its proper place, or we might use an algorithm that looks for adjacent pairs of numbers that are out of order and switches them around.)

After choosing an algorithm, it can then be translated into a Java program or a C++ program. Conversely, if we see a program to sort a list of numbers, it is reasonable to ask, "Which sorting algorithm does this program use?"

The above algorithm for developing a program contained a number of repetitive actions. In step 6, we say "go back to step 3". This is called **looping** and occurs frequently in algorithms. The idea is that one statement (or a small sequence of statements) is executed repeatedly, over and over.

Next, let's look at an example program. Our goal is to write some code that will print out a table of Fahrenheit and Celsius temperatures. We want to print out all temperatures between 0 °F and 100 °F, along with the corresponding temperature in Celsius. We'll have to print out one line after another, repeating the same computation over and over, so we'll use a **loop statement**. A loop statement is used to perform repetitive tasks, and this ability to relieve humans from boring, repetitive tasks is well appreciated.

First, we give the program, expressed in the Java language. Then, we'll discuss it.

```
1.    // Print a table of temperatures
2.    double f, c;
3.    f = 0;
4.    while (f <= 100) {
5.          c = (f - 32) * 5 / 9;
6.          System.out.print (f + " F is " + c + " C\n");
7.          f = f + 1;
8.    }
```

I've numbered the lines for easy reference, but these are not part of the program.

On line 1, we see a program **comment**. Comments are part of the program but are not executed. Instead, comments describe and document the program. They are intended to be read by humans looking at the program and, as such, they are written in a language like English. The compiler will recognize that this is a comment by the characters // and will ignore everything else on the line after the //.

Comments are a critical part of programs, particularly large and complex programs, even though the compiler ignores them. Programs must be read and modified by humans and the comments serve to explain and guide programmers through big, difficult-to-understand chunks of code.

This program also contains some **indentation** (lines 5-7). The indentation and spacing rules are quite liberal and allow the programmer to **format the program** in more or less any way he or she wants. The standard practice is to use indentation in ways that help humans understand the code.

For example, our program could also have been written without indentation and comments, like this:

```
double f,c;f=0;while(f<=100){c=(f-32)*5/9;
System.out.print(f+" F is "+c+" C\n");f=f+1;}
```

The compiler would have no difficulty understanding this mess, but it is much harder for the programmer to read, understand, and debug.

Our example program (Refer to the formatted version!) will work with two numbers, a temperature expressed in Fahrenheit—we'll call this "f"—and a temperature expressed in Celsius, which we'll call "c". For each of these, we will have a variable, so this program uses two variables, called "f" and "c".

A **variable** is a named piece of computer memory which can hold a single value at a time. Programs can manipulate many kinds of data. One important kind of data is numeric data and there are several ways a program can represent numbers.

In this program, we will use **floating-point numbers**. The ASCII code told us how to store one character in a single byte. With floating-point numbers, we can store a number in 8 bytes. Within this chunk of 8 bytes, we can place exactly one number. The number can be a fractional number, positive or negative. Here are some examples of numbers that can be stored in a floating-point variable:

> -5
> 3.1415
> 6.023 □ 10^{27}

The relationship between real numbers (as mathematicians understand them) and the numbers we can store in a floating-point variable is not exact. With the computer representation, we are restricted to 17 digits of accuracy and the exponent on the 10 is limited to about –300 to +300.

This representation of numbers will work for our program, but an important point is that there is always a difference between what is stored in the computer's memory and the outside world. When we speak of a **data representation**, we say how some information will be stored in computer memory. There are often other data representations and part of the programmer's task is to choose a data representation that will capture enough of the information to allow the program to perform its task, but we should always be clear that a thing is different from its representation. The data representation may either fail to capture some aspects of the thing or it may capture some aspects in incorrect or misleading ways.

For example, the number □ has a definite value, but this value cannot be captured precisely with a floating-point number. Instead, the closest we can come is:

> 3.141592653589793

In line 2 of our program, the variables “f” and “c” are introduced in a **variable declaration**:

```
double f, c;
```

The word “double” is a **keyword**, which means it is part of the Java language. Java has about 45 of these special words and each has a specific meaning when used in a program. Our program also uses other keywords. Each language has a slightly different set of keywords, but the common programming languages all use keywords like **double**, **int**, **if**, **else**, **while**, and **return**. (We’ll discuss the meanings of these common keywords later in this paper.)

On this line, we create two variables. A variable declaration tells the compiler the names of our variables (“f” and “c”) and the type of data they will hold. The keyword “double” is used to mean an “8-byte floating-point number”. In a variable declaration, “double” will be followed by a list of names.

Each variable has a **type**. The variable’s type indicates what kind of data can be stored in the variable. For example, we say, “The type of f is double.” This means that we can store a single floating-point number in “f”. At any one time, there will be a number stored in “f,” but this value can be changed during the execution of the program. Thus, we speak of a variable’s current **value**.

Each programming language has a very definite **syntax**, and the compiler will check to make sure the program is syntactically correct. For example, in a variable declaration, we must have a keyword (such as “double”), followed by a list of one or more variable names, separated by commas and followed by a semi-colon. There are many other syntactic rules, such as:

For every open parenthesis “(“ there must be a matching “)”.
The addition operator + should be placed between two expressions.

In a natural language like English, a sentence may be ungrammatical, but might still be understandable. However, in a programming language each statement must conform exactly to the syntax of the language.

There is quite a bit of complexity within the syntax of a programming language. For example, here are some other syntactically-correct declarations:

```
double w, x, y, z;
double tempInCelsius;
double pi = 3.1415;
```

The Java syntax makes extensive use of **semi-colons**. In general, there is a semi-colon after each statement. These serve no real purpose, but serve to tell the compiler where the statement ends. Like the period at the end of a sentence, they can help humans to read the program and may also help the compiler in making sense of programs with syntax errors.

The second example shows that we can make up any name we want for a variable. A made-up name is called an **identifier**. An identifier may consist of any sequence of letters, but must not contain any spaces or punctuation characters and must not conflict with the keywords of the language. It is often a good idea for the programmer to name variables with identifiers chosen to convey some intuitive meaning to other humans who read the program. You can probably guess what the variables "tempInCelsius" and "pi" will contain, but it is hard to know what "z" is.

The third example declaration shows that we can give a variable an initial value when it is created. In general, **variable initialization** is often a wise thing to do, but every variable will always contain a single value at every instant after it is created. In our program, the variables "f" and "c" will always have values, even before we move data into them.

Some languages will automatically store **default values** in variables when they are created. A reasonable default value for a double number is "0.0". In other languages, the variables will simply be created without being automatically initialized. In such a situation, the initial value will be determined from what combination of bits happens to be stored in that particular 8 bytes of memory. This is an unpredictable, essentially random value.

Furthermore, if such an uninitialized variable's value is used before being set, it means that during any one execution of the program, the output may differ depending on what just happened to be in memory before the program was executed. This unpredictability is considered bad since the same program, given the same input, may produce different outputs on different executions. This problem occurs in C++ but languages like Java go to extremes to avoid it.

There are other ways to represent numbers. One common representation is using **integers**. Here is a variable declaration that uses the keyword "int" to create a variable "a" which will hold an integer value.

```
int a;
```

Every integer value will be stored in 4 bytes. With 8 bits per byte, this means that 32 bits are used to store each integer.

Of course, there is an infinite supply of whole numbers:

..., -3, -2, -1, 0, 1, 2, 3, ...

but we can only work with a finite subset of these in a computer. Using “ints”, we can store any whole number between –2,147,483,648 and +2,147,483,647.

Again, we see that the data representation matches closely—but never exactly—the thing being represented. This range is limited but is generally useful in most of the counting tasks that are done by a computer.

It should also be noted that the computation done by a computer may not always be what is expected. For example, if we add two numbers, “a” and “b”, we expect the mathematically correct result. If we represent “a” and “b” with “ints”, the computer will get the correct answer, as long as the correct answer can be represented with an “int”.

For example, assume “a” contains 25 and “b” contains 10. Then the following statement, when executed, will set “c” to 35:

```
c = a + b;
```

However, if “a” happens to contain 2,000,000,000 and “b” contains 1,000,000,000, this statement will not do what you expect. In fact, when executed, this statement will set “c” to the value –129,496,729,667, which is quite unexpected. This value doesn’t even have the correct sign!

Many subtle bugs have arisen because the programmer has forgotten that the representation of numbers in computers is not always accurate and the behavior occasionally varies from mathematical theory. A similar bug resulted in the loss of a multi-million dollar Arianne spacecraft.

Our sample program contains three **assignment statements**. They are repeated here:

```
f = 0;
...
c = (f - 32) * 5 / 9;
...
f = f + 1;
```

The equal sign “=” is used in the assignment statement. To its left, we see a single variable and, to its right, we may see any expression. Although we use the “=” sign, there is no relation to mathematical equality. Instead, the “=” sign is used to indicate data movement.

The first assignment statement says “Move the zero value into the variable f.” We read this as “f gets zero.”

The last assignment, which is read “f gets f plus 1,” says to take the current value of “f”, add 1 to it, and move the resulting value into variable “f”. This statement will **increment** “f” by 1. For example, if “f” is 56 and this assignment statement is executed, then “f” will be changed to 57.

The middle assignment statement shows that, to the right of the “=”, we can have any arbitrary expression. In this case, we are using a simple formula to calculate the temperature in Celsius, given the temperature in Fahrenheit. The symbol “*” is used to indicate multiplication and “/” is used to indicate division. This computation involves subtraction, multiplication, and division, and will convert any Fahrenheit temperature into Celsius. We can also perform calculations using any other mathematical function (like sin or square-root) that we might find on a calculator as well as other, more obscure functions.

Our task is not to perform this computation once, but to perform it many times. We need to repeat this computation in a loop. Here is the relevant portion of the program:

```
3.     f = 0;
4.     while (f <= 100) {
5.            c = (f - 32) * 5 / 9;
6.            System.out.print (f + " F is " + c + " C\n");
7.            f = f + 1;
8.     }
```

Here we have two statements. The first, on line 3, is an assignment to initialize “f”. The second statement is a **while statement**, which is on lines 4 through 8, which will do the repetitive looping.

The while statement is an example of a **compound statement**. A compound statement can contain other statements inside of it. This is akin to the sort of grammatical nesting you find in English statements like:

The fact that (Sally went to the store) was discussed among the group.

Here, one statement contains another statement within it.

The general form of a while statement is:

`while (` *...conditional expression...* `) {` *...body statements...* `}`

In our program, the **conditional expression** is:

```
f <= 100
```

and the **<u>loop body</u>** consists of the statements:

```
c = (f - 32) * 5 / 9;
System.out.print (f + " F is " + c + " C\n");
f = f + 1;
```

Notice how indentation was used in the program to set the body statements apart from the main statement sequence. This is not strictly necessary, but makes the program much easier to understand.

A conditional expression is an expression which is either true or false. We use the characters "<=" to mean □ since there is no "□" character in the ASCII code. We can also use "<", ">", and ">=".

The conditional expression will be evaluated many times during the execution of this program. Sometimes it will be true and, at the end of the execution, it will become false. When

```
f <= 100
```

is false, it will be because

```
f > 100
```

is true.

The statements in the loop body will be executed many times. Each time they are executed, the loop body statements will do the following:

> Given a value of "f", compute the corresponding value of "c".
> Print out "f" and "c".
> Add 1 to "f".

The while statement works like this:

1. Evaluate the conditional expression.
2. If it is false, stop.
3. Execute the statements in the loop body.
    a. Given a value of "f", compute the corresponding value of "c".
    b. Print out "f" and "c".
    c. Add 1 to "f".
4. Go back to step 1.

The program as a whole initializes “f” to 0 and then executes this while statement to print out the table.

Each time the loop statements are executed, we say an **<u>iteration</u>** of the loop occurs.

Let’s walk through the execution of this loop, step by step. (This is called **<u>hand-simulation</u>** of a program and it is used to help programmers understand how a program will be executed.)

As the first step of the while loop, the conditional expression will be evaluated. The conditional expression will be true so the body statements will be executed. This is the first iteration of the loop. It will result in setting “c” to -17.77777777777778, printing the values of “f” and “c”:

```
0.0 F is -17.77777777777778 C
```

and finally incrementing “f” to 1.

Next, the loop will repeat: execution will jump back up to the beginning of the loop and the conditional expression will be evaluated a second time. The value of “f” has now changed, but the expression (f <= 100) will still be true so the body will be executed a second time.

In the second iteration, our program will print this line:

```
1.0 F is -17.22222222222222 C
```

Our program will continue looping, printing lines. Ultimately, it will print the line:

```
100.0 F is 37.77777777777778 C
```

Then, the program will increment “f” to 101.

On the next iteration, it will test the conditional expression:

```
f <= 100
```

This will now be false, which will cause **<u>loop termination</u>**: the while statement will terminate, without executing the body again. Since there are no more statements after the while loop, our program will complete.

Programs with loops should terminate but, due to program bugs, they may fail to terminate. As a result, the program may have an **<u>infinite loop</u>**. A non-terminating program will continue execution forever. If the body of the loop statement contains a statement that prints output, then such a program will produce a never-ending stream of output. Otherwise, the program will continue silently, computing away like a mental patient lost forever in his or her own thoughts.

Here is a non-terminating program:

```
i = 0;
while (i < 100) {
    System.out.print ("I will not chew gum in class.");
}
```

Non-terminating programs must be stopped from outside. For example, in the UNIX operating system you can type control-C to force the operating system to stop the program prematurely. This is necessary if the program is to be stopped. Sometimes, bugs within the operating system will cause the OS itself to go into an infinite loop. (After all, the OS is also a program!) In this case, you might need to push the reset button and re-start the OS.

Our example program also contains the following **print statement** to produce output:

```
System.out.print (f + " F is " + c + " C\n");
```

In addition to dealing with numeric data, programs can also work with **characters** and **string data**. A string is a sequence of characters. Java has another type of data called "String" and we can have statements such as the following:

```
String s, t;
s = "Hello";
t = s + " there!";
```

Here we are creating two variables called "s" and "t". In the second line, we assign the string of characters "Hello" to the variable "s".

Strings are enclosed in double quotes, like this:

```
"Hello"
```

while individual characters are enclosed in single quotes:

```
'H'
```

The plus sign (+) is used for **string concatenation** as well as numerical addition. String concatenation is an operation which takes two strings and glues them together to create a larger string. Thus, the assignment

```
t = s + " there!";
```

will set “t” to

```
"Hello there!"
```

Given values for the variables “f” and “c”, we see that the expression

```
f + " F is " + c + " C\n"
```

uses string concatenation to build the string we want to print. This string will end with the newline character, \n, so each time we print, the output will be on a different line.

Since we are working with strings, the “+” will be interpreted to mean string concatenation and not numerical addition. This occurs so much that there is a name for it: **overloading**. We say that the symbol “+” is overloaded with two meanings. Even though the “+” symbol has more than one meaning, the program as a whole is **unambiguous**, which means that there is no confusion about what the computer should do when the program is executed.

The following code in the program:

```
System.out.print (...String...);
```

tells the computer to print a line. There are many ways that a program can produce output. For example, it can open up a window, or make a noise, or print on the printer. In this case, we want the default output which, in Java, is referred to as “System.out”. This means to send the characters to wherever the user has instructed. Generally, these characters would appear on the screen but they could also be written out to a file.

Some programs are necessarily tied to specific input/output devices while other programs are written in a way such that **input / output redirection** is possible. Programs that take ASCII characters as their input and produce ASCII characters as their output are extremely useful and are called **filters**. The input can come either from the keyboard or from a plain-text file or even from another program! Their output can be directed either to the screen or to a plain-text file or to another program.

We can chain several filter programs together to make a **pipe**:

Keyboard □ Program □ Program □ ... □ Program □ Display

The output of one program is fed as input to the next program in the chain. The idea is that each program in the pipe processes the data a little bit. This allows complex functionality to be built from small pieces. The UNIX operating system makes heavy use of program pipes.

In Java, to read a value into a variable, we can use a statement like this:

```
x = System.in.readInt ();
```

[Experienced Java programmers will recognize that I am simplifying things for the sake of clarity and exposition.]

In the following code sequence, we are instructing the computer to get a value and then test it to see if it is positive:

```
int x;
x = System.in.readInt ();
if (x > 0) {
    System.out.print ("This is a positive number.");
}
```

This code contains an **if statement**, which has this general format:

`if (` *...conditional expression...* `) {` *...statements...* `}`

Like the while statement, the if statement is compound. The idea is that the conditional expression is first evaluated. If it is true, then the statements between the braces are executed. Otherwise, these statements are not executed.

The output of this program will depend on which number is given as input. If it is greater than zero, this program will print “This is a positive number.” If it is not, the print statement will be skipped.

There is a second form of the if statement, which is exemplified by this code:

```
if (x > 0) {
    System.out.print ("This is a positive number.");
} else {
    System.out.print ("This is not a positive number.");
}
```

The “if” and “else” are both keywords and they are often used in conjunction with braces { and }. We also saw braces in the while statement. In this example, we had only one statement in the **then-part** and one statement in the **else-part**, but we could have had several.

In the following code, we also set the variable "abs" to the absolute value of "x", in addition to printing a message:

```
if (x > 0) {
    System.out.print ("This is a positive number.");
    abs = x;
} else {
    System.out.print ("This is not a positive number.");
    abs = -x;
}
```

In general, braces are used to bracket or enclose a sequence of statements.

```
{
    Statement;
    Statement;
    Statement;
    ...
    Statement;
}
```

The braces tell the compiler exactly which statements are to be executed if the condition is true or which statements comprise the body of a loop statement.

As a matter of style and convention, the programmer will place each statement on a separate line. When the statements are to be executed sequentially, they should be indented by equal amounts. When a compound statement (like while or if) is used, the statements inside the compound statement will be indented and the braces will be lined up as we have done in our examples. This is not required by the compiler, but good **programming style** is necessary if a large program is to be comprehensible to humans.

From a theoretical perspective, it has been shown that the aspects of programs we have discussed so far are sufficient for writing *any* program. In other words, any programming language that includes the following features:

- integer variables
- assignment statements
- arithmetic operators like + and <
- while statements
- input / output statements
- sequentially arranged statements

is **Turing complete**. Any function that can be programmed on a digital computer can, in theory, be expressed as a program in any Turing complete language, although there is no guarantee that the program will be short or easy to understand.

From a practical point of view, many additional features have been added to programming languages to make them easy and convenient to use. Although these features add no increased theoretical capabilities, they make a huge difference in the usefulness of the language.

Turing completeness should not be confused with the **Turing Test**, which is a test to determine whether a computer exhibits human-level intelligence. To date, no computer has passed this famous test, but this society-shattering event may occur in our lifetimes, particularly if new and better programming languages have features that make it convenient and easy for programmers to create complex programs.

One of the most important conveniences is the ability to group a number of statements together and place them in a **function**. Here is an example, in Java:

```
1.      void DoTheWork (double x) {
2.          double y;
3.          y = (x – 32) * 5 / 9;
4.          System.out.print (x + " F is " + y + " C\n");
5.      }
```

In the first line, we are saying that we are defining a function, which we have named “DoTheWork”. A function is a little like a compound statement and we see matching braces and several indented statements in the **function body**.

The idea with a function is that we can **call the function** from elsewhere in the program. For example, we can re-write our program as:

```
6.      // Print a table of temperatures
7.      double f;
8.      f = 0;
9.      while (f <= 100) {
10.           DoTheWork (f);
11.           f = f + 1;
12.     }
```

In line 10, the function is called. This **call statement** is a new kind of a statement: it is a statement whose behavior the programmer has determined. To know how it will be executed, you must look at the function definition.

The while loop is still executed the same way: the body is executed repeatedly and, for each iteration, the function is called. Each time the function is called, it is executed with a different value of “f”.

The first line of the function is called the **function header** and in it we not only give the name of the function, but we also indicate that this function will take as its input a single floating-point value and will return no value.

Functions that do not return values are called **void functions**. Functions that return a value are said to be non-void functions. "DoTheWork" is a void function, but many functions return a value.

A function in a programming language is similar to a mathematical function. A mathematical function generally takes a single number, performs some computation, and results in some value. For example, we might define a function "g" as:

$$g(x) = (x-32)\cdot(5/9)$$

This mathematical function performs the conversion from Fahrenheit to Celsius, but it works with all numbers. Our function "DoTheWork" is restricted in the range of numbers it can handle correctly: it can handle all temperatures anyone can reasonable expect to encounter, but the point is that it cannot handle all values.

Also, "g" returns a value, whereas "DoTheWork" is a void function. It is used not for its resulting value, but it is used because it has a **side-effect**. It does more than just compute a value; it also prints a line on the output. A side-effect means that executing this function has more of an effect than just computing and returning a result. A function with a side-effect may print output or alter variables that are used in other parts of the program.

In general, the syntax of a function definition is:

ResultType FunctionName (...parameters...) { ... declarations and statements... }

The **result type** in our case is "void" and the function name is "DoTheWork". If we wish to define a function that will compute and return the value (without printing it), we could define the function as follows. We'll call it "g", since it models the mathematical function given above.

```
double g (double x) {
    double y;
    y = (x - 32) * 5 / 9;
    return y;
}
```

The first line is the header line and it indicates that "g" will return a double value. In Java, the type of the resulting value is written right before the function name.

Next, between the parentheses, we see “double x”, which means that the function will take a single value as an input. This value will be a double (a floating-point number) and will be called “x” within the function. Between the braces, we have what amounts to a small program. These lines are the function body.

The inputs to a function are called its **parameters**. This function takes a single parameter called “x”. Within the function body, we may use the parameters. Here, we see that “x” is used in the computation.

In the body of function “g”, we also see a new kind of a statement, the **return statement**. This statement is only used within functions. It indicates what value to return. In this example, we return the value of the variable “y”.

Given this new function, we will need to modify our main program.

```
double f, c;
f = 0;
while (f <= 100) {
      c = g(f);
      System.out.print (f + " F is " + c + " C\n");
      f = f + 1;
}
```

Here we see the assignment statement:

```
c = g(f);
```

The function is called and the value it returns is then stored into the variable “c”. Non-void functions may be used within any expression. In the next example, we use “g” within the conditional expression of an if statement:

```
if (g(f) >= 100) {
    System.out.print ("boiling hot");
}
```

A function may take zero or more parameters. In our functions “DoTheWork” and “g”, we had a single parameter called “x”. Each parameter will have a name and a type. Some functions may have no parameters. For example:

```
double foo () {
    ... Declarations and Statements ...
}
```

Some functions will take many parameters. This function takes 3 parameters:

```
double bar (int x, double y, int z) {
      ... Declarations and Statements ...
}
```

Here are example statements which call these functions with the appropriate number of values:

```
a = foo ();
b = bar (7, 3.1415, 48);
```

We could also call these functions providing values that will be computed at the time the function is called. In the following statement, which calls “bar”, we see 3 expressions used as values to the function. The second value even involves calling another function!

```
b = bar (a, x*g(y), ((i+j)*5)+k);
```

The names **foo** and **bar** are traditionally used by computer scientists in examples to stand for “any made-up name.” Of course a real program would use more meaningful function names like:

```
ComputeAdjustedGrossIncome (...)
Fahrenheit2Celsius (...)
```

Our program now has two parts: the main part and a function called “DoTheWork”. In general, the ability to break a program into smaller, separable parts is good. It allows the programmer to work on the parts in isolation.

In this example, our function is called from only one place, but the real leverage occurs when a function is called from several different places. In our example, we could either use a function or we could simply include the function body directly into the while statement. Perhaps breaking the program into pieces will make it clearer, but it doesn’t give us any real advantage in this example.

However, in many programs, a function is called from several places in the code. If this is the case, then the use of functions becomes critical. Without functions, we would have to copy the same statements into several places in the program.

For example, if our program performs Fahrenheit to Celsius conversions in several places, it is much better to place this computation in a function and then call this function from all those places. Later, if we find a bug in our computation, we only need to modify one place in the program: the function. If we had not used a function and had copied the statements in several places, then we would have to fix the same mistake several places in the program. We might easily make the fix in several places correctly, but forget or mess-up some of the other places. Without functions, we would also need to test each place independently.

Not only is a program without functions much longer, it is also correspondingly more difficult to create, understand, and debug.

Let's now look at a function from mathematics: the "factorial" function. Recall that this function is defined as

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n, \text{ for any } n \geq 1$$

For example:

$$\begin{aligned} 6! &= 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \\ &= 720 \end{aligned}$$

We will use the functional notation of programming, instead of the traditional ! notation from math.

$$\text{fact}(n) = n!$$

For example:

$$\begin{aligned} \text{fact}(6) &= 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \\ &= 720 \end{aligned}$$

Suppose we wish to create a function in Java which will be passed an integer "n" and which will return "fact(n)". We might notice the repetition involved in this definition and write the following function:

```
// Function to compute factorial
int fact (int n) {
    int result = 1;
    int i = 1;
    while (i <= n) {
        result = result * i;
        i = i + 1;
    }
    return result;
}
```

However, you may be familiar with another equivalent definition of factorial:

$$\begin{aligned}&\text{fact}(1) = 1\\&\text{fact}(n) = n \cdot \text{fact}(n - 1), \text{ for any } n > 1\end{aligned}$$

For example, to compute fact(7), we can use the definition to write:

$$\text{fact}(7) = 7 \cdot \text{fact}(6)$$

Since we know fact(6) is 720, we can conclude:

$$\begin{aligned}\text{fact}(7) &= 7 \cdot 720\\&= 5040\end{aligned}$$

The second definition of factorial is a **recursive definition**, since factorial is defined in terms of itself. At first, it may seem circular to define something in terms of itself, but it works in this case, since the factorial of each number is defined only in terms of the factorial of smaller numbers. Eventually, the first part of the definition—the definition of fact(1)—can be used.

The recursive definition of factorial leads to a recursive function, which is a function that, during its execution, may call itself. Here is a second, recursive version of fact:

```
1.     // Function to compute factorial, recursive version
2.     int fact (int n) {
3.         if (n == 1) {
4.             return 1;
5.         } else {
6.             return n * fact (n-1);
7.         }
8.     }
```

The "==" sign is used to test whether two numbers are the same. It is often a source of confusion that "==" in a program corresponds closely to "=" in math, while "=" in a program is used for data movement in the assignment statement.

This recursive function will test whether it is asked to compute fact(1); if so, an answer of 1 is immediately returned in line 4. If the value of the parameter is not 1, then we will execute the return statement in line 6.

Let us assume we are trying to evaluate fact(7). In line 6, the computer will subtract 1 from "n" to get 6. (Of course, "n" will not itself be changed.) Then, the fact function will be called a second time. We say that fact **calls itself recursively**, with the value 6. Assume for a moment that the function fact works correctly; then this recursive call will do some computations culminating in returning a value of 720. Finally, in line 6, the computer will multiply this result (720) by "n" (which is 7). This results in 5040, which will be returned.

One point of confusion is that while executing fact(7), we will begin executing the fact function again—for fact(6)—from the beginning. At this point, we will have 2 separate executions of fact occurring simultaneously. We say that there are multiple **invocations** of fact. One invocation will be computing fact(7) and the other will be computing fact(6).

The invocation for fact(7) will be suspended in the middle of line 6 when the invocation for fact(6) begins and will remain suspended until the computation of fact(6) has been completed. The new invocation—of fact(6)—will begin on line 1 and proceed to line 6. At this point, things get even more complex, since fact(6) will call fact(5) in line 6.

Now we have three invocations of the function. Each of these different invocations of fact will be separate. Each will have its own copy of "n". The invocations of fact(7) and fact(6) will be suspended, and the invocation of fact(5) will begin execution.

This process of recursive invocations will continue until the invocation of fact(1). Fact(1) will not invoke itself recursively. Instead, fact(1) will compute and return its result (namely, 1) immediately. As each invocation completes and returns, a suspended invocation is reawakened. Ultimately, fact(6) will return and fact(7) can complete its computation and return our final answer.

Recursive functions are difficult for everyone to grasp at first, but after a little practice this style of programming turns out to be very useful. Many problems have parts that interact and the programs to solve them tend very naturally to be recursive.

We have seen a number of conditional expressions so far. For example:

```
n == 1
f <= 100
```

We have also seen a number of numeric expressions, like these:

```
result * i
(x – 32) * 5 / 9
```

Both seem to involve variables, constant values, and operators, yet one results in numeric values and the other results in... what?

In addition to integers and floating-point numbers, programs can work with **logical values**. There are a lot of integers and a lot of floating-point values, but there are only two logical values: **true** and **false**. Logical values are also called **boolean values**, after a mathematician named George Boole.

In a program we often use variables which contain boolean values instead of numeric values. For example:

```
boolean b;
...
b = (f <= 100);
```

Here, we create a variable called "b" and, in a later assignment, we set it to either true or false. The assignment is analogous to this assignment:

```
x = (f + 100);
```

The difference is that the + operator returns a number while the <= operator returns a boolean value. It either returns the true value or the false value, depending on the value of "f" when the <= is evaluated.

Note that we used "..." to skip over some statements; we are assuming that we also have statements to set "f" to some value, but these are not shown. Otherwise, the comparison of "f" to 100 would be meaningless.

Once we have executed statements to set "b" to some value, we can use that value in subsequent statements. Perhaps we will wish to copy the value into some other variable:

```
boolean b2;
...
b2 = b;
```

We can use the boolean value directly in an "if" statement, like this:

```
if (b) {
    ... Some statements ...
} else {
    ... Other statements ...
}
```

There are several **logical operators** we can use to compute with logical values. For example, assume that we have two boolean variables, "b" and "c", and that we have given them logical values:

```
boolean b, c;
b = ...;
c = ...;
```

Assume we wish to test whether both "b" and "c" are true; we can use the **logical "and" operation**. The "and" operator is symbolized using &&.

For example, assume that "d" is another boolean variable. Then the following statement will set "d" to true only if both "b" and "c" are true. It will set "d" to false in all other cases:

```
d = b && c;
```

As another example, assume that we have two variables "x" and "y". We wish to print a message if they are both positive. This code will do the trick:

```
if ((x > 0) && (y > 0)) {
    System.out.print ("Both are positive");
}
```

The evaluation of the conditional expression in this example involved the evaluation of two **sub-expressions**:

```
(x > 0)
(y > 0)
```

Now let's assume we want to print the message whenever either "x" or "y" is positive. We can use the **logical "or" operation**, which is symbolized with ||. This code will work:

```
if ((x > 0) || (y > 0)) {
    System.out.print ("At least one is positive");
}
```

What exactly does the "or" operator do? We can answer that by writing the following table. Here, "a" and "b" represent arbitrary logical values, and "a || b" is the result. There are 4 possible combinations of values for "a" and "b". In this table, we simply list all possible values for "a" and "b" and show the result of "a || b".

a	b	a || b
false	false	false
false	true	true
true	false	true
true	true	true

Here is a table showing exactly how the logical "and" operator works:

a	b	a && b
false	false	false
false	true	false
true	false	false
true	true	true

There is also a **logical "not" operator**, which is sometimes referred to as **logical negation**. This operator takes a single logical value and flips it. Logical negation is indicated with the ! symbol. Here is a table giving all (two) values for "a" and showing the value of "!a", which is read "not a":

a	! a
false	true
true	false

We could re-write the previous code, which printed a message if either "x" or "y" is positive, as follows:

```
if (!((x <= 0) && (y <= 0))) {
    System.out.print ("At least one is positive");
}
```

It may not be immediately clear that this code does the same thing, but notice that

```
(x <= 0) && (y <= 0)
```

is true only if both "x" and "y" are non-positive. Its opposite, which is achieved with the logical negation, is true if either is positive. This is exactly when we want to print the message.

After practice, programmers become pretty good at writing and understanding such complex logical expressions, but as you can guess, this is a source of many **logical errors** in programs.

Next, assume that we would like to write a program to read in 100 numbers and print them out in reverse order. Clearly our program will need to remember all the numbers before the first one is printed. Perhaps we could code this with 100 variables, but there is a much better way.

We can store a large collection of values in an **array**. An array can be thought of as a sequence of variables, which we can refer to by number, rather than by name. In our program, we will use a single variable called "a", which will be an array of 100 values.

First, we need to declare the variable:

```
int [] a;
```

The brackets indicate that this variable will hold not just a single integer, but an array of integers.

Next, we need to create the array. In Java, we use this code:

```
a = new int [100];
```

Since each integer requires 4 bytes, this array will require 400 bytes. The above statement will request 400 bytes of memory to use for "a" and will set all 100 values to zero.

The individual components of the array are referred to as the **array elements**. Our array will have 100 elements.

We can refer to any individual element of the array using brackets. For example, we can ask what is stored in element number 15 by writing the following in our program:

```
a[15]
```

We might want to add 8 to element number 23. This assignment will do that:

```
a[23] = a[23] + 8;
```

The elements in the array are numbered starting from 0, not from 1. Thus, our array contains the following 100 elements:

```
a[0], a[1], a[2], ... , a[99]
```

Starting the numbering at zero may seem confusing but there are technical reasons why most programming languages do it this way.

Now we can present the program to read in 100 numbers and print them in reverse order:

```
// Declare variables and create the array.
int [] a;
int i;
a = new int [100];

// Read in 100 values and store them in the array.
i = 0;
while (i < 100) {
    a[i] = System.in.readInt ();
    i = i + 1;
}

// Write out the 100 values in reverse order.
i = 99;
while (i >= 0) {
    System.out.println  (a[i]);
    i = i - 1;
}
```

In this program, the number 100 was predetermined. If we want to reverse a list of 200 numbers, we will need to modify the program and re-compile it.

A different approach would be to make the program flexible. Let’s have the program begin by reading in a single number, “n”. It will then create an array of size “n” and proceed as before.

Here is the modified program:

```
// Declare variables.
int [] a;
int i, n;

// Read "n" and create an array of that size.
n = System.in.readInt ();
a = new int [n];

// Read in "n" values and store them in the array.
i = 0;
while (i < n) {
    a[i] = System.in.readInt ();
    i = i + 1;
}

// Write out the "n" values in reverse order.
i = n-1;
while (i >= 0) {
    System.out.println (a[i]);
    i = i - 1;
}
```

Next, let’s create some code to read in 100 values, sort them into increasing order and then print them. You might suggest that we should begin by reading “n” in order to make our program more flexible and more useful. This is a good idea, but we'll use 100 nevertheless, to keep the code simpler.

We'll start with a sketch of our algorithm, which we'll express in **pseudo-code**. Pseudo-code is not legal code. It may contain anything and can be written on the back of a napkin, but writing pseudo-code is usually the first step in the programming process.

```
Read in the array.
Sort the array.
Print out the array.
```

This is a good start, but we need to be a little more specific in the middle step.

```
Read in the array.
In a loop, for each element a[i] in the array...
   Locate the smallest element of a[i], a[i+1], ..., a[99].
   Put that element in position a[i].
Print out the array.
```

Let's elaborate this a little more:

```
Read in the array.
In a loop, let i take the successive values 0, 1, 2, 3, ..., 99 in the following...
   Find the smallest element between a[i] and a[99]. Let k be its position.
      Start by assuming a[i] is the smallest.
      Look at all elements between a[i+1] and a[99].
      See if any of them is smaller
   Swap the values of a[i] and a[k].
Print out the array.
```

We are now ready to write the final program:

```
int [] a;
int i, j, k, t;
a = new int [100];

// Read in the array.
i = 0;
while (i < 100) {
    a[i] = System.in.readInt ();
    i = i + 1;
}

// Sort the array.
i = 0;
while (i <= 98) {
    // Set k to the position of the smallest element
    //    of a[i]...a[99]
    k = i;     // Assume smallest is in position k=i.
    j = i+1;   // Run thru elements i+1 thru 99.
    while (j <= 99) {
        // if a[j] is smaller, then set k to its position.
        if (a[k] > a[j]) {
            k = j;
        }
        j = j + 1;
    }
    // Swap a[i] and a[k].
    t = a[i];
    a[i] = a[k];
    a[k] = t;
    i = i + 1;
}

// Write out the array.
i = 0;
while (i < 100) {
    System.out.println (a[i]);
    i = i + 1;
}
```

Note that this program contain several loops. In fact, one of the loops contains an **inner loop** within it. Loops can be nested and careful indentation becomes important in understanding such **nested loop** structures.

This program may require some study to understand. There are many ways it could be changed and improved; we make no claim this program is best.

This version of sort uses an algorithm called selection sort. There are many **sorting algorithms** which will run in less time than this algorithm, but they can be much more difficult to understand.

Execution speed is a measure of how long a program will take to run. In general, the time a program takes to run will depend on the input it is given. The general rule is, more input means the program will take longer to execute.

Our sorting program will do more work when the array is in reverse order to begin with, but the most critical factor is the size of the array we are sorting. When sorting an array of 100 items, execution speed is not so important, but when sorting millions of numbers, the performance of this algorithm will become a huge drawback. We'll need to use a faster sorting algorithm.

In many applications, **program performance** is an important engineering consideration. In a few simple, well-studied algorithms (such as sorting algorithms) a program's performance can be pre-computed analytically, but with most real-world programs the analysis is hopelessly complex. It is often easier to run the program on some sample inputs and measure the performance empirically.

Often, beginning programmers write programs that function correctly but have poor performance. Another problem occurs when programmers focus on performance too much, to the detriment of correctness. The programmer may choose an overly complex algorithm and fail to get it quite right. What is the value of a program that runs fast but that fails to perform correctly for some inputs?

As our next example task, let's think about writing a program to keep track of student records and university course information. Our primary focus will be on how the data is represented in the program and we will be less concerned with what we do with the data. Perhaps we will write code to print out which students are enrolled in which courses or code to compute each student's grade-point average (GPA). Either of these tasks will be possible once we have the data stored in the computer's memory. Here, we'll focus on the data representation.

For a large collection of data, a real organization might reasonably purchase some **database management software**. Such a software package will facilitate additional tasks like disk organization, data security, backup procedures, and simultaneous access by multiple users. For our purposes though, we'll store our data in the computer's main memory, not on disk. We will ignore many of the complex tasks handled by a database management system.

So far, we have discussed simple variables, which can hold a single value at a time, and arrays, which can hold a numbered collection of values. However, for this task we will use objects and pointers.

When thinking about our task, we notice that we need to store a number of pieces of data for each student. Let's assume that we keep the following information for each student:

Student info:
- student id number
- name
- address
- courses currently enrolled in

For each course, we'll keep the following information:

Course info:
- course name
- course number
- instructor's name
- meeting time
- room number

We could, of course, add more information, such as the number of credit hours for a course or the students' phone numbers or grades. Deciding exactly what information to represent and capture in the computer is the task of **data modeling**. For the sake of simplicity, we'll ignore this additional information.

In our program, we will create an **object** for each student. We'll use the name "Student" to describe these objects. If we have data for 100 students, we'll create 100 "Student" objects.

In our Java program, we'll include the following code to describe the Student objects:

```
class Student {
   int id;
   String name;
   String addr;
   Course [] courseList;
}
```

The word **class** is a keyword signifying that we are defining a new class of objects. After the keyword, we give the name we have made up for these objects: "Student".

Between the braces, we see a sequence of variable declarations. The first is:

```
int id;
```

We have decided to use an integer for the student's id number and we have used the name "id" for this piece of data.

For the student's name and address, we will use "name" and "addr". Each of these will be a String.

```
String name;
String addr;
```

Finally, we will use an array to store information about which courses a student is taking, since a student will usually take more than one course. We will call this array "courseList".

```
Course [] courseList;
```

Next, let's define a class called "Course". Each Course object will describe a course being taught:

```
class Course {
   String courseName;
   int courseNumber;
   String instructor;
   String time;
   String room;
}
```

We'll use Strings for the time and room number, since they may look like this:

```
"M/W 2:00"
"Lincoln Hall 205"
```

Let's assume that our program will read in information about the students and courses from the input. We'll ignore details, but for every student, our program must create a "Student" object, and for every course, we must create a "Course" object.

Let's begin with courses. The following Java statement will create a new "Course" object, which we'll store in a variable called "c".

```
c = new Course ("Physics",
                100,
                "J. Brown",
                "M/W 2:00",
                "Lincoln Hall 205");
```

This looks similar to the creation of an array, except that we must give values to use for "courseName", "courseNumber", etc.

When this statement is executed, what happens? A new object will be created in the computer's memory. The actual size of the object will depend on how much data it includes. Let's assume that this object takes (say) 100 bytes. So, 100 bytes will be used to store the information associated with this course.

Each byte in the computer's memory is numbered. One can ask what is stored in byte number 0, or what is stored in byte number 1, or byte number 2, and so on, up to the size of the memory on the machine.

Where will our new Course object be stored? Perhaps it will be stored in bytes 2001 through 2100. The operating system takes care of **memory management** and will "give" bytes to the program as needed. The OS will decide exactly which 100 bytes will be used to store this object. The programmer will not have to worry with this detail.

Let's assume that we have a total of 20 courses; we can use an array to hold these courses. Here is code to declare our array variable, which we'll call "allCourses", and code to create the array,

```
Course [] allCourses;
allCourses = new Course [20];
```

We will need code to read in the course information and create 20 Course objects. Here is some pseudo-code to do this.

```
Repeat 20 times {
    Read in the information about course "i".
    allCourses [i] = new Course (...the info for this course....)
}
```

The array "allCourses" has 20 elements. If each course takes 100 bytes, you might guess that this array will require

$$20 \cdot 100 = 2000 \text{ bytes}$$

but this is not the way it works. Each array element will not require the full 100 bytes to store the course information. Instead each element of the array will be only large enough to contain the address of the Course object. The Course object itself will be located elsewhere in memory.

Addresses are called **pointers**, so our array will contain 20 pointers. Our array is very much like an array of integers, where each integer is the address of a byte in memory. Memory addresses are 4 bytes long (just like integers) so our array will consume

$$20 \cdot 4 = 80 \text{ bytes.}$$

Of course we still need 2000 bytes to store the Course objects so we haven't saved any memory with pointers.

Looking back at the statement which created a new Course object,

```
   c = new Course ("Physics", ...);
```

we see that this statement will first allocate 100 bytes of memory, then move the initial data values into those bytes and, finally, store a pointer to the object into variable "c". The variable itself will only be 4 bytes in size.

Next, let's write some code to read in and create the Student objects. For simplicity, let's assume that each student will enroll in exactly 5 courses. We can create a new Student object with code like the following:

```
   id = ... Read in Student's id number ...
   nm = ... Read in Student's name ...
   addr = ... Read in Student's address ...
   a = new Course [5];
   ... Read in which 5 courses and initialize "a" accordingly...
   st = new Student (id, nm, addr, a);
```

In this code, we are using several variables ("id", "nm", "addr", "a" and "st") so we would need declarations of them somewhere else. The result of this code is to create a new Student object which we will refer to using the variable "st".

Notice that this code will create a new array with 5 elements. We are setting variable "a" to this new array. The new array will contain 5 pointers and each should be set to point to a Course object. The code to initialize this array may be rather complex and we won't go into details, but it will end by setting each element to point to one of the 20 Course elements.

Recall that we are assuming the Course objects are each 100 bytes in size. Without pointers, each Student object would require

$$5 \cdot 100 = 500 \text{ bytes}$$

for the course information, plus additional bytes for name, address and id number. With pointers, each Student object uses only

$$5 \cdot 4 = 20 \text{ bytes}$$

for the course information. This saving in memory space is real, since we have already created the Course objects.

Notice that many students will take the same course. Thus, we will have only one Course object (100 bytes) and all the Student objects will point to this shared object. If 50 students are taking "Physics" then, to store this information, we will need:

For the "Physics" Course object:
  $1 \cdot 100 = 100$ bytes
For one pointer to this object in each of 50 Student objects:
  $50 \cdot 4 = 200$ bytes
Total:
  300 bytes

The real benefit of using pointers is not that we save a lot of memory space, but that we represent each course with just one object. Each course has a "name", "instructor", "time", etc. This information is represented and stored in only one place. If there is a change to the data (for example, if the instructor is changed to "K. Smith"), then the data must be changed in only one place. Since the Course object is pointed to by many Student objects, all of these Student objects will effectively use the modified data.

Without pointers, changing the name of the instructor would require that we update the records of all students who are taking "Physics". While this might not seem too complex, in real-programs it becomes a nightmare of complexity. And if the data is updated incorrectly, you end up with some people taking the identical "Physics" class from different instructors, a nonsensical situation.

With pointers, we can also represent missing information in a natural way. For example, some students may be registered for fewer than 5 classes. In each Student object, we included an array called "courseList". We must store something in each element of this array, but we don't want to store a pointer to an object if the student is not taking a course.

There is a special pointer value called **null**. This special value can be stored into a variable when the actual data is missing. For example, if some Student object is missing course information, then its "courseList" will contain an element whose value is null.

Normally, each pointer is simply the address of some location in memory. For the null value, a zero is stored in the variable. It turns out that no object will ever be stored at address zero, so this special value can be easily distinguished from a real memory address.

With pointers, a whole collection of **pointer bugs** becomes possible. Let's imagine that, due to a program bug, some pointer variable contains the wrong address. Perhaps we have a variable that should point to a Course object, but which unfortunately points to a Student object. Or, it could be even worse. Perhaps the variable contains a more or less random number. What will happen when this pointer is used?

As the program executes, pointers are used from time to time. When a pointer is used, the execution will go to the memory bytes addressed by the pointer. If some variable is supposed to point to a Course object, then the program will assume the memory contains a Course object at that location. If there is a bug and this is not the case, then we have a serious problem.

As an example, assume that the program is trying to update a Course to change the instructor to (say) "K. Smith". This might be a normal operation allowed by this program to support the fact that schedule updates and corrections occur from time to time. But if there is a bug and the pointer is messed up, then the program will proceed anyway. It will follow the pointer to some memory location and will modify the bytes at that location. There is no checking; the memory is blindly updated and altered.

What is the consequence of such a bug? Unfortunately, it is impossible to say. We would need to know exactly which address got modified, what data was stored there before and what new data got written into memory. Generally, we cannot know these things, so the program exhibits **unpredictable behavior**. While exactly what happens next is difficult to predict, it is almost always bad.

Furthermore, since these things can change from run to run, pointer bugs often cause a program to have **non-repeatable** behavior, in which the program may behave correctly sometimes, but incorrectly other times. Anyone who has tried to fix a non-repeatable problem knows how difficult they can be to diagnose and repair.

A program with a bug will always do one of several things:

- Go into an infinite loop
- Attempt to do something nonsensical, causing abrupt program termination
- Produce obviously wrong output
- Produce wrong output that is mistaken for being correct

The last is perhaps the worst outcome! In our society, we are increasingly relying on computers and our actions are often determined by "what the computer says". Often humans accept computer output without questioning it and, when the computer says the wrong thing, the results may be catastrophic and lethal.

But more often, a pointer bug and the resulting damage will confuse the computer so much that the program will **crash** altogether. We also say the program **bombs** or **core-dumps**. A crash occurs when the operating system detects that the program has become confused and tried to do something illegal or nonsensical. For example, a program may try to access a region of memory that (for security reasons) it is not allowed to access or it might try to divide a number by zero.

When the OS detects such a violation, it will immediately terminate the program. We say that the program is aborted or killed. The OS may also display a cryptic **system error code** or message. These messages are meaningful only to people who understand machine code and processor architecture, so we can't discuss them here. In fact, they will often be meaningless even to the programmer, who may have difficulty determining exactly why the program crashed. The problem is that a bug may cause unpredictable or non-repeatable behavior, which subsequently causes the program to crash. Consequently, the error that causes program termination is not directly related to the actual bug.

The C++ language suffers from this sort of problem quite a lot, but the modern languages like Java go a long way in eliminating such unpredictable behavior. In Java, the use of pointers is tightly constrained by the compiler, in order to make it impossible to make pointer mistakes which can cause a program to have unpredictable behavior. Java programs can still run into problems, of course, since programmers still make mistakes and programs can still contain bugs, but the mistakes are all caught when they occur and are dealt with in a much safer, more predictable way.

Up until now, we have been assuming that there is a single processor, which means that the program is executed by the computer one instruction at a time. Although the instructions may be executed at a rate of a billion instructions per second, the instructions are executed in sequence, one before the next. We have assumed that our program runs to completion (or is killed) and then another program is executed.

As you know, a modern operating system will sometimes load several programs into memory at once and these programs will appear to be active simultaneously. In a typical OS, there will usually be several programs running at any one time, but each program is kept separate from the others and runs quite independently of any other active programs.

Modern operating systems go to great lengths to keep concurrently running programs isolated from each other. Each program is placed in a different region of memory and these regions are separated like islands. Thus, a pointer bug in one program is incapable of modifying or affecting in any way another currently running program. If there is any interaction between programs, it is very limited. For example, two programs might share data through a file.

In a computer with a single processor, only one instruction can be executed at a time, so it would seem impossible to do more than one thing at a time, to execute more than one program at any moment. And, in fact, in the early days of computing, a program would be loaded into the computer's memory, the program would be run until it completed and then the next program would be loaded into memory.

Today, we all use **multi-programmed** computers, in which several programs can be active at the same time. You may be running a word-processing application at the same time another program is fetching your email and a third program is burning a CD-ROM. Often, all these applications seem to be executing at the same time. It appears that there is a processor for each program and that they are all executing instructions simultaneously. But this is an illusion!

When several programs are running simultaneously on a computer with only a single processor, the effect of simultaneous execution is achieved through **time-slicing**. It works like this. The operating system will allow the processor to execute the instructions from Program A for a small duration of time. Then, the OS will stop the processor from working on program A and move it to Program B. The processor will then execute instructions from Program B for a while. Then the OS will interrupt it again and switch the processor to Program C. After each of the programs has had a chance to execute a little, the OS will return to Program A, and go around again giving each program "second helpings" or "second turns" of execution. This **round-robin scheduling** will continue indefinitely, allowing all programs to make progress simultaneously.

Typically, the time-slice is very small (say, 1/1000 of a second). This means that any one program will have several turns to execute every second. To the human, the switching occurs so frequently, that it appears that all programs are executing in a truly simultaneous fashion. Occasionally, the user will be aware of time-slicing when some program takes an unusually long turn at execution. (It is said to "hog the processor.") The result is that all other programs become frozen for a period of time. If the delay is longer than a second, the user may notice a delay in the output of any programs that are frozen.

It is also possible that some program will go into an infinite loop, effectively refusing to give other programs their turns with the processor. It is as if one program suddenly becomes belligerent and refuses to share. In older operating systems, the only solution is to restart the computer, while in more modern systems, the OS will continue with normal time-slicing. A program that is infinitely-looping will either produce no output or will produce an endless stream of repetitive output. Since most infinite loops tend not to produce output, the looping program will generally appear to be **frozen**. In either case, a looping program must be terminated from the outside, but before it is terminated, the other programs will continue to make normal progress in spite of the looping program. The other programs may be slowed down, but their functionality will not be disrupted.

Now let's take a look at a word-processing program. Under normal operation, the user types in characters and they get displayed on the screen. Assume that the program can also check spelling. Typically the task of checking spelling is quite time-consuming since the dictionary contains so many entries that must be consulted. The goal of the spelling checker is to check each word in the document and underline the words that are misspelled. Moreover, it will check the spelling as the user types and will underline misspelled words as they are typed.

The task of checking the spelling is straightforward but time-consuming. Ideally, the application would check the spelling after each character is typed, but the task is so time-consuming that a normal computer cannot keep up with a typical human typing rate. Receiving the characters from the keyboard and displaying them on the screen must have priority and must be done first. How is this program to be structured?

There are essentially two different **tasks**. The first task, which we shall call "key-reading", involves receiving characters, storing them in memory and displaying them. The second task, which we call "checking", will look at the characters stored in memory and, if any words are misspelled, it will display the underlining on the screen.

In this application, we want both tasks to make progress simultaneously. We don't want the "checking" to take too much time away from "key-reading." One solution might be to run both tasks as separate programs: the time-slicing of the operating system would allow both to make progress but, unfortunately, the two tasks would be isolated from each other. This will not work here since both tasks must access some shared objects in memory where the document's characters are stored.

In a simple program, as we have been discussing up to now, each statement is executed one after the other. Imagine looking at a program while it executes and imagine that we could highlight each statement as it is executed. Then, as execution proceeded through the program, one statement after the next would be highlighted. (Of course, this would occur very fast, but imagine that time is slowed down so we can see the different lines of the program light up in turn.) This single flow of control through the program is called the **thread of execution**.

A normal program has a single thread: at any one time, only one statement is highlighted. As execution winds its way through the instructions—taking this branch or that at the "if" statements, repeating statements over and over in "while" loops and jumping to and from functions from time to time—execution "threads" through the program statements.

The solution to our word processing problem is to create a **multi-threaded** program. In a multi-threaded program, the principle of time-slicing is applied within a single program. There is more than one flow of statement execution, more than one thread.

For example, in a program with two threads, two statements will be highlighted, at any one moment. As time goes by, execution will move from each highlighted statement to the following statement, in two separate areas of the program. Each thread will proceed on its merry way, independently of the other thread.

One way to understands multi-threaded programs is to imagine that a program with two threads is executed on a computer with two processors. Each processor implements one of the threads, so it is as if the program is being executed twice on two different computers. However, you must imagine that the two processors share their memory, like two Siamese computers, joined at the memory. There is only one copy of the objects in memory and these objects are shared by both processors. In our example, one processor will execute the "key-reading" task and the other will execute the "checking" task.

With time-slicing, multi-threaded programs can easily be run on computers with only a single processor. In fact, most common application programs, such as word processors, games and web browsers, are multi-threaded applications. Some applications may even have hundreds of threads.

From time to time, it will be necessary to coordinate the different threads. For example, if the "checking" task finishes checking all the words in the document, it will have to suspend execution and wait until the user types in some more words to check. Or perhaps the user hits some keys to save the file and quit the application; then the "key-reading" task will need to tell the "checking" task to stop execution altogether.

Modern programming languages like Java include a number of features to support multiple threads of execution. For example, one thread can communicate to another thread, saying something like, "Please stop and wait until I tell you to start back up" or "Please terminate your execution." One important operation occurs when one thread creates a new thread.

Threads may also be assigned different **priorities**. We might want to perform the "key-reading" task with greater priority than the "checking" task. The reason is that our word processing application should spend more time responding to the keyboard and displaying whatever is typed than checking the spelling, so that it will display the typed characters as quickly as they are typed. The "checking" task is a background task and is less important. The application should only try to find and underline misspelled words when it is not doing anything more important.

After a program is written, it will be compiled and an executable file for some computer will be produced. The executable can be executed on that computer over and over, but cannot be executed on other computers. As you know, a program for the PC will not run on a Mac or on a Sun computer since each of these computers has a different processor.

Furthermore, within some processor lines, there is enough difference between the various models that a program written for one will not run on another. For example, a program written for a newer Intel processor may not run on an older Intel processor, even though there is only a very slight difference between the processors within the Intel line. Generally the newer models are designed to have **backward compatibility**, which means that they can execute all of the instructions that could be executed on the older models, but the newer models also include newer features. Programs that are written for the newer models and that use these newer features will not run on the older processors.

When a program works on one computer but not another, there is a **portability problem**. The program cannot be moved to a different computer. Of course, this is a huge nuisance for computer users. Some corporations (particularly Microsoft) even use the portability problem in an attempt to trap their customers and prevent them from leaving.

Fortunately, computer scientists have found a clever way to create programs that will execute on a large number of computers and eliminate the portability problem, in spite of whatever decisions a manufacturer may make.

The idea is to have the compiler translate the source program, not to the Pentium or the G4 chip, but to a standardized machine, designed primarily to address the portability problem. Such a machine is called a **virtual machine**, since there is no actual chip that executes programs for it.

Let's look at the Java Virtual Machine in particular. There is no "Java Virtual Machine" chip, although the instructions of this machine look similar to instructions for any real microprocessor. In theory, some manufacturer could begin producing chips for it—and this may well happen in the future—but the point is that the Java Virtual Machine is a description of a processor, not a real processor. It is a specification, not a real machine.

Instead, the virtual machine is implemented in software. There is a program which simulates the Java Virtual Machine. This piece of software is called the **interpreter**. There are many versions of this interpreter: there is one for the PC, one for the Mac and one for the Sun computers. Java programs are not compiled for any particular machine; instead there is only one Java compiler and it produces code only for the Java Virtual Machine. Today, every computer comes with an interpreter for the Java Virtual Machine, which means that any Java executable program can be executed on any machine. The user does not need to worry about whether the executable will execute of his or her machine.

Today, most web browsers have interpreters built right into them for several popular languages like Java. When you go to some complex web pages, what really happens is that a compiled Java executable program is downloaded and run on your machine, using the interpreter that is built into your web browser. (Don't worry; the interpreter doesn't allow the Java program to read or write to your disk.) Running a program on your machine allows the web page to have a very complex behavior and to be quite interactive, all without communicating over the internet.

When programs on different computers interact with each other, it is often the case that one program runs on the user's computer and deals with issues related to the user interface; this program is called the **client** program. The other program, which runs on a remote computer and communicates with the client program, is called the **server** program. Generally, the server is concerned with managing and storing the data on some distant disk. A single server will usually communicate simultaneously with hundreds or millions of clients.

For example, the AOL software that many people load onto their computers is a client program which talks with the AOL server. Also, many people use an **email client** to send and retrieve email from a server computer.

As another example, your web browser is a general-purpose client program, which communicates with the millions of webpage servers on the internet. The beauty of being able to download a Java program and execute it on your machine using the Java Virtual Machine is that a particular server can momentarily customize your computer to make it perform in a way compatible with the server. This allows a human user to communicate with a remote computer, using a Java client program to act as the user interface on the user's computer.

The subject of programming is fascinating and it is a very active research area. The design, introduction and study of new languages is a hot and active field, attracting the smartest students.

There are many goals in **programming language research**, but one important goal is to create languages that are easier to use and which reduce programming errors. Another research goal is to create languages and compilers that are faster and more efficient. Also, many new languages are being designed for specialized applications or based on innovative new concepts. And of course it will remain important to understand, debug and generally live with all the **legacy code** which has been written in older programming languages and which runs our existing computers today.