Chapter 1. Introduction to Computing

The electronic computer is one of the most important developments of the twentieth century. Like the industrial revolution of the nineteenth century, the computer and the information and communication technology built upon it have drastically changed business, culture, government and science, and have touched nearly every aspect of our lives. This text introduces the field of computing and details the fundamental concepts and practices used in the development of computer applications.

Entering into a new field like computing is a bit like going to work in a country that you have never visited before. While all countries share some fundamental features such as the need for language and propensities for culture and trade, the profound differences in these features from one country to the next can be disorienting and even debilitating for newcomers. Further, it's difficult to even describe the features of a given country in any definitive way because they vary from place to place and they change over time. In a similar way, entering the field of computing can be disorienting and finding clear definitions of its features can be difficult.

Still, there are fundamental concepts that underlie the field of computing that can be articulated, learned and deployed effectively. All computing is based on the coordinated use of computer devices, called *hardware*, and the computer programs that drive them, called *software*, and all software applications are built using data and process specifications, called *data structures* and *algorithms*. These fundamentals have remained remarkably stable over the history of computing, in spite of the continual advance of the hardware and software technologies, and the continual development of new paradigms for data and process specifications.

This chapter defines the notion of computing, discusses the concepts of hardware and software, and concludes with an introduction to the development of software, called computer *programming*. The remainder of the text focuses in on the development of computer software, providing a detailed discussion of the principles of software as well as a snapshot of the current culture of the software development field. Processing, a Java-based development environment, is used throughout the first half of the text; the text then transitions to use of the full Java development environment.

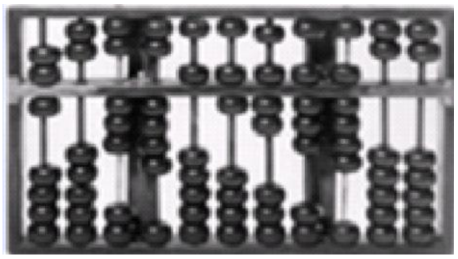
1.1. Computing

As noted above, the definition of computing is hard to pin down, but the *Computing Curricula 2005: The Overview Report* prepared by a joint committee of ACM, IEEE, and AIS gives the following definition: "In a general way, we can define computing to mean any goal-oriented activity requiring, benefiting from, or creating computers." This is a very broad definition that comprises the development of computer hardware, the use of computer applications, and the development of computer software. This text focuses on the last of these enterprises, the development of computer software.

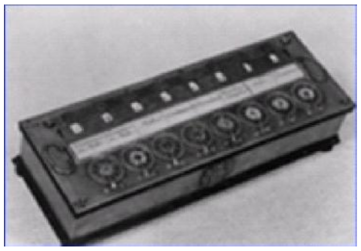
Because software development is based on computer hardware, this chapter will discuss the general nature of computer hardware and its relationship to software as a way to prepare students for software development. The authors hope that this text will engage a new generation of software developers including not only the mathematically and scientifically inclined students commonly found in programming courses, but also a new generation of students from the arts and humanities who are finding that computing is as relevant to their fields as it has ever been in the sciences.

1.2. Hardware

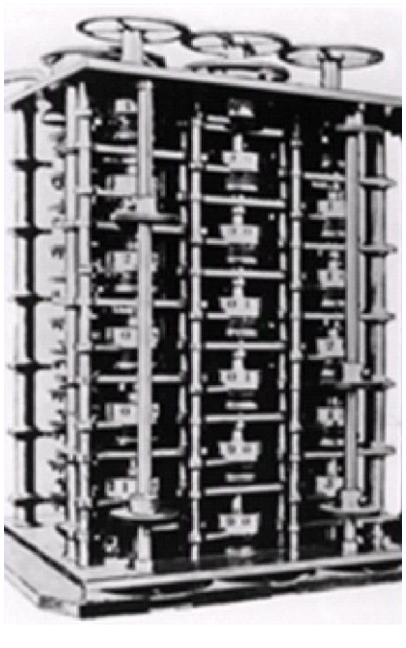
The term computer dates back to the 1600s. However, until the 1950s, the term referred almost exclusively to a human who performed computations. For human beings, the task of performing large amounts of computation is one that is laborious, time consuming, and error prone. Thus, the human desire to mechanize arithmetic is an ancient one.



One of the earliest devices developed for simplifying human arithmetic was the abacus already in use in ancient Mesopotamia, Asian, Indian, Persian, Greco-Roman, and Mezo-American societies and still in use today in many parts of the world. Comprised of an organized collection of beads or stones moved along rods or in grooves, an abacus is, like the modern computer, a "**digital**" arithmetic machine, in that its operations mimic the changes in **digits** that occur when humans do basic arithmetic calculations. However, not all of these abacus systems used *decimal* – base-10 – numerals; some of these societies used base-16, base-20, or base-60 numeral systems.



The young French mathematician **Blaise Pascal** (1623-1662) invented one of the first gear-based adding machines to help with the enormous amount of calculations involved in the computing of taxes. Operationally, the decimal version of the "Pascaline" had much in common with a genre of calculators that were commonly used by grocery store shoppers in the U.S. and elsewhere during the 1950s and 1960s.



In 1822, English mathematician Charles Babbage (1792-1871) unveiled the first phase of his envisioned "Difference Engine" which also used ten-position gears to represent decimal digits. It was capable of performing more complex calculations than the basic arithmetic of an adding machine like the Pascaline. However, the engineering of the Difference Engine became so complicated that, for this and other reasons, Babbage abandoned the project.

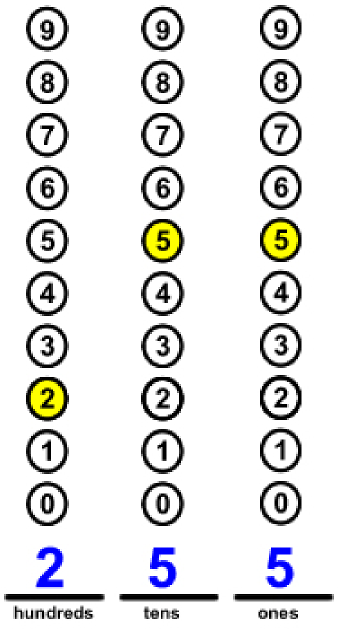
There are two main difficulties here, illustrating two key concepts in computing. First, these devices were "mechanical" – i.e., they were devices that required physically moving and interconnected parts. Such a device is almost certain to be slower, more prone to failure, and more difficult to manufacture than a device that has no moving parts.

In contrast, “electronic” devices such as vacuum tubes of the sort used in early radios have, by definition, no moving parts.

Thus, one of the earliest *electronic* digital computers, the ENIAC represented each decimal digit not with a 10-state *mechanical* device like a gear but, rather, with a column of 10 **vacuum tubes** which could electronically turn on and off to represent the 0-9 counting sequence of a decimal digit without requiring any physical movement.

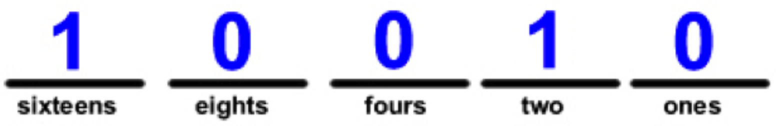
Engineered by **J. Presper Eckert** and **John Mauchly** at the University of Pennsylvania from 1943 to 1946, the 30-ton ENIAC required 18,000 vacuum tubes, consuming enormous amounts of electrical power for its day. This is largely because ENIAC required 10 vacuum tubes to represent each decimal digit.

In contrast, the *first* electronic digital computer developed by **John Atanasoff** and **Clifford Berry** at Iowa State University from 1937-1942, like all electronic digital computers today, used a *binary* – i.e., Base-2 numeral system.

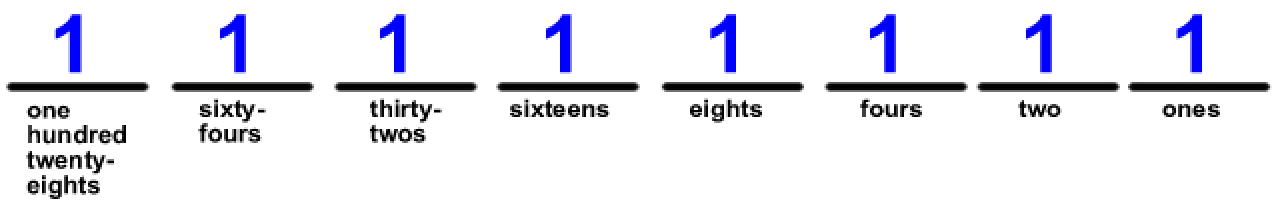


Decimal digits are based on powers of 10, where every digit one moves to the left represents another power of 10: ones (10^0), tens (10^1), hundreds (10^2), thousands (10^3), etc. Thus, the decimal number “**two hundred fifty-five**” is written as “**255**,” conceiving of it arithmetically as the sum of **2** hundreds, **5** tens, and **5** ones. Thus, to store this number, ENIAC would only have to turn on 3 vacuum tubes, but there are still a total of 30 vacuum tubes required just to represent all of the possibilities of these three digits.

On the other hand, **binary digits** – also known as “**bits**” -- are based on powers of **2**, where every digit one moves to the left represents another power of 2: ones (2^0), twos (2^1), fours (10^2), eights (10^3), sixteens (10^4), etc. Thus, in binary, the number **eighteen** would be written in Base-2 as **10010**, understood arithmetically as the sum of **1** sixteen, **0** eights, **0** fours, **1** two, and **0** ones:



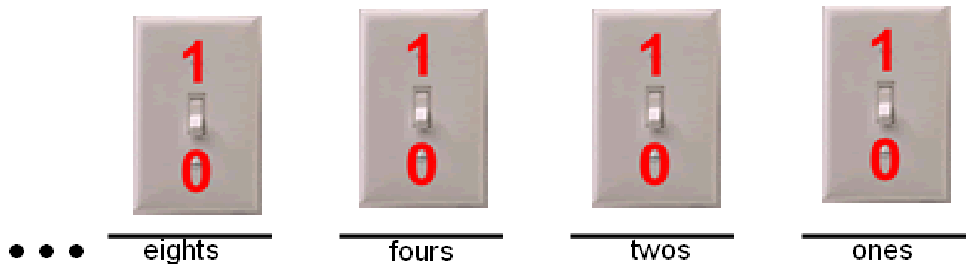
Likewise, the number “**two-hundred fifty-five**” would be written in binary numerals as **11111111**, conceived arithmetically as the sum of **1** one-hundred twenty eight, **1** sixty-four, **1** thirty-two, **1** sixteen, **1** eight, **1** four, **1** two, and **1** one :



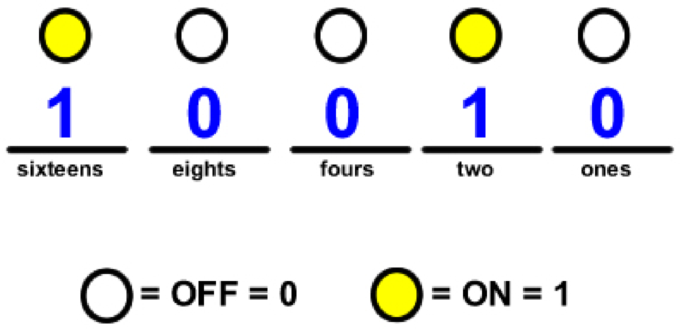
Why on earth would computer engineers choose to build a machine to do arithmetic using such a cryptic, unfamiliar form of writing numbers as a binary, Base-Two numeral scheme? Here's why.

In any digital numeral system, each digit must be able to count up to **one less than** the base. Thus, in the case of the Base-10 system, counting sequence of each decimal digit runs from **0** up to **9**, and then back to 0. To represent a decimal digit, then, one must be able to account for all 10 possibilities in the counting sequence, 0 through 9, so one must either use a device with ten possible states, like the ten-position gear used in the Pascaline, or ten separate devices, like the ten separate vacuum tubes used for each digit in the ENIAC.

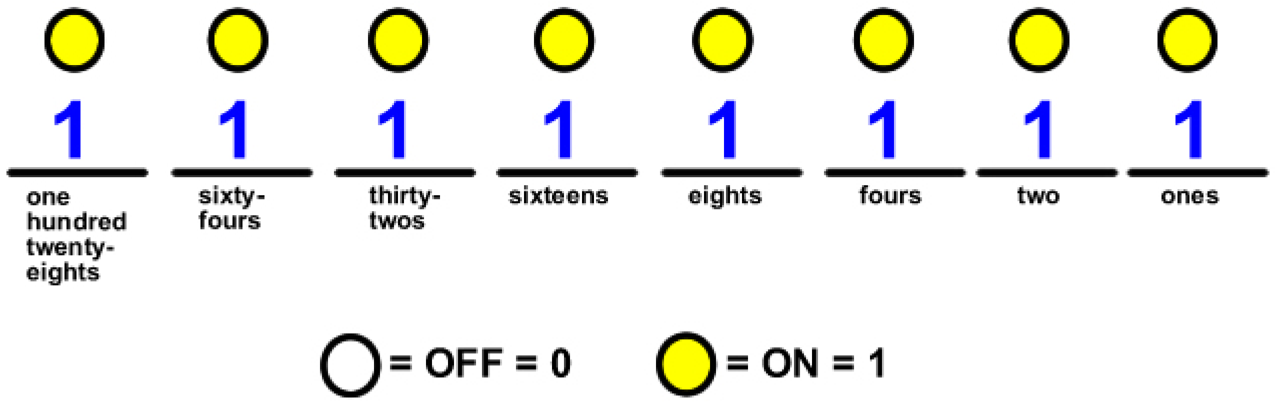
However, the binary numeral system is Base-2. Thus, given that its digits also need only to be able to count as high as one less than the base, this means that the counting sequence of each binary digit runs from **0** only up to **1**, and then back again to 0 already. In other words, whereas ten different numbers can appear in a decimal digit, 0 through 9, the only number that will ever appear in a binary digit is a **0** or a **1**. Thus, rather than having to account for the 10 possibilities of a decimal digit, one can represent a binary digit with only a single device that has two possible states. For example, one could represent each binary digit with a simple on/off switch, where the "**on**" position represents a **1** and the "**off"** position represents a **0**:



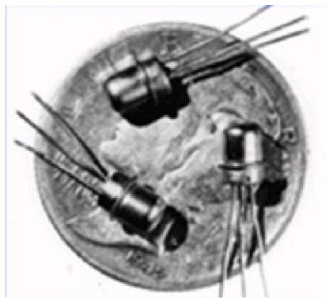
Similarly, in the Atansoff-Berry Computer, each binary digit could be represented with a **single** vacuum tube. Thus, the number "**eighteen**" could be represented with only **5** vacuum tubes, instead of the 20 the ENIAC required:



Likewise, the number "**two hundred fifty-five**" could be represented with only **8** vacuum tubes, instead of the 30 that ENIAC required:



Thus, in exchange for the cryptic unfamiliarity of binary representation, computer engineers gained an efficient way to make electronic digital computers through the use of two-state electronic devices.

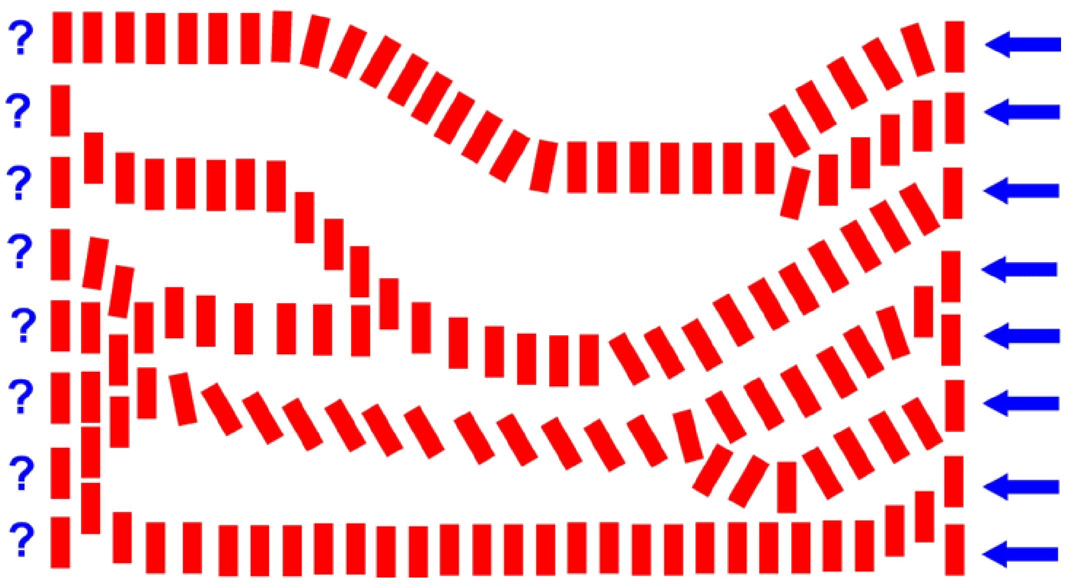


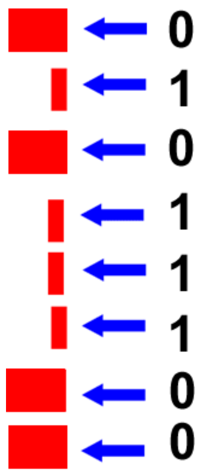
Just as radios with vacuum tubes were superseded by "transistor radios" beginning in the 1950s, so this "first generation" of digital computers based on vacuum tubes eventually gave way to a "second generation" that used the transistor as an even faster—and considerably smaller – non-moving, on-off switch for representing the 1 or 0 of a binary digit.

1.2.1. Processors

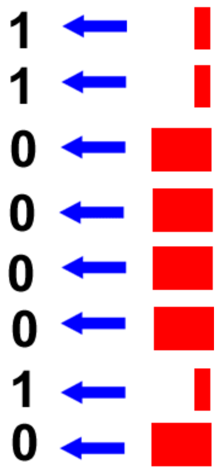
It is fairly easy to acquire a basic understanding of how a line of interlocking, 10-position gears can mimic the operations of decimal arithmetic. But it is far less obvious how an array of vacuum tubes or transistors, used as electronic on-off switches, mimic the operations of binary arithmetic.

One helpful analogy is that of a set of dominoes. Imagine a domino exhibition on a late-night talk show, where a domino champion sets up an elaborate maze of dominoes, knocks one of them over, and sets off an elaborate chain reaction of falling dominoes, lasting several minutes. Eventually, the sequence of falling dominoes reaches the end, and the last set of dominoes tumble over in a grand flourish. Similarly, imagine a set of dominoes on a table where there is a line of eight dominoes at one end, and another line of eight dominoes at the other end, with a maze of other dominoes in between. If you were to go to the eight dominoes at one end and knock over some or all of them, this would set off a chain reaction of falling dominoes in the maze laid out until, eventually, this chain reaction stopped at the other end where some or all of those eight dominoes would be knocked over as a result of this chain reaction.



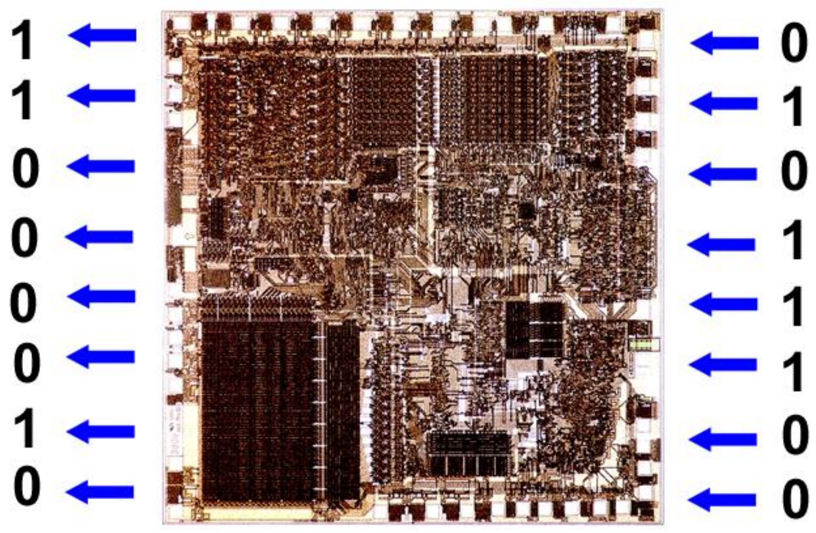


There are some similarities here to the way a processor works. A domino, like a transistor, is a two state device: just as a transistor can be in either an on or off position, a domino can be either standing up or lying down. Thus, like any other two state device, a domino or transistor can model the two possibilities that exist for a binary digit: a 0 or a 1. For example, we could think of a domino that is standing up as a 1, and a domino that is lying down as a 0. Knocking over some or all of the dominoes in that first row of eight, then, is like "inputting" an eight digit binary number into this domino "machine."



In a sense, this binary number is an “instruction” to this machine, specifying the particular set of chain reactions of these two-state devices that should take place. And, when this chain reaction is completed, and some or all of the eight dominoes at the other end are knocked over, it is as if this domino machine has "output" an eight digit binary number.

This domino analogy provides many similarities to the way a processor “chip” made up of transistors operates. Binary numbers representing a basic arithmetic operation -- add, subtract, multiply, divide -- flow into a processor in the form of "high" or "low" electrical signals on wires. This sets off a chain reaction among the literally millions of microscopic transistors, on-off switches, that make up the processor. When the chain reaction has completed, a binary number representing the result flows out on the wires leading away from the processor. The maze of transistors within the processor is designed so that output of the chain reaction is the one that represents the "right answer" for the input arithmetic instruction. The Intel 8088, the processor used in the original IBM PC, in fact, was an 8-bit processor, meaning that, in the course of each “instruction cycle,” an 8-digit binary number would be input, the “processing” (chain reaction) would take place, and a resulting 8-digit binary number would be output.

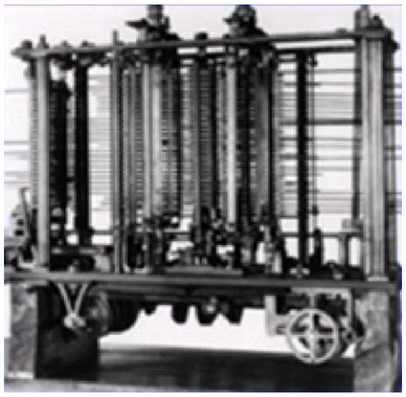


Thus, even today, a modern electronic digital computer is still, at the core of its hardware, a machine that performs basic arithmetic operations. More specifically, it is a machine that mimics or models the way that digits change when humans do basic arithmetic. What is remarkable about the way that today's computers model arithmetic is their extraordinary speed in doing so. Today's microprocessors are typically 32 bits or higher, meaning that their instructions are comprised of binary numbers that are 32 or more digits. Their instruction cycles are described in "gigahertz," meaning that such processors can perform literally billions of instruction cycles every second.

1.3. Software

The arithmetic power that such *hardware* provides is useless unless it can be put into service performing useful calculations in correct sequences involving meaningful numbers. It is computer *software* that provides such meaningful, useful direction. Indeed, it is the rise of software that has enabled computers to evolve from mere number crunchers into technologies that now enrich so many areas of human life.

Consider the analogy of computing one's taxes. A calculator can certainly be of assistance in this process, speeding up and improving the accuracy of the arithmetic that is involved. However, a calculator cannot compute your taxes for you. Rather, it is the tax form that specifies *which* arithmetic operations should be performed, in what *order*, and with what *numbers*. In this sense, a tax form has much in common with a computer *program*, which is a also a defined sequence of actions involving correct information that, when performed, produces a desired result. For example, in the case of the tax software that is widely available today, the computer program is modeled after the program for human action that is prescribed by the tax form.



Charles Babbage, already identified as a key figure in the history of computer hardware, is also a key figure in the history of software. After giving up on the "Difference Engine," Babbage began work on a much more sophisticated machine that he called his "Analytical Engine." The operation of this machine was to be far more versatile and automatic than his earlier invention. In hardware terms, Babbage conceived of a machine built to perform the basic operations of arithmetic upon numeric digits – i.e., a calculator. However, borrowing a technology from the automated "Jacquard" looms that began to appear during the early 1800s, Babbage planned to feed into his Analytical Engine sequences of metal cards with holes punched into them. Instead of being used to define a sequence of threads to incorporate into a particular weave, the punched cards would be used to define a sequence of basic arithmetic operations for the Analytical Engine to perform that, together, achieved a desired mathematical result. In other words, unlike previous calculating machines, the Analytical Engine would be *programmable*: just as a single automated loom could perform different weaves simply by switching sets of punched cards, so Babbage's Analytical Engine would be able to switch between different mathematical computations simply by changing the set of punched cards. To a remarkable degree, the Analytical Engine anticipated the fundamental "architecture" of the modern electronic computer in that it was organized into the four primary subsystems of processing, storage, input, and output.

Ada Lovelace, daughter of Lord Byron, was one of the few people other than Babbage who understood the Analytical Engine's enormous potential. She described the similarity of Jacquard's and Babbage's inventions: "The Analytical Engine weaves algebraic patterns just as the Jacquard loom weaves flowers and leaves," in both cases, simply by performing a carefully devised sequence of basic operations. Lovelace designed and wrote out demonstrations of how complex mathematical computations could be constructed entirely from sequences of the basic set of arithmetic operations of which the Analytical Engine would be capable. Ada Lovelace is often deemed to be "the first programmer," and her work certainly has much in it that recommends this title for her – even more, in fact, than is usually acknowledged.

In her writings, Lovelace identified that one of the key characteristics of a computer program is its carefully *sequential* nature. Mathematicians often use the term "algorithm" to refer to a specific sequence of operations that, if performed, will produce a certain desired result. Accordingly, one of the key activities of computer programming is often said to be "algorithm design," whereby a certain process is successfully broken down into a sequence that is comprised entirely of operations that a computer is able to perform. This notion of programming as designing a sequence of operations to accomplish some desired result also matches well to what is sometimes called the "**procedural**" paradigm of computer programming.

Lovelace also noted that a good computer program is one that is *general*, in the sense that the designed sequence of operations should be able to “be performed on an infinite variety of particular numerical values” rather than being designed to operate upon only a specific set of operands. For example, rather than designing a program to perform

$$(2 \times 10) - 5$$

a program that is "general” would be one designed to take in *any* three numbers, multiply the first two, and then subtract the third number from the result. This illustrates what Lovelace described as operations that are "independent" from “*the objects operated upon.*” In modern computing, this is often described in terms of the separation that is maintained between the "data" and the operations upon that data.

Lovelace also described what she called “cycles” of operations, where a certain desired result could be achieved by performing a certain subset of operations repeatedly. In modern computer programming, this is called a "loop." One simple illustration of this is the way that any multiplication operation

$$5 \times 4$$

can also be achieved by repeatedly performing a single addition operation:

$$5 + 5 + 5 + 5$$

This remarkable capability of the computer to automatically perform cycles of operation, with the results building upon each other, is part of the reason why Lovelace boldly claimed that such a machine would in fact be able to perform computations that had not ever “been actually worked out” previously by any human. This also underscores an important aspect of programming that is sometimes overlooked: namely, that the process of programming seldom consists of the mere encoding in a programming language of a fully envisioned idea of the software that was already completely worked out, conceptually. Rather, the process of programming is one that entails exploration, experimentation, discovery, creativity, and invention.

Perhaps even more impressive is the degree to which Ada Lovelace foresaw that the applications of computing would be well beyond science and mathematics. Lovelace carefully pointed out that the analytical engine operated not upon actual numbers but, rather, upon *symbols* of numbers, and that “the engine can arrange and combine” such symbols of numbers just as readily “if they were *letters* or any other *general* symbols.” In fact, said Lovelace, what is meant in this context “by the word *operation*” could refer to “any process which alters the mutual relation of two or more things, be this relation of what kind it may.” Thus, for Lovelace, part of the power of the computer is its ability to create and manipulate symbolic representations of the many entities and “great facts of the natural world.” However, Lovelace also identifies the extraordinary ability of the computer to create symbolic representations of certain aspects of the *relationships* between facts, objects, and living things, and, through its manipulations of those representations, create models of “those unceasing changes of mutual relationship which, visibly or invisibly, consciously or unconsciously to our immediate physical perceptions, are interminably going on in the agencies of the creation we live amidst.” Thus, given this “most general definition” of a computer as a manipulator of symbolic representations of facts, entities, and relationships, Lovelace suggested that the “new, fast, and powerful language” that we now call computer “programming” could potentially be of use not only to mathematics and science but to “all subjects in the universe.”

As we now know, computers can do so much more than the kind of arithmetic computations that are involved in doing one's taxes. This is because computers have indeed motivated software programmers to discover just how many actual and imaginary facts and entities can, to at least some degree, be modeled in the form of symbols that can be manipulated in ways that mimic actions, processes, phenomena, and relationships.

1.3.1. Object-Oriented Programming and Personal Computing

In certain ways, Lovelace anticipated a shift in conceptions of computer programming. She saw computing and programming as extending beyond the realm of numerical data and traditional notions of algebraic sequences of arithmetic operations. Instead, she saw programming as technology that enables the creation and manipulation of what we would now call computer “models”. This shift from a strictly procedural paradigm toward one that also allows for the option of what has come to be known as “object-oriented programming” has virtual “objects” with associated characteristics and actions as the fundamental “building blocks” of software.

Object-oriented programming (“OOP”) emerged largely from attempts in the latter half of the 1970s to develop a new generation of “personal computers” and “graphical user interfaces,” and the rapid rise in popularity of these technologies beginning in the latter half of the 1980s was accompanied by a similar rise to prominence of the object-oriented notions of programming that enabled them.

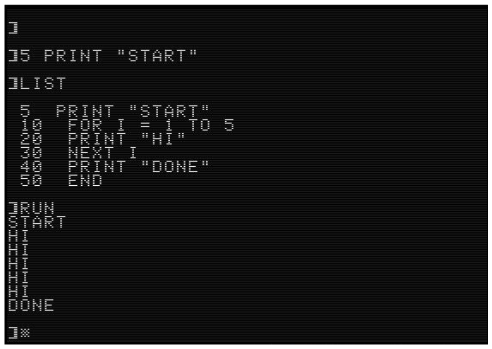
Cardboard mockup of a Dynabook

One of the first persons to use the term "personal computer" was Alan Kay who, over the course of the 1970s, led a team at Xerox Corporation's Palo Alto Research Center (PARC) in pursuit of the creation of a small, portable computer. The envisioned "Dynabook,” as it was called, bore a remarkable similarity to the "notebook" computers that would begin to appear two decades later. One of the most striking features of the Dynabook was its "graphical user interface" (GUI), which enabled users to interact with this computer by selecting and manipulating onscreen menus using a relatively unknown new pointing device called a "mouse,” rather than requiring the user to memorize and type cryptic commands in the manner that had characterized the previous "command line interface" paradigm. The Dynabook's GUI also enabled the user to operate in multiple onscreen "windows” that could provide different views simultaneously. The data created and manipulated by the user was represented on the screen in the form of interactive virtual "objects": passages of text, drawings, photographs, sounds and music, descriptive icons, and much more. Such a GUI was designed to create an interactive on-screen representation of an environment for action that would be immediately recognized and easily manipulated by the user. However, the “Smalltalk” software system of the Dynabook was also a graphical object-oriented programming environment. The driving vision of Kay’s team was one in which Dynabook users would begin by interacting with intuitive software created by others but, when they were ready, they would also be able to examine and even change the defined characteristics and behaviors of the virtual "objects” that comprised these software programs. In fact, the goal was to design a version of the

Smalltalk programming language that was so intuitive that even children using the Dynabook might eventually decide to author their own software programs. Early tests with Palo Alto school children had proven to be very promising in this regard.

As part of an attempt to demonstrate the value of the Dynabook project, an example of such a GUI was created that featured a virtual "desktop" designed to emulate the activities and workspace typical of office work in a way that even such an employee with no prior computer experience would find interacting with the system to be an intuitive, productive, and enjoyable experience. This experiment failed to convince Xerox executives; however, several years later in 1979, Apple Computer cofounder Steve Jobs saw a demonstration of this GUI.

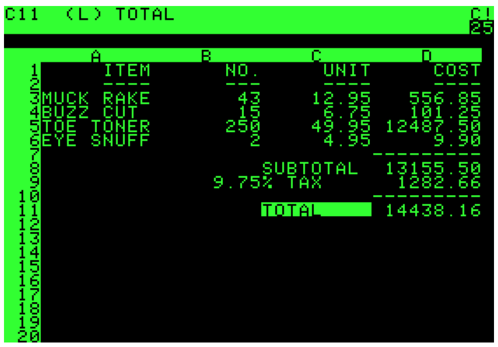
The Apple II personal computer system, first introduced in 1977, had, like other early "microcomputer" systems, interacted with users via a command line interface. However, after seeing the PARC system, Jobs said that within "ten minutes it was obvious to me that all computers would work like this someday." Jobs immediately set to work on creating the new generation of Apple computers that began when the Macintosh was introduced in January 1984. The graphical user interface of the Macintosh, as well as that of the Microsoft "Windows" operating system that was introduced a year later, bore much resemblance to the "desktop" GUI developed at Xerox PARC.



```
]
]5 PRINT "START"
]LIST

 5  PRINT "START"
 10  FOR I = 1 TO 5
 20  PRINT "HI"
 30  NEXT I
 40  PRINT "DONE"
 50  END

]RUN
START
HI
HI
HI
HI
HI
DONE
]
```

Like other early microcomputers, initially the Apple II was designed and marketed on the assumption that users would usually write their own software (probably using one of the introductory procedural programming languages of that era, such as the widely-used "BASIC").   However, another key event had taken place in 1979, one that subsequently propelled the Apple II to heights of popularity that went far beyond computer hobbyists and programmers.

"VisiCalc," a radically new and powerful "spreadsheet" software program that had been developed several years earlier for larger computer systems was successfully "ported" for use on the Apple II microcomputer system in 1979. VisiCalc is an example of what is called an "application" software program – an already-written software program that is distributed for use by others. In fact, VisiCalc is often said to be the Apple II's "killer app," a term which has come to refer to an application of a new technology that serves to



```
C11   (L) TOTAL                                C!
                                               25

          A         B         C         D
 1       ITEM      NO.      UNIT      COST
 2       ----      ---      ----      ----
 3MUCK RAKE         43     12.95    556.85
 4BUZZ CUT          15      6.75    101.25
 5TOE TONER        250     49.95  12487.50
 6EYE SNUFF          2      4.95      9.90
 7                                ----------
 8                     SUBTOTAL   13155.50
 9                  9.75% TAX      1282.66
10                                ----------
11                     TOTAL      14438.16
12
13
14
15
16
17
18
19
20
```

justify the adoption of that technology by a wide range of new users.[1] The availability of a version of the highly-regarded VisiCalc spreadsheet software for the Apple II – and, subsequently, for several other microcomputers, including those made by Atari and Commodore – served to convince many that the Apple II was not a mere novelty device but, rather, was a bona fide computer with the potential for serious business and scientific use. Similarly, when IBM Corporation, in response to this sudden and rapid rise in the popularity of microcomputers in the late 1970s fueled largely by VisiCalc, rushed to introduce a personal computer of it is own in 1981, the "killer app" that convinced the most consumers to purchase the IBM "PC" was, once again, a spreadsheet program: "Lotus 1-2-3."

Spreadsheet software provided novice computer users with the ability to undertake the kind of computational operations on numerical and textual data that were so strongly associated with computers without having to learn what they would have viewed as a "real" programming language. In fact, especially in light of the overall history of computing, it does not seem to be too much of a stretch to suggest that spreadsheet software provided a new kind of computer programming environment. Novice users certainly found the designing of computations within spreadsheets to be much more intuitive than doing so in standard programming languages. However, in retrospect, it seems unfortunate that spreadsheet software itself came to be understood so strongly as the "application software," rather than considering such spreadsheet software to be a kind of programming environment, where the spreadsheets that this software system enabled one to create for use by others were considered to be the resultant application software. The fact that this did not take place perhaps reflects the larger shift in the public understanding of the concept of "personal computing" that took place over the course of the 1980s, where Alan Kay's dream of a personal computer that was so graphical and so intuitive that even a child would be able to write software programs for it was displaced by a conception of the personal computer as a machine that *no one* should have to learn to program.

What seems to have been completely lost in the Apple's and Microsoft's adaptations of PARC's Dynabook vision was the idea of enabling novice computer users not only to use software written by others but also to subsequently proceed comfortably along a path of learning that would lead to an ability to begin to modify those software programs and, eventually, to create simple software programs of their own. This was central to Alan Kay's conception of the personal computer as a *medium*, one that provided power to construct unique kinds of *models* of ideas that could be experienced as a kind of performance. In fact, Kay credited the notions of both the theatrical and musical performance as having been highly influential upon the design of the Dynabook. Kay has said that he envisioned the personal computer as something like "an instrument whose music is ideas." In contrast, Kay has said, the version of personal computing inaugurated by Apple and Microsoft in the 1980s has made for two decades of a user experience that is more like "air guitar."[2] Instead of using the personal computer as a medium in which to design and stage performances of our own ideas, we instead perform within models of user experience that have been designed by others, ones that are often troublingly generic, constraining, and difficult with which to identify.

Indeed, along with the installation of the GUI of the Macintosh and Windows GUI as the dominant interface paradigm came a new conception of the computer "user" as the "*end user*" who, by definition,

[1] For example, the origins of Microsoft Corporation can be traced to a version of the BASIC programming language that was widely in use on larger computer systems but that Bill Gates and Paul Allen managed to port to one of the first commercially available microcomputer systems, the Altair 8800.
[2] Kay had said this well before the "Guitar Hero" computer/video game rose to such popularity!

was a non-programmer. This conception of "personal computing" as being the exact *opposite* of computer programming has persisted for more than two decades.

However, part of the reason why the Macintosh and Windows operating systems were ultimately such severe reductions in the PARC vision of personal computing, in spite of the many similarities of the GUIs, is that it simply couldn't be fully achieved affordably on the hardware technologies available at the time. Moreover, in spite of early successes, PARC researchers themselves came to realize that their vision of a version of Smalltalk as a programming language that would be intuitive had proven to be more difficult to achieve than expected.

Interestingly, Ada Lovelace also foresaw the kind of hype and disillusionment that can surround computer technology. "It is desirable," she said, "to guard against the possibility of exaggerated ideas that might arise as to the powers of the Analytical Engine. In considering any new subject, there is frequently a tendency, first, to *overrate* what we find to be already interesting or remarkable." However, Lovelace also noted that, on the other hand, "when we do discover that our notions have surpassed those that were really tenable," there is a tendency to swing too far in the opposite direction, "by a sort of natural reaction, to *undervalue* the true state of the case."

Perhaps it must be said that something of this sort has happened in regard to the kind of "binary" relationship that has come to be constructed in public perceptions of computer programming vs. personal computing. Indeed, at times, this diametric relationship even reaches the point of bigotry, where end users are derided as ignorant by programmers, and programmers are labeled as "geeks" by end users.

However, this new millennium has also come to see a renewed, gradually increasing interest in the idea of programming on the part of persons whose passionate use of personal computer software has led to a desire to break out of the end user "box" in order to customize their user experience or to invent new forms of user experience entirely. This is especially true in the case of certain areas of computing that have come to be known by such varying names as "digital art," "digital imaging," "digital media," and "digital design," where the personal computer is explicitly regarded as an artistic medium. Given that computer programming tends to be so strongly associated with numerical data of math, science, and business, it might initially seem surprising that persons who approach computing from a more artistic direction would be inclined toward learning to program. On the other hand, it is also the case that persons who undertake more artistic and expressive design processes are less likely to be accepting of the more generic and limiting workflows inherent in most commercially available application software. In this sense, perhaps it should not be surprising that so many digital photographers and designers of digital graphics, for example, should undertake to customize the applications they use by designing their own "filters" and "extensions" for such software, in spite of the degree of difficulty that is often involved in developing such customizations. For some, this artistic drive even leads them to teach themselves a standard programming language, which is certainly the most difficult method of acquiring such knowledge. In fact, the "Processing" programming language that is used in this book was originally developed in 2001 at the Massachusetts Institute of Technology for the purpose of making it easier for artists to create dynamic graphical art on a computer by combining images, animation, and interactions.

1.3.2. Programming Languages and Compiling

As was noted earlier, a modern electronic computer still remains, at its most fundamental hardware level, a binary, digital arithmetic machine. The "processor" that is at its core operates on "data" that are understood to be binary numbers. Each digit of such a data item consists of either a 0 or 1 and is represented and sent to the processor in the form of an electrical signal with a voltage that is evaluated to be either "high" or "low" which in turn serves to "flip" certain of the processor's on/off transistor switches in such a way that mimics the "inputting" of this binary number into the processor.

Similarly, each of the basic arithmetic operations that the processor performs – e.g., addition, subtraction, multiplication, and division – is itself described by a unique binary number. Thus, the instruction to perform a certain arithmetic operation can itself be sent to the processor as a binary number. A collection of binary digits represented as high/low electrical signals trigger the processor in such a way as to perform the particular chain reaction of transistor switches that mimics that particular arithmetic operation. This reaction then outputs a representation of the binary number that is the correct result for that operation with the given data.

Thus, at the most fundamental level, a software program that a computer's processor performs consists of a very long stream of binary numbers, some of which represent a particular arithmetic operation to be performed and others of which represent the data to be used in that operation.

```
100011 00011 01000 00000 00001 00100
```

In fact, early electronic computers in the 1940s and 1950s required a human programmer to input long sequences of binary numbers of this very sort in order to load a software program into the computer.

Fortunately, this is no longer the case. Over time, a wide variety of "high-level" programming languages have been developed that allow a programmer to write software programs that allow the use of decimal numbers and arithmetic symbols that are similar to the ones used when humans do math. Thus, adding 10 to the current value of a variable named "x" could be encoded as:

```
x + 10
```

High-level programming languages also allow data that consists of letters and punctuation marks and include commands that are very similar to words used in human languages. For example, in a typical high-level programming language, an instruction to print the greeting "Hello!" on the screen could be something like:

```
print("Hello!")
```

Nevertheless, in order to be executed on a computer, software written in a high-level programming language must first be translated into sequences of binary numbers, consisting entirely of 0s and 1s. This is because, at the hardware level of a computer, the only thing that can really be "input" into a processor chip are "high" and "low" electrical signals that represent such binary digits and that will set off the appropriate chain reaction in the processor's maze of on/off transistor switches.

This translation of software written in a high-level language into binary instructions that can be sent to the processor is called "compiling" a program. Instructions written in the high-level programming language, often called the "source code" of that software program, are fed into a special translation software program called a "compiler" which is designed to convert source code written in a particular high-level programming language into the needed sequence of binary instructions. The binary output created by the compiler that can be executed by the processor is often called the "binary code," "machine code," "object code," or "executable code" for that particular software program.

However, there have always been certain complications inherent in the compiling of software programs. First of all, the set of binary instructions that can be sent to a processor can vary from one kind of processor to another. As a result, source code that has been compiled for one kind of processor chip might not be executable on a different kind of processor chip. This is a big part of the reason why personal computer application software such as a word processor will typically be made available in separate "Macintosh" and "PC" versions: because the binary instructions in the executable code are always sent to a processor at the direction of a particular operating system, and the Macintosh and Windows operating systems themselves are software programs that have been designed and compiled for totally different categories of processor chips.[3] This is in contrast to the very similar "Unix" and "Linux" family of operating systems which can be compiled to operate on an enormous variety of processor chips, including the kinds of processor chips found in both Macintosh and Windows computers.

1.3.3. Platform Independence

The explosion in the popularity of the World Wide Web beginning in the latter half of the 1990s also served to underscore the problem of incompatible personal computer "platforms." From the very outset, the Internet was designed to be a "platform-independent" infrastructure. Thus, for example, because the "JPEG" digital image format was designed specifically for use on the World Wide Web, it was designed to be platform-independent. As a result, any given JPEG image file can be accessed, viewed, manipulated, and exchanged over the Internet by users of Macintosh, Windows/PC, and Unix/Linux personal computers alike.

The World Wide Web's platform-independent, multimedia *data* formats served also to provide an impetus for exploring the feasibility of developing platform-independent *software* programs. Indeed, this is much of the reason why the platform-independent "Java" programming language has risen to such heights of popularity over the course of the past decade. The source code of a software program written in Java is not compiled directly into the binary code for a specific processor chip. Rather, the Java source code is compiled for a "virtual machine" – that is, for a kind of "generic" processor that doesn't actually exist in any hardware form. Thus, the code that results from the Java compiler is not really executable binary code yet; rather, it represents an intermediate step called "byte code." However, the company that developed the Java platform, Sun Microsystems, freely provides software for each of the major computer platforms that will convert Java bytecode into the executable binary code required by that particular platform, thus eliminating the need for multiple compiled versions of the same software program.

Java is also an object-oriented language, where the fundamental building blocks of a software program are conceived to be a variety of virtual objects, each of which are defined to have certain characteristics and behaviors that can be employed to construct the overall functionality of the program that is desired. Initially, Java's object-oriented nature was also part of the reason for its appeal and rapid rise in popularity in industry. Accordingly, beginning in the late 1990s, Java was widely embraced as the high-level programming language of choice for introductory computer programming courses and was also adopted in many areas of industry.

[3] Exception: The recent Intel Macintoshes.

1.4. Processing

The complexity of the procedure used to enter a program, compile it, and execute it— commonly known as the **programming environment** — is often a hindrance to learning how to program. In a command-line environment, one may have to learn a collection of commands for the operating system (e.g., Unix) that is being used and, in addition, editor commands for entering and modifying the program. However, a variety of **integrated development environments** (**IDEs**) such as Visual Studio developed by Microsoft and the popular open-source IDE Eclipse are available that make this considerably easier. Typically, one first creates a new project, specifying the programming language being used, and then perhaps adds some libraries to the project, maybe rearranges some windows and creates a package and a text file, enters the source code for the program in this file and saves it, and finally builds the project. The resulting object program can be executed.

However, the Processing environment is one of the simplest to use. When it is started, a simple "sketch window" appears that has six buttons at the top, a *program editor window* below this, and a *text output window* at the bottom, as pictured on the right in Figure 1-1.

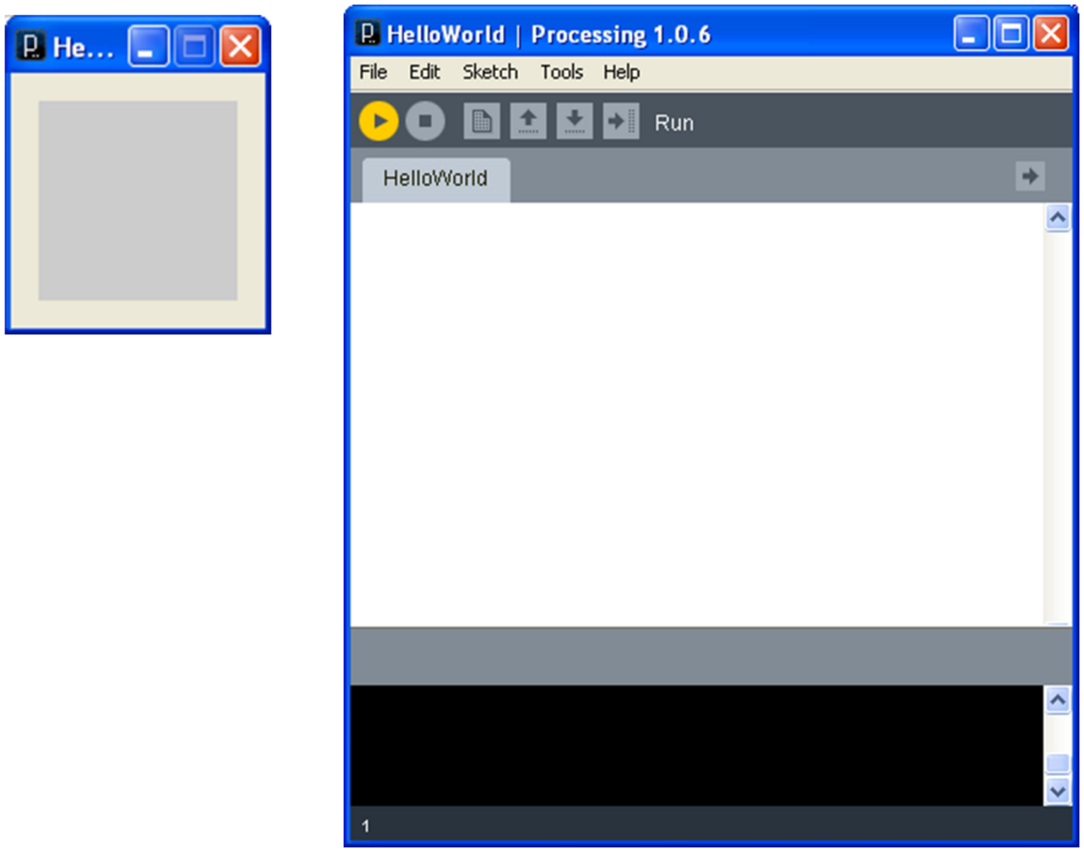
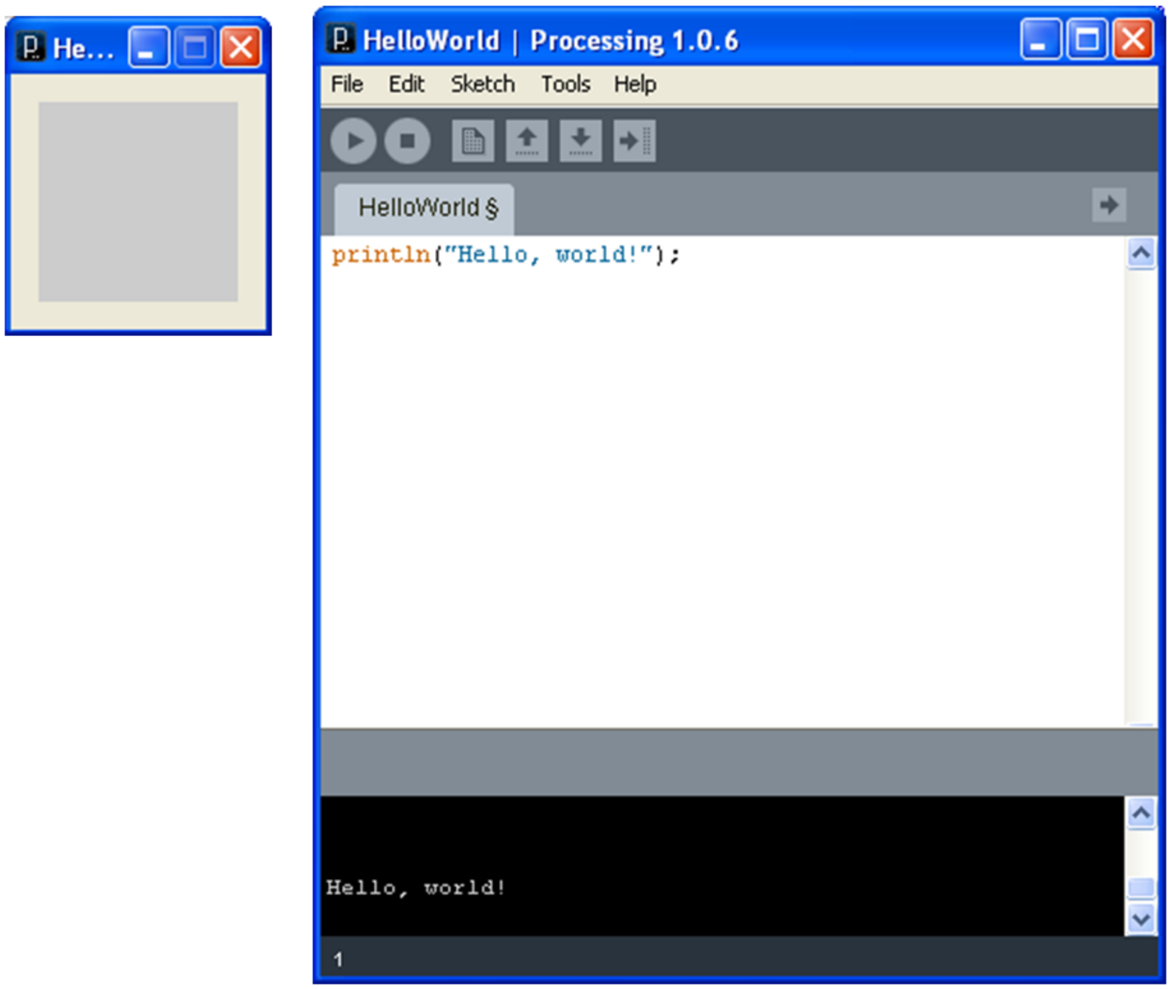


Figure 1-1. Program editor window and (empty) graphical output window

The program editor currently contains no program code — i.e., it contains an empty program. But if we click the leftmost (Run) button ▶ at the top of the sketch window, another window (pictured on the left in Figure 1.1) appears. It is called the *visual output window* because it contains graphical output produced by a program. In this example, no output is displayed because the program is empty.



```
println("Hello, world!");
```

```
Hello, world!
```

Figure 1-2. Text-based hello-world program

A program without any useful code is not very interesting, so we add this line:

```
println("Hello World!");
```

as shown in Figure 1-2. Once again, we click the Run button. The output:

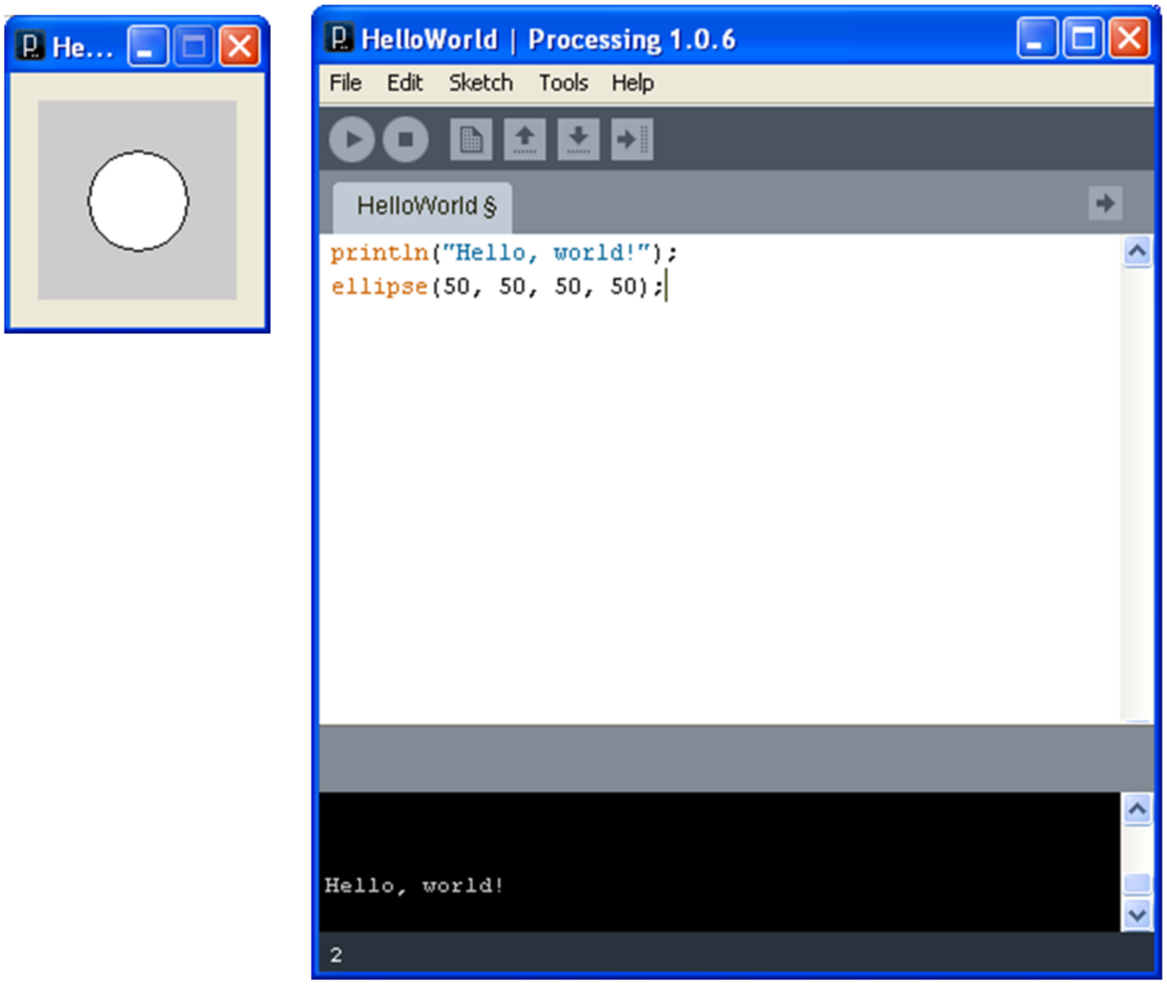
```
Hello World!
```

appears in the text output window. (No output appears in the visual output window because we are outputting text, not graphics.)

To obtain a simple graphic representation in addition to the text output, we could add the line

```
ellipse(50, 50, 50, 50);
```

This will produce an ellipse with center (50, 50), and major and minor axes both 50 (pixels) — that is, a circle with center (50, 50) and radius 50 — in the visual output window as shown in Figure 1-3.[4]



```
println("Hello, world!");
ellipse(50, 50, 50, 50);
```

```
Hello, world!
```

```
2
```

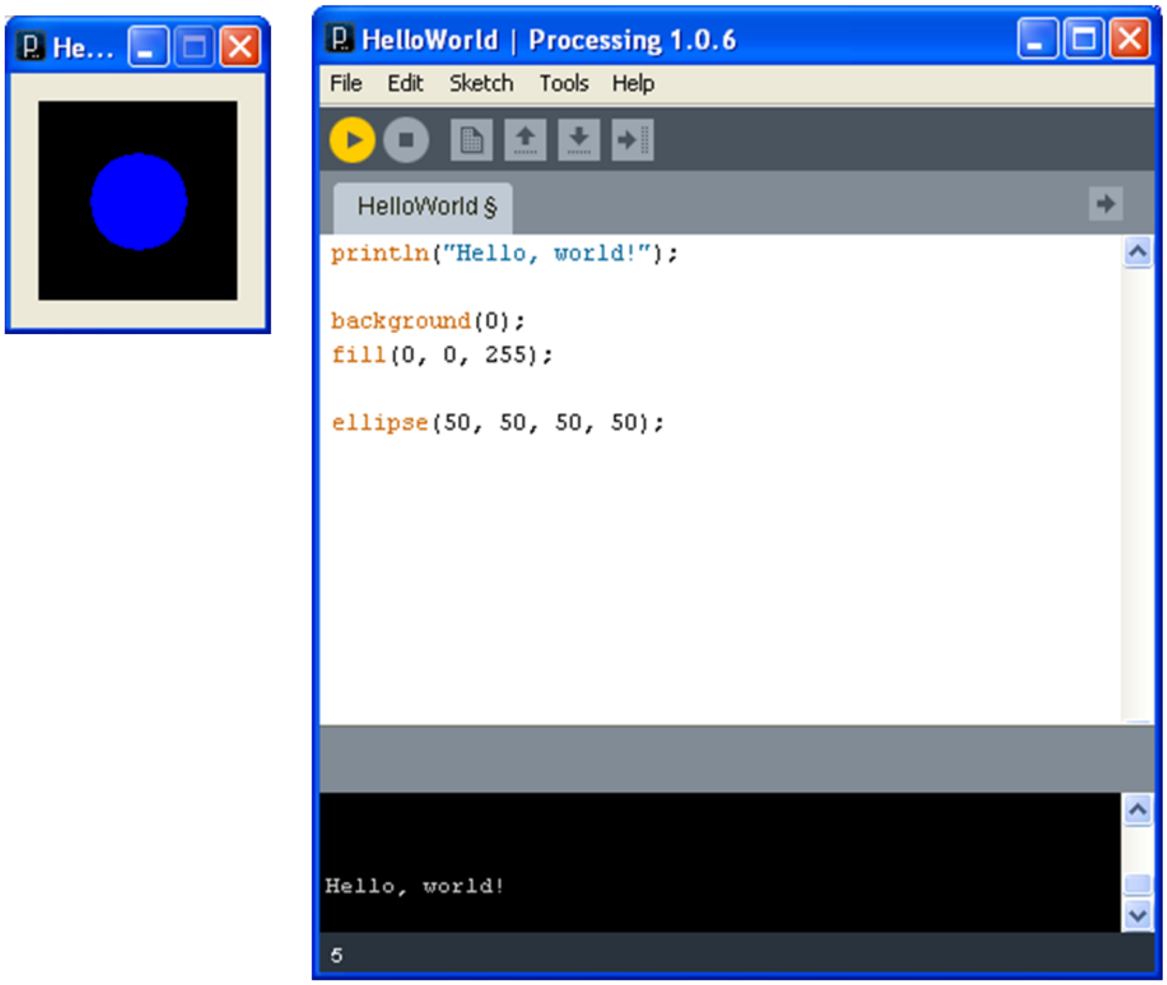
Figure 1-3. Graphical hello-world program

And if we wanted to add some color, the two lines of code:

```
    background(0);
    fill(0, 0, 255);
```

as shown in Figure 1-4 can be added to fill the background with one color (black as specified by the color value 0, which indicates no light) and the circle with another (blue as specified by the red-green-blue color triple indicating no red, no green and full intensity, 255, blue).

[4] The coordinate system used in the visual output window has the origin (0, 0) at the upper left corner, the positive *x*-axis directed to the right, and the positive *y*-axis directed downward.



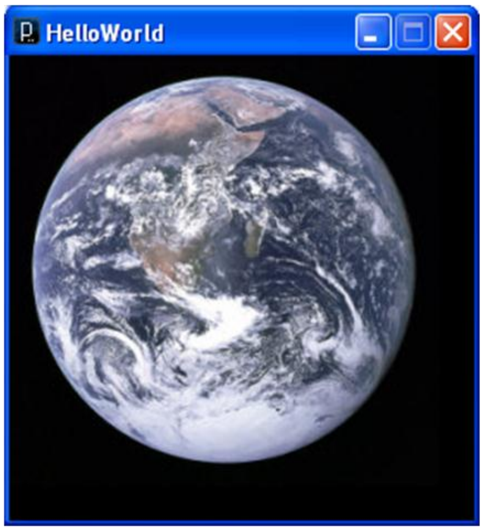
```
println("Hello, world!");

background(0);
fill(0, 0, 255);

ellipse(50, 50, 50, 50);
```

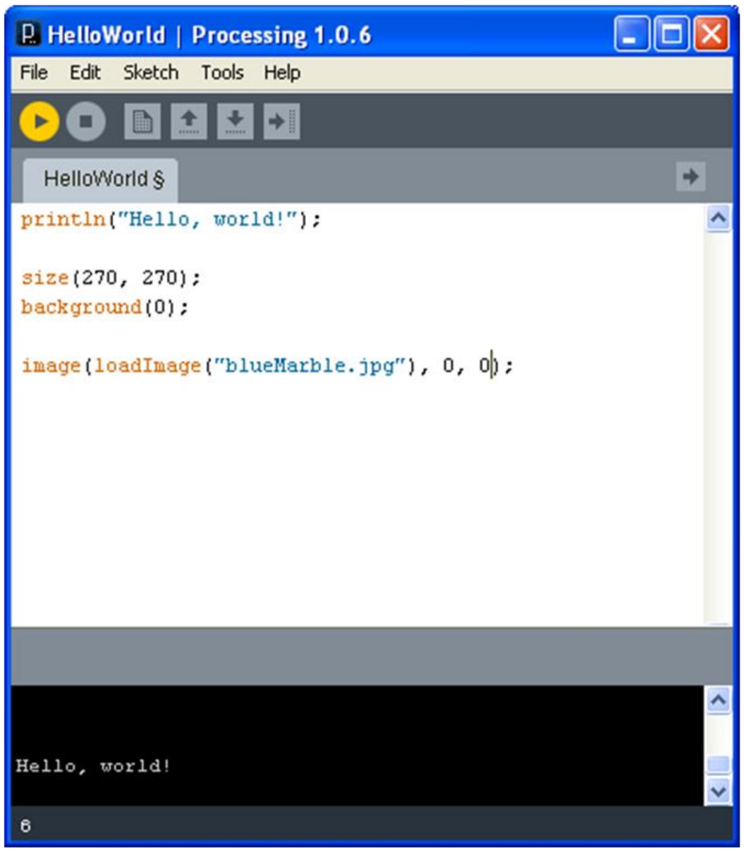
```
Hello, world!
```

```
5
```

Figure 1-4. Graphical hello-world with color

If we prefer to have a more realistic picture of the earth, we need only download an image file of the Earth into the folder containing our program and modify the program as shown in Figure 1-5.





```
println("Hello, world!");

size(270, 270);
background(0);

image(loadImage("blueMarble.jpg"), 0, 0);
```

```
Hello, world!
```

6

Figure 1-5. Hello-world program with a bitmap image of the Earth

The preceding examples are intended to provide a first exposure to the Processing environment and to give you some indication of how easy it is to do some exciting programming in it. We will explain the code in these examples and much more in the chapters that follow.