the initial version of which appeared in 1960. It probably still is the most commonly used language for these applications. Business languages are characterized by facilities for producing elaborate reports, precise ways of describing and storing decimal numbers and character data, and the ability to specify decimal arithmetic operations.

There have been few developments in business application languages outside the development and evolution of COBOL. Therefore, this book includes only limited discussions of the structures in COBOL.

1.2.3 Artificial Intelligence

Artificial intelligence (AI) is a broad area of computer applications characterized by the use of symbolic rather than numeric computations. Symbolic computation means that symbols, consisting of names rather than numbers, are manipulated. Also, symbolic computation is more conveniently done with linked lists of data rather than arrays. This kind of programming sometimes requires more flexibility than other programming domains. For example, in some AI applications the ability to create and execute code segments during execution is convenient.

The first widely used programming language developed for AI applications was the functional language Lisp (McCarthy et al., 1965), which appeared in 1959. Most AI applications developed prior to 1990 were written in Lisp or one of its close relatives. During the early 1970s, however, an alternative approach to some of these applications appeared—logic programming using the Prolog (Clocksin and Mellish, 2013) language. More recently, some AI applications have been written in systems languages such as C. Scheme (Dybvig, 2009), a dialect of Lisp, and Prolog are introduced in Chapters 15 and 16, respectively.

1.2.4 Web Software

The World Wide Web is supported by an eclectic collection of languages, ranging from markup languages, such as HTML, which is not a programming language, to general-purpose programming languages, such as Java. Because of the pervasive need for dynamic Web content, some computation capability is often included in the technology of content presentation. This functionality can be provided by embedding programming code in an HTML document. Such code is often in the form of a scripting language, such as JavaScript or PHP (Tatroe, 2013). There are also some markup-like languages that have been extended to include constructs that control document processing, which are discussed in Section 1.5 and in Chapter 2.

1.3 Language Evaluation Criteria

As noted previously, the purpose of this book is to examine carefully the underlying concepts of the various constructs and capabilities of programming languages. We will also evaluate these features, focusing on their impact on the software development process, including maintenance. To do this, we need a set of evaluation criteria. Such a list of criteria is necessarily controversial, because it is difficult to get even two computer scientists to agree on the value of some given language characteristic relative to others. In spite of these differences, most would agree that the criteria discussed in the following subsections are important.

Some of the characteristics that influence three of the four most important of these criteria are shown in Table 1.1, and the criteria themselves are discussed in the following sections.[2] Note that only the most important characteristics are included in the table, mirroring the discussion in the following subsections. One could probably make the case that if one considered less important characteristics, virtually all table positions could include "bullets."

Note that some of these characteristics are broad and somewhat vague, such as writability, whereas others are specific language constructs, such as exception handling. Furthermore, although the discussion might seem to imply that the criteria have equal importance, that implication is not intended, and it is clearly not the case.

1.3.1 Readability

One of the most important criteria for judging a programming language is the ease with which programs can be read and understood. Before 1970, software development was largely thought of in terms of writing code. The primary positive characteristic of programming languages was efficiency. Language constructs were designed more from the point of view of the computer than of the computer users. In the 1970s, however, the software life-cycle concept (Booch, 1987) was developed; coding was relegated to a much smaller role,

Table 1.1 Language evaluation criteria and the characteristics that affect them

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

2. The fourth primary criterion is cost, which is not included in the table because it is only slightly related to the other criteria and the characteristics that influence them.

and maintenance was recognized as a major part of the cycle, particularly in terms of cost. Because ease of maintenance is determined in large part by the readability of programs, readability became an important measure of the quality of programs and programming languages. This was an important juncture in the evolution of programming languages. There was a distinct crossover from a focus on machine orientation to a focus on human orientation.

Readability must be considered in the context of the problem domain. For example, if a program that describes a computation is written in a language not designed for such use, the program may be unnatural and convoluted, making it unusually difficult to read.

The following subsections describe characteristics that contribute to the readability of a programming language.

1.3.1.1 Overall Simplicity

The overall simplicity of a programming language strongly affects its readability. A language with a large number of basic constructs is more difficult to learn than one with a smaller number. Programmers who must use a large language often learn a subset of the language and ignore its other features. This learning pattern is sometimes used to excuse the large number of language constructs, but that argument is not valid. Readability problems occur whenever the program's author has learned a different subset from that subset with which the reader is familiar.

A second complicating characteristic of a programming language is **feature multiplicity**—that is, having more than one way to accomplish a particular operation. For example, in Java, a user can increment a simple integer variable in four different ways:

```
count = count + 1
count += 1
count++
++count
```

Although the last two statements have slightly different meanings from each other and from the others in some contexts, all of them have the same meaning when used as stand-alone expressions. These variations are discussed in Chapter 7.

A third potential problem is **operator overloading**, in which a single operator symbol has more than one meaning. Although this is often useful, it can lead to reduced readability if users are allowed to create their own overloading and do not do it sensibly. For example, it is clearly acceptable to overload + to use it for both integer and floating-point addition. In fact, this overloading simplifies a language by reducing the number of operators. However, suppose the programmer defined + used between single-dimensioned array operands to mean the sum of all elements of both arrays. Because the usual meaning of vector addition is quite different from this, this unusual meaning could confuse both the author and the program's readers. An even more extreme example of program confusion would be a user defining + between two vector operands to mean the difference between their respective first elements. Operator overloading is further discussed in Chapter 7.

Simplicity in languages can, of course, be carried too far. For example, the form and meaning of most assembly language statements are models of simplicity, as you can see when you consider the statements that appear in the next section. This very simplicity, however, makes assembly language programs less readable. Because they lack more complex control statements, program structure is less obvious; because the statements are simple, far more of them are required than in equivalent programs in a high-level language. These same arguments apply to the less extreme case of high-level languages with inadequate control and data-structuring constructs.

1.3.1.2 Orthogonality

Orthogonality in a programming language means that a relatively small set of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language. Furthermore, every possible combination of primitives is legal and meaningful. For example, consider data types. Suppose a language has four primitive data types (integer, float, double, and character) and two type operators (array and pointer). If the two type operators can be applied to themselves and the four primitive data types, a large number of data structures can be defined.

The meaning of an orthogonal language feature is independent of the context of its appearance in a program. (The word *orthogonal* comes from the mathematical concept of orthogonal vectors, which are independent of each other.) Orthogonality follows from a symmetry of relationships among primitives. A lack of orthogonality leads to exceptions to the rules of the language. For example, in a programming language that supports pointers, it should be possible to define a pointer to point to any specific type defined in the language. However, if pointers are not allowed to point to arrays, many potentially useful user-defined data structures cannot be defined.

We can illustrate the use of orthogonality as a design concept by comparing one aspect of the assembly languages of the IBM mainframe computers and the VAX series of minicomputers. We consider a single simple situation: adding two 32-bit integer values that reside in either memory or registers and replacing one of the two values with the sum. The IBM mainframes have two instructions for this purpose, which have the forms

```
A Reg1, memory_cell
AR Reg1, Reg2
```

where `Reg1` and `Reg2` represent registers. The semantics of these are

```
Reg1 ← contents(Reg1) + contents(memory_cell)
Reg1 ← contents(Reg1) + contents(Reg2)
```

The VAX addition instruction for 32-bit integer values is

```
ADDL operand_1, operand_2
```

whose semantics is

```
operand_2 ← contents(operand_1) + contents(operand_2)
```

In this case, either operand can be a register or a memory cell.

The VAX instruction design is orthogonal in that a single instruction can use either registers or memory cells as the operands. There are two ways to specify operands, which can be combined in all possible ways. The IBM design is not orthogonal. Only two out of four operand combinations possibilities are legal, and the two require different instructions, `A` and `AR`. The IBM design is more restricted and therefore less writable. For example, you cannot add two values and store the sum in a memory location. Furthermore, the IBM design is more difficult to learn because of the restrictions and the additional instruction.

Orthogonality is closely related to simplicity: The more orthogonal the design of a language, the fewer exceptions the language rules require. Fewer exceptions mean a higher degree of regularity in the design, which makes the language easier to learn, read, and understand. Anyone who has learned a significant part of the English language can testify to the difficulty of learning its many rule exceptions (for example, *i* before *e* except after *c*).

As examples of the lack of orthogonality in a high-level language, consider the following rules and exceptions in C. Although C has two kinds of structured data types, arrays and records (**`struct`**s), records can be returned from functions but arrays cannot. A member of a structure can be any data type except **`void`** or a structure of the same type. An array element can be any data type except **`void`** or a function. Parameters are passed by value, unless they are arrays, in which case they are, in effect, passed by reference (because the appearance of an array name without a subscript in a C program is interpreted to be the address of the array's first element).

As an example of context dependence, consider the C expression

```
a + b
```

This expression often means that the values of `a` and `b` are fetched and added together. However, if `a` happens to be a pointer and `b` is an integer, it affects the value of `b`. For example, if `a` points to a float value that occupies four bytes, then the value of `b` must be scaled—in this case multiplied by 4—before it is added to `a`. Therefore, the type of `a` affects the treatment of the value of `b`. The context of `b` affects its meaning.

Too much orthogonality can also cause problems. Perhaps the most orthogonal programming language is ALGOL 68 (van Wijngaarden et al., 1969). Every language construct in ALGOL 68 has a type, and there are no restrictions on those types. In addition, most constructs produce values. This combinational freedom allows extremely complex constructs. For example, a conditional can appear as the left side of an assignment, along with declarations and other assorted statements, as long as the result is an address. This extreme form of orthogonality leads to unnecessary complexity. Furthermore, because languages require a large number of primitives, a high degree of orthogonality results in an explosion of combinations. So, even if the combinations are simple, their sheer numbers lead to complexity.

Simplicity in a language, therefore, is at least in part the result of a combination of a relatively small number of primitive constructs and a limited use of the concept of orthogonality.

Some believe that functional languages offer a good combination of simplicity and orthogonality. A functional language, such as Lisp, is one in which computations are made primarily by applying functions to given parameters. In contrast, in imperative languages such as C, C++, and Java, computations are usually specified with variables and assignment statements. Functional languages offer potentially the greatest overall simplicity, because they can accomplish everything with a single construct, the function call, which can be combined simply with other function calls. This simple elegance is the reason why some language researchers are attracted to functional languages as the primary alternative to complex nonfunctional languages such as Java. Other factors, such as efficiency, however, have prevented functional languages from becoming more widely used.

1.3.1.3 Data Types

The presence of adequate facilities for defining data types and data structures in a language is another significant aid to readability. For example, suppose a numeric type is used for an indicator flag because there is no Boolean type in the language. In such a language, we might have an assignment such as the following:

```
timeOut = 1
```

The meaning of this statement is unclear, whereas in a language that includes Boolean types, we would have the following:

```
timeOut = true
```

The meaning of this statement is perfectly clear.

1.3.1.4 Syntax Design

The syntax, or form, of the elements of a language has a significant effect on the readability of programs. Following are some examples of syntactic design choices that affect readability:

- *Special words.* Program appearance and thus program readability are strongly influenced by the forms of a language's special words (for example, **`while`**, **`class`**, and **`for`**). Especially important is the method of forming compound statements, or statement groups, primarily in control constructs. Some languages have used matching pairs of special words or symbols to form groups. C and its descendants use braces to specify compound statements. All of these languages have diminished readability because statement groups are always terminated in the same way, which makes it difficult to determine which group is being ended when an **`end`** or a right brace appears. Fortran 95 and Ada (ISO/IEC, 2014) make this clearer by using a distinct closing syntax for each type of statement group. For example, Ada uses **`end if`** to terminate a selection construct and **`end loop`** to terminate a loop construct. This is an example of the conflict between simplicity that results in fewer reserved words, as in Java, and the greater readability that can result from using more reserved words, as in Ada.

Another important issue is whether the special words of a language can be used as names for program variables. If so, the resulting programs can be very confusing. For example, in Fortran 95, special words, such as `Do` and `End`, are legal variable names, so the appearance of these words in a program may or may not connote something special.

- *Form and meaning.* Designing statements so that their appearance at least partially indicates their purpose is an obvious aid to readability. Semantics, or meaning, should follow directly from syntax, or form. In some cases, this principle is violated by two language constructs that are identical or similar in appearance but have different meanings, depending perhaps on context. In C, for example, the meaning of the reserved word **`static`** depends on the context of its appearance. If used on the definition of a variable inside a function, it means the variable is created at compile time. If used on the definition of a variable that is outside all functions, it means the variable is visible only in the file in which its definition appears; that is, it is not exported from that file.

  One of the primary complaints about the shell commands of UNIX (Robbins, 2005) is that their appearance does not always suggest their function. For example, the meaning of the UNIX command `grep` can be deciphered only through prior knowledge, or perhaps cleverness and familiarity with the UNIX editor, `ed`. The appearance of `grep` connotes nothing to UNIX beginners. (In `ed`, the command */regular_expression/* searches for a substring that matches the regular expression. Preceding this with `g` makes it a global command, specifying that the scope of the search is the whole file being edited. Following the command with `p` specifies that lines with the matching substring are to be printed. So `g`*/regular_expression/*`p`, which can obviously be abbreviated as `grep`, prints all lines in a file that contain substrings that match the regular expression.)

1.3.2 Writability

Writability is a measure of how easily a language can be used to create programs for a chosen problem domain. Most of the language characteristics that affect readability also affect writability. This follows directly from the fact that the process of writing a program requires the programmer frequently to reread the part of the program that is already written.

As is the case with readability, writability must be considered in the context of the target problem domain of a language. It simply is not fair to compare the writability of two languages in the realm of a particular application when one was designed for that application and the other was not. For example, the writabilities of Visual BASIC (VB) (Halvorson, 2013) and C are dramatically different for creating a program that has a graphical user interface (GUI), for which VB is ideal. Their writabilities are also quite different for writing systems programs, such as an operation system, for which C was designed.

The following subsections describe the most important characteristics influencing the writability of a language.

1.3.2.1 Simplicity and Orthogonality

If a language has a large number of different constructs, some programmers might not be familiar with all of them. This situation can lead to a misuse of some features and a disuse of others that may be either more elegant or more efficient, or both, than those that are used. It may even be possible, as noted by Hoare (1973), to use unknown features accidentally, with bizarre results. Therefore, a smaller number of primitive constructs and a consistent set of rules for combining them (that is, orthogonality) is much better than simply having a large number of primitives. A programmer can design a solution to a complex problem after learning only a simple set of primitive constructs.

On the other hand, too much orthogonality can be a detriment to writability. Errors in programs can go undetected when nearly any combination of primitives is legal. This can lead to code absurdities that cannot be discovered by the compiler.

1.3.2.2 Expressivity

Expressivity in a language can refer to several different characteristics. In a language such as APL (Gilman and Rose, 1983), it means that there are very powerful operators that allow a great deal of computation to be accomplished with a very small program. More commonly, it means that a language has relatively convenient, rather than cumbersome, ways of specifying computations. For example, in C, the notation `count++` is more convenient and shorter than `count = count + 1`. Also, the **`and then`** Boolean operator in Ada is a convenient way of specifying short-circuit evaluation of a Boolean expression. The inclusion of the **`for`** statement in Java makes writing counting loops easier than with the use of **`while`**, which is also possible. All of these increase the writability of a language.

1.3.3 Reliability

A program is said to be reliable if it performs to its specifications under all conditions. The following subsections describe several language features that have a significant effect on the reliability of programs in a given language.

1.3.3.1 Type Checking

Type checking is simply testing for type errors in a given program, either by the compiler or during program execution. Type checking is an important factor in language reliability. Because run-time type checking is expensive, compile-time type checking is more desirable. Furthermore, the earlier errors in programs are detected, the less expensive it is to make the required repairs. The design of Java requires checks of the types of nearly all variables and expressions at compile time. This virtually eliminates type errors at run time in Java programs. Types and type checking are discussed in depth in Chapter 6.

One example of how failure to type check, at either compile time or run time, has led to countless program errors is the use of subprogram parameters in the original C language (Kernighan and Ritchie, 1978). In this language, the type of an actual parameter in a function call was not checked to determine whether its type matched that of the corresponding formal parameter in the function. An **`int`** type variable could be used as an actual parameter in a call to a function that expected a **`float`** type as its formal parameter, and neither the compiler nor the run-time system would detect the inconsistency. For example, because the bit string that represents the integer 23 is essentially unrelated to the bit string that represents a floating-point 23, if an integer 23 is sent to a function that expects a floating-point parameter, any uses of the parameter in the function will produce nonsense. Furthermore, such problems are often difficult to diagnose.[3] The current version of C has eliminated this problem by requiring all parameters to be type checked. Subprograms and parameter-passing techniques are discussed in Chapter 9.

1.3.3.2 Exception Handling

The ability of a program to intercept run-time errors (as well as other unusual conditions detectable by the program), take corrective measures, and then continue is an obvious aid to reliability. This language facility is called **exception handling**. Ada, C++, Java, and C# include extensive capabilities for exception handling, but such facilities are practically nonexistent in some widely used languages, for example C. Exception handling is discussed in Chapter 14.

1.3.3.3 Aliasing

Loosely defined, **aliasing** is having two or more distinct names in a program that can be used to access the same memory cell. It is now generally accepted that aliasing is a dangerous feature in a programming language. Most programming languages allow some kind of aliasing—for example, two pointers (or references) set to point to the same variable, which is possible in most languages. In such a program, the programmer must always remember that changing the

3. In response to this and other similar problems, UNIX systems include a utility program named `lint` that checks C programs for such problems.

value pointed to by one of the two changes the value referenced by the other. Some kinds of aliasing, as described in Chapters 5 and 9, can be prohibited by the design of a language.

In some languages, aliasing is used to overcome deficiencies in the language's data abstraction facilities. Other languages greatly restrict aliasing to increase their reliability.

1.3.3.4 Readability and Writability

Both readability and writability influence reliability. A program written in a language that does not support natural ways to express the required algorithms will necessarily use unnatural approaches. Unnatural approaches are less likely to be correct for all possible situations. The easier a program is to write, the more likely it is to be correct.

Readability affects reliability in both the writing and maintenance phases of the life cycle. Programs that are difficult to read are difficult both to write and to modify.

1.3.4 Cost

The total cost of a programming language is a function of many of its characteristics.

First, there is the cost of training programmers to use the language, which is a function of the simplicity and orthogonality of the language and the experience of the programmers. Although more powerful languages are not necessarily more difficult to learn, they often are.

Second, there is the cost of writing programs in the language. This is a function of the writability of the language, which depends in part on its closeness in purpose to the particular application. The original efforts to design and implement high-level languages were driven by the desire to lower the costs of creating software.

Both the cost of training programmers and the cost of writing programs in a language can be significantly reduced in a good programming environment. Programming environments are discussed in Section 1.8.

Third, there is the cost of compiling programs in the language. A major impediment to the early use of Ada was the prohibitively high cost of running the first-generation Ada compilers. This problem was diminished by the appearance of improved Ada compilers.

Fourth, the cost of executing programs written in a language is greatly influenced by that language's design. A language that requires many run-time type checks will prohibit fast code execution, regardless of the quality of the compiler. Although execution efficiency was the foremost concern in the design of early languages, it is now considered to be less important.

A simple trade-off can be made between compilation cost and execution speed of the compiled code. **Optimization** is the name given to the collection of techniques that compilers may use to decrease the size and/or increase the execution speed of the code they produce. If little or no optimization is done, compilation can be done much faster than if a significant effort is made to produce optimized code. The choice between the two alternatives is influenced by the environment in which the compiler will be used. In a laboratory for beginning programming students, who often compile their programs many times during development but use little code at execution time (their programs are small and they must execute correctly only once), little or no optimization should be done. In a production environment, where compiled programs are executed many times after development, it is better to pay the extra cost to optimize the code.

The fifth factor in the cost of a language is the cost of the language implementation system. One of the factors that explains the rapid acceptance of Java is that free compiler/interpreter systems became available for it soon after its design was released. A language whose implementation system is either expensive or runs only on expensive hardware will have a much smaller chance of becoming widely used. For example, the high cost of first-generation Ada compilers helped prevent Ada from becoming popular in its early days.

Sixth, there is the cost of poor reliability. If the software fails in a critical system, such as a nuclear power plant or an X-ray machine for medical use, the cost could be very high. The failures of noncritical systems can also be very expensive in terms of lost future business or lawsuits over defective software systems.

The final consideration is the cost of maintaining programs, which includes both corrections and modifications to add new functionality. The cost of software maintenance depends on a number of language characteristics, primarily readability. Because maintenance is often done by individuals other than the original author of the software, poor readability can make the task extremely challenging.

The importance of software maintainability cannot be overstated. It has been estimated that for large software systems with relatively long lifetimes, maintenance costs can be as high as two to four times as much as development costs (Sommerville, 2010).

Of all the contributors to language costs, three are most important: program development, maintenance, and reliability. Because these are functions of writability and readability, these two evaluation criteria are, in turn, the most important.

Of course, a number of other criteria could be used for evaluating programming languages. One example is **portability**, or the ease with which programs can be moved from one implementation to another. Portability is most strongly influenced by the degree of standardization of the language. Some languages are not standardized at all, making programs in these languages very difficult to move from one implementation to another. This problem is alleviated in some cases by the fact that implementations for some languages now have single sources. Standardization is a time-consuming and difficult process. A committee began work on producing a standard version of C++ in 1989. It was approved in 1998.

Generality (the applicability to a wide range of applications) and **well-definedness** (the completeness and precision of the language's official defining document) are two other criteria.

Most criteria, particularly readability, writability, and reliability, are neither precisely defined nor precisely measurable. Nevertheless, they are useful concepts and they provide valuable insight into the design and evaluation of programming languages.

A final note on evaluation criteria: language design criteria are weighed differently from different perspectives. Language implementors are concerned primarily with the difficulty of implementing the constructs and features of the language. Language users are worried about writability first and readability later. Language designers are likely to emphasize elegance and the ability to attract widespread use. These characteristics often conflict with one another.

1.4 Influences on Language Design

In addition to those factors described in Section 1.3, several other factors influence the basic design of programming languages. The most important of these are computer architecture and programming design methodologies.

1.4.1 Computer Architecture

The basic architecture of computers has had a profound effect on language design. Most of the popular languages of the past 60 years have been designed around the prevalent computer architecture, called the **von Neumann architecture**, after one of its originators, John von Neumann (pronounced "von Noyman"). These languages are called **imperative** languages. In a von Neumann computer, both data and programs are stored in the same memory. The central processing unit (CPU), which executes instructions, is separate from the memory. Therefore, instructions and data must be transmitted, or piped, from memory to the CPU. Results of operations in the CPU must be moved back to memory. Nearly all digital computers built since the 1940s have been based on the von Neumann architecture. The overall structure of a von Neumann computer is shown in Figure 1.1.

Because of the von Neumann architecture, the central features of imperative languages are variables, which model the memory cells; assignment statements, which are based on the piping operation; and the iterative form of repetition, which is the most efficient way to implement repetition on this architecture. Operands in expressions are piped from memory to the CPU, and the result of evaluating the expression is piped back to the memory cell represented by the left side of the assignment. Iteration is fast on von Neumann computers because instructions are stored in adjacent cells of memory and repeating the execution of a section of code requires only a branch instruction. This efficiency discourages the use of recursion for repetition, although recursion is sometimes more natural.