

# Chapter 1: Introducing Computer Programming

## Intended Learning Outcomes

After completing this chapter, you should be able to:

- Explain the difference between computers and other machines.
- Describe the purpose of the microprocessor's instruction set.
- Explain the relationship between the instruction set and machine language.
- List some common tasks that computer programs perform.
- Describe what programmers do.
- Summarize the stages of the software development cycle.
- Explain the importance of writing and communications for programmers.
- Explain the relationship between high-level programming languages and machine language.
- Distinguish between the purpose of a compiler and an interpreter.
- Explain the difference between standalone and network applications.
- Explain the difference between programming languages and markup languages.

## *Introduction*

Welcome! By the time you complete this book, you will have a good grasp of the logic and design of computer programs. You will have lots of opportunity to explore numerous sample programs, test your understanding, and develop your own Web-based applications. In the process you will learn a lot about syntax, control structures, application design, and best practices that will help you advance your personal goals, whether to become a professional programmer or Web developer, or to simply make use of these skills for personal interest.

To get started, in this first chapter we will explore the general process of programming and define some important term and practices. For a book that is supposed to be hands-on this chapter is mostly descriptive! Don't be too concerned if some of the topics don't make complete sense yet. Your understanding will deepen as you work through the chapters and develop your own applications

## *What is a computer program?*

A computer is a **programmable machine**. Most machines, such as vacuum cleaners or ceiling fans, are **hard-wired**, designed to perform one task only. But a computer is different. Computers can perform any number of tasks by reading and executing **computer programs**, or **software**. Each computer program contains instructions to drive the computer's hardware for a specific purpose. This ability to read programs is achievable because each computer contains a **microprocessor** which includes the computer's **instruction set**. The instruction set defines all of the basic commands that the computer can execute. These basic commands are very low-level activities

such as adding numbers, moving a piece of data from one location to another, or comparing values.

If you've ever wondered why some programs run on one computer but not another, it is because the computers have different instruction sets stored on their microprocessor. Each instruction in the instruction set is identified by a number, and so program instructions are issued as a list of numeric commands. These commands constitute the computer's **machine language**, which is the language that the computer actually understands. So a computer program is simply a set of instructions that tell the microprocessor to execute machine language commands in a particular sequence in order to achieve some purpose, for example to play a game, process employee wages, display an image, or communicate with another computer.

Although different computer programs may serve quite different purposes, all programs share some important characteristics. Here are some common tasks that any computer program might typically perform:

**Provide interactive environments for users:** programs may use text-based input/output or Graphical User Interfaces (GUI's) to interact with users. Interfaces may include graphics, animations, audio, video, and other multimedia features.

**Read and write data:** programs may create and delete files and databases, and read (query), write and modify (update) data in files and databases.

**Perform numerical calculations:** programs can add, subtract, multiply, and divide, and can build on these operations to perform much more complex calculations.

**Perform text-processing operations:** programs can validate data, convert data, search, sort and replace text, construct reports, messages or documents such as this textbook.

**Communicate with other programs and devices:** programs may exchange data with other programs, cameras, scanners, Web browsers, satellites, cell phones, ATM machines, etc.

**Control hardware:** programs can control robots, satellites, aircraft, automobiles, printers, and other computers.

A single computer program may perform any combination of these operations. For example a computer game may look up player and game information in a file or database, provide an interactive multimedia environment for the player to play the game, perform numerical calculations, and communicate with programs running on other computers to allow multi-user playing. Similarly a payroll program may perform text-processing operations (such as validation and conversion), query and modify a database, perform numerical calculations, and send checks to a printer. What this means is that, as a programmer, you will want to know how to write programs that might include any or all of these operations.

## ***What do programmers do?***

Programmers write program code, right? Well, actually programmers do a lot more than write code, and many programmers actually write very little code. One of the things that makes programming an attractive career for many men and women is that this work requires an appealing combination of **right-brain** (creative, problem-solving, brain-storming) and **left-brain** (logical, linear, sequential) activities. Another appeal is that there are a variety of career paths within the field, so you if you have a general aptitude you have a good chance of finding work suited to your personal interests. For example if you like to work with people you may find yourself drawn to software design, interface development, usability, or training. If you prefer to work with data, you may prefer server-side development, object design, or database administration.

Let's walk through the major stages in software development. This will clearly demonstrate the range of skills that programmers need in order to be successful in their work. Here we will simply summarize these steps so don't be concerned if this appears a little abstract right now. We will learn how to apply these steps to develop a working application in Chapter 3.

## 1. Evaluation of requirements

In order to develop an effective software application we first need to determine the program's requirements. This usually requires careful reading and listening, asking questions, and careful documentation. Understanding and documenting requirements takes time and skill. A common mistake that beginning (and not just beginning) programmers make is to rush this important step in order to begin coding. Rushing the requirements phase can actually be very costly in terms of time, money and even professional relationships. So learn to go slowly and carefully when you are presented with a requirements document or when you first meet with a client. The more time you spend analyzing your program requirements (and asking questions if you are unsure about anything), the more easily the solution will appear. To repeat: the most important skills a programmer needs at this phase are to listen, read, ask questions, document carefully, and communicate effectively with clients, managers, and other programmers.

## 2. Software design

Once you have a good idea of the software requirements, it's time to develop the design of your application. This may be a one-person job in the case of a small application or may involve a design team. Software design can actually become a career path and some (often the most experienced) programmers spend most of their time evaluating requirements and designing applications that other programmers will then code.

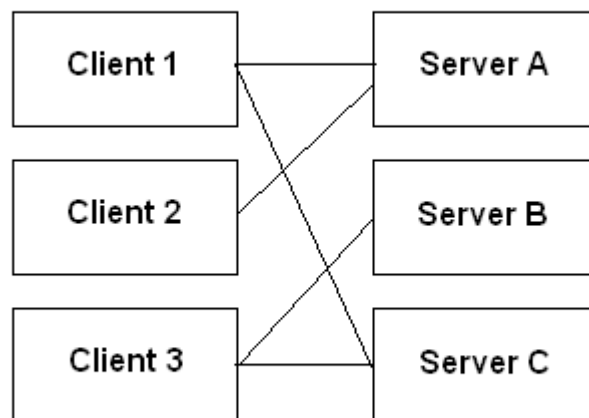
Unless your requirements are quite simple, your application design will most likely consist of multiple code segments or **modules**, each of which can be developed separately. There are many advantages to a modular approach to software. Each module can be developed and tested **separately**, often by different programmers or programming teams. The modules can be developed **concurrently** which speeds up development time. This approach also allows each module to be developed by programmers with the most **suitable skills**. And a modular approach promotes **reusability**: useful modules can be shared by many different applications. Module

design includes the design of testing procedures to test that a module will perform as expected when it is coded.

At this time, software designers commonly apply two very successful and widely-used strategies in order to achieve program modularity: **Client/server design** and **Object oriented programming (OOP)**. These approaches are not exclusive and are both often integrated into the design process. Here is a brief overview of client/server design and OOP:

**Client/server design:** a client application usually provides an interface to the user, waits for the user to request some action, then calls a server application to process the request and send back a response. When the client application receives the response this is presented to the user and the process repeats. A common example of a client/server program is a Web application, where the Web browser performs as a client, sending the user's requests to a Web server for processing and then displaying the response that is received from the server, so that the user can review the results and decide what to do next. You participate in a client/server interaction every time you use your browser to click a link, type and submit a URL, or submit a Web-based form.

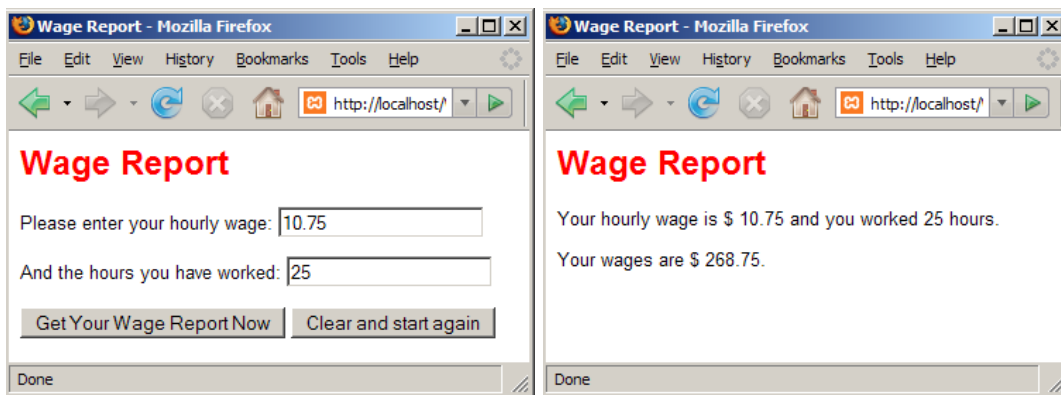
An important aspect of client/server design is that a client application may call any number of server applications as needed, and these applications only execute when they receive a request. A server application may receive requests from many clients, for example a Web server for a major newspaper or online store may receive and respond to thousands of requests from different users every second. Figure 1-1 illustrates a number of clients contacting various servers for different purposes.



**Figure 1-1: Client/server interactions**

A useful design tool associated with client-server design is **storyboarding** where programmers sketch out an application as a series of screens (the user interface). Each screen provides the user with information and provides options to request services from one or more server applications. This is a useful way to consider the application from the point of view of the user, and is helpful in defining the various

tasks that the application must perform. For example, Figure 1-2 shows the screens for a very simple client/server application to calculate an employee's pay based on their hours worked and hourly wage.



**Figure 1-2: A simple client/server interaction**

This example provides the user with two Web pages. When you type the URL to request the first page, the Web browser sends a request to the appropriate Web server, which sends back the data to display the page. The first page contains a form. When the user clicks the "Get Your Wage Report Now" button, the Web browser sends a second request to the Web server that includes the data that the user has entered into the form. The server executes a program that has been developed especially to process these inputs. This program generates the content for the second page, which the server sends back to the browser for display to the user.

Client/server applications may consist of a large number of screens and related interaction. Three primary advantages of client/server design are:

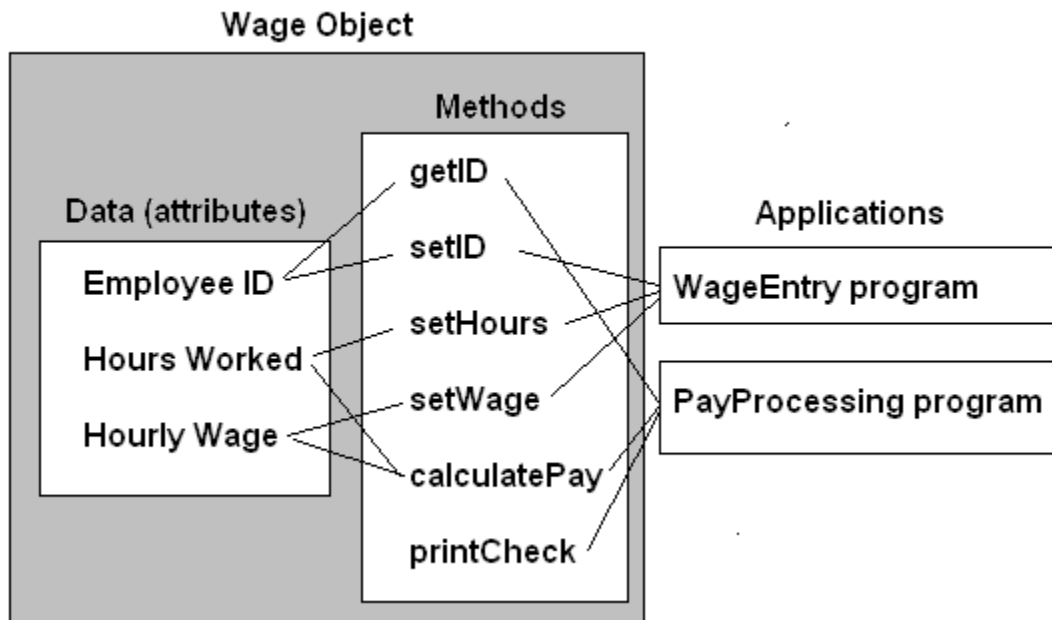
- A single server can deliver the same services to any number of clients.
- Changes to the interfaces and processing instructions for the application can be made on the server with no need to make changes on the client machines
- There is a high level of security since all data and processes are located on a single server.

In this book you will develop many small client/server applications similar to the example described above.

**Object Oriented Programming (OOP):** OOP allows a software designer to develop code independently of any particular software application. Instead code is developed to permit a broad range of **operations** to be performed on a specific **data set** of some kind, and this code can be used by any number of applications as needed. In OOP, the items that comprise the data set are known as **attributes** or **fields**, and the operations are known as **methods**. An **object** consists of a set of attributes along with the methods that are provided to work with the attributes.

To take a simple example, the data set (attributes) associated with an employee's wages might consist of an employee's ID, his or her hourly wage, and the hours that he or she worked. The operations (methods) that can be performed on this data might include obtaining the employee's ID, setting the ID, setting the hours worked,

setting the hourly wage, calculating the weekly pay, printing a pay check, etc. Together these attributes and methods define a Wage object. Once the object has been developed, different applications can use it for different purposes. Our Wage object might be used by a program that allows administrative assistants to enter employee wage information, and also by a program that processes weekly checks. The wage entry program might use the methods to set the ID, hours worked and hourly wage of a Wage object, while the pay processing program might use the methods to obtain the employee ID, calculate the pay, and print a paycheck. Figure 1-3 illustrates this example.



**Figure 1-3: An example of Object Oriented Design**

What makes object oriented programming especially important is that objects are usually designed and coded for use by any number of programs. The designer of the object is not thinking about any particular use for the data set, but is rather planning for all possible uses. Once an object has been developed in this way, it can be used by many different applications as needed, with no need to duplicate the code.

There are many other advantages to object oriented design. For example, we say that the data within an object is **encapsulated**, meaning that it is only directly accessible by the object's own methods. The methods therefore provide an **Application Programming Interface (API)** to the data. This approach prevents the inappropriate or incorrect use that could occur if applications could access the data directly.

Objects are defined for many different types of data. As a very different example the data set (attributes) associated with a clickable button that is to be displayed on your computer screen might consist of the button's background color, the text that is to be displayed on the button, the color of the text, whether or not the button is currently active, etc. The operations (methods) that can be performed on a button will include changing the colors, setting the text that the button displays, checking whether the button has been clicked, etc. Together these attributes and methods

might constitute a Button object. Once a button object has been defined, any programs that need to display buttons on the screen can create a Button object.

While object oriented programming is not covered in any detail in this book, Chapter 12 provides an introduction to this topic that will help you get started and prepare for further study.

Whether your programs are simple or complex, it helps greatly to stay away from a computer in the early phases of application design! Explore your design ideas using a pen and paper, sticky notes, a white board, even the backs of napkins! If you go to work as a programmer, you will find that this is how software design teams usually work. The reason is simple: using disposable materials prevents you from becoming too invested in any particular approach too quickly and encourages brainstorming and creativity. You will find that sketching out ideas on paper before you start coding will help you think things through and save you time in the long run. Once you have a clear idea of what to do you will be ready to develop your algorithms and application structure.

Important skills for software designers are creative thinking, organization, familiarity with data structures, a background in object oriented programming, writing documentation, and experience with client-server programming and interface design.

### 3. Algorithm development

Once you have a general design for your application, it is time to develop **algorithms** for each code module that the application will need. An algorithm is simply a set of clearly written, unambiguous instructions that have been developed to perform a task of any kind. Algorithms are a critical component of software design, often written in some mix of English and programming language syntax (known as **pseudocode**) that programmers can easily understand. You will learn to develop algorithms using pseudocode in Chapter 3. The actual work of coding an application is ideally a fairly straightforward process of converting a carefully developed algorithm into a specific programming language.

The skills required for these activities include careful attention to detail, logical thinking, documentation (writing), and general programming experience.

### 4. Application coding

This is the activity that is usually associated with programming! The algorithm for each program module is coded into a programming language. Each module is then carefully tested before the modules are assembled to produce the complete application. The most important skill required for coding is knowledge of the appropriate programming language, but programmers are expected to also have the experience to develop code efficiently, test thoroughly, find and fix errors (debugging) and document the code so that other programmers can refer to the documentation as needed.

Coding, testing, debugging and documenting require patience, thoroughness, and careful attention to detail.

## 5. Application testing

Once the application has been assembled, it is time for thorough testing. While the development team may perform many tests for **correctness**, it is also important to test for **usability**. A development team may produce a terrific application that has been thoroughly tested and debugged but turns out to be a disaster when provided to end users. Why? Because the end users find the interface confusing and cannot easily perform the tasks that are most important for their purposes. Usability testing brings the users into the development process and ensures that the needs and concerns of the user are taken into account before the product is distributed. In the case of larger applications, usability testing is often undertaken by usability experts who may observe users working with the product to determine where improvements can be made. In smaller applications, the programmer may simply work with the client to find problems and get feedback.

Testing takes a great deal of patience, attention to detail, and thoroughness. Usability testing requires good listening skills, careful observation, and (if you are the programmer) humility! It's very easy for any programmer to be so focused on his or her own design that the needs of the user become secondary. If you notice yourself getting impatient with a user who cannot figure out how to use your product, or who wants the software to perform differently, then it's probably time to step back, pay attention to the user's concerns, and reconsider your own design assumptions.

## 6. User support, training, software maintenance

So now you have a complete and well-tested application. The development process does not end there! You will also need to develop online or printed manuals and other documentation for use by your end users. You may also need to deliver some form of training. And the software must be maintained. Users will find problems that must be corrected, and suggest additions and improvements that will need continued programming support. User support and training is a career path for those who like to work with non-technical people and who can also communicate effectively with programmers and software designers.

Important skills are the ability to communicate with both technical and non-technical people, the ability to listen and to explain (verbally and in writing) procedures clearly and carefully, patience, and often a sense of humor!

### *The software development life cycle*

Taken together these activities constitute the software development life cycle:

- Evaluation of requirements
- Application design
- Algorithm development
- Application coding
- Application testing
- User support, training, software maintenance

This is not a precise list - in reality these various stages may not be so neat and sequential, and you will see somewhat different versions of this process in every programming textbook and in every workplace. Development teams may implement these various stages differently. But no matter how these stages are defined, no part of the development cycle should be treated carelessly. As you gain experience as a programmer you will more fully appreciate the special characteristics and importance of every step. And perhaps you can already see how the field of software design attracts people who are both creative and logical, who enjoy using both the left and right sides of their brain equally.

## ***The importance of writing and communicating***

Documentation and writing are frequently mentioned as important skills for programmers. Documentation is critical to software development and wherever your own career path takes you in the field of software design and development, you will need to be able to write carefully and communicate effectively. Clients, designers and managers must refer to well-written documents that clearly define requirements, design and code specifications. Everyone involved in the development process must listen carefully and communicate effectively. Programmers must document their working code so that other programmers can easily read and modify it (often the programmer who develops a piece of code will **not** be the programmer who is asked to make changes for the next version). Programmers must often give presentations to clients or to their team. The testing phase also requires extensive documentation that indicates what tests were applied, the results of these tests, and how problems were resolved. Software users will need course manuals and training materials. Lastly all maintenance procedures must be documented so that a complete record is always available regarding the current state and history of the software.

## ***What are programming languages?***

We have explored what a computer program **is**, and what programmers **do**, but what about programming **languages**? What is a programming language and why are there so many different languages?

As we have seen, a computer program is basically a sequence of instructions that direct the computer's microprocessor to perform various commands contained with the computer's **instruction set**. These commands are issued in **machine language**. Machine language commands are very **low-level** (for example adding two numbers, or copying a value from one memory location to another). It would be extremely time-consuming to write instructions in the 0's and 1's of machine language, and this code would be very error-prone and difficult to debug, maintain, or modify.

Instead of using machine language directly, we develop programs using **high-level programming languages**. Examples of current high-level languages are C++, C#, Java, PHP, Python, and Ruby. Examples of older high-level languages are C, Ada, BASIC, COBOL. Fortran, Pascal, and perl (older does not necessarily mean no longer used – many applications written in older languages are still in widespread use, and programmers are still needed to maintain and even update these programs).

A high-level programming language consists of a set of special words, symbols and operators that a programmer uses to write program instructions. These instructions

are often referred to as the program's **source code** and the process of writing source code is often simply called **coding**. High-level languages are quite easy for programmers to learn, and applications written in these languages can be developed very quickly and efficiently. However the computer can only understand machine language, so once a program has been written in a high-level language, the code must then be translated into machine language instructions. There are actually two approaches to translating high-level code to machine language, either by **compiling** the code or by **interpreting** the code. The approach depends on the programming language that you are using.

## ***Compilers and Interpreters***

Some programming languages are **compiling** languages. This means that the entire source code for a program is converted (or compiled) into an executable file (**.exe** file) by special software known as a **compiler**. The **.exe** file that is produced by the compiler contains the necessary machine language instructions to perform the required task. Once a program has been compiled into an executable file, the source code is no longer required to run the program. End users of the program simply receive the **.exe** file. The programmers keep the source code in order to perform updates and produce new versions. Usually when you purchase standalone software from a store you are buying an executable program that has been compiled. Two advantages of compiled programs are:

- Compiled programs tend to run faster
- The end user does not have access to the source code (and so cannot change the program).

Since a compiler generates an **.exe** file containing the machine language instructions for a specific microprocessor and operating system, the source code must be compiled separately for different platforms (Windows, Macintosh, Linux, etc). Also, each time the source code is modified, the new version must be compiled again to produce a new **.exe** file. The new **.exe** file must then be distributed to the end users.

Other programming languages are **interpreted** languages. Execution of programs written in interpreted languages is dependent on a special program known as an **interpreter**. An interpreter translates the source code into machine language one instruction at a time. Both the interpreter and the source code are needed **every time that the program is executed** since no executable file is created. One advantage of this approach is that the same source code can be used on any computer. For example computers running Windows, Macintosh or Linux operating systems can each use their own language interpreter to translate the source code into the appropriate machine language for that platform.

A disadvantage of using an interpreter is that the original source code must be distributed in order for the program to be used. But this approach works well for network-based programs since these programs are not distributed to users. In these cases, the source code is located on a server computer and executed each time a request is submitted by a client application. The source code can be modified quickly and easily with no need to recompile and redistribute the software every time a change is made. PHP is an interpreted programming language that is widely used to develop server-based Web applications.

Some languages provide both compiling **and** interpreting options, and some languages actually **combine** compiling with interpreting stages to achieve greater efficiency and platform independence. A notable example is the **Java** language. Java applications are compiled up to a point to produce an "almost executable" version known as **byte code** that incorporates many of the efficiencies of the compilation process, and then Java interpreters are provided for different platforms so that the same byte code can be distributed for execution on any machine (Windows, Macintosh, Linux, etc.) in order to achieve platform independence.

## ***So many languages!***

As computer technology evolves, new programming languages are continually developed to take advantage of the latest hardware and software design strategies. For example new languages were developed to implement the functionality of object oriented programming, and to allow client/server application development. There are literally thousands of different programming languages and often a computer programmer is expert in only a few of these. While each programming language has its own special syntax and characteristics, most languages are very similar in their overall characteristics and functionality, and use the same basic logical structures to write instructions. We will learn about these common characteristics as we work through this book. A programmer who is familiar with the general logic of programming, and who has experience coding in one or two languages can usually learn new languages quite quickly.

## ***Standalone and Network Applications***

Computer programs can be designed for use on individual machines (as **standalone** applications), or across networks (as **network** applications).

A standalone application is designed to provide a complete service on the **local** computer, usually the computer sitting on the user's desk. Standalone applications do not require any network connectivity, interacting only with the computer's **operating system** and other **utility software** on the local machine. If a new version of the application becomes available, or if updates are required, these must be also installed on every user's computer. Examples of standalone software are word-processing applications, spreadsheet applications, image-processing software, many games, etc. At this time, most of the programs that you install on your local computer are standalone applications.

**Network applications** are programs that run partly or entirely on **remote** computers, linked to the user across a network of some kind. The more traditional network application was simply installed on a single host computer and then accessed by many users remotely. Each user that signs on to the host computer is provided a user interface to work with the application. An increasingly important type of network application is a **client/server** application which consists of any number of component programs that work together across a network. Some components of a client/server application can be installed on a local computer, and perform as **client** programs. Other components of a network application are installed on network **servers** and perform as **server** programs. The server-based components respond to requests from client components of the application as needed. At the minimum the client component usually provides the user interface that allows the user to submit

information and view the results, while the server component does most of the processing.

You are using a client/server application whenever you use your Web browser. The **Web browser** performs as a local **client** component, providing your user interface and allowing you to send requests to **server** programs all over the world. So you are using a client/server application whenever you use your Web browser to obtain information, shop online, or play an online game. Other common examples of client/server applications are ATM's and e-mail programs.

Client/server applications are becoming more and more common because they allow us to obtain services and perform tasks without need for special software on our local computer. We can expect to make increasing use of Web-based client/server applications since these require only a local Web browser and an Internet connection, rather than special standalone applications that must be installed locally and continually updated.

## ***Markup Languages***

So far we have discussed the purpose of **programming** languages. As you now know, the purpose of a programming language is to allow a programmer to write instructions that **process** data. In other words, programming languages are used to perform operations that change existing data or generate new data for some purpose (for example to calculate wages, convert temperatures, or keep track of a game player's score).

Another type of language is a **markup language**. Markup languages are often used in conjunction with programming languages, but have a very distinct purpose. The purpose of a **markup language** is to provide markup instructions (usually in the form of **tags**) that simply **describe** data or indicate how data is to be **formatted**, or **rendered** (for example to define how data should be displayed on a Web page, or printed in a document). Your word-processing program uses a markup language to save formatting instructions with your document as you type a report or letter.

An important markup language now used extensively for data description is **eXtensible Markup Language (XML)**. XML is used to define how data should be structured for different purposes (for example to define the structure of e-mail messages, student records, bank transactions, legal abstracts, medical data, or weather data). These definitions allow us to establish standards that allow computer programs to more easily exchange and process data.

The markup language that is used to render data for display in Web browsers is **Hypertext Markup Language (HTML)**. The latest version of HTML is defined in XML and is known as **XHTML** (we will follow the XHTML standard in this book).

## ***Combining markup and programming languages***

In this course, you will learn the basics of programming by developing small, Web-based client/server applications using a programming language known as **PHP**, one of the most widely used programming languages for this type of application. Since you will use a Web browser to display your program output, you will also learn the

basics of the **HTML** markup language to format your application input and output for display in your browser window.

## **Summary**

A computer is a programmable machine. The computer's microprocessor includes the computer's **instruction set** which defines all of the basic commands that the computer can execute.

Commands to the microprocessor must be issued using the computer's **machine language**. A computer program is a set of machine language instructions that execute a sequence of commands to perform a useful task.

Different programs combine common components to achieve their purpose: interactive environments; read/write operations; numerical operations; text-processing, communication with other programs; control of hardware.

A software designer/developer requires a range of skills that combine creative problem-solving and logical processing. Documentation is an important part of the software process so writing and communication are important skills that are often not associated with this field.

Software development includes a number of stages: evaluation of requirements; application design; algorithm development; application coding; application testing; user support, training, software maintenance

Programs are written in programming languages which may be compiled or interpreted. The code containing the program instructions written in a specific language is known as **source code**. If the language is a **compiler**-based language the source code must be converted into an executable version which is then distributed for use. If the language is an **interpreted** language, the source code itself is distributed and this is then executed one line at a time by a language interpreter.

Some programs are designed to function as **standalone applications**, which means that they are installed locally and do not need access to other networks. A copy of a standalone application is required for every user. **Network applications** run over networks. A single network application can be installed on one computer and accessed across a network by many users.

An increasingly important type of network application is a **client/server application**, where client programs on local computers send requests to server programs on remote computers. These requests are processed and results returned to the client. A common example is a **Web-based client/server application** where a user's Web browser performs as a client to request services from server applications throughout the world.

Markup languages are not the same as programming languages. Programming languages provide instructions to **process** data. Markup languages provide tags to **describe** or format (**render**) data

In this course you will learn to develop simple Web applications using the **PHP** programming language and **HTML** markup language.

## ***Chapter 1 Review Questions***

1. A web application is an example of
  - a. Object Oriented Programming
  - b. Client/server design
  - c. A microprocessor
  - d. A standalone application
  
2. A program that requires the source code each time that it executes is using which method to convert the source code to machine language?
  - a. A compiler
  - b. An interpreter
  
3. Which approach is better when evaluating software requirements?
  - a. Determine the requirements as quickly as possible in order to move on to the design and coding phases.
  - b. Take time to analyze and clarify the application requirements.
  
4. What kind of thinking activities are most associated with the work of a programmer?
  - a. Left brain activities
  - b. Right brain activities
  - c. Both left AND right brain activities
  - d. Neither left NOR right brain activities
  
5. Which approach to software design focuses on data first?
  - a. Storyboarding
  - b. Object Oriented Programming
  - c. Client/server design
  - d. The microprocessor
  
6. Which language does the computer actually understand when it executes instructions for a program?
  - a. Markup language
  - b. High-level programming language
  - c. Object-oriented language
  - d. Machine language
  
7. What is the computer's instruction set?
  - a. The set of all programming languages that a computer can understand
  - b. The set of all software that is currently available on the computer
  - c. The basic set of commands that a computer can execute
  - d. The rules for using a high-level programming language

8. What is source code?
- a. Programming instructions written in a programming language
  - b. Program instructions that have been compiled into machine language
  - c. The code used to identify text characters from languages all over the world
  - d. The code used to identify memory addresses
9. What does an executable file contain?
- a. Programming instructions written in a programming language
  - b. Program instructions that have been compiled into machine language.
  - c. Formatted text
  - d. An audio image
10. Which term applies to an application model where one program calls another program in order to have some task performed?
- a. Client/server
  - b. Standalone program
  - c. Instruction set
  - d. An algorithm
11. What does an interpreter do?
- a. Reads and executes source code, one line at a time
  - b. Converts source code into an executable file.
  - c. Sends data from one program to another
  - d. Converts and displays text that has been marked up
12. What does a compiler do?
- a. Reads and executes source code, one line at a time
  - b. Converts source code into an executable file
  - c. Sends data from one program to another
  - d. Converts and displays text that has been marked up
13. When you use your Web browser to access information you are working with
- a. A client/server application
  - b. A standalone application
14. What is an algorithm?
- a. A set of instructions to meet a requirement of some kind
  - b. An executable file
  - c. Program instructions that have been compiled into machine language
  - d. Programming instructions written in a programming language
15. Which of the following is **not** a feature of modular application design?
- a. Concurrent development of each module
  - b. Reusable code

- c. Separate testing of each module
  - d. Less time needed to evaluate requirements
16. What category of language is used to describe data?
- a. Markup language
  - b. High-level programming language
  - c. Object oriented language
  - d. Machine language
17. What languages will you learn in this course?
- a. C++ and HTML
  - b. XML and Java
  - c. PHP and FORTRAN
  - d. PHP and HTML
18. Which is the correct order for these stages in the software development life cycle?
- a. application coding, application development, algorithm development
  - b. application development, algorithm development, application coding
  - c. algorithm development, application development, application coding
  - d. algorithm development, application coding, application development
19. Is a Web browser a client application or a server application?
- a. Client
  - b. Server
20. In the world of object oriented programming, operations on data are referred to as:
- a. Attributes
  - b. Fields
  - c. Methods
  - d. Actions