Chapter 1

Computers and Programming

We begin this text by describing the fundamental aspects of computers and programs, and we present computer programming as a three-step process:

1. *designing an architecture*
2. *defining the classes that comprise the architecture*
3. *writing the Java instructions that realize each class.*

1.1 What is a Computer?

Electronic computers can be found almost everywhere, but in general terms, a *computer* is any entity that can follow orders, more specifically, that can execute instructions. This classification includes humans (who are imperfect computers) as well as pocket calculators, programmable disc players, and conventional "PC"s.

This text is concerned with conventional computers, which must possess (at least) these two components:

- one (or more) *processors*. A processor executes instructions (e.g., arithmetic calculations or instructions to draw colors and shapes).

- *primary storage* (also known as "memory," "random access memory," or "RAM"). Primary storage acts as an electronic "scratch pad" that holds the instructions that the processor reads and executes and holds numbers and information ("data") that the processor calculates upon—primary storage is like the scratch pad that an accountant (a human "processor") uses.

Most computers have also *secondary storage* devices—internal disk drive ("hard drive"), compact disk drive, and diskette drive ("floppy drive")—whose data must be copied into the primary storage component before it can be read by the processor. Secondary storage is designed for portability (e.g., diskettes that can be carried from one computer to another) or for holding massive amounts of data, much like an accountant's filing cabinet can hold more information than the accountant's scratch pad.

The filing cabinet analogy just mentioned has generated some standard computer terminology: A *file* is a collection of instructions or data; files are themselves grouped into *folders* (also known as *directories*). Directories are normally kept in secondary storage, their contents copied into primary storage when needed by the processor.

Finally, a computer usually has several *input-output devices* (such as the display, keyboard, mouse, printer), which let a human supply information to the computer (say, by typing at the keyboard or clicking a mouse) and receive answers (by reading it on the display or from the printer).

If a computer can execute instructions, what kind of instructions can it execute? This depends—a conventional notebook computer of course cannot execute the instructions written in a cookbook, nor can it follow the instructions for driving a car from Chicago to Manhattan. (But there are special-purpose computers that *can* attempt the latter.)

The processor inside a typical computer performs a limited range of arithmetic-like instructions—numeric addition and subtraction and text copying. But to do even these simple tasks, numbers and text must be coded in sequences of 1's and 0's, called *bits*. This technique of writing numbers, text, and instructions to a computer is called *binary coding*. The collection of binary-coded instructions that a computer's processor can read and execute is called the computer's *machine language*. By using binary codings, a computer can compute with letters as well as numbers. Images (with colors, brightness, and shapes) can also be coded with binary codings.

Exercises

1. If a computer is indeed any entity that can follow orders, then give examples from real life of "computers."

2. List all the input-output devices that are attached to the computer you will use for your programming exercises.

3. Hand-held calculators are computers. What are the input-output devices for a calculator?

1.2 Computer Programming

Programming is the activity of writing instructions that a computer can execute. For

Figure 1.1: a program for baking "lemon cake"

```
1. Squeeze the juice from one lemon into a large bowl.
2. Cut the lemon's rinds into small pieces and add them to the bowl.
3. Mix three eggs, one cup of sugar, two tablespoons of flour,
   and one cup of milk into the bowl.
4. Pour the mixture into a square cake pan.
5. Heat an oven to 350 degrees Fahrenheit.
6. Insert the pan into the oven and wait 40 minutes.
```

the moment, forget about conventional computers—if your spouse can read and can operate an oven, then when you write instructions for baking a cake, you are "programming" your spouse—the instructions are a program, and you are a *programmer.*

The point here is that a *program* (in this case, for baking a cake) is a list of instructions written in a precise style where declarative verbs ("cut," "pour," "heat") state computational actions, and nouns ("egg," "flour," "bowl") state data with which computation is performed; see Figure 1 for an example.

A sequence of declarative instructions for accomplishing a goal, like that in Figure 1, is called an *algorithm.* The term *computer program*, is used to describe an algorithm that is written precisely enough so that a computer processor can read and execute the instructions. For historical reasons, the writing of computer programs is sometimes called *coding*, and the program itself is sometimes called *code.*

A computer program is almost always saved as a file on secondary storage. When someone wants the processor to execute the program's instructions, the programmer *starts* the program—this causes the program to be copied into primary storage, where the processor reads the instructions and executes them. Starting a program might be done with a mouse click on a program's *icon* (its picture on the display) or by typing some text with the keyboard.

Unfortunately, computer processors execute instructions written in machine language—the language of binary codings—and humans find this language almost impossible to write. For this reason, other languages have been designed for programming that are easier for humans to use. Examples of such *high-level programming languages* are Fortran, Cobol, Lisp, Basic, Algol, Prolog, ML, C++, and Java. Many of these languages look like the language of secondary-school algebra augmented with a carefully defined set of verbs, nouns, and punctuation symbols.

In this text, we use Java as our programming language.

When one writes a program in Java, there remains the problem of making the computer's processor understand the program. This problem is solved by the Java designers in two stages:

1. First, a translator program, called a *compiler*, is used to translate Java programs

into another language, called *Java byte code*, which is almost machine language. The programmer starts the compiler and tells it to translate her Java program to byte code. The resulting byte code is deposited, as a file, on secondary storage.

2. Next, when the programmer wishes the byte code executed, she starts a second program, the *Java interpreter* (also known as the *Java virtual machine* or *JVM*). The interpreter copies the byte code to primary storage and works with the processor to execute the machine-code versions of the byte-code instructions.

Programming languages like Java are useful for programming these kinds of activities:

- *scientific and mathematical calculations*, such as calculating the roots of a quadratic equation, drawing the curve for a polynomial, or printing a table of monthly payments for varying interest rates on a mortgage.
- *information processing*, such as editing and typesetting a letter, printing a file of paychecks, or drawing pictures on the console.
- *simulation*, such as imitating the cockpit of an airplane, simulating the next five days' weather, or playing a card game.

This text intends to show you how to use the Java language to program these activities. Along the way, you will learn practical and formal aspects of writing programs in good style.

Exercises

1. Locate a cookbook and study one of its recipes. Mark the declarative verbs, nouns, and precise quantities. Also, circle any instructions in the recipe that appear to you to be imprecise.

2. Arithmetic is often called the "first programming language." Pretend that this expression is a program

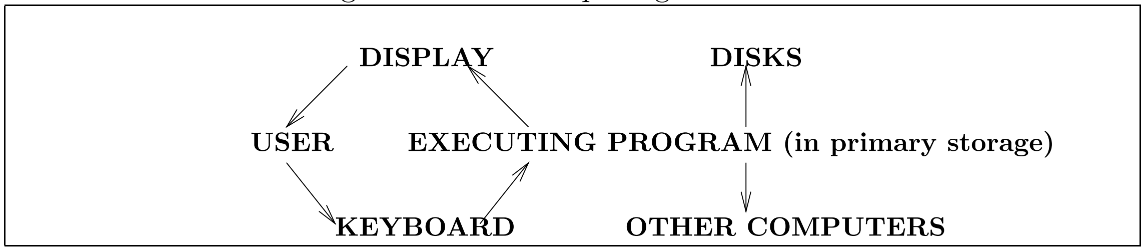
   ```
   (3 + 2) - 1 + (6 + 5)
   ```

   and pretend that you are a computer. List the steps you take with your pencil to execute the instructions in this program. (That is, compute the answer to the expression, one step at a time.)

3. Algebra is a programming language. List the steps you take to solve the value of `x` in this "program":

   ```
   3y - x = 3 + 2x
   ```

Figure 1.2: the computing environment



(Hint: the first step is to add `x` to both sides of the equation, producing the new equation, `3y - x + x = 3 + 2x + x`.)

4. Here is a small fragment of a Java program:

   ```
   int x = 3 + 2;
   int y = x + 1;
   displayOnTheConsole(y);
   ```

   Which parts of this program appear to be verbs? nouns? adjectives? algebraic expressions?

5. Propose a programming language for drawing colored bubbles and squares on a sheet of paper. What verbs will you include (e.g., "draw," "trace")? nouns ("circle," "red")? adjectives ("large," "dark")?

1.3 Programs Are Objects

When a computer program is executing (that is, the program has been copied into primary storage, and the processor is executing its instructions), the program does not "live" in isolation; it is surrounded by an environment consisting of a keyboard, display, disks, and even other computers. Figure 2 pictures such an environment. We use the term, *object*, as a generic term for each of the components in a computing environment. The EXECUTING PROGRAM is itself an "object," as is the computer's USER (a human being), who interacts with the machinery. Indeed, a program can itself consists of multiple objects (smaller programs).

An "object" is meant to be a generalization of a "person"—an object has an "identity," it knows some things, and it is able to do some things.

We use the term, *method*, to describe an object's skill or ability or activity—a thing that an object can do. For example, an executing word-processing program is an object that has methods for inserting words, moving words, correcting spelling errors, and typesetting documents. An executing program that calculates retirement savings is an object with methods for calculating rates of savings, interest rates, payment schedules, etc. And a graphics program is an object that has methods for drawing lines and geometric figures, painting the figures, moving them, and so on.

Similarly, a keyboard is an object that has methods for typing letters and numbers, and a display is an object that has methods for displaying text, colors, and shapes. A user has methods for typing text at a keyboard and reading answers on the display. (Users also have methods for eating, sleeping, etc., but these are not important to computer programming.)

Objects "communicate" with each other—one object can ask another to perform one of its methods. This is called *sending a message.* The arrows in Figure 3 indicate the directions in which messages are sent. For example, a USER might wish to know the square root of a number, so she types the number on the KEYBOARD, in effect sending a message to the KEYBOARD object. The KEYBOARD is the *receiver* of the message. (The USER is called the *client.*) The KEYBOARD reacts to the message by using one of its methods to convert key taps into a number, and it sends a message containing the number to the EXECUTING PROGRAM. The EXECUTING PROGRAM receives the message and uses its methods to compute the number's square root. The EXECUTING PROGRAM then communicates the square root to the DISPLAY, which shows number, as symbols, on the screen so that it reaches the USER's eyes.

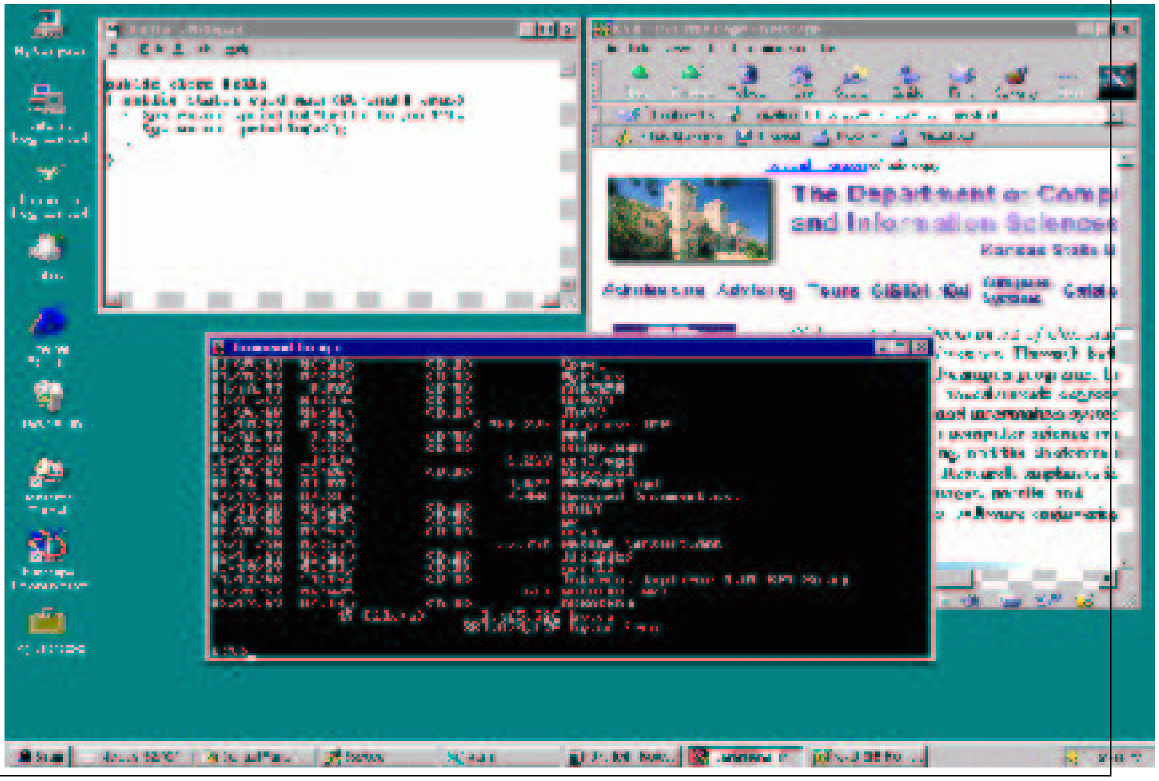
In this way, computation is performed by a series of communications between objects. A complex executing program might itself be a collection of interconnected objects—perhaps computing the square root of a number is completed only after several internal communications between the program's own executing subobjects.

Here is some commonly used terminology: When a human supplies information to a computer program, say, by typing on a keyboard or moving a mouse, the information is called *input* to the program. (Another term is *input events.*) Input can also be supplied to a program from information on a disk. When the program calculates a result or answer and this answer is displayed or saved on a disk, this is called the program's *output.* Programs are often called *software*, which is a pun on the term, *hardware*, which of course describes physical devices such as processors, displays, and disk drives.

1.4 Operating Systems and Windows

When a computer is first started, its processor looks on the computer's disk drive for a program to place first into primary storage to execute; the program it finds is called the *operating system.* An operating system is the computer's "controller program"; by displaying information on the display and by receiving messages from mouse and keyboard, the operating system helps the computer's user execute other programs. Often, a user's request to execute a program is little more than a mouse movement and a click, or it might be the typing of text within a *command window*

Figure 1.3: a multi-window display



```
public class Hello
{ public static void main(String[] args)
    [illegible]
    [illegible]
}
```

(or *command-prompt window*).

Prior to the 1980s, an operating system used the computer's display as one large command window. Programs were started by typing within the window, and the program would read its input data from the window and would display its output within the same window. All input and output were text—words and numbers.

Of course, modern operating systems create multiple windows on the display, where the windows might be command windows or windows created by executing programs. Figure 3 shows a display that holds three distinct windows: a command window, a window created by a word-processing program, and a window created by a Web-browsing program. These windows were created with the behind-the-scenes help of the operating system. The user interacts with the window by moving the mouse into it, typing, clicking, or reading. Icons appear along the left side of the display.

The point of Figure 3 is that the multiple windows that appear on the display are themselves distinct objects. The crude picture in Figure 2 suggested that the DISPLAY was one object, but modern operating systems make it possible for an executing program to communicate with multiple window objects. In a similar way, the multiple folders and files that reside in secondary storage are also distinct objects.

Exercise

Use a computer to start a program, like a game or a word processor. List all the windows that are created by the program, list the ways you give input information to the program, and list the ways the program displays output.

1.5 Software Architecture

The previous section stated that a typical computing environment contains many objects: keyboard, windows, executing program, files on secondary storage, etc. When we do some computing, we do not build objects like keyboards from scratch—we use the one we bought. And we buy programs and use them as well. But this text is about building programs rather than buying them—how do we do this?

Becoming a good programmer is not unlike becoming a good novelist, bridge-building engineer, or architect—time and effort must be invested in studying standard examples of the genre, disassembling and modifying them, learning basic techniques of composition/construction, and working plenty of exercises.

In novel writing, bridge construction, and house building, a beginner is tempted to start on the final product without first investing time in a *design*. A professional knows better. Consider house building: A modern house's design is specified by a blueprint. A blueprint possesses a "style" or *architecture*. A house can have a simple, one-room architecture, or it can have a multi-room, split-level, architecture, or it might even be multiple housing units connected by passageways and stairways.

The house builder uses the blueprint to construct a physical structure with walls, floors, electrical wiring, and plumbing. Almost no one would buy a house that was *not* constructed from a blueprint—the house would not be trustworthy.

Builders read blueprints; most do not design them. The highly trained individual who designs and draws the blueprint is an architect. A well trained architect knows about mathematics, physics, and art as well as house construction.

Machines, such as cars, automated bank tellers, and computers, also have architectures (see Figure 2), *and so do computer programs.* Writing a computer program involves more than merely writing instructions in Java—one must design the program's architecture, specify the architecture's components, and write Java instructions for each component. Here are the crucial steps:

- First, a program's architecture is specified by drawing a picture called a *class diagram.* The class diagram indicates the components, called *classes*, that form the entire program, much like a blueprint shows the rooms that form a house.

- Next, each class is designed as a collection of *methods*.

- Finally, each method is written as a sequence of Java instructions, and the coded methods for each class are saved as a file in secondary storage.

When the computer program is started, objects in primary storage are created from the classes in secondary storage, forming the executing program.

This text relies heavily on one particular architecture, called the *Model-View-Controller* (MVC) architecture, so we must acquire some intuition about it. To do so, we consider an automated bank teller machine (ATM), which is a machine built in MVC style.

When you use an ATM, you stand in front of a video screen and buttons. This is the *view* of the machine. When you insert your bank card and press buttons, you activate the ATM's *controller*, which receives your bank card and button presses and relays them into the bank, where your account is kept. The accounts inside the bank are the *model*, where information about your bank account is held. The model computes the transaction you requested and relays it back to the view—money appears.

In summary, the ATM has these components in its architecture:

- A *view object*—it presents the appearance of the ATM to its human user, and it possesses methods that receive requests ("messages") from the user and display results.

- A *controller object*—it has methods that control the transfer of information within the ATM by sending messages to the other components.

- A *model object*—its methods compute answers in response to messages sent to it by the controller.

Many appliances are built with MVC architectures (e.g., pocket calculators, video games, radios), and the MVC architecture adapts well to computer programs as well—have you ever played a computer game, where the game's view was projected on the console screen, the mouse movements activated the game's controller, and the computer computed the mouse movements and showed them on the screen? The computer game is a program with MVC architecture.

1.5.1 Class Diagrams

Real programs consist of thousands of instructions and are too huge to be written as one long file. As noted in the previous section, a program must be divided into manageably-sized files called *classes*. Each class's instructions are themselves grouped into meaningful subcollections, called *methods*—we use the term, "method," as in the section, "Programs Are Objects": A method is a specific computing activity that can be activated when a message is sent to it.

Begin footnote: More precisely, when the classes are copied into primary storage, they become *objects*, and the objects send messages that request the other objects to execute the methods. *End footnote*

The classes, methods and the manner in which they communicate are drawn as a *class diagram.*

A class diagram specifies an architecture from which a program can be constructed. For example, if we design a program with an MVC architecture, the class diagram displays a class that serves as the view, a class that serves as the controller, and a class that serves as the model.

Figure 4 displays a class diagram for a simple word-processing program in MVC architecture. The components—classes—are annotated with the methods that each possesses; for example, class MODEL owns an `insertText` method, so `insertText`-messages can be sent to the MODEL. When MODEL receives an `insertText`-message, the instructions within the `insertText` method execute.

Arrows show the direction in which messages are sent. (Connections without arrow heads suggest that messages might be sent in both directions; compare this to Figure 2.)

Here is an informal explanation of the behavior of the architecture: When a human user interacts with the executing word-processing program, her inputs are transferred from the mouse and keyboard to the INPUT VIEW—perhaps mouse clicks generate `getRequest` messages to the INPUT VIEW, and text typed into the edited document generate `getText` messages.

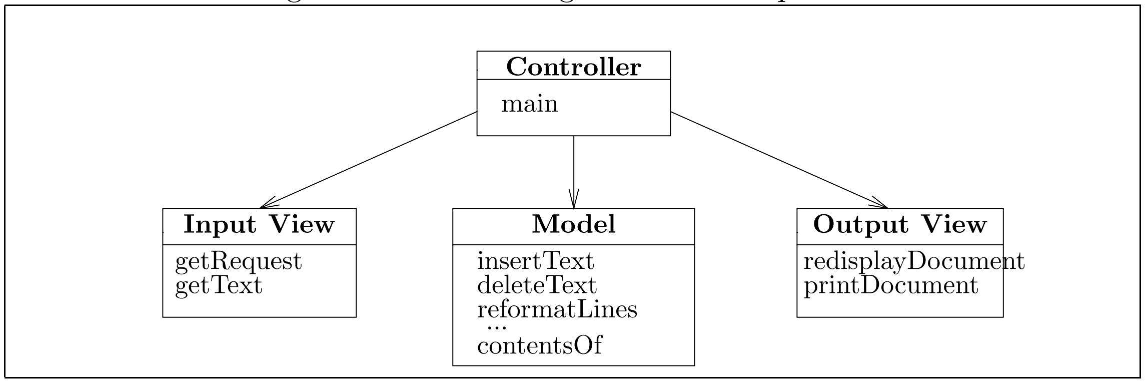
As in the ATM example, messages received by the INPUT VIEW are forwarded to the CONTROLLER component, which sends an appropriate message to the MODEL component. The MODEL is responsible for "modelling" the document the user is editing, so the MODEL has methods for editing and altering the document. (For example, a `getText` message to the INPUT VIEW causes the CONTROLLER to be contacted, which sends an `insertText` message to the MODEL. In this way, the representation of the document within primary storage gets updated.)

The CONTROLLER also sends messages to the OUTPUT VIEW, telling it to redisplay its presentation of the edited document on the display or to print the document when requested. (To display or print the document, the OUTPUT VIEW must send the MODEL a `contentsOf` message, asking for the current contents of the edited document.)

Of course, this explanation of the word processor's software architecture is informal, and additional design work is needed before Java instructions can be written for the methods and classes.

The programming examples we encounter in the next chapter have simple one- and two-class architectures, like one-room or two-room houses, but programs in subsequent chapters rise or fall on their complex designs, and we must rely on architectures

Figure 1.4: a class diagram of a word processor



and class diagrams to guide our way.

1.6 Summary

Each chapter will conclude with a summary of the terms, concepts, and programming examples that will be needed for later work. Here is the relevant material for Chapter 1:

New Terminology

- *computer*: an entity that executes instructions
- *processor*: the part of an electronic computer that executes instructions
- *primary storage*: the part of an electronic computer that holds the instructions and information that the processor reads to do its work. Also known as *memory*, *random access memory*, and *RAM*.
- *secondary storage*: the parts of an electronic computer that archives additional instructions and information. Examples are the internal "hard" disk, diskettes, and compact disks. For the processor to read the information on secondary storage, it must be copied into primary storage.
- *input device*: a mouse or keyboard, which supplies information to a computer
- *output device*: a display screen or printer, which presents information to a computer's user
- *file*: a collection of related information, typically saved on secondary storage
- *folder*: a collection of files, typically saved on secondary storage

- *bit*: the "atomic" unit of information within a computer—a "1" or a "0"
- *binary code*: a sequence of bits, read by the processor as instructions or information.
- *machine language*: the specific format of binary code read by a specific processor
- *algorithm*: a sequence of declarative instructions ("orders") for accomplishing a task.
- *computer program*: an algorithm written specifically for a processor to execute.
- *code*: traditional name for a computer program
- *programming language*: a language designed specifically for writing computer programs
- *compiler*: a computer program, that when executed, translates computer programs in one programming language into programs in another language
- *interpreter*: a computer program, that when executed, helps a processor read and execute computer programs that are not written in machine language
- *Java byte code*: the compiler for the Java programming language translates programs written in Java into programs in Java byte code
- *Java virtual machine (JVM)*: the interpreter for Java byte code
- *object*: a basic unit of an executing computer program
- *method*: an ability that an object possesses
- *message*: a communication that one object sends to another
- *client* object: an object that sends a message to a receiver
- *receiver* object: an object that receives a message sent by a client
- *input*: information given to a computer program for computation
- *output*: the answers computed by a program
- *hardware*: the physical components of a computer, e.g., processor and primary storage
- *software*: computer programs
- *operating system*: the controller program that starts when a computer is first switched on

- *command window*: a position on the computer display where a human can type instructions to a computer
- *software architecture*: the overall design of a computer program; analogous to a house's blueprint
- *class diagram*: a graphical presentation of a program's architecture
- *class*: a file containing a component of a computer program
- *Model-View-Controller (MVC) architecture*: a standard software architecture

Points to Remember

- Computer hardware and software are constructed from communicating objects. Examples of objects are the keyboard, the display, the windows on the display, files, and the executing program itself.
- Objects have methods, which are abilities that can be performed on request. Objects communicate requests to other objects to perform their methods by sending messages to them.
- A computer program is saved in a file in secondary storage. When the program is started, it is copied into primary storage, and the processor executes the instructions. We say that the program is executing.
- Programs written in the Java language are saved in files called classes; each class is a collection of methods. When a Java program is started, the program's classes are copied into primary storage; when a class is copied into primary storage, it becomes an executing object. In this way, an executing program is a collection of objects that send messages to one another.

1.7 Beyond the Basics

In this section and in the similarly named sections in subsequent chapters, we present material that will enhance your programming skills. This material is optional and can be skipped on first reading.

1.7.1 Stepwise Refinement

In the cooking world, Figure 1 is a "method" for baking lemon cake, and an Italian cookbook of is a collection, or "class," of such methods.

How do we write a method from scratch? A classic methodology for method writing is *stepwise refinement* (also known as *top-down design*), which is the process of writing an outline of a method and then refining the outline with more and more precise details until the completed method is the final result.

For example, perhaps we must write a method for making lasagna. We begin with an outline of the basic steps:

1. prepare the sauce

2. cook the pasta

3. place the pasta and sauce in a dish; cover with topping

4. bake the filled dish

Although many details are missing, we have a solid, "top level" design—an algorithm—that we can refine.

Next, we insert details. Consider Step 3; its *refinement* might read as follows:

- 3.1 Take a medium-sized dish.

- 3.2 Cover the bottom of the dish with sauce.

- 3.3 Next, place a layer of pasta noodles on top of the sauce.

- 3.4 Repeat steps 3.2 and 3.3 until the dish is filled.

- 3.5 If the cook desires, then sprinkle grated cheese on the top.

The refinement introduces specifics about filling the dish with pasta and sauce. In particular, notice the use of the words "Repeat ... until" in Step 3.4; this is a clever and standard way of saying that the step of layering pasta noodles can be repeated until a specific stopping point is reached. Similarly, Step 3.5 uses the words "if ... then" to indicate a statement that may or may not be performed, based upon the situation at the time the recipe is executed.

The other steps in the recipe are similarly refined, until nothing is left to chance.

If we were writing a method in the Java language, we would begin with a top-level design, like the one just seen, and apply stepwise refinement until all instructions were spelled out as statements in the Java language. We apply this technique to many examples in the chapters than follow.

Exercises

Use stepwise refinement to write algorithms for the following:

1. how to journey from your home to the airport
2. how to make your favorite pizza
3. how to change a flat tire on an automobile
4. how to add two fractions
5. how to perform the long division of one integer by another
6. how to calculate the sales tax (value-added tax) for a purchase in your community

1.7.2 Object-Oriented Design

A good programmer will rely on standard architectural patterns, like the MVC architecture seen in this chapter, as much as possible. But occasionally a part of an architecture (e.g., the MODEL portion of Figure 4) or even a complete new architecture must be designed specially. We encounter such situations in Chapter 7 onwards.

Object-oriented design can be used to design an architecture. Object-oriented design solves a problem by treating it as a simulation—a re-creation of reality—inside the computer. For example, games are simulations: When a child pretends to be a doctor or a fireman, she is creating a simulation. War games like chess and "battleship" are also simulations, and it is no accident that such games are available as computer programs.

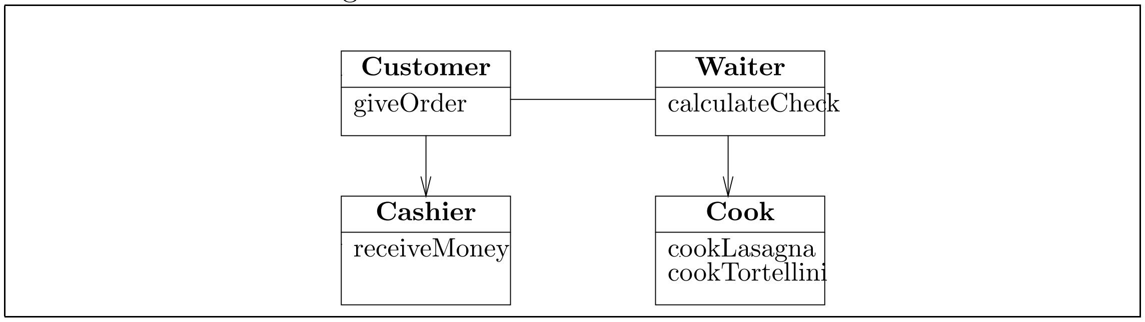
Indeed, computers are excellent devices for conducting simulations. Executing programs can create representations of people, televisions, airplanes, and even the weather. Computer simulations have proven valuable for studying traffic circulation in cities, predicting climate changes, and simulating the behavior of airplanes in flight. Some people argue that all computing activities are simulations, because a computer itself is a "simulation" of how a human being behaves when she calculates.

What is the relationship of simulation to program design? A simulation has actors or "objects" (e.g., people or pawns or battleships) who play roles. Each actor has abilities or "methods," and the actors communicate and interact to enact the simulation.

To gain some intuition about simulations, here an example. Say that you are hired as a consultant to help some entrepreneurs start a new Italian restaurant, and you must design an operating plan for the the restaurant and its personnel.

To solve this nontrivial problem, you simulate the restaurant and its personnel. The simulation is formed by answering several basic questions:

Figure 1.5: architecture of a restaurant



- *Components:* What are the forms of object one needs to solve the problem (in this case, to operate the restaurant)?
- *Collaborators:* What will be the patterns of communication between the objects? (Who talks with whom in the restaurant?)
- *Responsibilities:* What methods must each object have so that appropriate actions can be taken when there is communication? (What will the staff do when asked?)

The answers you give to these questions should lead to an appropriate operations plan—an architecture—for the Italian restaurant.

For the restaurant, one imagines the need for cooks, waiters, cashiers, and presumably, customers. (We will not worry about tables, chairs, pots, and pans, etc., for the moment.) A diagram of the components, responsibilities, and collaborations appears in Figure 5.

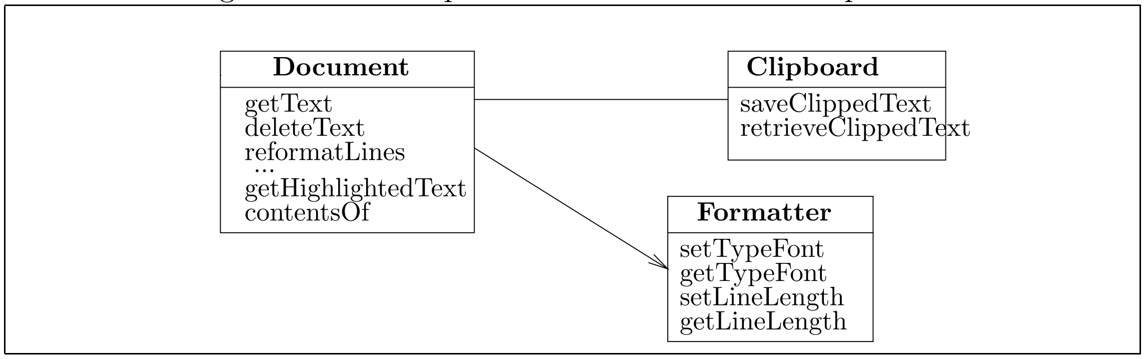
Based on the diagram, we speculate that a WAITER initiates activity by sending a `giveOrder` message to the CUSTOMER (that is, the WAITER asks the CUSTOMER what she wants to eat). The CUSTOMER's reply causes the the WAITER to send a message to the COOK to cook the appropriate dish, e.g., `cookLasagna`. In response, the COOK cooks the order, which is returned to the WAITER, who gives the food to the CUSTOMER.

Begin Footnote: An arrow, such as `WAITER --> COOK`, states that the WAITER can send a message to the COOK. The COOK is allowed to "reply" to the message by answering with food. The point is, the COOK does not initiate messages to the WAITER; it merely replies to the messages the WAITER sends it. *End Footnote*

When the CUSTOMER is finished, she sends a `calculateCheck` message to the WAITER to receive the check, and a `receiveMoney` message is sent to the CASHIER to pay the bill.

Computer programs can be designed this way as well, and it is easy to imagine how a computer game might be designed—consider an "Italian-restaurant computer game"—but even mundane information processing programs can be designed with

Figure 1.6: model part of a more realistic word processor



object-oriented design. For example, the MODEL portion of the word-processor architecture in Figure 4 is simplistic and can be replaced by one developed with object-oriented design; see Figure 6.

The Figure presents the model as consisting of a DOCUMENT component, which has the responsibility of holding the typed text and managing insertions and deletions. To help it with its responsibilities, the DOCUMENT sends messages to a FORMATTER, which manages font and line-length information. A third component, the CLIPBOARD, handles cut-and-paste actions. It takes only a little imagination to see that the DOCUMENT, FORMATTER, and CLIPBOARD are behaving much like the CUSTOMER, WAITER, and COOK seen earlier—actors communicating and interacting to accomplish a task.

In this manner, we can employ object-oriented design to creatively organize a program into distinct components with distinct responsibilities.

Exercises

Use object-oriented design to model the following:

1. a hamburger stand
2. a grocery store
3. an airport
4. a computer program that calculates spreadsheets

1.7.3 Classes Generate Objects

The class diagram in Figure 5 presented an architecture for an Italian restaurant. If we constructed a real restaurant from the diagram, however, we would certainly employ more than one waiter and one cook and we would certainly hope for more than one customer. From the designer's perspective, multiple waiter "objects" can be "constructed" from just the one *class* of waiter. If we review Figure 5, we note that the four classes (WAITER, COOK, CASHIER, and CUSTOMER) can be used to create a real restaurant with many customer objects, waiter objects, cook objects, and cashier objects. But regardless of the number of objects built, the responsibilities and collaboration patterns remain as indicated in the class diagram.

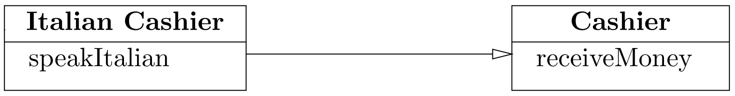
In addition, once we define a class, we can reuse it. For example, if yesterday we designed a vegetable market, and the vegetable market used cashiers, then the cashier class that we designed for the vegetable market can be reused to "build" cashiers for the Italian restaurant project.

We use the above analogy to emphasize that *classes generate objects*: When we design a Java program, we draw a class diagram that indicates the classes and their collaborations. Next, for each class in the diagram, we write a file of Java instructions that code the methods of the class. When the program is started, one or more objects are created from each of the classes, and the objects are placed in primary storage. The processor executes the instructions within the objects.

1.7.4 Frameworks and Inheritance

If we find ourselves designing lots of markets and restaurants, we might develop a set of classes and a basic architecture that can be used, with minor variations, over and over again. Such a collection is called a *framework*.

Frameworks often depend on *inheritance* to do their job. Inheritance lets us add minor customizations to a class. For example, our design of a basic CASHIER might need some customization to fit perfectly into the Italian restaurant, say, perhaps the cashier must speak Italian. We customize the CASHIER class by writing the new method, `speakItalian`, and inventing new name, say, ITALIAN CASHIER, for the customized class that possesses all the methods of the CASHIER plus the new method. The situation we have in mind is drawn like this in class-diagram style:



The ITALIAN CASHIER *inherits* (or "extends") the CASHIER class, because it has all the methods of the original plus the additional customizations. The large arrowhead in the diagram denotes this inheritance.

The attractive feature of inheritance is that the customizations are not inserted into the original class but are attached as extensions. Thus, several people can use the one and only original class but extend it in distinct ways. (For example, the basic CASHIER class might be extended into an Italian restaurant cashier and extended also into a bank cashier. For that matter, a restaurant might be built to have one CASHIER object (who does not speak Italian) and one ITALIAN CASHIER object (who does)).

We employ inheritance in Chapter 4 to generate graphics windows.

Another standard way of explaining inheritance is in terms of the *"is-a" relationship*, e.g., "An Italian-restaurant cashier is a cashier who can speak Italian." Such examples of inheritance abound in zoology, and here is one informal example:

- A mammal is an animal that is warm-blooded.
- A feline is a mammal that is cat-like.
- A tiger is a feline that has stripes.
- A lion is a feline that has a mane.
- A giraffe is a mammal that has an extremely long neck.

If we must write a simulation of a jungle, it makes sense to design one basic `animal` class, design a `mammal` class that inherits it, design a `feline` class that inherits it, etc. This gives us a coherent and economical design of the jungle animals. Also, if we wish to add birds, zebras, and other new animals later, we can easily integrate the new classes into the design.