Programming with Grace

Draft of February 3, 2014

Programming with Grace

Kim B. Bruce
Pomona College

Chapter 1

What is Programming Anyway?

Most of the machines that have been developed to improve our lives serve a single purpose. Just try to drive to the store in your washing machine or vacuum the living room with your car and this becomes quite clear. Your computer, by contrast, serves many functions. In the office or library, you may find it invaluable as a word processor. Once you get home, slip in a DVD or connect to a media site on the internet and your computer takes on the role of a television. Start up a flight simulator and it assumes the properties of anything from a hang glider to the Concorde. Launch an mp3 player and you suddenly have a music system. This, of course, is just a short sample of the functions a typical personal computer can perform. Clearly, the computer is a very flexible device.

While the computer's ability to switch from one role to another is itself amazing, it is even more startling that these transformations occur without making major physical changes to the machine. Every computer system includes both hardware, the physical circuitry of which the machine is constructed, and software, the programs that determine how the machine will behave. Everything described above can be accomplished by changing the software used without changing the machine's actual circuitry in any way. In fact, the changes that occur when you switch to a new program are often greater than those you achieve by changing a computer's hardware. If you install more memory or a faster network card, the computer will still do pretty much the same things it did before but a bit faster (hopefully!). On the other hand, by downloading a new application program through your web browser, you can make it possible for your computer to perform completely new functions.

Software clearly plays a central role in the amazing success of computer technology. Very few computer users, however, have a clear understanding of what software really is. This book provides an introduction to the design and construction of computer software in the programming language named Grace. By learning to program in Grace, you will acquire a useful skill that will enable you to construct software of your own and to learn other languages that are used in industry. More importantly, you will gain a clear understanding of what a program really is and how it is possible to radically change the behavior of a computer by constructing a new program.

A program is a set of instructions that a computer follows. We can therefore learn a good bit about computer programs by examining the ways in which instructions written for humans resemble and differ from computer programs. In this chapter we will consider several examples of instructions for humans in order to provide you with a rudimentary understanding of the nature of a computer program. We will then build on this understanding by presenting a very simple but complete example of a computer program written in Grace. Like instructions for humans, the instructions that make up a computer program must be communicated to the computer in a language that it comprehends. Grace is such a language. We will discuss the mechanics of actually communicating the text of a Grace program to a computer so that it can follow the instructions contained in the program. Finally, you have undoubtedly already discovered that programs don't always do what you expect them do to. When someone else's program misbehaves, you can complain. When this happens with a program you wrote yourself, you will have to figure out how to change the instructions to eliminate the problem. To prepare you for this task, we will conclude this chapter by identifying several categories of errors that can be made when writing a program.

1.1 Without Understanding

You have certainly had the experience of following instructions of one sort or another. Electronic devices from computers to cameras come with thick manuals of instructions. Forms, whether they be tax forms or the answer sheet for an SAT exam, come with instructions explaining how they should be completed. You can undoubtedly easily think of many other examples of instructions you have had to follow.

If you have had to follow instructions, it is likely that you have also complained about the quality of the instructions. The most common complaint is probably that the instructions take too long to read. This, however, may have more to do with our impatience than the quality of the instructions. A more serious complaint is that instructions are often unclear and hard to understand.

It seems obvious that a set of instructions is more likely to be followed correctly if they are easy to understand. This "obvious" fact, however, does not generalize to the types of instructions that make up computer programs. A computer is just a machine. Understanding is something humans do, but not something machines do. How can a computer understand the instructions in a computer program? The simple answer is that it cannot. As a result, the instructions that make up a computer program have to satisfy a very challenging requirement. It must be possible to follow them correctly without actually understanding them.

This may seem like a preposterous idea. How can you follow instructions if you don't understand them? Fortunately, there are a few examples of instructions for humans that are deliberately designed so that they can be followed without understanding. Examining such instructions will give you a bit of insight into how a computer must follow the instructions in a computer program.

First, consider the "mathematical puzzle" described below. To appreciate this example, don't just read the instructions. Follow them as you read them.

1. Pick a number between 1 and 40.
2. Subtract 20 from the number you picked.
3. Multiply by 3.
4. Square the result.
5. Add up the individual digits of the result.
6. If the sum of the digits is even, divide by 2.
7. If the result is less than 5 add 5, otherwise subtract 4.

8. Multiply by 2.
9. Subtract 6.
10. Find the letter whose position in the alphabet is equal to the number you have obtained (a=1, b=2, c=3, etc.)
11. Think of a country whose name begins with this letter.
12. Think of a large mammal whose name begins with the second letter of the country's name.

You have probably seen puzzles like this before. The whole point of such puzzles is that you are supposed to be surprised that it is possible to predict the final result produced even though you are allowed to make random choices at some points in the process. In particular, this puzzle is designed to leave you thinking about elephants. Were you thinking about an elephant when you finished? Are you surprised we could predict this?

The underlying reason for such surprise is that the instructions are designed to be followed without being understood. The person following the instructions thinks that the choices they get to make in the process (choosing a number or choosing any country whose name begins with "D"), could lead to many different results. A person who understands the instructions realizes this is an illusion.

To understand why almost everyone who follows the instructions above will end up thinking about elephants, you have to identify a number of properties of the operations performed. The steps that tell you to multiply by 3 and square the result ensure that after these steps the number you are working with will be a multiple of nine. When you add up the digits of any number that is a multiple of nine, the sum will also be a multiple of nine. Furthermore, the fact that your initial number was relatively small (less than 40), implies that the multiple of nine you end up with is also relatively small. In fact, the only possible values you can get when you sum the digits are 0, 9 and 18. The next three steps are designed to turn any of these three values into a 4 leading you to the letter "D". The last step is the only point in these instructions where something could go wrong. The person following them actually has a choice at this point. There are four countries on Earth whose names begin with "D": Denmark, Djibouti, Dominica and the Dominican Republic. Luckily, for most readers of this text, Denmark is more likely to come to mind than any of the other three countries (even though the Dominican Republic is actually larger in both land mass and population).

This example should make it clear that it is possible to *follow* instructions without understanding how they work. It is equally clear that it is not possible to *write* instructions like those above without understanding how they work. This contrast provides an important insight into the relationship between a computer, a computer program and the author of the program. A computer follows the instructions in a program the way you followed the instructions above. It can comprehend and complete each step individually but has no understanding of the overall purpose of the program, the relationships between the steps, or the ways in which these relationships ensure that the program will accomplish its overall purpose. The author of a program, on the other hand, must understand its overall purpose and ensure that the steps specified will accomplish this purpose.

Instructions like this are important enough to deserve a name. We call a set of instructions designed to accomplish some specific purpose even when followed by a human or computer that has no understanding of their purpose an *algorithm*.

There are situations where specifying an algorithm that accomplishes some purpose can actually be useful rather than merely amusing. To illustrate this, consider the standard procedure called long division. A sample of the application of the long division procedure to compute the quotient 13042144/32 is shown below:

$$
\begin{array}{rr}
 & 407567 \\
32 & \overline{)13042144} \\
 & \underline{128} \\
 & 242 \\
 & \underline{224} \\
 & 181 \\
 & \underline{160} \\
 & 214 \\
 & \underline{192} \\
 & 224 \\
 & \underline{224} \\
 & 0
\end{array}
$$

Although you may be rusty at it by now, you were taught the algorithm for long division sometime in elementary school. The person teaching you might have tried to help you understand why the procedure works, but ultimately you were probably simply taught to perform the process by rote. After doing enough practice problems, most people reach a point where they can perform long division but can't even precisely describe the rules they are following, let alone explain why they work. Again, this process was designed so that a human can perform the steps without understanding exactly why they work. Here, the motivation is not to surprise anyone. The value of the division algorithm is that it enables people to perform division without having to devote their mental energies to thinking about why the process works.

Finally, to demonstrate that algorithms don't always have to involve arithmetic, let's consider another example where the motivation for designing the instructions is to provide a pleasant surprise. Well before you learned the long division algorithm, you were probably occasionally entertained by the process of completing a connect-the-dots drawing like the one shown in Figure 1.1. Go ahead! It's your book. Connect the dots and complete the picture.

A connect-the-dots drawing is basically a set of instructions that enable you to draw a picture without understanding what it is you are actually drawing. Just as it wasn't clear that the arithmetic you were told to perform in our first example would lead you to think of elephants, it is not obvious looking at Figure 1.1 that you are looking at instructions for drawing an elephant. Nevertheless, by following the instructions "Connect the dots" you will do just that (even if you never saw an elephant before).

This example illustrates a truth of which all potential programmers should be aware. It is harder to devise an algorithm to accomplish a given goal than it is to simply accomplish the goal. The goal of the connect-the-dots puzzle shown in Figure 1.1 is to draw an elephant. In order to construct this puzzle, you first have to learn to draw an elephant without the help of the dots. Only after you have figured out how to draw an elephant in the first place will you be able to figure out where to place the dots and how to number them. Worse yet, figuring out how to place and number the dots so the desired picture can be drawn without ever having to lift your pencil from the paper can be tricky. If all you really wanted in the first place was a picture of an elephant, it would be

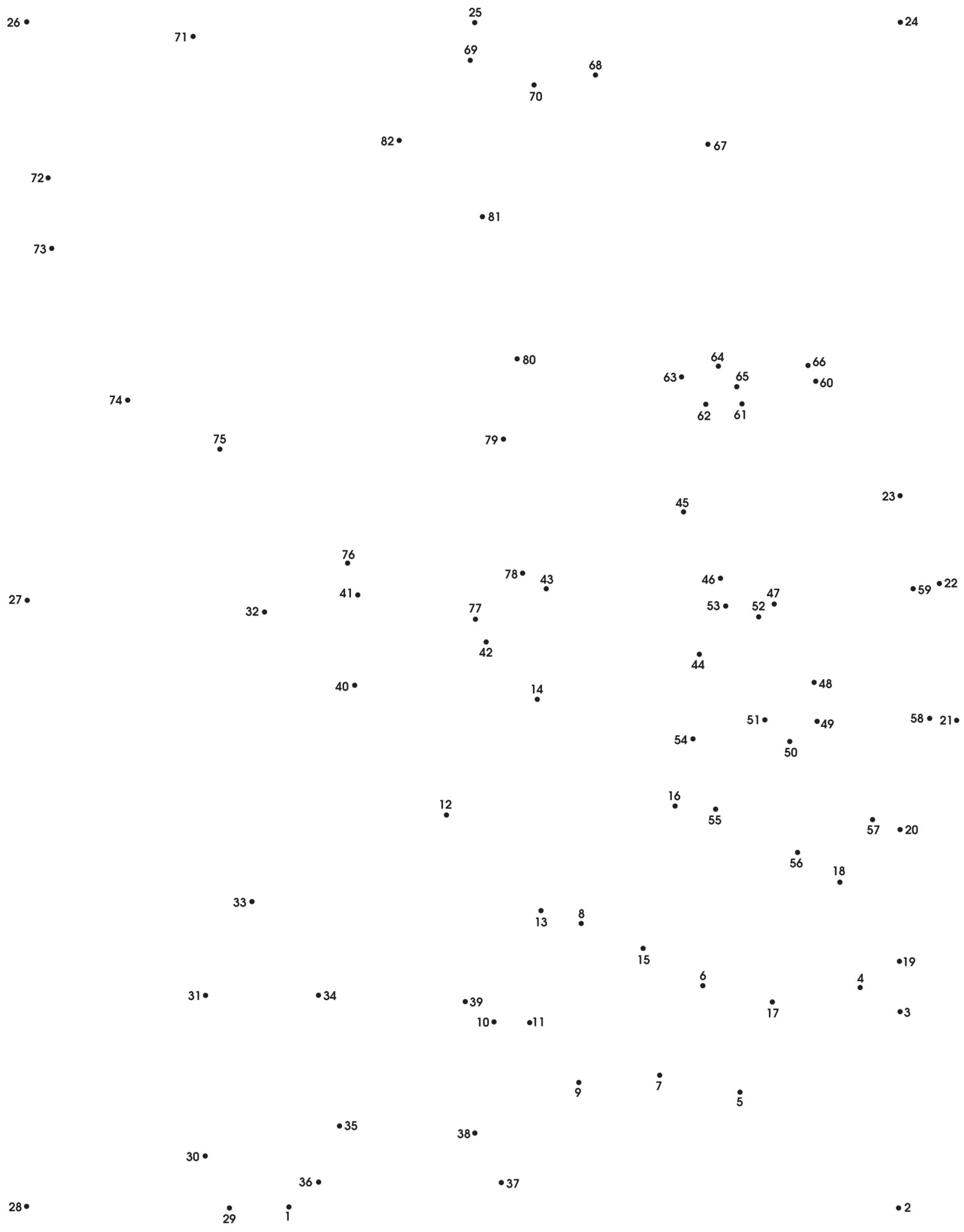


Figure 1.1: Connect dots 1 through 82 (©2000 Monkeying Around)

easier to draw one yourself. Similarly, if you have a division problem to solve (and you don't have a calculator handy) it is easier to do the division yourself than to try to teach the long division algorithm to someone who doesn't already know it so that they can solve the problem for you.

As you learn to program, you will see this pattern repeated frequently. Learning to convert your own knowledge of how to perform a task into a set of instructions so precise that they can be followed by a computer can be quite challenging. Developing this ability, however, is the key to learning how to program. Fortunately, you will find that as you acquire the ability to turn an informal understanding of how to solve a class of problems into a precise algorithm, you will be developing mental skills you will find valuable in many other areas.

1.2 The Grace Programming Language

An algorithm starts as an idea in one person's mind. To become effective, it must be communicated to other people or to a computer. Communicating an algorithm requires the use of a language. A program is just an algorithm expressed in a language that a computer can comprehend.

The choice of the language in which an algorithm is expressed is important. The numeric calculation puzzle that led you to think of Danish elephants was expressed in English. If our instructions had been written in Danish, most readers of this text would not understand them.

The use of language in a connect-the-dots puzzle may not be quite as obvious. Note, however, that we could easily replace the numbers shown with numbers expressed using the roman numerals I, II, III, IV, ... LXXXII. Most of you probably understand Roman numerals, so you would still be able to complete the puzzle. You would probably have difficulty, however, if we switched to something more ancient like the numeric system used by the Babylonians or to something more modern like the binary system that most computers use internally, in which the first few dots would be labeled 1, 10, 11, 100, 101, and 110.

The use of language in the connect-the-dots example is interesting from our point of view because the language used is quite simple. Human languages like English and Japanese are very complex. It would be very difficult to build a computer that could understand a complete human language. Instead, computers are designed to interpret instructions written in simpler languages designed specifically for expressing algorithms intended for computers. Computer languages are much more expressive than a system for writing numbers like the Roman numerals, but much simpler in structure than human languages.

One consequence of the relative simplicity of computer languages is that it is possible to write a program to translate instructions written in one computer language into another computer language. These programs are called *compilers*. The internal circuitry of a computer usually can only interpret commands written in a single language. The existence of compilers, however, makes it possible to write programs for a single machine using many different languages. Suppose that you have to work with a computer that can understand instructions written in language A but you want to write a program for the machine in language B. All you have to do is find (or write) a program written in language A that can translate instructions written in language B into equivalent instructions in language A. This program would be called a compiler for B. You can then write your programs in language B and use the compiler for B to translate the programs you write into language A so the computer can comprehend the instructions.

Each computer language has its own advantages and disadvantages. Some are simpler than others. This makes it easier to construct compilers that process programs written in these languages.

At the same time, a language that is simple can limit the way you can express yourself, making it more difficult to describe an algorithm. Think again about the process of constructing the elephant connect-the-dot puzzle. It is easier to draw an elephant if you let yourself use curved lines than if you restrict yourself to straight lines. To describe an elephant in the language of connect-the-dot puzzles, however, you have to find a way to use only straight lines. On the other hand, a language that is too complex can be difficult to learn and use.

In this text, we will teach you how to use a language named Grace to write programs. Grace is a new language that was designed to be used in teaching novices how to program in an object-oriented style, a style that we feel provides the best introduction to modern programming practices.

Other approaches to programming use programming languages like Java and Python. Java is a very popular object-oriented programming language, but it is designed for professional programmers. As a result it has many bells and whistles that are irrelevant for novice programmers, but that make the language fairly complex. These features are required in even relatively simple programs, making it a real challenge to introduce Java in a rational way to novice programmers. The original version of this book used Java, but the language complexities required us to focus more on the language idiosyncrasies than we would have hoped.

Python is another language that has become more popular for teaching novices. It is simpler than Java, but has its own idiosyncrasies. In particular, its support for object-oriented programming is missing some of the key features (e.g., information hiding) that is key to writing good programs.

Each of these language is also showing its age. Java was first released by Sun Microsystems in 1995, nearly 20 years before this book was published. Python is even older, with the first code released to the internet in 1991, and version 1.0 of the language released in 1994.

Twenty years later, we believe that Grace is a superior language for learning to program. Because it was designed for novices, the language designers have at each stage of development tried to keep the language as simple as possible, introducing new features only where necessary, and avoiding features that unnecessarily complicate the language.

Our approach to programming includes an emphasis on what is known as *event-driven programming*. In this approach, programs are designed to react to *events* generated by the user or system. The programs that you are used to using on computers use the event-driven approach. You do something – press the mouse on a button, select an item from a menu, etc. – and the computer reacts to the "event" generated by that action. In the early days of computing, programs were started with a collection of data all provided at once and then run to completion. Most text books still teach that approach to programming. In this text we take the more intuitive event-driven approach to programming. As a result, you will be able to create programs more like the ones you use every day.

1.3 Your First Encounter with Grace

The task of learning any new language can be broken down into at least two parts: studying the language's rules of grammar and learning its vocabulary. This is true whether the language is a foreign language, such as French or Japanese, or a computer programming language, such as Grace. In the case of a programming language, the vocabulary you must learn consists primarily of verbs that can be used to command the computer to do things like "**show** the number 47.2 on the screen" or "**move** the image of the game piece to the center of the window." The grammatical structures of a computer language enable you to form phrases that instruct the computer to perform several primitive commands in sequence or to choose among several primitive commands based on a user input.

When learning a new human language, one undertakes the tasks of learning vocabulary and grammar simultaneously. One must know at least a little vocabulary before one can understand examples of grammatical structure. On the other hand, developing an extensive vocabulary without any knowledge of the grammar used to combine words would just be silly. The same applies to learning a programming language. Accordingly, we will begin your introduction to Grace by presenting a few sample programs that illustrate fundamentals of the grammatical structure of Grace programs using only enough vocabulary to enable us to produce some interesting examples.

1.3.1 Programming Tools

Writing a program isn't enough. You also have to get the program into your computer and convince your computer to follow the instructions it contains.

A computer program like the one shown in the preceding section is just a fragment of text. You already know ways to get other forms of textual information into a computer. You use a word processor to write papers. When entering the body of an email message you use an email application like Apple Mail or Outlook, or you use a web-based mail program like gmail. Just as there are computer applications designed to allow you to enter these forms of text, there are applications designed to enable you to enter the text of a program.

Entering the text of your program is only the first step. As explained earlier, unless you write your program in the language that the machine's circuits were designed to interpret, you need to use a compiler to translate your instructions into a language the machine can comprehend. Finally, after this translation is complete you still need to somehow tell the computer to treat the file(s) created by the translation process as instructions and to follow them.

Typically, the support needed to accomplish all three of these steps is incorporated into a single application called an *integrated development environment* or IDE. It is also possible to provide separate applications to support each step. Which approach you use will likely depend on the facilities available to you and the inclination of the instructor teaching you to program. There are too many possibilities for us to attempt to cover them all in this text. To provide you with a simple way of running Grace programs, however, we will sketch how you can write and execute Grace programs using a web browser.

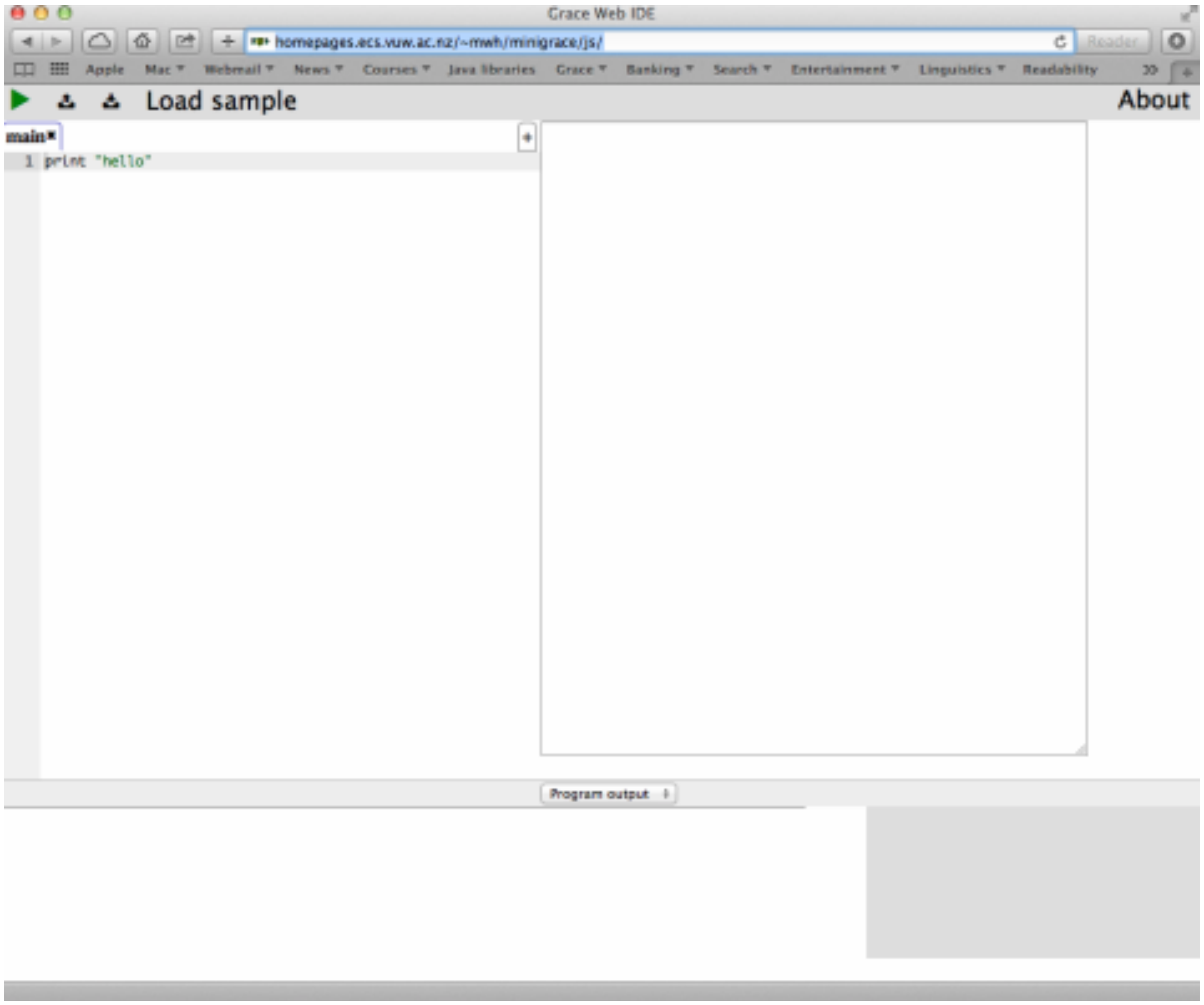
Take your favorite web browser (e.g., Safari, Chrome, Firefox, Internet Explorer, etc.) and go to the page with URL http://homepages.ecs.vuw.ac.nz/~mwh/minigrace/js/. You should get a page that looks roughly like that shown in Figure 1.2

If a page like this does not appear, you may have set your browser to block Javascript, the language that the web application is written in. If you have problems go to the preferences for your browser to see how to turn Javascript on again, if you have it turned off.

The web page comes up with a simple one-line Grace program in the editor pane in the upper left quadrant of the window.

```
print "hello"
```

When you click on the green ▷ symbol in the upper left corner of the window, the program in the editor pane will be compiled and executed. Try it! This should result in the string "hello" (without the double quotes) being printed in the pane just to the right of the editor pane.



```
print "hello"
```

Figure 1.2: A Web Page to Run Grace Programs

Congratulations, you have executed your first Grace program! Next, let's modify the program to make it more personal. Click in the edit pane to get the cursor to appear right after the "o" in "hello", but before the double quote. Add a comma character, a space and your name. Make sure they are all inside the double quotes. For example, if your name was Jilan, the program would read

```
print "hello, Jilan"
```

Now add a new line after the first in the edit pane:

```
print "This is your first Grace program!"
```

Again, click on the green ▷ symbol to execute the program. Assuming that you haven't made any typos, the program should print out:

```
hello, Jilan
This is your first Grace program!
```

Of course programs that just print out a greeting are not very interesting. In the next section we'll look at programs that can draw images in a window.

1.3.2 Methods & Definitions

A method is a named sequence of program instructions. Thus methods allow the programmer to specify names for operations that you may wish to use repeatedly. A simple example is the following greeting method

```
method greet(name) {
    print "Hello there, {name}"
}
```

When this is executed with a string replacing `name`, it will print a greeting. Thus executing `greet("Alessandro")` will print out the string "Hello there, Alessandro" in the output pane.

As you might guess from the example, surrounding an item inside a string literal by curly braces causes the material inside to be evaluated, and then inserted at that point in the string. If we had not surrounded `name` by curly braces, the method execution would have printed "Hello there, name", not at all what we wanted!

On the other hand, if we instead wrote `greet("Beau")`, the the string "Hello there, Beau" would be printed instead. The identifier `name` in the method header (the line starting with the word **method** is said to be a *parameter* of the method. It represents a value that can be supplied for each execution of the method. You can see that `name` is mentioned in the next line, where it is inserted after "Hello there," and the result printed.

Notice that neither occurrence of `name` is surrounded by quotes. That is because `name` is an identifier that represents a string. When the system is executing `greet("Alessandro")`, the value of `name` will be the string "Alessandro".

Thus methods give us a way of defining operations that can be executed with different values. You should already be familiar with similar operations in mathematics. For example, we could also define a method to square numbers:

```
method square(value) {
    value * value
}

print "The results of squaring 7 is {square(7)}."
```

Figure 1.3: Program to calculate the square of 7

```
method square(value) {
    value * value
}
```

When this method is provided with a number, it will return the square of that number (in computer science we usually use the symbol “*” to represent multiplication). Thus square(7) evaluates to 49.

We can use this method in a larger program as shown in figure 1.3. That program will print in the output

```
The result of squaring 7 is 49.
```

The curly braces in the print statement above are doing a bit more work than we have seen in the previously. Numbers and strings are different and incompatible in Grace. In general, an expression surrounded by curly braces inside a string literal is first evaluated, then converted to a string, and then inserted into the containing string. Thus, in the example above, square(7) is evaluated to the number 49, then converted to the string “49”, and then finally inserted into the string literal just before the final period.

Of course methods can be composed of more than one command

```
method betterGreeting(toName,fromName) {
   print "Hello there, {name}!"
   print "My name is {fromName}."
}
```

If we evaluate betterGreeting("Luis","Zelda"), the following will be printed:

```
Hello there, Luis!
My name is Zelda.
```

It is often useful to be able to name values in Grace. For example, the following definition associates the name `pi` with the value 3.14159.

```
def pi = 3.14159
```

Once defined in a program, we can use that name in other expressions

```
method area(radius) {
    pi*radius*radius
}
```

The ability to name things is critically important in programming. (As it is in real life. Imagine if everything needed to be specified by giving a complete description!) Any value in Grace can be associated with a name via a **def** declaration.

FIX! ►*Add something about indenting properly*◄

FIX! ►*NEED TO PUT THE FOLLOWING SOMEWHERE!!*◄

You may have noticed that sometimes we put parentheses around the material to be printed in a print statement, and sometimes we don't. You are always safe in putting in the parentheses. However, if you are printing out a single string literal (a string surrounded by double quotes) then the parentheses may be omitted, as the computer can use the double quotes to determine where the material to print starts and ends.

1.3.3 Objects

Not surprisingly, object-oriented programming is all about objects. Objects provide a way of encapsulating (putting together) data and operations (methods) on that data. Let's see how we can do this in Grace.

Grace provides an "object" expression to create objects. The object expression can contain definitions of methods as well as data associated with the object.

```
def firstPerson = object {
    def myName = "Shezad"
    def myAge = 21
    method greet(name) {
        print ("Hello there, {name}. My name is {myName}.")
    }
}
```

This object definition associates the identifier `firstPerson` with an object that contains definitions and methods representing an individual with name `"Shezad"`, who is 21 years old. The identifiers `myName` and `myAge` are specified in **def** statements within the object expression.

We can send a *method request* to an object by writing the object followed by a period, followed by the method name and any associated parameters.

```
firstPerson.greet("Jilan'')
```

Executing the above will print out

```
Hello there, Jilan.  My name is Shezad.
```

Of course, we may (and generally do) define many objects in a program. For example, we could also create objects representing a second person and a dog.

```
def secondPerson = object {
    def myName = "Charles"
    def myAge = 24
    method greet(name) {
        print ("Hello there, {name}. My name is {myName}.")
    }
}

def dog = object {
   def myName = "Rover"
   def myAge = 7
   def spayed = true
   method speak {
      print "bark"
   }
   method spin(n) {
      print "{myName} spins {n} times on command"
   }
}
```

The object associated with `secondPerson` has the same overall features (definitions and methods) as `firstPerson`, but the values associated with the identifiers `myName` and `myAge` are different. Because of that, if we evaluate

```
firstPerson . greet("Jilan'')
```

The program will print out

```
Hello there, Jilan.  My name is Charles.
```

Our `dog` also has definitions for `myName` and `myAge`, but it also has a field that indicates whether or not it has been spayed. Notice the value associated with `spayed` is not surrounded by quotes, because it is not a string. It is a built-in value, `true`. You won't be surprised to learn there is another built-in value, `false`. These two are said to be Boolean values. They are the only two Boolean values. We'll see later that these values are quite important in guiding program executions.

The `dog` also has two methods, `speak` and `spin`. The first takes no parameters, while the second takes a single parameter `n` representing the number of times `dog` should spin on command.

The point here is that some objects are similar to each other in that they have the same defined fields and methods, while others can be quite different.

1.3.4 Classes

In the last section we defined objects `firstPerson` and `secondPerson` that had exactly the same defined fields and method. They differed only by the values of the `myName` and `myAge` fields.

When we have a need for a number of objects with similar structures, we can define a method to generate those new objects. For example, look at the method `makePerson` below

```
method makePersonWithName(nameVal)age(ageVal) {
    object {
        def myName = nameVal
        def myAge = ageVal
        method greet(name) {
            print ("Hello there, {name}. My name is {myName}.")
        }
    }
}

def  firstPerson  = makePersonWithName("Shezad")age(21)
def secondPerson = makePersonWithName("Charles")age(24)
```

As you can see, we could define both `firstPerson` and `secondPerson` directly from the method.

WORRY: ►*Should I have put this method in an object to make it more comparable?*◄

Because the need to define similar objects is so common, we introduce new, somewhat simpler notation for this. Classes are compact ways of generating objects that have the same structure. The `class` declaration, as shown below, is very similar to constructs found in other object-oriented languages like Java, C#, and C++.

```
class aPerson.name(nameVal)age(ageVal) {
    def myName = nameVal
    def myAge = ageVal
    method greet(name) {
        print ("Hello there, {name}. My name is {myName}.")
   }
}

def  firstPerson  = aPerson.name("Shezad")age(21)
def secondPerson = aPerson.name("Charles")age(24)
```

The above is a bit more compact than the method `makePersonWithName()age()` as we don't have to put in the object expression, as that is implicitly there in `classes` as they are always used to create objects. In this course we will tend to favor using classes to create new objects.

FIX! ▶*Explain defs in objects are private, only methods can be invoked*◀

1.4 Generating Graphics with Objectdraw

Over the years, we (and many others!) have found that computer graphics is an excellent medium for teaching novices. First, the results are more interesting than just printing out lines of text as answers. Perhaps more importantly though, when the output of your program is an image (or perhaps even an animated image), it is easy to see when your program has errors, and the broken images can lead you to understand where you have made mistakes.

Our purpose in this course is not to make you an expert graphics programmer (though some of you may go on to become that). Instead we will use graphics as a tool to help you learn to program.

We begin in the next section to understand how coordinate systems work in computer graphics, and then quickly move on to show you how to create interactive programs using the objectdraw graphics library.

1.4.1 The Graphics Coordinate System

Programs that display graphics on a computer screen have to deal extensively with a coordinate system similar to that you have used when plotting functions in math classes. This is not evident to users of these programs. A user of a program that displays graphics can typically specify the position or size of a graphical object using the mouse to indicate screen positions without ever thinking in terms of x and y coordinates. Writing a program to draw such graphics, however, is very different from using one. When your program runs, someone else controls the mouse. Just imagine how you would describe a position on the screen to another person if you were not allowed to point with your finger. You would have to say something like "two inches from the left edge of the screen and three inches down from the top of the screen." Similarly, when writing programs you will specify positions on the screen using pairs of numbers to describe the coordinates of each position.

The coordinate system used for computer graphics is like the Cartesian coordinate systems studied in math classes but with one big difference. The y axis in the coordinate system used in computer graphics is upside down. Thus, while your experience in algebra class might lead you to expect the point (2,3) to appear below the point (2,5), on a computer screen just the opposite is true. This difference is illustrated by Figure 1.4, which shows where these two points fall in the

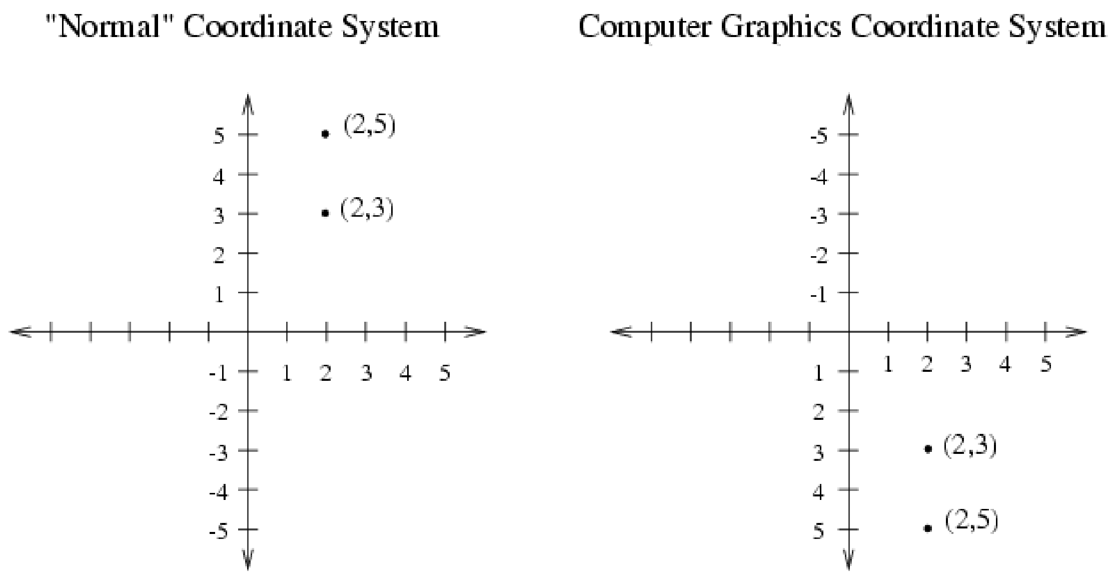


Figure 1.4: Comparison of computer and Cartesian coordinate systems

normal Cartesian coordinate system and in the coordinate system used to specify positions when drawing on a computer screen.

Our Grace programs that use graphics will draw items in a window that contains a canvas. We'll talk in more detail later about the canvas, but for now, just think of it as something like a painter's canvas. In our beginning programs the canvas will fill the entire window, though later we will see how to put other components in the window along with the canvas.

An object representing a location on the canvas can be constructed using a term of the form `aLocation.at(x,y)` where `x` and `y` are the x and y coordinates of the point. Thus the location corresponding to (2,3) can be constructed using `aLocation.at(2,3)`. Our constructors for geometric objects will use locations to specify where on the canvas they will appear.

Displaying text on a canvas is more complicated than just writing it to the program output area because we must specify where the item should be written. The objectdraw library allows a programmer to place a string on the canvas using an object constructor of the form `aText.at(`*`someLocation`*`)with(`*`someString`*`)on(`*`someCanvas`*`)`, where the italicized words are replaced with the actual information about what we want to display and where it goes.

The graphics that appear on a computer screen are actually composed of tiny squares of different colors called *pixels*. For example, if you looked at the text displayed by our first program with a magnifying glass you would discover it is actually made up of little squares as shown in Figure 1.5. The entire screen is organized as a grid of pixels. The coordinate system used to place graphics in a window is designed to match this grid of pixels in that the basic unit of measurement in the coordinate system is the size of a single pixel. So, the coordinates (30,50) describe the point that is 30 pixels to the right and 50 pixels down from the origin.

Another important aspect of the way in which coordinates are used to specify where graphics should appear is that there is not just a single set of coordinate axes used to describe locations anywhere on the computer's screen. Instead, there is a separate set of axes associated with each window on the screen and, in some cases, even several pairs of axes for different parts of a single

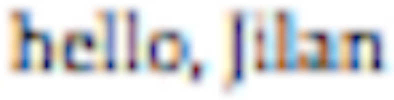


Figure 1.5: Text enlarged to make pixels visible

window.

Rather than complicating the programmer's job, the presence of so many coordinate systems makes it simpler. Many programs may be running on a computer at once and each should only produce output in certain portions of the screen. If you are running Microsoft Word at the same time as a web browser, you would not expect text from your Word document to appear in one of the browser's windows. To make this as simple as possible, each program's drawing commands must specify the window or other screen area in which the drawing should take place. Then the coordinates used in these commands are interpreted using a separate coordinate system associated with that area of the screen. The origin of each of these coordinate systems is located in the upper left corner of the area in which the drawing is taking place rather than in the corner of the machine's physical display. This makes it possible for a program to produce graphical output without being aware of the location of its window relative to the screen boundaries or the locations of other windows.

In many cases, the area in which a program can draw graphics corresponds to the entire interior of a window on the computer's display. In other cases, however, the region used by a program may be just a subsection of a window or there may be several independent drawing areas within a given window. Accordingly, we refer to a program's drawing area as a canvas rather than as a window.

1.4.2 A simple example

We will use the Grace program "TouchyWindow" given in Figure 1.6 to illustrate the basic concepts in our objectdraw graphics library. When the program is run, a window will pop up with the text "Click in this window" in the upper left. When the mouse button is depressed, the text "I'm touched" will appear in the middle of the window. When the mouse button is released, that text will disappear.

Let's walk through the code and see what each section does.

Dialect

The first line of the program specified the "dialect" of Grace that we are using for the program. Dialects are used to restrict the constructs available in the language or to add new features. The dialect `objectdrawDialect` is a dialect that provides commands for creating programs that draw and change items on the canvas. The new constructs explained in this section are all from the

```
dialect "objectdrawDialect"

// program that responds to a mouse press with a simple textual display
object {
    inherits aGraphicApplication . size (400,400)

    def clickTextLocation = aLocation.at(20,20)

    aText.at( clickTextLocation ) with ("Click in this window") on (canvas)

    // When the user presses mouse, write: I'm touched
    // on canvas at coordinates (180,200)
    method onMousePress(mousePoint){
        aText.at(aLocation.at(180,200))with ("I'm touched") on (canvas)
    }

    // When mouse is released, erase the canvas
    method onMouseRelease(mousePoint){
        canvas. clear
    }

    // create window and start graphics
    startGraphics
}
```

Figure 1.6: TouchyWindow program in Grace

objectdraw dialect of Grace. This dialect will make it much easier for us to write graphics programs in Grace.

inherits aGraphicApplication

Inheritance is used to bring in features to an object that have previously been defined in other objects. This program inherits features from `aGraphicApplication`.

When your program creates an object inheriting from `aGraphicApplication`, it will pop up a new window with a canvas pre-installed for drawing on. Thus virtually all of our programs using graphics will create one or more objects inheriting from `aGraphicApplication`.

Objects inheriting from `aGraphicApplication` will also be prepared to respond to mouse events if the programmer includes the right methods in the object. For example, this program includes the methods `onMousePress` and `onMouseRelease`. These methods are special in objects inheriting from `aGraphicApplication`, as they will be requested by the system whenever the user presses or releases the mouse.

The mouse event-handling method name must be spelled exactly as given here and there must be an identifier in parentheses after the method name. We'll talk later about the role played by that identifier.

The parameters after `size` indicate the size of the window. In the case of this example, the window will be 400 pixels wide by 400 pixels tall. If we instead wrote `aGraphicApplication.size(800,200)` then the window would be twice the width and half the height.

Coordinates in Grace

In interpreting your graphic commands, Grace will assume that the origin of the coordinate system is located at the upper left corner of the canvas in which you are drawing. The location of the coordinate axes that would be used to interpret the coordinates specified in our `TouchyWindow` example are shown in Figure 1.7. **FIX!** ▶*Image in in figure is not correct.*◀

Notice that the coordinates of the upper left hand corner of this window are (0,0). The window shown is 400 pixels wide and 400 pixels high. Thus the coordinates of the lower right corner are (400, 400). The "I'm touched"text is positioned so that it falls in a rectangle whose upper left corner has an x coordinate of 180 and a y coordinate of 200.

The computer will not consider it an error if you try to draw beyond the boundaries of your program's canvas. It will remember everything you have drawn and show you just the portion of these drawings that fall within the boundaries of your canvas.

1.4.3 Constructing Graphic Objects

As explained earlier, a text item is displayed on the canvas when the following expression is evaluated.

```
def clickTextLocation = aLocation.at(20,20)
aText.at( clickTextLocation ) with ("Click in this window") on (canvas)
```

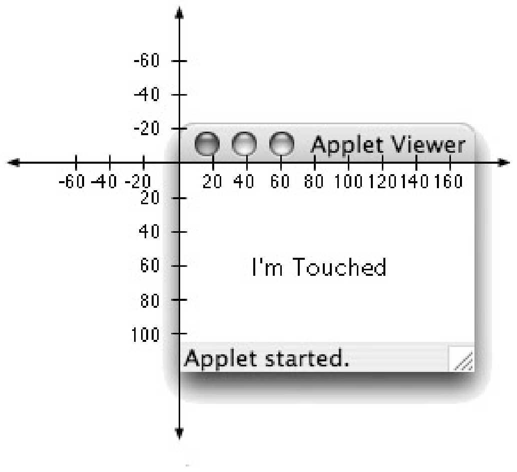


Figure 1.7: A program window and its drawing coordinate axes

In this case it will appear at `aLocation.at(20,20)`). That is, the x and y coordinates will both be 20. It will display "Click in this window" on the default canvas for graphics applications).

There is a second statement creating text in this program. It is in the method named `onMousePress`. We talk about when this command is executed in the next subsection.

1.4.4 Methods

The two methods in this program are named `onMousePress` and `onMouseRelease`. They are special methods that are associated with graphics applications. Not surprisingly, the system requests the first method when the user presses the mouse button down, and the second one when the mouse button is released. The `mousePoint` parameter in parentheses after each method name represents the location on the canvas where the mouse was pointing when it was pressed or released. It is not used in this program, but must be included anyway. We will talk about it in more detail later when it does get used.

As explained earlier, the code enclosed in curly brackets after the method name is executed when the method is requested by the system. Thus when the mouse button is pressed, a new text item stating "I'm touched" is displayed at x coordinate 180 and y coordinate 200 on the canvas. Similarly, when the mouse button is released, the canvas is cleared, as executing the command `canvas.clear` sends a request to the canvas to execute its `clear` method, which just erases everything on the canvas.

In general, the programmer is free to choose any appropriate name for a method. The method name can then be used in other parts of the program to cause the computer to obey the instructions within the methods body. Within a class that extends `aGraphicApplication`, however, certain method names have special significance. In particular, if such a class contains a method which is named `onMousePress` then the instructions in that methods body will be followed by the computer when the mouse is depressed within the programs window. That is why this particular program reacts to a mouse press as it does.

Exercise 1.4.1 *Rewrite the line of code in* `onMousePress` *of program* `TouchyWindow` *so that it now displays the message "Hello" 60 pixels to the right and 80 pixels down from the top left corner of the window.*

1.4.5 startGraphics

The final line in the object definition is `startGraphics`. When this command is executed, the window is displayed with whatever graphics have been created on the canvas to this point.

In the TouchyWindow example, when the program is run, the text with "Click in this window" will be displayed when `startGraphics` is executed. The code in the method `onMousePress` and `onMouseRelease` will not be executed until the system requests those method – which will happen when the user presses or releases the mouse button.

1.5 Creating other graphics items

Now that we have seen what a simple graphics program looks like, let's see how we can construct other graphics items on a canvas.

In each case we will need to supply the location where it will be drawn, information about the other attributes of the item (e.g., width and height) and the canvas it will be drawn on. For example, for a text item, we provided its location, the text to be displayed, and then canvas it should be drawn on.

To display a line we use the construction `aLine.from(`*`start`*`)to(`*`end`*`)on(`*`canvas`*`)` where *start* and *end* are locations giving the end points of the line, and *canvas* is the convas on which it is drawn.

Thus we can draw a line between the upper left and lower right corners of a canvas whose dimensions are 200 by 300, you would write:

```
aLine.from(aLocation.at(0,0))to(aLocation.at(200,300))on(canvas)
```

The line produced would look like the line shown in the window in Figure 1.8.

Similarly, to draw a line from the middle of the window, which has the coordinates (100,150), to the upper right corner, whose coordinates are (200,0), you would write:

```
aLine.from(aLocation.at(100, 150)to(aLine.at(200, 0) on (canvas)
```

Such a line is shown in Figure 1.9.

Using combinations of these construction statements, we could replace the single instruction in the body of the `onMousePress` method shown above with one or more other instructions. Such a modified program is shown in Figure 1.10.

The only differences between this example and `TouchyWindow` are that no text is written on the screen when this object is executed, and the commands in method `onMousePress` result in drawing two crossed lines when the button is pressed. The drawing produced is also shown in the figure.

There are several other forms of graphics you can display on the screen. The command:

```
aFramedRect.at(aLocation.at(20, 50)size(80, 40)on(canvas)
```

will display the outline, or perimeter, of an 80 by 40 rectangular box in your canvas. The pair `20, 50` specifies the coordinates of the box's upper left corner. The pair `80, 40` specifies the width and height of the box. If you replace the name `FramedRect` by `FilledRect` to produce the construction

```
aFilledRect.at(aLocation.at(20, 50)size(80, 40)on(canvas)
```

the result will instead be an 80 by 40 solid black rectangular box.

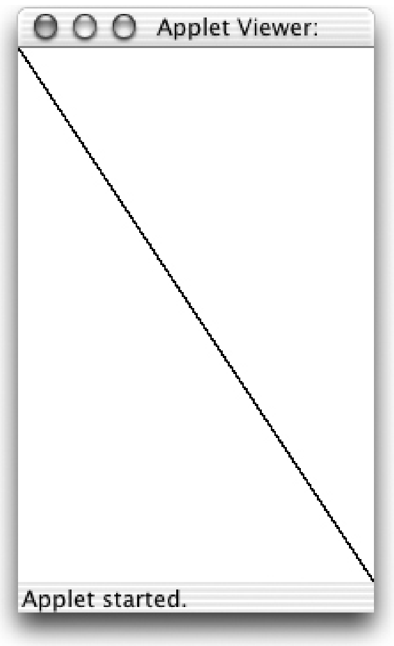


Figure 1.8: Drawing of a single line

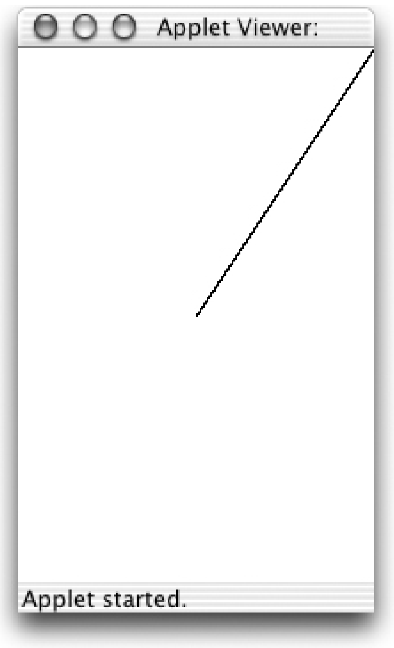
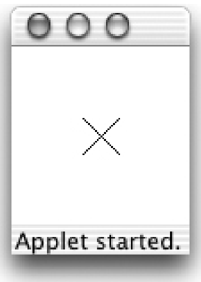


Figure 1.9: A line from (100,150) to (200,0)



```
dialect "objectdrawDialect"

// program that responds to a mouse press with a cross
object {
  inherits aGraphicApplication.size(400,400)

  // When the user presses mouse, draw a cross on canvas
  method onMousePress(mousePoint){
    aLine.from(aLocation.at(40,40))to(aLocation.at(60,60)) on (canvas)
    aLine.from(aLocation.at(60,40))to(aLocation.at(40,60)) on (canvas)
  }

  // When mouse is released, erase the canvas
  method onMouseRelease(mousePoint){
    canvas.clear
  }

  // create window and start graphics
  startGraphics
}
```

Figure 1.10: A program that draws two crossed lines.

The command:

```
aFilledOval.at(aLocation.at(20, 50)size(80, 40)on(canvas)
```

will draw a filled oval on the screen. The parameters are interpreted just like those to the `FilledRect` construction. Instead of drawing a rectangle, however, `FilledOval` draws the largest ellipse that it can fit within the rectangle described by its parameters. To illustrate this, Figure 1.11 shows what the screen would contain after executing the two constructions

```
aFramedRect.at(aLocation.at(20, 50)size(80, 40)on(canvas)
aFilledOval.at(aLocation.at(20, 50)size(80, 40)on(canvas)
```

The upper left corner of the rectangle shown is at the point with coordinates (20, 50). Both shapes are 80 pixels wide and 40 pixels high. You can create a framed oval by replacing `aFilledOval` by `aFramedOval` in the above command.

Other primitives allow you to draw additional shapes and to display image files in your canvas. A full listing and description of the available graphic object types and the forms of the commands used to construct them can be found in Appendix ?? For now, the graphical object generators `aText`, `aLine`, `aFramedRect`, `aFilledRect`, `aFramedOval`, and `aFilledOval` will provide enough

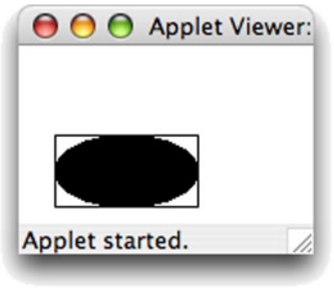


Figure 1.11: A `FilledOval` nested within a `FramedRect`

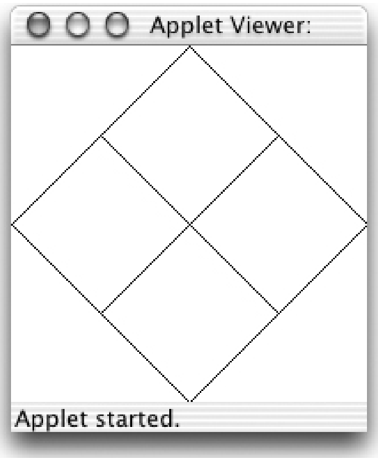
flexibility for our purposes.

Exercise 1.5.1 *Sketch the picture that would be produced if the following constructions were executed. You should assume that the canvas associated with the program containing these instructions is 200 pixels wide and 200 pixels high.*

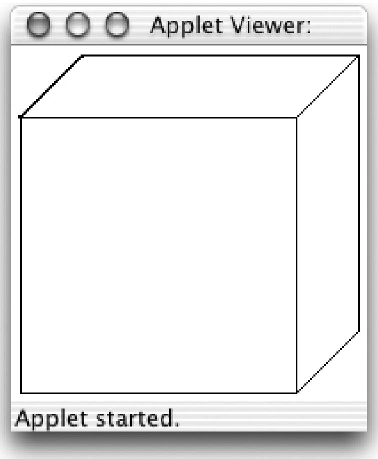
```
aLine.from(aLocation.at(0,100))to(aLocation.at(100, 0) on (canvas)
aLine.from(aLocation.at(200,100))to(aLocation.at(100,200) on (canvas)
aLine.from(aLocation.at(100,200))to(aLocation.at(0,100) on (canvas)
aFramedRect.at(aLocation.at(50,50))size(100, 100) on (canvas)
```

Exercise 1.5.2 *Write a sequence of Line and/or FramedRect constructions that would produce each of the drawings shown below. In both examples, assume that the drawing will appear in a 200 by 200 pixel window. For the drawing of the three dimensional cube, there should be a space 5 pixels wide between the cube and the edges of the window in those areas where the cube comes closest to the edges. The rectangle drawn for the front face of the cube should be 155 pixels wide and 155 pixels high. The two visible edges of the rear of the cube should also be 155 pixels long.*

a)



b)



1.6 Additional Event Handling Methods

In our examples thus far, we have used the two method names `onMousePress` and `onMouseRelease` to establish a correspondence between certain user actions and instructions we would like the computer to follow when these actions occur. In this section, we introduce several other method names that can be used to associate instructions with other user actions.

1.6.1 Mouse Event Handling Methods

In addition to `onMousePress` and `onMouseRelease`, there are five other method names that have special significance for handling mouse events. If you include definitions for any of these methods within a class that inherits `aGraphicApplication`, then the instructions within the methods you include will be executed when the associated events occur.

The definitions of all these methods have the same form. You have seen that the header for the `onMousePress` method looks like:

```
method onMousePress(point)
```

The headers for the other methods are identical except that `onMousePress` is replaced by the appropriate method name.

All of the mouse event handling methods are described below:

onMousePress specifies the actions the computer should perform when the mouse button is depressed.

onMouseRelease specifies the actions the computer should perform when the mouse button is released.

onMouseClick specifies the actions the computer should perform if the mouse is pressed and then quickly released without significant mouse movement between the two events. The actions specified in this method will be performed in addition to (and after) any instructions in `onMousePress` and `onMouseRelease`.

onMouseEnter specifies the actions the computer should perform when the mouse enters the program's canvas.

onMouseExit specifies the actions the computer should perform when the mouse leaves the program's canvas.

onMouseMove specifies the actions the computer should perform periodically while the mouse is being moved about without its button depressed.

onMouseDrag specifies the actions the computer should perform periodically while the mouse is being moved about with its button depressed.

Exercise 1.6.1 *Write the complete method header for the* `onMouseMove` *method.*

Exercise 1.6.2 *Write a method that draws a filled square on the canvas when the mouse enters the canvas. The square should be 100×100 pixels with the upper left corner at the origin.*

Exercise 1.6.3 *Write a complete program that will display "I'm inside" when the mouse is inside the program's window and "I'm outside" when the mouse is outside the window. The screen should be blank when the program first begins to execute and should stay blank until the mouse is moved in or out of the window.*

1.6.2 The Initialization code

Code in an object definition that is not included in a method body is executed when the object is evaluated. In graphics applications this code is generally responsible for creating the initial image presented on the canvas and doing any other initialization necessary before the user begins interacting with the program. Typically the last line in an object inheriting from `aGraphicApplication` will be `startGraphics`, which displays the window and readies it to respond to user actions.

1.7 To Err is Human

We all make mistakes. Worse yet, we often make the same mistakes over and over again.

If we make a mistake while giving instructions to another person, the other person can frequently figure out what we really meant. For example, if you give someone driving directions and say "turn left" somewhere you should have said "turn right" chances are that they will realize after driving in the wrong direction for a while that they are not seeing any of the landmarks the remaining instructions mention, go back to the last turn before things stopped making sense and try turning in the other direction. Our ability to deal with such incorrect instructions, however, depends on our ability to understand the intent of the instructions we follow. Computers, unfortunately, never really understand the instructions they are given so they are far less capable of dealing with errors.

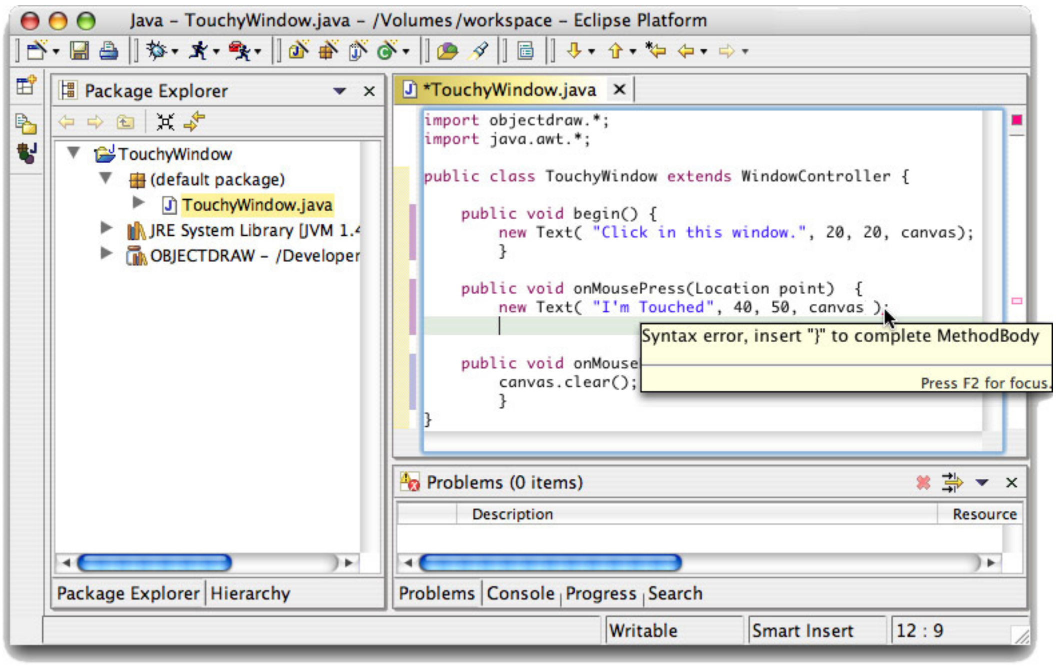
There are several distinct types of errors you can make while writing or entering a program. The computer will respond differently depending on the type of mistake you make. The first type of mistake is called a *syntax error*. The defining feature of a syntax error is that the IDE can detect that there is a problem before you try to run your program. As we have explained, a computer program must be written in a specially designed language that the computer can interpret or at least translate into a language which it can interpret. Computer languages have rules of grammar just like human languages. If you violate these rules, either because you misunderstand them or simply because you make a typing mistake, the computer will recognize that something is wrong and tell you that it needs to be corrected.

The mechanisms used to inform the programmer of syntactic errors vary from one IDE to another. The web IDE for Grace examines the text you have entered and indicates fragments of your program that it has identified as errors by displaying an error icon (a red "x") on the offending line at the left margin. If you point the mouse at the the error icon, the IDE will display a message explaining the nature of the problem. For example, If we accidentally left out the closing "}" after the body of the `onMousePress` method while entering the program shown in Figure **??**, the IDE would place a red "x" on the last line of the method. Pointing the mouse at the underlined semicolon would cause the IDE to display the message "syntax error, a method must end with '}" as shown in Figure 1.12. **FIX!** ►*Fix the figure*◄

The bad news is that your IDE will not always provide you with a message that pinpoints your mistake so clearly. When you make a mistake, your IDE is forced to try to guess what you meant to type. As we have emphasized earlier, your computer really cannot be expected to understand what your program is intended to do or say. Given its limited understanding, it will often mistake your intention and display error messages that reveal its confusion. For example, if you type **FIX!** ►*Need appropriate Grace error with obscure message to replace example below.*◄

```
canvas.clear;
```

instead of



```
import objectdraw.*;
import java.awt.*;

public class TouchyWindow extends WindowController {

    public void begin() {
        new Text( "Click in this window.", 20, 20, canvas);
        }

    public void onMousePress(Location point)  {
        new Text( "I'm Touched", 40, 50, canvas );
        |

    public void onMouse
        canvas.clear();
        }
}
```

```
Syntax error, insert "}" to complete MethodBody
Press F2 for focus.
```

Figure 1.12: Eclipse displaying a syntax error message

```
canvas.clear();
```

in the body of the `onMouseRelease` method of our example program, the IDE will *print something stupid*. In such cases, the error message is more likely to be a hindrance than a help. You will just have to examine what you typed before and after the position where the IDE identified the error and use your knowledge of the Java language to identify your mistake.

A program that is free from syntax errors is not necessarily a correct program. Think back to our instructions for performing calculations that was designed to leave you thinking about Danish elephants. If while typing these instructions we completely omitted a line, the instructions would still be grammatically correct. Following them, however, would no longer lead you to think of Danish elephants. The same is true for the example of saying "left" when you meant to say "right" while giving driving directions. Mistakes like these are not syntactic errors; they are instead called *logic errors*. They result in an algorithm that doesn't achieve the result that its author intended. Unfortunately, to realize such a mistake has been made, you often have to understand the intended purpose of the algorithm. This is exactly what computers don't understand. As a result, your IDE will give you relatively little help correcting logic errors.

As a simple example of a logical error, suppose that while typing the `onMouseRelease` method for the `TouchyWindow` program you got confused and typed `onMouseExit` instead of `onMouseRelease`. The result would still be a perfectly legitimate program. It just wouldn't be the program you meant to write. Your IDE would not identify your mistake as a syntax error. Instead, when you ran the program it just would not do what you expected. When you released the mouse, the "I'm Touched" message would not disappear as expected.

This may appear to be an unlikely error, but there is a very common error which is quite similar to it. Suppose instead of typing the name `onMouseRelease` you typed the name `onMooseRelease`.

Look carefully. These names are not the same. `onMooseRelase` is not the name of one of the special event handling methods discussed in the preceding sections. In more advanced programs, however, we will learn that it is sometimes useful to define additional methods that do things other than handle events. The programmer is free to choose names for such methods. `onMooseRelease` would be a legitimate (if strange) name for such a method. That is, a program containing a method with this name has to be treated as a syntactically valid program by any Grace IDE. As a result, your IDE would not recognize this as a typing mistake, but when you ran the program Grace would think you had decided not to associate any instructions with mouse release events. As before, the text "I'm Touched" would never be removed from the canvas.

There are many other examples of logical errors a programmer can make. Even in a simple program like `TouchyWindow`, mistyping screen coordinates can lead to surprises. If you mistyped an x coordinate as in

```
aText.at(aLocation.at(400,50))with("I'm Touched")on(canvas)
```

the text would be positioned outside the visible region of the program window. It would seem as if it never appeared. If the line

```
canvas.clear
```

had been placed in the `onMousePress` method with the line to construct the message, the message would disappear so quickly that it would never be seen.

Of course, in larger programs the possibilities for making such errors just increases. You will find that careful, patient, thoughtful examination of your code as you write it and after errors are detected is essential.

1.8 Summary

Programming a computer to say "I'm Touched." is obviously a rather modest achievement. In the process of discussing this simple program, however, we have explored many of the principles and practices that will be developed throughout this book. We have learned that computer programs are just collections of instructions. These instructions, however, are special in that they can be followed mechanically, without understanding their actual purpose. This is the notion of an algorithm, a set of instructions that can be followed to accomplish a task without understanding the task itself. We have seen that programs are produced by devising algorithms and then expressing them in a language which a computer can interpret. We have learned the rudiments of the language we will explore further throughout this text, Grace. We have also explored some of the tools used to enter computer programs, translate them into a form the machine can follow, and run them.

Despite our best efforts to explain how these tools interact, there is nothing that can take the place of actually writing, entering and running a program. We strongly urge you to do so before proceeding to read the next chapter. Throughout the process of learning to program you will discover that it is a skill that is best learned by practice. Now is a good time to start.