

Programming with Grace

Draft of May 16, 2014

Programming with Grace

Kim B. Bruce
Pomona College

DRAFT!

DO NOT DISTRIBUTE WITHOUT PRIOR PERMISSION

Website: <http://eventfuljava.cs.williams.edu>

Printed on May 16, 2014

©2014 Kim B. Bruce¹

Comments, corrections, and other feedback appreciated

kim@cs.pomona.edu

¹This book is partially based on *Java: An eventful approach* by Kim B. Bruce, Andrea Pohorecky Danyluk, and Thomas P. Murtagh, ©2006 by Pearson Education

Contents

1	What is Programming Anyway?	1
1.1	Without Understanding	2
1.2	The Grace Programming Language	6
1.3	Your First Encounter with Grace	7
1.3.1	Programming Tools	8
1.3.2	Methods & Definitions	10
1.3.3	Objects	12
1.3.4	Classes	13
1.3.5	Program Layout	14
1.4	Generating Graphics with Objectdraw	14
1.4.1	The Graphics Coordinate System	15
1.4.2	A simple example	17
1.4.3	Constructing Graphic Objects	19
1.4.4	Methods	19
1.4.5	startGraphics	20
1.5	Creating other graphics items	20
1.6	Additional Event Handling Methods	24
1.6.1	Mouse Event Handling Methods	24
1.6.2	The Initialization code	25
1.7	To Err is Human	26
1.8	Summary	28
2	What's in a name?	29
2.1	Naming and Modifying Objects	29
2.1.1	Mutator Methods	29
2.1.2	Definitions	31
2.1.3	Variable Declarations	33
2.1.4	Comments	34
2.1.5	Additional Mutator Methods	35
2.1.6	Exercises	36
2.2	Non-graphical Classes of Objects	39
2.2.1	The Class of Colors	39
2.2.2	The Location Class	42
2.3	Layering on the Canvas	43
2.4	Accessing the Location of the Mouse	46

2.5	Sharing Parameter Information between Methods	47
2.6	Summary	53
3	Working with Numbers	55
3.1	Writing numbers	55
3.2	Introduction to Accessor Methods	55
3.3	Accessing the Size of the Canvas	58
3.4	Expressions and Statements	59
3.5	Arithmetic Expressions	61
3.5.1	Ordering of Arithmetic Operations	63
3.6	Numeric Instance Variables	65
3.7	Displaying Numeric Information	67
3.7.1	Displaying Numbers as text	67
3.7.2	Using <code>print</code>	69
3.7.3	Mixing Text and Numbers	71
3.8	Random numbers	72
3.9	Summary	75
4	Making Choices	77
4.1	A Brief Example: Using the <code>if</code> Statement to Count Votes	77
4.2	The <code>if</code> Statement	80
4.2.1	Example: Using the <code>if</code> Statement with 2-D Objects	82
4.3	Understanding Conditions	83
4.3.1	Using Boolean Values	84
4.4	Selecting Among Many Alternatives	87
4.5	More on Boolean Expressions	89
4.6	Nested Conditionals	93
4.7	Summary	97
4.8	Chapter Review Problems	97
4.9	Programming Problems	100
5	Types, Numbers, and Strings	103
5.1	What is a type?	103
5.2	Types from <code>objectdraw</code>	103
5.3	Using type annotations	105
5.4	Numbers	109
5.4.1	The <code>math</code> module	110
5.5	Handy Sources of Numeric Information	110
5.5.1	Time flies	110
5.5.2	Sines and Wonders	111
5.6	Strings	114
5.7	Chapter Review Problems	117
5.8	Programming Problems	117

List of Figures

1.1	Connect dots 1 through 82 (©2000 Monkeying Around)	5
1.2	A Web Page to Run Grace Programs	9
1.3	Program to calculate the square of 7	11
1.4	Comparison of computer and Cartesian coordinate systems	15
1.5	Text enlarged to make pixels visible	16
1.6	TouchyWindow program in Grace	17
1.7	A program window and its drawing coordinate axes	18
1.8	Drawing of a single line	21
1.9	A line from (100,150) to (200,0)	21
1.10	A program that draws two crossed lines.	22
1.11	A FilledOval nested within a FramedRect	23
1.12	Eclipse displaying a syntax error message	27
2.1	An oval rises over the horizon	31
2.2	Declaring sun in the RisingSun program	32
2.3	Commented code for rising sun example	34
2.4	Rising sun program with reset feature	37
2.5	An application of the translate method	42
2.6	Display after one click	43
2.7	Display after second click	44
2.8	Display after many clicks	44
2.9	Today's forecast: Partly cloudy	45
2.10	Tonight's forecast: Partly sunny?	46
2.11	Using a different parameter name	48
2.12	A program to record mouse button changes	48
2.13	Connecting the ends of a mouse motion	49
2.14	A Program to track mouse actions	50
2.15	Scribbling with a computer's mouse	51
2.16	A simple sketching program	53
3.1	Program to make the sun scroll with the mouse	57
3.2	Drawing produced by the DrawGrid program	57
3.3	The sun rises over the horizon	61
3.4	Program to make the sun scroll with the mouse	62
3.5	Drawing desired for Exercise 3.5.1	64
3.6	Using a numeric instance variable	66

3.7	A computer counting program before the first click	68
3.8	A simple counting program.	69
3.9	Counting in the Grace console	70
3.10	Counting using the Grace console	71
3.11	Sample message drawn by dice simulation program	73
3.12	Simulating the rolling of a pair of dice	74
4.1	Screen shot of Voter program.	78
4.2	Code for Voter program.	79
4.3	Semantics of the if–else statement.	81
4.4	Code to display individual and total vote counts	81
4.5	Semantics of the if with no else.	82
4.6	Code for dragging a box.	85
4.7	Three stages of dragging a rectangle.	86
4.8	Voting for four candidates.	90
4.9	A summary of the boolean and comparison operators in Grace	92
4.10	Craps program illustrating nested conditionals.	96
4.11	Display for InvisibleBox	100
4.12	Display for Dicey	101
5.1	The type Graphic from objectdraw	106
5.2	The type Text from the objectdraw library	107
5.3	The type Line from the objectdraw library	107
5.4	The type Graphic2D from the objectdraw library	107
5.5	The Craps program with full type annotations	108
5.6	Grace program to measure mouse click duration.	111
5.7	Sample output of the ClickTimer program	112
5.8	A program to display Morse code as dots and dashes	115
5.9	Sample of Morse code program display	116
5.10	Display for DNAGenerator	118

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, connect to a media site on the internet and your computer takes on the role of a television. Start up Skype, FaceTime, or a Google Hangup and you have a videophone. 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 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 to do. 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 the individual instructions correctly without actually understanding their overall purpose.

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:

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

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

FIX! ► *Will need to modify when fix environment.* ◀

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 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 should come 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.

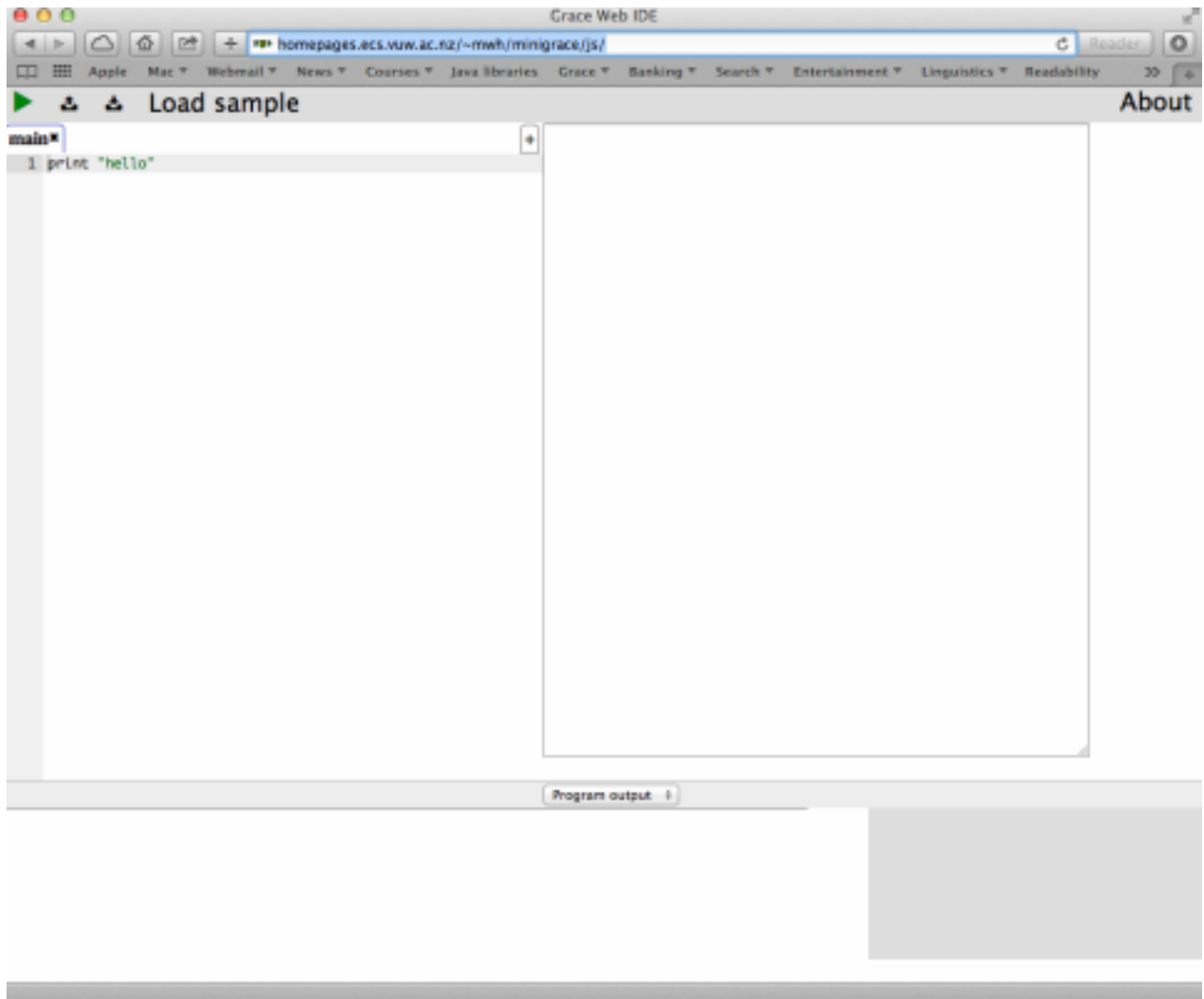


Figure 1.2: A Web Page to Run Grace Programs

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 method is requested 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".

```

method square(value) {
    value * value
}

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

```

Figure 1.3: Program to calculate the square of 7

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
}

```

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 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, {toName}!"
    print "My name is {fromName}."
}

```

When a method is requested, the code enclosed in braces is executed. Thus, if we evaluate `betterGreeting("Luis","Zelda")`, the following will be printed:

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

By the way, you will notice that when you see `betterGreeting("Luis","Zelda")` it is not obvious which parameter represents the greeter and which the greetee. We can improve things by giving a slightly better name to the method and separating the parameters:

```

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

```

Now when we write `bestGreetingTo("Luis")from("Zelda")` it is much clearer who is being greeted (“Luis”) and who is the greeter (“Zelda”). Whenever it seems helpful for understanding, we will separate the parameters with words in this way, rather than just separating them with commas.

It is often useful to be able to associate names to 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! ► *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 (grouping 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 `secondPerson.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 *class* to generate those new objects. For example, look at the class `makePerson` below

```

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)
```

As you can see, we could define both `firstPerson` and `secondPerson` directly from the method. Because of the parameters provided, `firstPerson` has name "Shezad" and age 21 (as before), while `secondPerson` has name "Charles" and age 24 (again, just as before).

It would be possible to write this instead as a method that returns an object. However, both for simplicity and because most object-oriented languages support classes like this, we will use the class construct in this text.

1.3.5 Program Layout

You may have noticed that we have been very careful in the indenting of our programs. Grace will insist that your program be laid out very consistently, as this will help you understand how pieces of a Grace program fit together.

All the items that are declared in an object should start at the same indentation, as in the examples above. Code in a method should start a few spaces in from the method header (the line starting with the word "method"), and all subsequent lines in the method should be at the same indentation. The closing curly bracket is generally put at the same indentation as the method header. Occasionally in this text we will put the closing curly bracket at the end of the last line of the method if we are pressed for space on the page, but we encourage you to always put it on a new line.

Lines that are continuations of a previous line should always be indented by a few more characters so the compiler knows it is a continuation. Otherwise it will interpret it as a new command and get confused. For example, suppose you write

```
def x = 13
  -7
```

This will result in `x` having the value of 13. While if instead you wrote

```
def x = 13
-7
```

then `x` will have the value `13-7` or 6, as it will assume that the `-7` is a continuation of the previous line. In the case where the `-7` was not indented, the system interpreted `-7` as starting a new command, even though it doesn't make much sense.

We strongly recommend that you avoid using the tab key in laying out your program. Some editors can be set so that tabs are automatically replaced by a fixed number of spaces. However, any tabs in your program text will cause an error in Grace programs.

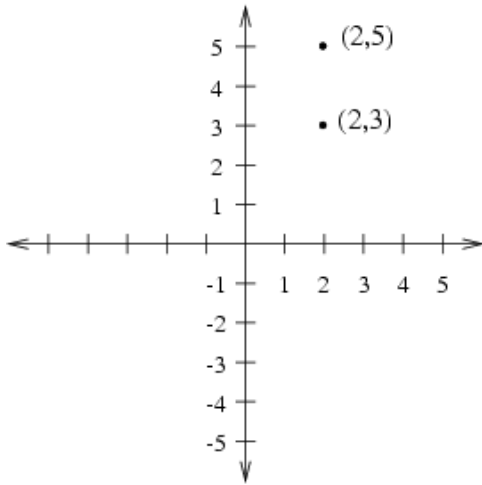
FIX! ► *Explain defs in objects are private, only methods can be invoked* ◀

1.4 Generating Graphics with Objectdraw

Over the years, we (and many others!) have found that computer graphics is an excellent medium for teaching novice programmers. 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.

"Normal" Coordinate System



Computer Graphics Coordinate System

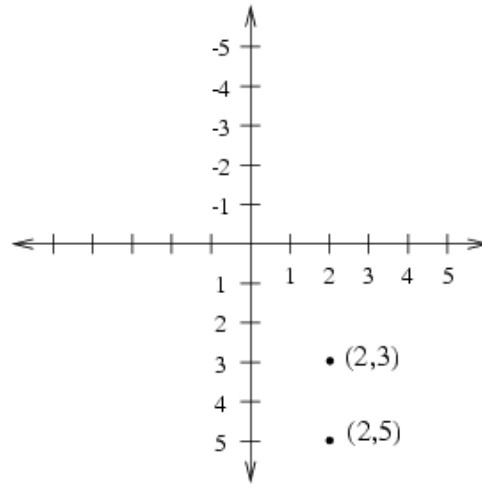


Figure 1.4: Comparison of computer and Cartesian coordinate systems

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 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

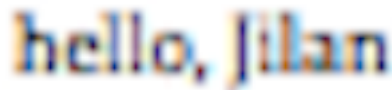


Figure 1.5: Text enlarged to make pixels visible

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 classes for constructing 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 string 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 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


```

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)
  def touchedTextLocation = aLocation.at(180,200)

  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(touchedTextLocation)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

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

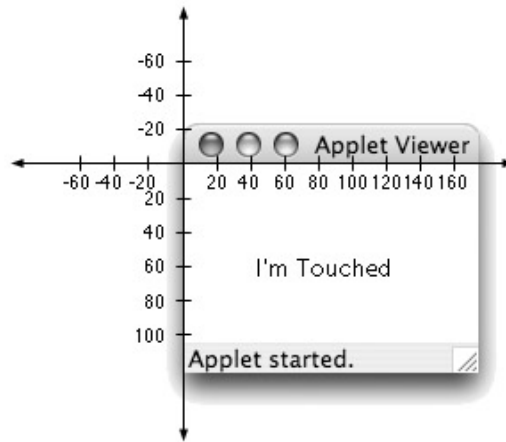


Figure 1.7: A program window and its drawing coordinate axes

dialect `objectdrawDialect` is a dialect that provides commands for creating programs that draw and modify items on the canvas. The new constructs explained in this section are all from the `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 be capable of popping 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`. Methods with these names are treated specially in objects inheriting from `aGraphicApplication`, as they will be requested by the system whenever the user presses or releases the mouse (or trackpad) button.

The parameters after the word `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 800 pixels wide and 200 high.

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 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 of the

window are (400, 400). The canvas in the window will be a bit smaller as the scroll bar on the right and bottom edges of the screen take up about 15 pixels each. Thus the lower right corner of the drawing area will be at about (385,385). **FIX!** ► *Fix library to make canvas given size?? Probably not.* ◀

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. However, it will only display the portion of these drawings that fall within the boundaries of your canvas.

1.4.3 Constructing Graphic Objects

The first lines in the object expression represent the definition of two locations and the construction of a text item on the canvas:

```
def clickTextLocation = aLocation.at(20,20)
def touchedTextLocation = aLocation.at(180,200)
```

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

The identifier `clickTextLocation` will represent the location with x and y coordinates which are both 20. The next line defines `touchedTextLocation` to correspond to the location with x coordinate 180 and y coordinate 200.

Finally the identifier `clickTextLocation` is used in constructing a text item at that location. Executing that object constructor will display “Click in this window” on the default canvas for graphics applications).

There is another statement creating text in the method named `onMousePress`:

```
aText.at(touchedTextLocation)with ("I'm touched") on (canvas)
```

When that construction is evaluated the text “I’m touched” will be displayed starting at the location named by `touchedTextLocation`.

In this text, we will tend to use definitions as abbreviations to make code more readable. In the above example, rather than defining `clickTextLocation`, we could have instead just written the text construction as:

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

However, we feel that giving the new location a name makes the code more readable.

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. As you might guess, 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 how to use it in a program later.

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 method’s 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 method’s body will be followed by the computer when the mouse is depressed within the program’s 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 the command `startGraphics` is executed, a 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 methods – 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 we want the object to be drawn, information about the other attributes of the item (e.g., it’s 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 the 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 canvas on which it is drawn.

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

```
def upperLeft = aLocation.at(0,0)
def lowerRight = aLocation.at(200,300)
aLine.from(upperLeft)to(lowerRight)on(canvas)
```

The line produced would look like the line shown in the window in Figure 1.8. **FIX!** ► *Again, the canvas is smaller than the window* ◀

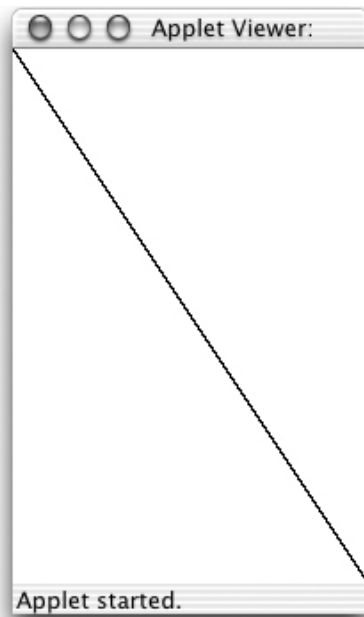


Figure 1.8: Drawing of a single line

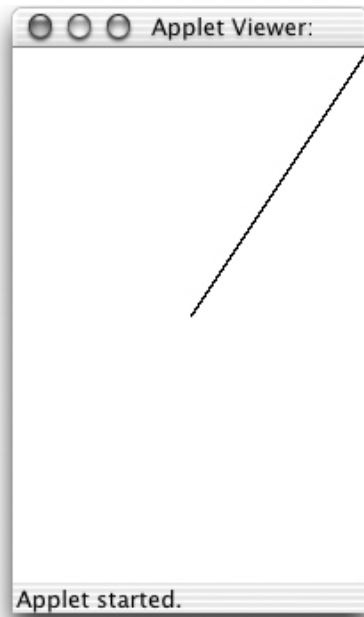


Figure 1.9: A line from (100,150) to (200,0)

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:

```
def midLocation = aLocation.at(100, 150)
def upperRightCorner = aLine.at(200, 0)
aLine.from(midLocation)to(upperRightCorner) 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 in the `TouchyWindow` program in Figure 1.6 with one or more other instructions. Such a modified program is shown in Figure 1.10. It draws two crossed



```
dialect "objectdrawDialect"
```

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

  def upperLeft = aLocation.at(40,40)
  def lowerRight = aLocation.at(60,60)
  def upperRight = aLocation.at(60,40)
  def lowerLeft = aLocation.at(40,60)

  // When the user presses mouse, draw a cross on canvas
  method onMousePress(mousePoint){
    aLine.from(upperLeft)to(lowerRight) on (canvas)
    aLine.from(upperRight)to(lowerLeft) 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.

lines in the center of the window.

The only differences between this example and `TouchyWindow` are that no text is written on the

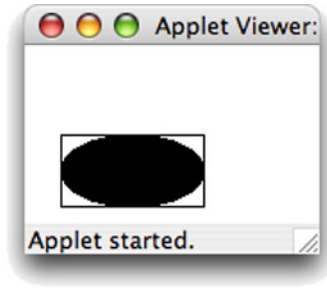


Figure 1.11: A `FilledOval` nested within a `FramedRect`

screen when this object is created, 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:

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

will display the frame, or outline, of an 80 by 40 rectangular box on the canvas. The coordinates 20, 50 is used to create the location of the box's upper left corner, which is associated with the identifier, `rectUpperLeft`. 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(rectUpperLeft) size (80, 40) on (canvas)
```

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

The command:

```
aFilledOval .at(rectUpperLeft) size (80, 40) on (canvas)
```

will draw a filled oval on the screen. The parameters are interpreted just like those to the `FilledRect` construction. However, instead of drawing a rectangle, `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(rectUpperLeft) size (80, 40) on (canvas)
aFilledOval .at(rectUpperLeft) 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 on 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 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.*

```
def loc1 = aLocation.at(0,100)
def loc2 = aLocation.at(100, 0)
def loc3 = aLocation.at(200,100)
```

```

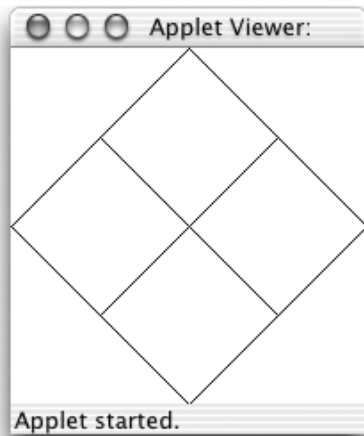
def loc4 = aLocation.at(100,200)
def loc5 = aLocation.at(50,50)

aLine.from(loc1)to(loc2) on (canvas)
aLine.from(loc3)to(loc4) on (canvas)
aLine.from(loc4)to(loc1) on (canvas)
aFramedRect.at(loc5)size(100, 100) on (canvas)

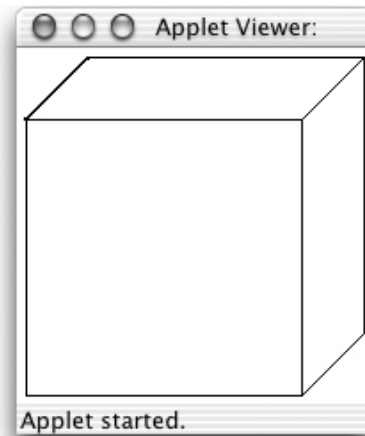
```

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 when the user performs an action with the mouse. When the user performs such an action, we the system generate a *mouse event* that can be handled by an object that extends `aGraphicApplication`.

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.

onMouseClicked 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.

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

If your computer uses a trackpad, touch screen, or other input mechanism, the above methods will be triggered by the equivalent actions with that input.

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 that is inside an object expression that is not included in a method body will be 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 object or application. 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.

In our `TouchyWindow` program shown in Figure 1.6, the initialization code created two locations and associated them with the names `clickTextLocation` and `touchedTextLocation`. With this information it wrote the instructions on the canvas. Finally, the command `startGraphics` created the window and made it ready to respond to mouse clicks. The code in the methods `onMousePress` and `onMouseRelease` was only executed after a mouse press or release event.

1.7 To Err is Human

FIX! ▶ *Section needs a lot of work once we decide on the IDE* ◀ 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 intent of 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 we discuss 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;
```

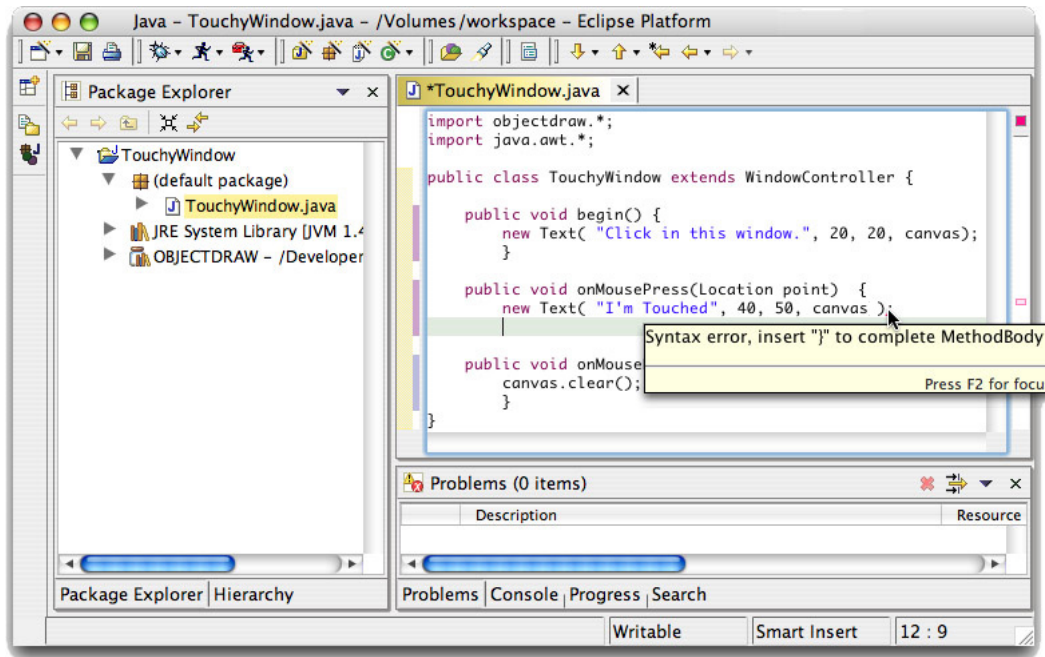


Figure 1.12: Eclipse displaying a syntax error message

instead of

```
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 Grace 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. Thus `onMooseRelease` 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

```
def oopsLoc = aLocation.at(400,50)
  aText.at(oopsLoc)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 how we can use a web browser to enter computer programs, translate them into a form the machine can follow, and run them.

Despite our best efforts to explain how the language and development system work, 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. We cannot stress enough that you can only learn to program through writing lots of programs. Now is a good time to start.

Chapter 2

What's in a name?

An important feature of a programming language is that the vocabulary used can be expanded by the programmer. Suppose you want to draw a line that ends at the current position of the mouse. The actual location of this point will not be determined until the program you write is being used. To talk about this position in your program you must introduce a name that will function as a place holder for the information describing the mouse's position. Such names are somewhat like proper names used to refer to the character in a story. You cannot determine their meanings by simply looking them up in a standard dictionary. Instead, the information that enables you to interpret them is part of the story itself. In this chapter, we will continue your introduction to programming in Grace by discussing in more detail how to introduce and use such names in Grace programs. In addition, we will introduce additional details of the primitives used to display graphics.

2.1 Naming and Modifying Objects

Constructions like:

```
aLine.from(oneLocation) to (anotherLocation) on (canvas)
```

provide the means to place a variety of graphic images on a computer screen. Most programs that display graphics, however, do more than just place graphics on the screen. Instead, as they run they modify the appearance of the graphics they have displayed in a variety of ways. Items are moved about the screen, buttons change color when the mouse cursor is pointed at them, text is highlighted, and often items are simply removed from the display. To learn how to produce such behavior in a Grace program, we must learn about operations that change the properties of objects after they have been constructed. These operations are called *mutator methods*, based on the non-biological meaning of the word “mutate”, to change or alter.

2.1.1 Mutator Methods

Just as each class of graphical objects has a specific name that must be used in a construction, each mutator method has a specific name. The names are chosen to suggest the change associated with the method, but there are some subtleties. For example, there are two mutator methods that can be used to move a graphical object to a new position on the screen. They are named `moveBy` and `moveTo`. The first tells an object to move a certain distance from its current position. The second is used to move an object to a position described by a pair of coordinates regardless of its previous

position.

With most mutator methods, including `moveBy` and `moveTo`, the programmer must specify additional pieces of information that determine the details of the operation applied. For example, when you send a request `moveBy` to an object, you need to tell it how far. The syntax used to provide such information is similar to that used to provide extra information in a construction. A comma-separated list of values is placed in parentheses after the method name. Thus, to move an object associated with the identifier `someObj` by 30 pixels to the right and 15 pixels down the screen one would say:

```
someObj.moveBy(30,15)
```

While the use of a mutator method shares portions of the syntax of a construction, there are major syntactic and conceptual differences between the two. A construction produces a new object. A mutator method is used to modify an already existing object. To make this clear, let us consider a simple example.

Many programs start by displaying an entertaining animation. With our limited knowledge of Grace, we can't yet manage an entertaining animation, but we can, with a bit of help from the program's user, create a very simple animation. In particular, we can write a program that displays a circle near the bottom of the canvas and then moves the circle up a bit each time the mouse is clicked. With a bit of imagination, you can think of the circle as the sun rising at the dawn of a new day.

Without knowing anything about mutator methods, you should be able to imagine the rough outline of such a program. Basically, from the description it is clear that the object inheriting from `aGraphicApplication` needs to construct a `FilledOval`. It is also clear that the program will need to define an `onMouseClicked` method that uses the `moveBy` mutator method. You don't know enough to write this method yet. The fact that `onMouseClicked` will be used, however, should tell you a bit more about the code in the object. If the user of our program needs to click the mouse to get it to function, then, just as we did in our improved version of `TouchyWindow`, we should include code to display instructions telling the user to do this. So, your first draft of an object expression might look something like:

```
object {
  inherits aGraphicApplication . size (200,200)

  def startSun = aLocation.at(50, 150)
  def startInstructions = aLocation.at(20, 20)

  aFilledOval .at(startSun) size (100, 100) on (canvas)

  aText.at( startInstructions ) with ("Click the mouse repeatedly") on (canvas)

  startGraphics
}
```

This code will produce an image like that shown in Figure 2.1. In an effort to make it look like the sun is rising over the horizon, the oval is positioned so that its bottom half extends off the bottom of the window leaving only the top half visible. Of course, it might help your imagination if the "sun" were yellow, but we will have to wait until we learn a bit more Grace before we can fix that.

Now, consider how we would complete the program by writing the `onMouseClicked` method. To make an object move directly upwards, we need to use the `move` method specifying 0 as the distance

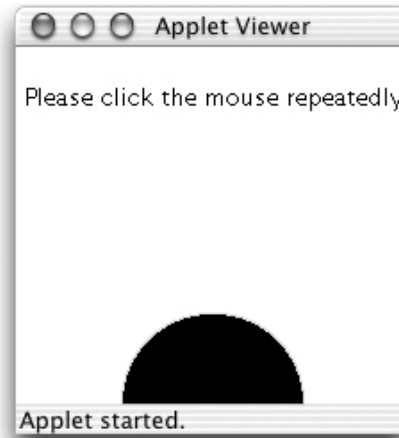


Figure 2.1: An oval rises over the horizon

to move horizontally and some negative number for the amount of vertical motion desired since y coordinates decrease as we move up the screen. Something like:

```
moveBy(0,-5);
```

might seem appropriate. The problem is that if this is all we say, Grace will not know what to move. In the body of the **object** expression above, we construct two graphical objects, the filled circle and the text displaying the instructions. Since they are both graphical objects, we could move either of them. If all we say is `moveBy`, then Grace has no way of knowing what we want moved.

To avoid ambiguities like this, Grace won't let us simply say `moveBy`. Instead we have to tell a particular object to move. In general, Grace requires us to identify a particular object as the target whenever we wish to use a mutator method. You have already seen an example of the Grace syntax used to provide such information. In our first example program, when we wanted to remove a message from the screen we included a line of the form:

```
canvas.clear
```

`clear` is a mutator method associated with drawing areas. The word `canvas` is the name Grace gives to the area in which we can draw. Saying `canvas.clear` tells the area in which we can draw that all previous drawings should be erased. When we tell an object to perform a method in this way we say we have *requested* the method. Alternately, we might say that we sent a `clear` message to the `canvas`.

In general, to apply a method to a particular object, Grace expects us to provide a name or some other means of identifying the object followed by a period and the name of the method to be used. So, in order to move the oval created in the body of the **object** expression, we first have to tell Grace to associate a name with the oval.

2.1.2 Definitions

First, we have to choose a name to use. Grace puts a few restrictions on the names we can pick. Names that satisfy these restrictions are called *identifiers*. An identifier must start with a letter. After the first letter, you can use any combination of letters, digits, underscores, or single quotes ('). So, we could name our oval something like `sunspot`, `oval2move` or `sun.ra'`. Case is significant, so `sun`

```

dialect "objectdrawDialect"

object {
  inherits aGraphicApplication . size(200,200)

  def startSun = aLocation.at(50, 150)
  def startInstructions = aLocation.at(20, 20)

  def sun = aFilledOval.at (startSun) size (100, 100) on (canvas)

  aText.at( startInstructions ) with ("Click the mouse repeatedly") on (canvas)

  method onMouseClick(point){
    sun.moveBy(0,-5)
  }

  startGraphics
}

```

Figure 2.2: Declaring sun in the RisingSun program

is different from Sun. An identifier can be as long (or short) as you like, but it must be just one word (i.e. no blanks or punctuation marks are allowed in the middle of an identifier). A common convention used to make up for the inability to separate parts of a name using spaces is to start each part of a name with a capital letter. For example, we might use a name like `ovalToMove`. It is also a convention to use identifiers starting with lower case letters to name variables to help distinguish them from the names of types (which we will talk about later).

A sequence of letters, numbers, single quotes, and underscores can be used as an identifier in a Grace program even if it has no meaning in English. Grace would be perfectly happy if we named our box `e2dLiw0'`. It is much better, however, to choose a name that suggests the role of an object. Such names make it much easier for you and others reading your code to understand its meaning. We suggested earlier that you could think of the display produced by the program we are trying to write as an animation of the sun rising. In this case, `sun` would be an excellent name for the oval. We will use this name to complete this example.

As we saw in the last chapter, the syntax of a definition is very simple. For each name you plan to use, you enter the word **def** followed by the name you wish to introduce, an equals sign, and the value of the identifier. So, to define the name `sun`, which we intend to use to refer to a `FilledOval`, we would add the declaration:

```
def sun = aFilledOval.at (startSun) size (100, 100) on (canvas)
```

Now that we have associated a name with the oval, we can tell grace to move it by writing `sun.moveBy(0,-5)`

As a result, the contents of the program file for our animation of the sunrise might begin with the code shown in Figure 2.2.

The form and placement of a definition within a program determines where in the program the name can be used. This region is called the *scope* of the name. In particular, we will want to refer to the name `sun` in the body of the **object** expression and in the `onMouseClick` method of the program we are designing. The declaration of names that will be used in several methods should be placed

within the braces that surround the body of our object (or class), but outside any of the method bodies. We recommend that these definitions be placed before all the method declarations in order to make them easier to find when reading the rest of the code. Only code within the object we are defining is allowed to refer to this name. Method names, by way of contrast, are generally accessible outside of the class. We will learn later how to modify the visibility of definitions and methods.

One purpose of requiring an identifier to be introduced via a definition is to enable Grace to give you helpful feedback if you make a mistake. Suppose that after deciding to use the name `sun` in our program we made a typing mistake and typed `sin` in one line where we meant to type `sun`. It would be nice if when Grace tried to run this program it could notice such a mistake and provide advice on how to fix the error similar to that provided by a spelling checker. To do this, however, Grace needs the equivalent of a dictionary against which it can check the names used in the program. The definitions a programmer includes in their program constitute this dictionary. If Grace encounters a name that was not declared, it reports it as the equivalent of a spelling mistake.

2.1.3 Variable Declarations

In the previous section we saw how we could associate names to objects via definitions. As we have seen with the `sun` example, we may request mutator methods on these objects, and they may change their properties. For example, when we evaluated `sun.moveBy(0,-5)`, the object associated with `sun` shifted up 5 pixels on the screen. However, once a value is associated with a name in a definition, that name may not refer to a different object. For example, once the system has processed the definition, we could not associate `sun` with a rectangle or even a different filled oval.

If we want a name that can be associated with different objects over time then we must declare a *variable*. We can see the differences between definitions and variables in everyday conversation. Proper names are usually used as definitions: "Barack Obama" refers to the individual who served as U.S. president from 2008 to 2016. In 2007, the phrase "the president of the U.S." referred to George Bush. In 2009, it referred to Barack Obama.

When we introduce identifiers in a language, we need to decide what kind of identifier they will be. If they will always refer to the same object then we introduce them with a definition. If they sometimes refer to one object, but at others a different one then they will be represented via a variable – that is, what they refer to may vary. A simple variable declaration uses the keyword **var** as follows:

```
var president := "Barack Obama"
```

There are two key differences between a variable declaration and a definition. First, we use the keyword **var** instead of **def**. Second, the value is associated with a variable using `:=`, while the value is associated with a definition using `=`. The `=` should suggest that the name is always identical to the value associated with it, while the variant of `:=` indicates that different values may be associated with the name over the course of the execution of the program.

A variable declared in an object is known as an instance variable. We will see later that we can also declare variables inside methods.

It is also possible to declare an instance variable without assigning it a value. Of course we will have to assign a value to it before we use it. Otherwise we will get an error.

We will see several examples of variables in programs later in this chapter.

2.1.4 Comments

In the complete version of the program, we introduce one additional and very important feature of Grace, the *comment*. As programs become complex, it can be difficult to understand their operation by just reading the Grace code. It is often useful to annotate this code with English text that explains more about its purpose and organization. In Grace, one can include such comments in the program text itself as long as you follow conventions designed to enable the computer to distinguish the actual instructions it is to follow from the comments. This is done by preceding such comments with a pair of slashes (“//”). Any text that appears on a line after a pair of slashes is treated as a comment by Grace.

The program we are writing seems a good example in which to introduce comments. Although the program is short and simple, it is not clear that someone reading the code would have the imagination to realize that the black circle created by the program was actually intended to reproduce the beauty of a sunrise. The Grace language isn’t rich enough to allow one to express non-technical ideas in code, but we can include them in comments. We include some general guidance on using comments to make your programs easier to read and understand in Appendix ??.

```
dialect "objectdrawDialect"

// A program that produces an animation of the sun rising.
//The animation is driven by clicking the mouse button.
//The faster the mouse is clicked, the faster the sun will rise.
object {
  inherits aGraphicApplication . size(200,200)

  // location where the sun starts
  def startSun = aLocation.at(50, 150)

  // location for the instructions
  def startInstructions = aLocation.at(20, 20)

  // Circle that represents the sun
  def sun = aFilledOval.at (startSun) size (100, 100) on (canvas)

  // Place instructions on the screen
  aText.at( startInstructions ) with ("Click the mouse repeatedly") on (canvas)

  // Move the sun up a bit each time the mouse is clicked
  method onMouseClick(point){
    sun.moveBy(0,-5)
  }

  startGraphics
}
```

Figure 2.3: Commented code for rising sun example

The object definition in Figure 2.3 is preceded by three lines of comments. We add a line of comment for each of the major segments of the program. In this case, the definitions of the two locations and circle, the display of instructions, and the declaration of the `onMouseClick` method.

2.1.5 Additional Mutator Methods

There are several other operations that can be applied to graphical objects once we have the ability to associate names with the objects. For example, as we mentioned earlier, there is a mutator method named `moveTo` which moves an object to a specific location on the screen.

Many objects also have mutator methods that are designed to look like assignment statements. As we will see below, there is a method named `isVisible :=` that can be used to temporarily remove or add a graphical item from or to the screen. We can use these methods to extend the behavior of our `RisingSun` program.

First, the version of the program shown above becomes totally uninteresting after the mouse has been clicked often enough to push the filled oval off the top of the canvas. Once this happens, additional mouse clicks have no visible effect. It would be nice if there was a way to tell the program to restart by placing the sun back at the bottom of the canvas. Second, as soon as the user starts to click the mouse, the instructions asking the user to click become superfluous. Worse yet, at some point, the rising sun will bump into the instructions. It would be nice to remove the instructions from the display temporarily and then restore them when the program is reset.

All that we need to do to permit the resetting of the sun is to add the following definition of the `onMouseExit` method to our `RisingSun` class.

```
method onMouseExit(point) {  
    sun.moveTo(startSun)  
}
```

With this addition, the user can reset the program by simply moving the mouse out of the program's canvas. When this happens, the body of the `onMouseExit` method will tell Grace to move the sun oval back to its initial position.

Making the instructions disappear and then reappear is a bit more work. In order to apply mutator methods to the instructions, we will have to tell Grace to associate a name with the `Text` created to display the instructions. As we did to define the name `sun`, we will have to both declare the name we wish to use and associate it with the creation of the `Text`.

An obvious name for this object is "instructions." If this is our choice, then we would need to tell Grace that we planned to use this name by adding a declaration of the form:

```
def instructions = aText.at( startInstructions )  
    with("Click the mouse repeatedly")on(canvas)
```

to the body of the **object** expression.

As in this example, it is sometimes helpful to split a long command into several lines. It can make your code much easier to read and understand. Using multiple lines for one command like this is perfectly acceptable in Grace. You can split an instruction between two lines at any point where you could type a space except within quoted text. Be careful, however, to make sure the continuation line is further indented than the beginning of the statement. You will also find that the use of indentation and blank lines can make groups of related commands stand out to the reader. These issues are discussed further in Appendix ??.

The mutator method named `isVisible :=` provides the means to temporarily remove or add a bit of graphics from the display. To make the text disappear when the mouse is clicked, we would include an instruction of the form:

```
instructions . isVisible := false
```

in the `onMouseClicked` method. Each time the mouse is clicked, the instructions will be told to become invisible. Of course, once they are invisible, telling them to become invisible again has no effect.

This mutator method is intentionally designed to look like a variable assignment statement, because it changes an attribute (i.e., a quality or characteristic) of the object. The main difference is that the left side of the statement includes a “.”.

When the program is reset, we want the instructions to reappear. To do this, we need to include an instruction of the form

```
instructions . isVisible := true;
```

in the `onMouseExit` method. The complete text of this revised program is shown in Figure 2.4.

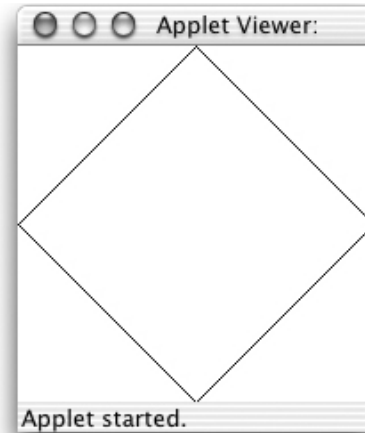
The `isVisible` method should only be used to hide an object when it is being removed from the canvas *temporarily*. If an object is being removed *permanently*, that is, you know that your program will never make it visible again, there is another method named `removeFromCanvas` that is more appropriate. The `removeFromCanvas` method irreversibly removes a graphical object from the display. There is no way to put an object that has been removed in this way back on the display. When an object is made invisible, the system must save information needed to display the object in case it is made visible again. This consumes space in the computer’s memory. Using `removeFromCanvas` instead allows the system to totally remove information about the object from the computer’s memory.

2.1.6 Exercises

Below are several exercises in which we will ask you to consider how one could write a program that displayed a diamond that appeared to grow a bit each time the mouse was clicked. The following constructions will produce a drawing of the initial diamond shape desired if executed in a program whose canvas is 200 by 200 pixels.

```
def left = aLocation.at(0,100)
def right = aLocation.at(200,100)
def top = aLocation.at(100,0)
def bottom = aLocation.at(100,200)

aLine.from(top)to(right)on(canvas)
aLine.from(top)to(left)on(canvas)
aLine.from(bottom)to(right)on(canvas)
aLine.from(bottom)to(left)on(canvas)
```



The complete program we have in mind won’t actually make the diamond grow. Instead, all it will do is move each of the four lines drawn by the constructions a bit closer to the corner closest to the line each time the mouse is clicked. For example, each time the mouse is clicked, the line in the upper left corner of the window will be moved one pixel to the left and one pixel up so that it ends up closer to the upper left corner of the window. The line that starts in the upper right quarter of the window, on the other hand, will be moved one pixel to the right and one pixel up with each click of the mouse. If this is done, after a number of clicks, the display will look like picture shown below. The four lines won’t be any longer than they were at the start, but they will appear to be part of a bigger diamond than was originally drawn, a diamond that is too big to fit in the window.

```

dialect "objectdrawDialect"

// A program that produces an animation of the sun rising.
// The animation is driven by clicking the mouse button.
// The faster the mouse is clicked, the faster the sun will rise.

object {
  inherits aGraphicApplication . size (200,200)

  // location where the sun starts
  def startSun = aLocation.at(50, 150)

  // location for the instructions
  def startInstructions = aLocation.at(20, 20)

  // Circle that represents the sun
  def sun = aFilledOval .at (startSun) size (100, 100) on (canvas)

  // Place instructions on the screen
  def instructions = aText.at( startInstructions ) with ("Click the mouse repeatedly") on (canvas)

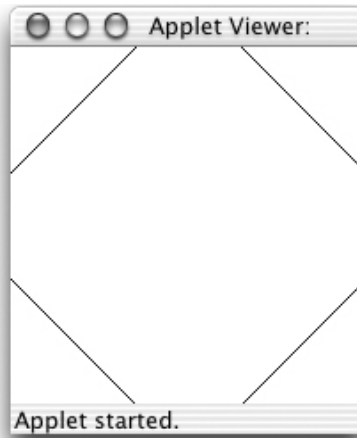
  // Move the sun up a bit each time the mouse is clicked
  method onMouseClick(point){
    sun.moveBy(0,-5)
    instructions . isVisible := false
  }

  // Move the sun back to its starting position and redisplay
  method onMouseExit(point){
    sun.moveTo(startSun)
    instructions . isVisible := true
  }

  startGraphics
}

```

Figure 2.4: Rising sun program with reset feature



Exercise 2.1.1 Before we can move one of these lines or any other graphical object, we must first associate a name with the object. A list of possible names that might be used to refer to the line of the diamond shape drawn by the construction

`aLine.from(bottom)to(right)on(canvas)`

are shown below. For each name, indicate whether it would be:

- **invalid** according to Grace's rules for forming names,
- **inappropriate** as a name for this particular line because it would not help a person working with the program remember the purpose of the name or it does not conform to Grace's naming conventions, or
- **appropriate** as a name for this particular line.

In each case, briefly explain your answer.

<code>3rdLine</code>	<code>thirdLine</code>	<code>line3</code>
<code>Leftlower</code>	<code>lower right line</code>	<code>lowerLeft</code>
<code>southEast</code>	<code>S.E.</code>	<code>south-east</code>

Exercise 2.1.2 Suppose that you selected the instance variable names `leftToTop`, `topToRight`, `rightToBottom`, and `bottomToLeft` to describe the four lines that form the diamond. Show the declarations required to introduce these names in a Grace program.

Exercise 2.1.3 Assuming that the names `leftToTop`, `topToRight`, `rightToBottom`, and `bottomToLeft` had been chosen to describe the four lines that form the diamond and that they had been declared appropriately, show how the constructions shown above would be turned into definitions that both created the lines and associated the names listed with them. Where should these commands be placed if you want the lines to appear in their initial positions as soon as the program is started?

Exercise 2.1.4 Assuming that the names `leftToTop`, `topToRight`, `rightToBottom`, and `bottomToLeft` had been declared appropriately and associated with the lines of the diamond using assignment statements, show the statements required to invoke the `move` method on each line to move the line closer to the nearest corner of the program's window. Each line should be moved one pixel horizontally and one

pixel vertically. In which method should these commands be placed if you want the lines to move each time the mouse is clicked?

Exercise 2.1.5 Assuming that the names `leftToTop`, `topToRight`, `rightToBottom`, and `bottomToLeft` had been declared appropriately and associated with the lines of the diamond using assignment statements, describe the effect the following statements would have on the lines each time they were executed. In addition, sketch the contents of the window after these lines were executed 100 times.

```
leftToTop.moveBy(0,1);
topToRight.moveBy(0,1);
rightToBottom.moveBy(0,-1);
bottomToLeft.moveBy(0,-1);
```

2.2 Non-graphical Classes of Objects

We have seen how to construct graphical objects of several varieties, associate names with them and apply methods to these objects. All of the objects we have worked with, however, have shared the property that they are graphical in nature. When we create any of these objects, they actually appear somewhere on the computer's screen. This is not a required property of objects that can be manipulated by a Grace program. In this section we will introduce two classes of objects that are related to producing graphics but do not correspond to particular shapes that appear on your screen.

2.2.1 The Class of Colors

So far, all the graphics we have drawn on the screen have appeared in black, as black is the default color in which graphics are drawn. To add a little variety to the display, we can change the color of any graphical object in the object draw library by a statement of the form

```
grObj.color := newColor
```

where `grObj` is the graphical object and `newColor` is the new color. Just as we saw earlier with `isVisible`, `color:=` is actually a method request with parameter `newColor`.

It would certainly be an improvement to make the sun displayed by our `RisingSun` program appear yellow or any other more “sun-like” color than black. It is quite simple to make this change. All we have to do is add a line to reset the color to our object. The revised program would look like:

```
dialect "objectdrawDialect"
```

```
// A program that produces an animation of the sun rising.
//The animation is driven by clicking the mouse button.
//The faster the mouse is clicked, the faster the sun will rise.
object {
  inherits aGraphicApplication . size(200,200)

  // location where the sun starts
  def startSun = aLocation.at(50, 150)

  // location for the instructions
  def startInstructions = aLocation.at(20, 20)

  // Circle that represents the sun
```

```

def sun = aFilledOval.at (startSun) size (100, 100) on (canvas)
sun.color := yellow

// Place instructions on the screen
aText.at( startInstructions ) with ("Click the mouse repeatedly") on (canvas)

// Move the sun up a bit each time the mouse is clicked
method onMouseClick(point){
    sun.moveBy(0,-5)
}

startGraphics
}

```

When resetting the color, we must specify a color value. The simplest way to specify the color is to use one of Grace's built-in color names. We chose `yellow` for our sun, but if we wanted a more dramatic sunrise we could have instead used `red`. The `objectdraw` library in Grace provides names for all the basic colors including white, black, green, red, gray, blue, cyan, magenta, and neutral. Of course, there are too many shades of each color to assign a name to every one. Accordingly, the `objectdraw` library provides a class for creating new colors.

We have seen that names in Grace can be associated with objects like the drawing area (`canvas`), or graphical objects we have created. You might suspect that if a color can be associated with a name it might be thought of as an object. In this case, you would be right.

If we want to set the color of our sun to something not included in the small set of colors that have names, we can describe the particular color we want by writing:

```
aColor.r (...) g (...) b (...)
```

as long as we know the right information to use in place of the dots between the parentheses.

When we want to construct a new color for use in a program, we must provide a numerical description of the color as parameters to the construction. Grace uses a system that is fairly common for specifying colors on computer systems. Each color is described as a mixture of the three primary colors: red, green and blue.¹ The mixture desired is specified by giving three numbers, each of which specifies how much of a particular color should be included in the mixture. Each of the numbers can range from 0 ("use none of this particular primary color") to 255 ("use as much of this color as possible"). If the numbers for the red, green, and blue components are all 0's then the color is black. If all are 255 then the color is white. If the green component is 255 and the other two are 0 then not surprisingly we get green. If both the red and blue components are 255, while the green is 0 then we get a shade of purple. So if you want to make the sun purple you could construct the desired color by saying:

```
aColor.r(255)g(0)b(255)
```

Colors are objects. Therefore, just as you can associate names with objects such as `FilledOvals`, you can associate names with colors. If you wanted to use the color purple in a program you might first define the identifier `purple` as:

```
def purple = aColor.r(255)g(0)b(255)
```

You could then make the sun purple by replacing the statement used to make the sun yellow by:

¹If you thought the primary colors were red, yellow and blue you aren't confused. Those are the primary colors when mixing materials that absorb light (like paint). When mixing light itself (as in flashlight beams or the light given off by the phosphors on a computer screen), red, blue and green act as the primary colors. Mixing red and blue lights produces purple. Mixing red and green produces yellow. Mixing all three primary colors together produces white.


```
sun.color := purple
```

It is worth noting that it is not actually necessary to introduce a new name to use a color created using a construction. When we update the color of an object, we have to provide the color we wish to use as to the right of `:=`, but we can describe the desired color in several ways. In particular, we could make the sun purple by simply writing

```
sun.color := aColor.r(255)g(0)b(255)
```

The construction describes the color just as well as the name `purple` from Grace's point of view.

In general, wherever Grace allows us to identify an object or value by name, it will accept any other phrase that describes the equivalent object.

Just as we can build a color using numbers representing their red, green, and blue components, we can send method requests to colors asking for those same numeric components. The names of the methods are `red`, `green`, and `blue`. Thus if `someColor` is an identifier associated with a color, `someColor.red` will return a number corresponding to the red component.

If we wish to print out information about a string, we can also ask it for a string representation of the color using the method `asString`. For example, if `purple` is as defined above then `purple.asString` returns the string `"rgb(255,0,255)"`, which could be used in a print statement.

We will find it useful to include a method `asString` for most of our objects so that we can retrieve and print useful information about them. For example, if we execute the following code snippet:

```
def origin = aLocation(0,0)
def myRect = aFramedRect.at (origin) size (50,100) on (canvas)
print (myRect.asString)
```

then the system will print out `"FramedRect at (0,0) with height 50, width 100, and color rgb(0,0,0)"`.

In fact, because `asString` is provided for all objects, you may leave off the `asString` in a print statement and the system will automatically apply the method. Thus you may simply write `print(myRect)` and the exact same string will be printed.

Unlike our graphical objects, colors are immutable. That is, there are no methods that modify its state. While we can use the mutator method `moveBy` on a framed rectangle in order to change its position, there are no mutator methods for colors. If you want a color that is slightly different from an existing color, you must create a new color rather than modifying the original.

For example, suppose you wanted a color that was a bit lighter than the existing color associated with the identifier `someColor`. Then we could create a new color whose components are each 5 units greater than that by writing

```
aColor.r(someColor.red+5)g(someColor.green+5)b(someColor.blue+5)
```

Exercise 2.2.1 *Write a method that creates a red oval when the mouse is clicked. The oval should be 100 pixels in width and 75 pixels in height and should appear 50 pixels down from the top of the canvas and 50 pixels in from the left of the canvas.*

```

dialect "objectdrawDialect"

object {
  inherits aGraphicApplication . size (400,400)
  var horizontalCorner := aLocation.at(0,0)
  var verticalCorner := aLocation.at(0,0)

  method onMouseClick(point: Location){
    aFilledRect .at( verticalCorner )size( 5, 200) on (canvas )
    aFilledRect .at( horizontalCorner )size( 200, 5) on (canvas )
    verticalCorner := verticalCorner . translate (10,0)
    horizontalCorner := horizontalCorner . translate (0,10)
  }

  startGraphics
}

```

Figure 2.5: An application of the translate method

2.2.2 The Location Class

As we have seen above, we can use a triple of numbers to construct an object representing a color. Earlier we saw that we could use pairs of numbers to create locations in our drawing window by constructing objects using `aLocation.at(x,y)`.

Like the color objects described above, locations are immutable. There are no mutator methods that they respond to. The methods that are associated with locations are `x`, `y`, `translate`, and `distanceTo`.

Not surprisingly, the first two return the `x` and `y` coordinates of the location. Writing `startingLocn . translate (dx,dy)`, where `dx` and `dy` are numbers returns a new location that is `dx` to the right and `dy` down from `startingLocn`. The method request `startingLocn.distanceTo(endLocn)` returns the distance from `startingLocn` to `endLocn`.

To clarify how `translate` works, consider the sample program shown in Figure 2.5. Each time the mouse is clicked, this program will draw what appear to be a pair of thick, perpendicular lines, though in reality they are very thin rectangles. The first lines drawn will intersect at the upper left corner of the window. With each click, the lines drawn will be placed to the left and below the preceding lines so that the window is eventually filled with a grid pattern.

The program includes two variables named `verticalCorner` and `horizontalCorner`. Each of these names describes the location at which one of a pair of `FilledRect`s will be drawn when the user clicks the mouse.

When the object expression is evaluated the computer creates two objects that both describe the point at the origin of the coordinate system, the upper left corner of the canvas. One of these objects is associated with the name `verticalCorner` and the other with the name `horizontalCorner`. Although they both refer to the origin initially, two distinct variables are needed because they will be updated to describe different locations using the `translate` method in other parts of the program.

The names assigned to these two locations reflect the way they are used in the program's other method, `onMouseClick`. Each time the mouse is clicked, this method creates two long, thin rectangles on the screen. The rectangle created by the first construction in `onMouseClick` is a long vertical rectangle. The position of its upper left corner is determined by `verticalCorner`. The other rectangle



Figure 2.6: Display after one click

is a long horizontal rectangle and its position is determined by `horizontalCorner`.

Since both of the `Locations` initially describe the upper left corner of the canvas, the first time the mouse is clicked, the rectangles created by the execution of the first two lines of `onMouseClicked` will produce a drawing like that shown in Figure 2.6.

The last two commands in `onMouseClicked` tell the `Location` objects named `verticalCorner` and `horizontalCorner` to create new locations using `translate` so that they describe new positions on the screen. `verticalCorner.translate(10,0)` returns the position 10 pixels to the right of its initial position. That new location is then associated with `verticalCorner` by the assignment statement with `verticalCorner` on the left.

The expression `horizontalCorner.translate(0,10)` provides a location 10 pixels down from `horizontalCorner`. After this value has been calculated, `horizontalCorner` is updated to this new location.

When these assignment statements are completed, nothing changes on the screen. The program's window will still appear as shown in Figure 2.6 after they have been performed. However, the variables will refer to the new positions

The next time the mouse is clicked the two constructions at the beginning of `onMouseClicked` are performed based on the new locations associated with the two variables. Accordingly, the vertical rectangle created will appear a bit farther to the right and the horizontal rectangle will appear a bit farther down the screen as shown in Figure 2.7. After these rectangles are created, the last two lines of the method will again create new location farther to the right and down the screen so that the rectangles produced by the next click will appear at the correct locations.

This process will be repeated each time the mouse is clicked. After about 15 additional clicks, the window will be nearly filled with a grid of lines as shown in Figure 2.8. Just a few more clicks will complete the process of filling the screen.

2.3 Layering on the Canvas

Now that we can set the color of a graphical object, we could actually make a much prettier picture for our rising sun program. We will add a blue sky and some white clouds to go with the sun. The

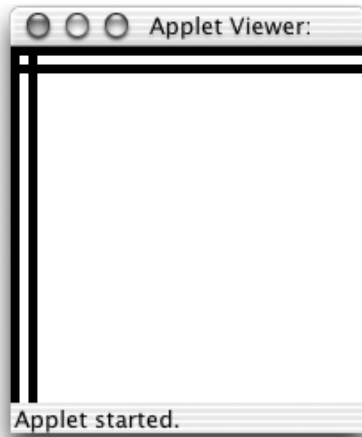


Figure 2.7: Display after second click

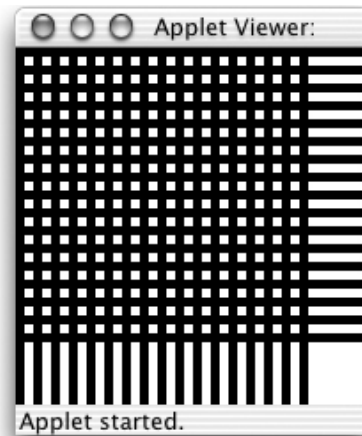


Figure 2.8: Display after many clicks

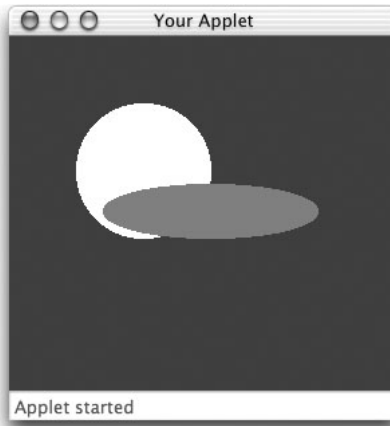


Figure 2.9: Today's forecast: Partly cloudy

picture we have in mind is shown in Figure 2.9.

While we could just create the objects for the sky, sun, and cloud, we will need to associate them with names so that we can set the colors:

```
def origin = aLocation.at(0,0)
def startSun = aLocation.at(50, 150)
def startCloud = aLocation.at(70, 1100)

def sky = aFilledRect.at ( origin ) size (300, 300) on (canvas)
def sun = aFilledOval.at(startSun) size (100, 100) on (canvas)
def cloud = aFilledOval.at(startCloud) size (160, 40) on (canvas)
```

We then set their colors appropriately using their names:

```
sky.color := blue
sun.color := yellow
cloud.color := white
```

Suppose we changed this code by reordering the definitions so that the sun is created after the cloud as in:

```
def sky = aFilledRect.at ( origin ) size (300, 300) on (canvas)
def cloud = aFilledOval.at(startCloud) size (160, 40) on (canvas)
def sun = aFilledOval.at(startSun) size (100, 100) on (canvas)
```

This will change the picture drawn so that it resembles the drawing shown in Figure 2.10. The sun and the cloud still overlap, but now part of the cloud is hidden behind the sun rather than the other way around. (We know, this can never happen in real life, but let's pretend!)

The canvas views the collection of objects it has been asked to display as a series of layers. When a new object is created it is placed in a layer above all the older objects on the canvas. When two objects overlap, the object on the lower level will be partially or completely hidden by the object on the upper level. In the code to produce Figure 2.9, the sun was created before the cloud and therefore was drawn as if it was underneath the cloud. In the revised code, the moon was created last and therefore is drawn as if it is above the cloud.

None of the methods we have considered so far changes this layering. Changing an object's color or moving it will not change the way in which it is drawn when it overlaps with other objects.

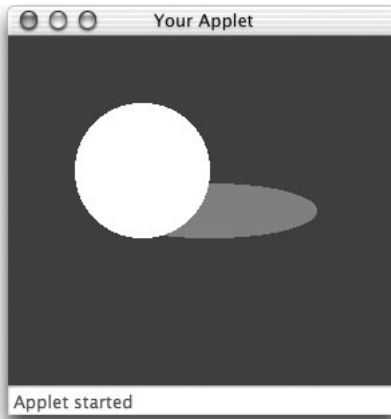


Figure 2.10: Tonight’s forecast: Partly sunny?

If we included code to move the sun as in our `RisingSun` program, the sun would move vertically on the screen, but it would still remain on a layer below the cloud. It would therefore appear to slide upward while remaining behind the cloud. Even resetting the `isVisible` attribute preserves this ordering. If we reset the `isVisible` attribute of an object from false to true, it will reappear underneath the same objects that have been above it before it was hidden. In particular, it does not appear at the top level as if it had just been created.

There are, however, several methods that are provided to enable a programmer to rearrange the layering of objects on the screen. The methods `sendForward` and `sendBackward` move an object up or down one layer. The methods `sendToFront` and `sendToBack` move an object to the top or bottom of all the layers drawn. Thus, if we had created the picture shown in Figure 2.9 and then executed either the command

```
cloud.sendBackward
```

or the command

```
sun.sendToFront
```

the picture displayed would change to look like Figure 2.10.

2.4 Accessing the Location of the Mouse

In the header of every mouse event handling method we have included the identifier `point` in parentheses after the name of the method. Now that we have explained what a location is, we can explain the purpose of this phrase. It provides a means by which we can refer to the point at which the mouse cursor was located when the event handled by a method occurred. Basically, within the body of a mouse event handling method that includes the identifier `point` we can use the name `point` to refer to a location object that describes where in the canvas the mouse was positioned.

As a simple example, we can write a variant of our very first example program “`TouchableWindow`”. This new program will display a bit of text on the screen when the mouse is pressed, just like `TouchableWindow`, but:

1. it will display the word “`Pressed`” instead of the phrase “`I’m Touched.`”,

2. it will place the word where the mouse was clicked instead of in the center of the canvas, and finally
3. it will not erase the canvas each time the mouse is released

The code for this new example is shown below.

```
dialect "objectdrawDialect"

// program that displays the word "pressed" wherever
// the mouse is pressed
object {
  inherits aGraphicApplication . size (400,400)

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

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

Note that the name `point` is used in the construction that places the Text “Pressed” on the screen. Because `point` is included in the method’s header, Grace knows that we want the computer to make this name refer to the location at which the mouse was clicked. Therefore Grace will place the word “Pressed” wherever the mouse is pressed.

An identifier that is enclosed by parentheses in a method header is known as a *formal parameter* or just *parameter*. As in variable declarations and definitions, we are free to use any name we want when we declare a formal parameter. There is nothing special about the name `point` (except that we have used it in all our examples so far). We can choose any name we want for the mouse location as long as we place the name in parentheses in the header of the method. For example, the program shown in Figure 2.11, which uses the name `mousePosition` as its formal parameter instead of `point`, will behave exactly like the version that used the name `point`.

2.5 Sharing Parameter Information between Methods

Another important aspect of the behavior of formal parameter names like `mousePosition` and `point` is that each formal parameter name is meaningful only within the method whose header contains its declaration. To illustrate this, consider the program shown in Figure 2.12. This program displays the word “Pressed” at the current mouse position each time the mouse button is pushed, and it displays the word “Released” at the current mouse position each time the mouse is released. The `onMousePress` method in this example is identical to the corresponding method from the Pressed example except for the name chosen for its formal parameter. The `onMouseRelease` method is also quite similar to the earlier program’s `onMousePress`.

Exercise 2.5.1 *When the above program is executing, if the user clicks the mouse (a quick press and release without moving the mouse), then the words “Pressed” and “Released” will be written on top of each other. What happens if the user presses the mouse, drags it to somewhere else on the canvas and only then releases the mouse?*

```

dialect "objectdrawDialect"

// program that displays the word "pressed" wherever
// the mouse is pressed
object {
  inherits aGraphicApplication . size(400,400)

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

  // create window and start graphics
  startGraphics
}

```

Figure 2.11: Using a different parameter name

```

dialect "objectdrawDialect"

// program that displays the words "pressed" and
// "released" when the mouse is pressed or released
object {
  inherits aGraphicApplication . size(400,400)

  // When the user presses mouse, write "Pressed"
  // on canvas at current mouse location
  method onMousePress(pressPosition)->Done{
    aText.at( pressPosition )with ("Pressed") on (canvas)
  }

  // When the user releases mouse, write "Released"
  // on canvas at current mouse location
  method onMouseRelease(releasePoint)->Done{
    aText.at( releasePoint )with ("Released") on (canvas)
  }

  // create window and start graphics
  startGraphics
}

```

Figure 2.12: A program to record mouse button changes

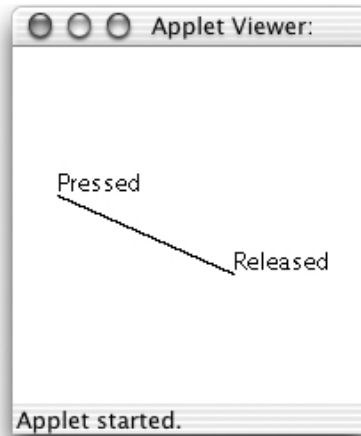


Figure 2.13: Connecting the ends of a mouse motion

Suppose now that we wanted to write another program that would display the words “Pressed” and “Released” just as the `UpsAndDowns` example but would also draw a line connecting the point where the mouse button was pressed to the point where the mouse button was released. A snapshot of what the window of such a program would look like right after the mouse was pressed, dragged across the screen, and then released is shown in Figure 2.13.

Given that the `UpsAndDowns` program has names that refer to both the point where the mouse button was pressed and the point where the mouse button was released, it might seem quite easy to modify this program to add the desired line drawing feature. In particular, it probably seems that we could simply add a construction of the form:

```
aLine.from( pressPosition )to( releasePoint )on(canvas)
```

to the program’s `onMouseRelease` method.

THIS WILL NOT WORK!

Because `pressPosition` is declared as a formal parameter in `onMousePress`, Grace will not allow the programmer to refer to it in any other method. Grace will treat the use of this name in `onMouseRelease` as an error and refuse to run the program. In general, parameter names can not be used to share information between two different methods.

If we want to share information about the mouse location between two event handling methods, we must use formal parameters and instance variables together. We have already seen that instance variables and definitions can be used to provide information to methods. In the `RisingSun` example, evaluating the object created the oval that was then associated with the identifier `sun`. The method `onMousePress` later moved the oval by sending a method request to the object associated with `sun`.

In order to write a program to draw a line between the points where the mouse was pressed and released, we will have to associate an instance variable name with the location where the mouse was pressed. This variable will then make it possible for `onMousePress` to share the needed information with `onMouseRelease`. We will choose `firstPoint` as the name for this variable.

The completed program will look like the code shown in Figure 2.14.

When the mouse is pressed, the assignment

```
firstPoint := pressPosition ;
```

```

dialect "objectdrawDialect"

// program that displays the words "pressed" and
// "released" when the mouse is pressed or released
// It also draws a line from the pressed to the
// released location.
object {
  inherits aGraphicApplication . size(400,400)

  // Where mouse was pressed
  var firstPoint

  // When the user presses mouse, write "Pressed"
  // on canvas at current mouse location
  method onMousePress(pressPosition)->Done{
    aText.at( pressPosition )with ( "Pressed" ) on (canvas)
    firstPoint := pressPosition
  }

  // When the user releases mouse, write "Released"
  // on canvas at current mouse location
  method onMouseRelease(releasePoint)->Done{
    aText.at( releasePoint )with ( "Released" ) on (canvas)
    aLine.from( firstPoint )to( releasePoint )on(canvas)
  }

  // create window and start graphics
  startGraphics
}

```

Figure 2.14: A Program to track mouse actions

associates the name `firstPoint` with the Location of the mouse. This assignment is interesting in several ways. It is the first assignment we have encountered in which the right side of the equal sign represents something other than the construction of a new object. In the general form of the assignment statement, the expression on the right side of the `:=` can be any phrase that describes the object we wish to associate with the name on the left. So, in this case, rather than creating a new object, we take the existing location object that is named `pressPosition` and give it a second name, `firstPoint`.

Immediately after this assignment, the location that describes the mouse position has two names. This may seem unusual, but it isn't. Most of us can also be identified using multiple names (e.g. your first name, a nickname, or Mr. or Ms. followed by your last name).

This assignment also illustrates the fact that a variable in a Grace program may refer to different things at different times. Suppose when the program starts you click the mouse at the point with coordinates (5,5). As soon as you do this, `onMousePress` is invoked and the name `firstPoint` is associated with a location that represents the point (5,5). If you then drag the mouse across the screen, release the mouse button and then press it again at the point (150,140), `onMousePress` is invoked again and the assignment statement in its body tells Grace to associate `firstPoint` with a location that describes the point (150,140). At this point, Grace forgets that `firstPoint` ever referred



Figure 2.15: Scribbling with a computer's mouse

to (5,5). A variable in Grace may refer to different objects at different times, but at any given time it refers to exactly one thing (or to nothing if no value has yet been assigned to the name).

Exercise 2.5.2 *Why did we declare `firstPoint` to be a variable rather than a definition?*

In fact, the values associated with most instance variables are changed frequently. To illustrate the usefulness of such changes, we can write a simple drawing program.

Complex drawing programs provide many tools for drawing shapes, lines, and curves on the screen. We will write a program to implement the behavior of just one of these tools, the one that allows the user to scribble on the screen with the mouse as if it was a pencil. A sample of the kind of scribbling we have in mind is shown in Figure 2.15. The program should allow the user to trace a line on the screen by depressing the mouse button and then dragging the mouse around the screen with the button depressed. The program should not draw anything if the mouse is moved without depressing the button.

The trick to writing this program is to realize that what appears to be a curved line on a computer screen is really just a lot of straight lines hooked together. In particular, to write this program what we want to do is notice each time the mouse is moved (with the button depressed) and draw a line from the place where the mouse started to its new position. Each time the mouse is moved with its button depressed, Grace will follow the instruction in the `onMouseDown` method. So, within this method, we want to include an instruction like:

```
aLine.from( previousPosition )to( currentPosition )on(canvas)
```

where `previousPosition` and `currentPosition` are names that refer to the previous and current positions of the mouse. The trick is to also include statements that will ensure that Grace associates these names with the correct locations.

Associating the correct location with the name `currentPosition` is easy. When `onMouseDown` is invoked, the computer will automatically associate the current mouse location with whatever name we choose to use as the method's formal parameter name. So, if the header we use when declaring `onMouseDown` looks like:

```
onMouseDown(currentPosition)
```

we can assume that the name `currentPosition` will refer to the location of the mouse when the method is invoked.

Getting the correct `Location` associated with `previousPosition` is a bit trickier. Think carefully for a moment about the beginning of the process of drawing with this program. The user will position the mouse wherever the first line is to be drawn. Then the user will depress the mouse button and begin to drag the mouse. The first line drawn should start at the position where the mouse button was depressed and extend to the position to which the mouse was first dragged. This situation is similar to the problem we faced when we wanted to draw a line between the point where the mouse was depressed and the point where the mouse was released. The position at which the mouse button is first depressed will be available through the formal parameter of the `onMousePress` method but we need to access it in the `onMouseDrag` method because this is the method that will actually draw the line. We can arrange for `onMousePress` to share the needed information with `onMouseDrag` by declaring the name `previousPosition` as an instance variable and including an appropriate assignment in `onMousePress` to associate this name with the position where the mouse button is first pressed.

Given this analysis, we will include an instance variable declaration of the form:

```
var previousPosition
```

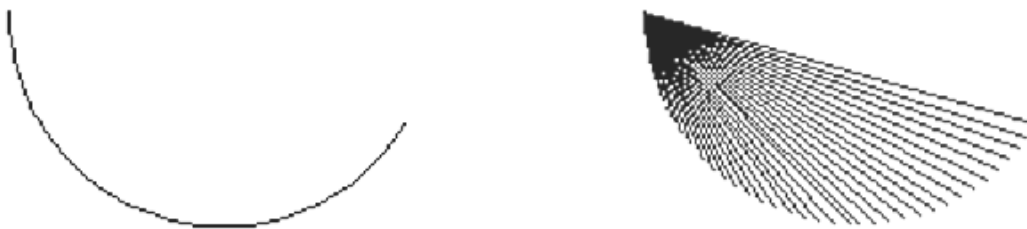
and then write the following definition for `onMousePress`:

```
method onMousePress(pressPosition) {  
    previousPosition := pressPosition ;  
}
```

We also know that the `onMouseDrag` method must contain the line construction shown above and that its formal parameter should be named `currentPosition`. So, a first draft of this method would be:

```
onMouseDrag(currentPosition) {  
    aLine.from( previousPosition )to( currentPosition )on(canvas)  
}
```

Unfortunately, if we were actually to use this code, the program would not behave as we want. For example, if we were to start near the upper left corner of the screen and then drag the mouse in an arc counterclockwise hoping to draw the picture shown below on the left, the program would actually draw the picture shown on the right.



The problem is that we are not changing the point associated with `previousPosition` as often as we should. In `onMousePress`, we tell the computer to make this name refer to the point where the mouse is first pressed and it continues to refer to this point until the mouse is released and pressed again. As a result, as we drag the mouse all the lines created start at the point where the mouse button was first pressed. Instead, after the first line has been drawn, we always want `previousPosition` to refer to the mouse's position when the last line was drawn. To do this, we must add the assignment

```

dialect "objectdrawDialect"

object {
  inherits aGraphicApplication . size (400,400)

  // beginning point for next line
  var previousPosition :Location

  // Choose color randomly for next line and remember start
  method onMousePress(pressPosition) -> Done {
    previousPosition := pressPosition
  }

  // Draw lines to follow mouse
  method onMouseDrag(currentPosition) -> Done{
    aLine.from( previousPosition )to( currentPosition )on(canvas)
    previousPosition := point
  }

  startGraphics
}

```

Figure 2.16: A simple sketching program

statement:

```
previousPosition := currentPosition
```

at the end of the `onMouseDrag` method yielding the complete program shown in Figure 2.16.

Exercise 2.5.3 *Explain why the assignment statement is still needed in `onMousePress` as shown below even though `previousPosition` is always updated in `onMouseDrag`:*

```

method onMousePress(pressPosition) {
  previousPosition := pressPosition ;
}

```

2.6 Summary

In this chapter we explored the importance of the use of names to refer to the objects our programs manipulate. Instance variable and definition names were used to share information between methods, and formal parameter names provided a means to pass information from outside the program into a method body.

Identifiers introduced in definitions need to have a value associated with them in the definition. Once associated with an object, they may not be associated with other objects, though mutator methods requested on the object may change its attributes. Variables have to be assigned a value using `:=` before they are used. They could be assigned to a new object at any time using `:=`. Formal parameters are introduced in method headers. They are associated with values when the method is

executed. For example, when `onMousePress` is executed, the formal parameter is associated with the location of the mouse at that time.

We learned more about displaying simple graphical objects on our program's canvas and learned how to modify the properties of these objects using mutator methods. In addition, we learned that the `objectdraw` library contains classes that can generate objects representing locations and colors. These objects are not themselves visible on our screen, but are used in creating and modifying graphic images on the canvas.

In case you did not notice, the last example program we discussed, the `Scribble` program, was different from most of the other examples in one important regard. It actually is (at least part of) a useful program. This reflects the fact that the features we have explored are fundamental to the construction of all Grace programs. With this background, we are now well prepared to expand our knowledge of the facilities Grace provides.

Chapter 3

Working with Numbers

In the preceding chapter, we used numbers extensively to manipulate graphical objects. They were used to specify coordinates, dimensions, and even colors. While we used numbers to describe what we wanted to do to various graphical objects, we did not do much of interest with the numbers themselves. Numbers are of critical importance to Grace. Just as Grace provides operations to let us work with graphical objects, it provides operations to let us work with numbers. In this chapter we will explore some of these operations. We will see how to obtain numerical values describing properties of existing objects, how to perform basic arithmetic computations, how to work with numeric variables, and how to display numeric values.

3.1 Writing numbers

We've already seen several examples of using numbers in Grace, but there are some rules you should be aware of: integral values can be written without decimal points and may optionally be preceded by a minus sign: 5, 147, -12, 0.

Non-integral values can be written using a decimal point. If a decimal point is used, there must be a digit on either side. Thus 15.3, -1.7, and 0.0005 are fine, but 12. and .7 are not legal in Grace.

FIX! ► *The following is in the spec, but not currently supported in Grace.* ◀ Scientific notation is available to write very large or very small values. Thus 1.65e12 represents $1.65 * 10^{12}$, while 8.02e-5 represents $8.02 * 10^{-5}$.

3.2 Introduction to Accessor Methods

When we create a new location by evaluating a term like `aLocation.at(50, 150)` we use two numbers to create a single location. There are many situations where it is useful to do the opposite. That is, we have a location object and want to access the numerical values of the x and y coordinates associated with that location.

Suppose, for example, that we decided to change the `RisingSun` program so that rather than having to click the mouse to move the sun, the user could simply drag the mouse up and down and the sun would follow it. The desired behavior is similar to the way in which the scrollbars found in many programs react to mouse movement. If you grab the scroll box displayed in a vertical scrollbar you can move the scroll box up and down by moving the mouse, but you can not move the scroll box to the left or right. Similarly, in the program we have in mind, even if the mouse is dragged

about in spirals, the circle that represents the sun should only move straight up and down so that its y coordinate is always the same as the y coordinate of the mouse's current position.

In this new version of `RisingSun`, which we will name `ScrollingSun`, we need to replace the `onMouseClicked` method from the previous version with an `onMouseDown` method of the form:

```
method onMouseDrag(mousePosition ) {  
    sun.moveTo( ... );  
}
```

The question is what should we provide as parameter information to the `moveTo` method.

We want the sun to move to the position in the canvas whose x coordinate is the same as the sun's initial x coordinate, 50, and whose y coordinate is equal to the y coordinate of the mouse. We can handle the x coordinate by simply typing 50 as the first parameter to the `moveTo` method. The hard part is providing the y coordinate of the mouse's position.

The location associated with `mousePosition` certainly contains enough information to determine the y coordinate of the mouse. Grace lets us ask the location for this information through a mechanism called an *accessor method*. Like the mutator methods discussed in the preceding chapter, a small collection of accessor methods is associated with each class of objects. Location objects support two accessor methods, `x` and `y`.

To use an accessor method we write a name that refers to the object that is the target of the request followed by a period, the name of the method to be applied, and, if necessary, a parenthesized list of parameter values. So, to position the sun appropriately in the `onMouseDown` method, we can calculate the new location `newSunLocn` using `mousePosition.y` and then use that to set the new location of sun

```
newSunLocn := aLocation.at(50,mousePosition.y)  
sun.moveTo(newSunLocn)
```

You should observe that the notation for using accessor methods is identical to the notation used for mutator methods. In the case that no parameters are provided, you do not need to include a set of empty parentheses after the name of the method.

The complete text of the revised program is shown in Figure 3.1. With the exception of the substitution of the new `onMouseDown` method for the old `onMouseClicked` method, the only difference between this program and the one shown in Figure 2.3 is that the instructions displayed by the `begin` method have been modified.

Accessor methods are also associated with objects of the graphics classes introduced in the last chapter. The location and dimensions of a graphical object can be accessed using methods named `x`, `y`, `width` and `height`. Like the methods discussed above, these accessor methods provide numeric information about an object. In addition, there are accessor methods associated with graphical objects that provide other forms of information. There is a method named `color` that returns the color of a graphical object. Similarly, `location` returns a location object describing an object's current position.

Exercise 3.2.1 *Is the `translate` method on locations an accessor method? Why or why not?*

Exercise 3.2.2 *What is the difference between an accessor method and a mutator method? What can an accessor method do that a mutator method cannot?*


```

dialect "objectdrawDialect"

object {
  inherits aGraphicApplication . size (200,200)
  // location where the sun starts
  def startSun = aLocation.at(50, 150)

  // location for the instructions
  def startInstructions = aLocation.at(20, 20)

  // Circle representing the sun
  def sun = aFilledOval . at(startSun) size (100, 100) on (canvas)

  var newSunLocn := startSun

  // Display instructions
  aText.at( startInstructions ) with ("Drag the mouse up or down") on (canvas)

  // Drag to raise or lower the sun
  method onMouseDrag(mousePosition){
    newSunLocn := aLocation.at(50,mousePosition.y)
    sun.moveTo(newSunLocn)
  }

  startGraphics
}

```

Figure 3.1: Program to make the sun scroll with the mouse

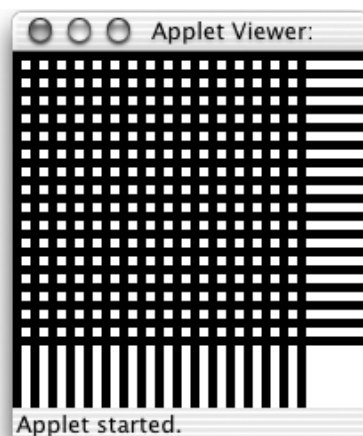


Figure 3.2: Drawing produced by the DrawGrid program

3.3 Accessing the Size of the Canvas

In the last chapter, to keep things simple, we assumed that we knew the size of the windows in which our programs would run. For example, the `DrawGrid` program presented in Section 2.2.2 draws a grid like the one shown in Figure 3.2. The bars in this grid are created by constructions of the form:

```
aFilledRect .at( verticalCorner ) size (5, 200)on(canvas)
aFilledRect .at( horizontalCorner ) size (200, 5)on(canvas)
```

placed within the program's `onMouseClicked` method. The vertical rectangles created by these constructions are 200 pixels tall and the horizontal rectangles are 200 pixels wide. The resulting drawing looks fine if the window is exactly 200 by 200, which is the window size we showed in the figures that illustrated the drawings the program would produce, but they would not look right if the window was larger. If the window was wider, we would want the program to draw wider horizontal rectangles. If the window was taller, we would want taller vertical rectangles.

When we write a program, the size of the window is determined by the parameters in `aGraphicApplication.size (...)`. Thus most of the programs we have written so far specify a window that is 200 by 200 pixels. However, a programmer may change this value at any time. If the rest of the program is not changed, then the graphic images in the window may no longer appear in the correct positions.

There are two ways we can avoid this problem. One is to provide a definition that specifies the width and height of the window. E.g.

```
def windowWidth = 200
def windowHeight = 200
object {
  inherits aGraphicApplication . size (windowWidth,windowHeight)
  ...   windowWidth .... windowHeight
}
```

As long as the programmer is careful to only use the identifiers `windowWidth` and `windowHeight` rather than the literal 200, then the program should continue to work correctly even if `windowWidth` and `windowHeight` were changed.

An alternative way to accomplish this is to write programs that determine the actual size of the canvas while running and adjust the objects they draw accordingly. We have seen that the canvas provides a mutator method named `clear`. It also supports two accessor methods named `width` and `height` which allow a program to determine the dimensions of the drawing area. Like the `x` and `y` methods associated with locations, these methods do not expect any parameter values. To produce a version of `DrawGrid` that would work correctly in any size canvas, we would simply replace the occurrences of the number 200 in the constructions shown above with appropriate uses of accessor methods to obtain the following code:

```
aFilledRect .at( verticalCorner ) size (5, canvas.height) on (canvas)
aFilledRect .at( horizontalCorner ) size (canvas.width, 5) on (canvas)
```

Exercise 3.3.1 Write a program that draws an X through the canvas using the `height` and `width` accessor methods of the canvas.



3.4 Expressions and Statements

It is important to observe that accessor methods serve a very different function than do mutator methods. A mutator method instructs an object to change in some way. An accessor method requests some information about the object's current state.

Simply asking an object for information is rarely worthwhile by itself. We also need to tell Grace what to do with the information requested. We would never write an instruction of the form:

```
mousePosition.y
```

Such an instruction would tell Grace to ask the `Location` named `mousePosition` for its `y` coordinate but make no use of the information. Instead, we use accessor methods in places within a program where Grace expects the programmer to describe an object. For example, an accessor method can be used on the right hand side of an assignment statement to describe the value that should be associated with the name on the left side of the assignment symbol, as in the statement

```
lastY := mousePosition.y
```

or to describe a parameter to a construction as in

```
var newLocn := aLocation.at(10, mousePosition.y)
```

or to describe the parameters for another method, like `moveTo`.

This notion of a phrase that describes an object or value is important enough to deserve a name. Such phrases are called *expressions*. We have already seen several different sorts of phrases that Grace recognizes as examples of expressions.

Where numeric information is needed, we have often explicitly included the numbers to use by typing *numeric literals* like "50" and "150" as in the invocation

```
newLocn := aLocation.at(50,150)
```

In other situations, *accessor method invocations* have been used to describe numeric values as in

```
newLocn := aLocation.at(50,point.y)
```

Where non-numeric information was needed, we have either used a *construction* to create the needed information as in

```
sun.color := aColor.r(200)g(100)b(0)
```

or provided a *name* that was associated with the desired object as in

```
sun.color := purple
```

Thus, numeric literals, instance variable names, constructions, and invocations of accessor methods can all be used as expressions.

In any context where it is necessary to describe an object, Grace will accept any of the forms of

expressions we have introduced. In a context where Grace expects the programmer to describe a color, we can equally well use a name associated with a color, a construction of the form `aColor.r (...)` `g (...)` `b (...)`, or an invocation of the `color` accessor method. Grace is, however, picky about the type of value described by an expression. In a context where Grace expects us to provide a color, we can't provide an expression that describes a number or a location instead.

Not all phrases found in a Grace program are expressions. The invocation of a mutator method such as:

```
sun.moveBy(0,-5)
```

is an example of a phrase that is not an expression. This phrase contains subparts that are expressions, the numeric literals `0` and `-5`, but is not an expression itself because it does not describe a value. Instead of describing a value, this phrase instructs Grace to perform an action. Such phrases are called *instructions* or *statements*. Statements instruct Grace to perform actions that either produce output visible to the user or alter the internal state for the computer in a way that will affect the future behavior of the program. The body of each method we define in a Grace program must be a sequence of statements.

We have seen three types of statements at this point. The invocation of a mutator method, such as

```
sun.moveBy(0,-5)
```

is one type of statement. The second type of statement is the assignment statement. It instructs the computer to perform the action of associating a name with an object or value. Note that the phrase on the right side of an assignment must be an expression.

The third type of statement we have encountered is the construction. We have used instructions like:

```
aText.at(mousePosition) with ( "Pressed" ) on (canvas)
```

to place graphics on the canvas. We have already stated, however, that a construction is an expression. Which is it? The answer is both. A construction like the one shown above describes an object. Therefore it can be used in contexts where expressions are required. At the same time, the construction of a graphical object involves the action of changing the contents of the display. Accordingly, the construction by itself can be viewed as a statement.

There are constructions that merely describe an object without having an associated action that affects any aspect of the state of the program. For example,

```
aLocation.at(10, 20)
```

It does not make much sense to use such a construction as a command, because a program that contained such a command would behave the same if the command were removed. Grace, however, does not prevent the programmer from writing such nonsense. In fact, Grace will allow the programmer to use many kinds of expressions as if they were commands by simply placing them on new lines. In a sensible program, however, the only expressions we have introduced so far that make sense as commands are constructions of graphical objects.

Exercise 3.4.1 Which of the following statements could actually be useful in a program and which could not? Explain.

- a. `sun.color`
- b. `aText.at(point)with("Hello")on(canvas)`
- c. `aColor.r(60)g(60)b(60)`

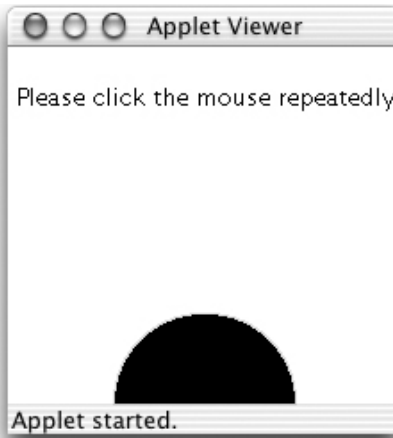


Figure 3.3: The sun rises over the horizon

d. `myLocation := aLocation.at(50, 50)`

3.5 Arithmetic Expressions

Sometimes it is very useful to describe a numeric value to Grace by providing a formula to compute the number. For example, to describe the x coordinate of a point slightly to the left of the current mouse position we might say something like:

`mousePosition.x - 10`

Grace allows the programmer to use such formulae and calls them *arithmetic expressions*. As an example of the use of arithmetic expressions, we can make some additional improvements to our `ScrollingSun` program.

Using arithmetic expressions involving the `width` and `height` methods of the canvas, we can revise the `ScrollingSun` program so that it adjusts the size and position of the circle that represents the sun based on the size of the canvas. To maintain the proportions used in the original program as shown in Figure 3.3:

- the diameter of the circle should be half the width of the canvas,
- the left edge of the circle should fall one quarter of the width of the canvas from the edge of the canvas,
- initially, the top of the circle should be placed so that half the circle is visible above the horizon. To do this, the top of the circle should be one half of its diameter above the bottom of the canvas.

It would also be appropriate to center the text of the instructions horizontally on the canvas. The indentation of the text from the left edge of the canvas should be equal to that on the right side. So, it should be half of the difference between the width of the text and the width of the canvas.

```

dialect "objectdrawDialect"

object {
  inherits aGraphicApplication . size(200,200)
  // location where the sun starts
  def startSun = aLocation.at(canvas.width/4, canvas.height - canvas.width/4)

  // location for the instructions
  def startInstructions = aLocation.at(20, 20)

  // Circle representing the sun
  def sun = aFilledOval.at(startSun) size (canvas.width/2, canvas.width/2) on (canvas)
  sun.color := yellow

  var newSunLocn := startSun

  // Display instructions
  def instructions = aText.at( startInstructions )
    with ("Drag the mouse up or down") on (canvas)

  // Drag to raise or lower the sun
  method onMouseDrag(mousePosition){
    newSunLocn := aLocation.at(canvas.width/4, mousePosition.y)
    sun.moveTo(newSunLocn)
    instructions . isVisible := false
  }

  // Move sun back to original position when mouse exits window
  method onMouseExit(mousePosition){
    sun.moveTo(startSun)
  }

  startGraphics
}

```

Figure 3.4: Program to make the sun scroll with the mouse

Each of these verbal descriptions can be turned into a formula, which can then be used in the program. The diameter of the circle, which is the value that should be specified as the width and height in the `FilledOval` construction, would be described as

`canvas.width/2`

The x coordinate value for the left edge of the circle would be given by the formula

`canvas.width/4`

The y coordinate for the top of the circle would be described as

`canvas.height - canvas.width/4`

Finally, the x coordinate for the left edge of the instructions should be

`(canvas.width - instructions . width) / 2`

The complete `ScrollingSun` program using such formulae is shown in Figure 3.4. In most cases, we have simply replaced a number used as an expression by the appropriate formula. The only

slight complication is the code to center the text. We can not use the `width` method associated with the text object until it has been constructed because, unlike with other graphical objects, we do not specify its width when we construct it. So, when we construct the text we just use 0 as its x coordinate value. Then, once it exists we use the `width` method to figure out how big it is. Finally, we use `moveTo` to place the text where it belongs.

The arithmetic expressions shown in the preceding examples use only two of the arithmetic operators, subtraction and division. It is also possible to use multiplication and addition. The symbols used to indicate addition, subtraction and division are the standard symbols from mathematics: `+`, `-`, and `/`. Multiplication is represented using an asterisk, `*`. Thus, to say “2 times the width of the canvas” one would write `2 * canvas.width`

The values being operated upon are called *operands*. In the example above of multiplication, the operands are `2` and `canvas.width`.

Other fairly common arithmetic operations are exponentiation, written `^`, and modulus, written `%`. Exponentiation raises numbers to a power. For example `3 ^ 2` evaluates to 9. The modulus operator finds the remainder when the first number is divided by the second. For example, `14%3` is 2. If you performed a long division of 14 by 3, the quotient would be 4, with a remainder of 2. Unlike many programming languages, Grace does not have an operator that gives the integer quotient. Instead `14/3` is 4.666667.

The following table summarizes the most commonly used arithmetic operators in Grace.

<code>+</code>	addition
<code>-</code>	subtraction
<code>*</code>	multiplication
<code>/</code>	division
<code>^</code>	exponentiation
<code>%</code>	modulus

Exercise 3.5.1 Write the `begin` method for a program that draws two lines that form a cross (consisting of a horizontal line and a vertical line as shown in Figure 3.5) on the canvas with the intersection of the two lines in the center of the canvas. The program should draw the cross correctly no matter what the size of the canvas.

3.5.1 Ordering of Arithmetic Operations

Two of the arithmetic expressions used in the `ScrollingSun` program shown in Figure 3.4 illustrate an issue a programmer must be aware of when writing such expressions: the rules used to determine the order in which operations are performed. The first of these is the expression

```
(canvas.width - instructions.width)/2
```

which is used to position the instructions. The second determines the initial y coordinate for the top of the sun:

```
canvas.height - canvas.width/4
```

Both involve a subtraction and a division. The first, however, uses parentheses to make it clear that the subtraction should be performed first and that the result of the subtraction should be divided by 2. The correct interpretation of the second expression is not as clear. In fact, Grace will first divide the width of the canvas by 4 and then subtract the result of this division from the height of

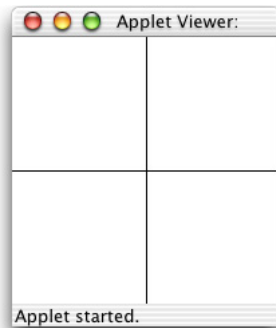


Figure 3.5: Drawing desired for Exercise 3.5.1

the canvas. In the absence of parentheses that dictate otherwise, Grace always performs divisions in an expression before subtractions. Thus, the second formula is equivalent to the formula:

`canvas.height - (canvas.width/4)`

The rule that division is performed before subtraction is an example of a *precedence rule*. When evaluating simple arithmetic expressions, Grace follows two basic precedence rules:

- Perform divisions and multiplications before additions and subtractions. We therefore say that division has higher precedence than addition but that division and multiplication are of equal precedence.
- When performing operations of equal precedence (i.e. additions and subtractions or divisions and multiplications) perform the operations in order from left to right as written.

Parentheses can be used to override these precedence rules as seen in the first example above. Any part of a formula enclosed in parentheses will be evaluated before its result can be used to perform operations outside the parentheses.

The operators \wedge and $\%$ are not assigned a precedence in Grace, so if they are in an expression with other operators then parentheses must be included to determine the order of evaluation.

Exercise 3.5.2 *What are the values of the following expressions?*

a. $4 + 3 * 8 / 2 - 3$

b. $(4 + 3) * 8 / 2 - 3$

c. $4 + (3 * 8) / 2 - 3$

d. $4 + 3 * 8 / (2 - 3)$

e. $(4 + 3) * 8 / (2 - 3)$

f. $(4 + 3 * 8) / 2 - 3$

3.6 Numeric Instance Variables

In the previous chapter, we saw that it is sometimes necessary to associate instance variable names with `Locations` or other objects to enable one method to communicate information to commands in another method. Unsurprisingly, it is often useful to associate names with numeric values in a similar way. We can illustrate this by adding yet another feature to our `RisingSun` program.

As the real sun rises, the sky becomes brighter and brighter. Suppose we wanted to try to simulate this in our program. For this example we will return to the original interface where the user clicks repeatedly to make the sun rise. Now, when the sun is near the bottom of the screen, we would like the background to be filled with a dark shade of gray. We can do this by constructing a `FilledRect` as big as the canvas and setting its color to an appropriate shade of gray. As the user clicks, we can make the background become lighter by using `setColor` to replace the original shade of gray with lighter and lighter shades until it is eventually white.

We have seen that each color is described by a triple of numbers giving the amounts of red, green, and blue in the color. Shades of gray correspond to triples in which all three values are the same. The bigger the number used, the brighter the shade. So,

```
aColor.r(0)g(0)b(0)
```

describes black,

```
aColor.r(50)g(50)b(50)
```

describes a dark shade of gray,

```
aColor.r(200)g(200)b(200)
```

would be a fairly light shade of gray and

```
aColor.r(255)g(255)b(255)
```

is white.

To control the brightness of the background, we would like to associate an instance variable name with the number to be used to generate the shade of gray currently desired. We will use the name `brightness` for this variable. This name can then be used to construct shades of gray for the background by using the construction:

```
aColor.r(brightness)g(brightness)b(brightness)
```

To use such a name, of course, we must first declare the name and then add assignment statements to ensure that it is associated with the correct number at each time as the program executes.

In the declaration, we will associate the name `brightness` with a number corresponding to a dark shade of gray by including an assignment statement of the form:

```
var brightness := 50
```

Each time the user clicks the mouse, we want to associate a larger number with `brightness`. We can do this by including the assignment statement

```
brightness := brightness + 3
```

in `onMouseClicked`. This statement tells the computer to take the current value associated with the name `brightness`, add one to it, and then associate the name `brightness` with the result. The first time the mouse is clicked, `brightness` will be associated with the value 50 specified in the declaration. The result of adding 3 to 50 is 53. So, after the assignment statement is executed, `brightness` will be associated with the value 53. The next time the mouse is clicked, Java will add 3 to the new value of `brightness`, 53, and set it equal to 56. Thus, each time the mouse is clicked, the value of `brightness` will become one greater and the color generated by the construction

```
aColor.r(brightness)g(brightness)b(brightness)
```

will become a little bit brighter.

With these details we can complete the program. The code is shown in Figure 3.6.

```
dialect "objectdrawDialect"

object {
  inherits aGraphicApplication.size(200,200)

  def origin = aLocation.at(0,0)

  // sky created with brightness
  def sky = aFilledRect.at(origin) size (canvas.width, canvas.height) on (canvas)
  var brightness := 50
  sky.color := aColor.r(brightness)g(brightness)b(brightness)

  // location where the sun starts
  def startSun = aLocation.at(canvas.width/4, canvas.height - canvas.width/4)

  def sun = aFilledOval.at(startSun) size (100, 100)on(canvas)

  // location for the instructions
  def startInstructions = aLocation.at(20, 20)

  aText.at(startInstructions)with("Please click the mouse repeatedly")on(canvas)

  // Each time mouse is clicked, sun rises more and sky brightens
  method onMouseClick(point){
    sun.moveBy(0,-5)
    brightness := brightness + 3
    sky.color := aColor.r(brightness)g(brightness)b(brightness)
  }

  startGraphics
}
```

Figure 3.6: Using a numeric instance variable

Exercise 3.6.1 *What would you expect to happen in `LightenUp` if you changed the assignment of `brightness` to `brightness := 200` and then changed the first line in the `onMouseClicked` method to the following?*

`brightness := brightness - 3`

Exercise 3.6.2 *What would you expect to happen in `LightenUp` What happens if you keep clicking the mouse for a long time? What might go wrong? Run the program and see what happens.*

Exercise 3.6.3 *Suppose instead of a gray sky in the example, we wanted the sky to start black, but become a lighter blue each time the mouse is clicked. How would you change the code to make this happen.*

3.7 Displaying Numeric Information

We have seen how we can use the computer's ability to work with numbers to produce better drawings on the computer's screen. Sometimes, however, it is the numbers themselves rather than any drawing that we really want to see. The main purpose of many computer programs is to perform numerical calculations. Examples include programs that determine your taxes, determine your GPA, and estimate the time required to travel from one point to another. Such programs often simply display the numbers they compute rather than a drawing a graph of some sort on the screen. Even programs that are not primarily focused on numerical computations often need to display numerical information. For example, a word processor might need to display the current page number. In this section we will describe two mechanisms in Grace that can be used to display numerical information.

3.7.1 Displaying Numbers as text

As a very simple first example, let's make the computer count. You probably don't remember it, but at some point in your early childhood you most likely impressed some adult by demonstrating your remarkable ability to count to 10 or 20 or maybe even higher. To enable the computer to produce an equally impressive demonstration of its counting abilities, we will construct a program that will count. It will start at 0 and move on to the next number each time the mouse is clicked. The current value will be displayed on the computer's screen.

We have already introduced one mechanism that can be used to display numbers on the screen. We just didn't mention that it could be used with numbers at the time. In the very first program in Chapter 1, we used a construction of the form:

`aText.at(clickTextLocation)` with ("[Click in this window](#)") on (canvas)

and explained that the `aText` construction requires three parameters:

- the location on the canvas where the information should be displayed, and
- the information to be displayed,
- the canvas.

In all the examples of text constructions we have seen thus far, the second parameter has been a sequence of characters surrounded by quotes. In fact, the second parameter to `aText.at()with()on()` can be any expression that returns a string.

Any object can return a string from an `asString` method request. We will ensure that each new object we define has an `asString` method that returns a string that describes the object. If we do not define `asString` ourselves, it will instead return a default string value that may not be very helpful.

In particular, if we define a variable

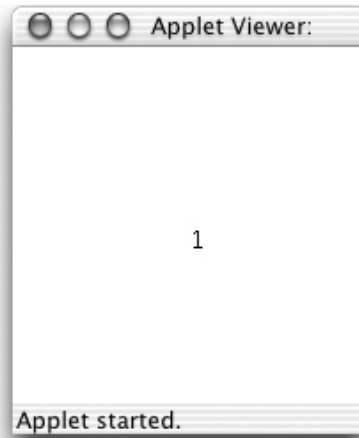


Figure 3.7: A computer counting program before the first click

```
var theCount := 0;
```

with the intent of using it to count up from one, and then we execute a construction of the form:

```
aText.at(aLocation.at(100,100))with(theCount.asString)on(canvas)
```

Grace will display the current value of the variable `theCount`, 0, at the point (100,100) in the program's window as shown in Figure 3.7.

Of course, displaying the number 0 isn't counting. The program we want to construct should start by displaying 0, but the first time the user clicks the mouse, we want to replace 0 by 1. On the next click, we want to replace 1 by 2 and so on.

In case you didn't notice, one of the examples considered in this chapter already demonstrates how to teach a computer to count. In the version of the rising sun program in which the background became brighter as the sun rose, the operations we performed on the variable named `brightness` essentially told the computer to count upwards starting at 50. The instruction that we used to progress through the different values of `brightness` was

```
brightness := brightness + 3
```

A very similar assignment statement involving the variable `theCount`:

```
theCount := theCount + 1;
```

is what we need to complete the counting program described above.

Such a counting program is shown in Figure 3.8. The body of the `onMouseClicked` method uses the assignment statement shown above to associate the next counting number with `theCount` each time the mouse is clicked. Note that changing the value associated with the variable `theCount` does not cause the value displayed by the text object named `countDisplay` to change. To change the text displayed, we update the `contents` of the item. This changes the information displayed by a text object. Like the second parameter expected in a text construction, this information can be a quoted sequence of characters or any other expression that returns a string. The statement

```
countDisplay.contents := theCount.asString
```

in the `onMouseClicked` method tells Grace to change the information displayed by the text object named `countDisplay` that was created earlier to the string corresponding to the current value of `theCount`.

An alternative to resetting the `contents` would be to clear the canvas and then construct a new

```

// A program to count as high as you can click
dialect "objectdrawDialect"

object {
  inherits aGraphicApplication . size(200,200)

  var theCount := 0

  def countPosn = aLocation.at(100,100)
  def countDisplay = aText.at(countPosn)with(theCount.asString)on(canvas)

  method onMouseClick(point){
    theCount := theCount+1
    countDisplay.contents := theCount.asString
  }

  startGraphics
}

```

Figure 3.8: A simple counting program.

text object displaying the new value of `theCount`. Construction of a new object, however, is a fairly time consuming process. When possible, it is better to reuse an existing object rather than create a new one. Accordingly, in an example like this it is preferable to update `contents`.

There are several other mutator methods that can be applied to text objects. In Chapter 1 we already showed that the attribute `isVisible` could be reset with text objects. In addition, the `moveBy` and `moveTo` methods can be used to reposition text objects, just as they can be used with rectangles and ovals. Finally, there are several special mutator methods for use with text objects. For example, the methods `fontSize` and `fontSize :=` methods can be used to access or change the size of displayed text. For example,

```
countDisplay.fontSize := 24
```

could be added after the definition `countDisplay` if we wanted to increase the size of the numbers displayed.

3.7.2 Using `print`

In a program that mixes graphical output with numerical or other textual information, text objects and the `contents :=` method are the most appropriate tools for displaying textual information. In programs that only display textual information, there is another tool that is often simpler and more appropriate, `print`.

All the output displayed by the Grace programs we have considered so far appears in the pop-up window associated with the name `canvas`. These programs can, however, display output in another window provided by the Grace system. There is no special name that can be used to refer to this

```

dialect "objectdrawDialect"

object {
  inherits aGraphicApplication . size(200,200)

  var theCount := 0
  def instructionPosn = aLocation.at(20,100)
  def instructions = aText.at(instructionPosn)with("Click to make me count")on(canvas)

  method onMouseClick(point){
    theCount := theCount+1
    print(theCount)
  }

  startGraphics
}

```

Figure 3.9: Counting in the Grace console

window from within your program. It is simply known as the *output* or *console window*, and is found in the area to the right of the program text.

The console window is more limited than the canvas in that it can only be used for text. On the other hand, it is more convenient for the display of text than is the canvas.

To display information in the console you use a method named `print`. This method takes a single parameter specifying the information to be displayed. Any expression that evaluates to a string can be used as a parameter to `print`. In particular, you can certainly use a string surrounded by double quotes or the results of sending an `asString` method request to a number (or any other object, for that matter).

In fact, `print` is a special method in that if you evaluate it with an argument, and that argument is not a string, it applies the `asString` method request before printing the result. As a result,

```
print(countDisplay.asString)
```

behaves exactly the same as

```
print(countDisplay)
```

A revised version of our counting program that uses the Grace console to display values as it counts is shown in Figure 3.9. The only text this version displays on the canvas is a message telling the user to click in order to make the program count. This will be displayed instead of the number 0 when the program first starts. The first time the user clicks, 1 is placed in the Grace console by the `print` in the `onMouseClick` method. Each succeeding click will place another value in the console window.

When `print` is used, you do not have to provide coordinates to specify where the text should be displayed. The Grace console window displays the information you provide to these methods much as text might be displayed in a word processor's window. Each time your program executes a `print`, the text specified is placed on a new line in the Grace console window, below any text placed there by previous uses of `print`.

Once the console window fills up, the older text scrolls off the top of the window leaving the newer lines visible. A scroll bar is provided so that a person running your program can look at the older items if desired. Figure 3.10 shows how both the canvas and the Grace console window might

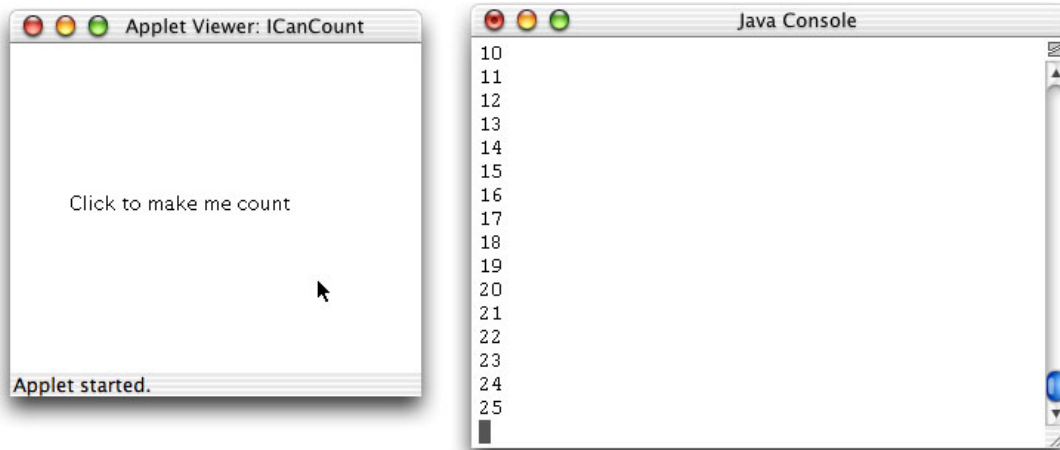


Figure 3.10: Counting using the Grace console

look after this program is run and its user clicks 25 times.

3.7.3 Mixing Text and Numbers

Often, a number displayed all by itself has little meaning. The difference between just displaying “3” and displaying “Strike 3” or “3 p.m.” or “Line 3” can be quite significant. Accordingly, in many programs rather than just displaying a number on the screen it is desirable to display a number combined with additional text that clarifies its meaning. Luckily, this can be done easily in Grace with both text objects and `print`.

When specifying the information to be displayed in a text object or on the Grace console, we can combine quoted text with numeric information by enclosing the numeric information in curly braces: “{...}”. Suppose, for example, that we wanted our counting programs to display a message like “You have clicked 3 times” instead of just displaying 3 on the third click. For the version that uses text objects to place the information on the screen, we could accomplish this by replacing the command

```
countDisplay.contents := theCount.asString
```

shown in Figure 3.8 with the command

```
countDisplay.contents := "You have clicked {theCount} times"
```

When the string is evaluated, `theCount` is evaluated, converted to a string (implicitly using `asString`) and that string is inserted into the surrounding string.

Similarly, for the `print` version shown in Figure 3.9 we could replace the command

```
print ( theCount )
```

with the command

```
print ( "You have clicked {theCount} times");
```

In fact, you can place any expression at all in the curly braces inside a string. When the string is evaluated, the expression in curly braces is evaluated, converted to a string using `asString`, and then inserted into the string.

When Grace sticks together bits of text, it doesn’t think about things like words. It just sticks the letters and digits it is given together. This means you have to include all the characters you

want displayed, including any spaces. If you look carefully at the `print` command shown above, you will notice that there is a space before the open brace “” and before the close brace “”. If these were not included, Grace would display the text

```
You have clicked3times
```

instead of displaying

```
You have clicked 3 times
```

as desired.

Exercise 3.7.1 *What output would you expect from the following statements? Assume `count` is 34.*

- a. `print("The count is: count - 3")`
- b. `print("The count is: {count} - 3")`
- c. `print("The count is: {count -3}")`

Exercise 3.7.2 *Suppose you are trying to write a simple program to help someone practice the multiplication tables. The program repeatedly displays a message of the form:*

```
What is 5 x 9?
```

in a `Text` object named `question` that is created in the program’s `begin` method. The program uses two variables named `factor1` and `factor2` that are assigned randomly generated numbers between 0 and 9 to determine which question to display. For example, the question shown above would be displayed if `factor1` equaled 5 and `factor2` equaled 9.

Write the statement needed to update the message displayed by `question` after new values for `factor1` and `factor2` are chosen. (Don’t worry about how the user tells the program the correct answer.)

3.8 Random numbers

“Pick a number. Any number. ...”

You might expect to hear this phrase from the hawker at a carnival game table. You might not expect it to be a useful instruction to give a computer within a Grace program, but just the opposite is true. There are many programming contexts in which it is handy to be able to ask the computer to pick a random number for you. Obvious examples are game programs. Programs that deal cards, simulate the tossing of dice, or simulate the spinning of a roulette wheel all need ways of picking items randomly. In addition to game programs, there are many programs that simulate the behavior of real systems for practical purposes that need ways to incorporate the randomness of the real world in their calculations. With this in mind, Grace and most other programming systems include what are called random number generators.

The method `randomIntFrom(m)to(n)` returns a randomly selected integer between `m` and `n`, inclusive. Thus evaluating `randomIntFrom(7)to(12)` will return a randomly chosen integer from 7 to 12, inclusive. That is, it will return any of 7, 8, 9, 10, 11, or 12, with each occurring equally likely.

Suppose that you wanted to write a program to simulate a board game in which at each turn the player rolls two dice. Our method for generating random integers can be used to generate numbers

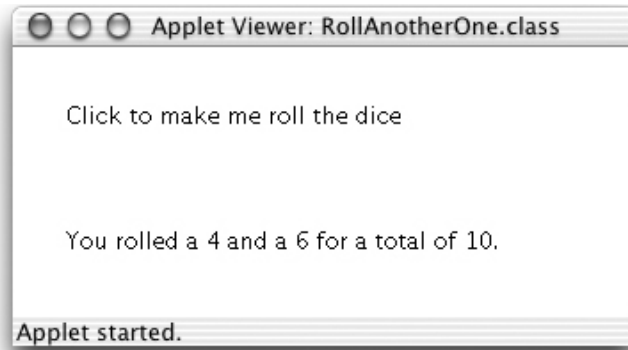


Figure 3.11: Sample message drawn by dice simulation program

just like a single die.¹ To illustrate the use of this, we will construct a simple program that simulates the rolling of a pair of dice each time the mouse is clicked.

When the user clicks the mouse, we need to calculate the new values of the dice. We will provide two variables, `die1` and `die2`, which will each be assigned a value generated by our expression above that supplies random integers between 1 and 6.

The complete code of a simple program to simulate rolling two dice is shown in Figure 3.12. A sample of the program's output is shown in Figure 3.11.

Let's talk through the code to make sure you understand why each piece is there. As usual, we are working in the `objectdraw` dialect. In order to show all of each text item we set the window with the program output to be 300 pixels wide rather than the 200 we have used before. The height remains at 200 pixels.

We then define the two location `promptLocation` and `resultLocation` where we will be displaying text. We will generally place definitions like this early in our programs. Putting them up top makes it easy to find and change them if we don't like the way the output appears on the canvas. For similar reasons, we introduce the definition of `numSides` to name the number of sides on each die. If we were to change to a different size die (e.g., with 12 sides), a change in this one place in the program would automatically fix the rest of the code.

Now we create a text object to display the instructions to the user. We do not bother to give it a name as we will never need to refer to it again in the program. However, we do give the name `result` to a text object whose contents are initially an empty string, i.e., a string of length 0. We will update its contents in the `onMouseClicked` method to give us information about the new values of the dice.

We also declare the variables `die1` and `die2`, but don't yet assign them a value as we will not roll the dice until the user clicks the mouse.

The `onMouseClicked` method evaluates our complex arithmetic expression twice to get two new random numbers, assigning them to our two variables. We then update `result` to show the values of the dice. Recall that the expressions within “`{ ... }`” will be evaluated and then have their string equivalents inserted into the surrounding string.

¹The English word for the little cubes you roll while playing many board games has an irregular plural form. If you have several of these cubes, you call them dice. If you have just one then it is a die.

```

dialect "objectdrawDialect"

object {
  inherits aGraphicApplication . size (300,200)

  // positions of text to be displayed
  def promptLocation = aLocation.at(30,30)
  def resultLocation = aLocation.at(30,100)

  // How many sides our dice have
  def numSides = 6;

  // A Text message giving instructions to the player
  aText.at(promptLocation)with("Click to make me roll the dice")on(canvas)

  // Display a prompt and create the Text used to display the results
  def result = aText.at(resultLocation) with ("") on (canvas)

  // The values of the two dice after a roll
  var roll1
  var roll2

  // Roll the dice with each click
  method onMouseClick(point) {
    roll1 := randomIntFrom(1)to(numSides)
    roll2 := randomIntFrom(1)to(numSides)
    result .contents :=
      "You rolled a {roll1} and a {roll2} for a total of { roll1 + roll2 }"
  }

  startGraphics
}

```

Figure 3.12: Simulating the rolling of a pair of dice

Exercise 3.8.1 *Write a program that draws a rectangle named `box` at the location given by (50, 50). The box should be 50 pixels wide and the height of the box should be determined by an expression involving random numbers that generates values for the height between 10 and 100 pixels. The height of `box` should change each time the mouse is clicked.*

3.9 Summary

Early applications of computers involved scientific and engineering problems that required large amounts of numerical computation. The first computers clearly earned the name “computer”. When using a word processor, reading your email, or browsing the web, it is easy to forget that computers perform arithmetic computations. Even in programs that do not appear to involve numbers, however, numerical computations continue to play a role. For example, a word processor has to do arithmetic just to determine how many words will fit on a line.

In this chapter, we have explored a few of the mechanisms Grace provides to perform numerical computations. We explored the differences between statements and expressions in Grace programs. Statements are phrases in a program that instruct the computer to perform an action that will change the visible state of the computer or change the value associated with some variable name. Expressions are phrases that describe a value or object to be used in the program. We have seen that Grace recognizes several forms of expressions: literals like “12”, variable names, constructions, invocations of accessor methods, and arithmetic formulae involving the operations of addition, subtraction, multiplication, division, exponentiation, and modulo. We also saw how to use a method returning random numbers to generate numbers in a given range.

We showed how to instruct Grace to produce textual output that included numerical information using both text objects and the `print` method. Text objects are used when such information is to be included in the display of a program that also produces graphical output. The `print` method provides a simpler mechanism for producing text output that is appropriate for programs that only produce textual output in the Grace console.

Chapter 4

Making Choices

To write interesting programs we must have a way to make choices in Grace. For example, we might need to perform different instructions depending on whether a user has clicked inside a particular rectangle. In this chapter we show how to make choices in Grace using the `if` statement.

Conditional statements like the `if` statement are programming constructs capable of choosing between blocks of code to execute. These statements provide enormous expressive power to programmers, and yet are very easy to use and understand because they mimic the way we think. For instance, the following form a conditional statement in English:

“*If* it’s sunny outside *then* we will play frisbee, *otherwise* we will play cards.”

An example of a conditional statement that is a bit more relevant to our concerns with programming might be:

“*If* the mouse location is contained in the rectangle *then* display message “success”.
Otherwise display message “missed”.”

In Grace, the `if` statement is the most commonly used conditional statement. It comes in a variety of forms that we will explore, each of which is useful for certain situations. With the help of conditionals we will write programs to determine a win or loss in the game of craps and to figure out what to do on a weekend based on the weather and your finances.

After presenting a brief example illustrating the use of the `if` statement, we formally introduce several of its variations that will allow us to handle more complex situations. We also introduce the boolean data type for expressions that can be either *true* or *false*. Finally, we provide advice on how to use conditionals clearly and effectively so that your programs can be understood correctly by the Grace compiler and, more importantly, by other programmers.

4.1 A Brief Example: Using the `if` Statement to Count Votes

We illustrate the use of conditionals with a simple example of a program to count votes in an election. To accomplish this we will divide the program’s canvas in half vertically so that the left and right sides represent candidates A and B respectively (see Figure 4.1). A mouse click on either side will be treated as a vote for that half’s candidate.

Recall the program `ICanCount` in Figure 3.8, which keeps track of the number of times the user has clicked the mouse. The code for its `onMouseClicked` method is given below:

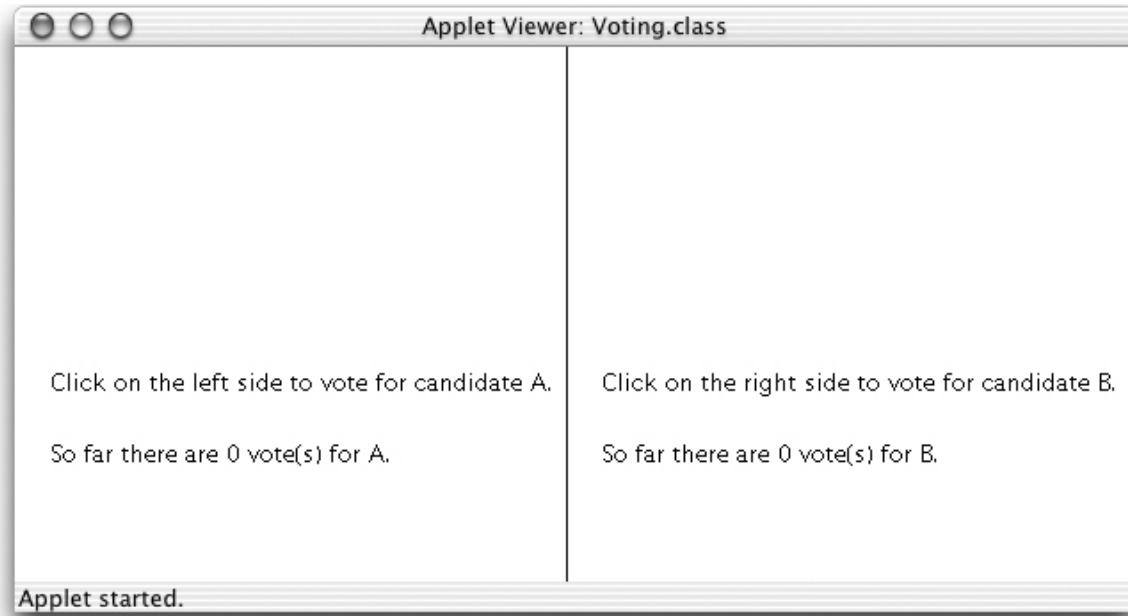


Figure 4.1: Screen shot of Voter program.

```

method onMouseClick(point){
    theCount := theCount+1
    print (theCount)
}

```

Whereas this method simply counts *all* mouse clicks on the canvas, by using the `if` statement our voting program's `onMouseClick` method will be able to discriminate between votes for candidate A and B.

The code for program `Voting` is given in Figure 4.2. The constructs used should be familiar by now. The program starts by defining some `x` and `y` coordinates and some locations for the text objects to be created. An advantage to using definitions to provide names for coordinates is that if they need adjusting, it is easy to modify them in one place while the results show up everywhere.

The program creates four new text objects, the top two of which display voting instructions while the bottom two display the current tally for each candidate. The last line of the before `onMouseClick` draws the vertical line dividing the canvas.

The `onMouseClick` method, on the other hand, contains a new programming construct. The `if` statement is used to determine which candidate gets each vote and then updates the appropriate Text object to display the new total. To decide who receives each vote, the program compares the `x` coordinate of the location of the mouse click to that of the middle of the canvas. Note that `canvas.width/2` refers to the `x` coordinate of the middle of the canvas.

The `if` statement allows the programmer to make choices about which statements are executed in a program based on a *condition*, an expression whose value is either true or false. In the sample program, when the mouse is clicked the program must determine whether to give a vote to candidate A or B. The condition is whether the `x` coordinate of the mouse click, obtained by evaluating `point.x`, is less the `x` coordinate of the middle of the canvas. The condition is written in Grace as

```

dialect "objectdrawDialect"

object {
  inherits aGraphicApplication . size (600,200)

  def DISPLAY_A_X = 5 // x-coordinate of location of A's votes
  def DISPLAY_B_X = canvas.width/2 + DISPLAY_A_X // x-coordinate of location of B's votes

  def DISPLAY_Y_TOP = canvas.height/2 - 20 // y coordinates of instructions
  def DISPLAY_Y_BOT = canvas.height/2 + 20 // y coordinates of vote counts

  // locations for instructions and vote counts
  def instructionsALocn = aLocation.at(DISPLAY_A_X, DISPLAY_Y_TOP)
  def countALocn = aLocation.at(DISPLAY_A_X, DISPLAY_Y_BOT)
  def instructionsBLocn = aLocation.at(DISPLAY_B_X, DISPLAY_Y_TOP)
  def countBLocn = aLocation.at(DISPLAY_B_X, DISPLAY_Y_BOT)

  // locations of top and bottom of separating line
  def topLine = aLocation.at(canvas.width/2,0)
  def bottomLine = aLocation.at(canvas.width/2,canvas.height)

  var countA := 0 // current counts of votes for A and B
  var countB := 0

  aText.at(instructionsALocn) with ("Click on the left side to vote for candidate A.")
    on (canvas)
  aText.at(instructionsBLocn) with ("Click on the right side to vote for candidate B.")
    on (canvas)

  // Display number of votes for A and B
  def countAText = aText.at(countALocn) with ("So far there are {countA} votes for A.")
    on (canvas)
  def countBText = aText.at(countBLocn) with ("So far there are {countB} votes for B.")
    on (canvas)

  // Line separating voting spaces
  aLine.from(topLine) to (bottomLine) on (canvas)

  // Update votes and display vote counts
  method onMouseClick(point) {
    if (point.x < (canvas.width/2)) then {
      countA := countA + 1
      countAText.contents := "So far there are {countA} votes for A."
    } else {
      countB := countB + 1
      countBText.contents := "So far there are {countB} votes for B."
    }
  }
}

startGraphics
}

```

Figure 4.2: Code for Voter program.

`point.x < canvas.width/2`. If the condition is *not* satisfied (*i.e.*, if `point.x` is greater than or equal to `canvas.width/2`) then the code after the `else` keyword is executed.

The two lines of code following the line containing the `if` are grouped together by a pair of matching curly braces. A sequence of statements surrounded by curly braces in this manner is called a *block*. Similarly, the two statements immediately following the `else` also form a block. If the *condition* of an `if` statement is true, the block of statements immediately after the condition is executed. Otherwise, the block immediately after the `else` is executed.

Exercise 4.1.1 *What are the conditions in the following statements? For example, in the statement, “When it is cold outside, I wear a hat,” the condition is “it is cold outside”.*

- a. *When it is Tuesday, I have piano lessons.*
- b. *Since today is Halloween, it must be October.*
- c. *If Bobby says yes, we will go to the prom.*
- d. *Yesterday’s game determined the wild card, so if the Red Sox won, they made it to the playoffs.*

4.2 The `if` Statement

Now that we have seen the `if` statement in action, let’s carefully examine its syntax and meaning.

The code in the Voting example given in Figure 4.2 contains a form of conditional statement called the `if–else` statement. Its syntax is:

```
if ( condition ) {  
    if–part      // statements to be executed when condition is true  
} else {  
    else–part   // statements to be executed when condition is false  
}
```

The text `condition` in the syntax template represents an expression whose value is true or false. The phrases `if–part` and `else–part` represent sequences of Grace statements. We’ve included comments to make it clear when each of these sequences of statements is executed, even though these comments are not part of the formal syntax.

When an `if–else` statement is executed, the computer first determines whether *condition* is true. If so, it executes the statements in the block of code surrounded by the first pair of curly braces, “{” and “}”, called the *if–part*, and then skips over the rest of the statement. Otherwise (*i.e.*, if *condition* is false), it skips over the `if–part` and will instead execute the block of statements after the `else` keyword, called the *else–part*. Exactly one of the two blocks of code is processed when the `if–else` statement is executed. When that block is completed, execution resumes immediately after the `if–else` statement. This execution sequence is illustrated in Figure 4.3.

The following example shows how execution resumes with the statements that follow an `if–else` statement. Suppose we want to modify our Voting program so that it always displays the total number of votes. Let `totalText` be defined as a text object earlier in the program. Figure 4.4 shows a revised version of `onMouseClicked` that displays the current vote total. Each time the user clicks on the canvas, the line updating the contents of `infoTotal` will be executed regardless of which half of the screen the mouse was clicked on.

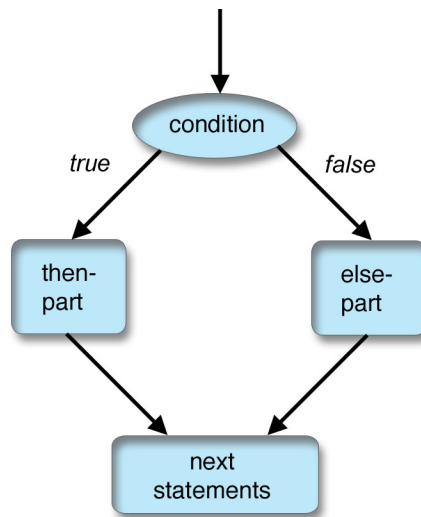


Figure 4.3: Semantics of the if–else statement.

```

// Update votes and display vote counts
method onMouseClick(point) {
  if (point.x < (canvas.width/2)) then {
    countA := countA + 1
    countAText.contents := "So far there are {countA} votes for A."
  } else {
    countB := countB + 1
    countBText.contents := "So far there are {countB} votes for B."
  }
  totalText.contents := "Votes so far: {countA + countB}"
}

```

Figure 4.4: Code to display individual and total vote counts

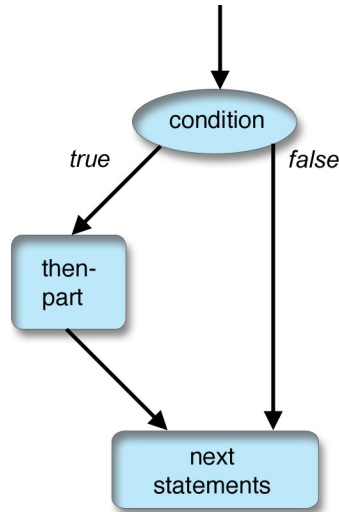


Figure 4.5: Semantics of the if with no else.

There are many situations in which we don't need an else-part. Fortunately, there is a simple variant of the if-else statement, the if statement, which does not have the else keyword or the else-part.

```

if ( condition ) {
    if-part    // statements to be executed when condition is true
}
  
```

If condition is true, then the if-part is executed as before. If it is false, however, the program simply moves on to the *next statement* after the if-part because there is no else-part to execute. The execution sequence is illustrated in Figure 4.5.

Exercise 4.2.1 Write the following statements as if-else or if statements. For example, the statement, “On Mondays I ride my bike to class. On other days, I walk.” can be rewritten as “If it is Monday, I ride my bike to class. Else, I walk.”

- a. On Sunday I eat pancakes. On other days, I eat cereal.
- b. I go to class on weekdays, but on weekends, I watch movies.
- c. In the summer I always wear sandals, but during the other seasons, I wear sneakers.
- d. If it is raining, I bring an umbrella.

4.2.1 Example: Using the if Statement with 2-D Objects

Suppose we want to write a program that begins by displaying a square on the canvas. If the user clicks inside the square then the computer moves the square 50 pixels to the right. If the click is not inside the square, nothing happens.

How can we determine if a point is inside a square? We could compare the coordinates of the point to the locations of the left, right, top, and bottom edges of the square. However, such tests

are so common that all of the two dimensional geometric objects (*e.g.*, filled and framed rectangles and ovals) provide the accessor method `contains` that does this for us.

For example, let `square` be a variable referring to a framed rectangle, and let `point` refer to a location. Then the expression `square.contains(point)` evaluates to *true* if the object held in `square` contains the point, and *false* otherwise.

For the sake of brevity we will only write out the `onClick` method of this program. We assume that `square` and `xOffset` have been declared and initialized elsewhere.

```
method onClick( point ) {
    if ( square.contains( point ) ) {
        square.move( xOffset, 0 )
    }
}
```

An English translation of the above method would read: “If the square contains the point where the mouse was clicked *then* tell the square to move to the right by `xOffset` pixels. Otherwise do nothing.” (Of course, the computer code doesn’t say, “Do nothing.” Instead it simply omits the *else*–part.)

We will present further variations of *if* statements later in this chapter, but first we will explore the kinds of expressions that can be used to form the *condition* part of these statements.

Exercise 4.2.2

- a. What would happen if you clicked the mouse inside `square` given the `onClick` method shown below? Assume `xOffset` is 60.

```
method onClick(point) {
    if ( square.contains( point ) ) {
        square.move( xOffset, 0 )
    }
    square.move( -xOffset/2, 0 )
}
```

- b. What would happen with the above code if you clicked the mouse outside `square`.
- c. What would happen if you now clicked the mouse outside of `square` and `onClick` was changed to the following?

```
method onClick(Location point) {
    if ( square.contains( point ) ) {
        square.move( xOffset, 0 )
    } else {
        square.move( -xOffset/2,0 )
    }
}
```

4.3 Understanding Conditions

Comparison operators like “`<`” are used in expressions that evaluate to either *true* or *false*, called boolean expressions. `contains` contains several comparison operators. They include:

`<` , `>` , `==` , `<=` , `>=` , `!=`

We must use `==` to test for equality because `=` has already been used for definitions. Because `≠` isn't available on all keyboards, the symbol `!=` (read as “does not equal”) stands for inequality. Because keyboards also do not always include the symbols `≤` or `≥`, Grace uses the combinations `<=` and `>=` in their places.

Be very careful not to confuse “`=`” and “`==`”. The first is only used in definitions, while the second is used only for comparisons. Keep in mind as well that “`:=`” is used for assignments to variables.

Using these comparison operators, we can write `x > 4`, `y != (z+17)`, and `(x+2) <= y`. Depending on the values of the variables, each of these expressions will evaluate to either *true* or *false*. Comparison operators are not assigned a precedence, so expressions being compared must be surrounded by parentheses unless the expression is just a simple identifier or method request. Thus in the examples above, `z+17` and `x+2` are surrounded by parentheses.

Suppose the current value of `x` is 3, `y` is 6, and `z` is -10. Here are the results of evaluating the above expressions:

- `x > 4` is *false* because 3 is not greater than 4.
- `y != (z+17)` is *true* because 6 is different from -10+7.
- `(x+2) <= y` is *true* because 3+2 is less than or equal to 6.

Exercise 4.3.1 *Suppose the current value of `x` is 2, `y` is 4, and `z` is 15. Determine whether each of the following conditions evaluates to true or false.*

- `(x + 2) < y`.
- `(z - 3 * x) != (y + 5)`.
- `(x * y) == (z - 9)`.
- `z >= (3 * y)`.

4.3.1 Using Boolean Values

Grace expressions can have the values *true* and *false*. Just as one can write down integer values directly as 17, -158, or 47, we can write down boolean values directly in Grace as *true* or *false*. We can also declare variables that can hold boolean values.

There are a large number of expressions in Grace that return boolean values. As we have just seen, combining two integer-valued expressions with one of the comparison operators, `<`, `>`, `<=`, `>=`, `==`, and `!=`, results in a boolean value. We have also seen the method `contains` that returns a boolean value.

If `ok` is a variable holding boolean values and `x` represents a number then the following are valid statements:

```
ok := true
ok := (x >= 3)
```

In each case the expression of the right hand side evaluates to a boolean value and hence can be assigned to a boolean variable.

We will use both the `contains` method and boolean variables in the implementation of a new program `WhatADrag`. The complete code listing for this program is given in Figure 4.6. The program

```

dialect "objectdrawDialect"

object{
  inherits aGraphicApplication . size (400,400)
  // create box
  def startPosn = aLocation.at(200,100)
  def box = aFilledRect .at(startPosn) size (30,30) on (canvas)

  var lastPoint // point where mouse was last seen

  // whether the box has been grabbed by the mouse
  var boxGrabbed := false

  // Save starting point and whether point was in box
  method onMousePress(point) {
    lastPoint := point
    boxGrabbed := box.contains(point)
  }

  // if mouse is in box, then drag the box
  method onMouseDrag(point) {
    if ( boxGrabbed ) then {
      box.moveBy( point.x - lastPoint.x, point.y - lastPoint.y )
      lastPoint := point
    }
  }

  startGraphics
}

```

Figure 4.6: Code for dragging a box.

begins by displaying a box on the screen. If the user presses the mouse down while it is pointing inside of the box. and then drags the mouse, the box will follow the mouse on the canvas. If the mouse is not pointing in the box when the mouse button is pressed, then dragging the mouse should have no effect, even if the mouse happens to cross the box at some point during the drag.

Let's look at the code in the mouse-handling methods to see how we can program this behavior. As soon as the mouse button is pressed, the `onMousePress` method is executed. The first assignment in `onMousePress`,

```
lastPoint := point
```

results in saving the current location of the mouse (as held in parameter `point`) as the value of variable `lastPoint`. The location is saved so that it can be used when `onMouseDrag` is executed later.

The second assignment in `onMousePress`,

```
boxGrabbed := box.contains( point )
```

determines and then remembers whether the box contained the point where the mouse was initially pressed. Because the result of evaluating `box.contains(point)` is a boolean value, `boxGrabbed` will represent either true or false. This value of `boxGrabbed`, which reflects whether the user actually pressed the mouse down inside box, will be used in `onMouseDrag` to determine whether the box should

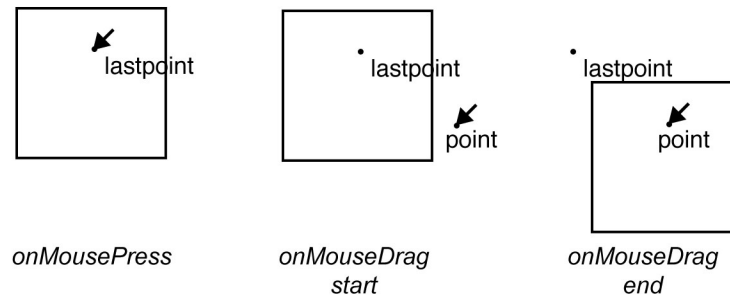


Figure 4.7: Three stages of dragging a rectangle.

be moved.

As usual, the `onMouseDown` method will be executed when the user drags the mouse. At that time the values of the variables `boxGrabbed` and `lastPoint` become relevant.

If the value of `boxGrabbed` is true, `box` will be moved by the distance between the last location of the mouse, saved in `lastPoint`, and the current position, held in `point`. The distance in each of the horizontal and vertical directions is needed to actually perform the move. The horizontal and vertical distances are obtained by evaluating `point.x - lastPoint.x` and `point.y - lastPoint.y`, respectively.

Figure 4.7 illustrates three stages of dragging a box in program `WhatADrag`. In the leftmost picture, the mouse button has been pressed with the mouse inside the rectangle. After the execution of `onMouseDown`, the location of the mouse is stored in the instance variable `lastPoint`. In the middle picture, the mouse has been dragged down and to the right, and `onMouseDown` has just begun execution. The current location of the mouse is held in the parameter `point`, but the `if` statement has not yet been executed. The rightmost picture shows what has happened immediately after the move message has been sent to `box` in the `if` statement during the execution of `onMouseDown`. The rectangle has been dragged to the right and down by the difference between the x coordinates and by the difference between the y coordinates of `point` and `lastPoint`. The update of the value of `lastPoint` to the location held in `point` is not shown.

Exercise 4.3.2 *Rather than using a variable to recall whether we clicked on the box, we might have written a condition that tested to see if `box` contained `point` each time the mouse was dragged as shown below.*

```

method onMouseDrag( point ) {
  if ( box.contains(point) ) then {
    box.move( point.x - lastPoint.x,
              point.y - lastPoint.y )
    lastPoint := point
  }
}

```

This would not quite work as desired. On the other hand, replacing the condition in the `if` statement with `box.contains(lastPoint)` would produce a program with exactly the same behavior as the version

that used the variable `boxGrabbed`. Explain why the version using `point` as a parameter to `contains` would not work. In what circumstances and how does the other one behave differently?

Exercise 4.3.3 Suppose we replace the method `onMouseDown` in program `WhatADrag` by the following code:

```
// if mouse is in box, then drag the box
public void onMouseDrag( Location point ) {
    if ( boxGrabbed ) then {
        box.moveTo( point )
    }
}
```

This is simpler than the code in Figure 4.6. For example, we no longer need to keep track of `lastPoint`. However, it does not result in as nice behavior. Explain why!

Exercise 4.3.4 Suppose the statement `lastPoint := point` inside of method `onMouseDown` in Figure 4.6 was placed after the end of the `if` statement rather than inside the `if`-part. How would this change the appearance on the screen during execution?

4.4 Selecting Among Many Alternatives

The `if-else` statements discussed earlier in this section are particularly suitable when we have to choose between executing two different blocks of statements at some point in a program. However, sometimes there are more than two alternatives that have to be considered.

As a simple example, consider how to assign letter grades based on numeric scores on an examination. Scores greater than or equal to 90 are assigned an “A,” scores from 80 to 89 are assigned a “B,” scores from 70 to 79 are assigned a “C,” and those below 70 are assigned “no credit.” Suppose that a variable `score` of type `int` contains the numeric score of a particular examination. We would like to display the appropriate letter grade in a `Text` item, `gradeDisplay`. In this situation we have not just two, but *four* different possibilities to worry about, so it is clear that a simple `if-else` statement is insufficient.

Grace allows us to extend the `if-else` statement for more than two possibilities by including one or more `elseif` clauses in an `if` statement. Thus we can display the appropriate grade using the following statement:

```
if ( score >= 90 ) then {
    gradeDisplay.contents := "The grade is A"
}
elseif ( score >= 80 ) then {
    gradeDisplay.contents := "The grade is B"
}
elseif ( score >= 70 ) then {
    gradeDisplay.contents := "The grade is C"
}
else {
    gradeDisplay.scontents := "No credit is given"
}
```

When we execute this `if` statement, the computer first evaluates the boolean expression `score >= 90`. If that is true, an “A” will be displayed and execution will continue after the last `else-part`

of the statement. However, if it is false the program will evaluate the next boolean expression, `score >= 80`. If that is true, a grade of “B” will be displayed and execution will continue after the last `else`–part. If not, the expression `score >= 70` will be evaluated. If that is true then a grade of “C” will be displayed and execution will continue after the `else`–part. Otherwise, the statement in the `else`–part will be executed and a grade of “no credit” will be displayed.

When checking to see if the student should be given a “B,” why didn’t we also have to check whether `score < 90`? The reason is that to get to the condition `score >= 80`, the previous test, `score >= 90`, must have already *failed*. That is, we *only* execute the `elseif` clauses if `score` is less than 90. The same reasoning shows that we do not need to check if `score < 80` when we determine whether to give a “C” as the grade. We can always count on the fact that the conditions within the `if`–`else` statements are evaluated sequentially, and therefore that all previous tests in the `if` statement have failed before determining whether to execute the next block.

We can summarize the execution of an `if` statement including `elseif`’s as follows:

- Evaluate the conditions after the `ifs` and `elseif`’s in order until one is found to be true.
- Execute the statements in the block following that `if` or `elseif` and then resume execution with the first statement after the entire `if` or `elseif` statement.
- If none of the conditions is true and there is an `else`–part, then execute the statements in the `else`–part. If there is no `else`–part, don’t execute any of the statements in the `if` statement.
- Finally, continue execution with the first statement after the `if` statement.

If there is an `else` clause in an `if` statement, then it must be the very last part of the `if`. That is, no further `elseif`’s are allowed after a plain `else` clause.

Exercise 4.4.1 *What is the output when the following pieces of code are executed? Why does the output of the three pieces of code differ? Assume `hamburgerPrice` is 7.*

a.

```
if (hamburgerPrice < 2) then {
    print "This hamburger is super cheap."
} elseif (hamburgerPrice < 4) then {
    print "This hamburger is inexpensive."
} elseif (hamburgerPrice < 6) then {
    print "This hamburger is fairly inexpensive."
} elseif (hamburgerPrice < 8) then {
    print "This hamburger is moderately priced."
} elseif (hamburgerPrice < 10) then {
    print "This hamburger is pricey!"
} else {
    print "This hamburger is super expensive!"
}
```

```
if (hamburgerPrice >= 10) {
    print "This hamburger is super expensive."
} elseif (hamburgerPrice < 10) then {
    print ("This hamburger is pricey!")
} elseif (hamburgerPrice < 8) then {
```



```

    print ("This hamburger is moderately priced."
} elseif (hamburgerPrice < 6) then {
    print ("This hamburger is fairly inexpensive."
} elseif (hamburgerPrice < 4) then {
    print ("This hamburger is inexpensive."
} else {
    print ("This hamburger is super cheap!"
}

```

b.

```

if (hamburgerPrice < 2) then {
    print ("This hamburger is super cheap.")
} if (hamburgerPrice < 4) then {
    print ("This hamburger is inexpensive.")
} if (hamburgerPrice < 6) then {
    print ("This hamburger is fairly inexpensive.")
} if (hamburgerPrice < 8) then {
    print ("This hamburger is moderately priced.")
} if (hamburgerPrice < 10) then {
    print ("This hamburger is pricey!")
} if (hamburgerPrice >= 10) then {
    print ("This hamburger is super expensive!")
}

```

4.5 More on Boolean Expressions

The if–else statement in the Voting program at the beginning of this chapter was sufficient because there were only two candidates to consider when assigning a new vote. However, for more than two candidates the `elseif` clause introduced in the last section becomes necessary.

In the next example our program must choose between 3 candidates, A, B, and C, who each have been allotted a vertical third of the canvas. We will not rewrite the entire program here, but will instead focus on the `onMouseClicked` method.

Let `leftSeparator` and `rightSeparator` be constants representing the x coordinates of the vertical lines that divide the canvas into the three pieces. The variables `countA`, `countB`, and `countC` will keep track of the number of votes for the three candidates. Here is the code:

```

// Update votes and display vote counts for 3 candidates
method onMouseClick( point ) {
    if ( point.x < leftSeparator ) { // clicked in left section
        countA := countA + 1
        infoA.contents := "So far there are {countA} votes for A."
    }
    elseif ( point.x < rightSeparator ) { // clicked in center
        countB := countB + 1
        infoB.contents := "So far there are {countB} votes for B."
    }
    else { // clicked in right section
        countC := countC + 1
        infoC.contents := "So far there are {countC} votes for C."
    }
}

```

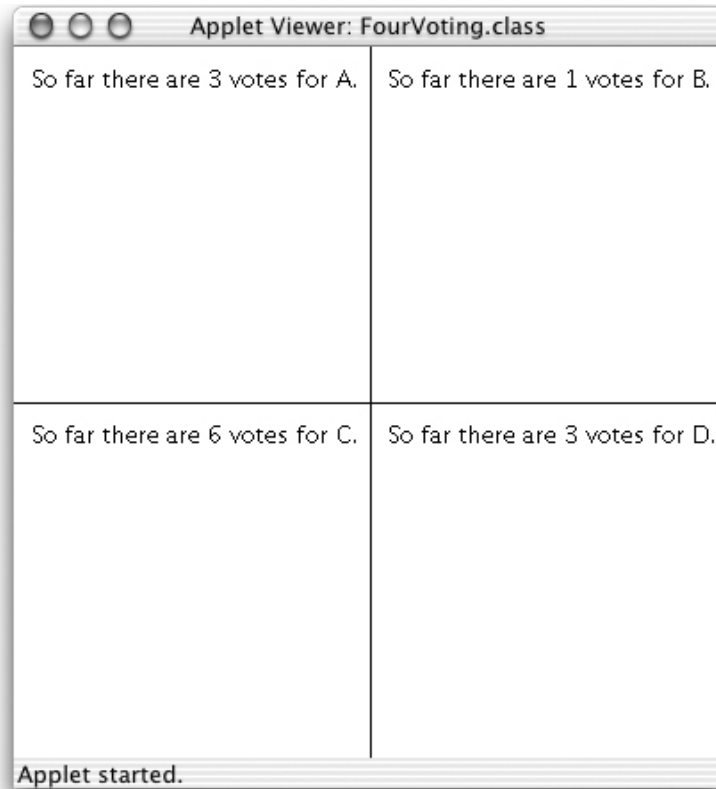


Figure 4.8: Voting for four candidates.

When determining whether the click was in the center section, why didn't we have to check that `point.x` was to the right of `leftSeparator`? As with our earlier example, the reason is that to even get to the second condition we *know* that the test `point.x < leftSeparator` must have failed (that is, it must have evaluated to *false*).

This same style solution works for determining whether clicks are in 4 or more vertical strips. However, once we get to four candidates it might make sense to divide the screen both vertically and horizontally rather than into 4 narrow, vertical strips as shown in Figure 4.8. We will count clicks in the upper-left hand corner as votes for A, in the upper-right hand corner as votes for B, lower-left for C, and lower-right for D. Thus to be a vote for A, the location where the user clicked must be *both* above the horizontal line *and* to the left of the vertical line. How can we write this as a condition?

In Grace we use the *and* operator, `&&`, between two boolean expressions to indicate that *both* must be true for the entire expression to be true. For example, we can ensure that `x` is positive and `y` is negative by writing `(x > 0) && (y < 0)`. While it would be convenient, computer programming languages do not usually allow us to combine two inequalities, as in the expression `1 <= x <= 10`. Instead we must write it out as `(1 <= x) && (x <= 10)`. *Note that all the parentheses are necessary as otherwise Grace wouldn't know how to group the operations.*

The `&&` is an example of a logical, or boolean, operator in Grace. Just as the “+” operator on

numbers takes two number values and returns a number, boolean operators take two boolean values and returns a boolean value. The `&&` (and) operator returns true exactly when both boolean values are true. Thus `(x > 0) && (y < 0)` will be true only if *both* `x` is greater than 0 *and* `y` is less than 0.

In order to ensure that `point` is in the upper-left corner of the canvas we would need to write:

```
if ( (point.x < midX) && (point.y < midY) ) then { // upper-left
    ...
}
```

The `if`-part is only executed if *both* `point.x < midX` *and* `point.y < midY`. If either one of those boolean expressions is false, then the entire condition evaluates to false and the `if`-part is not executed.

Here is the complete code to assign votes to four candidates:

```
// Update votes and display vote counts
method onMouseClick( point ) {
    if ( point.x < midX && point.y < midY ) then { // upper-left
        countA := countA + 1
        infoA.setText( "So far there are " + countA + " votes for A." )
    }
    elseif ( point.x >= midX && point.y < midY ) { // upper-right
        countB := countB + 1
        infoB.setText( "So far there are " + countB + " votes for B." )
    }
    elseif ( point.x < midX && point.y >= midY ) { // lower-left
        countC := countC + 1
        infoC.setText( "So far there are " + countC + " votes for C." )
    }
    else { // lower-right
        countD := countD + 1
        infoD.setText( "So far there are " + countD + " votes for D." )
    }
}
```

Just as Grace uses `&&` to represent the logical *and* operation, it uses `||` to represent the logical *or* operator. Thus `x < 5 || y > 20` will be true if `x < 5` *or* `y > 20`. In general, if b_1 and b_2 are boolean expressions, then $b_1 || b_2$ is true if *one* or *both* of b_1 and b_2 are true.

English contains both an *inclusive* and an *exclusive* “or”. The inclusive “or” evaluates to true if either or both operands are true. The exclusive “or” evaluates to true only if exactly one of the operands is true. The `||` operator represents the inclusive “or”. Using the “not” operator, `!`, defined below, you can get the effect of the exclusive “or” on boolean expressions `b1` and `b2` by writing `(b1 || b2) && !(b1 && b2)`, which states that one of `b1` and `b2` is true, but not both.

A good example illustrating the use of the *or* operation is determining whether someone playing the game of craps has won or lost after his or her first roll of the dice. The “shooter” in craps throws two dice. If the numbers on the faces of the dice add up to 7 or 11, then the shooter wins. A sum of 2, 3, or 12 results in an immediate loss. With any other result, play continues in a way that will be described later.

The `if` statement below has three branches that encode the relevant outcomes of the first roll of the dice. The value of `status` is simply a text object that, as usual, has been created in the object.

```
if ((roll == 7) || (roll == 11)) then { // 7 or 11 wins on first throw
    status.contents := "You won!"
} elseif ((roll == 2) || (roll == 3) || (roll == 12)) then { // 2, 3, or 12 lose on 1st throw
    status.contents := "You lose!"
```

operator	meaning
<code>&&</code>	<i>and</i>
<code> </code>	<i>or</i>
<code>!</code>	<i>not</i>
<code>==</code>	<i>equal</i>
<code>!=</code>	<i>not equal</i>
<code><</code>	<i>less than</i>
<code><=</code>	<i>less than or equal</i>
<code>></code>	<i>greater than</i>
<code>>=</code>	<i>greater than or equal</i>

Figure 4.9: A summary of the boolean and comparison operators in Grace

```

} else {
    status.contents := "The game continues"
}

```

The `if` portion determines if the player has won by checking if the `roll` was 7 or 11. The `elseif` portion determines if the player has lost by checking if `roll` was 2, 3, or 12. Finally the `else` portion is executed if further rolls of the dice will be required to determine whether the player wins or loses.

The last boolean operator to be introduced here is `!`, which stands for “not”. For example, the expression `!box.contains(point)` will be true exactly when `box.contains(point)` is false, *i.e.*, when `box.contains(point)` is *not* true.

Although we can use `!` with equations and inequalities, it is usually clearer to rewrite the statement using a different operator. For instance, `!(x == y)` is more simply written as `x != y`, and `!(x < y)` is simplified to `x >= y`. Similarly, `!(x <= y)` is equivalent to `x > y`, which is much easier to read.

Figure 4.9 summarizes the most common operators in Grace that give a boolean result.

Exercise 4.5.1 Suppose that the current value of x is 6, y is -2, and z is 13. For each of the following conditions, determine whether they evaluate to true or false.

- $((x - 6) < y) \ \&\& \ (z == (2 * x + 1))$.
- $!((x - 6) < y) \ \&\& \ (z == (2 * x + 1))$.
- $((x - 6) < y) \ || \ (z == (2 * x + 1))$.
- $!(((x - 6) < y) \ || \ (z == (2 * x + 1)))$.

Before we move on, we should note a few last points about `&&` and `||`. The first is to be sure and use the double version of each of the symbols `&` and `|`.

Second, both `&&` and `||` in Grace are implemented as “short circuit” operations. What this means is that Grace will cease evaluating an expression involving one of these operators as soon as it can determine whether the entire expression is true or false.

For example, suppose a program includes a declaration of an variable x representing a number, and that it contains the expression $(x > 10) \ \&\& \ (x \leq 20)$. If the computer evaluates that expression

when x has value 3, it will only evaluate $x > 10$ without even considering $x \leq 20$. Because $x > 10$ is false, the expression will first evaluate to `false && (x <= 20)`. As a result, Grace can determine that the final value of the `&&` expression must be false no matter what the value of $x \leq 20$. However, if $x > 10$ had been true, the rest of the expression would have to be evaluated to determine whether the entire `&&` expression evaluates to true or false.

While you are unlikely to care much in this case whether the second argument is evaluated, there are other cases in which evaluating the second argument may result in an error. For example if the boolean expression `(x != 0) && (3/x > 17)` were not evaluated in this short circuit fashion, then if x were 0 at run-time, a run-time error would result when $3/x$ is evaluated.

Expressions involving the `||` operator are also evaluated in a short circuit fashion. If the left side evaluates to true then Grace knows that the entire `||` expression *must* evaluate to true, so it does not bother to evaluate the right side. Conversely, if the left side is false then the right side must be evaluated in order to determine the final value of the entire `||` expression.

4.6 Nested Conditionals

Occasionally we run into problems that would require complex boolean conditions if they are handled using `if` or `elseif` statements as we have seen them used so far. Rather than constructing these complex conditions, we will introduce alternative structures for supporting the program logic. Happily, we don't need to introduce any more syntax in order to handle them; we just need to combine `if` statements in different ways.

Suppose it is a summer weekend and you are trying to figure out what to do. Your choice of recreation will depend on the weather and how much money you have to spend. The following table lists the various options and choices, where the row headings represent your possible financial situations and the column headings represent the weather possibilities.

	sunny	not sunny
rich	outdoor concert	indoor concert
not rich	ultimate frisbee	watch TV

The table entries represent the suggested recreational activity given the financial situation represented by the row and the weather as represented by the column. Thus if you are feeling rich and it is not sunny, you might want to go to an indoor concert. If you are not feeling rich and it is sunny, you might play some ultimate frisbee.

How can we represent these choices with an `if` statement? Let `rich` and `sunny` be variables of type `boolean`, and let `activityDisplay` be a variable of type `Text` that will display the selected activity. The `if` statement below uses `elseif` clauses to represent the four choices.

```

if ( sunny && rich ) then {
    activityDisplay .contents := "outdoor concert"
} elseif ( !sunny && rich ) then {
    activityDisplay .contents := "indoor concert"
} elseif ( sunny && !rich ) then {
    activityDisplay .contents := "ultimate frisbee"
} else { // !sunny &&& !rich
    activityDisplay .contents := "watch TV"
}

```

FIX! ► *Minigrace does support this precedence. Should it?* ◀ As we will discuss in more detail in the next chapter, `!` has *higher precedence* than `&&`. This means that the not operator, `!`, will always be applied before the `&&` operator. Thus the condition `!sunny && rich` is evaluated by first evaluating `!sunny` and then using the `&&` operation to determine whether both `!sunny` and `rich` are true.

This code correctly represents all four options, but is rather verbose and loses the nice structure of the table. A related problem is that by the time we arrive at the last case the program has evaluated three fairly complex boolean expressions.

We can write this so that only two evaluations of boolean variables are ever made, and furthermore that they are made without the added complication of *negation* or *and* operators. This is accomplished by *nesting* if statements. A nested if statement involves including one or more if statements inside another, as in the example below.

```
if ( sunny ) {
  if ( rich ) {
    activityDisplay .contents := "outdoor concert"
  }
  else { // not rich
    activityDisplay .contents := "ultimate frisbee"
  }
}
else { // not sunny
  if ( rich ) {
    activityDisplay .contents := "indoor concert"
  }
  else { // not rich
    activityDisplay .contents := "watch TV"
  }
}
```

The advantage to using these nested if statements is that the organization is quite similar to that of the table. There is an outer if–else statement that determines whether `sunny` is true. This corresponds to choosing either the first or second column of the table. Inside the outer if–part there is an if–else statement that determines whether `rich` is true. This corresponds to figuring out which row of the table applies. For example, if `rich` is false, the outcome should correspond to the first column and second row of the table, and hence the `activity` should be “play ultimate”. The else–part corresponding to it not being sunny is handled in a similar fashion.

Style Note: The nested if–else statements are indented from the outer ones in order to make the code easier to read and understand. Grace compilers enforce this indenting because human readers generally need the cues of indenting to understand complex code like this.

Aside from the indenting, another thing that makes this code easy to understand is the inclusion of comments. In particular, notice how each else clause includes comments indicating under which conditions the else–part is executed. This has the advantage of making it absolutely clear to the reader under what circumstances this code is executed. The more complex the conditional, the more important these comments become. We strongly urge all programmers to include such comments.

While the use of nested if–else statements yields code that is not quite as compact and simple to understand as the table, it is much easier to see its correspondence to the table than the version involving only `elseif` clauses. Notice in particular that no matter what the values of `sunny` and `rich` are, only two boolean variables are ever evaluated during the execution of this code, so it is not

only clearer, but it is faster as well!

Exercise 4.6.1 Use nested conditionals to rewrite the `onMouseClicked` method for tabulating votes of four candidates from Section 4.5.

Exercise 4.6.2 Using the information provided in the table and nested conditionals, write an *if-statement* that displays which course a student should take based on their interests in math and writing. Let `likesMath` and `likesWriting` be variables holding boolean values, and let `course` refer to a text object that will display the recommended course.

	<i>likes writing</i>	<i>doesn't like writing</i>
<i>likes math</i>	<i>Economics</i>	<i>Calculus</i>
<i>doesn't like math</i>	<i>English</i>	<i>Psychology</i>

In Figure 4.10 we provide another example of complex choices being represented by nested *if-else* statements. This program provides the code to simulate a complete game of craps. The rules of craps are as follows:

The shooter rolls a pair of dice. If the shooter rolls a 7 or 11, it is a win. If the shooter rolls a 2, 3, or 12, it is a loss. If the shooter rolls any other number, that number becomes the “point”. To win, the shooter then must roll the “point” value again before rolling a 7. Otherwise it is a loss.

The program simulates a roll of the dice by using a random number generator every time the user clicks the mouse. In order to implement the rules given above, we must organize the game logic in a way that can be represented using *if* statements. Notice that the rules for winning are quite different depending on whether this is the player’s first throw. For instance, if it is the first throw, then rolling a 7 results in a win, but if it is a second or subsequent throw, then 7 results in a loss. Therefore we will organize the first level of conditional to determine whether it is the first throw.

In order to make such a choice, the Craps program in Figure 4.10 declares a variable, `newGame`, to remember whether this is the first throw of a new game.

The outer *if* statement in the method `onMouseClicked` has the following structure:

```
if ( newGame ) { // starting a new game
    ...
}
else {           // continuing trying to make the point
    ...
}
```

The *if*-part of this code is itself an *if* statement with three branches, each of which encodes the relevant actions to be taken based on the first roll of the dice. Recall that we saw a simplified version of this example earlier in the chapter. The new code is reproduced below:

```
if (( roll == 7 ) || ( roll == 11 )) then { // 7 or 11 wins on first throw
    status.contents := "You won!"
} elseif (( roll == 2 ) || ( roll == 3 ) || ( roll == 12 )) then { // 2, 3, or 12 lose on 1st throw
    status.contents := "You lose!"
} else { // Set the roll as the new point to be made and continue game
    status.contents := "Try for your point!"
    point := roll
    newGame := false // set for continuing game
}
```

```

dialect "objectdrawDialect"

object {
  inherits aGraphicApplication . size(400,400)
  var newGame := true    // True if starting new game
  var point    // number trying to roll to win
  def maxVal = 6        // max number on die

  def welcomeMessageLoc = aLocation.at(20,50) // location of text items
  def statusLoc = aLocation.at(10,70)
  def instructionLoc = aLocation.at(10,90)

  // display instructions
  def welcomeMessage = aText.at(welcomeMessageLoc) with ("Let's play craps!") on (canvas)
  aText.at(instructionLoc) with ("Click to roll the dice") on (canvas)

  def status = aText.at(statusLoc) with ("") on (canvas) // display status of game
  var roll // total score for a roll of two dice

  // For each click, roll the dice and report the results
  method onMouseClick( pt ) {
    // get values for both dice and display sum
    var roll := randomIntFrom(1)to(maxVal) + randomIntFrom(1)to(maxVal)
    welcomeMessage.contents := "You rolled a {roll}!"

    if (newGame) then { // starting a new game
      if (( roll == 7) || ( roll == 11)) then { // 7 or 11 wins on first throw
        status.contents := "You won!"
      } elseif (( roll == 2) || ( roll == 3) || ( roll == 12)) then { // 2, 3, or 12 lose on 1st throw
        status.contents := "You lose!"
      } else { // Set the roll as the new point to be made and continue
        game
        status.contents := "Try for your point!"
        point := roll
        newGame := false // set for continuing game
      }
    } else { // continuing trying to make the point
      if ( roll == 7) then { // 7 loses when trying for point
        status.contents := "You lose!"
        newGame := true // set to start new game
      } elseif ( roll == point) then { // making the point wins!
        status.contents := "You won!"
        newGame := true
      } else { // keep trying
        status.contents := "Keep trying for {point} ..."
      }
    }
  }
}
startGraphics
}

```

Figure 4.10: Craps program illustrating nested conditionals.

Rather than having a separate branch for each possible value of the roll of the dice, there are only three. These branches correspond to winning, losing, and establishing a point to be made on subsequent rolls. The variable `newGame` remains true in the first two branches, so it need not be updated. Only the third branch requires setting `newGame` to false.

Let us now examine the `else`-part of the outer `if` statement. Like the `if`-part, this nested `if` statement also has 3 branches, though the second and third conditions are quite different from those that handle the first roll:

```
if ( roll == 7) then {           // 7 loses when trying for point
    status.contents := "You lose!"
    newGame := true             // set to start new game
} elseif ( roll == point) then { // making the point wins!
    status.contents := "You won!"
    newGame := true
} else { // keep trying
    status.contents := "Keep trying for {point} ..."
}
```

In this statement, both of the first two choices result in setting `newGame` back to true because they represent the end of a game with either a win or a loss. The third statement merely asks the player to continue rolling, so `newGame` remains false.

Exercise 4.6.3 *Try writing out this program using only a single `if` statement with many `elseif` clauses. It should become painfully clear why nested `if` statements are useful in situations with complex logic.*

4.7 Summary

In this chapter we introduced conditional statements and the boolean data type. The major points discussed were:

- The `if-else` statement is used when different code is to be executed depending on the value of a condition.
- `if` statements without an `else` clause are used when extra code is to be executed in one case, but nothing extra is needed in the other.
- `if` statements with `elseif` clauses are used if there are more than two cases to be considered in a choice. `if` statements with `elseif` clauses may or may not be terminated with an `else`-part, at the programmer's option.
- Nested `if` statements can be used to represent complex logic.
- Identifiers `true` and `false` represent boolean values. Comparison operators return boolean values. Boolean expressions can be combined with the boolean operators `&&`, `||`, and `!`.

4.8 Chapter Review Problems

Exercise 4.8.1 *Define the following Grace terms:*

- a. *negation*
- b. *block*
- c. *!=*
- d. *condition*

Exercise 4.8.2 *How would you express the following as operators in Grace?*

- a. *greater than*
- b. *or*
- c. *equal*
- d. *less than or equal*
- e. *not*
- f. *and*
- g. *not less than*

Exercise 4.8.3 *The database at a doctor's office stores information about each patient. For each piece of information listed below, decide whether or not it would be appropriate to store the information as a boolean.*

- a. *patient's birthday*
- b. *patient's gender*
- c. *patient's address*
- d. *whether the patient has visited within the last year*
- e. *patient's insurance company*
- f. *whether the patient's last visit is fully paid*

Exercise 4.8.4 *Fix the problem(s) in the following code:*

```

if ( x = 5 ) then {
    message.contents := "You win!"
}
else if ( x < 5 ) then {
    message.contents := "You lose!"
}
else if ( x > 5 ) then {
    message.contents := "Try again!"
}

```

Exercise 4.8.5 *What are the values of the following expressions if $x = 8$, $y = -5$ and $z = 2$.*

- a. $(x + y) \neq z$
- b. $!((2 * x + 3 * y) \geq z)$
- c. $((x - z) * 2) < (z - 2 * y)$
- d. $(y - z + x) \geq 0$
- e. $(x - 4 * z + 1) == y$

Exercise 4.8.6 *Why is the following code more complex than it needs to be? Simplify it using else-if*

```

if ( (score <= 100) && (score >= 90)) then {
  gradeDisplay.contents := "You got an A"
}
if ((score < 90) && (score >= 80)) then {
  gradeDisplay.contents := "You got a B"
}
if ((score < 80) && (score >= 70)) then {
  gradeDisplay.contents := "You got a C"
}
if (score < 70) then {
  gradeDisplay.contents := "You don't get a grade"
}

```

Exercise 4.8.7 *What are the values of the following expressions if $x == -2$, $y == -1$ and $z = 4$.*

- a. $((x + y) < z) \parallel (((4 * y + z) > x) \&\& (x > y))$
- b. $((x + y) < z) \parallel ((4 * y + z) > x) \&\& (x > y)$
- c. $!((x + y) < z) \parallel (((4 * y + z) > x) \&\& (x > y))$
- d. $((x + y) < z) \parallel ((4 * y + z) > x) \&\& !(x > y)$
- e. $!((x + y) < z) \parallel (((4 * y + z) > x) \&\& !(x > y))$

Exercise 4.8.8 *Assume that $x = 2$, $y = -3$ and $z = 5$. What are the values of x , y and z after the following code has been executed?*

```

a.
  if ( (3 * x + y) <= (z - 1)) then {
    x := y + 2 * z
  } else {
    y := z - y
    z := x - 2 * y
  }

  if ( x > (y + z) ) then {
    y := y - 1
    x := x + 1
  }

```

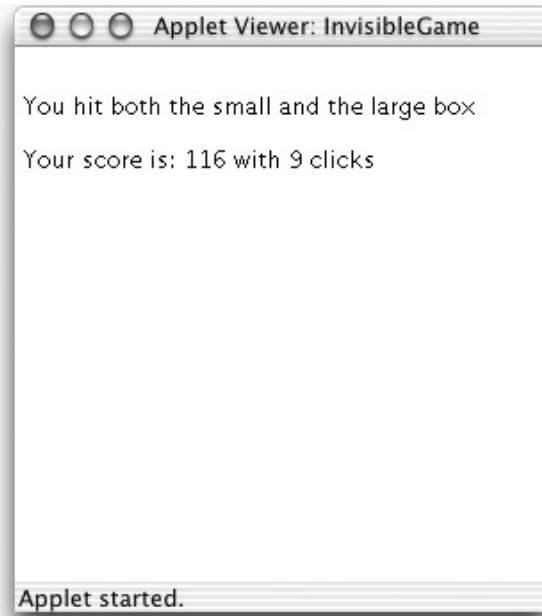


Figure 4.11: Display for InvisibleBox

- b. **Exercise 4.8.9** *Another variation of the game craps is to play what is called the “No Pass Line”. Now rolling a seven or eleven on the first roll loses and a two, three, or twelve wins. After the point is set, rolling a seven wins, while rolling the point again loses. Show how to modify Craps so that you now are playing the “No Pass Line” instead.*

4.9 Programming Problems

Exercise 4.9.1 *Create the game InvisibleGame to master your use of conditionals. The game will have the following features:*

- *When the game begins it will create three “invisible” boxes. The boxes will all be square but of different sizes. One should be 30 pixels wide, the second one 45 pixels wide and the third 80 pixels wide. These boxes should be created randomly anywhere on the screen.*
- *The user will click the mouse and try to hit the boxes when doing so.*
- *When the mouse exits the window, the user will be notified of his/her success. The output should look like Figure 4.11.*
- *When the mouse re-enters the window, all variables should be reset and the boxes moved to new random locations.*
- *Points are assigned as follows*
 - *200 points: Hitting all three boxes*

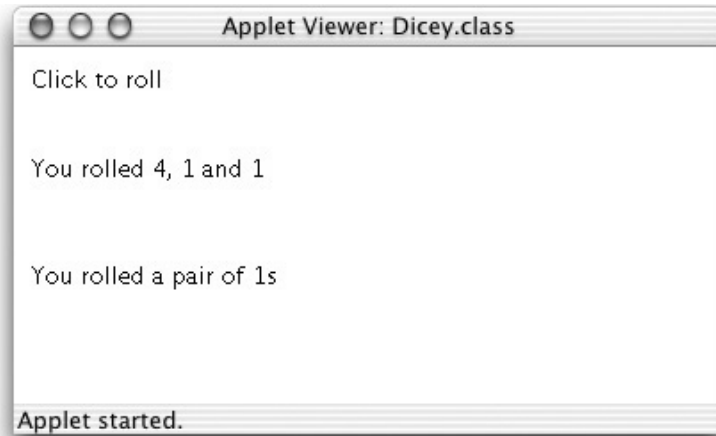


Figure 4.12: Display for Dicey

- 150 points: *Hitting the small box and the medium box*
 - 125 points: *Hitting the small box and the large box*
 - 110 points: *Hitting the medium box and the large box*
 - 100 points: *Hitting only the small box*
 - 75 points: *Hitting only the medium box*
 - 50 points: *Hitting only the large box*
 - -1 point: *For each mouse click*
- *Set it up so that the window is 300 pixels wide and 300 pixels high.*

Exercise 4.9.2 *Write a simple game with three dice called Dicey, which may remind you of a more popular game with five dice. When the user clicks the mouse, the three dice are “rolled” and the results are displayed on the screen. The display will also include whether the user rolled 3 of a kind, a pair or nothing of particular interest. See Figure 4.12*

Chapter 5

Types, Numbers, and Strings

The programs that we are now writing in Grace are starting to get complex enough now that it will be useful to get help from the computer in writing correct programs. We all make mistakes when we are doing things, so it will help to anticipate that eventuality and have assistance ready.

It probably won't surprise you to know that it is easier to fix errors when they are discovered early. I've put together lots of furniture pieces and children's toys. If I put something in wrong early on, but don't discover the error until I'm nearly done, I generally have to take everything apart and essentially start over. However, if I recognize the error right away, it is much easier to fix the problem and complete the task.

In this chapter, we will introduce the notion of type annotations in order to document the programmer's intentions. These type annotations will not only make it easier for a reader to figure out what your program is trying to do, but they will also be machine checkable so that you can be warned if the Grace compiler detects any inconsistencies in your program. Especially helpfully, this machine check will happen before your program is executed, helping you to find errors faster and typically providing better information on where the error occurs.

Once we introduce the notion of type, we talk about three kinds of types in some detail. The first two to be discussed are the types `Number` and `String`. We then talk about the types of the geometric objects introduced in the `objectdraw` library.

5.1 What is a type?

In Grace, a type is a specification of the methods that an object can respond to. It is also useful to think of a type as the collection of all objects that can respond to the methods specified by the type.

Familiar types include numbers (written `Number`), booleans (written `Boolean`), and strings (written `String`). We'll come back and talk about the methods in those types, but let's first consider the types of objects from the `objectdraw` library.

5.2 Types from `objectdraw`

In section 2.2.2 we specified the operations available on locations. These operations are specified formally in the type `Location` from the `objectdraw` library:

```
type Location = {  
    // coordinates of location
```

```

x->Number
y->Number

// return new location shifted by dx to right and dy down
translate(dx:Number,dy:Number)->Location

// return distance from this location to other
distanceTo(other:Location)->Number

// return if other at same place as self
==(other:Location)->Boolean

// return string representation of location
asString->String
}

```

Each of the items in the `Location` type is the specification of a method supported by locations. Thus the first two items specify that an object of type `Location` has methods `x` and `y` that each return a number. The next item specifies that there is a method `translate` that takes two parameters `dx` and `dy` that are numbers and returns an object that is another `Location`.

In general, types are defined in declarations of the following form:

```

type NewType = {
  ...
  m(p1:T1 ,..., pn:Tn) -> RType
  ...
}

```

In this declaration, `NewType` is the name of the type being defined. Each line of the type expression specifies a method of the type.

In the example above, `m` is the name of the method. It takes `n` arguments or parameters. Each parameter is specified with a name and its associated type. Above, parameter `p1` has type `T1`. A “:” is used to indicate the association of identifier and type. If there is more than one parameter, the parameters (and their types) are separated by commas, and the entire collection of parameters is surrounded by parentheses. (If a method has no parameters then the parentheses may be omitted.) Finally the type of value returned by the method is placed after the parameters, separated from the parameters by “->” representing an arrow.

Going back to the example of the type `Location`, we can see that methods `x` and `y` take no parameters, but always return a number. Thus if `locn` is an identifier of type `Location`, then both `locn.x` and `locn.y` are legal expressions, and when evaluated should return numbers (assuming that `locn` has been initialized properly).

The method `distanceTo` specified in `Location` takes one parameter of type `Location` and has return type `Number`. Thus if `locn1` and `locn2` are associated with locations, then `locn1.distanceTo(locn2)` is a legal expression that will return a number representing the distance between the two locations.

The methods `==` and `asString` are included in all types. If `loc1` and `loc2` are generated by `aLocation` (and hence have type `Location`) then `loc1 == loc2` will be true if and only if the two locations have the same `x` and `y` coordinates.

All of the graphic items from the `objectdraw` library satisfy the type `Graphic` shown in Figure 5.1. Reading through the methods specified by the type we see that there are methods to retrieve the `x` and `y` coordinates of the graphics object as well as its `location`.

The next two methods in `Graphic` are used to retrieve and reset whether the object is visible on the canvas. Method `isVisible` returns true if and only if the object is currently visible, while method `isVisible :=` can be used to reset the visibility of the object. Thus `gobj.isVisible := true` sets it so that `gobj` is visible on the canvas, while `gobj.isVisible := false` makes it invisible. The `isVisible :=` method does not return a value, it just resets the visibility of the object. As a result we annotate the return type to be `Done`, indicating that nothing further will be done with the result. Because most mutator methods don't return values, they generally will have a return type of `Done` as well.

Methods like `isVisible:=` that end with `:=` are treated specially in Grace. Unlike regular methods, a programmer may insert one or more spaces before the `:=`. Second, the parameter of the method need not be surrounded by parentheses.

`Graphic` then has two methods to move the object, `moveTo`, which takes a `Location` as a parameter, and `moveBy`, which takes two numbers indicating how far it should move to the right and down. Both have return type `Done` as they don't return a value. The methods `color` and `color :=` are like `isVisible`, giving us access to or the ability to change the color of the object.

The methods `sendForward`, `sendBackward`, `sendToFront`, and `sendToBack` are used to change the relative order of objects on the canvas. They take no parameters and return `Done`. The final two methods are used to either add the graphic object to a canvas or remove it.

The methods in `Graphic` are available to all graphic objects in the `objectdraw` library, but they also contain other specialized methods.

For example, objects created via `aText` have type `Text`, shown in Figure 5.2. The `&` in the first line of the type definition indicates that `Text` contains all of the methods in type `Graphic` as well as those listed after the keyword `type`. In this case we can access the width of the text object as well as get and set the contents of the string and the `fontSize` of the string displayed on the canvas.

Lines and two-dimensional graphic objects will have somewhat different types. For example, lines have endpoints, while rectangles and ovals have length and width. The definition of `Line` is in Figure 5.3. Objects generated from `aLine` will satisfy this type.

Finally all of the two-dimensional graphics objects, including both filled and framed rectangles and ovals will satisfy the type `Graphic2D`, which is provided in Figure 5.4. The methods `contains` and `overlaps` both return booleans. The first returns true if the location parameter is inside the receiver graphic object, while the second returns true if the receiver and the parameter overlap.¹

5.3 Using type annotations

We can use type annotations in our program in order to make it easier to understand and to provide the Grace compiler with information needed to find errors earlier. The key is to annotate every identifier with its type when it is introduced. Thus we will provide types on definitions, variable definitions, and method headers – including types for parameters as well as the return type of the method.

To illustrate this, the `Craps` program from the last chapter, but with full type information, is given in Figure 5.5

As can be seen in the figure, we have annotated all definitions and variables with the appropriate type information. Thus `newGame` will always represent an object of type `Boolean`, while `status` will

¹ A couple of provisos: `overlaps` returns true if the bounding rectangles of the objects overlap. Thus, for example, it may return true for two ovals even if they don't overlap, but their bounding rectangles do.

```

type Graphic = {
  // location of object on screen
  x -> Number
  y -> Number
  location -> Location

  // Is this object visible on the screen?
  isVisible -> Boolean

  // Determine if object is shown on screen
  isVisible :=( _: Boolean) -> Done

  // move this object to newLocn
  moveTo(newLocn: Location) -> Done

  // move this object dx to the right and dy down
  moveBy(dx: Number, dy: Number) -> Done

  // set the color of this object to c
  color :=( c: GColor) -> Done

  // return the color of this object
  color -> GColor

  // Send this object up one layer on the screen
  sendForward -> Done

  // send this object down one layer on the screen
  sendBackward -> Done

  // send this object to the top layer on the screen
  sendToFront -> Done

  // send this object to the bottom layer on the screen
  sendToBack -> Done

  // Return a string representation of the object
  asString -> String

  // Add this object to canvas c
  addToCanvas(c: Canvas) -> Done

  // Remove this object from its canvas
  removeFromCanvas -> Done
}

```

Figure 5.1: The type Graphic from objectdraw

```

type Text = Graphic & type{
  // return the string shown in the text object
  contents -> String

  // reset the string shown in the text object to s
  contents:=(s:String) -> Done

  // return the width of the text object
  width -> Number

  // return the font size of the text object
  fontSize -> Number

  // reset the font size of the text object
  fontSize:=(size:Number) -> Done
}

```

Figure 5.2: The type Text from the objectdraw library

```

type Line = Graphic & type {
  // start and end of line
  start -> Location
  end -> Location

  // set start and end of line
  start:=(start ':Location) -> Done
  end:=(end':Location) -> Done
  setEndPoints(start ':Location,end':Location) -> Done
}

```

Figure 5.3: The type Line from the objectdraw library

```

type Graphic2D = Graphic2D & type {
  // dimensions of object
  width->Number
  height->Number

  // Change dimensions of object
  setSize(width:Number,height:Number)->Done
  width:=(width:Number)->Done
  height:=(height:Number)->Done

  // Does this object contain locn
  contains(locn:Location)->Boolean

  // Does other overlap with this object
  overlaps(other:Graphic2D)->Boolean
}

```

Figure 5.4: The type Graphic2D from the objectdraw library

```

dialect "objectdrawDialect"

object {
  inherits aGraphicApplication . size(400,400)

  var newGame:Boolean := true    // True if starting new game
  var point:Number     // number trying to roll to win

  def maxVal:Number = 6 // max number on die

  def welcomeMessageLoc:Location = aLocation.at(20,50) // location of text items
  def statusLoc:Location = aLocation.at(10,70)
  def instructionLoc :Location = aLocation.at(10,90)

  // display instructions
  def welcomeMessage:Text = aText.at(welcomeMessageLoc) with ("Let's play craps!") on (canvas)
  aText.at(instructionLoc) with ("Click to roll the dice") on (canvas)

  def status:Text = aText.at(statusLoc) with ("") on (canvas) // status of game
  var roll :Number // total score for a roll of two dice

  // For each click, roll the dice and report the results
  method onMouseClick( pt:Location ) -> Done {
    // get values for both dice and display sum
    roll := randomIntFrom(1)to(maxVal) + randomIntFrom(1)to(maxVal)
    welcomeMessage.contents := "You rolled a {roll}!"

    if (newGame) then { // starting a new game
      if (( roll == 7) || ( roll == 11)) then { // 7 or 11 wins on first throw
        status.contents := "You won!"
      } elseif (( roll == 2) || ( roll == 3) || ( roll == 12)) then { // 2, 3, or 12 lose on 1st throw
        status.contents := "You lose!"
      } else { // Set the roll as the new point to be made and continue game
        status.contents := "Try for your point!"
        point := roll
        newGame := false // set for continuing game
      }
    } else { // continuing trying to make the point
      if ( roll == 7) then { // 7 loses when trying for point
        status.contents := "You lose!"
        newGame := true // set to start new game
      } elseif ( roll == point) then { // making the point wins!
        status.contents := "You won!"
        newGame := true
      } else { // keep trying
        status.contents := "Keep trying for {point} ..."
      }
    }
  }
}
startGraphics
}

```

Figure 5.5: The Craps program with full type annotations

always refer to an object of type `Text`.

The method `onClick` takes a parameter of type `Location`, with a return type of `Done` indicating that it is a mutator method. All of the other mouse handling methods also have a single parameter of type `Location` and have a return type of `Done`.

5.4 Numbers

Most programming languages have different types of numbers. For example, many languages have different types for integers and numbers with fractional parts like 3.14159. In Grace we don't distinguish between these types and just have the single type `Number`, which represents both integers and numbers with fractional parts.

Most of the methods of type `Number`, presented in Figure ?? should be familiar to you. The operations `+`, `-`, `*`, and `/` are the usual arithmetic operations. The method written **prefix** `-` is the negation operator and is written before the receiver: `-n`. The specification **prefix** tells Grace to expect it before rather than after the receiver. As described in Chapter 3, `%` is the modulus operator, so `m % n` is the remainder when `m` is divided by `n`. Method `truncate` throws away the fractional part of a number, returning the integral part. Thus `5.34.truncate` is 5, while `-5.34.truncate` is -5.

The comparison operators `==`, `!=`, `>`, `>=`, `<`, `<=` are also the familiar operations. The method `asString` returns a string representation of the number. Method `inBase(b)` returns a string representation of the receiver in base `b`

```
type Number = {
  // return value of sum of receiver and other
  +(other:Number) -> Number
  // return value of difference of receiver and other
  -(other:Number) -> Number
  // return value of product of receiver and other
  *(other:Number) -> Number
  // return value of quotient of receiver and other
  /(other:Number) -> Number
  // return value of remainder when receiver divides into other
  %(other:Number) -> Number
  // return value of receiver raised to power of other
  ^(other:Number) -> Number
  // return integer part of receiver
  // (i.e., what is left after throwing away fractional part)
  truncate -> Number
  // return value of negative of receiver
  prefix - -> Number
  // return whether receiver and other are the same number
  ==(other:Number) -> Number
  // return whether receiver and other are different
  !=(other:Number) -> Number
  // return whether receiver is less than other
  <(other:Number) -> Boolean
  // return whether receiver is less than or equal to other
  <=(other:Number) -> Boolean
  // return whether receiver is greater than other
  >(other:Number) -> Boolean
  // return whether receiver is greater than or equal to other
```

```

>=(other:Number) -> Boolean
// return string representation of receiver
asString -> String
// return the receiver as a string expressed in base b
inBase(b:Number) -> String
}

```

We have explained that the common mathematical operations are methods, yet the syntax of an expression using `+`, for example, does not look like a method request.

5.4.1 The math module

Modules in Grace are pre-defined objects that contain definitions that can be imported into another program. The predefined math module in Grace includes other methods that are useful with type `Number`, including trigonometric functions such as `sin`, `cos`, `tan`, `asin`, `acos`, and `atan`, where the last three correspond to the arcsine, arccosine, and arctangent functions.

Suppose we have a Grace program in which the statement:

```
import "math" as mathObj
```

The string `"math"` after the keyword `import` is the name of the library as defined in Grace. The identifier after the key word `as` is the name of the object as it is used in the program. Thus in this program, we would write `mathObj.sin(0)` to obtain the value of the sine of 0 or `mathObj.cos(3.14159)` for the value of the cosine of a number close to π .²

5.5 Handy Sources of Numeric Information

Grace provides a number of methods in its libraries that can be used to generate useful numerical information. Several of these features are described in this section.

5.5.1 Time flies

There are many signs that computers keep track of the time. Most likely, the computer you use displays the current time of day somewhere on your screen as you work. Your computer can tell you when each of your files was created and last modified. While your web browser downloads large files, it probably displays an estimate of how much longer it will take before the process is complete.

The `sys` module in Grace provides a number of methods that can be used by a programmer. The one that we are interested measures the time elapsed during a program. To use this method, you must import module `sys` and then evaluate `sys.elapsed`, which will return a `Number` indicating the number of seconds since the program started executing.

For example, if you run a program containing the following code

```
import "sys" as system
...
print "{system.elapsed} seconds have elapsed since this program started"
```

Grace might produce the following output

```
0.005 seconds have elapsed since the program started
```

²Much of the time we use the same name before and after the `as` keyword. Thus we could write `import "math" as math` and then we could write `math.random`. We didn't do that here to emphasize the independence of the string and the identifier name.

```

dialect "objectdrawDialect"
import "sys" as system

// A program to measure the duration of mouse clicks.
object {
  inherits aGraphicApplication . size(400,400)
  // coordinates where message should be displayed
  def textLocation:Location = aLocation.at(30,50)

  // When the mouse button was depressed
  var startingTime: Number

  // Used to display length of click
  def message:Text = aText.at(textLocation)with("Please depress and release the mouse")
    on(canvas)

  // Record the time that the button is pressed
  method onMousePress( point:Location ) -> Done{
    startingTime := system.elapsed
  }

  // Display the duration of the latest clic
  method onMouseRelease( point:Location ) -> Done {
    message.contents :=
      "You held the button down for {system.elapsed - startingTime} seconds"
  }

  startGraphics
}

```

Figure 5.6: Grace program to measure mouse click duration.

The elapsed method can be very helpful in measuring short intervals of time within a program. This is an ability we will need in many programs in the following chapters. To use elapsed to measure an interval, we simply ask Grace for the time at the beginning of the interval we need to measure and then again at the end. The length of the interval can then be determined by subtracting the starting time from the ending time.

As an example of how to do such timing, a program that will measure the duration of a click of the mouse button is shown in Figure 5.6. The program first uses `system.elapsed` in the `onMousePress` method. It assigns the time returned to the variable `startingTime` so that it can access the value after the mouse has been released. In `onMouseRelease`, the program computes the difference between the time at which the mouse was pressed and released. A sample of what the program's output might look like is shown in Figure 5.7. **FIX!** ► *picture not right* ◀

5.5.2 Sines and Wonders

If you review all the things you have learned to do with numbers in Grace, you should be unimpressed at best. Think about it. For just \$10 you could go to almost any store that sells office or school supplies and buy a pocket calculator that can compute trigonometric functions, take logarithms,



Figure 5.7: Sample output of the ClickTimer program

raise any number to any power and do many other complex operations. On the other hand, with Grace and a computer that probably costs 100 times as much as a calculator, all you have learned to do so far is add, subtract, divide, and multiply. In this section we will improve this situation by introducing a collection of methods that provide the means to perform more advanced mathematical calculations.

FIX! ► *No absolute value at the moment, fix when added to Number – also sort* ◀

Absolute values

The method `absValue` can be used to compute the absolute value of a number. The expressions

```
math.absValue( 17 )
```

and

```
math.absValue( -17 )
```

both yield the int value 17, while the expressions

```
math.absValue( -34.2 )
```

and

```
math.absValue( 34.2 )
```

both yield the double value 34.2.

The `absValue` method is sometimes used to test whether numeric are nearly equal. If you test to see whether two `Number` values are exactly equal using `==`, the test may yield `false` even when the values being compared should be equal. This occurs because the computer only records about 15 digits of any double value. For example, although by definition

$$a = \sqrt{a} \times \sqrt{a}$$

the Grace expression

```
3 == math.sqrt(3)*math.sqrt(3)
```

will evaluate to `false`. Since `sqrt(3.0)` produces a value that only approximates $\sqrt{3}$, the product `Math.sqrt(3.0)*Math.sqrt(3.0)` evaluates to 2.9999999999999996. Somewhat confusingly if you execute


```

if ((sqrt(3) * sqrt(3)) == 3) then {
    print "same"
} else {
    print "different: value is {sqrt(3) * sqrt(3)}"
}

```

the system will print out `different : value is 3` because the system rounds the answer before printing.

To deal with such inaccuracies, it is often better to test whether two values are approximately equal rather than exactly equal. If `a` and `b` are two numeric values, then a test of the form

```
if ( math.absValue( a - b ) < EPSILON )
```

where `EPSILON` is a constant with an appropriately small value, is a simple way to test for approximate equality.

Exercise 5.5.1 *Below, you will find examples of several well-known mathematical equalities and inequalities. Using the methods discussed in this section, translate each of these mathematical statements into a Grace expression. Each of the expressions you write should produce a boolean as a result. Given that the mathematical statements are facts, all these expressions should evaluate to true. In fact, however, as a result of the limited precision with which computers represent double values, some of the formulas you produce may evaluate to false. You do not need to make any effort to address these potential “mistakes” in your answers.*

For example, if the question asked you to convert the “Triangle inequality”:

$$|x + y| \leq |x| + |y|$$

the correct answer would be the Grace expression:

```
math.absValue( x + y ) <= math.absValue( x ) + math.absValue( y )
```

Convert each of the following mathematical statements into Grace expressions:

a. *The reverse triangle inequality -*

$$|x - y| \geq ||x| - |y||$$

b. *the formula for the total return t received when p dollars are invested for y years at an annual rate r with interest compounded monthly -*

$$t = p(1 + r/12)^{12y}$$

c. *The triple angle identity for the cosine function:*

$$\cos 3t = 4 \cos^3 t - 3 \cos t$$

d. *A version of the half-angle identify for the sine function:*

$$|\sin \frac{t}{2}| = \sqrt{\frac{1 - \cos t}{2}}$$

e. *The exponentiation formula for the log function (use the natural log):*

$$\log x^p = p \log x$$

5.6 Strings

By now it should be clear that Grace programs can manipulate many kinds of data and that the language classifies the different kinds of data it can manipulate into types. Also, it is undeniably clear that one of the most important types of data that computers process is text. Text-based programs such as word processors and e-mail applications are among the most important and widely used applications on personal computers. Therefore, it should not be a surprise that Grace has a type for manipulating textual data. This type is named `String`.

We have been using `String` values since our very first example programs in Chapter `Chap::firstSip`. Quoted pieces of text like the message

```
"I'm Touched"
```

which appeared in those first programs are literals describing values of the `String` type. The type `String` is immutable, just like `Number`. Thus there are no mutator methods defined on `String`.

We have seen that Grace provides useful facilities for including the results of calculations in a string. In Figure 1.3, we saw that

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

resulted in evaluating `square(7)`, converting the answer 49 to the string `"49"`, and then inserting it into the string so that the program printed

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

In general, when an expression is surrounded by curly braces in a string, the expression will be evaluated, converted to a string by applying the method `asString` and then inserted into the surrounding string. This operation is known as *string interpolation*.

A useful operator provided for strings is the concatenation operator, written `++`. It is used to “glue” two strings together to form a new one. This can be helpful when a string is too long to fit on a single line. For example:

```
print ("This string is way too long to fit on a single line, so we must "++  
      "break it into two pieces.")
```

A couple of points about this example. First, notice the blank at the end of the string on the first line of the statement. That is needed so that we don’t end up with `”mustbreak”` in the combined string. Second, notice that we have surrounded the concatenation of the two strings by parentheses. That is necessary so Grace knows where it should stop printing. The general rule is that whatever is being printed should be surrounded by parentheses. However, if a single literal string is being printed then the parentheses may be omitted as it the double quotes mark the beginning and end of the string.

The fact that Grace considers `String` a type implies that we can do several things with `Strings` that Grace allows us to do with any type. We can use `Strings` as parameters. We have done this in both `aText` constructions and invocations of the `print` method. We can also define instance variables of `String` type and use assignment statements to associate values with these variables.

To illustrate the potential value of using `String` variables, consider the program shown in Figure 5.8. This program implements a simple interface one might use to practice encoding information in Morse code.

In Morse code, each letter of the alphabet is encoded using a sequence of long and short signals. These signals are called dashes and dots because a standard way to represent the signals on paper is to draw a dash for a long signal and a dot for a short signal. Thus,

```
. . . - - - . . .
```

```

// A program to display Morse code entered by pressing
// the mouse as dots and dashes on the canvas.

dialect "objectdrawDialect"
import "sys" as sys

object {
  inherits aGraphicApplication . size ( 400, 400 )
  // where to show codes
  def displayPosition : Location = aLocation.at( 30, 30 )

  // Minimum time (in seconds) for a dash
  def dashTime: Number = 0.2

  // String to hold sequence entered so far
  var currentCode: String := "Code = "

  // Time when mouse was last depressed
  var pressTime: Number

  // Text used to display Morse code on canvas
  def display = aText.at( displayPosition ) with ( "Code = " ) on (canvas)

  // Record time at which mouse was depressed
  method onMousePress ( point: Location ) -> Done {
    pressTime := sys.elapsed
  }

  // Add . or - depending on how long mouse held before release
  method onMouseRelease( point: Location ) -> Done {
    if ( (sys.elapsed - pressTime) > dashTime ) then {
      currentCode := currentCode ++ " -"
    } else {
      currentCode := currentCode ++ " ."
    }
    display .contents := currentCode
  }

  startGraphics
}

```

Figure 5.8: A program to display Morse code as dots and dashes

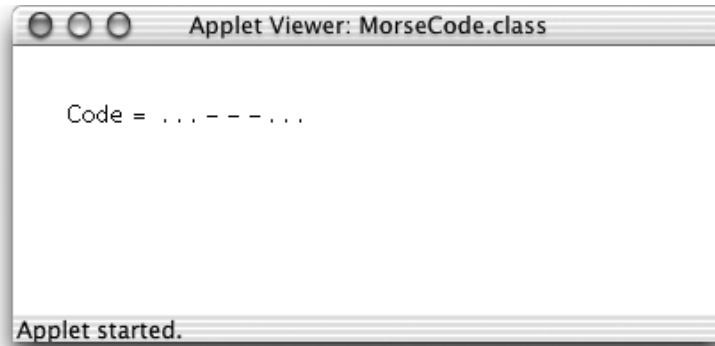


Figure 5.9: Sample of Morse code program display

represents a message composed of three short signals followed by three long signals followed by three short signals. This happens to be the Morse code for the distress signal “SOS”.

The program in Figure 5.8 translates short and long presses of the computer’s mouse button into a sequences of dots and dashes. If someone running the program used the mouse to enter the sequence discussed above, the program would display the output shown in Figure 5.9 The idea is that a person trying to learn Morse code could use the mouse to practice signaling and then look at the sequence of dots and dashes the program displays on the screen to see if they got it right. This means that we don’t need to worry about which sequences of dots and dashes go with which letters of the alphabet to write the program. All we need to know about Morse code is that a short signal is a dot and a long signal is a dash.

The program uses a String variable named `currentCode` to keep track of the message it should display on the screen. Initially, the display will simply be

```
Code =
```

Then, each time the user clicks the mouse, the program will add a dot or a dash to the message. If the first thing the user does is a short click, the message will become

```
Code = .
```

If this is followed by two long clicks, the message will become

```
Code = . - -
```

and so on.

To add a dash to the value of the variable `currentCode`, the program uses an assignment statement of the form

```
currentCode := currentCode ++ " -";
```

which can be found in method `onMouseRelease`. This statements looks like and acts a great deal like the similar assignment we have used to increment number variables

```
count := count + 1;
```

It takes the existing text associated with `currentCode`, adds a dash to it, and then tells Grace to make this the new value of `currentCode`.

The program uses the `sys.elapsed` method described in Section 5.5.1 to access the current time when the mouse is pressed and released. It compares the time that elapses between these two events to a value, `dashTime`. If the elapsed time is greater than this, it executes the assignment shown

above to add a dash. Otherwise it uses a similar assignment to add a dot. Either way, after the mouse is released it uses `contents :=` to update the `Text` object named `display` to hold the extended sequences of dots and dashes.

Like all other types, `String` supports `==` and `asString`.³ `Grace` provides many other accessor methods to operate on `Strings`. For example, `currentCode.size` will produce a `Number` value telling how many characters long a `String` is. We will consider these functions and other aspects of using `Strings` in more detail in Chapter ??.

5.7 Chapter Review Problems

Exercise 5.7.1 *Write the code necessary to define a `Number` called `num`, initialized to 5, and a `Location` called `point`, initialized to (50, 50). What are the differences in the definitions? Be sure to include the type information in each definition.*

Exercise 5.7.2 *Fill in the condition for the following code. Assume `myNumber` has been declared as a `Number`.*

```
if ( // myNumber is evenly divisible by either 3 or 4 ) then {  
    // do something  
}
```

5.8 Programming Problems

Exercise 5.8.1 *Write a program that displays the number of seconds that have elapsed between two clicks of the mouse. The program should not display a value after the first mouse click, but should display a value after every subsequent click of the mouse.*

Exercise 5.8.2

- a. *DNA stands for deoxyribonucleic acid and is a very long polymer of four different types of nucleotide bases. These four bases are Guanine(G), Adenine (A), Thymine (T), and Cytosine(C). It is the sequence of these bases that uniquely determine DNA. Write a simple program called `DNAGenerator` that may impress any friends of yours with interests in biology. This program will generate a strand of DNA by randomly adding one of the the four letters “G”, “A”, “T” or “C” each time the mouse is clicked. It also displays the generated DNA strand as shown in Figure 5.10.*
- b. *Modify your program so that no more than 20 bases are added to the DNA strand.*

Exercise 5.8.3 *In Morse code, a single dot is used to represent the letter “E”, a single dash represents a “T”, and the letter “D” is represented by a dash followed by two dots. This means that the sequence*

-. .

³It might be hard to see why `String` would need `asString`, but it would be automatically called if a string variable is interpolated in another string.

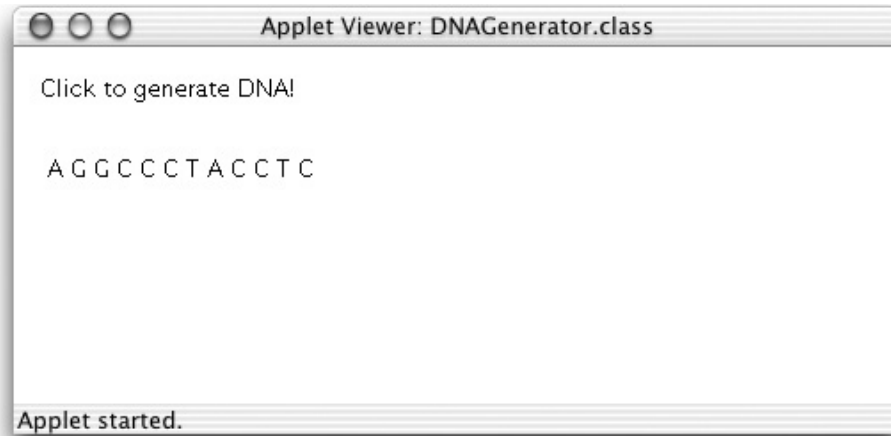


Figure 5.10: Display for DNAGenerator

has two possible interpretations. It could be a single letter “D” or it could be the word “TEE”.

The lengths of the pauses between dots and dashes are used to resolve such ambiguities. If the pause between a pair of dots is short, then the two signals are treated as part of a single letter. If the pause is long, the signals are treated as parts of separate letters. To make it possible to represent such long and short pauses in our graphical representation of sequences of dots and dashes we might represent the sequence for “D” as

- . .

and the sequence for “TEE” as

-

That is, we would put more spaces between pairs of symbols if the pause between them was longer. Thus, the signal for “TED” would look like

- . . . - . . .

Modify `MorseCode` so that it adds extra spaces for long pauses. Assume a pause is long if its duration equals (or exceeds) that of a dash.