# Teaching *Java: An Eventful Approach*

Kim Bruce, Andrea Danyluk, and Tom Murtagh

July 7, 2006

# Contents

**0 Introduction and Overview** **5**

0.1 Pedagogical Principles . . . . . 5

0.2 Sample Programs . . . . . 8

0.3 Programming Exercises . . . . . 8

0.4 How to Prepare to Teach with These Materials . . . . . 9

0.5 Summary . . . . . 9

**1 What is Programming Anyway?** **11**

**2 What's in a Name?** **13**

**3 Working with Numbers** **15**

**4 Making Choices** **17**

**5 Primitive Types, Operators, and Strings** **19**

**6 Classes** **21**

**7 Control Structures** **23**

**8 Declarations and Scope** **27**

**9 Active Objects** **29**

**10 Interfaces** **31**

**11 Graphic User Interfaces in Java** **33**

**12 Recursion** **37**

**13 General Loops in Java** **39**

**14 Arrays** **41**

**15 Multidimensional Arrays** **43**

**16 Strings** **45**

**17 Inheritance** **47**

**18 Exceptions** **51**

**19 Streams** **53**

**20 Searching and Sorting** **55**

**21 Introduction to Object-Oriented Design** **57**

# Chapter 0

# Introduction and Overview

The purpose of this document is to provide advice on how to teach the material in our textbook, *Java: An Eventful Approach*. Because the approach taken in this text is different from that in most other texts, we decided it would be useful to share some of the insights that we have gained by teaching this material over the last five years.

We include a separate discussion of each of the chapters of the text book. In this chapter, however, we present some of the general principles that lay behind our approach to the material as well as discuss other items related to the way we integrate programming in the course.

## 0.1 Pedagogical Principles

The preface of the text, in particular Section 0.3, explains the benefits of the primary innovative aspects of this text:

1. A graphics library for an objects-first approach
2. Event-driven programming
3. Objects-first
4. Concurrency early

We urge you to read about those aspects there, as, rather than repeating that discussion here, in this document we will focus on the practical aspects of teaching this material.

We begin by listing several principles that have guided our development of this material and that influence the way we approach our teaching:

1. Strive to use motivating and fun examples whenever possible.
2. Present new concepts and constructs via simple, concrete examples first. Move from the particular to the general, rather than the other way around.
3. Present explanations at the right level of abstraction. Too many irrelevant details too early confuse students.
4. Practice a "just-in-time" approach to introducing features so that students know why they need to know something.

5. Provide syntactic support early in the semester. This provides scaffolding so that students can focus on the important principles they need to learn, rather than be overwhelmed with less important details.

6. By the end of the course, provide students with the information necessary to remove the scaffolding and use standard Java constructs.

We discuss these in more detail in the rest of this section.

**Strive to use motivating and fun examples.** Students used to be excited if they could succeed in getting the computer to print out tables of numbers: multiplication tables, bowling scores, etc. But our students today have always had access to computers, and to them computers are no longer amazing devices. Instead, they are simple-to-use devices that have taken on the role of household appliances.

Most students have never used a command-line interface to interact with computers. They are used to using the mouse to navigate, using buttons and menus to determine what actions the program is to take, and typing in relatively free-form information into portions of a window to communicate information. It is the rare program that is started and then runs to completion, providing a final answer. Instead users interact with a program as it runs.

This is one reason that we decided to use event-driven programming with visually interesting programs in our text, in our classroom lectures, and in our assignments. We use graphics extensively and introduce animations relatively early in the text. We use event-driven programming because the sample programs can be made interactive, helping to make them more interesting as well as easier to debug. We have found that all of these features make programs more interesting to students while supporting the introduction of the important concepts of object-oriented programming. We strongly urge you to use and create similar programs to motivate your students.

**Present new concepts and constructs via concrete examples first. Move from the particular to the general, rather than the other way around.** We have found that novices generally need very concrete examples before they are able to understand new concepts. We generally try to present a problem that students aren't yet able to solve well, show how the use of a new concept or construct makes it possible to solve the problem, and then abstract away to the general case, typically ending by showing other applications.

Many times we have found it helpful to take a problem through several iterations, obtaining better and better solutions.

**Present explanations at the right level of abstraction.** Computer science is a field that emphasizes building appropriate abstractions. At what level of abstraction should we explain concepts to students? We could certainly begin by talking about memory, logic gates, and circuits, and work our way up to micro-instructions, machine language instructions, and then to higher-level instructions. However, if the goal is to teach students how to program in a high level language like Java, then it is easier for students to understand concepts explained at a level that better matches the language that they will be using. Just as it is not necessarily helpful to discuss the way a carburetor or fuel injection system works when teaching someone to drive, it is not necessarily helpful to discuss low-level features of a hardware or software system when trying to teach novices to program.

As a result, while we teach event-driven programming, we never discuss the event queue that underlies the implementation of event-driven programming. Rather than discussing pointers and bits stored in memory locations, we instead discuss associating names with values. We will give additional examples of this when we discuss how we teach the different chapters of the text.

**Practice a "just-in-time" approach to introducing features.** It is tempting to explain all of the facets of a concept when introducing it to students. However, providing all of the details may overwhelm a student. As a result we generally try to provide just enough detail at the beginning so that a student can use the concept in simple examples. We introduce more details as appropriate.

Thus we introduce declarations of variables in Chapter 2, but don't talk about the scope of variables, private vs. public, and related concepts until Chapter 8. We introduce `while` loops in Chapter 4, but don't talk about `for` or `do ... while` loops until Chapter 13, just before the introduction of arrays. The `String` data type is introduced in Chapter 5 (and literal strings are used even earlier), but most operations on strings are not discussed until Chapter 16.

Postponing these details lets us get to more interesting material faster, while not burdening students with having to choose between too many alternatives before they have enough experience to make good choices.

**Provide syntactic support early in the semester so that students can focus on the important principles they need to learn, rather than be overwhelmed with less important details.** The objectdraw library contains the class `WindowController`, an extension of `JApplet` that provides enough hooks that students can write event-driven programs using mouse actions without having to learn about and declare listeners. Mouse-handling methods provided in that class take parameters that represent locations on the screen, rather than more complex events that must be disassembled to get information about the location of the mouse action. An exceptionless `pause` method is provided with the `ActiveObject` class to replace the `sleep` method in `Thread` so that animations can be presented early, without the need to teach exceptions first.

These items are provided to reduce the clutter of language constructs that distract students from the key principles that we are trying to get across. As teachers we wish to present new concepts as simply as possible so that students can focus on the concepts rather than less important syntax that is not as relevant in simple contexts.

**Provide students with the information necessary to use standard Java constructs by the end of the course.** While we provide syntactic support for students early in the course, our goal is for students to understand the standard Java programming style. Thus while we provide simple mechanisms for event-driven programming with mouse actions in the early chapters, in Chapter 11 we teach students how to associate listeners with GUI components and use the standard Java event-handling methods. The early, simplified mouse-event handling teaches students the event-driven programming style used in standard Java. As a result, students later find it easy to program using the somewhat more complex syntax used with standard Java event handling covered in Chapter 11.

While students can take the objectdraw library with them and use it in non-commercial projects, Appendix D explains how to achieve similar results using standard Java. Covering this material is optional. We found that students (and instructors) like to know how to write the same sort of programs in Standard Java. Much of that information will not make sense until after students have covered topics like exceptions.

When you compare what has to be done in standard Java to simulate what we have provided in the objectdraw library,[1] we think you'll agree that there are great pedagogical advantages to hiding some of these complexities for novices. Because we typically do not use the objectdraw library in our second course, our students do learn the standard Java constructs at that point. You may wish to do the same or teach these constructs at some other point in your curriculum.

## 0.2 Sample Programs

We feel that students will learn more if they see as many different examples as possible. As a result, we have tried to present different programs in our lectures from those used in the text. The on-line materials for this course include a number of such examples. We expect that a student who has understood the examples presented in class will generally be able to understand the examples in the text on his or her own.

In the discussions of how we teach each chapter we will often refer to the on-line class examples. Instructors may wish to use our examples or create similar examples of their own. Occasionally it will be helpful to cover some of our particular class examples as preparation for some of the more substantial programming exercises discussed below, so be aware if you are using our suggested laboratories.

## 0.3 Programming Exercises

Programming is not a spectator sport. One cannot become a good golfer or tennis player just by watching coaches. Similarly, students cannot learn to program just by listening to lectures and working out small examples. They need to engage in writing more substantial programs.

While almost all of the chapters in the text end with a section of programming problems, these are designed to be relatively short exercises that test skills introduced in the chapter. Students should also write successively larger programs as the course proceeds.

Some courses using this text will have scheduled or in-class laboratories, while others will have students work on programming exercises on their own time. Nevertheless, we will refer to the larger programming exercises discussed below as laboratory exercises in order to distinguish them from the "Programming Problems" included at the end of chapters.

We have included among the on-line materials a number of laboratory exercises that we have found useful in teaching from this text. The labs, especially early in the term, often have two or more parts. The first part is usually designed to be a relatively simple variant of examples that are in the text or presented in class. This part of the exercise is designed to ensure that the students understand the basic constructs that have been introduced. Later parts of the exercise are designed to have the student explore the concepts in more depth and gain practice in putting together successively more complex programs.

Especially early in the term, we often provide students with "starter" Java program files. These program files might include import statements, the class header, and headers for many of the methods, while omitting variable declarations and method bodies. While steadily less support is

[1]See Appendix D of the text.

given as the semester proceeds, we have generally felt that it is better to provide students with support for the more mechanical aspects of the programming so as not to distract them from the more challenging parts that we wish them to focus on. After they have written several programs, we expect them to fill in those more mechanical pieces.

We have taught with these materials in scheduled laboratories staffed with faculty and assistants assigned to the course who are knowledgeable about the course and the exercises. We are happy to provide students with support so that they do not get stuck for a long time on a technical detail. Being present in the labs also gives us the opportunity to see what the students are having trouble with, so that we can adjust lectures to compensate when necessary.

How to grade labs is always a challenge. We want students to help each other and get whatever help is necessary from the instructor and assistants. As a result we decided to break our assignments into two categories: "regular programs" that count only as a small portion of the final grade and for which students can receive more help, and "test programs" that are treated more like take-home exams. We encourage you to think about similar ways of encouraging students to help each other while still obtaining enough information for assessment of student understanding.

## 0.4 How to Prepare to Teach with These Materials

We urge you to at least skim through the advice in the rest of this document well before you start teaching from the text. Obviously you should write a good number of programs using this library before starting to teach with our materials, but it also helps to have a good idea of how the whole course fits together before getting started with it in the classroom.

One of the most important things to do in teaching with these materials is to find the most appropriate pace to use with your students. The text includes a number of topics not normally covered in most introductory courses, e.g., event-driven programming, the use of GUI components, and animations via concurrency. However, we find that because these topics are used to reinforce other ideas, they don't take up a lot of extra time. That is, other topics can be presented a bit faster because of the things learned with these extra topics.

Nevertheless, we urge you not to be overly ambitious the first time you teach from our text. If your current introductory course only gets through one dimensional arrays in a term, be satisfied if you get through the first fourteen chapters of this book. Different institutions have terms of different lengths and students with different ability levels. Adjust the pace so that it makes sense for your students and we believe you will find this a successful way of teaching students to program in an object-oriented style.

Our students typically have one laboratory exercise due each week. You may find that our sample on-line labs cover too much too fast for your students. We have very good students, very short semesters (12 weeks), and our students only take 4 courses per term. You may find it better to stretch some of the labs over two weeks or to increase or decrease what the students are expected to accomplish in the labs. Again, you should calibrate the labs to be roughly comparable in difficulty with previous versions of your introductory course.

## 0.5 Summary

We hope that you will enjoy teaching with our materials, and hope that you find this instructor's manual helpful. The main website for the text is at `http://eventfuljava.cs.williams.edu/`.

If you have any suggestions for improvements of any of our materials, please write to us at `eof@cs.williams.edu`.

Chapter 1

# What is Programming Anyway?

In this and successive chapters we will explain to you, the instructor, how we approach the material in each chapter of the textbook in class, and how we fit it in with our labs. Obviously every teacher has a different style, but we hope that the description of what we do will be helpful.

The primary goal of this chapter is to give students a basic understanding of what it means to program a computer. We approach this goal in two ways. First, we discuss the abstract notion of an algorithm, providing examples that are not computer related. Second, we introduce just enough details of Java programming to take the students through the process of entering, running, and even debugging a simple program. We hope that this material will enable students to understand that while writing a program is "merely" a matter of telling a computer what to do, it is quite different from telling a computer what to do by clicking a button or selecting a menu item. At the end of this chapter, students should at least understand that a program is written at one point in time, compiled at another, and finally run after the first steps are complete.

In the process of exploring the basics of how a program is written and run, we also introduce the general structure of a class extending `WindowController` and the basics of the primitives for drawing supported by our library. We explain the upside-down graphics coordinate system, discuss constructors for several graphics classes, and discuss simple event-handling using methods like `begin` and `onMouseClick`. Rather than explaining each of these separately and in great detail, we introduce the ideas to students via a series of simple examples. The two examples that we use in class are the `MakeBox` and `UpDown` examples. These have the advantage of not using variables, which are not introduced until the second chapter.

We cover the material in Chapter 1 relatively quickly in class, trying to get across the main ideas, and assuming that students will really learn the material by working through our first lab. Like many instructors, we tend to spend more time than we would like on administrative details in the first lecture. We also spend time discussing the notion of an algorithm (and why it is hard for humans to write them). Finally, we quickly go over the ideas of writing and compiling programs.

The first lab includes a very extensive handout that first walks students through creating a picture using a program that we call `NotPhotoshop`. `NotPhotoshop` allows students to exercise constructors and methods using Java syntax. However, rather than writing code, the student initiates execution of a method or construction of an object by clicking on a button that brings up a dialog box with slots for all the parameters needed for the method. Students draw the picture using `NotPhotoshop` and then write a Java program that draws the same picture. We have found this to be very effective in helping students understand concretely what each of the commands actually does. Moreover, `NotPhotoshop`'s user interface presents the correct Java syntax and requires the same parameters as the program constructs, so it makes the transition to writing Java much easier.

The handout that we distribute for this first lab also contains extensive directions on how to enter programs in the IDE being used and how to run programs. Students seem to find it easier to follow these directions than to follow oral directions on what to do. You will need to modify the version of the lab kept with our materials to include a description of the IDE that your class is using. We encourage you to modify any and all of the labs that we provide in order to make them more relevant to your own course.

In general students find this lab to be very helpful in getting familiar with creating graphics using the objectdraw library and running Java applets in the development system, as well as giving them a much more concrete feeling for Java and the results of executing code.

# Chapter 2

# What's in a Name?

The most important idea in this chapter is the notion of associating a name with an object. We urge you to look carefully at how we discuss instance variable declarations and assignment statements, as we do this quite differently from most authors.

Declarations are used to introduce names that may be used to refer to objects of particular types. We do *not* tell students that declarations allocate memory to store a value! We do *not* tell students that variables hold a pointer to an object!

Assignment statements are used to associate names with values. We do *not* tell students that they result in storing something in the memory slot associated with the variable, and we certainly don't tell them that the value stored is a pointer or address in memory.

This will likely seem strange to you if you have had experience with languages like C, C++, Pascal, or other imperative languages. But an object-oriented language with a semantics like that of Java supports a very different higher-level conceptual model.

How do we explain this to students? First, we try to avoid saying "assign a value to a variable". Instead we say that an assignment is the association of a name with a value. More metaphorically, we tell the students that they can think of a variable as something that can be attached to an object (think of a sticky name tag). An assignment statement of the form

```
x = someObj;
```

results in variable `x` being associated with (stuck on) the object (let's call it obj) that is associated with identifier `someObj`. Thus the sequence of assignments:

```
x = someObj;
y = someObj;
```

results in both of the variables `x` and `y` being associated with obj. I.e., they both end up sharing a reference to obj.

Thus if we execute:

```
x.move(10,20);
```

the obj is moved by (10,20). Because `y` is also associated with obj, the object `y` references has moved (after all, they are both obj).

If we then execute:

```
x = otherObj;
```

where `otherObj` has a value different from obj, then the variable `x` now refers to that different object. As a result, sending a `move` message to `x` will no longer have any impact on the object associated with `y`.

If you consistently explain variable names as moveable tags that are associated with objects, and avoid talking about sticking things in slots in memory, everything is much easier. Those of us who grew up with pointers and know the internal representation of objects is as pointers have a very difficult time not talking about that. However, the association metaphor is much, much easier for students to understand.

The same descriptions work for primitive values like integers. Because they are invariant, there is no difficulty with imagining a single integer "2" with lots of variables associated with it. For example,

```
x = 2;
y = x;
```

results in the variable `x` being associated with 2, and then `y` also being associated with 2. The statement

```
x = x + 1;
```

results in first evaluating `x + 1` as 3, and then associating `x` with 3. Notice that `y` remains associated with 2 as the assignment statement has no impact on the value `y` is associated with.

The only difference between constants and variables is that a constant always refer to the same object (even though its state may change as the result of sending it messages), while a variable may refer to different objects at different times, i.e., what it refers to *varies*.

We again present the material in this chapter via examples. In lecture we use the `Spirograph` program as a first example in which we use an instance variable to remember a location. This example is also the first to actually use the value of a parameter. It also gives us the excuse to talk about the class `Location`, representing the coordinates of a point on the canvas. It is necessary to distinguish between a location and a graphic image (e.g., of a point) on the screen.

The example program `Scribble` is a very minor variant of `Spirograph` that is obtained by updating the starting position every time the mouse is dragged. We emphasize the importance of the changes in program behavior that resulted by adding one line to the `onMouseDrag` method.

Our practice has been to go through these first two chapters relatively quickly, assuming that students will pick up a lot of the details by working through the handout for the first lab and by reading the book. We know that students resist reading textbooks, but they will be even more resistant if they discover that everything they are reading has already been covered in class. Thus we make clear that there is material in the text that they need to know and is not covered in the lecture. We reinforce our expectations that they read the text by asking for questions on the reading (and previous lecture) at the beginning of class.

# Chapter 3

# Working with Numbers

This is a relatively straightforward chapter. In class we continue to introduce concepts via a series of examples, depending on students to read the chapter for the gory details. We begin by using accessor methods that provide numeric information.

We usually start with something like the `CrossHairs` example in which we need to extract the x and y coordinates of the location of the mouse (provided by the system as a parameter to mouse-handling methods) and then update only one of the coordinates of the endpoint of each line, leaving the other fixed.

We point out the differences between primitive and object values, though we discuss only `int` and `double` at this time. Here the key ideas to get across are that you cannot construct primitive values and you cannot send messages to them. Instead there are operations like +, *, etc.

We introduce `int` variables with the program `ClickCounter`. This is a good time to discuss the bizarre (to the students) statement:

```
x = x + 1;
```

Emphasize to students that this is not an equation, but instead is interpreted just like the earlier assignment statements. Working by example is often the easiest way to explain this.

`ClickCounter` also shows how integers and `String`s are displayed on the screen via `Text` objects. This is a good time to explain the operator "+" between strings and how concatenating a number with a string converts the number to a string before concatenation. The program `MouseMeter` provides another good example of constructing complex strings and displaying them on the screen.[1]

This is a good time to start introducing students to named constants. Two things are necessary to make something a constant:

1. include `static final` as part of the definition,

2. provide an initial value as part of the declaration.

It is also the case that `static final` variables can be initialized in the constructor of a class. However, at this point students don't yet know how to write classes aside from those that extend `WindowController`, and hence that have no constructor. We prefer to simply tell them what makes sense at this point rather than to give the full "legal" definition of when a constant may be initialized.

[1]We come back and study strings in much greater detail in Chapter 16.

An important consideration here is that in Java constants are not allowed to depend on anything created when a program starts up, aside from other constants. In particular a constant may not depend on `canvas`, which is not created until the program starts up. This means that students may not declare constants that involve any of our graphics primitives, such as `FilledRect`, whose constructor involves `canvas`. This is often a source of errors for students, so please emphasize this to your classes.

We work hard at getting students to use named constants, and take off points if they don't use them to make their programs more readable. Please note that in many of the sample programs in the text we have omitted the constant declarations to save space, though we use the constant names. The full declarations can be found in the on-line versions of these programs.

The chapter ends with a section on random numbers. This seemed to be the most logical place to put this material in the book, but we typically introduce this at the time it is needed in the course, usually just before a lab or demo program that uses them.

# Chapter 4

# Making Choices

The purpose of this chapter is to introduce students to making decisions using conditional statements. As usual we introduce the relevant concepts via a series of examples. One series that has proved popular among our students involves a series of (rather silly) basketball games. The advantage of these is that they provide lots of alternatives / improvements as you teach more concepts.

A first example is a simple modification of the `ClickCounter` program discussed earlier that just kept track of the number of times the mouse was clicked. This program draws an oval on the screen representing a basketball hoop. The user gains two points each time the mouse is clicked inside the hoop.

Two constructs must be introduced in order to write such a program: the `if` statement and the `contains` method, the latter of which is supported in all of the graphics classes. We do not introduce booleans immediately, just talk about conditions being expressions that return true or false. This first version can be enhanced with an "`else`" clause to print a message, "You missed", when the user misses the hoop.

A slightly more sophisticated version requires the user to drag a circle (representing a basketball) to the hoop before releasing the mouse and (hopefully) scoring. This provides the excuse to look at more interesting conditional statements. The `onMouseDrag` method contains a conditional (only drag the ball if the mouse was on the ball), while the `onMouseRelease` method now contains either a nested conditional or a condition containing a "&&".

This program (or another similar example) is a very important example for students to see and understand as we do lots of other examples that also involve dragging things around. The structure of the methods is quite important here, and will come up within other contexts (both of dragging other things around and also with leading students into understanding looping constructs). The key is remembering the location of the mouse in `onMousePress` and remembering to update it after each drag event. See the code in the text for details. The pattern of initializing a variable and then updating it each time an action is taken (e.g., responding to an event or executing a loop body) is something students will see in multiple contexts.

There are two ways of keeping track of whether or not an object (like the ball in the above example) should be dragged at each execution of `onMouseDrag`. The initially more straightforward way (illustrated by Exercise 4.3.2 in the text) involves checking each time whether the previous location of the mouse was over the ball, while the more elegant (and incidentally more efficient) solution is to check when the mouse is first pressed whether or not the mouse was on the ball, and remember that information in a boolean variable.

We always cover the second way, using it as an excuse to talk about the `boolean` data type, and often cover the first. However, if you do cover the first way (checking every time there is a drag event) do make sure that you use the previous location of the mouse as the parameter of the `contains` method, and explain why using the current location will not work (i.e., if the mouse is moved quickly, the new location of the mouse may no longer be on the ball after moving). See the code in Figure 4.6 and Exercise 4.3.2.

Sections 4.4 through 4.6 of the text take the students through a variety of other constructs using conditionals. These include multiple alternatives (resulting in the use of `else if`), the use of boolean operators like "&&", "||", and "!", and nested conditionals. Be sure to point out to students that boolean operations use short-circuit evaluation. They may have to be reminded of this later in the course in circumstances in which problems may arise (e.g., checking an array bound before using it). We have used a variety of examples to illustrate these ideas (including slowly building up the pieces of a pong game). See the examples on-line or make up your own sample programs.

The `Pong` game example illustrates a strategy that we occasionally use in class. We demo a program that uses constructs that we will not get to for another week or so, in this case `ActiveObject`. The advantage of doing this is that students are motivated to learn how to do the "cool" example, and we use that to slowly build up the pieces that they need to do it. Of course we could just go through a bunch of simple (and not very exciting) programs and then spring the final program on them after they know all of the constructs, but then they will have to wade through a fairly complex program, rather than seeing how it was built a step at a time.

Chapter 5

# Primitive Types, Operators, and Strings

We emphasize the difference between primitive and object types to students:

1. Primitive types have no constructors.
2. You cannot send messages to values of primitive types. Primitive types have operators instead.

The concept of equality between values is very hard for students. We spend some time working on the difference between "`==`" and "`equals`", trying to make clear that "`equals`" is nearly always the right choice for comparing objects unless you want to know whether two expressions refer to the same object. We emphasize that `Strings` should always be compared only with the `equals` method. Students keep forgetting this, so it is useful to remind them several times through the course.

We mention all of `int`, `long`, `float`, and `double`, but explain that, because of the way Java's libraries are set up, we will really only use `int` and `double`. When explaining about `double` and its operators, we emphasize the complexities of mixed-mode arithmetic. We also emphasize the different results of division with different types, e.g., 2/3 vs. 2.0/3.0. We call students' attention to precedence rules, though we don't talk about them much in class. Instead we encourage students to use parentheses if there is any danger of confusion.

We do not mention the object types `Integer`, `Double`, `Boolean`, etc. in this course. The only reason to introduce them is if at some point you create data structures containing elements of type `Object`. However, now even that is less necessary if you are using Java 5, as the system automatically converts back and forth between `int` and `Integer`, for example. See the comments in Chapter 14 about `ArrayList`.

Feel free to cover the material in sections 5.5 and 5.6 on mathematical functions and strings when they are needed in examples and homework. Students can also look up this material as it is needed. In general we tend to skim this chapter fairly lightly, as the material is not particularly exciting. We expect that students can pick up most of the details that they may need by reading it, as there is not much that is very complicated here.

# Chapter 6

# Classes

This is a key chapter. Students have been writing classes that are extensions of `WindowController` from the beginning of the course, but now they get to design their own classes from scratch.

The main new feature that students must learn is the constructor. They have been invoking constructors in the *construction* statements (e.g., `new FramedRect(...)`) since the first programs introduced, but they haven't written them yet themselves.

There are two main things we emphasize in the definition of a constructor. First, it plays a role very much like the `begin` method, except that it can take parameters (and, for graphics classes, usually will contain at least a parameter for the `canvas`, which, by contrast, is automatically available in extensions of `WindowController`). Second, there are some syntactic differences between writing constructors and methods. The constructor must always have the same name as the class being constructed, and there is no return type (e.g., `void`) in the header of the constructor. Aside from those distinctions, we tend to encourage students to think of constructors as being like the `begin` method.

Our initial examples of classes are usually geometric, and employ methods similar to those in the graphics library. We have used examples including a T-shirt and a basketball class. Consider motivating the introduction of a new class by talking about how the code would have to be written if there were no class (e.g., lots of instance variables that have to be manipulated to do anything). We talk about how having the class and its associated methods makes it easier to work with the abstraction (though of course you must do the initial work of writing the class).

An important feature of a good first example of a class is that it should make sense to have multiple instances. This is necessary because students often confuse classes and objects. Hopefully students' experiences in creating multiple objects from the graphics classes will make this easier to understand.

We emphasize that some methods, e.g., `move`, can be written by just sending similar messages to all of the pieces. Other methods, e.g., `contains`, require some extra thinking, and may not involve all of the instance variables. Finally, there are other methods like `moveTo` that can be quite complex if written out in detail. (We ask the students to tell us what the picture would look like if we just blindly sent `moveTo` to all of the pieces.) With the `moveTo` method we discuss how the `move` method already does something similar, and mention how nice it would be to take advantage of that existing code. Typically, we get a student to suggest using `move` by calculating how far each piece should move, though we may have to prod them a bit.

We like examples like these because the methods and their signatures (parameters and return types) are very familiar to the students. Moreover, some of the methods are quite simple, others take a bit more thought, and yet others require some real cleverness (or hard work).

When we invoke the `move` method from within `moveTo`, we typically first write it as `this.move` in order to introduce `this` to the students as a name for the object whose code is currently being executed and to be more parallel with their earlier experience that all messages must be sent to receivers. We then mention that occurrences of `this` as receivers may be omitted, but they may need to be written if, for example, `this` is passed as a parameter of another method call.

We also mention the use of local variables, and why it is better, where possible, to use them rather than instance variables, but we often put off a more detailed discussion of when to use parameters, instance variables, and local variables until we cover Chapter 8.

We also discuss overloaded methods, but try to get students to restrict their usage to cases where the methods do exactly the same thing. We attempt to reinforce this by encouraging students to write overloaded methods and constructors so that they all call another version of the method or constructor that contains the details of the code. For example, a `moveTo` method defined with a parameter of type `Location` will call the version that takes `x` and `y` parameters. We explain this by saying that it is easier to fix mistakes when there is really only one significant piece of code to be examined, even though there are multiple versions with slightly different parameters.

# Chapter 7

# Control Structures

The goal of this chapter is to introduce key concepts involving control structures, as well as to introduce `while` loops and, optionally, `switch` statements. While conditional statements were introduced earlier, this is the point where we bring together many ideas relevant to both conditionals and loops. This includes a more extensive discussion of boolean expressions as well as style guidelines for writing clearer control structures.

Note that the text does not cover `for` loops or `do`...`while` loops until Chapter 13, just before the introduction to arrays. This approach lets us focus on just one kind of loop initially so that students need not be concerned with selecting between loops until they are comfortable with the general concept of loops.

You will likely have noticed that all of our programs involving dragging a mouse around already involve repetition. However, we have a single method (`onMouseDrag`) which is called repeatedly, rather than introducing a special syntax for repetition.

We take advantage of the fact that we have already been doing repetitive actions to introduce `while` loops. Students are already familiar with some of the basics of loops from their experience with dragging. Executing the `onMousePress` method typically initializes a variable like `lastPoint`. Execution of the `onMouseDrag` method performs an activity (moving the object) and then updates `lastPoint` in preparation for the next execution of `onMouseDrag`.

We start by writing a class that does a repetitive drawing, like the example in the text that draws grass. In class we often draw railroad ties on a track. The idea is that the `begin` method initializes variables and does some drawing, but every time the mouse button is clicked, a further piece of the drawing is completed. The code of the `onMouseClick` method not only adds the piece of the drawing, but also updates a variable so that the next drawing will go in the correct place. We also add a conditional to ensure that we don't draw too much if we click too many times. So far this is not much different from the earlier drawing programs in its general structure.

We then illustrate how the program works, but complain about having to do all of the clicking to finish the drawing. At that point we show students that by replacing the `if` by `while` we can get the complete picture done with a single click, explaining the difference between the semantics of `if` and `while`.

The nice thing is that they have already seen all the key steps of designing a loop: initializing variables before the loop, adding a boolean guard to exit properly, and updating variables in preparation for the next loop. This makes it easier for students to understand the complexities of loops.

We also look at more variations of loops, including those where more than one variable is changed each time through the loop (e.g., in drawing a laundry basket) and nested loops (like the bricks example in the text). One could at this point talk about more variations of loops (e.g., priming loops, exiting in the middle, etc.), but we tend to prefer moving on to the next chapter and talking about animations, as that gives us a better context and motivation for discussing some of these complexities. Some of these details are also covered in Chapter 13.

We expect our students to read carefully the style guidelines for control structures and warn them that they will be tested on homeworks and exams to see if they can detect and correct bad style, and we will be marking them down if they use bad style in code. It is helpful to put up several examples of bad style and have the students figure out how to improve the code.

You may wish to put more or less weight on deMorgan's laws depending on how complicated the boolean expressions students encounter will be. If all students take a good discrete math course covering this material, you may choose to put it off until then.

The discussion of `switch` statements is optional. While `switch` statements can be very handy in certain very specialized circumstances (where you can list all the options as corresponding to `int`s), they are not that generally useful and the C-style syntax is very error-prone. If we discuss the `switch` statement, we warn them repeatedly not to forget the `break` statements.

Incidentally, this is usually the part in the course where we start introducing students to style guidelines (e.g., as in Appendix A). While there will still be some guidelines that will have to be introduced later, we would rather introduce the guidelines as early as they make sense. It can be harder to break students of bad habits than to teach them properly from the beginning. Students always complain about style guidelines, and it is harder to justify them with the relatively short programs that students write in this course. Nevertheless, we try to emphasize how all organizations have such guidelines and how it makes it easier for everyone looking at the code (including the programmer) to understand what is going on. If all else fails, we simply tell the students that we can't understand their code without well structured and commented code. Sometimes we tell students that we won't (be able to) help them with problems in their code unless it is well commented and formated. Of course, many IDE's have commands that will format the code, but students will often resist anyway, alas.

At about this time our students are starting to encounter their first serious problems in debugging their programs. You might find it helpful to take out some time in class (or if you have a scheduled lab) to discuss general notions of debugging. Here are some tips we give our students:

1. Plan ahead. Write a careful design of your program before ever sitting down at a keyboard and IDE. Make sure you understand why your program should work before ever approaching the computer. If you don't understand it, the computer certainly won't get it right! Generally students are most successful with designs if they avoid writing Java, but they find it hard to resist. We require students to bring more and more elaborate designs to lab sessions, and we grade these. This serves as an important incentive for students to do a good job on these.

2. Once the design is in hand, write and debug the code in small pieces. Your goal should be to get each piece to work before going on to the next part. Typing in everything at once will result not only in a program that doesn't work, but you won't have a very good idea where the problem is. If a small addition has resulted in the mistake, you are more likely to be able to find out what went wrong.

3. Localize the problem. If you've been building things up slowly, you probably already have a

good idea of where the problem is. If there is any question about where it is, narrow it down further before attempting fixes. Use `System.out.println` to print out intermediate values so that you can see precisely what is going on in the program.

You may find it helpful to write special code whose sole purpose is to test various parts of the program. Even though you will eventually throw that code away, it will have served a valuable purpose in helping you find your errors more quickly.

# Chapter 8

# Declarations and Scope

This chapter addresses a number of important issues that have been bubbling to the surface in earlier chapters. You can decide to go through these carefully in class or just skim them in class and have students read for the details.

Around this time in the course students should start seeing the need for private helper methods in classes they write. By deciding which are public and which are private they determine the public interface of the class. Because students were previously told that all instance variables should be private and all methods public, this is their first opportunity to really learn the difference between what those labels mean.

Students have seen identifiers declared as either instance variables, parameters, or local variables. They benefit from seeing a careful description of when each should be used. This can be covered either back in Chapter 6 or now. In either case, it is worthwhile emphasizing to students the importance of selecting the appropriate declaration. This is also a good time to emphasize to students issues of scope. Instance variables have scope of the entire program, while parameters have scope limited to the method, and local variables are restricted to the block in which they are defined. For example, if a local variable is declared in an `if` block of a conditional, then it is not available inside the `else` block.

It is also important to discuss what happens when an identifier is introduced in a block in which the identifier is already within scope. This creates what is known as a "hole" in the scope of the original declaration. We explain to students that when this happens with an instance variable and parameter, then the instance variable can be accessed by writing "this." in front of it. Some students find this liberating, expecially in constructors, where they can use the same name for the parameter and the instance variable it is used to initialize. Other students find it extremely confusing. We suggest hitting it lightly and then not using it much in examples so as not to confuse those for whom it is too confusing.

The chapter ends with a discussion of `static`. We don't make too much of this. Students will use static methods (for example the methods in the `Math` library), but they will never need to write their own static methods here.

This chapter presents a collection of important technical ideas that are grouped here for convenience. Think of it as a time out from presenting new constructs in order to get a deeper understanding of some aspects of the program constructs already introduced. You may teach this material in a single segment here or present the ideas when they come up in the earlier chapters.

# Chapter 9

# Active Objects

In this chapter we introduce students to the use of threads and concurrency in order to support animations. We do this for several different reasons. The two most obvious reasons are that programs involving animations are fun for the students and that animations provide a good place for students to get experience with loops, one of the topics introduced in the previous chapter.

However, those readers who are experienced with event-driven programming are likely aware that it is important that the event-handling thread not get tied up executing code that runs for any substantial amount of time (e.g., more than a few hundred milliseconds). The event-handling thread is the thread that executes the `begin` method and the mouse-handling methods in classes extending the `WindowController` class. If it is tied up running code, then it will not be available to respond to user actions or even to repaint the screen. This is why straightforward attempts to creat animations without separate threads run into significant difficulties – typically only the initial screen set up and the final configuration show up. As a result, threads are essential for serious event-driven programming.

It may seem completely crazy to introduce concurrent programming this early in a text designed for novices, but students don't realize that concurrent programming can be tricky. Because the world involves interacting objects that are operating in parallel, students see that as a natural way to be programming.

Of course, there are things that can go wrong. For example, two different threads can attempt to update the same variable at the same time. Handling this typically requires the use of synchronization blocks. We don't want to have to address those complexities at this point, so we restrict ourselves to programs where there is no danger of simultaneous update of variables by different threads.

The only thing that keeps us from using the Java `Thread` class directly in this chapter is that the `sleep` method throws a non-runtime exception, and we don't want to deal with exceptions until later (Chapter 18). As a result we introduce a new `ActiveObject` class that includes a `pause` method that does not throw exceptions (we implement it with a `sleep` method that does nothing when an interrupt is encountered).

If you have programmed with Java threads, you will find everything else about using `ActiveObject`s familiar. A class extending `ActiveObject` must have a run method. The run method is invoked by calling the `start()` method at the end of the constructor (or you can create the object and then later send the `start()` method to it). The `start` method sets up the machinery for the separate thread and then automatically invokes the `run()` method.

As usual we introduce extensions of `ActiveObject` via a series of examples, One of the important ideas in working with active objects is figuring out how to get the various objects to communicate with each other. It is relatively easy for the class extending `WindowController` to communicate with the class extending `ActiveObject`, as the active object is usually created by the `WindowController` extension, which can retain a reference to the active object in an instance variable. For example, it is common for the window controller to send a message controlling the behavior of the active object when triggered by a mouse action.

However, it is a bit tricker for the active object extension to send messages to other objects. Parameters are the basic way in which objects communicate with each other. In most cases, extra information that an active object will need to know is communicated through the parameters of the contructor. Recall that the `run()` method takes no parameters, so no communication can take place there.

We include a description of the standard Java class `Image` and the objectdraw class `VisibleImage` in this chapter because there are usually nice examples of animations that use jpeg or gif images – typically obtained over the web, though students can create their own.

This is generally a good time to take a moment and remind students that they have now seen three different kinds of classes.

1. Classes that extend `WindowController`,
2. Classes that extend `ActiveObject`, and
3. Classes that don't extend anything.

The first two, because they extend particular classes, expect methods with particular names. The first expects the `begin` method and the various mouse-event handling names, while the second expects a `run` method and needs to be started running by invoking the `start` method. The first group typically initializes instance variables in the `begin` method (or in the variable declarations), while the last two groups use constructors for the same purpose. We do not speak here in any detail about what "`extends`" means. Instead we put that off until Chapter 17. Here we simply lay out some rules for how each of these kinds of classes works.

When covering the material in this class, we typically also take time to go over some more complex examples of loops, e.g., one loop followed by another, nested loops, etc. We find that repeatedly asking students to help us design loops provides them a deeper understanding of how that design process works.

In our courses, we give students two lab exercises on active objects. The first, `BoxBall`, is primarily an exercise that reinforces the uses of parameters. [Alas, no matter how much we focus on parameters in class, understanding why and how they are used is still a problem for many students!] The second, `Frogger`, is a fun game that is the first fairly complex program that students write. It uses four different classes, two of which extend `ActiveObject`.

One warning, if you make up your own exercises or sample programs using active objects, please be careful that there is no danger of two different threads updating the same variable, as interference will almost certainly result.

# Chapter 10

# Interfaces

Introducing interfaces before subclasses is relatively unusual, but we think it is a very good thing to do. We chose to introduce interfaces at this point in the text for two reasons. The first is that introducing interfaces gives us the opportunity to talk about "dynamic method invocation" before dealing with the complexities of inheritance. The second, and more concrete, is that listener interfaces are necessary for using standard Java event handling, which we cover in the next chapter.

While the preparation for standard Java event handling determined the exact placement of the discussion of interfaces in the text, we believe there are strong pedagogical reasons for introducing interfaces early in a first course. One important reason is that interfaces present an early introduction to abstraction. When we say that an identifier has a type given by an interface, we are only claiming that it will respond to certain methods. We have no idea what the instance variables are or the details of the method definitions. We know that this is a good thing in that it keeps our code from being too dependent on certain implementation decisions. It also enables the value of an identifier to be an object from one of several different classes that implement that interface.

We talk about the "`implements`" clause in class declarations as specifying a contract. The class promises to provide method bodies for all the methods listed in the interface. As a result, objects from that class can be associated with identifiers with type given by the interface because any message name specified in the interface may be sent to the object.

In the text we illustrate this use of interfaces with classes representing happy and sad faces. In class we typically use a laundry program example where we randomly generate either shirts or pairs of pants to drag to the correct laundry basket. This takes an earlier laundry program lab on conditionals and makes it more interesting (and allows students to see how it could become more general with even more clothing items). This particular example works well for us because our students have earlier worked on a lab with square items of laundry, and have then seen a variant that uses t-shirts. This new variant allows students to see how interfaces can make a difference in a context that is already very familiar to them.

Having shown the students how the program works with both shirts and pants, we point out to them that while the general results of sending `move` to t-shirts and pants is similar, the concrete details are quite different. In particular, a different collection of instance variables get sent `move` messages in the two cases. This gives us an opportunity to emphasize the fact that each object is reponsible for knowing how to respond to a method.

When we see a message being sent to a variable with an interface type, we need not know before execution time which code is going to be executed. The message goes to the object, and the object determines which code is going to be executed by going to the set of methods provided by its class.

This notion of dynamic method invocation (i.e., the receiver of the message determines which method body is selected) is one of the key principles of object-oriented languages, and students need to understand it well. We believe that presenting it in the context of interfaces allows us to focus on this principle without cluttering the discussion with the complexities of inheritance. As a result, we urge you to make sure that your students get a good understanding here. We return to practice the notion of dynamic method invocation in both the recursion and inheritance chapters.

We find it helpful to reveal to students that we used interfaces extensively in the objectdraw library. For example all geometric classes implement `DrawableInterface`, while those that have height and width (all but `Line` and `Text`), satisfy `Drawable2DInterface`. This can also be a good time to explain to students how to use javadoc documentation. Appendix C of the text includes a relatively brief explanation of how that documentation works. It is put in an appendix so that the instructor can decide when to cover it.

Finally, while it may easily be postponed to Chapter 18 on inheritance, you may find it useful to cover the notion of "`extends`" for interfaces. It is presented initially as a simple way of avoiding repeating long lists of method names, but we then note that if `IE extends I` then an object from a class implementing `IE` can be used in any context expecting an object from `I`. This is the first time that students are exposed to the notion commonly referred to as subtype polymorphism. It is a slightly more complicated notion than dynamic method invocation arising from different classes implementing the same interface, but is clearly related.

In summary, it is useful to cover interfaces for several reasons. First, they are necessary to discuss listeners in standard Java event-handling. Second, they provide a notion of abstraction that allows a variable to hold objects from different classes at run-time. Because of this, student learn about "dynamic method invocation", the notion that an object is responsible for determining which code is executed when it receives a message. Finally, the notion of subtyping can be introduced here if you choose to talk about extending interfaces. Aside from the important topic of inheritance, covered in Chapter 18, we have now covered the key notions of object-oriented programming (though of course we have yet to cover a great many other notions relevant to general programming).

# Chapter 11

# Graphic User Interfaces in Java

At this point in the class, the students know how to do a lot of things, but the only forms of input that they know how to handle are mouse actions. One of the main purposes of this chapter is to provide them with other forms of I/O, including clicking on buttons, selecting entries from pop-up menus, and filling in text fields with values. This will require teaching students how to lay out GUI components and how to perform standard Java event-driven programming.

Because students have already been using event-driven programming with mouse events, they are already familiar with how to do event-driven programming. Moreover, we're not particularly interested in teaching our students how to do beautiful and complex GUI layout. As a result, we are able to provide them with a fairly simple approach to building user interfaces.

One word of warning. Our approach to building GUI components is based on the Java Swing library. If you accidentally use AWT components instead of Swing then your program will exhibit strange behavior. It is easy to leave off a "J" in a component like `JPanel`, so do be careful. Unfortunately many of the programs need classes from both the javax.swing and java.awt packages, so you may accidentally import a class from java.awt that you didn't want to use, and accidentally mix it with swing components. One way to avoid that is to never write

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

but instead import each class individually. E.g.,

```
import java.awt.ActionEvent;
```

The text starts out with text fields, but we placed that section first primarily for instructors who wanted to add text fields to their user interfaces without going into all of the other GUI components. We have typically covered a component like the `JComboBox` first. [The `JComboBox` looks like a pop-up menu.] Like the text field, one can use a `JComboBox` without having to handle the events generated by the component. For example, we use a program `ComboBoxDrawing` that places a geometric figure on the screen. The precise figure drawn is determined by the current selection in the combo box.

In teaching students how to add GUI components, we give them a checklist:

1. Create the component and initialize it if necessary. A `JComboBox` object needs to be created and then initialized by adding the labels with the `addItem` method.

2. Add the item to the content pane of the `WindowController` extension, and then validate the pane.

For the second item there is a difference between Java 1.4 and Java 5. In Java 1.4, one must obtain the content pane, add the item to that pane, and then validate the pane when all the components have been added. E.g.,

```
Container contentPane = getContentPane();
contentPane.add(figureMenu, BorderLayout.SOUTH);
contentPane.add(colorMenu, BorderLayout.NORTH);
contentPane.validate();
```

(The default layout manager for `WindowController` extensions is `BorderLayout`, so new items can be placed in the `NORTH`, `EAST`, `SOUTH`, or `WEST` part of the window. Note that the `CENTER` section is filled with the canvas.)

However, in Java 5 one can just add to the `WindowController` extension directly and the system will automatically add it to the content pane. E.g.,

```
this.add(figureMenu, BorderLayout.SOUTH);
this.add(colorMenu, BorderLayout.NORTH);
this.validate();
```

(You can leave out the occurrences of `this.`)

You will need to decide which way you teach the students. We are inclined for now to directly access the content pane as that full code is what is executed at run time anyway, and the code will work with Java 1.4 and earlier compilers. However, we can also understand wanting to simplify the code if your students are running Java 5.

Once students understand how to create and install a GUI component on the screen, they are ready to understand how to wire things up to handle events. For simplicity, all of our examples have the `WindowController` extension acting as the listener for all events. While this is not the best practice for more complex programs, it has the advantage of being relatively straightforward and matching what was done with the mouse events. In later courses we would teach students how to use inner classes to handle events – or to use more general external classes, but we prefer to avoid that complexity for novices.

Because the `WindowController` object will always be the listener, we can give a very straightforward recipe on how to wire up the class to be a listener:

1. Add `this` as the appropriate kind of listener to the GUI item. E.g.,

   ```
   menu.addActionListener(this);
   ```

2. Add a declaration that the class implements the appropriate listener interface. E.g.,

   ```
   public class C extends WindowController implements ActionListener { ... }
   ```

3. Add the method promised by the listener interface:

   ```
   public void actionPerformed(ActionEvent evt) { ... }
   ```

A sample program like `DoubleComboBox` illustrates to students how this works.

Java only allows the programmer to place one GUI component at a time in a slot in a layout manager, so we must go to more work to place several items in the same slot, say in the south region in `BorderLayout`. This provides the excuse for introducing `JPanel`s.

We have found that this progression of first teaching how to create, initialize and install components, then how to hook them up to respond to events, and finally how to create slightly more complex GUI layouts works well with students. It allows us to slowly increase the complexity without overloading the students. We provide a GUI cheat-sheet with this information on it, as well as the idiosyncrasies of each of the different kinds of GUI components. We try to convey the impression that they are all really pretty much the same, and there is no reason to worry about memorizing the details (we tend to look them up or consult our cheat sheet when we build graphic user interfaces ourselves).

You can spend as much or as little time as you like on the various components. We feel that the buttons, labels, and text fields are the most important, but usually at least cover lightly components such as sliders and text areas (which are handy for output).

The last section of the chapter is on mouse and key events. You can skip completely the info on mouse events, but we find it useful to remark to them that they have been using a slightly simplified version of mouse-event handling with objectdraw, but the ideas are essentially the same. We find the key event handling very useful in some of our later projects, but this material can either be covered at this time or postponed until it is needed.

In summary, one can spend a lot of time on this material or just a very small amount. It depends on what is important to you in your own course. Our students have found it a welcome relief from the introduction of new concepts of previous weeks, so we typically linger over it a bit while students are working on our “test programs”, programs that count for a higher percentage of their midterm grade. However, it is difficult to make the case that covering this material in great detail is essential to an introductory course. In particular, while it is worthwhile to show students something about different layout managers, it is generally not worth the time to teach students how to create elaborate layouts. For their later programming projects, we often provide students with assistance (i.e., provide code) for doing the layout so that they can focus on the parts of the programs that are more important.

# Chapter 12

# Recursion

Most introductory text provide only a cursory presentation of recursion and put it off among the optional chapters at the end of the book. We have chosen instead to place recursion in a central location in our text. While it is possible to skip this chapter entirely without causing problems with the flow of the rest of the book, we'd like to try to convince you that it can be good to cover recursion early.

The first times that we taught with Java, we followed the usual practice of putting the material on recursion toward the end of the term. However, we discovered that once we started covering arrays, multiple dimensional arrays, strings, exceptions, and files, our students tended to forget the object-oriented principles we had been teaching them, instead developing a more procedural style. In particular, we found it difficult to convince students to encapsulate arrays inside of classes, even when the class could be designed to present a much nicer interface to the application program than a bare array.

Partially as a result of this, we decided to try moving the recursion section earlier and shifting from an emphasis on procedural recursion to structural recursion. We found that this early introduction had several advantages:

1. Structural recursion reinforces object-oriented concepts like class and interface, as well as dynamic method dispatch.

2. Students are more motivated to learn recursion as it is the only way they know at this point to hold collections.

We discuss this in more detail in the paper "Why Structural Recursion Should Be Taught Before Arrays in CS 1", which is available on the instructor's web site for the text.

With this extra reinforcement of object-oriented concepts using structural recursion we discovered that students were more likely to use classes to encapsulate arrays during those later sections, so we now always introduce recursion before arrays.

There are many ways of presenting structural recursion. In Section 1 of this chapter we first present a class that uses a while loop for drawing nested rectangles and then point out that we can't move it because we can't keep track of all of the pieces. We then present a recursive solution in which the base case is represented by the `rest` instance variable holding the value `null`. However, we favor the solution in which there is an interface representing nested rectangles, a base class representing an empty collection of rectangles, and a recursive class representing one or more nested rectangles.

An example we often use in class is a target. We typically give an illustration of how one can draw this with a while loop, but don't bother to show the simple recursive class where the base case is represented by a null instance variable. Instead we launch immediately into the version with interface, base class, and recursive class. The interface is simple, containing only methods needed to drag around the target, such as `move` and `contains`. We usually also include constants representing the diameter of the center and the distance between consecutive rings.

The base class is pretty trivial, just involving a filled oval. The recursive class contains two instance variables, one representing the outermost ring, and the other representing the rest of the target (and having type `TargetInterface`). The interesting part of this solution (and the "better recursive solution" to nested rectangles) is that the only conditional statement in the two classes is in the constructor for the recursive class. Dynamic method invocation takes care of all other choices for methods. For example, the `move` method for the recursive case always sends `move` messages to both the outer ring and the rest, while the `move` method for the base case simply moves the filled oval representing the bullseye.

To help students understand how these programs work, we assign students to play the roles of the various recursive objects. For example, if we start with a bullseye and two outer rings, one student plays the bullseye (base case), another plays the small target with outer ring and inner object corresponding to the bullseye, while a third plays a larger target with a larger outer ring and the smaller target. Now when a move message is sent to the larger target, it moves the outer ring and then tells the smaller target to move. Similarly the smaller target moves its outer ring and then tells the bullseye to move. The bullseye then just moves.

Acting out recursions like this makes recursion more concrete. They can see that each object gets its own copy of a script. The recursive cases all get copies of the scripts from the recursive class while the base object gets the simpler base script.

The examples in the text of nested rectangles and URL list are essentially just recursive linked lists. In class we do the target example described earlier and scribbles that can be dragged. After we show a couple of examples like this, we highlight the similarities to students.

Of course there is a big difference between seeing and being able to write recursive examples. We give a check list for how to design these structures in Section 12.1.3. We would love to be able to prove that this recipe always results in correct programs, but we cannot assume that all of our students know mathematical induction (though we hint at how this would work for our students who have seen induction). As a result, in Section 12.1.4 we provide some intuition as to why our recipe will give correct results.

Our strategy in covering this material first has been to reinforce the basic ideas of recursion by showing repeated examples that are essentially simple linked lists. However, we conclude with a sampling of other examples that expose students to other recursive programming patterns including structures with mulitple recursive sub-parts and examples of algorithmic recursion.

Thus we finish Section 12.1 with a subsection on drawing broccoli. Many objects in nature are recursive, and broccoli is one of them (parsley is another, see Exercise 12.5.2). We like to use this example as it is one that cannot easily be created with an array in place of recursion. Students are always amazed when the very complex broccoli appears on the screen, and even more amazed when you can drag it around.

Section 12.2 discusses recursive methods and provides fairly standard examples of fast exponentiation and the towers of hanoi puzzle. Recursive methods show up again in Chapter 20 on searching and sorting, where both iterative and recursive methods are provided for the algorithms.

Chapter 13

# General Loops in Java

This chapter pulls together lots of loose ends with loops. It covers `for` and `do ... while` loops, as well as talking about common problems with loops, such as off-by-one errors, infinite loops, and problems with termination conditions that involve comparisons of doubles. The purpose of this chapter is to help students solidify their understanding of loops of all forms. This chapter is placed at this point in the text in order to prepare students for the counting loops typically involved in working with arrays.

# Chapter 14

# Arrays

Arrays are a standard way of grouping together homogeneous collections of data. This chapter introduces one-dimensional arrays, illustrates ways in which they can be used, and explains common algorithms for manipulating arrays. Chapter 20 on searching and sorting also includes many examples using arrays.

Our stategy in this chapter is to introduce how to work with arrays starting with simple uses and building to more complex ones. For example, we first show examples of "full" arrays, emphasizing the use of indices and loops to iterate through the elements of the array. We then move on to variable sized arrays, and finally to adding and removing elements from arrays (as opposed to just updating elements). These are substantially more complex as they require sliding elements of the array to the right or left to make a space for a new element or to remove a gap left from an old one.

This is the longest chapter in the book. However some of this is due to the detailed examples covered there. Don't feel that you have to cover all of the material in the same depth presented here. For example, array initializers are not essential for the material in the rest of the book.

There are a number of common problems that students encounter in using arrays. Students will need to be reminded that declaring an array does not create it, and creating the array does not create the entries of the array. Students will also find it difficult at first to get used to indices of arrays starting at 0 rather than 1. Another common difficulty for students is learning the difference between the capacity of the array (the number of slots created when the array is created) and the number of elements currently in the array (which generally needs to be kept track of with an instance variable). Students must also learn to make sure there is room in the array before adding new elements. Examples will help for all of these.

A source of possible confusion for students is that one obtains the length of an array by accessing its (public) instance variable `length` rather than sending it a message. (This violates our style rules that say that instance variables should never be public.) This is especially annoying in that in Chapter 16 students learn that strings support a `length()` method, but do not have the publicly accessible instance variable, making it hard for students to remember where to use each.

A favorite example we use of an array is a drawing program where the user can create new geometric objects, which are kept in an array. This example can be used to show some important array algorithms. You can search for the top-most element that contains the point where the user clicks. The element clicked on can be moved to the top layer of the window and shifted to be the last element of the array. Similarly, a object clicked on can be removed from the screen and array.

We also show a reimplementation of a structural recursion problem using arrays. For example, we typically redo the draggable scribble example with an array to show an alternative way of looking at the problem. We emphasize that with problems like this it is important to hide the array implementation in a class that is provided with methods that make sense in the application (e.g., `contains` and `move` in the case of scribbles).

Java 5 supports two new features relevant to arrays. The first is the "`for-each`" loop (that unfortunately does not use the keyword `each`). This provides a short way of writing a `for` loop that iterates through all the elements of an array. However, it is not useful if the loop only goes through part of the array.

There is another useful feature in Java 5 that you might want to mention, but is not covered in the text. It is the collection class, `ArrayList<E>`, which represents a flexible array holding elements of type `E`. The main advantage of the `ArrayList` over a regular array is that if the data structure runs out of space, it will double the amount of space available to hold elements. It is implemented with an array as an instance variable, and every time it runs out of space a new array that is twice as big is created, and all of the old elements are copied into the new array. While this means there are occasional long delays, on average adding elements to an `ArrayList` takes only twice as much time as for an array, a reasonable trade-off given that one will never get stuck as a result of running out of space. If your students are mathematically inclined, this can be a useful computation to show in that the results are not intuitively obvious.

While an `ArrayList` is more object-oriented than an array, we currently are inclined to feel that it is important for students to encounter arrays (though some of us are less certain). Students don't seem to find the use of a type variable to be instantiated as particularly troublesome. The big advantage of this class in Java 5 over earlier versions of the language is that it restricts the type of the values in the data structure to be the type replacing `E` in the declaration. Similarly, one need not cast elements removed from the data structure.

One disadvantage of bringing up `ArrayList` is that one may be forced to talk about the fact that the type variable may be instantiated with `Integer`, but not `int`, as primitive types cannot be substituted for type variables. Because Java 5 automatically converts back and forth from `int`s to `Integer`s, you need not worry about converting elements, but it is unfortunate to have to talk about that difference at all.

While this chapter doesn't do much with graphics or active objects, our in-class examples and lab exercises do use them. For example, we often assign the "Simon" problem, which simulates a memory game where the user must repeat a collection of tones that correspond to flashing lights. The programmer must insert pauses between playing the notes or the sounds all run together, so an active object must be used to play the sequence of tones.

# Chapter 15

# Multidimensional Arrays

The purpose of this chapter is to introduce more complex data structures for holding homogenous collections of objects. While we only discuss two-dimensional arrays here, the principles introduced generalize to higher-dimensional arrays.

Two dimensional arrays are interpreted in Java as arrays of arrays. Thus we open this chapter with a calendar application where there is a row for each month, and where the month corresponds to an array of days of the month, ranging in length from 28 to 31.

However, most of the time we are interested in rectangular matrices, so sections 15.2 and later focus on that topic.[1] Students should see examples of traversing a matrix in row-major order (with column entries changing the most rapidly) as well as column-major order, and know that the difference between these is accomplished by reversing the outer and inner for loops for the traversal.

Don't forget to mention to students that many real examples involves arrays of objects, each of which may have an array as an instance variable. For example, a student object may contain an array of classes. Each of those classes likely includes a field with the instructor, class hours (perhaps an array), meeting place, and the collection of students taking the class (probably represented as an array. That fact that the student references a class, which itself references the student makes it a bit more complicated to update, but causes no difficulty with the representation.

The text includes several example classes using two dimensional arrays. A very simple example involves checking for a winner in tic-tac-toe. Many students will find the example of smoothing images in the text interesting. There are similar scientific applications using heat transfer on a plane that can also be covered. Of course, there is also the old chestnut of the game of life.

Our favorite assignment for this material is the "nibbles" game. It involves a snake being guided across a field by the user looking for food. If the snake runs over food then it "eats" and grows. If it runs into itself or the wall, then it dies. Students have some difficulty in keeping the matrix and the graphics in sync. As a result we usually have students start out using a playing field of our design that encapsulates the two-dimensional array, and only after they get their snake class working properly, go back and write either all or part of the field class. You might try to entice your top-notch students to write a game with two players.

[1]Omitting the non-rectangular arrays altogether causes no problems with later parts of the text.

# Chapter 16

# Strings

Strings are very important in Java. In particular, text input and input from text fields are interpreted as strings. One can even go out over the web and grab an entire web page as a string. As a result, it is important to be able to operate on these strings in order to find whatever information is needed by the program.

Cover as much or as little of this chapter as you like. We find it useful to cover after arrays as `String`s are essentially represented as an encapsulated array of characters. Students have already been using strings for quite a while, so some of this will be review, though the methods will be new to them.

Strings are the most unusual Java data type. They are objects, yet can be written as literals with no use of "`new`". Moreover, they can be operated on with an infix operator, "`+`". This is all very convenient for us, but somewhat surprising.

Here are some key ideas to emphasize to your students:

1. The empty string, written `""`, and the string with value `null` are not the same. The second is uninitialized and cannot be used in any string operations, while the first is quite fine.

2. Strings are immutable. The string operations do not change the string receiving the message, they create a new string. Misunderstanding this is the most common error we see involving strings.

   There is a class `StringBuffer` for mutable strings. We have chosen not to cover the `StringBuffer` class in depth in this text. Instead there is a box on page 450 that talks about how `StringBuffer` differs from `String`. We don't expect students to use `StringBuffer` in our classes.

3. Only use "`.equals`" to compare strings. Java implementations tend to optimize strings so that all occurrences of the same string that are determined at compile time refer to the same object. However, when strings are constructed on the fly or obtained via input they will generally not be "`==`" to each other, even when they are "`.equal`". Keep reminding them not to fall into the trap.

4. Java does not emphasize the use of characters, but there are some occasions (e.g., in the results of the `charAt` method) where they do arise. Understanding them can also be helpful in checking input for consistency (e.g., that all characters in a text field are really digits).

You can get away with covering less than we include here if you wish. For example, we show how to check if a string represents an integer in the chapter on exceptions.

5. Escape characters, especially '`\n`' for new line, are handy for students to know when they are composing output for the console or a `TextArea`.

6. The method `parseInt` of `Integer` should be covered somewhere in your course. You may have already covered it when talking about using the `TextField` class. If not, be sure to cover it here.

7. Students should learn how to build up more complex strings in loops of the form:

```
while (...) {
   ...
   answerString = answerString + newVal;
}
```

While we want students to be able to use the various string methods, we haven't asked our students to memorize them. If they know how to look things up with Javadoc then they should be fine.

There are some interesting projects that become possible with strings. They can involve tasks such as taking information from web pages (which can be handled simply as a long string) and building a nice GUI interface from the information, or writing a very simple spam filter. While the students don't yet know how to get data off the network, the instructor can provide a class to take care of that for students, promising to reveal how it works later.

# Chapter 17

# Inheritance

This inheritance chapter can be covered comfortably any time after Chapter 11 on GUI components, and with a bit more work can be covered even earlier. Why did we include it more than three-quarters of the way through a book that emphasizes an objects-first approach?

There are several reasons we chose to delay the introduction of inheritance:

1. We feel that inheritance is a more advanced (and complicated) topic that can be better understood when students have more programming experience.

2. Important concepts of object-oriented programming like encapsulation, subtyping (using an object of one type in a context that expects another), and dynamic method invocation can best be explained separately from inheritance. Explaining these concepts at the same time as method override in inheritance can be too complicated for novices.

3. Inheritance can be abused by inexperienced users. We feel that inheritance should be an important part of the general notion of object-oriented design, a topic that fits better into a second course or later in computer science. In particular, it is our experience that inheritance often does not work well unless the classes were designed for inheritance from the beginning. Put a slightly different way, using inheritance often requires refactoring the original classes to find the best class hierarchy. (See items 14 and 15 in Block's excellent "Effective Java Programming Language Guide", Addison Wesley, 2001, for more details.)

Of course if you disagree with this reasoning, then you are welcome to cover this material earlier in the course. On occasion we have covered it right after GUI components, for example.

There is a lot of material in this chapter. You need not cover all of it, but you should be sure to explain enough about inheritance so that students understand the use of the word `extends` in classes extending `WindowController` and `ActiveObject`. Students should also understand how `equals` and `toString` are inherited from `Object`, and can be (and often should be) overridden in classes written by students.[1]

It is important to discuss the difference between `public`, `protected`, and `private` here. Unfortunately, an accurate description of `protected` would require a good understanding of packages. Because we have limited ourselves to being only consumers of packages here, students don't know

[1]More advanced programmers should also redefine the `hashCode` method when `equals` is redefined, but that won't mean anything to students at this level.

enough for the accurate explanation to make sense. We urge you to explain that `protected` instance variables and methods are available to extending classes and then just mention that there are other rules for visibility that have to do with packages that you won't be discussing in this class. We tell the ugly truth in the footnote at the bottom of pages 471 and 472 of the text.

It is worth noting that some people feel that protected instance variables are the spawn of the devil, and that instance variables should only be private. We do not feel that strongly ourselves, so we do use a few protected instance variables in the text. Feel free to accuse us of heresy for that to your students if you are so inclined.

Other mysteries that can be explained with an understanding of inheritance include the accessibility of the instance variable `canvas` in extensions of `WindowController` (`canvas` is a `protected` instance variable that is inherited). Restrictions on where `getImage` can be invoked are similarly due to it being inherited from `Controller`, which is itself a superclass of `WindowController`.

We cover inheritance in more detail in our own course, including a fairly extensive discussion of overriding methods. We explain overriding with a series of examples of falling objects. First we start with a generic `FallingObject` class that includes an instance variable of type `Drawable2DInterface`. The constructor initializes the speed and stopping height of the falling object, but does not create the falling object itself. The class also includes a `run` method that slowly moves the object down the screen until it gets to the stopping height. We then define subclasses that simply add a new constructor that calls the superclass constructor (required by Java!), creates the falling object and then invokes the `start` method. We do this to create falling leaves and sleet.

This example provides a very simple introduction to inheritance where only the constructor varies between classes. A more interesting variant occurs when we also override the `run` method. We define a `Tomato` class that extends `FallingObject`. The redefined `run` method calls the old, but then draws a picture of a "splatted" tomato at the bottom of the screen.

Finally, we decide it would be more interesting if we allowed more exotic behavior after an object falls all way to the ground. To support this, we modify the original `FallingObject` class so that the method `hitBottom` is called at the end of the `run` method.[2] This allows us to provide more interesting behavior without worrying about the details of what originally happened at the end of the `run` method. For example, in `hitBottom` in `FallingObject`, the method `removeFromCanvas` is sent to the falling object. However in the `Tomato` class we can instead simply distort the shape of the object. For fun, we also have a falling cow object that moos and runs off the screen when it hits the ground. Feel free to add your own weird falling objects.

This last design is interesting because in each case the object executes the original `run` method inherited from `FallingObject`, but the `hitBottom` method called from `run` represents different code for each of the falling objects.

There are lots of other examples we can go back and redo at this point. For example, we can redo the earlier laundry examples using pants and shirts. We can add a `LaundryImpl` class that implements the `Laundry` interface and is ready to be extended for particular kinds of laundry items. It has an instance variable, `parts`, that is an array of `DrawableInterface`. That array is created in the constructor, but no objects are added to it. However, the code for all of the other methods is included, as well as an `addComponent` method. Now we can define a subclass for `Shirt`, for example, by having the constructor call the super constructor and then call `addComponent` to add all of the pieces for the shirt. We can do something similar for pants and any other laundry items

[2]This notion of refactoring a class to better adapt it for inheritance is quite common in practice.

we wish to include.

We also tie together the notion of `implements` for classes and interfaces with `extends` for both interfaces and classes. We note that each of these gives rise to a subtype relationship where an object from a class can be used in a context expecting either a superclass or an interface that it implements, or anything obtainable via a chain of `extends` and `implements`.

We tend not to do much with abstract classes and methods or with `instanceof`, but we have included coverage of these concepts if you are interested in teaching them.

We'd like to leave you with a final warning on the proper use of inheritance. Sometimes programmers will use inheritance where they should use composition. Classes in the subclass relation should be in the "is-a" relation to each other. E.g., a `Leaf` as we defined it is a `FallingObject`. However, to pick on a bad example from the Sun Java libraries, the class `Stack` there extends `Vector`, but a `Stack` is not a `Vector` because vectors support behavior that stacks don't have (e.g., adding elements anywhere in the collection). In that case, the `Stack` class should have had an instance variable of type `Vector` to hold the data rather than extending `Vector`. This would have been a good example of using composition rather than inheritance to build more complex objects.

# Chapter 18

# Exceptions

The immediate reason for talking about exceptions here is that the following chapter on streams requires exceptions. Of course students have been seeing error messages involving exceptions, e.g., `NullPointerException`, `ArrayIndexOutOfBoundsException`, and `NumberFormatException`, for quite a while, so they will likely appreciate learning at least a bit about exceptions. However, if you have no interest in covering streams aside from the new `Scanner` class in Java 5, you could skip this chapter as a whole. Nevertheless we would urge you to consider at least talking about how to catch exceptions if there is sufficient time in your term.

We would recommend at least covering Section 18.1, which discusses the `try-catch` block and how that can be used to handle errors in handling input (e.g., from text fields that are to be treated as `int`s. Section 18.2 explains how to extract information from the exception object on what went wrong.

We start by only discussing `RuntimeException`s, which need not be caught inside the methods that throw them. Thus it is easy to talk about exceptions propagating back to calling methods without requiring the addition of a `throws` clause to the method header. Checked exceptions don't arise naturally until streams, so you may wish to postpone discussing them until you cover that chapter.

It is also useful to talk about catching different exceptions with different `catch` clauses. To explain this you will need to give some explanation of the inheritance hierarchy of exceptions, at least to mention that `Exception` is a superclass of all of the other exceptions.

Students will be interested to see how exceptions can be thrown by the programmer, but this material may certainly be skipped if time is an issue.

# Chapter 19

# Streams

Streams are the mechanisms used by Java to import information from outside of the computer memory. Data is read from or written to a hard drive or remote file servers using streams. Streams can also be used to access and create web pages or communicate with other services available over the internet. As a result, it is important for students to know how to use streams.

There are a number of details that must be taken care of in order to use streams in Java. Rather than asking students to memorize them, we simply encourage them to understand our code and to copy the code to their own programs. Another option would be to create a streams "cheat-sheet" summarizing the most relevant information about files. No matter which approach you take, you will need to emphasize to students that the system can throw exceptions when attempting to open or close a file, as well as every time a read is attempted. They will need to surround such statements with `try-catch` clauses and provide appropriate error handling.

One complication to using streams is that applets – at least when embedded in web pages – cannot access files. (You will notice that our on-line stream demos that are embedded in web pages will not run as applets.) As a result, we must now teach students how to write applications rather than applets. If all we want is to simply pop up a window in which we can embed GUI components, this is not hard, and it is explained in section 19.3. Section D.6 in Appendix D talks about the slightly more ambitious task of embedding an applet in an application, but the simple approach explained in section 19.3 will be sufficient as long as you are not also using a canvas to draw on.

Beyond simple reading from and writing to files, there are a few other items that we like to introduce to students. Rather than expecting users to type in file names for I/O, it is much nicer to use a dialog box. We introduce `JFileChooser` in Section 19.4 to allow users to use the regular system dialog boxes for either opening or saving files.

We also like to show students that now that they can read and write files, they can also read web pages and open sockets. This opens up a number of possible sample programs or lab exercises. Students can read in a web page and construct a GUI interface that provides nice access to the information. They can also open a mail connection, allowing applications like writing a spam filter to sort out the junk mail. Another application is a version of the Pictionary game where one student draws a picture that is displayed on another student's screen, and then that student tries to guess what the picture is of.

The advantage of Java streams is that it is as easy to open a socket across the network as it is to read a file. The disadvantage of Java streams is that it is as hard to read a file as to open a socket across the network. Java 5 attempts to deal with the problem by introducing the `Scanner` class for simple I/O. If you don't care about more sophisticated I/O of the sort covered in this chapter, you might want to skip this chapter (and perhaps the preceding chapter on exceptions) and just talk about the `Scanner` class. The following two URL's point to articles about the `Scanner` class. You will no doubt be able to find more on line with a search.

`http://java.sun.com/developer/JDCTechTips/2004/tt1201.html#1`

`http://www.bluej.org/objects-first/supplement/java5-supplement.html#scanner`

# Chapter 20

# Searching and Sorting

While computers still spend a fair amount of their time searching and sorting, and of course a company like Google depends for its survival on how good a job they do at this, we believe the real reason to cover searching and sorting in an introductory class is to show students how to analyze the complexity of algorithms, and to show them that clever algorithm design can have a large impact on the efficiency of programs.

We tend to prefer to provide recursive algorithms and programs for searching and sorting, because we believe the algorithms are more easily understood that way.

For example, a binary search operates on a sorted array by comparing the key of the middle element of the array with the key being looked for. If the middle element is too small then discard the middle element and all of those smaller, and perform a binary search on the remaining elements of the array. If it is too big, discard the middle element and all those larger, and perform a binary search on the remaining elements. If you find the element with the appropriate key, return its index. If there are no elements to be searched left in the array, return -1.

An insertion sort on an array of n elements is accomplished by first performing an insertion sort on the first n-1 elements of the array and then inserting the last element where it goes amongst those already sorted.

Of course, you can explain the algorithms iteratively, instead, or discuss both ways of thinking about them. We have only provided the recursive version of the merge sort, though the iterative version is suggested in Exercise 20.3.14.

When we present this material in class, we like to use sticky notes on the board to represent the elements in the array. That way we can move them around to illustrate how the algorithm works. We also have a demo program on-line that animates each of the algorithms using a list of bars to represent the contents of the array.

There are two other new things introduced in this chapter. We introduce pre- and post-conditions to document methods. We believe that these make it much easier to understand why the algorithms work. We also give relatively informal arguments for the complexity of each of these algorithms. You can do better if your students understand mathematical induction. We do not introduce the "big-O" notation here, but instead provide tables to illustrate how rapidly the differences grow when the complexity classes differ. These should be sufficient to make the point that real differences are possible when you are more clever with algorithm design.

# Chapter 21

# Introduction to Object-Oriented Design

The chapter on design was placed last in the text not because we thought it should be the last chapter covered, but instead because there were too many places it could logically be placed. We typically cover this material sometime after Chapter 9 and before chapter 16. Section 21.7 refers to the implementations of nested rectangles using an array and with structural recursion, but it would be easy to replace this with another example or to postpone this section on encapsulation and information hiding until a later lecture than the rest of the chapter.

We believe that good object-oriented design is hard for novices. Therefore the focus when we teach this course is on showing students examples of good object-oriented design, rather than having them become experts in object-oriented design.

We occasionally cheat a bit on good design in the interests of keeping things simple. For example we would normally use separate inner classes as listeners to different GUI components in our own programs, but we put off introducing inner classes (aside from perhaps sneaking in a quick example in a later lecture of this course) until a follow-up course. Nevertheless we do our best to present good designs, and compare alternative designs where possible – e.g., when we introduce classes at the beginning of Chapter 6 and at the beginning of Chapter 12 when we introduce recursive structures. We generally provide students with at least the names of classes and their responsibilities when we provide laboratory assignments. Just as students cannot learn to write until they understand how they read, we feel that students need to see lots of good designs before they can do good object-oriented design on their own.

The purpose of this chapter is to explain the basics of object-oriented design; for example that properties of objects generally correspond to instance variables, and behavior is reflected in methods. We push hard on the notion of abstraction. When we model something in the computer we do not represent all of its properties or behavior – only those aspects which are relevant for the current problem. That is, we abstract away those aspects that are not important for our application.

We also emphasize the other side of abstraction – information hiding. Instance variable should always be private (or protected), while only those methods that are designed to be used by external objects should be public. In particular, methods that are designed to simplify methods or access particulars of the implementation, and make little or no sense outside of the class, should not be public. This makes it easier later to make changes to the implementation. As we tell students, the only programs that don't get changed are those that are designed for one-time use or are so bad that no one wants to use them. Successful programs always undergo change (ask them how many programs they use are at version 1.0).

While the text introduces this material with a new application, a shell game, we find it is most useful to students to present this in terms of large programs that they have worked on (or are currently working on). For example, our students generally write the `Frogger` game after covering Chapter 9 on active objects. That game has classes `Frogger` (to control the game), `Frog`, `Lane`, and `Vehicle`, the last two of which are extensions of `ActiveObject`. We provide the students with the design for this program and explain what many of the methods need to do. While students are working on this or later, we find it useful to help them understand why we created the design in this way.

If your class is small enough (or you can break off a group that is small enough), it is useful to have the class help you do the design by talking about what you need each of the pieces to do. Identifying the objects and classes is not particularly difficult, but it takes some work to figure out what methods each of them should have. In particular there are a couple of ways in which the responsibility for determining whether the frog has been run over by a vehicle.

We put headings on the board for each class and then ask the students to suggest what methods are needed, and what they should do. This results in writing the specification of a method (which turns into its header comment). As we design, we discover that we need to save information in instance variables for later use. As in the example in the text, we often need to backtrack, changing our mind about the best ways of doing things.

If the class participates, you may end up with a design slightly different from the one given with the assignment or actual program, but it will be similar enough that students can see where the instructor's design came from. It is also useful for students to see that there are several alternative designs, and to understand that one can analyze the pros and cons of each.

One warning: try to resist creating too many "setters" and "getters" for instance variables. Having too many of these methods usually is a sign that your class design is too low level. The methods should reflect the intended behavior of the object, not its implementation.

One suggestion is first to talk about the design of a program that students have already turned in so that they can see how the design resulted in the program and how successive refinement contributed to the details of the actual code. You can then turn to the design of a program that students are currently working on and help them see that the first few steps of the design follow the same pattern, but then leave them to finish the design on their own.