

Data-Driven Programs

The primary focus of today's class is the idea of **data-driven programs**, which are programs in which the flow of execution is controlled by the data structures. Data-driven programs are usually shorter, more flexible, and easier to maintain than programs that incorporate the same information directly into the program design.

Programmed instruction courses

Twenty-five years or so ago, when computers were just starting to be used as general-purpose tools, there was a movement within the educational community to use computers as part of the teaching process. One of the proposed techniques for doing so is called **programmed instruction**, a process in which a computerized teaching tool asks a series of questions so that previous answers determine the order of subsequent questions. If a student is getting all the right answers, the programmed instruction process skips most of the easy questions and moves quickly on to more challenging topics. For the student who is having trouble, the process moves more slowly, leaving time for repetition and review.

Let's suppose you have been assigned the problem of writing an application that makes it possible to present material in a programmed instruction style. In a nutshell, your program must be able to

- Ask the student a question
- Get an answer from the student
- Move on to the next question, the choice of which depends on the student's response

Such a program will likely be much simpler than a commercial application, but you can easily design a program that illustrates the general principles involved.

The importance of using a flexible data structure

It is possible to design a programmed instruction application as a set of methods. Each method asks a question, reads in an answer, and then calls another method appropriate to the answer the student supplies. Such a program, however, would be difficult to change. Someone who wanted to add questions or design an entirely new course would need to write new methods. Writing methods is simple enough for someone who understands programming, but not everyone does. Most programmed instruction courses are designed by teachers in a specific discipline; those teachers are usually not programmers. Forcing them to work in the programming domain limits their ability to use the application.

As the programmer on the project, your goal is to develop an application that presents a programmed instruction course to the student but allows teachers without programming skills to supply the questions, expected answers, and cross-reference information so that your application can present the questions in the appropriate order. To do so, the best approach is to design your application as a general tool that takes all data pertaining to the programmed instruction course from a file. If you adopt this approach, the same program can present many different courses by using different data files.

In designing the application, you need to begin by considering such broad questions as:

- What are the overall requirements of the general problem? In particular, you need to understand the set of operations your program must support, apart from any specific domain of instruction.

- How can you represent the data for the programmed instruction course in the context of your program? As part of the design phase, you need to develop an appropriate data structure consisting of some combination of records and arrays.
- What should a course data file look like? As you make this decision, you need to keep in mind that the data file is being edited by nonprogrammers whose expertise is in the specific domain under consideration.
- How do you convert the external representation used in the data file into the internal one?
- How do you write the program that manipulates the internal data structures?

The rest of this handout considers each of these questions in turn.

Framing the problem

At one level, it is easy to outline the operation of the program. When your program runs, its basic operation is to execute the following steps repeatedly:

1. Ask the student the current question. A question consists of one or more lines of text, which you can represent as strings.
2. Request an answer from the student, which can also be represented as a string.
3. Look up the answer in a list of possibilities provided for that question. If the answer appears in the list, consult the data structure to choose what question should become the new current question. If the student's answer does not match any of the possibilities provided by the course data file, the student should be informed of that fact and given another chance at the same question.

Many details are missing from this outline, but it is a start. Even at this level, the outline provides some insight into the eventual implementation. For example, you know that you need to keep track of what the “current question” is. To do so, it makes sense to number the questions and then store the current question number in a variable.

Writing the program itself turns out to be one of the easier pieces of the task; the harder problems arise in representing the data structure. For the program to be general and flexible, all the information that pertains to an actual course must be stored in a data file, not built directly into the program. The program's job is to read that data file, store the information in an internal data structure, and then process that structure as outlined earlier in this section. Thus, your next major task is to design the data structures required for the problem so that you have a context for building the program as a whole.

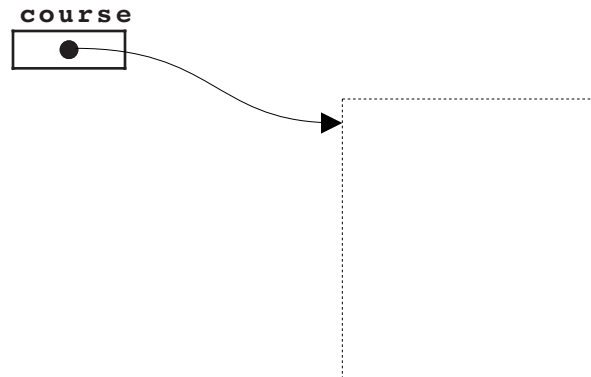
The process of designing the data structures has two distinct components. First, you have to design an *internal* data structure for use by the program. The internal data structure consists of type definitions that combine arrays and records so that the resulting types mirror the organization of the real-world information you seek to represent. Second, you must design an *external* data structure that indicates how the information is stored in the data file. These two processes are closely related, mostly because they each represent the same information. Even so, the two structures are tailored to meet different purposes. The internal structure must be easy for the programmer to use. The external structure must make it easy for someone to write a course, without making it too difficult for the program to manipulate.

Designing the internal representation

The first step in the process is to design a data structure that incorporates the necessary information. In most cases, it is easiest to design the data structure from the top down,

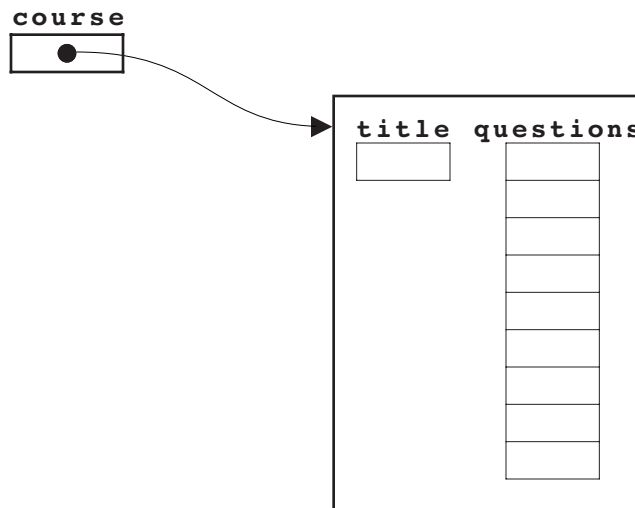
starting with the highest-level structure and then refining it by specifying more and more of the details. Here, the highest level is the course as a whole. That course contains a set of questions, and each question contains a set of answers and their associated transitions.

In designing any data structure, one of the most important concepts is that of **encapsulation**, which is the process of combining related pieces of data into structures that can be treated as complete units. For most applications, the encapsulation process is hierarchical and must be considered at varying levels of detail. At the highest level, you think of the entire structure as a single object, which contains all the information you will need for the course. That object is a reference to some data collection, the details of which are not important at the highest level of detail:

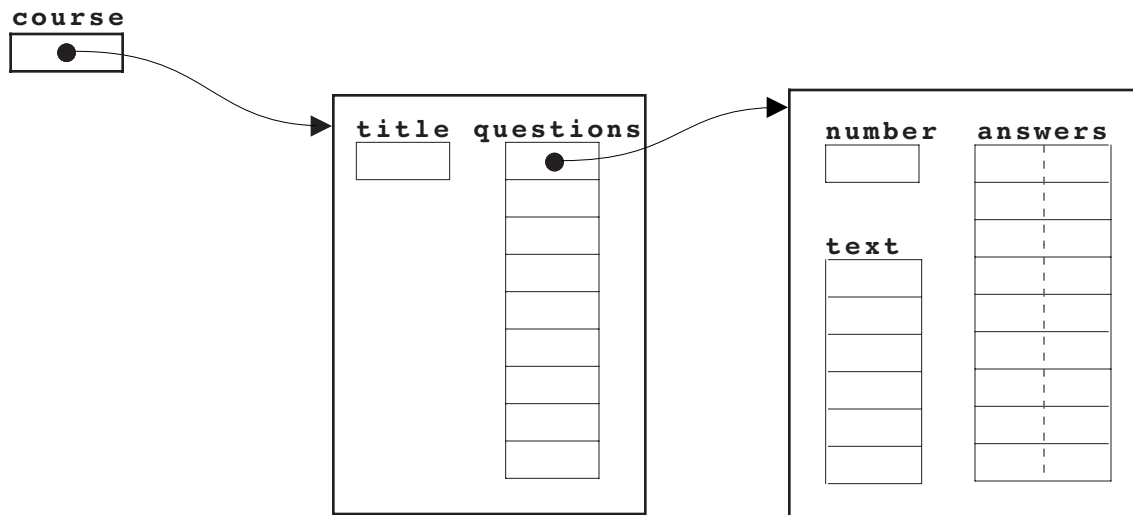


Whenever you need to pass the entire course to a method, all you have to do is pass the variable `course`, which is a small, easily manipulated pointer that gives you access to the other data. It is only when a method or method needs to manipulate the individual fields of the data structure that it has to look inside the structure to see its details.

Given that your current purpose is to design the internal structure, however, you do need to understand what is contained within the dotted box in the diagram above. Intuitively, you know that the data structure as a whole contains a list of questions, although it may make sense to include other information as well, such as the title of the course. At first glance (although we will reconsider this point later), it seems as if the questions should be an array of some as-yet-undefined structure representing a single question. Thus, the data structure at the current level of decomposition looks like this:



Each question, however, must store the number and text of the question, along with some structure that associates possible responses with next-question indicators. Thus, the entire structure will look something like this:



Designing the external structure

Before you turn to the details that will allow you to code the internal data structures, it often helps to think about how to represent the information within the data file. Files are simply text, and the organization provided by the data structure hierarchy in Java must be expressed in the design of the file format. The file structure design must also make it easy for someone to write and edit, even if that person is not a programmer. Thus, you should choose a representation that is as simple as possible. In this case, it seems easiest to write out each question, one after another, along with its likely answers. So that the computer can tell where the answers to one question stop and the next question begins, you must define some convention for separating the question-and-answer units. A blank line works well in this context, as it does in most file structures. Thus, in individual units separated by blank lines, you have the data for each question and its answers.

But what goes into each question-and-answer unit? First of all, you need the text of the question, which consists of individual lines from the file. You also need some way to indicate the end of the question text, and the easiest way, both for you and for the course designer, is to define a sentinel. In this program, I have arbitrarily chosen a line of five dashes to indicate the end of the question text. Furthermore, you must allow the course designer to specify the answer/next-question pairs. Here, I have chosen to represent both of these values on a single data line consisting of the answer text, followed by a colon, followed by the index of the next question. I could have chosen other formats, but this design seems as if it would be easy for a novice to learn. Thus, the data for an individual question entry in the file looks like this:

```

True or false: The earth revolves around the sun.
-----
true: 3
false: 2
  
```

The question text consists of a single line, after which there are two acceptable answers. If the user types in **false**, the program should go to question 2; if the user types in **true**, it should move on to question 3.

This example brings up an interesting question. How do you assign question numbers to each of these entries? One approach would be to arrange the questions in the file and number them sequentially. This strategy makes life easy for you as the programmer. The problem is that it makes life difficult for the person writing a course.

To understand why this is so, suppose that the course designer wants to add a new question near the beginning of an existing course. All subsequent questions move down by one, all the question numbers change, and the course designer has to spend a considerable amount of time renumbering all the next-question indicators. A better approach is to let the person who writes the question give it a number. For example, if the sample question about the earth and sun were question #1 in the database, its entry in the file would begin with its question number, as follows:

```
1
True or false: The earth revolves around the sun.
-----
true: 2
false: 3
```

The advantage of allowing the course designer to supply question numbers is that it makes editing the course much easier. Someone who wants to add a new question can just give it a question number that hasn't been used before. None of the other question numbers need to change. The course designer can then insert the new question anywhere in the data file, because there is no longer any reason that the question numbers need to be consecutive.

One implication of allowing the course designer to choose question numbers is that those numbers need not be consecutive. In fact, it generally makes designing a course easier if the numbers are *not* consecutive, because then you can assign numbers so that they reflect the logical structure of the course. For example, you could decide that questions numbered between 1 and 100 were part of one topic, questions between 101 and 200 were part of another topic, and so on. In all likelihood, you wouldn't fill up these ranges, so there would be gaps in the sequence.

The idea that the question numbers are not necessarily consecutive has implications for the data structure as well. If the question numbers are completely arbitrary and don't indicate the sequence of the question, then the structure that contains them is not really an array in the traditional sense. What you need instead is a map between question numbers and the object that maintains the data structure for a particular question.

Coding the program

Once you have defined the internal data structure and the external file format, the process of writing the code for the teaching machine program is surprisingly straightforward, as long as you decompose the entire task into simpler methods using stepwise refinement. The complete program is shown in the slides for the lecture, which begin on page 8.

The value of a data-driven design

The **TeachingMachine** program takes all its data from the course data file. The questions it asks, the answers it accepts, and even the sequencing of the questions are supplied by the data file, not by the program. Programs that control their entire operation on the basis of information from a database are said to be **data-driven**. Data-driven programs are usually shorter, more flexible, and easier to maintain than programs that incorporate the same information directly into the program design.

To illustrate just how flexible a data-driven system like **TeachingMachine** can be, it is useful to show the program in operation. According to the initial goals of the project, the **TeachingMachine** program would be used for a traditional programmed instruction course, such as the file **JavaReview.txt** whose first few questions are:

```
Java programming review
1
Would you like help with int or boolean type?
-----
int:      2
bool:    10

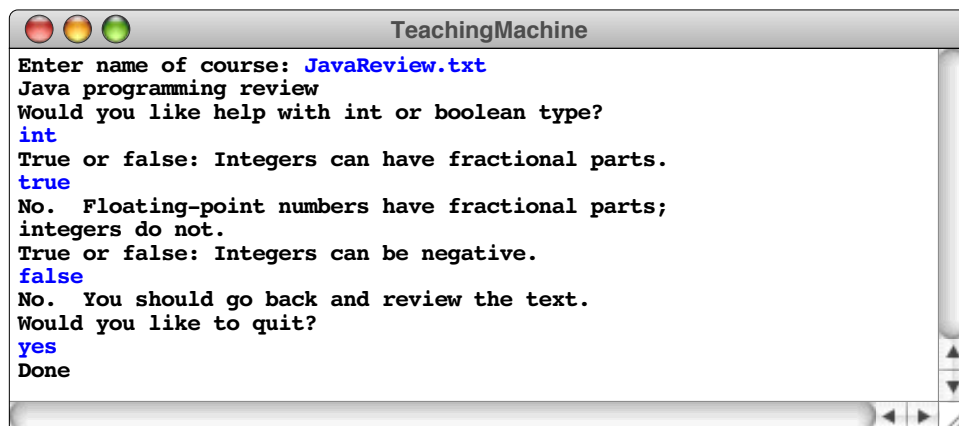
2
True or false: Integers can have fractional parts.
-----
true:     3
false:    5

3
No. Floating-point numbers have fractional parts;
integers do not.
True or false: Integers can be negative.
-----
true:     5
false:    4

4
No. You should go back and review the text.
Would you like to quit?
-----
yes:      0
y:        0
no:       1
n:        1

. . . file continues with additional questions . . .
```

When the **TeachingMachine** program is used in conjunction with this data file, a student might see the following sample run:



The screenshot shows a window titled "TeachingMachine". Inside, the text from the data file is displayed, with user input in blue. The input sequence is: "JavaReview.txt", "int", "true", "false", and "yes". The window has a standard Mac OS X title bar with red, yellow, and green buttons. A scrollbar is visible on the right side of the text area.

```
TeachingMachine
Enter name of course: JavaReview.txt
Java programming review
Would you like help with int or boolean type?
int
True or false: Integers can have fractional parts.
true
No. Floating-point numbers have fractional parts;
integers do not.
True or false: Integers can be negative.
false
No. You should go back and review the text.
Would you like to quit?
yes
Done
```

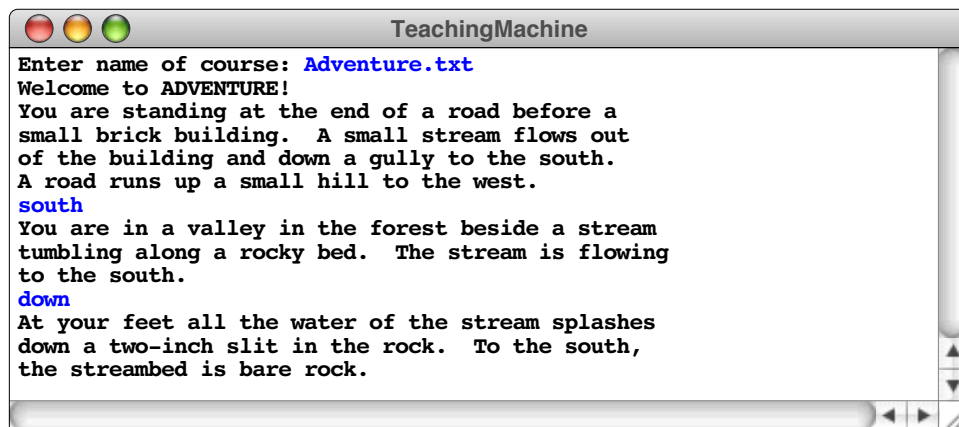
Because the program is data-driven, however, you could use it in entirely different contexts. For example, the following file contains the first few lines from the data file **Adventure.txt**, which is adapted from the original computer adventure game developed by Willie Crowther that serves as the foundation for Assignment #6:

```
Welcome to ADVENTURE!
1
You are standing at the end of a road before a
small brick building.  A small stream flows out
of the building and down a gully to the south.
A road runs up a small hill to the west.
-----
south: 2
north: 8
in:    8
west:  9

2
You are in a valley in the forest beside a stream
tumbling along a rocky bed.  The stream is flowing
to the south.
-----
south: 3
down:  3
north: 1

3
At your feet all the water of the stream splashes
down a two-inch slit in the rock.  To the south,
the streambed is bare rock.
-----
north: 2
south: 4
down:  4

. . . file continues with additional rooms . . .
```



As the sample run shows, someone who uses the **TeachingMachine** program in conjunction with the file **Adventure.txt** will perceive the program's purpose very differently than someone who runs it with the **JavaReview.txt** file. Even though the **TeachingMachine** program has not changed at all, the programmed instruction course has become an adventure game. The only difference is the data file.