Eric Roberts
CS 106A

Handout #55
May 23, 2012

Assignment #7—Adventure

The vitality of thought is in adventure.
— Alfred North Whitehead, *Dialogues,* 1953

Due: Monday, June 4, 5:00 P.M.
Last possible submission date: Thursday, June 7, 5:00 P.M.

Welcome to the final assignment in CS 106A! Your mission in this assignment is to write a simple text-based adventure game in the tradition of Will Crowther's pioneering "Adventure" program of the early 1970s. In games of this sort, the player wanders around from one location to another, picking up objects, and solving simple puzzles. The program you will create for this assignment is considerably less elaborate than Crowther's original game and it therefore limited in terms of the type of puzzles one can construct for it. Even so, you can still write a program that captures much of the spirit and flavor of the original game.

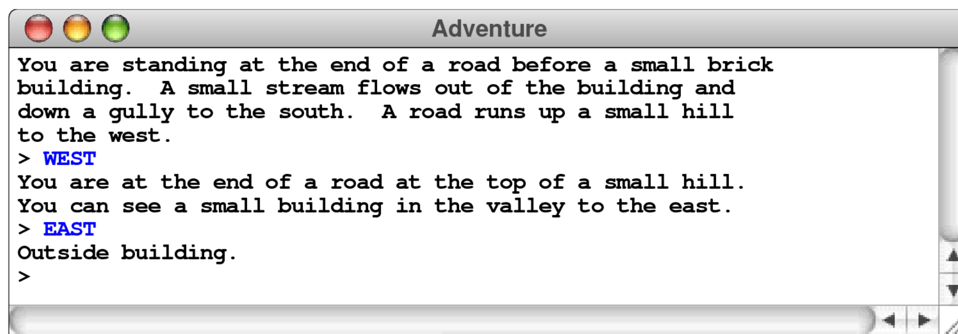
Because this assignment is large and detailed, it takes quite a bit of writing to describe it all. This handout contains everything you need to complete the assignment, along with a considerable number of hints and strategic suggestions. To make it easier to read, the document is divided into the following sections:

1. Overview of the adventure game 2
2. The **`Adventure`** class 5
3. The **`AdvRoom`** and **`AdvMotionTableEntry`** classes 6
4. The **`AdvObject`** class 11
5. Implementing the adventure game 12
6. Strategy and tactics 15
7. Administrative rules (partners, late days, and the like) 16

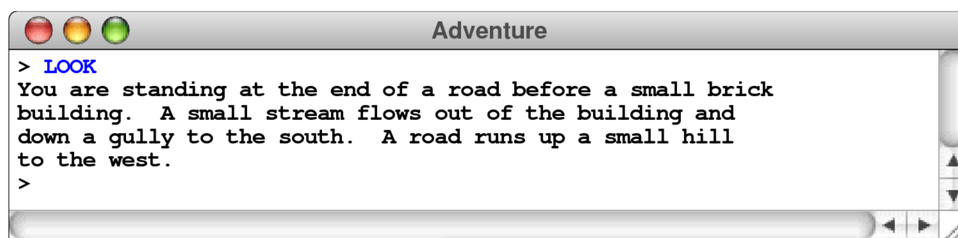
Try not to be daunted by the size of this handout. The code is not as big as you might think; in terms of the number of lines of code you have to write, it's only about half again as large as the Yahtzee program from Assignment #5. If you start early and follow the suggestions in the "Strategy and tactics" section, things should work out well.

Section 1
Overview of the Adventure Game

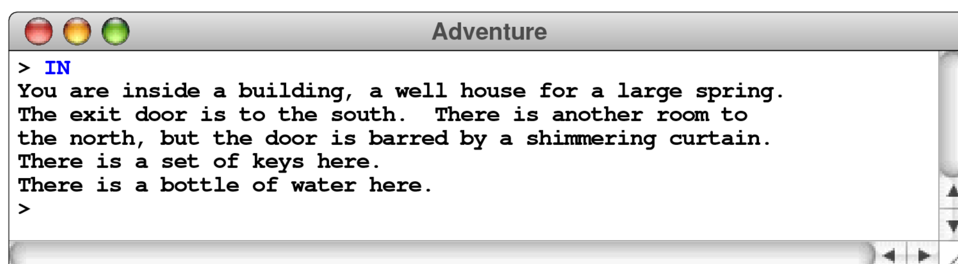
The adventure game you will implement for this assignment—like any of the text-based adventure games that were the dominant genre before the advent of more sophisticated graphical adventures like the Myst/Riven/Exile series—takes place in a virtual world in which you, as the player, move about from one location to another. The locations, which are traditionally called "rooms" even though they may be outside, are described to you through a written textual description that gives you a sense of the geography. You move about in the game by giving commands, most of which are simply an indication of the direction of motion. For example, in the classic adventure game developed by Willie Crowther, you might move about as follows:



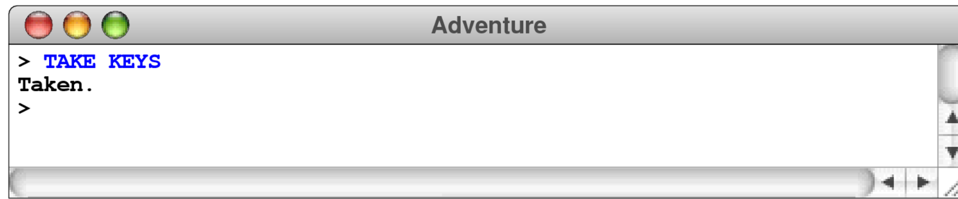
In this example, you started outside the building, followed the road up the hill by typing **WEST**, and arrived at a new room on the top of the hill. Having no obvious places to go once you got there, you went back toward the east and ended up outside the building again. As is typical in such games, the complete description of a location appears only the first time you enter it; the second time you come to the building, the program displays a much shorter identifying tag, although you can get the complete description by typing **LOOK**, as follows:



From here, you might choose to go inside the building by typing **IN**, which brings you to another room, as follows:



In addition to the new room description, the inside of the building reveals that the adventure game also contains objects: there is a set of keys here. You can pick up the keys by using the **TAKE** command, which requires that you specify what object you're taking, like this:



The keys will, as it turns out, enable you to get through a grating at the bottom of the streambed that opens the door to Colossal Cave and the magic it contains.

Overview of the data files

Like the teaching machine program in Handout #53, the adventure program you will create for this assignment is entirely *data driven.* The program itself doesn't know the details of the game geography, the objects that are distributed among the various rooms, or even the words used to move from place to place. All such information is supplied in the form of data files, which the program uses to control its own operation. If you run the program with different data files, the same program will guide its players through different adventure games.

To indicate which data files you would like to use, the adventure program begins by asking you for the name of an adventure. To get the adventure game illustrated above, you would begin by typing **Crowther**, which selects the collection of files associated with a relatively sizable subset of Will Crowther's original adventure game. For each adventure, there are three associated data files that contain the name of the adventure as a prefix. For the **Crowther** adventure, for example, these files are

- **CrowtherRooms.txt**, which defines the rooms and the connections between them. In these examples, you have visited three rooms: outside of the building, the top of the hill, and the inside of the well house.
- **CrowtherObjects.txt**, which specifies the descriptions and initial locations of the objects in the game, such as the set of keys.
- **CrowtherSynonyms.txt**, which defines several words as synonyms of other words so you can use the game more easily. For example, the compass points **N**, **E**, **S**, and **W** are defined to be equivalent to **NORTH**, **EAST**, **SOUTH**, and **WEST**. Similarly, if it makes sense to refer to an object by more than one word, this file can define the two as synonyms. As you explore the Crowther cave, for example, you will encounter a gold nugget, and it makes sense to allow players to refer to that object using either of the words **GOLD** or **NUGGET**.

These data files are not Java programs, but are instead text files that describe the structure of a particular adventure game in a form that is easy for game designers to write. The adventure program reads these files into an internal data structure, which it then uses to guide the player through the game.

Your program must be able to work with any set of data files that adhere to the rules outlined in this handout. In addition to the three files with the **`Crowther`** prefix, the starter folder also contains file named **`TinyRooms.txt`** that contains only three rooms with no objects and no synonyms and a set of three files with the prefix **`Small`** that define a much smaller part of the **`Crowther`** cave. Your program should work correctly with any of these files, as well as other adventure games that you design yourself.

The detailed structure of each data file is described later in this handout in conjunction with the description of the module that processes that particular type of data. For example, the rooms data file is described in conjunction with the **`AdvRoom`** class.

Overview of the class structure

The adventure game is divided into the following principal classes:

- **`Adventure`**—This is the main program class and is by far the largest module in the assignment. This class is yours to write, but—as will be true for all the classes you implement for this assignment—the public methods are specified in the starter files.
- **`AdvRoom`**—This class represents a single room in the game. This class is also yours to write. The private methods that decompose parts of the operation are yours to design, but the specification of the public methods used to communicate with other modules is specified. This class is closely linked with the **`AdvMotionTableEntry`** class, which is described in the same section. That class is provided as part of the starter project.
- **`AdvObject`**—This class represents one of the objects in the game. As with the **`AdvRoom`** class, you have to implement this class although the public methods are specified.
- **`AdvMotionTableEntry`**—This class defines a record type that combines a direction of travel, the room one reaches by moving in that direction, and an optional object that enables the motion. The definition of this class is extremely simple and is provided in the starter project.

The structure of each of these classes is described in detail in one of the sections that follow.

Even though the code for these components is substantial, your job is made considerably easier by the following properties of the assignment:

1. Most of the methods used to communicate among the classes have already been designed for you. The public methods in the **`AdvRoom`** and **`AdvObject`** classes are completely specified; all you need to do is implement them.
2. The project works from the very first moment that you get it. Each of the classes you need to design is supplied to you in the form of a **`.jar`** file containing several "magic classes" that implement all of the methods you need. The starter files are written as subclasses of those magic classes; your job is to remove the superclass designation and implement the necessary methods yourself. Providing these magic superclasses makes it much easier to test your code as you go because you can rely on our implementations.

Section 2
The **Adventure** Class

The main program class is called **Adventure** and will contain most of the code you have to write for this assignment. This is the class that opens the data files, assembles the data structures, maintains the list of words known by the game, interacts with the user, and implements the various commands. For the most part, you will have the opportunity to figure out how to decompose the entire operation into reasonably sized pieces and for choosing what data structures and methods to use in the underlying implementation.

The starter file for the **Adventure** class appears in Figure 2, which includes a definition for the **Adventure** class with an empty body. The implementation of this class is provided by **AdventureMagicSuperclass** in the **AdventureMagic.jar** library that comes with the starter folder. As the comments in Figure 2 make clear, your job is to take away the "magic" implementation that we've provided and replace it with your own code. When you finish, your version of **Adventure** class will extend **ConsoleProgram** instead of **AdventureMagicSuperclass** and contain complete implementations of a **run** method along with any private methods and instance variables you need.

The **Adventure** program has the following responsibilities:

1. Ask the user for the name of an adventure, which indicates what data files to use
2. Read in the data files for the game into an internal data structure
3. Play the game by reading and executing commands entered by the user

Understanding how to implement these aspects of the game, however, requires you to learn more about the **AdvRoom** and **AdvObject** classes, which are described in the next two sections. Section 5 then returns to the question of how to implement the **Adventure** class, which represents the lion's share of the assignment.

Figure 2. The Adventure.java starter file

```
/*
 * File: Adventure.java
 * --------------------
 * This program plays the CS106A Adventure game.
 */

/**
 * This class is the main program class for the Adventure game.
 */

public class Adventure extends AdventureMagicSuperclass {

   ////////////////////////////////////////////////////////////////////////
   // To complete the assignment, you need to change the superclass    //
   // to ConsoleProgram and write the methods for this class.          //
   ////////////////////////////////////////////////////////////////////////

}
```

Section 3
The AdvRoom and AdvMotionTableEntry Classes

The **AdvRoom** class represents an individual room in the game. Each room in the game is characterized by the following properties:

- A room number, which must be greater than zero
- Its name, which is a one-line string identifying the room
- Its description, which is a multiline array describing the room
- A list of objects contained in the room
- A flag indicating whether the room has been visited
- A motion table specifying the exits and where they lead

The **AdvRoom** stores this information in its private data structure and then makes that information available to clients through the public methods exported by the class. These methods are included in the starter file but are commented out, ensuring that the program uses the implementations from the superclass. When you are ready to write the code for the **AdvRoom** class, you need to remove the **extends** clause from the class definition, uncomment each of the methods, and then fill in the necessary code. As with the **Adventure** class itself, you will need to define instance variables and helper methods to complete the class implementation.

The methods in the **AdvRoom** class are described in detail in the comments in the starter file. For easier reference, a short description of each method appears in Figure 3.

Figure 3. Methods in the AdvRoom class

getRoomNumber()	Returns the room number.
getName()	Returns the room name, which is its one-line description.
getDescription()	Returns an array of strings that correspond to the long description of the room.
addObject(*obj***)**	Adds an object to the list of objects in the room.
removeObject(*obj***)**	Removes an object from the list of objects in the room.
containsObject(*obj***)**	Checks whether the specified object is in the room.
getObjects()	Returns an array of all the objects in the room.
setVisited(flag)	Sets the flag indicating that this room has been visited as specified by the value of the parameter **flag**. Calling **setVisited(true)** means that the room has been visited; calling **setVisited(false)** restores its initial state.
hasBeenVisited()	Returns **true** if the room has previously been visited.
getMotionTable()	Returns the motion table associated with this room. Each motion table is an array of **AdvMotionTableEntry** objects that specify the motion verb, the destination room, and an optional "key" required to traverse that passage.
AdvRoom.readRoom(*rd***)**	Creates a new room by reading its data from the specified reader. If no data is left in the reader, this method returns **null** instead of an **AdvRoom** value.

The rooms data file

As in the teaching machine example from Handout #53, the information for the individual rooms is not part of the program but is instead stored in a data file. One of your responsibilities in completing the implementation of the **`AdvRoom`** class is to write the static method **`readRoom(rd)`**, which creates a new **`AdvRoom`** object by reading that description from the rooms file for that adventure. At first glance, the data files for rooms look almost exactly like those for the teaching machine. For example, **`TinyRooms.txt`** looks like this:

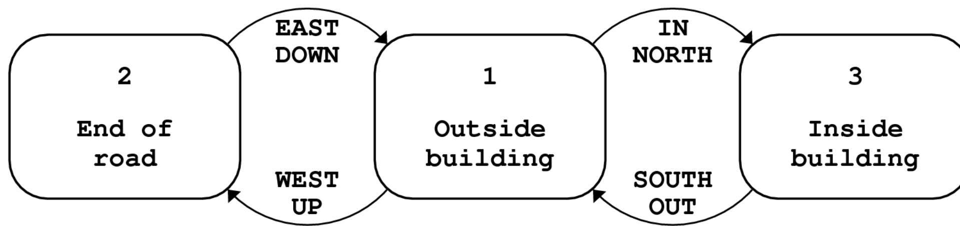
```
1
Outside building
You are standing at the end of a road before a small brick
building.  A small stream flows out of the building and
down a gully to the south.  A road runs up a small hill
to the west.
-----
WEST      2
UP        2
NORTH     3
IN        3

2
End of road
You are at the end of a road at the top of a small hill.
You can see a small building in the valley to the east.
-----
EAST      1
DOWN      1

3
Inside building
You are inside a building, a well house for a large spring.
-----
SOUTH      1
OUT        1
```

The only obvious differences between the external file format for the teaching machine and the adventure game are

1. The title line is missing (the **`TeachingMachine.java`** program requires a course title on the first line)
2. Each of the entries for an individual room includes a short description (such as **`Outside building`** or **`End of road`**) as well as the extended description.

These changes are in fact extremely minor.

In thinking about an adventure game—particularly as the player, but also as the implementer—it is important to recognize that the directions are not as well behaved as you might like. There is no guarantee that if you move from one room to another by moving north, you will be able to get back by going south. The best way to visualize the geographic structure of an adventure game is as a collection of rooms with labeled arrows that move from one room to another, as illustrated by the following diagram of the connections defined in **`TinyRooms.txt`**:



Extensions to the connection structure

If the adventure program allowed nothing more than rooms and descriptions, the games would be extremely boring because it would be impossible to specify any interesting puzzles. For this assignment, you are required to implement the following features that provide a basis for designing simple puzzles that add significant interest to the game:

- *Locked passages.* The connection data structure must allow the game designer to indicate that a particular connection is available only if the player is carrying a particular object. That object then becomes the key to an otherwise locked passage. In the file format, such locked passages are specified by adding a slash and the name of an object after a room number.
- *Forced motion.* If the player ever enters a room in which one of the connections is associated with the motion verb **`FORCED`**, the program should display the long description of that room and then immediately move the player to the specified destination without waiting for the user to enter a command. This feature makes it possible to display a message to the player and is in fact identical to the design that Willie Crowther adopted in his original adventure game.

Both of these features are illustrated by the segment of the **`SmallRooms.txt`** data file shown in Figure 4 at the top of the next page. If the player is in room 6, and tries to go down, the following two lines in the connection list come into play:

```
DOWN      8/KEYS
DOWN      7
```

The first line is active only if the player is carrying the keys. In this case, the player moves into room 8, which is the beginning of the underground portion of the game. If not, the **`DOWN`** command takes the user to room 7. Because the motion entries include the verb **`FORCED`**, the program prints out the long description for room 7 and then moves the player back to room 6, as shown in the following sample run:

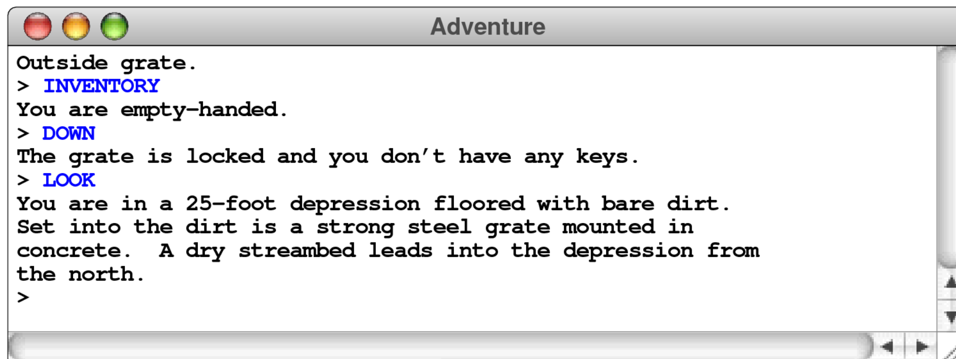


Figure 4. Excerpt from `SmallRooms.txt`

```
6
Outside grate
You are in a 25-foot depression floored with bare dirt.
Set into the dirt is a strong steel grate mounted in
concrete.  A dry streambed leads into the depression from
the north.
-----
NORTH      5
UP         5
DOWN       8/KEYS
DOWN       7

7
Above locked grate
The grate is locked and you don't have any keys.
-----
FORCED     6

8
Beneath grate
You are in a small chamber beneath a 3x3 steel grate to
the surface.  A low crawl over cobbles leads inward to
the west.
-----
UP         6
OUT        6
IN         9
WEST       9
```

It is possible for a single room to use both the locked passage and forced motion options. The **`CrowtherRooms.txt`** file, for example, contains the following entry for the room just north of the curtain in the building:

```
70
Curtain1
-----
FORCED    71/NUGGET
FORCED    76
```

The effect of this set of motion rules is to force the user to room 71 if that user is carrying the nugget and to room 76 otherwise. When you are testing your code for locked and forced passages, you might want to pay particular attention to rooms 70 through 76 in the **`CrowtherRooms.txt`**. These rooms implement the shimmering curtain that marks the end of the game.

The `AdvMotionTableEntry` class

There are several possible strategies one might have chosen to represent the table of connections in each room to its neighbors. In this assignment, you should store the room connections as a list of objects each of which is an instance of the class **`AdvMotionTableEntry`**. The complete definition of this class is included with the starter file and appears in full in Figure 5 on the next page. You could easily have designed this class yourself, but we decided to give it to you to make the assignment just a little bit simpler.

Figure 5. The `AdvMotionTableEntry` class

```
/*
 * File: AdvMotionTableEntry.java
 * ------------------------------
 * This class keeps track of an entry in the motion table.
 */

/**
 * This class is used to store a single entry in the motion table.
 * In keeping with modern object-oriented design, the instance variables
 * of this class are private and can be obtained only through the accessor
 * methods getDirection, getDestinationRoom, and getKeyName.
 */

public class AdvMotionTableEntry {

/**
 * Creates a new motion table entry indicating that moving in the
 * direction specified by the string dir should take the player to
 * the specified room.  If the key parameter is not null, it specifies
 * the name of an object that the player must be carrying to travel
 * along this passage.
 * @param dir The verb indicating the direction of motion (e.g., WEST)
 * @param room The number of the room to which that passage leads
 * @param key The name of an object needed to unlock that passage
 */
   public AdvMotionTableEntry(String dir, int room, String key) {
      direction = dir.toUpperCase();
      destinationRoom = room;
      keyName = (key == null) ? null : key.toUpperCase();
   }

/**
 * Returns the direction name from a motion table entry.
 * @return The string specifying the direction of motion
 */
   public String getDirection() {
      return direction;
   }

/**
 * Returns the room number to which a particular direction leads.
 * @return The number of the destination room
 */
   public int getDestinationRoom() {
      return destinationRoom;
   }

/**
 * Returns the name of the object required for travel along a locked
 * passage, or null if the passage is always available.
 * @return The name of the object used as a key, or null if none
 */
   public String getKeyName() {
      return keyName;
   }

/* Private instance variables */
   private String direction;
   private int destinationRoom;
   private String keyName;
}
```

Section 4
The `AdvObject` Class

The **`AdvObject`** class keeps track of the information about an object in the game. The amount of information you need to maintain for a given object is considerably less than you need for rooms, which makes both the internal structure and its external representation as a data file much simpler. The entries in the object file consist of three lines indicating the word used to refer to the object, the description of the object that appears when you encounter it, and the room number in which the object is initially located. For example, the data file **`SmallObjects.txt`** looks like this:

```
KEYS
a set of keys
3

LAMP
a brightly shining brass lamp
8

ROD
a black rod with a rusty star
12
```

This file indicates that the keys start out in room 3 (inside the building), the lamp initially resides in room 8 (beneath the grating), and the rod can be found in room 12 (the debris room). The entries in the file may be separated with blank lines for readability, as these are here; your implementation should work equally well if these blank lines are omitted.

The objects, of course, will move around in the game as the player picks them up or drops them. Your implementation must therefore provide a facility for storing objects in a room or in the user's inventory of objects. The easiest approach is to use an **`ArrayList`**, which makes it easy to add and remove objects. Short descriptions of the methods in the **`AdvObject`** class appear in Figure 6.

Figure 6. Methods in the `AdvObject` class

`getName()`	Returns the object name, which is the word used to refer to it.
`getDescription()`	Returns the one-line description of the object. This description should start with an article, as in "a set of keys" or "an emerald the size of a plover's egg."
`getInitialLocation()`	Returns the initial location of the object. If this method returns 0, that is taken to mean that the player is holding it.
`AdvObject.readObject(`*rd***`)`**	Creates a new object by reading its data from the specified reader. If no data is left in the reader, this method returns **`null`**.

Section 5
Implementing the Adventure Game

As noted in the introduction to this assignment, implementing the `Adventure` class represents the lion's share of the work. Before you start in on the code, it will simplify your life considerably if you spend some time thinking about the data structures you need and what the overall decomposition looks like. The most relevant model is the teaching machine described in Handout #53, but there are several important differences that you will have to keep in mind.

The `run` method for the `Adventure` class must execute each of the following steps:

1. Ask the user for the name of an adventure, which indicates what data files to use
2. Read in the data files for the game into an internal data structure
3. Play the game by reading and executing commands entered by the user

Asking the user for the name of the adventure is nothing more than a call to `readLine`. The other two phases are substantial enough to warrant subsections of their own.

Reading in the data files

Once you have the name of the adventure, the next phase in the program is to read in the data files that contain all the information necessary to play the game. As noted in the section entitled "Overview of the data files" on page 3, every adventure game is associated with three text files whose names begin with the adventure name. For example, if the adventure name is `Crowther`, these files are named `CrowtherRooms.txt`, `CrowtherObjects.txt`, and `CrowtherSynonyms.txt`. The formats of the first two files have already been described in the discussion of the `AdvRoom` and `AdvObject` classes. Each of those classes, moreover, includes a static method that reads the data for one room or one object from a `BufferedReader`. All the `Adventure` class has to do, therefore, is

1. Open the appropriate file to obtain the `BufferedReader` object.
2. Create an empty data structure for the rooms or objects, as appropriate.
3. Call `AdvRoom.readRoom` or `AdvObject.readObject` to read in a new value.
4. Add the new room or object to your data structure.
5. Repeat steps 3 and 4 until `readRoom` or `readObject` returns `null`.
6. Close the reader.

The rooms file must be present in every adventure, and your program should print an appropriate error message if that file is missing. If the objects file is missing—as it is for the `Tiny` adventure—your program should simply assume that there are no objects.

The only file whose format you haven't seen is the synonyms file, which is used to define abbreviations for commands and synonyms for the existing objects. The synonym file consists of a list of lines in which one word is defined to be equal to another. The `CrowtherSynonyms.txt` file, for example, appears in Figure 7. This file shows that you can abbreviate the `INVENTORY` command to `I` or the `NORTH` command to `N`. Similarly, the

Figure 7. The `CrowtherSynonyms.txt` file

```
Q=QUIT
L=LOOK
CATCH=TAKE
RELEASE=DROP
I=INVENTORY
N=NORTH
S=SOUTH
E=EAST
W=WEST
U=UP
D=DOWN
BACK=OUT
GOLD=NUGGET
BAG=COINS
NEST=EGGS
BOTTLE=WATER
```

user can type **`GOLD`** to refer to the object defined in the object file as **`NUGGET`**. As with the objects file, the synonyms file is optional. If it doesn't exist, your program should simply assume that there are no synonyms.

The hard part of reading the data files is not the operational aspects of reading lines from a data file and dividing the line up into pieces, but rather designing the data structure into which you store the data. Each line of the synonyms file consists of two strings separated by an equal sign. You can separate the string into its component pieces in any of a number of ways, but you also have to figure out how you want to store the information so that it is useful to the program.

Executing commands

Once you have read in the data, you then need to play the game. The user will always start in room 1 and then move around from room to room by entering commands on the console. The process or reading a command consists of the following steps:

1. Read in a line from the user.
2. Break the line up into a verb representing the action and an object (if specified) indicating the target of that action. In the game you have to write, the object is relevant only for the **`TAKE`** and **`DROP`** commands, but your extensions might add other verbs that take objects as well. In this phase, you should make sure to convert the words to uppercase and check the synonyms table to ensure that you're working with the canonical form of each word. For example, if the user types **`RELEASE GOLD`**, your program should decide that the verb is **`DROP`** and the object is **`NUGGET`**.
3. Decide what kind of operation the verb represents. If the word appears in the motion table for some room, then it is a motion verb. In that case, you need to look it up in the motion table for the current room and see if it leads anywhere from the current room. If it isn't a motion verb, the only legal possibilities (outside of any extensions you write) is that it is one of the six built-in action verbs described in Figure 8: **`QUIT`**, **`HELP`**, **`LOOK`**, **`INVENTORY`**, **`TAKE`**, and **`DROP`**. In you have an action verb, you have to call a method that implements the appropriate action, as outlined in the following section. In any other case, you need to tell the user that you don't understand that word.

Figure 8. The built-in action verbs

QUIT	This command signals the end of the game. Your program should stop reading commands and exit from the run method.
HELP	This command should print instructions for the game on the console. You need not duplicate the instructions from the stub implementation exactly, but you should certainly give users an idea of how your game is played. If you make any extensions, you should describe them in the output of your **HELP** command so that we can easily see what exciting things we should look for.
INVENTORY	This command should list what objects the user is holding. If the user is holding no objects, your program should say so with a message along the lines of “You are empty-handed.”
LOOK	This command should type the complete description of the room and its contents, even if the user has already visited the room.
TAKE *obj*	This command requires a direct object and has the effect of taking the object out of the room and adding it to the set of objects the user is carrying. You need to check to make sure that the object is actually in the room before you let the user take it.
DROP *obj*	This command requires a direct object and has the effect of removing the object from the set of objects the user is carrying and adding it back to the list of objects in the room. You need to check to make sure that the user is carrying the object.

In previous versions of this assignment, I’ve insisted that students implement the action verbs be defining commands as a Java class hierarchy. While that strategy is certainly more extensible, it is clearly overkill for an assignment with just six commands. It’s much easier to use a cascading **if** statement that first checks if the verb is **"QUIT"**, then checks to see if it’s **"HELP"**, and so on.

Section 6
Strategy and Tactics

Even though the adventure program is big, the good news is that you do not have to start from scratch. You instead get to start with a complete program that solves the entire assignment because each of the classes you need to write is implemented as a subclass of a library stub that performs all of the necessary functions. Your job is simply to replace all of the stubs with code of your own. In your final version, the implementation of **`Adventure`** should be a direct subclass of **`ConsoleProgram`** and the **`AdvRoom`** and **`AdvObject`** classes should not specify a superclass at all.

The following suggestions should enable you to complete the program with relatively little trouble:

- *Start as soon as you finish NameSurfer.* This assignment is due in less than two weeks, which is a relatively short amount of time for a project of this size. If you wait until the day before this assignment is due, it will be impossible to finish.
- *Get each class working before you start writing the next one.* Because the starter project supplies magic superclasses that implement each of the classes you need to write, you don't have to get everything working before you can make useful progress. Work on the classes one at a time, and debug each one thoroughly before moving on to the next. My suggestion is to start with **`AdvObject`** and **`AdvRoom`**, and then to move on to the more difficult implementation of **`Adventure`** itself.
- *Use the smaller data files for most of your testing.* Don't try to test your code on the **`Crowther`** data files. These files take time to read in and are complicated only because of their scale. The **`Tiny`** data files are appropriate for the basic functionality, and the **`Small`** data files have examples of the required features. When you finish your implementation, it makes sense to try the larger data files just to make sure everything continues to work in the larger context.
- *Test your program thoroughly against the handout and the version that uses the magic superclasses.* When you think you've finished, go back through the handout and make sure that your program meets the requirements stated in the assignment. Look for special cases in the assignment description and make sure that your program handles those cases correctly. If you're unsure about how some case should be handled, play with the version containing the stub code and make sure that your program operates in the same way.

Section 7
Administrative Rules

Project teams

As I will discuss in class, you are encouraged to work on this assignment in teams of two, although you are free to work individually as well. Each person in a two-person team will receive the same grade, although individual late-day penalties will be assessed as outlined below. If you want to work with someone else who has the same section leader, you don't need any special approval. If you want to work with a student in someone else's section, you need to get the approval of both section leaders and then send evidence of that approval—along with the name of the section leader who has agreed to grade the project—to both section leaders and the TA (`jkeeshin@cs.stanford.edu`).

From time to time, project teams don't work out too well. If you are having problems with your teammate, the two of you should come and talk to me about it so that we can work out whatever difficulties have emerged.

Grading

Given the timing of the quarter, your assignment will be evaluated by your section leader without an interactive grading session.

Due dates and late days

As noted on the first page of this handout, the final version of the assignment is due on Monday, June 4. You may use late days on this assignment, except that the days are now *calendar* days rather than *class* days (which makes sense given that class isn't meeting). If you submit the assignment by 5:00P.M. on Tuesday the 5th, you use up one day late, and so forth. All Adventure assignments, however, must be turned in by 5:00P.M. on Thursday, June 7, so that your section leaders will be able to grade it.

On the Adventure assignment, late-day accounts are calculated on an individual basis. Thus, if you have carefully saved up a late day but your partner has foolishly squandered his or hers early in the quarter, you would not be penalized if the assignment came in on Tuesday, but your partner would.