Eric Roberts and Jerry Cain
CS 106J

Handout #44
May 24, 2017

Assignment #6—Adventure

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

Due: Wednesday, June 7, 5:00 P.M.
Last possible submission date: Friday, June 9, 5:00 P.M.
Note: This assignment may be done in pairs

Welcome to the final assignment in CS 106J! 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.

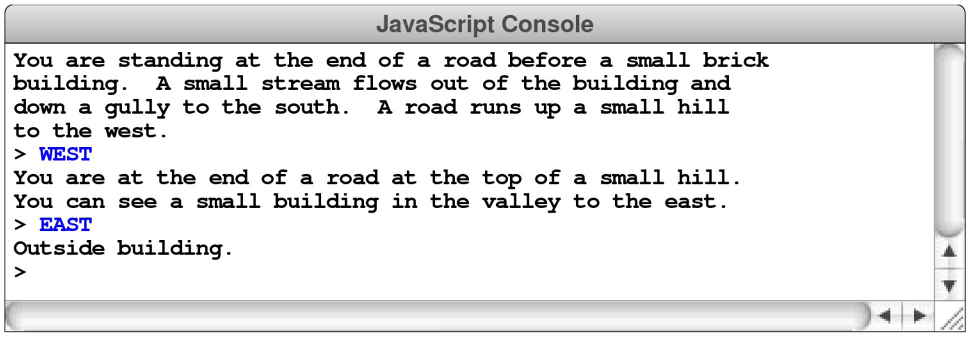
Because this assignment is large and detailed, it takes quite a bit of writing to describe it all. This handout contains everything you need to complete the assignment, along with a considerable number of hints and strategic suggestions. To make it easier to read, the document is divided into the following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

1. Overview of the adventure game 2
2. The **`Adventure.js`** program 5
3. The **`AdvRoom`** and **`AdvMotionTableEntry`** classes 6
4. The **`AdvObject`** class 11
5. The **`AdvGame`** class 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 large as you might think. If you start early and follow the suggestions in the "Strategy and tactics" section, things should work out well.

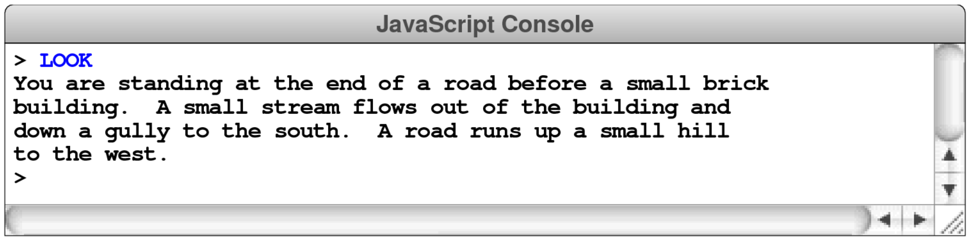
Section 1
Overview of the Adventure Game

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:



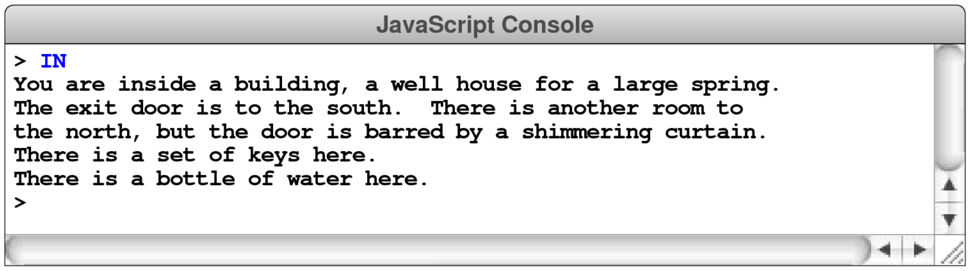
```
You are standing at the end of a road before a small brick
building.  A small stream flows out of the building and
down a gully to the south.  A road runs up a small hill
to the west.
> WEST
You are at the end of a road at the top of a small hill.
You can see a small building in the valley to the east.
> EAST
Outside building.
>
```

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:



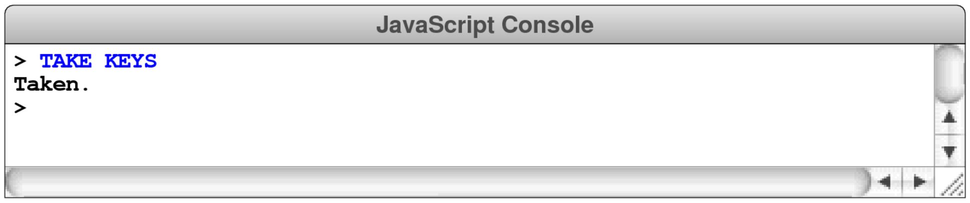
```
> LOOK
You are standing at the end of a road before a small brick
building.  A small stream flows out of the building and
down a gully to the south.  A road runs up a small hill
to the west.
>
```

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



```
> IN
You are inside a building, a well house for a large spring.
The exit door is to the south.  There is another room to
the north, but the door is barred by a shimmering curtain.
There is a set of keys here.
There is a bottle of water here.
>
```

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:



JavaScript Console

```
> TAKE KEYS
Taken.
>
```

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.

In these examples, user input appears in uppercase so that it is easier to see. Your program should recognize commands in lowercase or in some combination of the two.

Overview of the data files

Like the teaching machine program in Chapter 8, 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 between one and three data files, all of which 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 files:

- `Adventure.js`—This file contains the main function and all the `import` statements. The code provided in the starter file is complete, and you shouldn't need to change anything in this file at all.
- `AdvGame.js`—This file defines the `AdvGame` class, which implements the game. The class exports two methods: the `AdvGame` factory method which creates the internal data structures from the data files and the `play` method called by the main program to play the game. This class is the most complex one in the assignment and is yours to write, although this handout offers suggestions for how to decompose the problem in the section on "Strategy and tactics." The `AdvGame.js` file also exports the function `readSynonymFile`, which reads a map of word equivalences from a data file.
- `AdvRoom.js`—This file defines the `AdvRoom` class, which represents a single room in the game. This class is also yours to write. The methods you need to export are all listed in the comments. All you have to do is implement them.
- `AdvMotionTableEntry.js`—This file defines a simple class that encapsulates the information the game needs to keep track of a direction, a destination room, and an optional special object required for travel. This class has been implemented for you and is supplied in its finished form in the starter folder. Although there may be some utility in having you write the `AdvMotionTableEntry` class, it also serves as a good example that you can use as a model for your own classes.
- `AdvObject.js`—This file defines the `AdvObject` class, which represents an objects in the game. As with the `AdvRoom` class, you have to implement this class although the exported methods are specified in the comments.
- `AdvMagicStub.js`—This file contains complete implementations of all the classes you have to write, but that implementation is written in a compiled form that will offer no assistance whatever in writing your own solution. The value of `advMagicStub.js` is that it ensures that the Adventure game works from the very first moment that you download it from the web site. Your job is to replace the functions provided by `AdvMagicStub.js` one at a time until the program is running only your own code.

The structure of each of these files is described in detail in one of the following sections.

Section 2
The **Adventure.js** File

The main program lives in a file called **Adventure.js**, which we have supplied in its complete form. The contents of the **Adventure.js** file appear in Figure 1. The main program has the following responsibilities:

1. Ask the user for the name of an adventure, which indicates what data files to use
2. Call the factory method for **AdvGame** to initialize the game and read the data files
3. Invoke the method **game.play** to play the game

The only part of the **Adventure.js** file that is likely to seem unfamiliar is the process it goes through to read a line from the console. In most languages, the code would simply ask the user for the name of the adventure, wait for the user to enter a line, and then continue on. JavaScript does not allow a process to wait but instead relies on events and callback functions. The main program defines a function named **callback** and passes that function to the **console.requestInput** method, as described in section 5.7 of the text. When the user enters a line, JavaScript invokes the **callback** function, passing in the input line. If that line contains the name of an adventure game that the **AdvGame** factory method can read, the **callback** function invokes the **play** method on the resulting object. If not, **callback** reports the failure and gives the user another chance. You will have to write similar code in **AdvGame.js** to read the user's commands.

Figure 1. The Adventure.js starter file

```
/*
 * File: Adventure.js
 * ------------------
 * This program plays the CS 106J Adventure game.
 */

import "file";
import "stub";
import "AdvGame.js";
import "AdvRoom.js";
import "AdvObject.js";
import "AdvMotionTableEntry.js";
import "AdvMagicStub.js";

/* Main program */

function Adventure() {
   console.log("Welcome to Adventure!");
   var callback = function(name) {
      var game = AdvGame(name);
      if (game === null) {
         console.log("That is not the name of a valid adventure.");
         console.requestInput("Enter the name of the adventure: ", callback);
      } else {
         game.play();
      }
   };
   console.requestInput("Enter the name of the adventure: ", callback);
}
```

Section 3
The `AdvRoom` and `AdvMotionEntry` Classes

The **`AdvRoom`** class represents an individual room in the game. The **`AdvRoom.js`** starter file includes a definition of the **`AdvRoom`** factory method, which begins—at least in the starter file—with the following line:

```
var room = AdvRoomMagic();
```

This line defines the variable **`room`** as the result of calling **`AdvRoomMagic`**, which creates the entire data structure for the room but keeps the details of that structure hidden inside the stubs. As you work through the implementation, you will add new methods to the **`room`** object that replace the magical versions provided by **`AdvRoomMagic`**. When you have finished defining all the necessary methods, you can change the first line to the following one, which creates an empty structure that the subsequent code fills in:

```
var room = { };
```

Each room in the game is characterized by the following properties:

- A room name, which is a string without any spaces used to identify the room
- A short description, which is a one-line string identifying the room
- A long description, which is an array of lines describing the room
- An array 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`** class stores this information in fields within each object and not within the closure of the **`AdvRoom`** factory method. Closures prohibit external access to the values of the field, which is ordinarily a good thing. In this case, however, the **`AdvRoomMagic`** class needs access to these fields and must know the correct field names. Those names are given in the starter file, but their values are all initialized to **`null`**, as shown in Figure 2.

The field definitions in the starter file are followed by a list of the methods that the **`AdvRoom`** class must export. Each of these methods includes a descriptive comment, but the assignment statement that defines the method has been commented out. Your job is to replace each of these methods—preferably one at a time—with your own code that implements the necessary operations. For easier reference, a short description of each method appears in Figure 3 at the top of the next page.

Figure 2. Data fields defined by the `AdvRoom` starter file

```
/* Fields */

   room.name = null;                /* The name of the room                 */
   room.shortDescription = null;    /* A one-line short description         */
   room.longDescription = null;     /* An array for the long description    */
   room.objects = null;             /* The objects in this room             */
   room.visited = null;             /* A flag indicating already visited    */
   room.motionTable = null;         /* Array of AdvMotionTableEntry objects */
```

Figure 3. Methods in the `AdvRoom` class

`getRoomName()`	Returns the room name.
`getShortDescription()`	Returns the one-line description of the room.
`getLongDescription()`	Returns an array of strings containing 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 a flag indicating that this room has been visited.
`hasBeenVisited()`	Returns `true` if the room has previously been visited.
`getMotionTable()`	Returns the motion table array associated with this room.

The rooms data file

As in the teaching machine example in Chapter 8, 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 `AdvRoom.js` implementation is to replace the magical definition of the `readRoomsFile` with one that reads the data from the file whose name is formed by concatenating the name of the adventure with the string `"Rooms.txt"`. The result of the `readRoomsFile` function is a JavaScript map in which the keys are the room names and the values are the `AdvRoom` objects that contain the data.

At first glance, the data files for rooms look almost exactly like those for the teaching machine. For example, `TinyRooms.txt` looks like this:

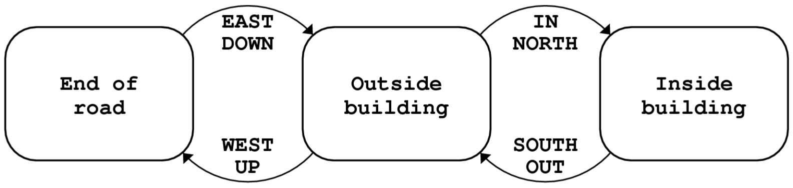
```
OutsideBuilding
Outside building
You are standing at the end of a road before a small brick
building.  A small stream flows out of the building and
down a gully to the south.  A road runs up a small hill
to the west.
-----
WEST       EndOfRoad
UP         EndOfRoad
NORTH      InsideBuilding
IN   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

EndOfRoad
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       OutsideBuilding
DOWN       OutsideBuilding

InsideBuilding
Inside building
You are inside a building, a well house for a large spring.
The exit door is to the south.
-----
SOUTH      OutsideBuilding
OUT        OutsideBuilding
```

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

1. The rooms are named rather than numbered, which makes the files easier to write.
2. The title line is missing (the **`TeachingMachine.js`** program requires a course title on the first line).
3. 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 all minor. Even in the teaching machine example, the question numbers were stored in a map so that the course designer could choose any convenient numbering scheme. Storing room names in a map is just as easy.

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 rooms file, locked passages are indicated by including an object name after a slash.
- *Forced motion.* If the player ever enters a room in which one of the connections is associated with the motion verb **`FORCED`** (and the player is carrying any object that the **`FORCED`** verb requires to unlock the passage) 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 uses precisely the same strategy that Willie Crowther used in his original implementation.

Examples of each of these features appear in the excerpt from the **`SmallRooms.txt`** data file shown in Figure 4 at the top of the next page. If the player is in the room named

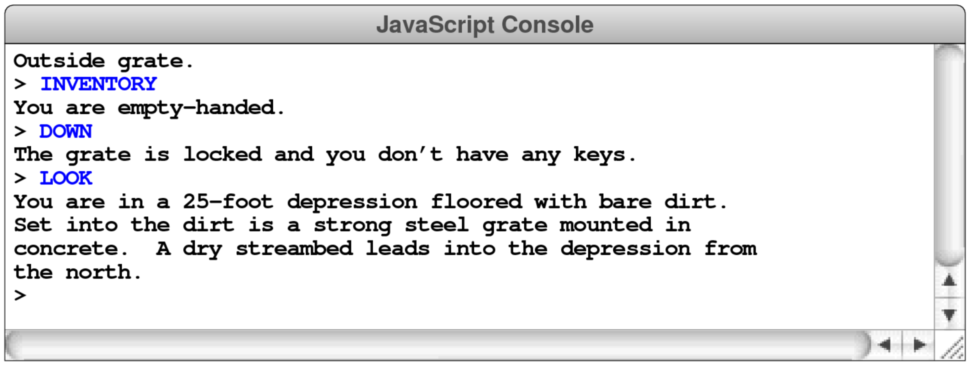
Figure 4. Excerpt from `SmallRooms.txt`

```
OutsideGrate
Outside grate
You are in a 20-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      SlitInRock
UP         SlitInRock
DOWN       BeneathGrate/KEYS
DOWN       MissingKeys

MissingKeys
Missing keys
The grate is locked and you don't have any keys.
-----
FORCED     OutsideGrate

BeneathGrate
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         OutsideGrate
OUT        OutsideGrate
IN         CobbleCrawl
WEST       CobbleCrawl
```

`OutsideGrate` and tries to go down, the following two lines in the connection list come into play:

```
DOWN        BeneathGrate/KEYS
DOWN        MissingKeys
```

The first line is used only if the player is carrying the keys. In this case, a player holding the keys would move into the room named **`BeneathGrate`**. If the player is not carrying the keys, the **`DOWN`** command takes the user to a room named **`MissingKeys`**. Because the motion entries for the room named **`MissingKeys`** include the verb **`FORCED`**, the program prints out the long description for that room and then moves the player back to the room named **`OutsideGrate`**, as shown in the following sample run:



```
Outside grate.
> INVENTORY
You are empty-handed.
> DOWN
The grate is locked and you don't have any keys.
> LOOK
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.
>
```

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:

```
Curtain1
Curtain1
-----
FORCED     Curtain2/NUGGET
FORCED     MissingTreasures
```

The effect of this set of motion rules is to force the user to the room named `Curtain2` if the user is carrying the nugget and to the room named `MissingTreasures` otherwise. When you are testing your code for locked and forced passages, you might want to pay particular attention to the last eight rooms in the `CrowtherRooms.txt`. These rooms implement the shimmering curtain that marks the end of the game.

The other special case you need to implement is some way to stop the program. The motion entries in the rooms file include a special room name `EXIT`, which is used to stop the game. Your code needs to check for `EXIT` and then stop asking for new commands.

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 an array whose elements are instances of `AdvMotionTableEntry`. The complete definition of this class is included with the starter file and appears in full in Figure 5. You could easily have designed this class yourself, but we decided to give it to you to make the assignment a bit simpler.

Figure 5. The `AdvMotionTableEntry` class

```
/*
 * File: AdvMotionTableEntry.js
 * ----------------------------
 * This class keeps track of an entry in the motion table.  In keeping
 * with modern object-oriented design, the fields of this object can
 * be retrieved only by calling methods.
 */

/*
 * Creates a new motion table entry from the specified components.  The
 * direction parameter is the motion verb, destinationRoom is the name of
 * the room to which that verb takes you, and key is either null or the
 * name of an object the player must be carrying to traverse this passage.
 */

function AdvMotionTableEntry(direction, destinationRoom, key) {
   return {
      getDirection: function() { return direction; },
      getDestinationRoom: function() { return destinationRoom; },
      getKeyName: function() { return key; }
   };
}
```

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 name of the room in which the object is initially located. For example, the data file **`SmallObjects.txt`** looks like this:

```
KEYS
a set of keys
InsideBuilding

LAMP
a brightly shining brass lamp
BeneathGrate

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

This file indicates that the keys start out in the room named **`InsideBuilding`**, the lamp initially resides in the room named **`BeneathGrate`**, and the rod can be found in the room named **`DebrisRoom`**. 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. You may assume that there is only one blank line between each entry and there are no extraneous blank lines at the end of the file.

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 a JavaScript array, which makes it easy to add and remove objects. Short descriptions of the methods in the **`AdvObject`** class appear in Figure 6.

The **`AdvObject`** class also contains a stub implementation of the **`readObjectsFile`** function, which reads in the object data file. As with **`readRoomsFile`**, the result should be a JavaScript map in which the keys are the object names and the values are the objects themselves. If the file describing the objects does not exist (as is true for the Tiny adventure), **`readObjectsFile`** should return an empty object.

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. A location of **`"PLAYER"`** is taken to mean that the player is holding it.

Section 5
The AdvGame Class

The **AdvGame** class contains most of the code you have to write for this assignment. This is the class that assembles the data structures, reads commands from the user, and executes those commands. The starter version of **AdvGame.js** appears in Figure 7, in which both the **AdvGame** class and the **readSynonymsFile** are implemented as stubs.

As noted in the introduction to this assignment, implementing the **AdvGame** 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 what information you need to maintain and how you can best decompose the **play** method into reasonably sized pieces. The most relevant model is the teaching machine described in Chapter 8, but there are several important differences that you will have to keep in mind.

Figure 7. The starter code for the AdvGame.js file

```
/*
 * File: AdvGame.js
 * ----------------
 * This file defines the AdvGame class, which records the information
 * necessary to play a game.
 */

/*
 * Creates an AdvGame object by reading data from a set of data files
 * that begin with the specified prefix:
 *
 *    prefix + "Rooms.txt"      Contains the data for the rooms
 *    prefix + "Objects.txt"    Contains the data for the objects
 *    prefix + "Synonyms.txt"   Contains a table of synonyms
 *
 * Only the Rooms.txt file must exist.  If it is missing, the
 * AdvGame factory function returns null.
 */

function AdvGame(prefix) {

   var game = AdvGameMagic(prefix);

//   game.play = function() { ... add this function ... }

   return game;

}

/*
 * Reads a file containing synonyms so that the user can use any of
 * several words to indicate the same action or object.  The data
 * structure is a JavaScript object in which the key is the synonym
 * and the value is the definition.  If no synonym file exists, this
 * function returns  an empty object.
 */

function readSynonymsFile(filename) {
   return readSynonymsFileMagic(filename);
}
```

The factory method for **`AdvGame`** must read each of the three data files for the adventure and use them to initialize fields in the **`game`** object: one for rooms, one for objects, and one for synonyms. 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 sequence of lines in which one word is defined to be equal to another. The **`CrowtherSynonyms.txt`** file, for example, appears in Figure 8. This file shows that you can abbreviate the **`INVENTORY`** command to **`I`** or the **`NORTH`** command to **`N`**. Similarly, the user can type **`GOLD`** to refer to the object defined in the object file as **`NUGGET`**. If the synonyms file doesn't exist, your program should assume that there are no synonyms.

The **`readSynonymsFile`** function is one of the easiest functions you have to write for this assignment. Each line of the synonyms file consists of two strings separated by an equal sign. You have to separate the string into its component pieces and put each synonym-value pair into a JavaScript map, which is then returned as the value of the function.

Executing commands

Once you have read in the data, you then need to play the game, which is implemented by the **`play`** method. The first thing the **`play`** method has to do is to record the fact that the player always starts in the first room specified in the rooms file, which the code for **`readRoomsFile`** stores in the data structure under the name **`"START"`** as well as the room's own name. You need to look up **`"START"`** in this structure and store the result in a local variable that records the current room. You also have to maintain a local variable that records the objects the player is carrying, which starts out as an empty array.

The next step is to distribute the various objects to their starting positions. To do so, you will need to iterate through the collection of objects, not by position number as you would in an array, but instead by the name of each object. This operation requires a different form of the **`for`** loop, which is not yet described in the text. If the data structure for the objects is stored in **`game.objects`**, you can cycle through each name by writing a **`for`** loop that looks like this:

Figure 8. 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
```

```
for (var name in game.objects) {
   var obj = game.objects[name];
   . . . code to work with that object . . .
}
```

It is important to note that the initial location of an object can be the string `"PLAYER"`, even though this value is usually a room name. If the initial location is `"PLAYER"`, your code should add the object to the array of objects the player is carrying.

Once you've set everything up, you then need to write the code that allows the player to move from room to room by entering commands on the console. The process of reading a command consists of the following steps:

1. Request an input line from the user using `console.requestInput`.
2. When the callback function specified in `requestInput` is called, you need to break the input line up into a verb representing the action and an optional object indicating the target of that action. In the required parts of the assignment, the object is relevant only for the **TAKE** and **DROP** commands, but your extensions might add other verbs that take objects as well. You should also convert the words to uppercase and check your data structures to see if the words are synonyms. For example, if the user types **RELEASE GOLD**, your code should change the verb to **DROP** and the object to **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 9: **QUIT**, **HELP**, **LOOK**, **INVENTORY**, **TAKE**, and **DROP**. If you have an action verb, you have to call a helper function that implements the appropriate action. In any other case, you need to tell the user that you don't understand that word.

Figure 9. The built-in action verbs

QUIT	This command signals the end of the game. Your program should stop reading and executing user commands.
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.

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 have the advantage of starting with a complete program that solves the entire assignment because each of the classes you need to write is implemented by a function in **`AdvMagicStub.js`**. Your job is simply to replace the stubs with code of your own. In your final version, you should be able to delete the import lines for both the **`"stub"`** library and **`"AdvMagicStub.js"`**.

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

- *Start as soon as possible.* 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 stub implementations for 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. In most cases, you can even implement the classes one function at a time, relying on the stub implementation to do the rest of the work. Our suggestion is to start with **`readSynonymsFile`**, which is the easiest method to implement. When you have that working, you can then go on to implement **`AdvObject`** and **`AdvRoom`**, probably in that order. Once you have those working, you can move on to the implementation of **`AdvGame`** itself.
- *Use the smaller data files for most of your testing.* Don't try to test new 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, the web demos, and the version that uses the magic stub implementations.* 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 on the Breakout and Enigma assignments, 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.

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 Wednesday, June 7. 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 Thursday the 8th, you use up one day late, and so forth. All Adventure assignments, however, must be turned in by 5:00P.M. on Friday, June 9, 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 Thursday, but your partner would.