Objects as Maps

Jerry Cain
CS 106AJ
November 9, 2018
slides courtesy of Eric Roberts

The Concept of a Map

- One of the most important applications of JavaScript objects uses them to associate pairs of data values. In computer science, the resulting data structure is called a ***map***.
- Maps associate a simple data value called a ***key*** (most often a string) with a ***value**,* which is often larger and more complex.
- Examples of maps exist everywhere in the real world. A classic example is a dictionary. The keys are the words, and the values are the corresponding definitions.
- A more contemporary example is the World-Wide Web. In this example, the keys are the URLs, and the values are the contents of the corresponding pages.

Maps and JavaScript Objects

- In the context of CS 106AJ, the most obvious example of a map is the JavaScript object, which precisely implements the map concept. The keys are strings, and the values are arbitrary JavaScript values.

- When you use an object as a map, you supply the key as a string expression using the square-bracket notation, as in

  ```
  map[key]
  ```

  If the key is defined in the map, this selection returns the value. If no definition has been supplied, the selection returns the constant **`undefined`**.

- Map selections are assignable, so that you can set the value associated with a key by executing an assignment statement:

  ```
  map[key] = value;
  ```

Iterating Through Keys in an Object

- One of the common operations that clients need to perform when using a map is to iterate through the keys.
- JavaScript supports this operation using an extended form of the **`for`** statement, which has the following form:

```
for (let key in map) {
   let value = map[key];
   . . . code to work with the individual key and value . . .
}
```

- In JavaScript, this extended form of the **`for`** loop can process the keys in any order.

Using Maps in an Application

- Before going on to create new applications of maps, it seems worth going through the example from the text, which uses a map to associate three-letter airport codes with their locations.
- The association list is stored in a text file that looks like this:

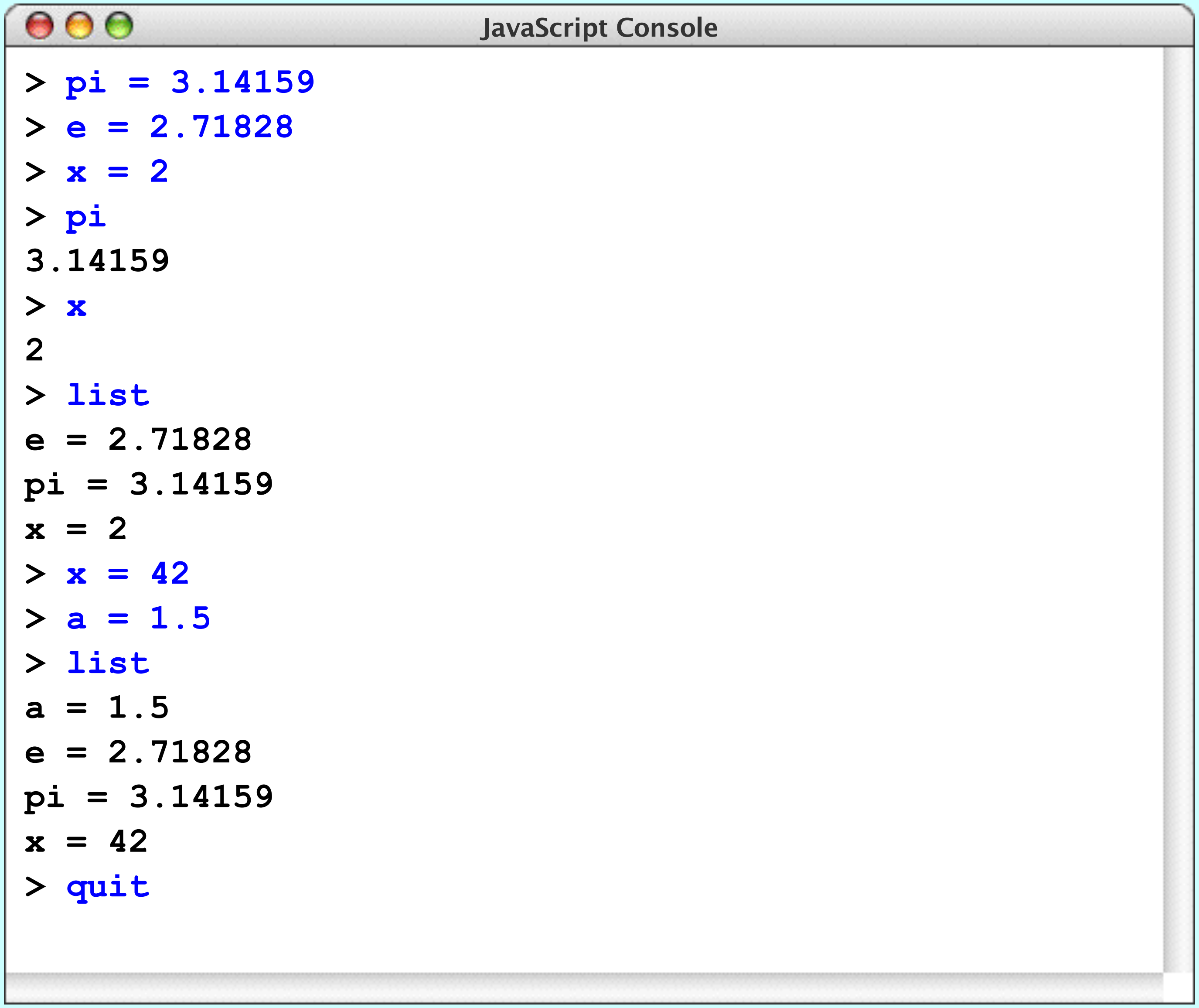
```
ATL=Atlanta, GA, USA
ORD=Chicago, IL, USA
LHR=London, England, United Kingdom
HND=Tokyo, Japan
LAX=Los Angeles, CA, USA
CDG=Paris, France
DFW=Dallas/Ft Worth, TX, USA
FRA=Frankfurt, Germany
    ⋮
```

- The **`AirportsCodes.js`** program shows how to read this file into a JavaScript object and publish its content to the console.

Symbol Tables

- Programming languages make internal use of maps in several contexts, of which one of the easiest to recognize is a ***symbol table**,* which keeps track of the correspondence between variable names and their values.
- The **`SymbolTable.js`** application included in the example programs for this lecture implements a simple test of a symbol table that reads lines from the console, each of which is one of the following commands:
  - A simple assignment statement of the form *var* **`=`** *number*.
  - A variable alone on a line, which displays the variable's value.
  - The command **`list`**, which lists all the variables.
  - The command **`quit`**, which exits from the program.
- The sample run on the next slide illustrates the operation of the **`SymbolTable.js`** program.

Sample Run of `SymbolTable.js`



```
> pi = 3.14159
> e = 2.71828
> x = 2
> pi
3.14159
> x
2
> list
e = 2.71828
pi = 3.14159
x = 2
> x = 42
> a = 1.5
> list
a = 1.5
e = 2.71828
pi = 3.14159
x = 42
> quit
```

Reading from the Console

- Implementing the **`SymbolTable`** application requires two new features that you have not yet seen:
  1. Reading a line from the console.
  2. Iterating through all the keys in the map to implement **`list`**.
- Reading a line from the console is more difficult in JavaScript than it is in most languages, simply because JavaScript depends on an interaction model based on events and callback functions.
- The best strategy for reading a line of user input uses the **`console.requestInput`** method in the following form:

  `console.requestInput(`*prompt***`,`** *callback***`);`**

- If you need to read lines in a loop, the easiest way is to include another **`requestInput`** call in the callback function.

The StateCodes.js Program

```
/**
 * Reads state codes from the user and reports the corresponding
 * state name. Entering a blank line terminates the program.
 */
function StateCodes() {
   let callback = function(text) {
      let lines = JSFileChooser.convertToLineArray(text);
      let stateCodes = parseStateCodes(lines);
      requestInputFromUser(stateCodes);
   };
   JSFileChooser.chooseTextFile(callback);
}
```

The `parseStateCodes` Function

```
/**
 * Reads the specified file and returns a map between keys
 * and values. The format of the data file is a series of
 * lines that look like this:
 *
 *     code=state
 *
 * Once the map has been created, clients can lookup a key by
 * writing stateCodes[code].
 */
function parseStateCodes(lines) {
   let stateCodes = {};
   let line = lines.shift();
   while (line !== undefined) {
      let ep = line.indexOf("=");
      let code = line.substring(0, ep).trim();
      let state = line.substring(ep + 1).trim();
      stateCodes[code] = state;
      line = lines.shift();
   }
   return stateCodes;
}
```

The **requestInputFromUser** Function

```
/**
 * Repeatedly prompts the user for state codes, printing
 * the corresponding state for each one.  The function returns
 * when the user types in the empty string.
 */
function requestInputFromUser(stateCodes) {
   let processResponse = function(response) {
      let code = response.trim();
      if (response !== "") {
         let state = stateCodes[code];
         if (state === undefined) {
            console.log("There is no state with code " + code);
         } else {
            console.log(state);
         }
         console.requestInput("State code: ", processResponse);
      } else {
         console.log("All done!");
      }
   };
   console.requestInput("State code: ", processResponse);
}
```

Maps Can Be Fast

- If you think about the underlying implementation of maps, your first thought is likely to be that JavaScript looks through a list of keys to find the key in question and returns the corresponding value. That approach takes time proportional to the number of keys.
- Maps, however, can be implemented much more efficiently than that. As you will learn if you go on to CS 106B, maps can be implemented so that the result is delivered almost instantly or, more accurately, so that the time required is constant no matter how many keys there are.
- To show that this result might in fact be possible, consider the state code example. If the state names are stored in an 26×26 array in which the indices correspond to the first and second letters in the code, finding the name is simply an array access.

The End