Arrays as Objects

earlier, we introduced the concept of software objects: programming structures that encompass both data (properties) and operations (methods)

- e.g., a string object has properties (e.g., `length`) and methods (e.g., `charAt`) that enable the programmer to easily store and manipulate text

like strings, JavaScript *arrays* are objects that encapsulate multiple values and their associated properties and methods

- unlike strings (which store only text characters), the items in an array can be of any data type
- an array consists of a sequence of items, enclosed in square brackets and separated by commas
- when assigning values to arrays:
  - an item can appear in an array more than once
  - array items can be specified as expressions
  - arrays can be empty

```
responses = ['yes', 'no', 'maybe'];
nums = [1, 2, 3, 2+1, 7*7, 2*5-1];
misc = [1.234, 'foo', 7-5, true, 3, 'foo'];
empty = [ ];
```

Accessing Items in an Array

since an array stores a series of values, its associated memory cell can be envisioned as divided into components, each containing an individual value

	misc[0]	misc[1]	misc[2]	misc[3]	misc[4]	misc[5]
misc	1.234	'foo'	2	true	3	'foo'

array items are assigned sequential indices, allowing programmers to identify an item by its corresponding index

array access is accomplished by specifying the name of the array object, followed by the index of the desired item in brackets

- the first item is `misc[0]`, the second is `misc[1]`, …, the last is `misc[length-1]`
- if an index is specified beyond the scope of an array, the access yields `undefined`

Array Access Example

we now can understand how the `RandomOneOf` function from `random.js` works

- the input to the function is a nonempty array of arbitrary items
- the `RandomInt` function is called to pick a random index from the array (between 0 and the last index, `list.length-1`)
- the bracket notation is used to access the item at that index, which is then returned by the function

```
function RandomOneOf(list)
// Given : list is a nonempty list (array)
// Returns: a random item from the list
{
    var randomIndex;

    randomIndex = RandomInt(0, list.length-1);

    return list[randomIndex];
}
```

Assigning Items in an Array

array items can be assigned values just like any variable

- suppose the array `misc` has been assigned to store the following

```
misc = [1.234, 'foo', 7-5, true, 3, 'foo'];
```

- the assignment `misc[0] = 1000;` would store the value 1000 as the first item in the array, overwriting the value that was previously there

	misc[0]	misc[1]	misc[2]	misc[3]	misc[4]	misc[5]
misc	1000	'foo'	2	true	3	'foo'

if the index in an assignment statement is beyond the array's current length, the array will automatically expand to accommodate the new item

- the assignment `misc[8] = 'oops';` would store 'oops' at index 8

	misc[0]	misc[1]	misc[2]	misc[3]	misc[4]	misc[5]	misc[6]	misc[7]	misc[8]
misc	1000	'foo'	2	true	3	'foo'	undefined	undefined	'oops'

From Strings to Arrays

so far, our Web pages have handled each input from the user as a single string

- this approach is limiting since many programming tasks involve separately processing an arbitrary number of words or numbers entered by the user

- recall the Pig Latin page

- we might want to generalize the page so that it translates entire phrases instead of just words

- that is, the user would enter an arbitrary sequence of words and each word would be translated

- this task would be very difficult using our current set of programming tools

String split Method

JavaScript strings provide a method, `split`, for easily accessing the components of a string

- the only input required by the `split` method is a character (or sequence of characters) that serves as a delimiter for breaking apart the string
- the `split` method separates the string into component substrings at each occurrence of the delimiter, and returns an array consisting of those substrings

```
user = 'Grace Murray Hopper';
arr1 = user.split(' ');       // ASSIGNS arr1 to be the array
                              //   ['Grace', 'Murray', 'Hopper']
```

- as was the case with strings, `/[ … ]/` can be used to specify groups of characters

`phrase.split(' ')`	breaks the string `phrase` into an array of items, delimited by a single space
`phrase.split(': ')`	breaks the string `phrase` into an array of items, delimited by a colon followed by a space
`phrase.split(/[ \t\n,]/)`	breaks the string `phrase` into an array of items, delimited by a single space, tab (`\t`), newline (`\n`), or comma
`phrase.split(/[ \t\n,]+/)`	breaks the string `phrase` into an array of items, delimited by any sequence of spaces, tabs (`\t`), newlines (`\n`), and/or commas

Arrays of Numbers

some applications involve reading and manipulating sequences of numbers

- e.g., suppose we wanted to calculate the average of some number of grades
- we could enter the numbers in one big string, separated by spaces, then
  - use the split function to separate out the individual numbers
  - then, traverse the resulting array and calculate the sum and average of the numbers

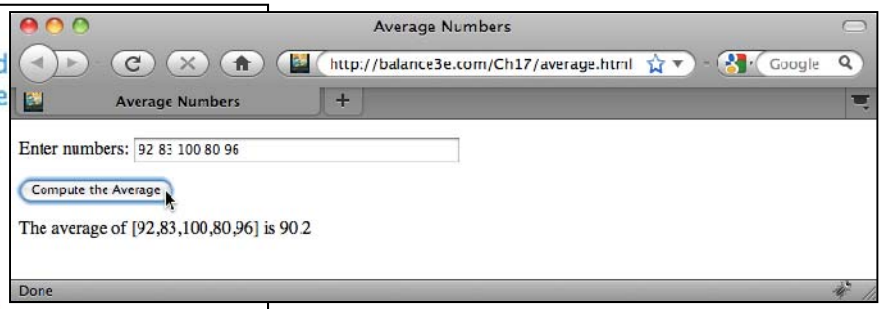
note: even if the input string contains numerical digits, `split` returns each array item as a string (similar to text box/area access)

- `arrays.js` contains a function that traverses an array and converts each item to a number

```
function ParseArray(strArray)
// Assumes: strArray is an array of strings representing numbers
// Returns: a copy of array with items converted to numbers
{
  var numArray, index;

  numArray = [ ];                          // CREATE EMPTY ARRAY TO STORE NUMS

  index = 0;                               // FOR EACH ITEM IN strArray
  while (index < strArray.length) {        //   CONVERT TO NUMBER AND STORE
      numArray[index] = parseFloat(strArray[index]);
      index = index + 1;
  }

  return numArray;                         // FINALLY, RETURN THE NUMS
}
```

Average Page

```
 1. <!doctype html>
 2. <!-- average.html                                    Dave Reed
 3. <!-- This page utilizes the arrays.js library file to store
 4. <!-- an array of numbers and compute their average.
 5. <!-- ================================================
 6.
 7. <html>
 8.  <head>
 9.    <title> Average Numbers </title>
10.    <script type="text/javascript" src="arrays.js"></script>
11.    <script type="text/javascript">
12.      function ShowAvg()
13.      // Assumes: numsBox contains a sequence of numbers
14.      // Results: displays the average of the numbers in outputDiv
15.      {
16.        var str, strArray, numArray;
17.
18.        str = document.getElementById('numsBox').value;
19.        strArray = str.split(/[ ,]+/);   // SPLIT STRING INTO AN ARRAY
20.        numArray = ParseArray(strArray); // CONVERT ARRAY ELEMENTS TO NUMS
21.
22.        document.getElementById('outputDiv').innerHTML =
23.            'The average of [' + numArray + '] is ' + Average(numArray);
24.      }
25.    </script>
26.  </head>
27.
28.  <body>
29.    <p>
30.      Enter numbers: <input type="text" id="numsBox" size=40 value="">
31.    </p>
32.    <p>
33.      <input type="button" value="Compute the Average" onclick="ShowAvg();">
34.    </p>
35.    <div id="outputDiv"></div>
36.  </body>
37. </html>
```



this page averages an arbitrary number of grades (entered as a string in a text box)

- utilizes `split` to split the string into an array of strings
- utilizes `ParseArray` to convert into an array of numbers

Example: Dice Stats

recall our web page for simulating repeated dice rolls and recording the number of times that a specific total was obtained

- statistical analysis predicts that, given a large number of dice rolls, the distribution of totals will closely mirror the percentages listed below

Dice total	Likelihood of Obtaining That Total
2	2.8%
3	5.6%
4	8.3%
5	11.1%
6	13.9%
7	16.7%
8	13.9%
9	11.1%
10	8.3%
11	5.6%
12	2.8%

Approach 1: Separate Counters

to obtain a valid distribution of dice totals, we would need to simulate a large number of rolls and simultaneously count the occurrences of every total

- this can be accomplished by defining 11 counters, each corresponding to a particular total
- however, this would be *extremely* tedious

```
count2 = 0;                          // INITIALIZE EACH
count3 = 0;                          // COUNTER
count4 = 0;                          // (CORRESPONDING TO
.                                    // THE NUMBER OF 2'S,
.                                    // 3'S, 4'S, ... 12'S)
.
count12 = 0;

rep = 0;                             // INITIALIZE rep COUNTER
while (rep < 1000) {                 // AS LONG AS ROLLS REMAIN
     roll = RandomInt(1, 6) + RandomInt(1, 6); // GET NEXT ROLL
     if (roll == 2) {                // IF ROLLED 2,
           count2 = count2 + 1;      // ADD 1 TO 2'S COUNT
     }
     else if (roll == 3) {           // ELSE IF ROLLED 3,
           count3 = count3 + 1;      // ADD 1 TO 3'S COUNT
     }
     else if (roll == 4) {           // ELSE IF ROLLED 4,
           count4 = count4 + 1;      // ADD 1 TO 4'S COUNT
     }
     .                               // SIMILAR CASES FOR
     .                               // ROLLS 5 THROUGH 11
     .
     else if (roll == 12) {          // ELSE IF ROLLED 12,
           count12 = count12 + 1;    // ADD 1 TO 12'S COUNT
     }

     rep = rep + 1;                  // GO ON TO NEXT REP
}
```

- requires separate assignment statements for all 11 counters
- requires a cascading if-else statement with 11 cases
- not easily generalized – what if we wanted to use 8-sided dice?

Approach 2: Array of Counters

instead of representing each counter as a separate variable, we can define the counters as items in an array

- all 11 counters can be stored in an array and initialized via a single assignment statement
- any individual counter can be accessed and updated via its corresponding index
  - since the first possible total is 2, its count is stored in index 0
  - the next possible total, 3, has its count stored in index 1
  - …
  - for an arbitrary `roll`, its count is stored in index `roll-2`
- the resulting code is shorter, simpler, and easier to generalize

```
count = [0,0,0,0,0,0,0,0,0,0,0];                    // INITIALIZE ALL COUNTERS

 rep = 0;                                           // INITIALIZE REP COUNTER
 while (rep < 1000) {                               // AS LONG AS ROLLS REMAIN
      roll = RandomInt(1, 6) + RandomInt(1, 6);     //  GET RANDOM ROLL OF DICE
      count[roll-2] = count[roll-2] + 1;            //  ADD 1 TO COUNTER
      rep=rep+1;                                    //  GO ONTO NEXT REP
}
```

Dice Stats Page

code for maintaining statistics on repeated dice rolls can be integrated into a Web page

- the number of rolls to be simulated is entered by the user into a text box
- a button is defined to call the code for repeatedly simulating the roll and maintaining stats, with the text box contents as input
- the numbers of rolls for each total are then displayed in a page division