Abstraction

Abstraction is the process of ignoring minutiae and focusing on the big picture

- in modern life, we are constantly confronted with complexity
- we don't necessarily know how it works, but we know how to use it

  e.g., how does a TV work? a car? a computer?

we survive in the face of complexity by abstracting away details

- to use a TV/car/computer, it's not important to understand the inner workings
- we ignore unimportant details and focus on those features relevant to using it
  
  e.g., TV has power switch, volume control, channel changer, ...

JavaScript functions (like Math.sqrt) provide computational abstraction

- a function encapsulates some computation & hides the details from the user
- the user only needs to know how to call the function, not how it works

- Chapter 7 introduced simple user-defined functions
  
  could encapsulate the statements associated with a button, call the function as needed
to write general-purpose functions, we can extend definitions to include:
1) parameters, 2) local variables, and 3) return statements

```
function FUNCTION_NAME(PARAMETER1, PARAMETER2, ..., PARAMETERn)
// Assumes: DESCRIPTION OF ASSUMPTIONS MADE ABOUT PARAMETERS
// Returns: DESCRIPTION OF VALUE RETURNED BY FUNCTION
{
    var LOCAL1, LOCAL2, ..., LOCALn;
    STATEMENTS_TO_PERFORM_THE_DESIGNED_COMPUTATION
    return OUTPUT_VALUE;       // optional
}
```

- **parameters** are variables that correspond to the function’s inputs (if any)
  - parameters appear in the parentheses, separated by commas

- **local variables** are temporary variables that are limited to that function only
  - if require some temporary storage in performing calculations, then declare local variables using the keyword `var`, separated by commas
  - a local variable exists only while the function executes, so no potential conflicts with other functions

- **a return statement** is a statement that specifies an output value
  - consists of the keyword `return` followed by a variable or expression
we have seen that variables are useful for storing intermediate steps in a complex computation
- within a user-defined function, the programmer is free to create new variables and use them in specifying the function’s computation
- however, by default, new variables used in a function are *global* (i.e., exist and are accessible anywhere in the page)

- *but what if the same variable name is already used elsewhere?*

to avoid name conflicts, the programmer should declare temporary variables to be *local*
- a variable declaration is a statement that lists all local variables to be used in a function (usually the first statement in a function)
- general form:

```
var LOCAL_1, LOCAL_2, . . . , LOCAL_n;
```
consider a simple ESP test

1. user thinks of a number between 1-4
2. clicks on the button to see the computer's pick

number is declared local to PickNumber
• only exists while the function executes
most of the predefined function we have considered expect at least one input
e.g., Math.sqrt takes a number as input, and returns its square root as output
\[
\text{Math.sqrt}(9) \Rightarrow 3
\]

e.g., Math.max takes two numbers as inputs, and returns the maximum as output
\[
\text{Math.max}(7, 3) \Rightarrow 7
\]

in English, the word *parameter* refers to some aspect of a system that can be
varied in order to control its behavior

- in JavaScript, a parameter is a variable (declared inside the function's parentheses)
  whose value is automatically initialized to the corresponding input value when the
  function is called
- parameters allow the same function to perform different (but related) tasks when
  called with different input values
better design: have a button for each guess

- to guess 1, the user clicks the 'Guess 1' button
- instead of 4 different functions (that behave similarly), have 1 function with a parameter

the number corresponding to the guess is passed in as an input, displayed as the guess
Multiple Inputs

if a function has more than one input,
- parameters in the function definition are separated by commas
- input values in the function call are separated by commas
- values are matched to parameters by order
  1st input value in the function call is assigned to the 1st parameter in the function
  2nd input value in the function call is assigned to the 2nd parameter in the function
  ...

```javascript
function OldMacVerse(animal, sound)
  // Assumes: animal is the name of an animal, sound is the sound it makes
  // Results: displays a verse of the song "Old MacDonald Had a Farm" in outputDiv
  
  document.getElementById('outputDiv').innerHTML = 
  '<p>Old MacDonald had a farm, E-I-E-I-O.<br>' +
  'And on that farm he had a ' + animal + ', E-I-E-I-O.<br>' +
  'With a ' + sound + ' here, and a ' + sound + ' there,<br>' +
  'here a ' + sound + ', there a ' + sound +
  ', everywhere a ' + sound + '.<br>' +
  'Old MacDonald had a farm, E-I-E-I-O.<p>;
```

<input type="button" value="Cow Verse"
onclick="OldMacVerse('cow', 'moo');">
Parameters and Locals

parameters play an important role in functions

- they facilitate the creation of generalized computations
- i.e., the function defines a formula, but certain values within the formula can differ each time the function is called

parameters are special instances of local variables

- when the function is called, memory cells are allocated for the parameters and each input from the call is assigned to its corresponding parameter
- once a parameter has been assigned a value, you can refer to that parameter within the function just as you would any other variable
- when the function terminates, the parameters “go away,” and their associated memory cells are freed

parameters are declared and initialized automatically

- do not declare them as local variables
Functions with Return

displaying results using an INNERHTML assignment or `alert` is OK for some functions

- for full generality, we need to be able to return an output value, which can then be used in other computations

  e.g.,

  ```javascript
  number = Math.sqrt(9);
  cm = InchesToCentimeters(in);
  ```

  a return statement can be added to a function to specify its output value

  - when the return statement is reached, the variable or expression is evaluated and its value is returned as the function's output

  general form:

  ```javascript
  return OUTPUT_VALUE;
  ```
Function Libraries

functions such as InchesToCentimeters can be added to the HEAD of a page
- tedious if the function is to be used in many pages
- involves creating lots of copies that all must be maintained for consistency

the alternative for general purpose functions is to place them in a library file
- a library file is a separate text file that contains the definitions of one or more JavaScript functions
- it can be loaded into any page by adding an HTML element to the HEAD

```html
<script type="text/javascript" src="LIBRARY_FILENAME"></script>
```

advantages of library files:
- avoids duplication (only one copy of the function definition)
- makes it easy to reuse functions (simply load the library file into any page)
- makes it easy to modify functions (a single change to the library file automatically affects all pages that load the library)
the convert.js library file is loaded into the page
- this makes the InchesToCentimeters function accessible within the page
  - since ConvertToCms is specific to this page, it directly in the HEAD (as opposed to a library file)
Errors to Avoid

When beginning programmers attempt to load a JavaScript code library, errors of two types commonly occur:

1. if the SCRIPT tags are malformed or the name/address of the library is incorrect, the library will fail to load
   - this will not cause an error in itself, but any subsequent attempt to call a function from the library will produce
     
     ```
     Error: Object Expected (using Internet Explorer)
     or
     Error: XXX is not a function (using Firefox), where XXX is the entered name
     ```

2. when you use the SRC attribute in a pair of SCRIPT tags to load a code library, you cannot place additional JavaScript code between the tags
   - think of the SRC attribute as causing the contents of the library to be inserted between the tags, overwriting any other code that was erroneously placed there

     ```
     <script type="text/javascript" src="FILENAME">
       ANYTHING PLACED IN HERE WILL BE IGNORED
     </script>
     ```

   - if you want additional JavaScript code or another library, you must use another pair of SCRIPT tags
the `random.js` library contains useful functions for generating random values.

<table>
<thead>
<tr>
<th>Function</th>
<th>Inputs</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>RandomNum</td>
<td>Two numbers (low and high limits of a range); e.g., RandomNum(2, 4.5)</td>
<td>A random number from the range low (inclusive) to high (exclusive)</td>
</tr>
<tr>
<td>RandomInt</td>
<td>Two integers (low and high limits of a range); e.g., RandomInt(1, 10)</td>
<td>A random integer from the range low to high (both inclusive)</td>
</tr>
<tr>
<td>RandomChar</td>
<td>A nonempty string; e.g., RandomChar('abcd')</td>
<td>A random character taken from the string</td>
</tr>
<tr>
<td>RandomOneOf</td>
<td>A list of options in square brackets, separated by commas; e.g.,</td>
<td>A random value taken from the list of options</td>
</tr>
<tr>
<td></td>
<td>RandomOneOf(['yes', 'no'])</td>
<td></td>
</tr>
</tbody>
</table>

any page can utilize the functions by first loading the `random.js` library:

```html
<script type="text/javascript" src="http://balance3e.com/random.js"></script>
```

for example, could revise the ESP Test page to use `RandomInt`:

```javascript
number = RandomInt(1, 4);
```
Design Example

consider the task of designing an online Magic 8-ball® (Mattell, Inc.)

- must be able to ask a yes/no type question
- receive an answer (presumably, at random)

could use:

- a text box for entering the question
- a DIV element for displaying the answer
- a clickable image for initiating the action – which involves calling a function to process the question, select an answer, and display it in the DIV
Designing Functions

functions do not add any computational power to the language
  - a function definition simply encapsulates other statements

still, the capacity to define and use functions is key to solving complex problems, as well as to developing reusable code
  - encapsulating repetitive tasks can shorten and simplify code
  - functions provide units of computational abstraction – user can ignore details
  - functions are self-contained, so can easily be reused in different applications

when is it worthwhile to define a function?
  - if a particular computation is complex—meaning that it requires extra variables and/or multiple lines to define
  - if you have to perform a particular computation repeatedly

when defining a function, you must identify
  - the inputs
  - the computation to be performed using those inputs
  - the output