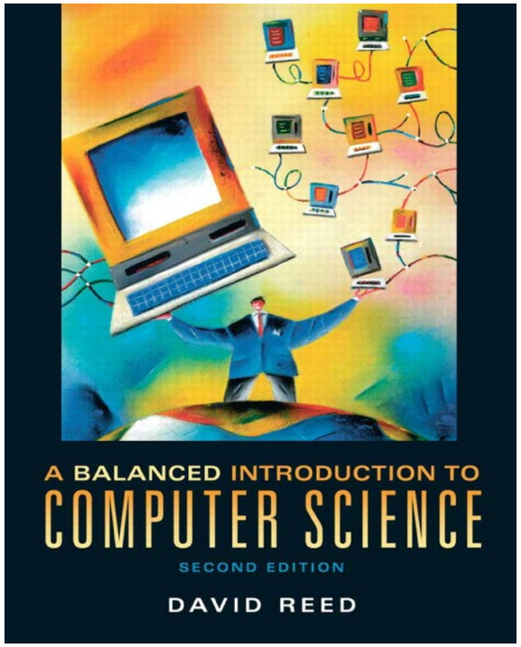
A Balanced Introduction to Computer Science, 2/E

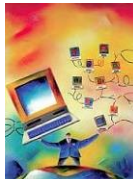
David Reed, Creighton University

©2008 Pearson Prentice Hall
ISBN 978-0-13-601722-6



Chapter 13
Conditional Repetition

Conditional Repetition



an if statement is known as a *control statement*

- it is used to control the execution of other JavaScript statements

- provides for conditional execution
- is useful for solving problems that involve choices
  - *either do this or don't, based on some condition* (if)
  - *either do this or do that, based on some condition* (if-else)

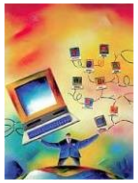
closely related to the concept of conditional execution is *conditional repetition*

- many problems involve repeating some task over and over until a specific condition is met

- e.g., rolling dice until a 7 is obtained
- e.g., repeatedly prompting the user for a valid input

- in JavaScript, *while loops* provide for conditional repetition

While Loops



a *while loop* resembles an if statement in that its behavior is dependent on a boolean condition.

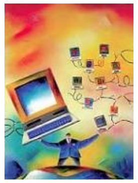
- however, the statements inside a while loop's curly braces (a.k.a. the *loop body*) are executed *repeatedly* as long as the condition remains true
- general form:

```
while (BOOLEAN_TEST) {
    STATEMENTS_EXECUTED_AS_LONG_AS_TRUE
}
```

when the browser encounters a while loop, it first evaluates the boolean test

- if the test succeeds, then the statements inside the loop are executed in order, *just like an if statement*
- once all the statements have been executed, program control returns to the beginning of the loop
- the loop test is evaluated again, and if it succeeds, the loop body statements are executed *again*
- this process repeats until the boolean test fails

While Loop Example



example: roll two dice repeatedly until doubles are obtained

```
roll1 = RandomInt(1, 6);                // SIMULATE THE DICE ROLLS,
roll2 = RandomInt(1, 6);                // STORE IN VARIABLES, AND DISPLAY
document.write("You rolled: " + roll1 + " " +roll2 + "<br />");

while (roll1 != roll2) {                // AS LONG AS YOU DON'T HAVE DOUBLES,
    roll1 = RandomInt(1, 6);            // ROLL AGAIN AND DISPLAY THE ROLLS
    roll2 = RandomInt(1, 6);
    document.write("You rolled: " + roll1 + " " +roll2 + "<br />");
}
document.write("DOUBLES!");             // DISPLAY THE FACT THAT YOU HAVE DOUBLES
```

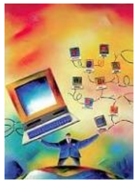
sample output:

```
You rolled: 6 4
You rolled: 4 1
You rolled: 2 4
You rolled: 6 3
You rolled: 5 2
You rolled: 1 5
You rolled: 5 2
You rolled: 6 5
You rolled: 1 2
You rolled: 5 5
DOUBLES!
```

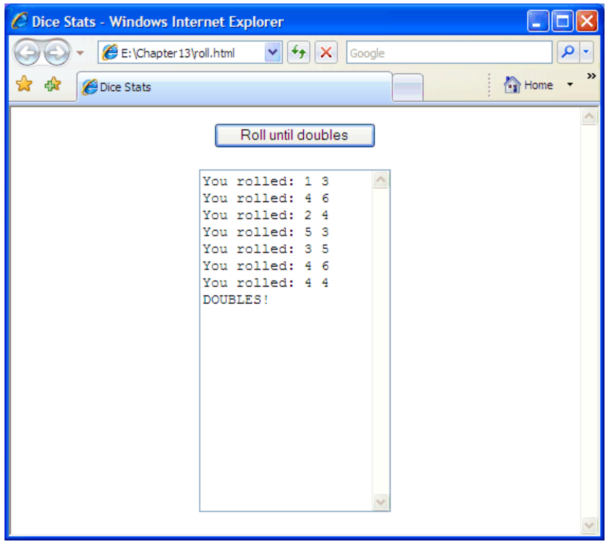
note: even though while loops and if statements look similar, they are very different control statements

- an *if statement* may execute its code 1 time or not at all
- a *while loop* may execute its code an arbitrary number of times (including not at all)

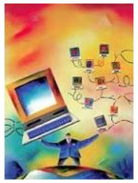
While Loop Page



```
<html>
 <!-- roll.html                                        Dave Reed -->
 <!-- This page simulates dice rolls until doubles are obtained. -->
 <!-- ============================================================ -->

 <head>
   <title> Dice Stats </title>
   <script type="text/javascript"
           src="http://dave-reed.com/book/random.js">
   </script>
   <script type="text/javascript">
     function RollUntilDoubles()
     // Assumes: OutputBox is available for output
     // Results: rolls and displays dice until doubles are obtained
     {
      var roll1, roll2;

      roll1 = RandomInt(1, 6);                    // ROLL AND DISPLAY DICE
      roll2 = RandomInt(1, 6);
      document.getElementById("OutputBox").value =
          "You rolled: " + roll1 + " " + roll2 + "\n";

      while (roll1 != roll2) {                    // WHILE NOT DOUBLES,
        roll1 = RandomInt(1, 6);                  // ROLL AGAIN AND DISPLAY
        roll2 = RandomInt(1, 6);
        document.getElementById("OutputBox").value =
            document.getElementById("OutputBox").value +
            "You rolled: " + roll1 + " " + roll2 + "\n";
      }
      document.getElementById("OutputBox").value =
           document.getElementById("OutputBox").value + "DOUBLES!";
    }
   </script>
 </head>

 <body>
   <div style="text-align:center">
   <input type="button" value="Roll until doubles"
             onclick="RollUntilDoubles();" />
      <br /><br />
      <textarea id="OutputBox" rows="20" cols="20"></textarea>
   </div>
 </body>
</html>
```



```
You rolled: 1 3
You rolled: 4 6
You rolled: 2 4
You rolled: 5 3
You rolled: 3 5
You rolled: 4 6
You rolled: 4 4
DOUBLES!
```



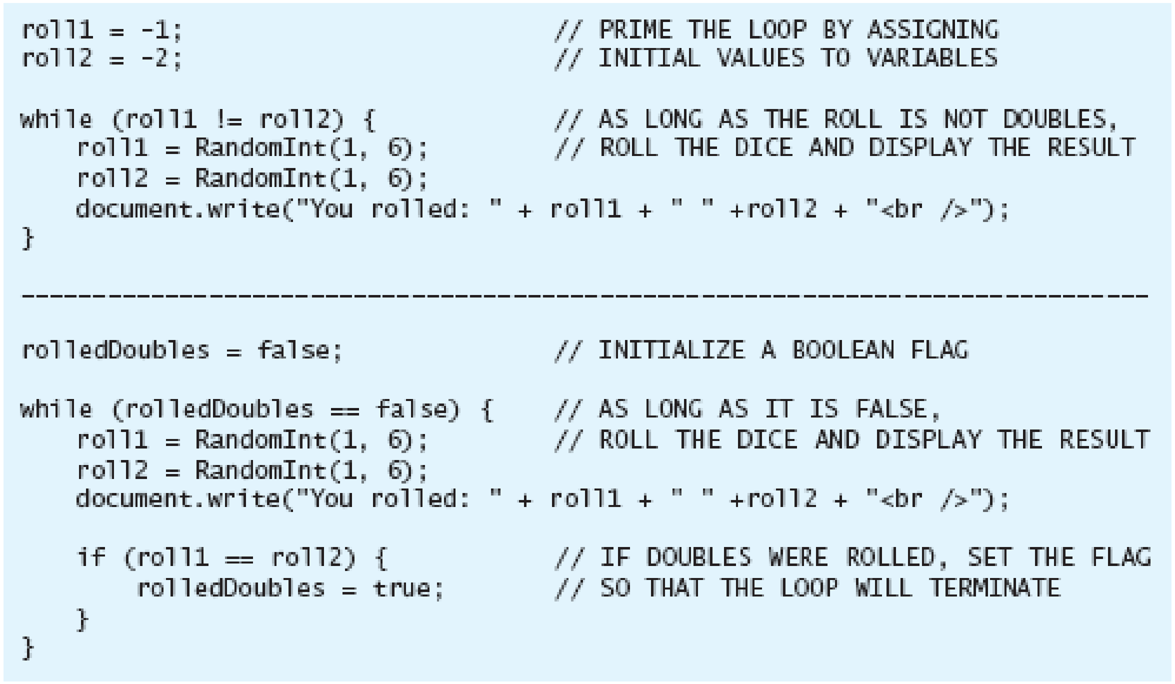
Avoiding redundancy

note the redundancy in the code

- must perform the initial dice roll before the loop begins
- then, have to repeatedly re-roll inside the loop

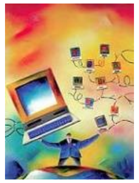
can avoid this by either:

- "priming the loop" with default values that allow the loop to execute
- defining a Boolean "flag" to determine when the loop should continue



```
roll1 = -1;                        // PRIME THE LOOP BY ASSIGNING
roll2 = -2;                        // INITIAL VALUES TO VARIABLES

while (roll1 != roll2) {           // AS LONG AS THE ROLL IS NOT DOUBLES,
    roll1 = RandomInt(1, 6);       // ROLL THE DICE AND DISPLAY THE RESULT
    roll2 = RandomInt(1, 6);
    document.write("You rolled: " + roll1 + " " +roll2 + "<br />");
}

------------------------------------------------------------------------------

rolledDoubles = false;             // INITIALIZE A BOOLEAN FLAG

while (rolledDoubles == false) {   // AS LONG AS IT IS FALSE,
    roll1 = RandomInt(1, 6);       // ROLL THE DICE AND DISPLAY THE RESULT
    roll2 = RandomInt(1, 6);
    document.write("You rolled: " + roll1 + " " +roll2 + "<br />");

    if (roll1 == roll2) {          // IF DOUBLES WERE ROLLED, SET THE FLAG
        rolledDoubles = true;      // SO THAT THE LOOP WILL TERMINATE
    }
}
```

Loop Tests



note: the loop test defines the condition under which the loop continues

- this is often backwards from the way we think about loops

- e.g., read input until you get a positive number (i.e., until input > 0)

```
while (input <= 0) { . . . }
```

- e.g., keep rolling dice until you get doubles (i.e., until roll1 == roll2)

```
while (roll1 != roll2) { . . . }
```

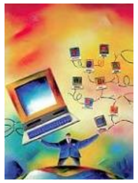
- e.g., keep rolling dice until you get double fours (i.e., until roll1 == 4 && roll2 = 4)

```
while (roll1 != 4 || roll2 != 4) { . . . }
```

DeMorgan's Law: `!(X && Y) == (!X || !Y)`

`!(X || Y) == (!X && !Y)`

Counter-Driven Loops



since a while loop is controlled by a condition, it is *usually* impossible to predict the number of repetitions that will occur

- e.g., how many dice rolls will it take to get doubles?

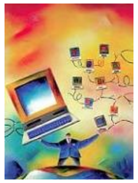
a while loop can also be used to repeat a task some fixed number of times

- implemented by using a while loop whose test is based on a counter
- general form of counter-driven while loop:

```
repCount = 0;
while (repCount < DESIRED_NUMBER_OF_REPETITIONS) {
    STATEMENTS_FOR_CARRYING_OUT_DESIRED_TASK
    repCount = repCount + 1;
}
```

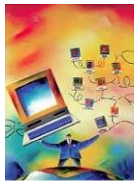
- the counter is initially set to 0 before the loop begins, and is incremented at the end of the loop body
  - the counter keeps track of how many times the statements in the loop body have executed
  - when the number of repetitions reaches the desired number, the loop test fails and the loop terminates

Counter-Driven Loops



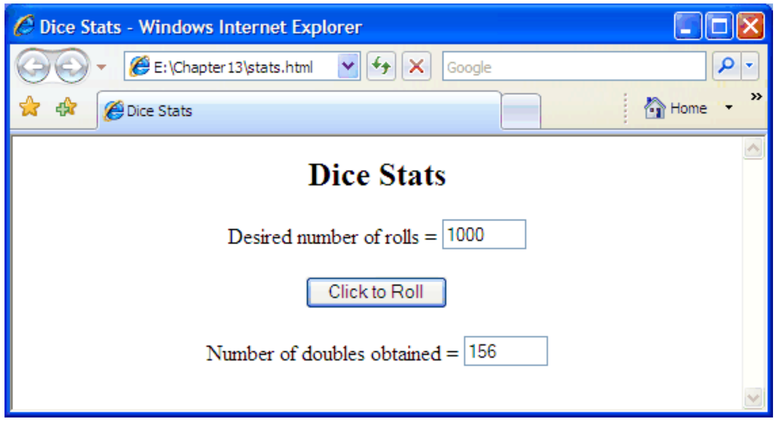
examples:

```
repCount = 0;                          // INITIALIZE THE REPETITION COUNTER
while (repCount < 100) {               // AS LONG AS < 100 REPETITIONS
    document.write("HOWDY!<br />");    //   WRITE "HOWDY!"

    repCount = repCount + 1;           // INCREASE THE REPETITION COUNTER
}

------------------------------------------------------------------------------

repCount = 0;                          // INITIALIZE THE REPETITION COUNTER
while (repCount < 1000) {              // AS LONG AS < 1000 REPETITIONS
    roll1 = RandomInt(1, 6);           // SIMULATE AND DISPLAY THE ROLLS
    roll2 = RandomInt(1, 6);
    document.write("You rolled: " + roll1 + " " +roll2 + "<br />");

    repCount = repCount + 1;           // INCREASE THE REPETITION COUNTER
}
```



Counter-Driven Loops Page

```
<html>
 <!-- stats.html                                         Dave Reed -->
 <!-- This page simulates dice rolls and maintains stats.    -->
 <!-- ==================================================== -->

 <head>
  <title> Dice Stats </title>
  <script type="text/javascript"
          src="http://dave-reed.com/book/random.js">
  </script>
  <script type="text/javascript">
      function RollRepeatedly()
      // Assumes: repsBox contains a number
      // Results: simulates that many dice rolls, displays # of doubles
      {
        var doubleCount, totalRolls, repCount, roll1, roll2;

        doubleCount = 0;                          // INITIALIZE THE COUNTER

        totalRolls = document.getElementById("repsBox").value;
        totalRolls = parseFloat(totalRolls);

        repCount = 0;
        while (repCount < totalRolls) {           // REPEATEDLY:
            roll1 = RandomInt(1, 6);              // SIMULATE THE DICE ROLLS
            roll2 = RandomInt(1, 6);

            if (roll1 == roll2) {                 // IF DOUBLES,
                doubleCount = doubleCount + 1;    // INCREMENT THE COUNTER
            }

            repCount = repCount + 1;              // INCR. REPETITION COUNTER
        }

        document.getElementById("countBox").value = doubleCount;   // DISPLAY #
      }
  </script>
 </head>

 <body>
  <div style="text-align:center">
    <h2>Dice Stats</h2>
    <p>
     Desired number of rolls =
    <input type="text" id="repsBox" size="6" value="1000" />
    </p>
    <p>
     <input type="button" value="Click to Roll" onclick="RollRepeatedly();" />
    </p>
    <p>
     Number of doubles obtained =
     <input type="text" id="countBox" size="6" value="0" />
    </p>
  </div>
 </body>
</html>
```

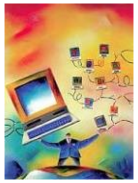


Dice Stats

Desired number of rolls = 1000

Number of doubles obtained = 156

Infinite Loops



the browser will repeatedly execute statements in the body of a while loop as long as the loop test succeeds (evaluates to true)

- it is possible that the test will always succeed and the loop will run forever

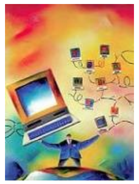
```
repCount = 0;
while (repCount < 100) {
    document.write("HOWDY!<br />");
}
```

- a loop that runs forever is known as an *infinite loop* (or a *black hole loop*)

- to guard against infinite loops, make sure that some part of the loop test changes inside the loop
  - in the above example, `repCount` is not updated in the loop so there is no chance of terminating once the loop starts

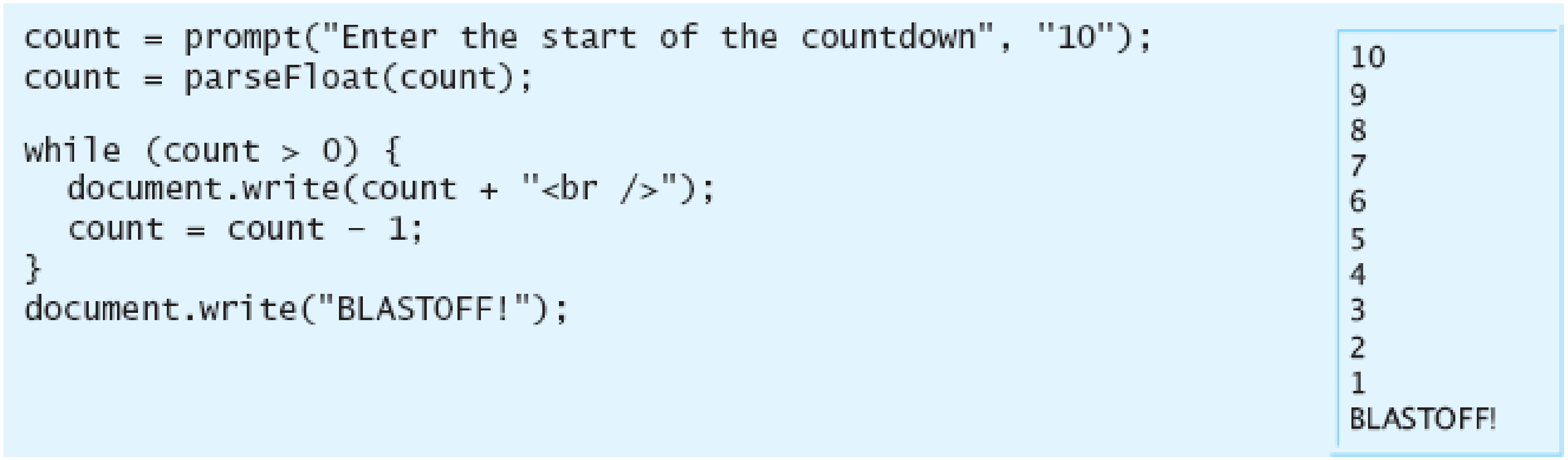
- an infinite loop may freeze up the browser
  - sometimes, clicking the Stop button will suffice to interrupt the browser
  - other times, you may need to restart the browser

Variables and Repetition



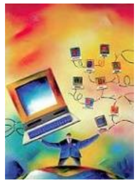
any variable can be employed to control the number of loop repetitions and the variable can be updated in various ways

example: countdown



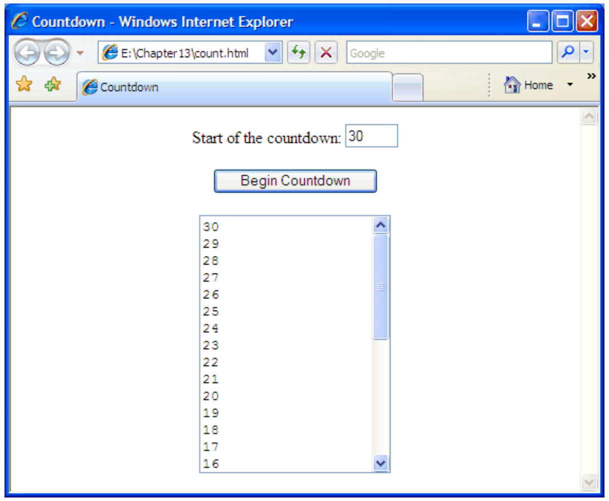
```
count = prompt("Enter the start of the countdown", "10");
count = parseFloat(count);

while (count > 0) {
  document.write(count + "<br />");
  count = count - 1;
}
document.write("BLASTOFF!");
```

```
10
9
8
7
6
5
4
3
2
1
BLASTOFF!
```



Countdown Page

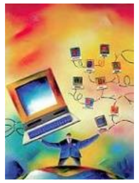
```
<html>
 <!-- count.html                                  Dave Reed -->
 <!-- This page displays a countdown in a text area.        -->
 <!-- ================================================== -->

 <head>
   <title> Countdown </title>
   <script type="text/javascript">
     function Countdown()
     // Assumes: countBox contains a number
     // Results: displays a countdown from that number in outputArea
     {
        var count;

        count = document.getElementById("countBox").value;  // GET INITIAL VALUE
        count = parseFloat(count);                           // IN NUMBER FORM

        document.getElementById("outputArea").value = "";   // CLEAR TEXT AREA

        while (count > 0) {                                  // WHILE NOT AT 0,
        document.getElementById("outputArea").value =       // DISPLAY COUNT
           document.getElementById("outputArea").value + count + "\n";
        count = count - 1;                                   // AND DECREMENT
        }

        document.getElementById("outputArea").value =
           document.getElementById("outputArea").value + "BLASTOFF!";
     }
   </script>
 </head>

 <body>
   <div style="text-align:center">
     <p>
      Start of the countdown:
      <input type="text" id="countBox" size="4" value="10" />
     </p>
     <p>
      <input type="button" value="Begin Countdown" onclick="Countdown();" />
     </p>
     <p>
      <textarea id="outputArea" rows="15" cols="20"></textarea>
     </p>
   </div>
 </body>
</html>
```



Start of the countdown: 30

```
30
29
28
27
26
25
24
23
22
21
20
19
18
17
16
```

Example: Hailstone Sequences



an interesting unsolved problem in mathematics: hailstone sequence

1. start with any positive integer
2. if the number is odd, then multiply the number by three and add one; otherwise, divide it by two
3. repeat as many times as desired

- for example: 5, 16, 8, 4, 2, 1, 4, 2, 1, 4, 2, 1, …

it is conjectured that, no matter what positive integer you start with, you will always end up in the 4-2-1 loop

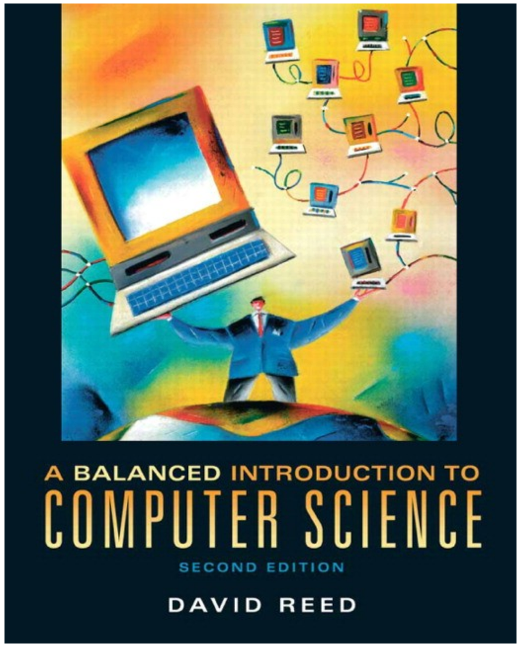
- this has been verified for all starting number up to 1,200,000,000,000
- but, it still has not been proven to hold for ALL starting numbers

- we can define a JavaScript function for experimenting with this problem
  - the `hailstone` function will generate the sequence given a starting number

A Balanced Introduction to Computer Science, 2/E

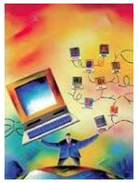
David Reed, Creighton University

©2008 Pearson Prentice Hall
ISBN 978-0-13-601722-6



Chapter 15
JavaScript Strings

Strings as Objects



so far, your interactive Web pages have manipulated strings in simple ways

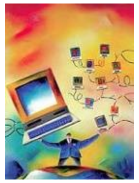
- use `prompt` or a text box/area to input a word or phrase
- store that text in a (string) variable
- incorporate the text in a message, possibly using + to concatenate

strings are different from numbers and Booleans in that they are *objects*

- a *software object* is a unit of code that encapsulates both data and operations that can be performed on that data
- a string is a software object that models words and phrases

data:	a sequence of characters, enclosed in quotes
operations include:	make upper case, make lower case, determine the number of characters, access a particular character, search for a particular character, …

Object-Oriented Programming



objects are fundamental in the dominant approach to developing software systems: *object-oriented programming (OOP)*

- OOP encourages programmers to design programs around software objects
  - the programmer identifies the real-world objects involved in a system (e.g., for a banking program: bank account, customer, teller, …)
  - then designs and builds software objects to model these real-world objects

- OOP is effective for managing large systems, since individual objects can be assigned to different teams and developed independently
- OOP also supports code reuse, since the same or similar objects can be combined in different ways to solve different kinds of problems

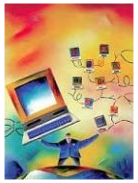
example: a doorbell button

- has physical components/properties: color, shape, label, …
- has functionality: when you press the button, the bell rings

an HTML button is a software object that models a real-world button

- has physical components/properties: color, shape, label, …
- has functionality: when you click on the button, JavaScript code is executed

Properties and Methods



using object-oriented terminology,

- the characteristics of an object are called *properties*
  - e.g., a string object has a `length` property that identifies the number of characters in the string
- the operations that can be performed on the string are called *methods*
  - e.g., the `toLowerCase` method makes a copy of the string with all upper-case letters converted to lower-case

properties and methods are not new concepts

- a property is a *special kind* of a variable (it stores a value)
- a method is a *special kind* of function (it performs some action)

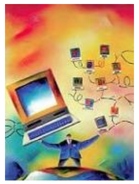
what is special is that they are associated with (or "belong to") an object

- e.g., each string object will have its own variable to stores it length

to access an object property, specify: object name, a period, property name

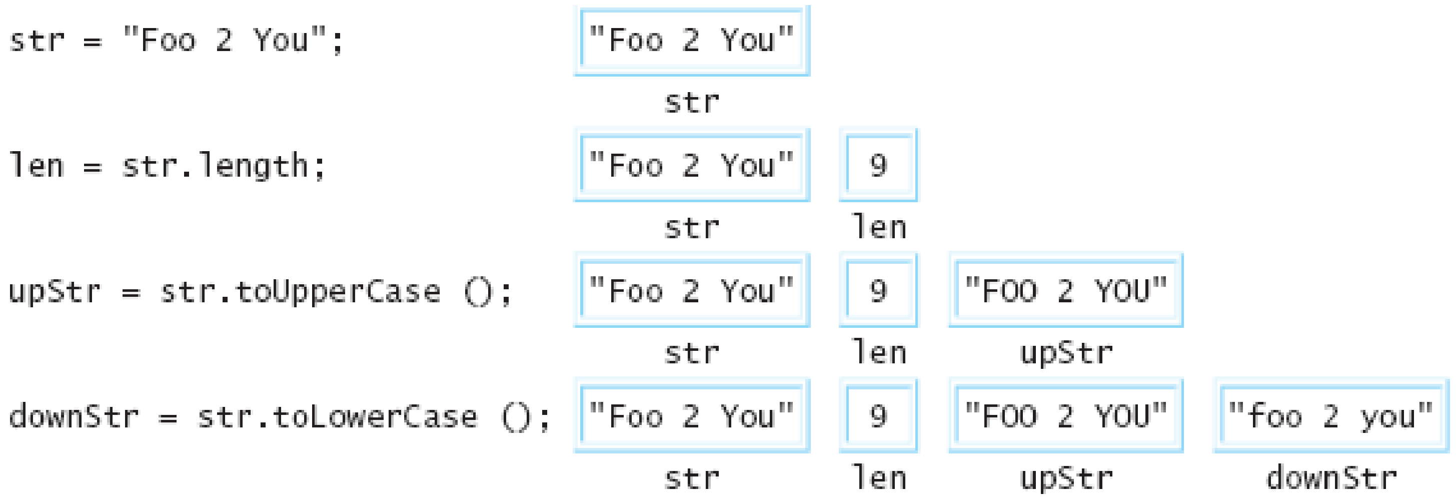
```
str1 = "foo";                     str2 = "Hi there";

len1 = str1.length;               len2 = str2.length;
```

Properties and Methods



similarly, to call a method: object name, period, method call

- e.g., `str.toLowerCase()` calls the `toLowerCase` method on `str` (which *returns* a lowercase copy of the string)
- e.g., `str.toUpperCase()` calls the `toUpperCase` method on `str` (which *returns* an uppercase copy of the string)

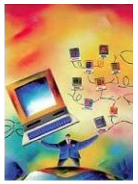


```
str = "Foo 2 You";
len = str.length;
upStr = str.toUpperCase ();
downStr = str.toLowerCase ();
```

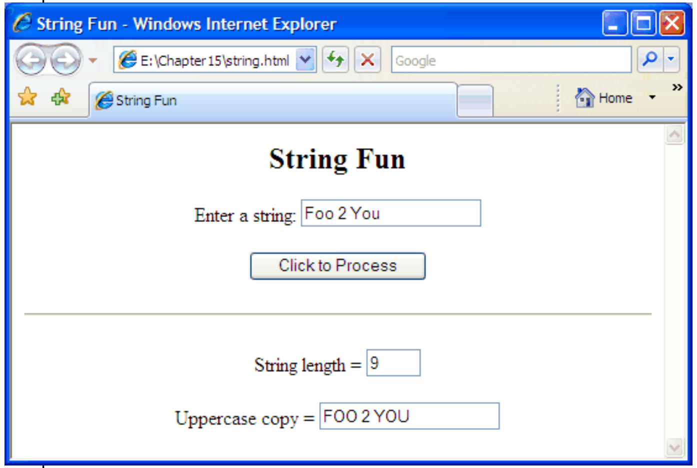
note: the `toLowerCase` and `toUpperCase` methods do not change the string object they are called on (only an assignment can do that!)

- instead, they return modified copies of the string

String Manipulation Page



```
<html>
  <!-- string.html                                          Dave Reed -->
  <!-- This page demonstrates several string properties and operations -->
  <!-- ================================================================ -->

  <head>
    <title> String Fun </title>
    <script type="text/javascript">
      function Process()
      // Assumes: strBox contains a string
      // Results: displays the outcome of string operations in other boxes
      {
        var str;

        str = document.getElementById("strBox").value;

        document.getElementById("lengthBox").value = str.length;
        document.getElementById("upperBox").value = str.toUpperCase();
      }
    </script>
  </head>

  <body>
    <div style="text-align:center">
      <h2>String Fun</h2>
      <p>
       Enter a string: <input type="text" id="strBox" size="20" value="" />
      </p>
      <p>
       <input type="button" value="Click to Process" onclick="Process();" />
      </p>
      <hr />
      <p>
       String length = <input type="text" id="lengthBox" size="3" value="0" />
      </p>
      <p>
       Uppercase copy = <input type="text" id="upperBox" size="20" value="0" />
      </p>
    </div>
  </body>
</html>
```



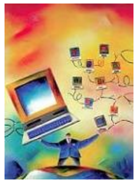
String Fun

Enter a string: Foo 2 You

String length = 9

Uppercase copy = FOO 2 YOU

Common String Methods



useful methods exist that allow programmers to access and manipulate individual components of a string

- components are identifiable via *indices*, or numbers that correspond to the order in which individual characters occur in a string
- indices are assigned in ascending order from left to right, so that the first character in the string is at index 0

the `charAt` method provides access to a single character within the string

- it takes an index as an input and returns the character at that particular index

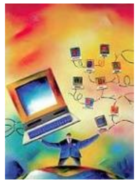
```
word = "foo";
ch = word.charAt(0);                // ASSIGNS ch = "f"
```

the `substring` method provides access to an entire sequence of characters within the string

- it takes two numbers as inputs, representing the starting (inclusive) and ending (exclusive) indices of the substring, and returns the substring

```
word = "foo";
sub = word.substring(1, 3);         // ASSIGNS sub = "oo"
```

String Access/Concatenation

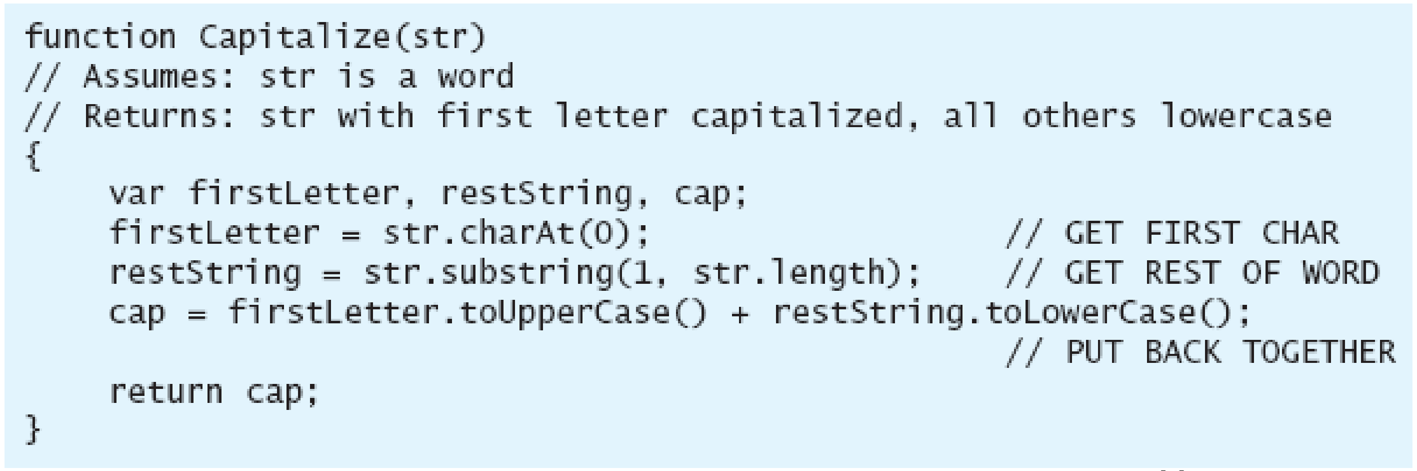


recall: the concatenation operator (+) can join strings together

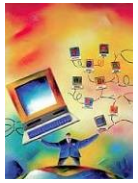
assuming the variable `word` stores a string value, what affect would the following assignment have?

```
word = word.charAt(0) + word.substring(1, word.length);
```

the following function takes a string as input and uses string method calls to create (and return) a capitalized version of that string



```
function Capitalize(str)
// Assumes: str is a word
// Returns: str with first letter capitalized, all others lowercase
{
    var firstLetter, restString, cap;
    firstLetter = str.charAt(0);                     // GET FIRST CHAR
    restString = str.substring(1, str.length);       // GET REST OF WORD
    cap = firstLetter.toUpperCase() + restString.toLowerCase();
                                                     // PUT BACK TOGETHER
    return cap;
}
```



Searching Strings

the `search` method traverses a string in order to locate a given character or substring

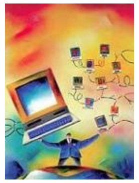
- it takes a character or string as input and returns the index at which the character or string first occurs (or -1 if not found)

```
str = "banana";
num1 = str.search("n");   // ASSIGNS num1 = 2 since the character
                          //   "n" first occurs at index 2
num2 = str.search("ana"); // ASSIGNS num2 = 1 since the string
                          //   "ana" first occurs at index 1
num3 = str.search("z");   // ASSIGNS num3 = -1 since the character
                          //   "z" does not occur anywhere
```

simple application: determine whether a string is a single word or a phrase

- if the string contains no spaces, the call `str.search(" ")` will return -1, indicating that the string value consists of a single word
- if `str.search(" ")` returns a nonnegative value, then the presence of spaces signifies a phrase containing multiple words

General Searches



there are times when you want to search for a type of character, rather than a specific value

example: converting a word into Pig Latin

- if a word contains no vowels or begins with a vowel, the characters "way" are appended to the end of the word

  nth → nthway   apple → appleway

- if a word begins with a consonant, its initial sequence of consonants is shifted to the end of the word followed by "ay"

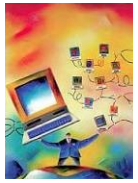
  banana → ananabay   cherry → errychay

in order to distinguish between these two cases, must `search` for the first vowel

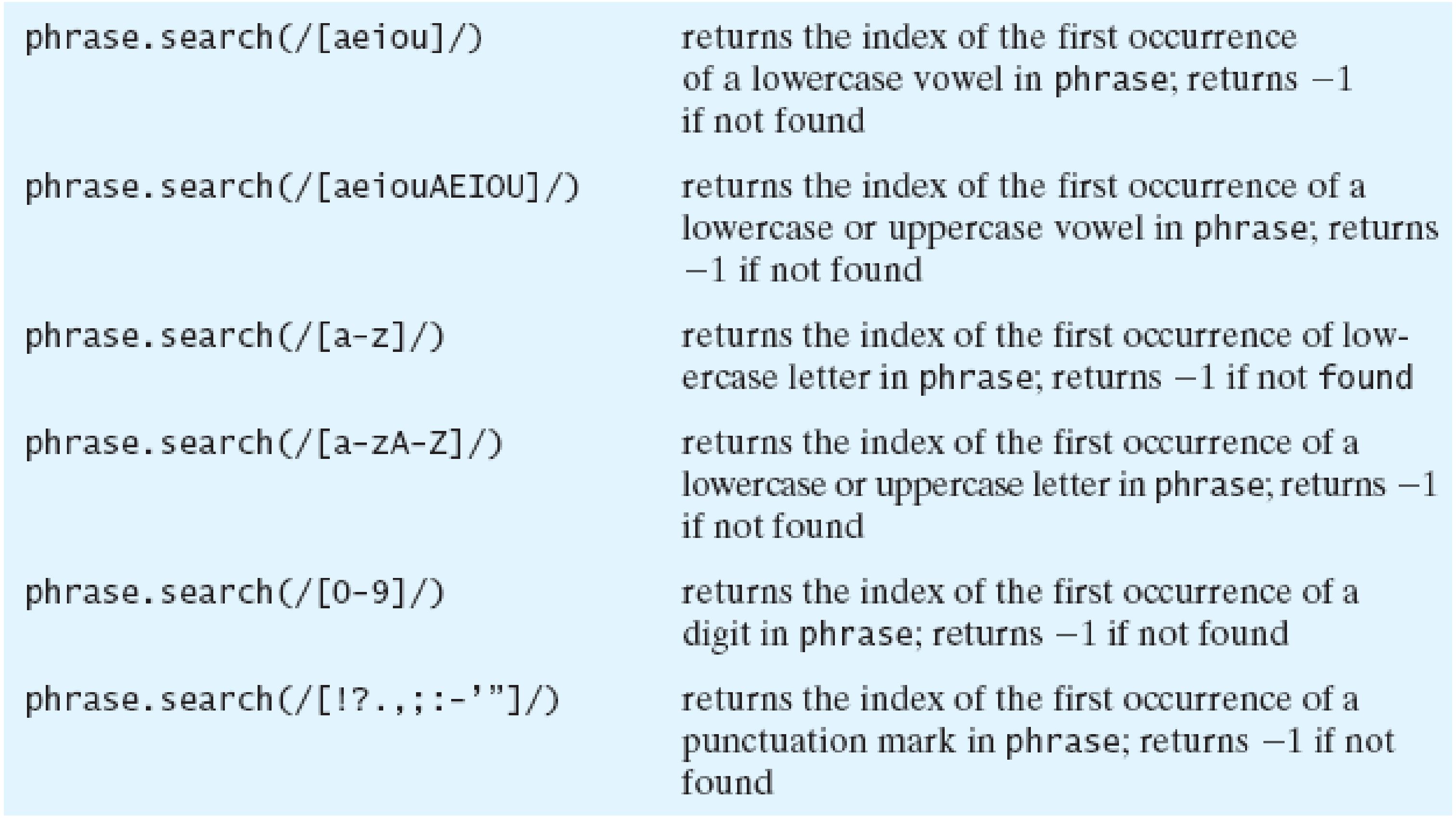
- then, use the `substring` method to break the string into parts and the + operator to put the pieces back together (with "ay")

cherry → erry + ch + ay = errychay

General Searches

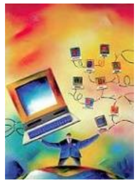


rather than having to search for vowels individually, an entire class of characters can be specified using /[. . .]/



`phrase.search(/[aeiou]/)`	returns the index of the first occurrence of a lowercase vowel in `phrase`; returns −1 if not found
`phrase.search(/[aeiouAEIOU]/)`	returns the index of the first occurrence of a lowercase or uppercase vowel in `phrase`; returns −1 if not found
`phrase.search(/[a-z]/)`	returns the index of the first occurrence of lowercase letter in `phrase`; returns −1 if not found
`phrase.search(/[a-zA-Z]/)`	returns the index of the first occurrence of a lowercase or uppercase letter in `phrase`; returns −1 if not found
`phrase.search(/[0-9]/)`	returns the index of the first occurrence of a digit in `phrase`; returns −1 if not found
`phrase.search(/[!?.,;:-’”]/)`	returns the index of the first occurrence of a punctuation mark in `phrase`; returns −1 if not found

Strings and Repetition

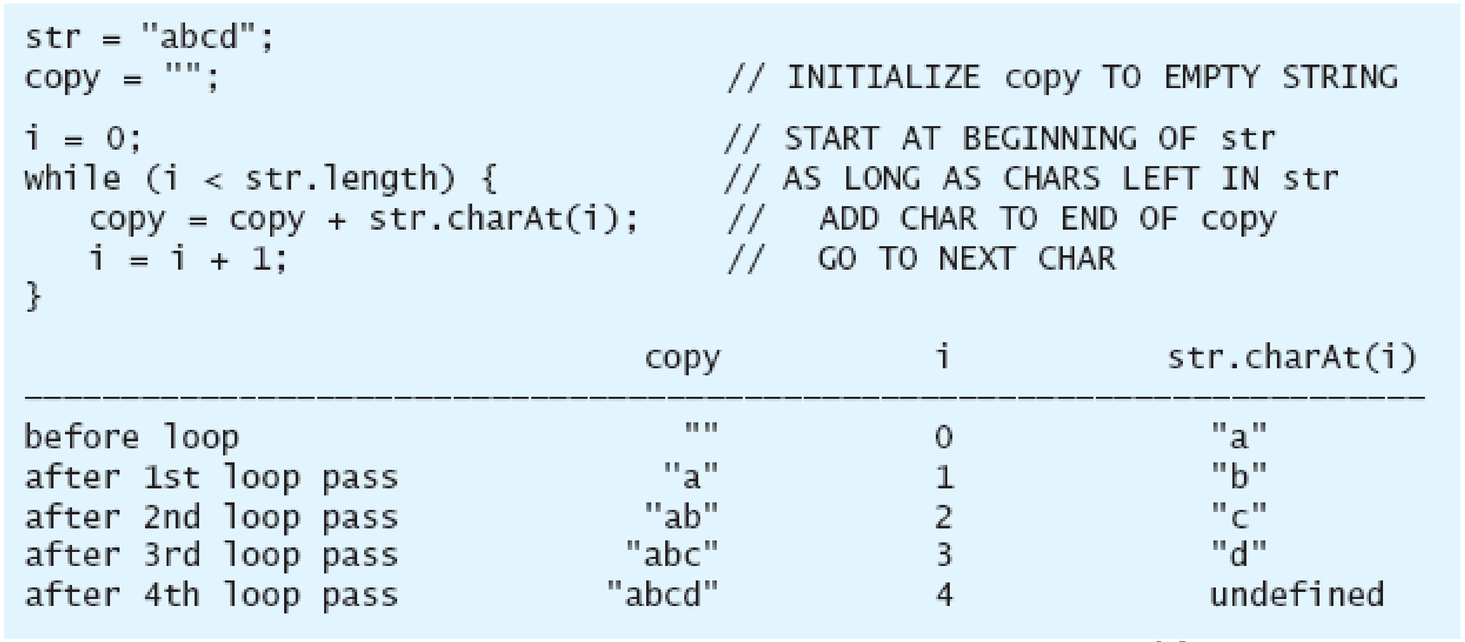


some tasks involve repeatedly performing the same operations

- to accomplish such tasks, we can combine while loops with string methods such as `charAt` and `search`

example: a while loop used to access and process each character in a string

- the characters that comprise the string are concatenated one-by-one onto another string, resulting in an exact copy



```
str = "abcd";
copy = "";                          // INITIALIZE copy TO EMPTY STRING

i = 0;                              // START AT BEGINNING OF str
while (i < str.length) {            // AS LONG AS CHARS LEFT IN str
    copy = copy + str.charAt(i);    //   ADD CHAR TO END OF copy
    i = i + 1;                      //   GO TO NEXT CHAR
}

                             copy        i        str.charAt(i)
-----------------------------------------------------------------
before loop                    ""        0          "a"
after 1st loop pass           "a"        1          "b"
after 2nd loop pass          "ab"        2          "c"
after 3rd loop pass         "abc"        3          "d"
after 4th loop pass        "abcd"        4          undefined
```