Control Statements

Jerry Cain
CS 106AJ
October 6, 2017

Once upon a time . . .

Holism vs. Reductionism

In his Pulitzer-prizewinning book, computer scientist Douglas Hofstadter identifies two concepts—holism and reductionism—that turn out to be important as you begin to learn about programming. Hofstadter explains these concepts using a dialogue in the style of Lewis Carroll:

Achilles: I will be glad to indulge both of you, if you will first oblige me, by telling me the meaning of these strange expressions, “holism” and “reductionism.”

Crab: Holism is the most natural thing in the world to grasp. It’s simply the belief that “the whole is greater than the sum of its parts.” No one in his right mind could reject holism.

Anteater: Reductionism is the most natural thing in the world to grasp. It’s simply the belief that “a whole can be understood completely if you understand its parts, and the nature of their ‘sum’.” No one in her left brain could reject reductionism.

Statement Types in JavaScript

- Statements in JavaScript fall into three basic types:
  - Simple statements
  - Compound statements
  - Control statements

- Simple statements are typically assignments, function calls, or applications of the ++ or -- operators. Simple statements are always terminated with a semicolon.

- Compound statements (also called blocks) are sequences of statements enclosed in curly braces.

- Control statements fall into two categories:
  - Conditional statements that specify some kind of test
  - Iterative statements that specify repetition

Control Statements

Boolean Expressions

- JavaScript defines two types of operators that work with Boolean data: relational operators and logical operators.
- There are six relational operators that compare values of other types and produce a true/false result:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equals</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Not equals</td>
</tr>
</tbody>
</table>

For example, the expression n <= 10 has the value true if n is less than or equal to 10 and the value false otherwise.

- There are also three logical operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>Logical NOT</td>
</tr>
</tbody>
</table>

p && q means both p and q
p || q means either p or q (or both)
![p means the opposite of p]
Notes on the Boolean Operators

• Remember that JavaScript uses = for assignment. To test whether two values are equal, you must use the \( \texttt{===} \) operator.
• It is not legal in JavaScript to use more than one relational operator in a single comparison. To express the idea embodied in the mathematical expression
  \[
  0 \leq x \leq 9
  \]
you need to make both comparisons explicit, as in
  \[
  0 \leq x \land x \leq 9
  \]
• The \( \lor \) operator means \textit{either or both}, which is not always clear in the English interpretation of or.
• Be careful when you combine the \( \lor \) operator with \( \land \) and \( \lor \) because the interpretation often differs from informal English.

The if Statement

• The simplest of the control statements is the \texttt{if} statement, which occurs in two forms. You use the first when you need to perform an operation only if a particular condition is true:

\[
\texttt{if (condition) } \{ \text{ statements to be executed if the condition is true } \}
\]

• You use the second form whenever you need to choose between two alternative paths, depending on whether the condition is true or false:

\[
\texttt{if (condition) } \{ \text{ statements to be executed if the condition is true } \} \quad \text{ else } \{ \text{ statements to be executed if the condition is false } \}
\]

The switch Statement

\[
\texttt{switch } \{ \text{ expression } \} \{ \text{ case } v_1: \text{ statements to be executed if expression is equal to } v_1; \text{ break; case } v_2: \text{ statements to be executed if expression is equal to } v_2; \text{ break; more cases clauses if needed... default: \text{ statements to be executed if no values match break; } } \}
\]

Short-Circuit Evaluation

• JavaScript evaluates the \( \land \) and \( \lor \) operators using a strategy called \textit{short-circuit mode} in which it evaluates the right operand only if it needs to do so.
• For example, if \( n \) is 0, the right operand of \( \land \) in
  \[
  n \land 0
  \]
is not evaluated at all because \( n \land 0 \) is \texttt{false}. Because the expression
  \[
  \texttt{false } \land \text{ anything }
  \]
is always \texttt{false}, the rest of the expression no longer matters.
• One of the advantages of short-circuit evaluation is that you can use \( \land \) and \( \lor \) to prevent errors. If \( n \) were 0 in the earlier example, evaluating \( x \times n \) would result in a division by zero.

Functions Involving Control Statements

• The body of a function can contain statements of any type, including control statements. As an example, the following function uses an \texttt{if} statement to find the larger of two values:

\[
\texttt{function max(x, y) } \{ \text{ if } (x > y) \{ \text{ return } x; \} \text{ else } \{ \text{ return } y; \} \}
\]

• As this example makes clear, \texttt{return} statements can be used at any point in the function and may appear more than once.

Example of the switch Statement

The \texttt{switch} statement is useful when a function must choose among several cases, as in the following example:

\[
\texttt{function monthName(month) } \{ \text{ switch (month) } \{ \text{ case } 1: \text{ return "January"; case } 2: \text{ return "February"; case } 3: \text{ return "March"; case } 4: \text{ return "April"; case } 5: \text{ return "May"; case } 6: \text{ return "June"; case } 7: \text{ return "July"; case } 8: \text{ return "August"; case } 9: \text{ return "September"; case } 10: \text{ return "October"; case } 11: \text{ return "November"; case } 12: \text{ return "December"; default: \text{ return undefined; } } \}
\]
The **while** Statement

- The **while** statement is the simplest of JavaScript's iterative control statements and has the following form:

  ```javascript
  while (condition) {
      statements to be repeated
  }
  ```

- When JavaScript encounters a **while** statement, it begins by evaluating the condition in parentheses.
- If the value of condition is **true**, JavaScript executes the statements in the body of the loop.
- At the end of each cycle, JavaScript reevaluates condition to see whether its value has changed. If condition evaluates to **false**, JavaScript exits the loop and continues with the statement following the end of the **while** body.

The **for** Statement

- The **for** statement in JavaScript is a powerful tool for specifying the structure of a loop independently from the operations the loop performs. The syntax looks like this:

  ```javascript
  for ( init ; test ; step ) {
      statements to be repeated
  }
  ```

- JavaScript evaluates a **for** statement as follows:
  1. Evaluate init, which typically declares a control variable.
  2. Evaluate test and exit from the loop if the value is **false**.
  3. Execute the statements in the body of the loop.
  4. Evaluate step, which usually updates the control variable.
  5. Return to step 2 to begin the next loop cycle.

The **factorial** Function

- The **factorial** of a number n (which is usually written as n! in mathematics) is defined to be the product of the integers from 1 up to n. Thus, 5! is equal to 120, which is 1*2*3*4*5.

- The following function definition uses a **for** loop to compute the factorial function:

  ```javascript
  function fact(n) {
      let result = 1;
      for (let i = 1; i <= n; i++) {
          result = result * i;
      }
      return result;
  }
  ```

The **digitSum** Function

```javascript
function digitSum(n) {
    let sum = 0;
    while (n > 0) {
        sum += n % 10;
        n = Math.floor(n / 10);
    }
    return sum;
}
```

Exercise: Reading **for** Statements

Describe the effect of each of the following **for** statements:

1. ```javascript
   for (let i = 1; i <= 10; i++)
   ```
   This statement executes the loop body ten times, with the control variable i taking on each successive value between 1 and 10.

2. ```javascript
   for (let i = 0; i < n; i++)
   ```
   This statement executes the loop body n times, with i counting from 0 to n-1. This version is the standard Repeat-n-Times idiom.

3. ```javascript
   for (let n = 99; n >= 1; n -= 2)
   ```
   This statement counts backward from 99 to 1 by twos.

4. ```javascript
   for (let x = 1; x <= 1024; x *= 2)
   ```
   This statement executes the loop body with the variable x taking on successive powers of two from 1 up to 1024.

The **factorialTable** Function

```javascript
function factorialTable(min, max) {
    let result = 1;
    for (let i = 1; i <= max; i++) {
        result = result * i;
    }
    return result;
}
```
Comparing `for` and `while`

- The `for` statement
  ```javascript
  for (init; test; step) {
    statements to be repeated
  }
  ```

  is functionally equivalent to the following code using `while`:

  ```javascript
  init;
  while (test) {
    statements to be repeated
    step;
  }
  ```

- The advantage of the `for` statement is that everything you need to know to understand how many times the loop will run is explicitly included in the header line.

The Checkerboard Program

```javascript
const GWINDOW_WIDTH = 500; /* Width of the graphics window */
const GWINDOW_HEIGHT = 300; /* Height of the graphics window */
const N_COLUMNS = 8; /* Number of columns */
const N_ROWS = 8; /* Number of rows */
const SQUARE_SIZE = 35; /* Size of a square in pixels */

function Checkerboard() {
  let gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
  let x0 = (gw.getWidth() - N_COLUMNS * SQUARE_SIZE) / 2;
  let y0 = (gw.getHeight() - N_ROWS * SQUARE_SIZE) / 2;
  for (let i = 0; i < N_ROWS; i++) {
    for (let j = 0; j < N_COLUMNS; j++) {
      let x = x0 + j * SQUARE_SIZE;
      let y = y0 + i * SQUARE_SIZE;
      let sq = GRect(x, y, SQUARE_SIZE, SQUARE_SIZE);
      let filled = ((i + j) % 2 !== 0);
      sq.setFilled(filled);
      gw.add(sq);
    }
  }
}
```