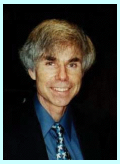
Control Statements

Jerry Cain
CS 106AJ
October 5, 2018
slides courtesy of Eric Roberts

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.

Control Statements

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

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:

`===`	Equals	`!==`	Not equals
`<`	Less than	`<=`	Less than or equal to
`>`	Greater than	`>=`	Greater than or equal to

  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:

`&&`	Logical AND	`p && q` means both `p` and `q`
`||`	Logical OR	`p || q` means either `p` or `q` (or both)
`!`	Logical NOT	`!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 `===` 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

  $$0 \le x \le 9$$

  you need to make both comparisons explicit, as in

  ```
  0 <= x && x <= 9
  ```

- The `||` operator means *either or both,* which is not always clear in the English interpretation of *or*.
- Be careful when you combine the `!` operator with `&&` and `||` because the interpretation often differs from informal English.

Short-Circuit Evaluation

- JavaScript evaluates the `&&` and `||` operators using a strategy called ***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 `&&` in

  ```
  n !== 0 && x % n === 0
  ```

  is not evaluated at all because `n !== 0` is `false`. Because the expression

  ```
  false && anything
  ```

  is always `false`, the rest of the expression no longer matters.
- One of the advantages of short-circuit evaluation is that you can use `&&` and `||` to prevent errors. If `n` were 0 in the earlier example, evaluating `x % n` would result in a division by zero.

The `if` Statement

- The simplest of the control statements is the `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:

```
if (condition) {
   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:

```
if (condition) {
   statements to be executed if the condition is true
} else {
   statements to be executed if the condition is false
}
```

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 `if` statement to find the larger of two values:

```
function max(x, y) {
   if (x > y) {
      return x;
   } else {
      return y;
   }
}
```

- As this example makes clear, `return` statements can be used at any point in the function and may appear more than once.

The `switch` Statement

The `switch` statement provides a convenient syntax for choosing among a set of possible paths:

```
switch ( expression ) {
  case v1:
     statements to be executed if expression is equal to v1
     break;
  case v2:
     statements to be executed if expression is equal to v2
     break;
  . . . more case clauses if needed . . .
  default:
     statements to be executed if no values match
     break;
}
```

Example of the `switch` Statement

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

```
function monthName(month) {
   switch (month) {
     case  1: return "January";
     case  2: return "February";
     case  3: return "March";
     case  4: return "April";
     case  5: return "May";
     case  6: return "June";
     case  7: return "July";
     case  8: return "August";
     case  9: return "September";
     case 10: return "October";
     case 11: return "November";
     case 12: return "December";
     default: return undefined;
   }
}
```

The `while` Statement

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

```
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 from the loop and continues with the statement following the end of the `while` body.

The `digitSum` Function

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

n	19
0	19

Console

```
digitSum(1729) = 19
```

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:

```
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.

Exercise: Reading `for` Statements

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

1. `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. `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. `for (let n = 99; n >= 1; n -= 2)`

   This statement counts backward from 99 to 1 by twos.

4. `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 `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:

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

The `factorialTable` Function

```
function factorialTable(min, max) {
function fact(n) {
   let result = 1;
   for (let i = 1; i <= n; i++)
      result = result * i;
   }
   return result;
}
```

5040

n	7	result	5040	8	8

Console

```
-> factorialTable(0, 7);
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
->
```

Comparing `for` and `while`

- The `for` statement

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

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

```
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

```
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);
      }
   }
}
```

The End