

Chapter 7: Javascript: Control Statements

CS 80: Internet Programming

Instructor: Mark Edmonds

Background and Terminology

Algorithm

- What is an algorithm?
 - A procedure for solving a problem. Consists of:
 1. The actions to be executed
 2. The order in which the actions are to be executed
 - Notice: this definition has nothing to do with a programming language, program statements, etc.
 - * We are abstracting away code into *problem solving*

Background and Terminology

Pseudocode

- A way to express the essence of an algorithm without using a programming language
- Informally, it is a way to express what and how the algorithm does something, but doesn't specify syntax
- Why is this useful?
 - Syntax is cumbersome, many trivial details of the code do not correspond to the overarching problem the algorithm solves

Background and Terminology

Pseudocode

- Example: Leap year

```
1 // leap year pseudocode
2 if year is divisible by 400 then
3   is_leap_year
4 else if year is divisible by 100 then
5   not_leap_year
6 else if year is divisible by 4 then
```

```
7   is_leap_year
8   else
9   not_leap_year
```

- This is in between code and English!
- This can be directly converted to code, regardless of programming language used

Background and Terminology

Pseudocode

- Example: Compound conditional leap year:

```
1  // leap year pseudocode
2  if (year is divisible by 4 and not divisible by 100) then
3      is_leap_year
4  else
5      not_leap_year
```

- Notice we are just describing the logic behind the program without consideration of syntax

Control Statements

- Prior to the **if...else** statement, our programs executed sequentially
 - Programs executed from top to bottom
- **if...else** introduced *branching*, or a *conditional jump*, meaning the program would "choose a fork in the road" based on some boolean evaluation
 - Branching enables more powerful programs, since the state of particular variables can vary with each execution
 - **if...else** gave us control over what program statements to executes for cases we cared about

Control Statements

- History lesson: the **goto** statement
 - Instructed a program to jump to a particular line in the program
 - Think of a function: "goto the starting line of the called function" and terminates with a "goto to the next line of the calling function"
 - This is a largely deprecated programming practice

- Enabled a "spaghetti programming" where a program was like a whole bunch of spaghetti laid out, with each piece of spaghetti representing a **goto** statement

Control Statements

- Now, imperative languages (Javascript, C, C++, Java, Python, ...) use *structured programming* that consists of three *control structures*
 1. *sequence structure* - execute the program linearly, one right after another
 2. *selection structure* - select or ignore a program statement (**if...else**)
 3. *repetition structure* - repeat the program statement(s) a certain number of times (**while**, **do...while**, **for**, **for...in**)

Control Statements

- We can model the control structure of a program with graphical models that show the control flow of the program
 - A flowchart of the possible program executions
 - Useful for modeling sub components of a program, but infeasible to model an entire program's execution

Control Statements

Javascript Keywords

JavaScript reserved keywords				
break	case	catch	continue	default
delete	do	else	false	finally
for	function	if	in	instanceof
new	null	return	switch	this
throw	true	try	typeof	var
void	while	with		
<i>Keywords that are reserved but not used by JavaScript</i>				
class	const	enum	export	extends
implements	import	interface	let	package
private	protected	public	static	super
yield				

Fig. 7.2 | JavaScript reserved keywords.

Figure 1: Javascript keywords

Control Statements

- **if** statement
 - Already covered
 - Important new terminology in terms of control flow:
 - * **if** is a *single-selection* statement. It selects or ignores a single action (program statement(s))
 - * We can think of an **if** statement as having a single entry point and single exit point

Control Statements

- **if...else** statement
 - Already covered
 - **if...else** is a *double-selection* statement
 - * Selects among two actions

Control Statements

- Ternary Operator (Conditional Operator)

- We can shorthand the if else statement with a ternary operator of the following form:

```
1 cond ? true_action : false_action
```

- Example:

```
1 document.writeln( student_grade >= 60 ? "Passed" : "Failed" ); //  
  immediately using the result  
2 var courseResult = student_grade >= 60 ? "Passed" : "Failed"; //  
  assignment to the result
```

Control Statements

Ternary Operator (Conditional Operator)

- Differences with **if...else**
 - Ternary operator **returns** a value
 - This is how the above two examples work (collapse/evaluate the ternary to see...)
 - * E.g. if `student_grade` was 50, this the same as calling `document.writeln("Failed");` or assigning `pass_fail = "Failed";`

Dangling else's

- We are permitted to write nested **if...else** statements
- We also don't have to include the curly braces {}, which start a **block statement**
 - Block statements are groupings of program statements
 - You can think of them like compound statements, in the same sense of compound conditionals

Dangling else's

Variable Scope

- Example block statement:

```
1 // javascript blocks and scope  
2 var a1 = 3;  
3 {  
4   var a2 = 5;  
5 }
```

```
6 console.log(a1 + a2);
```

- This behavior is different from C/C++!
 - Block statements do not introduce **scope**

Dangling else's

Variable Scope

- Scope is the "lifetime of a variable"
 - When a variable/function goes out of scope, it is not valid to use that variable or function again
 - In the example above, an equivalent C/C++ code segment would fail to compile because `a2` would be out of scope
 - The scope of `a2` would be from its declaration to its closing curly brace
- Back to why this matters for `if...else...`

Dangling else's

- If we don't include the curly braces, we have an implicit block statement
 - But what problems might we encounter with nested `if...else`'s?

Dangling else's

- Consider the following possibilities (hint: the indentation does not affect the semantic meaning)

```
1 // dangling else's
2 if ( x > 5 )
3     if ( y > 5 )
4         document.writeln( "<p>x and y are > 5</p>" );
5     else
6         document.writeln( "<p>x is <= 5</p>" );
```

```
1 // dangling else's
2 if ( x > 5 )
3     if ( y > 5 )
4         document.writeln( "<p>x and y are > 5</p>" );
5 else
6     document.writeln( "<p>x is <= 5</p>" );
```

Dangling else's

- The first indentation reflects the semantics. Why?

```
1 // dangling else's
2 if ( x > 5 )
3     if ( y > 5 )
4         document.writeln( "<p>x and y are > 5</p>" );
5     else
6         document.writeln( "<p>x is <= 5</p>" );
```

```
1 // dangling else's
2 if ( x > 5 )
3     if ( y > 5 )
4         document.writeln( "<p>x and y are > 5</p>" );
5 else
6     document.writeln( "<p>x is <= 5</p>" );
```

Dangling else's

- If there is no included block statements, a single statement is grouped to the **if**
 - **if...else** is considered a single conditional statement
 - This is part of JavaScript syntax, and is very common across programming languages
- What's the solution?
 - Using block statements fixes this problem because it enforces which **if** the **else** belongs to

Dangling else's

- Fix:

```
1 // dangling else's
2 if ( x > 5 ){
3     if ( y > 5 )
4         document.writeln( "<p>x and y are > 5</p>" );
5 } else
6     document.writeln( "<p>x is <= 5</p>" );
```

- I (personally) recommend always wrapping conditionals, loops, etc. with block statements:

```
1 // dangling else's
2 if ( x > 5 ) {
3     if ( y > 5 ){
4         document.writeln( "<p>x and y are > 5</p>" );
5     }
6 } else {
7     document.writeln( "<p>x is <= 5</p>" );
8 }
```

Dangling else's

- Consider another error-prone situation:

```
1 // dangling else's
2 if ( grade >= 60 )
3     document.writeln( "<p>Passed</p>" );
4 else
5     document.writeln( "<p>Failed</p>" );
6     document.writeln( "<p>You must take this course again.</p>" );
```

- Under what circumstances will "You must take this course again" be printed to the user?

Dangling else's

```
1 // dangling else's
2 if ( grade >= 60 )
3     document.writeln( "<p>Passed</p>" );
4 else
5     document.writeln( "<p>Failed</p>" );
6     document.writeln( "<p>You must take this course again.</p>" );
```

- The Javascript interpreter does not read indentation for semantics
- The last line is not associated with the **else**
- Semantic version:

```
1 // dangling else's
2 if ( grade >= 60 )
3     document.writeln( "<p>Passed</p>" );
4 else
```



```
5 document.writeln( "<p>Failed</p>" );  
6 document.writeln( "<p>You must take this course again.</p>" );
```

Dangling else's

- Fix:

```
1 // dangling else's  
2 if ( grade >= 60 )  
3   document.writeln( "<p>Passed</p>" );  
4 else {  
5   document.writeln( "<p>Failed</p>" );  
6   document.writeln( "<p>You must take this course again.</p>" );  
7 }
```

Dangling else's

- The main point: don't trust indentation!
 - Use explicit block statements (through curly braces)
 - * You must use a block if you have more than a single statement under your conditional/loop
 - * I do this all the time, no matter. I personally believe it offers a cleaner, more consistent code style that has better defined semantics

Dangling else's

- Technical semantics (the actual logic here):
 - **if** and **else** will associate with the next statement
 - That statement can be a single program statement, or a compound statement
 - * By making the next statement a block statement, we avoid the problem completely, even if we only execute one statement in that block statement.
 - Removes ambiguity in all cases while adding minimal amount of lines to your program (not that ever print source code anyway, so the length of your program doesn't really matter)

Errors

- When debugging, we need to evaluate what is causing the problem in our program and how we can fix it
- Three main types of errors:
 - *Syntax errors* - invalid code. The compiler/interpreter cannot successfully execute the program. Will be detected by the computer. Basically means you violated the rules of the language. Example: forgetting to put a closing curly brace at the end of an **if...else**.
 - *Semantic errors* - valid code, but does not produce the results you expect. Example: using the wrong variable or operator somewhere in your code
 - *Design errors* - valid code and produces the results you expect. However, your understanding of the problem is wrong. Example: using the wrong formula for something.

Errors

- Semantic and design errors are very similar but have different implications for debugging
 - A semantic error means we understand the problem and need to adjust our code to reflect that understanding
 - A design error means we don't understand the problem and will never be able to produce a working program
- A design error is a more significant problem than semantic error!

Errors

- Ways to fix errors:
 - *Syntax error*: computer will usually give a hint as to what's causing the problem. Go inspect your code to see what might be wrong (generally easy, but can be incredibly frustrating)
 - *Semantic error*: assuming you have correct pseudocode, something in your program doesn't match the pseudocode. Compare the two and make sure the operations match up. This is when you can't believe you've wasted an hour verifying your pseudocode matches your code but you mistyped a + for a *.
 - *Design error*: Your pseudocode is wrong. Fix the pseudocode first. There's no specific advice to give since this error is always problem-specific. This is when you email your professor/supervisor/(maybe) customer for guidance.

Repetition (loops)

- Repeat an action a number of times while a condition is true

- Example shopping pseudocode:

```
1 While there are more items on my shopping list
2 Purchase next item and cross it off my list
```

- When the condition becomes false, the loop exits
- Critical part: the loop must do something that will eventually cause the condition to be false
 - Otherwise, we are stuck in an infinite loop!

Repetition (loops)

- Example for shopping:

```
1 // shopping list
2 var shopping_list = ["pants", "groceries", "car"];
3 var i = 0;
4 // purchase all items in the list
5 while(i < shopping_list.length)
6 {
7     purchase(shopping_list[i]); // purchase the item
8     i = i + 1; // move to the next item
9 }
10 function purchase(item){
11     window.alert("Purchased " + item);
12 }
```

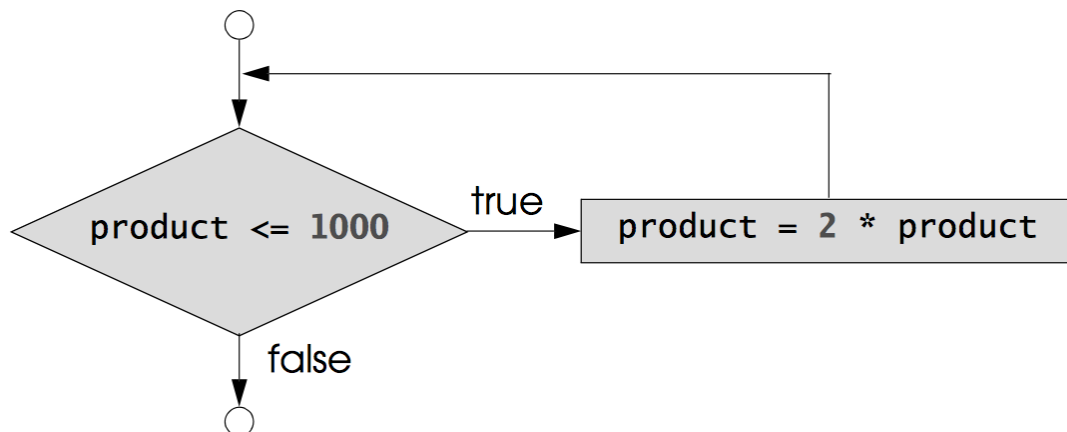
Repetition (loops)

- If you think visually, consider the following code flowchart

```
1 var product = 2;
2 while ( product <= 1000 )
3 {
4     product = 2 * product;
5 }
```

Repetition (loops)

- If you think visually, consider the following code + flowchart



- We can think about a loop as a repeated cycle in a flowchart until a condition becomes false

Exercise

Class Average

- Write pseudocode and javascript to average a class's scores on an exam.
- The program should prompt the user for the number of students in the class, then ask for each score.
- The scores should be printed nicely into a table with the average at the bottom.

Exercise

Class Average

- Pseudocode:

```
1 Set total to zero
2 Set grade counter to zero
3 Input number of students
4 Print table header
5 While grade counter is less than number of students
6   Input the next grade
7   Add the grade into the total
8   Add one to the grade counter
9   Print grade to table
10 Set the class average to the total divided by number of students
11 Print the class average to table
```

Exercise: class_average.html

```
1 <!DOCTYPE html>
2 <html>
3
4 <head>
5   <meta charset="utf-8">
6   <title>Class Average Problem</title>
7   <style>
8     table,
9     th,
10    td {
11      border: 1px solid black;
12    }
13
14    table {
15      border-collapse: collapse;
16    }
17
18    th,
19    td {
20      padding: 15px;
21      text-align: left;
22    }
23  </style>
24  <script>
25    var total; // sum of grades
26    var gradeCounter; // number of grades entered
27    var grade; // grade typed by user
28    var average; // average of all grades
29    // initialization phase
30    total = 0; // clear total
31    gradeCounter = 0;
32    var numStudents = window.prompt("Enter the number of students: ", "
    0");
33    numStudents = parseInt(numStudents);
34
35    if (numStudents == 0)
36    {
37      document.writeln("The number of students is 0");
38    }
39    else
```

```

40     {
41         //setup table
42         document.writeln("<table>");
43         document.writeln("<caption><strong>Exam scores</strong></caption>
44         ");
45         document.writeln("<thead>\n<tr>\n<th>Student Number</th>\n<th>
46         Grade</th>\n</thead>");
47         document.writeln("<tbody>");
48
49         // loop, processing phase
50         while (gradeCounter < numStudents) {
51             // prompt for input and read grade from user
52             grade = window.prompt("Enter integer grade:", "0");
53             // convert grade from a string to an integer
54             grade = parseInt(grade);
55             // add gradeValue to total
56             total = total + grade;
57             // add 1 to gradeCounter
58             gradeCounter = gradeCounter + 1;
59             // add data to table
60             document.writeln("<tr>\n<td>" + gradeCounter + "</td>\n<td>" +
61             grade + "</td>\n</tr>");
62         } // end while
63
64         // calculate the average
65         average = total / numStudents;
66         // display average of exam grades
67         document.writeln("</tbody>");
68         document.writeln("<tfoot>\n<tr>\n<th>Average</th>\n<th>" +
69         average + "</th>\n</tr>\n</tfoot>");
70         document.writeln("</table>");
71     }
72     </script>
73 </head>
74 <body>
75 </body>
76 </html>

```

Exercise

Real Estate License

- A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, 10 of the students who completed this course took the licensing exam. Naturally, the college wants to know how well its students performed.
- You've been asked to write a script to summarize the results. You've been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam and a 2 if the student failed.

Exercise

Real Estate License

- Your script should analyze the results of the exam as follows:
 1. Input each test result (i.e., a 1 or a 2). Display the message "Enter result" on the screen each time the script requests another test result.
 2. Count the number of test results of each type.
 3. Display a summary of the test results indicating the number of students who passed and the number of students who failed.
 4. If more than eight students passed the exam, print the message "Bonus to instructor!"

Exercise

Real Estate License

- Pseudocode:

```
1 Initialize passes to zero
2 Initialize failures to zero
3 Initialize student to zero
4 While student counter is less than ten
5   Input the next exam result
6   If the student passed
7     Add one to passes
8   Else
9     Add one to failures
10  Add one to student counter
11 Print the number of passes
12 Print the number of failures
```

```
13 If more than eight students passed
14 Print "Bonus to Instructor!";
```

Exercise: bonus.html

```
1 <!DOCTYPE html>
2 <!-- Fig. 7.11: analysis.html -->
3 <!-- Examination-results calculation. -->
4 <html>
5
6 <head>
7   <meta charset="utf-8">
8   <title>Analysis of Examination Results</title>
9   <script>
10    // initializing variables in declarations
11    var passes = 0; // number of passes
12    var failures = 0; // number of failures
13    var student = 0; // student counter
14    var result; // an exam result
15    // process 10 students; counter-controlled loop
16    while (student < 10) {
17      result = window.prompt("Enter result (1=pass,2=fail)", "0");
18      if (result == "1")
19      {
20        passes = passes + 1;
21      }
22      else
23      {
24        failures = failures + 1;
25      }
26      student = student + 1;
27    } // end while
28    document.writeln("<h1>Examination Results</h1>");
29    document.writeln("<p>Passed: " + passes +"; Failed: " + failures +
30      "</p>");
31
32    if (passes > 8)
33    {
34      document.writeln("<p>Bonus to instructor!</p>");
35    }
36  } // end script
37</script>
```



```

36 </head>
37
38 <body></body>
39
40 </html>

```

Assignment Operators

- Modifying a variable (changing its value) is extremely common. We have a shorthand way doing this:

```

1 c = c + 3;
2 c += 3;

```

- More generally, any statement of the form:

```
1 variable = variable operator expression;
```

- Can always be written as:

```

1 variable operator= expression; // operator could be +, -, *, /, %
2 // for example
3 c *= 4; // multiply by 4

```

Assignment Operators

Assignment operator	Initial value of variable	Sample expression	Explanation	Assigns
+=	c = 3	c += 7	c = c + 7	10 to c
-=	d = 5	d -= 4	d = d - 4	1 to d
*=	e = 4	e *= 5	e = e * 5	20 to e
/=	f = 6	f /= 3	f = f / 3	2 to f
%=	g = 12	g %= 9	g = g % 9	3 to g

Figure 2: Assignment operators

Increment and Decrement

Operator	Example	Called	Explanation
++	++a	preincrement	Increment a by 1, then use the new value of a in the expression in which a resides.
++	a++	postincrement	Use the current value of a in the expression in which a resides, then increment a by 1.
--	--b	predecrement	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	b--	postdecrement	Use the current value of b in the expression in which b resides, then decrement b by 1.

Figure 3: Increment and decrement operators

Increment and Decrement

- Key difference:
 - pre changes the value, then returns the new value
 - post returns the current value, then change the value
 - These operators break PEMDAS; they have a higher precedence than *, /, %

Increment and Decrement

- Consider the following. Carefully consider the value of `counter1` and `counter2`:

```

1  var a = 10;
2  var counter1 = 0;
3  var counter2 = 0;
4  var i = 0;
5  while(counter1++ < a)
6  {
7      //loop 1
8      console.log("Loop 1, i: ", i);
9      i++;
10 }
11 i=0;
12 while(++counter2 < a)
13 {

```

```
14 //loop 2
15 console.log("Loop 2, i: ", i);
16 i++;
17 }
18 console.log(counter1);
19 console.log(counter2);
```

Increment and Decrement

- What will be the final value of `counter1` and `counter2`?

Increment and Decrement

- What will be the final value of `counter1` and `counter2`?
 - `counter1` will be 11 (loop 1 runs 10 times, but counter1 is incremented an extra time (post-increment))
 - `counter2` will be 10 (loop 2 runs 9 times)

Additional Repetition Structures

- **for**
 - Functionally equivalent to **while**

```
1 for(initialization_statement; loop_condition; loop_end_statement)
2 {
3     // loop body
4 }
5 // in practice
6 for (var i = 0; i < 10; i++)
7 {
8     // loop body
9 }
10 // which is the same as
11 var i = 0;
12 while (i < 10){
13     i++;
14 }
```

Additional Repetition Structures

- **do...while**

- Like a while loop, but guarantees that the loop will execute at least once
- Condition is checked at the end of the loop

```

1 var i = 0;
2 do {
3     // loop body
4     i++;
5 }
6 while(i < 10);

```

Example: sortedList.html

```

1 <!doctype html>
2
3 <!-- sortedList.html -->
4 <!-- Input several names and display them in a sorted list. -->
5 <html>
6     <head>
7         <meta charset="utf-8" />
8         <title>Sorted List</title>
9     </head>
10    <body>
11        <h1>People</h1>
12        <ol id="ol"></ol>
13        <script>
14 var name; // name entry
15 var people = []; // array of people
16
17 // get names
18 while (true) {
19     name = prompt("Name: ", "Done");
20     if (name === "Done") {
21         break; // stop getting names
22     }
23     people.push(name); // add name to end of array
24 }
25
26 // sort the array

```

```
27 people.sort();
28
29 // output the list
30 document.getElementById("ol").innerHTML = "<li>" + people.join("</li><
    li>") + "</li>";
31     </script>
32 </body>
33 </html>
```