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

```
// leap year pseudocode
if year is divisible by 400 then
  is_leap_year
else if year is divisible by 100 then
  not_leap_year
else if year is divisible by 4 then
```

```
  is_leap_year
else
  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:

```
// leap year pseudocode
if (year is divisible by 4 and not divisible by 100) then
  is_leap_year
else
  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

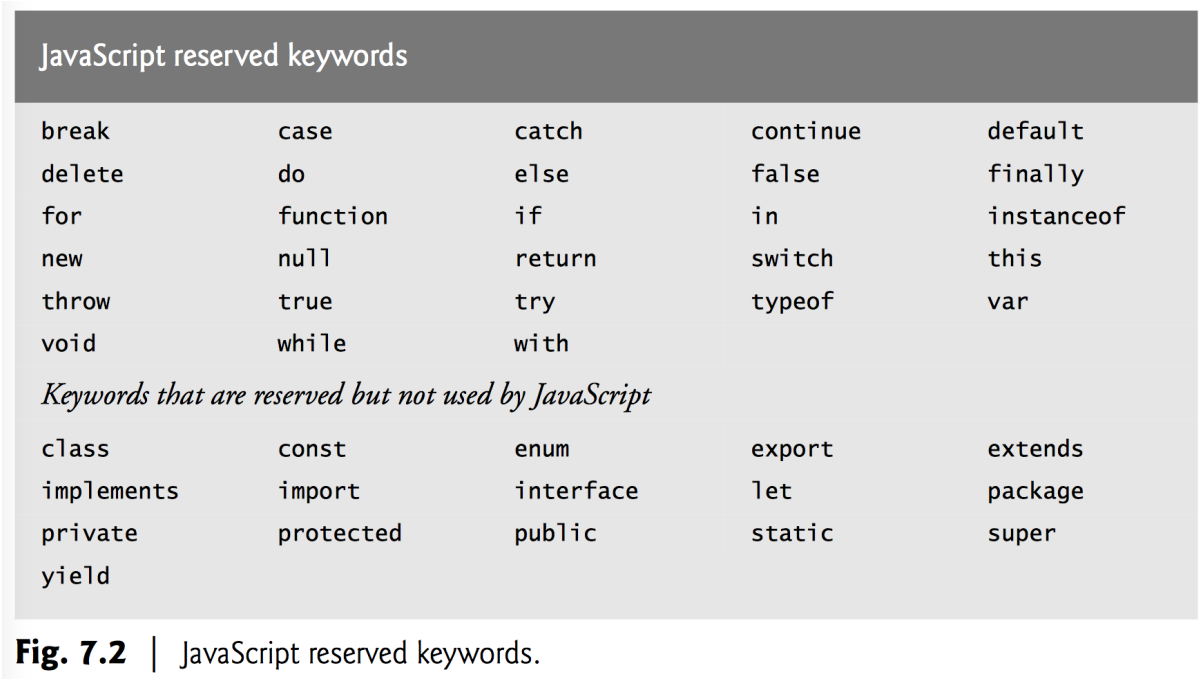
- Now, imperative languages (Javascript, C, C++, Java, Python, ...) use *structed programming* that consists of three *control structures*
  1. *sequence structure* - execute the program linearly, one right after another
  2. *selection structure* - select or ignore a program statement (`if...else`)
  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		

Keywords that are reserved but not used by JavaScript

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:

```
cond ? true_action : false_action
```

- Example:

```
document.writeln( student_grade >= 60 ? "Passed" : "Failed" ); //
    immediately using the result
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:

```
// javascript blocks and scope
var a1 = 3;
{
  var a2 = 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 segement 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 &gt; 5</p>" );
5   else
6     document.writeln( "<p>x is &lt;= 5</p>" );
```

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

Dangling `else`'s

- The first indentation reflects the semantics. Why?

```
// dangling else's
if ( x > 5 )
  if ( y > 5 )
    document.writeln( "<p>x and y are &gt; 5</p>" );
  else
    document.writeln( "<p>x is &lt;= 5</p>" );
```

```
// dangling else's
if ( x > 5 )
  if ( y > 5 )
    document.writeln( "<p>x and y are &gt; 5</p>" );
else
  document.writeln( "<p>x is &lt;= 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:

```
// dangling else's
if ( x > 5 ){
  if ( y > 5 )
    document.writeln( "<p>x and y are &gt; 5</p>" );
} else
  document.writeln( "<p>x is &lt;= 5</p>" );
```

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

```
// dangling else's
if ( x > 5 ) {
  if ( y > 5 ){
    document.writeln( "<p>x and y are &gt; 5</p>" );
  }
} else {
  document.writeln( "<p>x is &lt;= 5</p>" );
}
```

Dangling `else`'s

- Consider another error-prone situation:

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

```
// dangling else's
if ( grade >= 60 )
  document.writeln( "<p>Passed</p>" );
else
  document.writeln( "<p>Failed</p>" );
  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:

```
// dangling else's
if ( grade >= 60 )
  document.writeln( "<p>Passed</p>" );
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 succesfully 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 forumla for something.

Errors - Semantic & Design

- 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!

Fixing 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.

Reptition (loops)

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

- Example shopping pseudocode:

```
While there are more items on my shopping list
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!

Reptition (loops)

- Example for shopping:

```
// shopping list
var shopping_list = ["pants", "grocercies", "car"];
var i = 0;
// purchase all items in the list
while(i < shopping_list.length)
{
  purchase(shopping_list[i]); // purchase the item
  i = i + 1; // move to the next item
}
function purchase(item){
  window.alert("Purchased " + item);
}
```

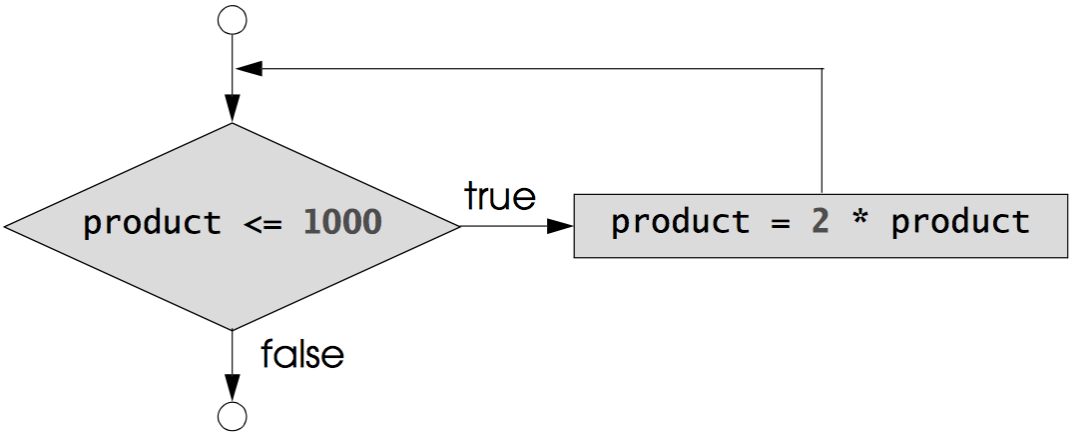
Reptition (loops)

- If you think visually, consider the following code flowchart

```
var product = 2;
while ( product <= 1000 )
{
  product = 2 * product;
}
```

Reptition (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:

```
Set total to zero
Set grade counter to zero
Input number of students
Print table header
While grade counter is less than number of students
  Input the next grade
  Add the grade into the total
  Add one to the grade counter
  Print grade to table
Set the class average to the total divided by number of students
Print the class average to table
```

Exercise: `class_average.html`

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Class Average Problem</title>
  <style>
    table,
    th,
    td {
      border: 1px solid black;
    }

    table {
      border-collapse: collapse;
    }

    th,
    td {
      padding: 15px;
      text-align: left;
    }
  </style>
  <script>
    var total; // sum of grades
    var gradeCounter; // number of grades entered
    var grade; // grade typed by user
    var average; // average of all grades
    // initialization phase
    total = 0; // clear total
    gradeCounter = 0;
    var numStudents = window.prompt("Enter the number of students: ", "
        0");
    numStudents = parseInt(numStudents);

    if (numStudents == 0)
    {
      document.writeln("The number of students is 0");
    }
    else
```

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

```
Initialize passes to zero
Initialize failures to zero
Initialize student to zero
While student counter is less than ten
  Input the next exam result
  If the student passed
    Add one to passes
  Else
    Add one to failures
  Add one to student counter
Print the number of passes
Print the number of failures
```

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

Exercise: `bonus.html`

```
<!DOCTYPE html>
<!-- Fig. 7.11: analysis.html -->
<!-- Examination-results calculation. -->
<html>

<head>
  <meta charset="utf-8">
  <title>Analysis of Examination Results</title>
  <script>
    // initializing variables in declarations
    var passes = 0; // number of passes
    var failures = 0; // number of failures
    var student = 0; // student counter
    var result; // an exam result
    // process 10 students; counter-controlled loop
    while (student < 10) {
      result = window.prompt("Enter result (1=pass,2=fail)", "0");
      if (result == "1")
      {
        passes = passes + 1;
      }
      else
      {
        failures = failures + 1;
      }
      student = student + 1;
    } // end while
    document.writeln("<h1>Examination Results</h1>");
    document.writeln("<p>Passed: " + passes +"; Failed: " + failures +
       "</p>");

    if (passes > 8)
    {
      document.writeln("<p>Bonus to instructor!</p>");
    }
  </script>
```

```
</head>

<body></body>

</html>
```

Assignment Operators

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

```
c = c + 3;
c += 3;
```

- More generally, any statement of the form:

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

- Can always be written as:

```
variable operator= expression; // operator could be +, -, *, /, %
// for example
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`:

```
var a = 10;
var counter1 = 0;
var counter2 = 0;
var i = 0;
while(counter1++ < a)
{
  //loop 1
  console.log("Loop 1, i: ", i);
  i++;
}
i=0;
while(++counter2 < a)
{
```

```
    //loop 2
    console.log("Loop 2, i: ", i);
    i++;
}
console.log(counter1);
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 (postincrement))
  - `counter2` will be 10 (loop 2 runs 9 times)

Additional Repetition Structures

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

```
for(initialization_statement; loop_condition; loop_end_statement)
{
  // loop body
}
// in practice
for (var i = 0; i < 10; i++)
{
  // loop body
}
// which is the same as
var i = 0;
while (i < 10){
  i++;
}
```

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

```
var i = 0;
do {
  // loop body
  i++;
}
while(i < 10);
```

Example: `sortedList.html`

```
<!doctype html>

<!-- sortedList.html -->
<!-- Input several names and display them in a sorted list. -->
<html>
  <head>
    <meta charset="utf-8" />
    <title>Sorted List</title>
  </head>
  <body>
    <h1>People</h1>
    <ol id="ol"></ol>
    <script>
var name; // name entry
var people = []; // array of people

// get names
while (true) {
    name = prompt("Name: ", "Done");
    if (name === "Done") {
        break; // stop getting names
    }
    people.push(name); // add name to end of array
}

// sort the array
```

```
people.sort();

// output the list
document.getElementById("ol").innerHTML = "<li>" + people.join("</li><
    li>") + "</li>";
     </script>
  </body>
</html>
```