From Pseudocode to “Real” Code

• Once we have expressed an algorithm in pseudocode, we need one more step to turn it into something that machines can do for us: conversion into an actual programming language, or “real” code

• For this course, that programming language is JavaScript — chosen because it is built-in to most Web browsers, which means you already have it on whatever computer you may be using

• This handout hopes to serve as a guide for converting pseudocode into JavaScript

Pseudocode vs. Programming Languages

Unlike pseudocode, programming language code is meant to be “understood” and run by the computer — this is where the rubber meets the road:

• Programming language code is much more precise (and thus less flexible and less “forgiving”) than pseudocode

• Programming languages may have their own distinct symbols and “look,” which might vary significantly from the original pseudocode

• Programming languages may have multiple variations for the same concept (e.g., repetitions, conditionals)
Naming and Comments in JavaScript

- The table below shows how our previous pseudocode notation translates into JavaScript
- They are similar, with JavaScript needing some additional symbols at times, such as semi-colons and braces

<table>
<thead>
<tr>
<th>Pseudocode</th>
<th>JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>name ← value</code></td>
<td><code>var name = value;</code></td>
</tr>
</tbody>
</table>
| `procedure name(input1, input2, ...)
  algorithm body`                | `var name = function(input1, input2, ...)
  algorithm body`             |
| `// Comment.`                    | `// One-line comment, or...
/* Comment consisting of
  multiple lines. */`         |

Repetitions and Conditionals

<table>
<thead>
<tr>
<th>Pseudocode</th>
<th>JavaScript</th>
</tr>
</thead>
</table>
| `while (condition)
  (code to repeat)`             | `while (condition) {
  code to repeat
}` |
| `list ← [first, second, ...]
for each (member in list)
  (code to repeat)`              | `var list = [first, second, ...];
for (var index = 0; index < list.length; index += 1) {
  var member = list[index];
  code to repeat
}` |
| `if (condition) then
  (code if condition is true)`  | `if (condition) {
  code if condition is true
}` |
| `if (condition) then
  (code if condition is true)
else
  (code if condition is false)` | `if (condition) {
  code if condition is true
} else {
  code if condition is false
}` |
# [Some] Built-In Operations

<table>
<thead>
<tr>
<th>Pseudocode</th>
<th>JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>← (assign an expression to a name)</td>
<td>=</td>
</tr>
<tr>
<td>+ (addition), – (subtraction)</td>
<td>+, –</td>
</tr>
<tr>
<td>× (multiplication), ÷ (division)</td>
<td>*, /</td>
</tr>
<tr>
<td>= (equal to), &lt;&gt; (not equal to)</td>
<td>===, !==</td>
</tr>
<tr>
<td>&lt;, &lt;= (less than [or equal to])</td>
<td>&lt;, &lt;=</td>
</tr>
<tr>
<td>&gt;, &gt;= (greater than [or equal to])</td>
<td>&gt;, &gt;=</td>
</tr>
<tr>
<td>integer division (no remainder)</td>
<td>parseInt(dividend / divisor)</td>
</tr>
<tr>
<td>remainder after division (modulo)</td>
<td>% (e.g., “((x % 2) === 1)” tests whether x is odd)</td>
</tr>
<tr>
<td>random number from min–max</td>
<td>Math.round((max – min) * Math.random()) + min</td>
</tr>
</tbody>
</table>

## Returning Answers and Invoking Other Algorithms

<table>
<thead>
<tr>
<th>Pseudocode</th>
<th>JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>return result</td>
<td>return result;</td>
</tr>
<tr>
<td>procedure algorithm(input)</td>
<td>var algorithm = function(input) {</td>
</tr>
<tr>
<td>code for algorithm</td>
<td>code for algorithm</td>
</tr>
<tr>
<td>...</td>
<td>};</td>
</tr>
<tr>
<td>algorithm(actualInput)</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>algorithm(actualInput);</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>procedure partialAnswer(input)</td>
<td>var partialAnswer = function(input) {</td>
</tr>
<tr>
<td>code for partialAnswer</td>
<td>code for partialAnswer</td>
</tr>
<tr>
<td>return output</td>
<td>return output;</td>
</tr>
<tr>
<td>value ← partialAnswer(someInput)</td>
<td>var value = partialAnswer(someInput);</td>
</tr>
</tbody>
</table>
Lists (a.k.a. Arrays)

<table>
<thead>
<tr>
<th>Pseudocode</th>
<th>JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>// Creating an empty list.</td>
<td>/* 2 choices: */ var emptyList = [];</td>
</tr>
<tr>
<td>emptyList ← [ ]</td>
<td>/* or: */ var emptyList = new Array();</td>
</tr>
<tr>
<td>// Accessing or assigning an item.</td>
<td>var item = list[index];</td>
</tr>
<tr>
<td>item ← list[index]</td>
<td>list[index] = value;</td>
</tr>
<tr>
<td>list[index] ← value</td>
<td></td>
</tr>
<tr>
<td>add value to list</td>
<td>list.push(value);</td>
</tr>
<tr>
<td>sort list “lexically,” ascending</td>
<td>list.sort(); // Caution: “a” comes after “Z”!</td>
</tr>
<tr>
<td>sort list numerically, ascending</td>
<td>list.sort(function(a, b) { return a - b; });</td>
</tr>
<tr>
<td>number ← smallest number in list</td>
<td>var number = Math.min.apply(Math, list);</td>
</tr>
</tbody>
</table>

In all cases, include var only for the first time that you assign an expression to a name.

Interacting with the User

<table>
<thead>
<tr>
<th>Pseudocode</th>
<th>JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>input ← information from user (prompted by a message)</td>
<td>var input = prompt(message);</td>
</tr>
<tr>
<td>display message</td>
<td>alert(message);</td>
</tr>
</tbody>
</table>

The examples below work only for the course’s JavaScript Scratch Page:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>retrieve text entered into the “Input 1” field on the JavaScript scratch page</td>
<td>var form = document.getElementById(&quot;scratch&quot;);</td>
</tr>
<tr>
<td></td>
<td>var input1Field = form.input1;</td>
</tr>
<tr>
<td></td>
<td>var input1Text = input1Field.value;</td>
</tr>
<tr>
<td>display message at the bottom of the JavaScript scratch page</td>
<td>var displayBox = document.getElementById(&quot;display&quot;);</td>
</tr>
<tr>
<td></td>
<td>displayBox.innerHTML = message;</td>
</tr>
</tbody>
</table>
Example Conversions from Pseudocode to JavaScript

- There’s much more to JavaScript (especially with regard to what’s “built-in”) than shown here, but the preceding tables should be enough to translate the pseudocode that you’ve seen so far into real programs that you can run within a browser.

- The overall approach would be:
  - Write out your pseudocode, and test it by hand to make sure that it does produce the expected results.
  - Refer to the preceding tables to convert each pseudocode segment into its JavaScript equivalent.

```
procedure countCoins(amount, denomination)
    currentAmount ← amount
    coinCount ← 0
    while (currentAmount ≥ denomination)
        currentAmount ← currentAmount − denomination
    return coinCount

procedure makeChange(amount)
    currentAmount ← amount
    quarters ← countCoins(currentAmount, 25)
    currentAmount ← currentAmount − (25 × quarters)
    dimes ← countCoins(currentAmount, 10)
    currentAmount ← currentAmount − (10 × dimes)
    nickels ← countCoins(currentAmount, 5)
    currentAmount ← currentAmount − (5 × nickels)
    pennies ← countCoins(currentAmount, 1)
    return [quarters, dimes, nickels, pennies]

var countCoins = function(amount, denomination) {
    var coinCount = 0;
    while (currentAmount >= denomination) {
        coinCount = coinCount + 1;
        currentAmount = currentAmount - denomination;
    }
    return coinCount;
}

var makeChange = function(amount) {
    var currentAmount = amount;
    var quarters = countCoins(currentAmount, 25);
    currentAmount = currentAmount - (25 * quarters);
    var dimes = countCoins(currentAmount, 10);
    currentAmount = currentAmount - (10 * dimes);
    var nickels = countCoins(currentAmount, 5);
    currentAmount = currentAmount - (5 * nickels);
    var pennies = countCoins(currentAmount, 1);
    return [quarters, dimes, nickels, pennies];
}

procedure listRPM(factor1, factor2)
    if (factor1 > factor2) then
        term1 ← factor2
        term2 ← factor1
    else
        term1 ← factor1
        term2 ← factor2
    addendList ← []
    while (term1 > 0) {
        if (term1 is odd) then
            add term2 to addendList
        term1 ← halfWithoutRemainder(term1)
        term2 ← double(term2)
    }
    product ← 0
    for each (number in addendList)
        product ← product + number
    return product

var listRPM = function(factor1, factor2) {
    var term1 = factor1; var term2 = factor2;
    if (factor1 > factor2) {
        term1 = factor2; term2 = factor1;
    }
    var addendList = [];
    while (term1 > 0) {
        if ((term1 % 2) == 1) {
            add addendList.push(term2);
        }
        term1 = parseInt(term1 / 2);
        term2 = term2 * 2;
    }
    var product = 0;
    for (var index = 0; index < addendList.length; index += 1) {
        product = product + addendList[index];
    }
    return product;
}
```