

A Hitchhiker's Guide to JavaScript Syntax

Written by Ben Yan, originally for the course ❤️ CS106S: Coding for Social Good ❤️.

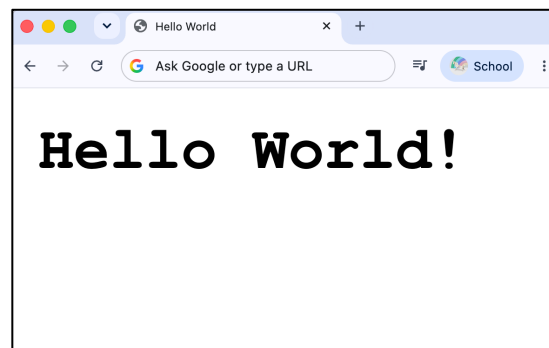
This handout records much of the basic syntax needed to write JavaScript code and programs, from print statements, to declaring and updating variables, to defining and calling functions, to conditional logic.

Topic	Pages
Printing Hello World	2
Debugging and JavaScript Console	2
Comments	2
JavaScript Arithmetic	3
Variables	4
Functions	4 – 5
Conditional Logic	5 – 6
Short Circuit Evaluation	7
More on Expression Evaluation	7 – 8
Function Declarations vs Expressions	8
Nested Functions and Closure	9

The last two sections are more esoteric compared to other programming languages, but useful for the JavaScript we teach in ✨ CS106AX ✨. **They are especially needed when writing graphics & animation programs**, where nested functions—functions defined and written in the body of other functions—completely run riot.



Syntax details for more involved data types and troublemakers—strings, arrays, objects—can be found in the corresponding lecture's notes, to avoid overwhelming you with JavaScript here ☺. You can also read about **for** and **while** loops in my *Lecture 2: JavaScript Fundamentals* notes.



Hello World

Here's a simple Hello World program in JavaScript!

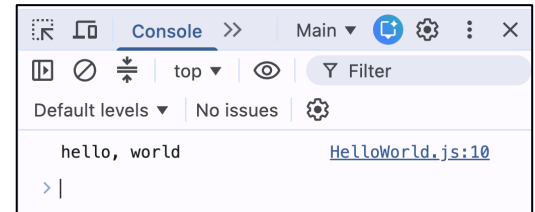
```
console.log("Hello World!");
```



You use `console.log(text)` to print to the console, as opposed to say `print(text)` in Python. If you try calling `print` in JavaScript, something very funny happens ☺ (it opens the physical printer app!)

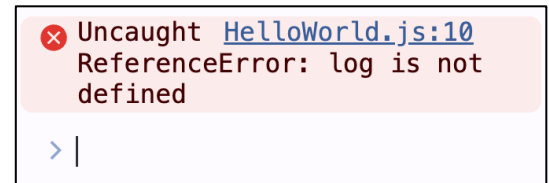
Debugging Console

To see where output from `console.log()` is delivered—unless you're using the `JSConsole.js` library to funnel it straight onto the webpage itself—you'll want to open the relevant HTML file for your JavaScript program in a browser.



Then, right-click the webpage, and click **Inspect** (as in, the notorious *Inspect Element* tool). In the resulting pop-up window, switch from the **Elements** tab to the **Console** one. Alternatively, on a Mac, you can press **Command**, then **Option**, then **J**, holding down each key while pressing the next.

This is a JavaScript console where you can type and run commands in real-time, but most importantly, **any prints from `console.log` statements in your program will show up here!** It's also where syntax and runtime errors in our code will be flagged, and you can also see the line numbers where they occur (see *right*), which is not a bad place to look 🔍.

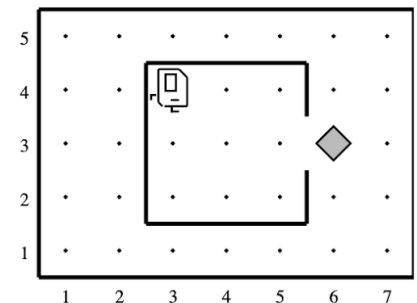


I introduced a bug in Line 10 by replacing `console.log()` with just `log()` ☺.

Comments

In JavaScript, comments begin with a `//`, and while they're not executed as lines of actual code, they can help improve readability and clarity, and signpost to the reader how your program is structured, e.g.

```
if (condition) { // robot goes left
    some code here ...
} else if (other condition) { // robot goes right
    some code here ...
} else { // robot stays on same square
    some code here ...
} don't worry about the if / else syntax here, just the commenting syntax!
```



To have comments span multiple contiguous lines, you can enclose the comment with a `/*` at the start, and a `*/` to mark the end. These are known as block or multiline comments. For instance,

```
/* Introducing myself to the computer, in a
   friendly way in case they take over the world */
console.log("Hi there, my name is Ben!");
```

When I first wrote that comment years ago, it was very much a joke, but it does seem kinda real now...

JavaScript Arithmetic

Note that math (e.g., the `+`, `-`, `*`, `/` operators) works similarly to other languages like Python and Java.

JavaScript represents all numbers as a unified `Number` type, so **the `/` operator between two integer values will perform regular division**, as opposed to integer division where the decimal part is truncated.

```
1 + 1; // => 2
10 - 4; // => 6
2 * 7; // => 14
3 / 2; // => 1.5
```

The `%` is the remainder or modulo operator, i.e. `n % k` evaluates to the remainder when `n` is divided by `k`. When `n < 0`, it's the negative of the remainder when the positive equivalent of `n` is divided by `k`.

```
4 % 2; // => 0
5 % 2; // => 1
6 % 2; // => 0
10 % 26; // => 10
```

Remainders can be useful for capturing alternating or "wrap-around" behavior in programs, e.g., time-keeping on clocks. If it's **8:00 PM** right now, then **7** hours later, it'll be **3:00 AM** (hopefully you don't stay up this late on a p-set!), as times wrap around the 12-hour clock. This could have been computed with the following JavaScript expression:



```
(8 + 7) % 12; // 8 + 7 => 15 o'clock hmm 🤔
// but clock only has 12 hours (0 - 11), so 15 % 12 => 3:00
```

The math library in JavaScript provides some useful operations as well, e.g.,

JavaScript Math Library Methods	
<code>Math.floor()</code>	
Returns	the largest integer not exceeding the input <code>x</code> , e.g., <code>Math.floor(3.14)</code> returns <code>3</code>
<code>Math.round(x)</code>	
Returns	the closest integer to the input <code>x</code> , e.g., <code>Math.round(2.78)</code> returns <code>3</code>
<code>Math.pow(x, y)</code>	
Returns	the value <code>x</code> raised to the <code>y</code> power, i.e. the exponent <code>x ^ y</code> .
<code>Math.sqrt(x)</code>	
Returns	the square root of <code>x</code> . Alternatively, you could call <code>Math.pow(x, 0.5)</code> .
<code>Math.max(x, y, ...)</code>	
Returns	the smallest of all the arguments. You won't be surprised there's a <code>Math.min(x, y, ...)</code> .

Lastly, having a semicolon (`;`) at the end of each statement is technically optional, as in it functionally it makes no difference in JavaScript. However, in 106AX, **our style expectations are that you include it** 😊

Variables

We use the **let** keyword to define variables of any type (whether numbers, strings, arrays, objects, etc.) The syntax in JavaScript takes the form:

```
let variableName = expression;
```

In the statement above, = is the assignment operator, and it assigns a value on the right side to the variable on the left side. It can also re-assign a new value to an existing variable! Here are examples.

```
let numClasses = 4;
numClasses = 5; // modifies value to 5
numClasses = numClasses - 2; // decreases to 5 - 2 => 3

let totalUnits = 12;
let avgUnitsPerClass = totalUnits / numClasses;
// the above variable equals 12 / 3 => 4
```



We also have the addition assignment (+=) operator, which is a pretty neat way to add a value to an existing variable. You can also do -= to subtract from a variable, *= to have it multiplied, and so forth.

```
numClasses += 1; // now back equal to 4
// same as numClasses = numClasses + 1, but shorter ☺
```

If you'd like to **increment** by precisely 1, you can do **numClasses++**; and similarly, if you'd like to **decrement** by precisely 1, you can do **numClasses--**; as a stylish shorthand.

For variables with fixed values, we can elect to use the **const** keyword instead.

```
const CS106AX_UNITS = 5; // 5-unit class, if you're an undergrad that is!
const LITERS_IN_A_GALLON = 3.785;
/* const variables should be all caps, with an underscore between words */
```

In older and legacy JavaScript code, you may encounter the keyword **var** in the place of **let** to declare variables. I used to code with **var** lol 🤖. As a general principle, avoid using **var**, it's washed and unc; if you're curious, see [here](#) for its subtle difference from **let**, which has to do with variable scope!

Functions

The general structure for defining a function in JavaScript is

```
function functionName( list of input parameters ) {
    list of statements in function body
}
```

You can optionally **return** a single value (of any type), with a statement of form **return expression**;, which communicates a value back to the caller function.

To see this in action, consider the following function, **addTwoNumbers**, which simply accepts two numbers as input, adds them together and gives back the result.

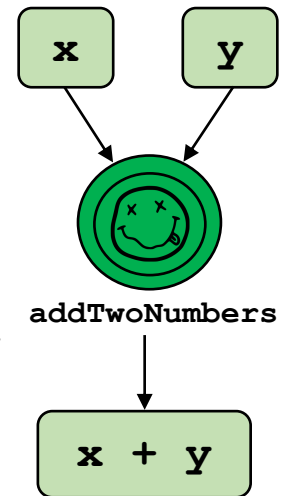


```
function addTwoNumbers(x, y) {
  let sum = x + y;
  return sum;
}
```

To call or invoke a function in JavaScript, you supply the function name, and pass in values or variable expressions for each input parameter.

```
let sum1 = addTwoNumbers(3, 4); // => 7
let sum2 = addTwoNumbers(-1, Math.PI); //=> 2.1415...

// can also compose function calls
let sum3 = addTwoNumbers(addTwoNumbers(1, 2), 3);
// this turns into addTwoNumbers(3, 3) => 6
```



If a function has a return value, when calling that function, you typically assign the result of the function call to a variable, e.g., `let result = func(inputs...)`; , which stores the value for future use ☺.

As another example, here's a JavaScript function that converts 🌡️ Celsius to Fahrenheit temperatures:

```
function celsiusToFahrenheit(celsius) {
  return 9 / 5 * celsius + 32;
}
```

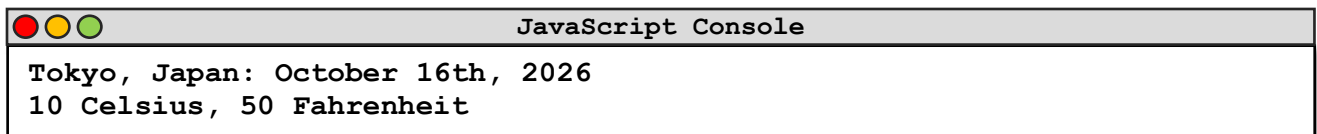
And here's a vignette of code that makes good use of the function above.

```
let TokyoC = 10;
let TokyoF = celsiusToFahrenheit(TokyoC);

console.log("Tokyo, Japan: October 16th, 2026")
console.log(TokyoC + " Celsius, " + TokyoF + " Fahrenheit");
```



The result that's printed to console is the accumulation of the string fragments above, to form:



Conditional Logic

The comparison operators `===` and `!==` check for equality and non-equality conditions, respectively

Note: We use three equal signs instead of two! It's quirky I know, but I think it's kind of beautiful ☺, and you'll get used to it. The other operators `<`, `<=`, `>=`, `>` behave as mathematically expected.

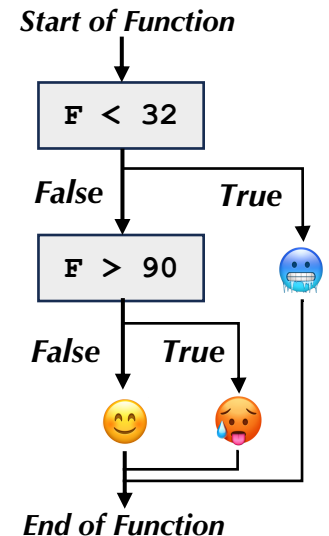
```
3 === 3 // => true
3 < Math.PI // => true
0.999 === 1 // => false
1/2 + 1/3 + 1/4 + 1/5 + 1/6 + 1/7 + 1/8 >= 2 // => false
```

Fun Fact: In general, use `===`, but JavaScript does support double equals (`==`), which is also called “*loose equals*”, as well as `!=`. Unlike `===`, *loose equals* doesn’t require the two values match in data type, and will convert one or both values into a matching type, which can lead to funny results, e.g.,

```
" " == 0; // => true, as the empty string is converted to a number, 0
" " == []; // => true, as the empty array is casted to an empty string ""
false == "0"; // => true, as both the Boolean false and string "0" are casted to number 0
```

Like most modern programming languages, JavaScript supports **if**, **else if**, and **else** conditional statements, for a program to behave differently or **branch under varying conditions**. To see these in action, consider a 🐼 **Goldilocks** function that takes, as input, the temperature outside, and judges it.

```
function Goldilocks(fahrenheit) {
  if (fahrenheit < 32) {
    console.log("It's too cold outside!");
  }
  // checks this only if above condition fails
  else if (fahrenheit > 90) {
    console.log("It's too warm outside!");
  }
  // only when both conditions above fail
  else{
    console.log("The weather is just right.");
  }
}
```



Note this function will print that it's "**just right**" only when the Fahrenheit temperature is between **32** and **90**, i.e. it's **at least 32** degrees AND it's **at most 90** degrees. Otherwise, it's too warm or cold ☹️.

Say instead we want a function that **returns the Boolean true** if the temperature is "just right" 😊, and **false** if it's too hot 🥵 or too cold 🥶. For this, in JavaScript, we can leverage the `&&` operator for logical AND, the `||` operator for the logical OR, and the unary operator `!` for the logical NOT.

```
function isGoldilocks(fahrenheit) {
  return fahrenheit >= 32 && fahrenheit <= 90;
}
```

Here's a function, **isNotGoldilocks**, which—surprise—returns the exact opposite.

```
function isNotGoldilocks(fahrenheit) {
  return fahrenheit < 32 || fahrenheit > 90;
  // true when weather too cold OR too hot, false when just right
}
```

Alternatively, you could call **isGoldilocks**, and flip the result with the logical NOT operator, i.e. `!`.

```
function isNotGoldilocks(fahrenheit) {
  return !isGoldilocks(fahrenheit);
  // with the ! operator, true becomes false, and false becomes true
}
```

Short Circuit Evaluation

JavaScript evaluates the `&&` and `||` operators using a time-saving strategy called **short-circuit mode**, in which it evaluates the right operand only if it needs to do so. For instance, in the Boolean expression

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

if `n` equals `0`, the right operand of `&&` is not evaluated at all, because `n !== 0` is **false**. In general,

`false && anything`

is always **false**, meaning the rest of the expression no longer matters. Similarly, the expression

`true || anything`

is always **true**, so the rest of expression is irrelevant to the result, and thus is not actually evaluated 😊.

An advantage of short-circuit evaluation is that you can strategically use `&&` and `||` to prevent runtime errors. If `n` were `0` in the earlier example, evaluating `x % n` would crash due to division by zero ☹️.

More on Expression Evaluation

JavaScript obeys the PEMDAS rule—perhaps reminiscent of one’s grade school math courses—for arithmetic **order of operations**. That is, in order of highest to lowest precedence, we evaluate

- **Parentheses** `()`
- then **Exponents** `**`
- then **Multiplication** `*`, **Division** `/`, and **Remainder** `%` operators (*tiebreaker, left to right*)
- then **Addition** `+` and **Subtraction** `-` operators (*tiebreaker, left to right*)



As a rule of thumb, we evaluate **arithmetic operators first** (`+`, `-`, `/`, `*`), then **comparison operators** (`<`, `<=`, `>`, `>=`, `===`, `!==`), then **logical / Boolean operators** (`&&`, `||`). One niche exception is logical NOT (`!`), which has high priority, and is evaluated before arithmetic operators but not parentheses.

For example, consider the expression `2025 > 2020 + 6 || 3 % 3 === 3`, which evaluates to

Arithmetic operators	→	<code>2025 > 2020 + 6 3 % 3 === 3</code>
Comparison operators	→	<code>2025 > 2026 0 === 3</code>
Logical operators	→	<code>false false</code>
		<code>false</code>



Let’s also consider the **evaluation of string expressions** 😊. Note that with JavaScript strings, **`+` is the concatenation operator, which combines strings end-to-end with no intervening characters**, e.g., `"blue" + "red"` evaluates to `"bluered"`. A string and a number be concatenated into a string as well, e.g., `"class" + 2030` evaluates to `"class2030"`, and `670 + "Lomita"` to `"670Lomita"`.

When concatenating multiple values in a row, note order of operations rules, keep track of type (string or number), and evaluate two at a time. For instance, the expression `"messier" + 2 * 3 + 7` evaluates to `"messier" + 6 + 7`, then `"messier6" + 7`, and lastly, `"messier67"`. Meanwhile, `2 * 3 + 7 + "messier"` evaluates to `6 + 7 + "messier"`, then `13 + "messier"`, and lastly, `"13messier"`. Spooky, but it’s different due to left-to-right rules and `+` as addition vs. concatenation!

Functions Declarations vs. Expressions

We've talked about two main ways to create a function in JavaScript, which are **declarations** and **expressions**. A **declaration** simply uses the **function** keyword and specifies a function name, e.g.,

```
function greet(name) {
    console.log("Hello " + name + "!");
}
```



The second way, which we witness more often in graphical programs (especially ones that involve interactivity / animation), creates a **function expression**, and assigns it to a variable that can be called.

```
let greet = function (name) {
    console.log("Hello " + name + "!");
}
```



A key functionality difference is that **declared** functions are fully loaded before any code has even run. Thus, you can call / refer to them even before the line of code they're defined in, e.g., the below is ok!

```
drawCircle(); // executes the (pre-loaded) function below!
```

```
function drawCircle() {
    ... code that draws a randomly colored circle on the canvas ...
}
```

Meanwhile, the code below, involving a function **expression**, would crash ☹. This is because function expressions are defined only when we reach their line of code upon executing the program line-by-line.

```
drawCircle(); // Reference Error: Can't access before initialization
```

```
let drawCircle = function() {
    ... same code that draws a randomly colored circle on the canvas ...
}
```

Despite the runtime risks lol, you may stylistically prefer a function expression with **let** to **emphasize that functions**, like any other data type in JavaScript (e.g., numbers, strings, arrays), **are stored as data variables**, and thus, **they be passed as parameters into other functions**, e.g., in our **Draw Dots** program,

```
let drawDot = function(e) {
    ... (Lecture 4) code that draws a filled dot centered at where the user clicked ...
}

gw.addEventListener("click", drawDot); // passes function as variable in!
```

You can also pass a function as an input parameter without explicitly assigning it a variable name, e.g., below. A caveat is that you can't refer to this function later in your code, as it's not stored in a variable.

```
gw.addEventListener("click", function(e) {
    ... same code that draws a filled dot centered at where the user clicked ...
});
```

Nested Functions and Closures

JS supports nested functions, i.e. **functions defined in the body of another function**. Here's an example that may resonate with you, with Stanford, the GovCo area, and two of its dorms: Potter and Schiff ☺.

```
function Stanford() {
  let Farrillaga = 341; /* address is 341 Galvez St */
  let Tresidder = 459; /* address is 459 Lagunita Drive */
  ... code here can access Farrillaga and Tresidder ...

  function GovCo() {
    let Ricker = 238; /* address is 238 Santa Teresa St */
    ... code here can access Ricker, Farrillaga, and Tresidder ...

    function Potter() {
      let Cats = 3;
      ... code here can access Cats, Ricker, Farrillaga, and Tresidder ...
    }

    function Schiff() {
      let Dogs = 4;
      ... code here can access Dogs, Ricker, Farrillaga, and Tresidder...
    }
  }
}
```



The **closure** of a function is defined as a **record storing a function together with all the variables it had access to at the time of creation**. One can think of closure metaphorically as an ID keycard that retains access to these variables, even when the outer function has stopped executing. In JavaScript, **the scope rules are that each function has access to its local variables, but also upward access to local variables in its outer functions**. However, the reverse is not true: an outer function cannot access its inner functions' own variables, creating a membrane seal of privacy, e.g., a **Govco resident*** not in **Potter** or **Schiff** can't access their **Cats** or **Dogs**, whereas a **Potter resident*** can access **Cats**, but also has keycard access to variables **Ricker**, **Farrillaga**, and **Tresidder** in the nesting dolls of its outer functions.



***resident** = line of code residing in a function (my metaphor ☺)

