Human-Computer Interaction Programming Studio

COGS121 Spring 2019

Instructor: Philip Guo

JavaScript Language Fundamentals

YouTube videos of these lectures are available at:

https://www.youtube.com/playlist?list=PLzV58Zm8FuBJFfQN5il3ujx6FDAY8Ds3u

Learning Objective

to give you a practical understanding of JavaScript as a programming language so that you can more easily learn specialized frameworks and libraries later as needed.

Outline

- What is JavaScript?
- Why learn JavaScript fundamentals?
- Learn the fundamentals via code examples

What is JavaScript?

(what are your thoughts? this isn't a rhetorical question.)

GitHub Octoverse 2017 | High
Philip
Secure | https://octoverse.github.com

The fifteen most popular languages on GitHub

by opened pull request

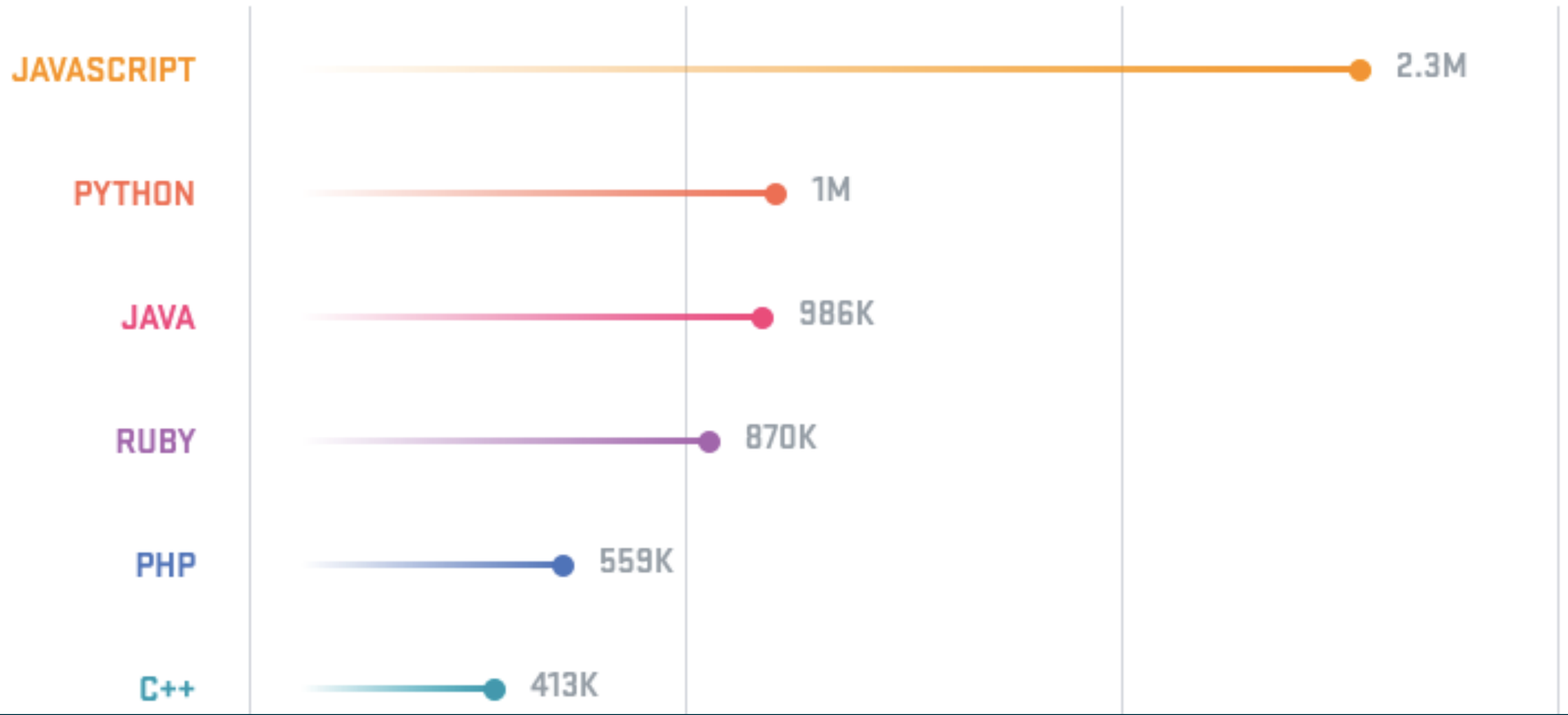
GitHub

01 GitHub is home to open source projects written in 337 unique programming languages—but especially JavaScript.

02

03

04



stackoverflow

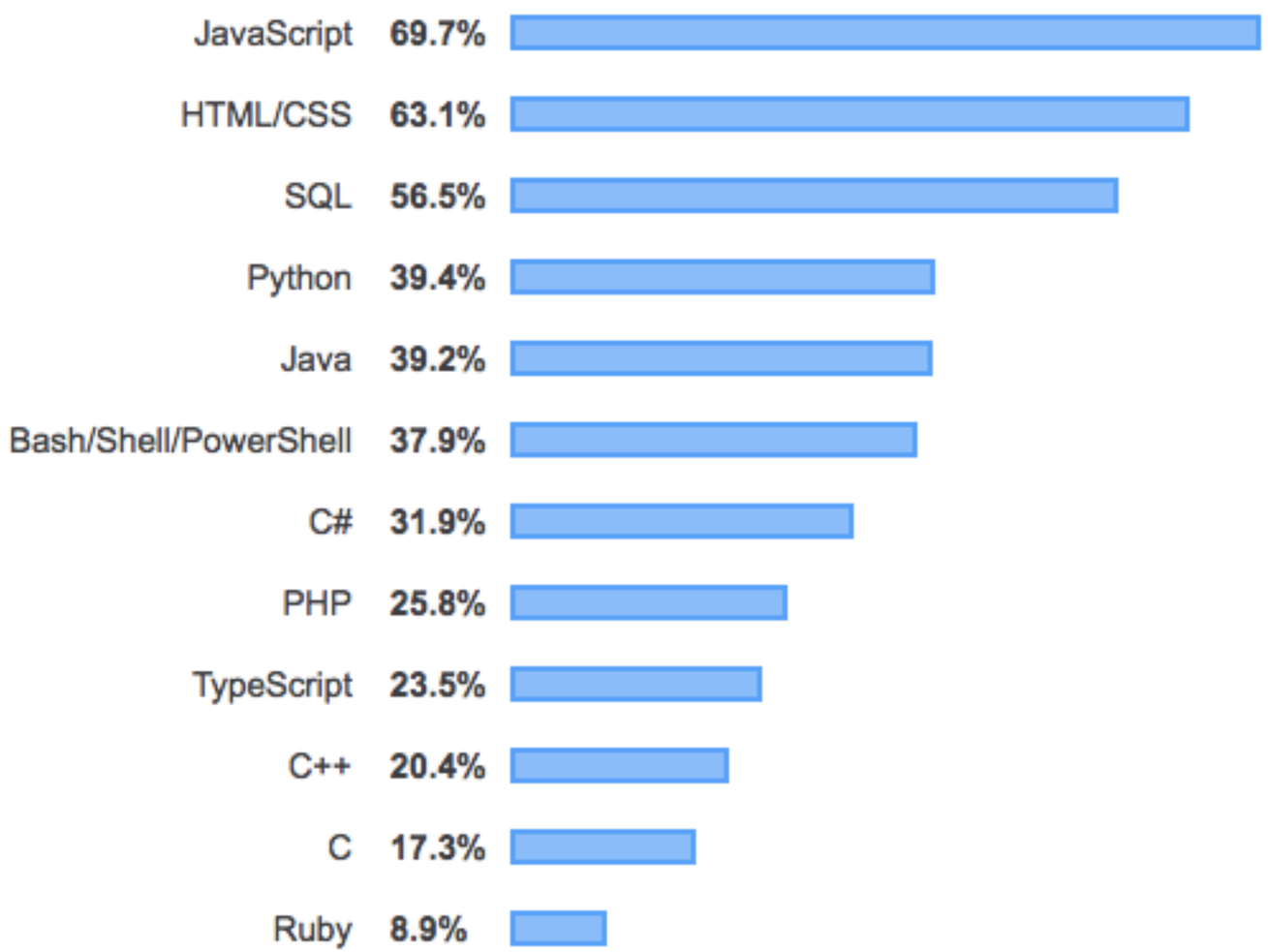
Back to top

Take control of your job search.

Most Popular Technologies

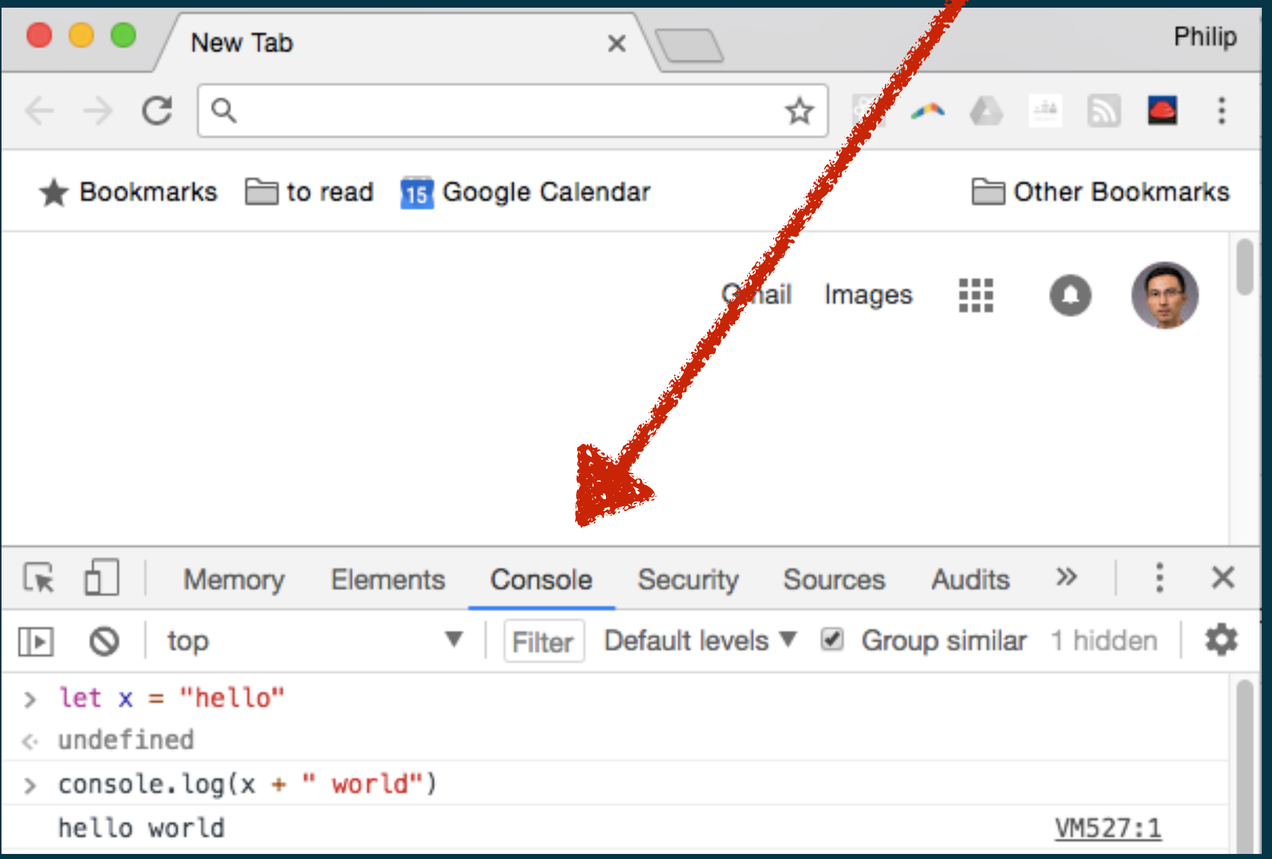
Programming, Scripting, and Markup Languages

All Respondents | Professional Developers

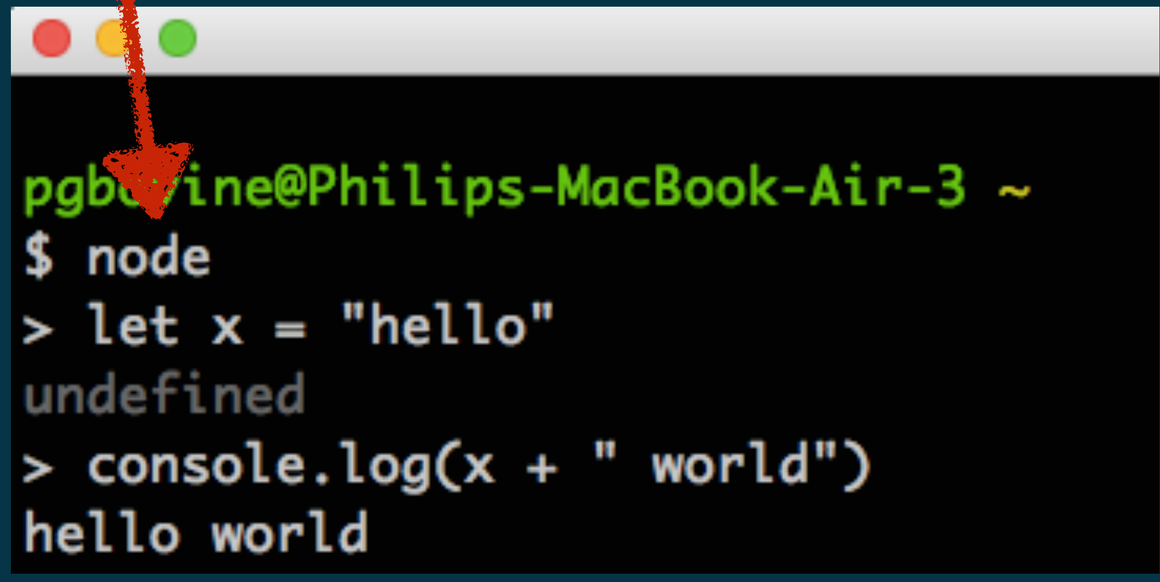


JavaScript programming language

It's just a normal language. Similar-ish to Python.



Runs in all web browsers



Runs on the command-line on any computer (like Python)

(also starting to be used for native mobile apps, but less common)

https://insights.stackoverflow.com/survey/2019#developer-profile

koverflow

v

ults

er Profile

aphy

loper Roles

rience

ation

graphics

uating Competence

Outside Work

ogy

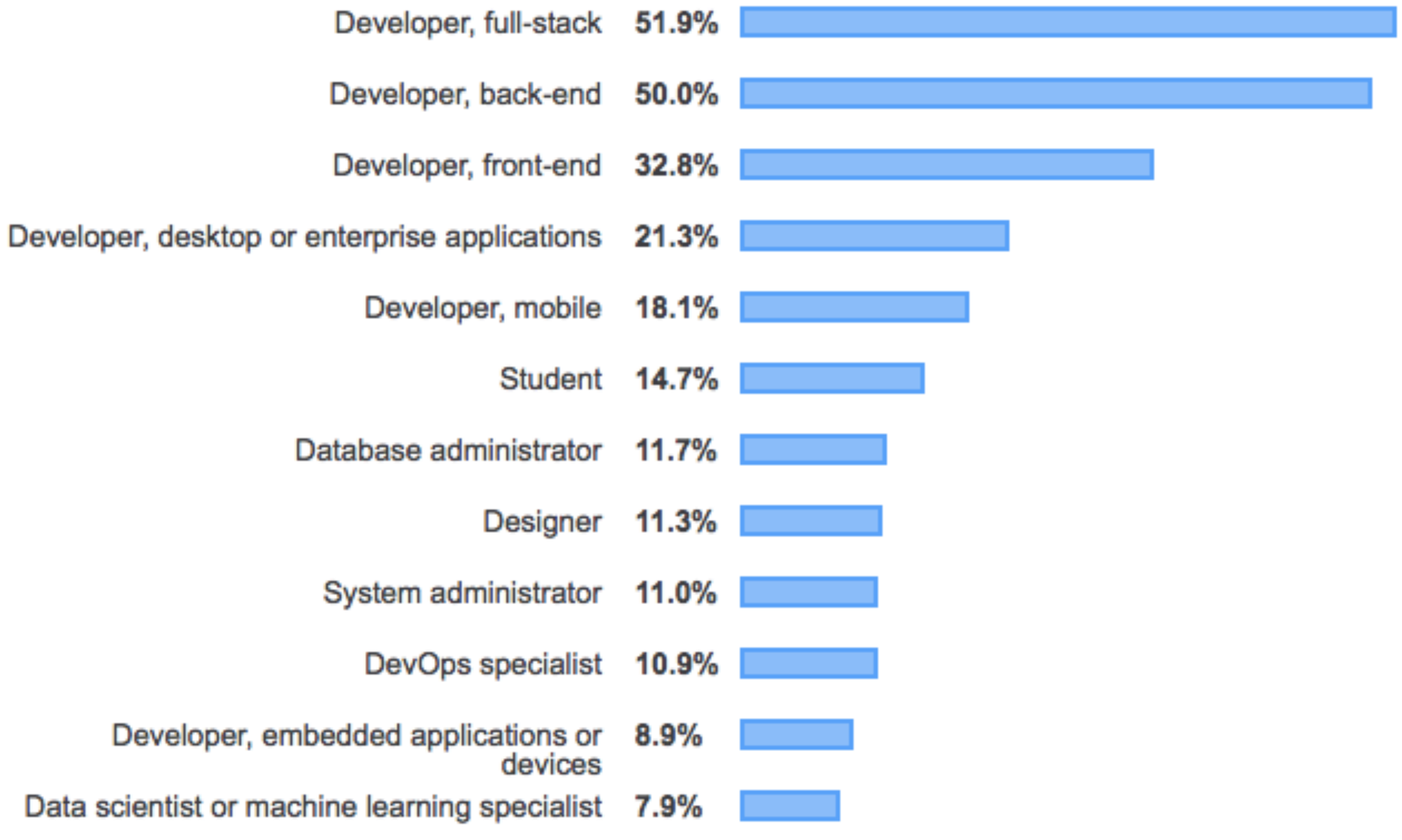
nity

ology

op

Developer Type

All Respondents | United States Unweighted | ★ United States Weighted by Gender



JavaScript programming language

Web browser API
(select and modify webpage elements, set mouse click handlers, etc.)

jQuery

Frontend web frameworks
e.g., Angular, React, Vue

Node.js server API

Backend web frameworks
e.g., Express

Database APIs
e.g., MySQL, MongoDB

JavaScript programming language

New libraries and frameworks crop up every month. It's **impossible** to keep up with latest developments …

Example of head-spinning complexity: Web Developer Roadmap - 2018

jQu

PIs

e.g., Angular, React, Vue

JavaScript programming language

… but a good understanding of the language itself will give you confidence when working with any framework or environment.

jQu

PIs

”

e.g., Angular, React, Vue

JavaScript programming language

That's why we're going to spend several lectures on JavaScript language fundamentals.

jQu

PIs

e.g., Angular, React, Vue

JavaScript programming language

This may be review for some of you, but even experienced web developers have gaps in their understanding of fundamentals.

jQu

PIs

"

e.g., Angular, React, Vue

JavaScript programming language

JavaScript has had a drama-filled history since it came out in 1995 … lots of complexity about which language features to use to still support older web browsers.

We will learn the **ES6** version of the language released in 2015, and ***loosely*** follow coding style guides from Google and Airbnb.
(you don't need to study these guides, they're just references)

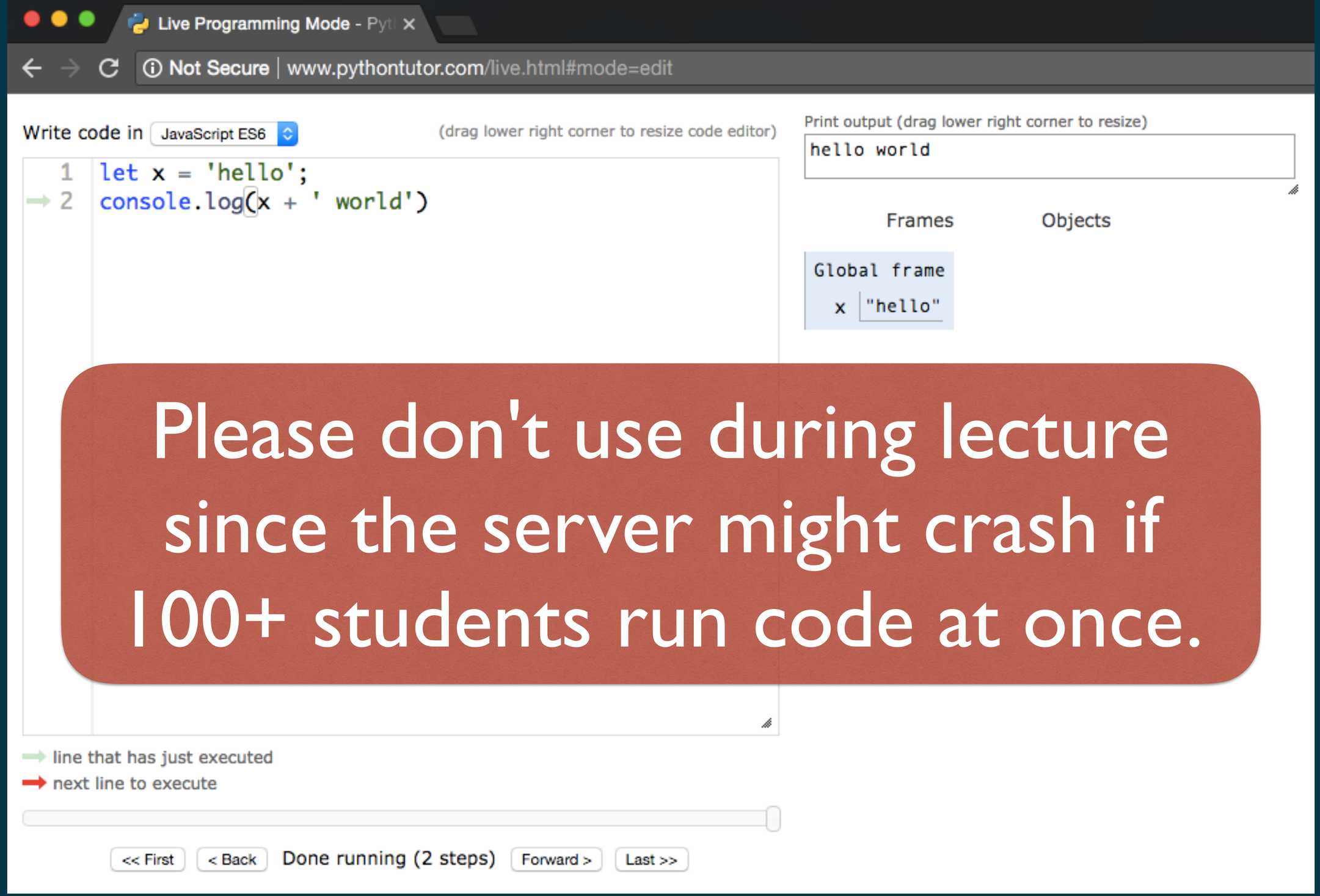
I will be loosely following these style guides for writing modern JavaScript code. However, you will likely find a lot of old code on the internet that does not conform to these styles. That's OK! For your project, as long as your code works and meets the project requirements, that's fine.

Prerequisite: I'm assuming you already have a fair amount of programming experience from prior classes and/or jobs. This isn't an introduction to programming.

Prerequisite 2: I'm assuming you've had some prior exposure to JavaScript via web programming.

Scope: I will cover the subset of JavaScript that you are most likely to encounter in practice as a novice- to intermediate-level full-stack web developer.

These lectures will be me showing lots of code examples. I will update these slides and put up videos after lecture, so don't worry if you aren't able to understand everything now. **Ask me questions!**



My JavaScript visualizer: http://pythontutor.com/live.html#mode=edit&py=js&demo=1

Chrome File Edit View History Bookmarks People Window Help

New Tab

Search G

Other Bookmarks

Always Show Bookmarks Bar ⇧⌘B
✓ Always Show Toolbar in Full Screen ⇧⌘F

Stop ⌘.
Force Reload This Page ⇧⌘R

Enter Full Screen ^⌘F
Actual Size ⌘0
Zoom In ⌘+
Zoom Out ⌘−

Cast...

Developer ▶

View Source ⌥⌘U
Developer Tools ⌥⌘I
Inspect Elements ⌥⌘C
JavaScript Console ⌥⌘J
Allow JavaScript from Apple Events

You've g

Now you can browse privately, and other
activity. However, downloads and bookmarks will be saved. Learn more

Chrome **won't save** the following information:

- Your browsing history
- Cookies and site data
- Information entered in forms

Your activity **might still be visible** to:

- Websites you visit
- Your employer or school
- Your internet service provider

Elements Console Sources Network Performance Memory Application »

top Filter Default levels

```
> let x = 'hello';
<· undefined
> console.log(x + ' world');
  hello world                                    VM85:1
<· undefined
>
```

Console What's New ×

Variables and values of primitive types

Declaring variables:
use const or let

```
const x = 12345;
```

```
let x = 12345;
```

```
var x = 12345; // seen in older code
```

```
x = 12345; // don't do this; confusing
```

Updating the value of a variable

```
const x = 12345;
x = 42; // error
```

```
let x = 12345;
x = 42; // OK
```

A single variable can hold values of any type (like Python, unlike Java)

```
let x = 12345;
x = 42;
x = 'hello';
x = true;
x = null;
x = undefined; // avoid this; confusing
```

(JavaScript is a *dynamically typed* language)

Use console.log() to print to terminal or browser console

```
let x = 12345;
x = 42;
console.log(x);
x = 'hello';
console.log(x);
x = true;
console.log(x);
```

JavaScript's main primitive types
(remember, *value*s have types, variables don't)

```
let x = 12345; // number
console.log(typeof(x));

x = 3.14159;  // also a number
x = 'hello';  // string
x = "hello";  // OK, but prefer 'hello'
x = true;     // boolean
x = null;
x = undefined;
```

Operations on primitive types
(remember, *value*s have types, variables don't)

```
let x = 12345;
let y = x * -1;
```

```
let x = 'hello';
let y = 'hello' + 'world';
```

```
let x = true;
let y = !x;
```

Variable scoping

So far we've been putting all of our variables in a single global scope

```
let x = 12345;
let y = 'hello';
let z = true;
let x = 42; // error, already declared in this
            // scope! remove 'let' and it's OK
```

Create new scopes with { braces }

```
let x = 12345;
let y = 'hello';
let z = true;
{
  let x = 42; // what happens without 'let'?
  console.log(x);
}
console.log(x);
```

Each { } is called a *block*. 'let' and 'const' variables are *block-scoped*.

What happens if you use 'const'?
('const' is just like 'let', except you can't reassign)

```
const x = 12345;
{
  const x = 42;
  console.log(x);
  // you can't reassign 'x' here
}
console.log(x);
// you can't reassign 'x' here either
```

Each { } is called a *block*. 'let' and 'const' variables are *block-scoped*.

let and const variables exist only in their defined scope and inner scopes

```
{
  let x = 42;
  console.log(x);
}
console.log(x); // what happens here?
```

```
let x = 42;
{
  console.log(x); // what happens here?
}
console.log(x);
```

What happens if you use 'var'?
(modern style guides say: don't use 'var')

```
var x = 12345;
{
  var x = 42;
  console.log(x);
}
console.log(x);
```

'let' and 'const' variables are *block-scoped*. But 'var' are NOT block-scoped.

When would you actually use {braces} ?

```
let x = 12345;
let y = 'hello';
let z = true;
if (z) { // if statements (or if-else)
  let x = 42;
  console.log(x);
} else {
  let x = -3.14159;
  console.log(x);
}
console.log(x);
```

When would you actually use {braces} ?

some places you might use braces to define new scopes:

- if statements
- switch statements
- for loops
- while loops
- functions

Google and Airbnb style guides:

Use '**const**' as much as you can.

Use 'let' only when you really need to.

Compound values: arrays and objects

So far all of our variables have held only one item at once ...

```
let x = 12345;
let y = 'hello';
let z = true;
x = 42; // replaces 12345
x = 'bob'; // replaces 42
```

What if you want x to hold more than one item?

Declare arrays using [brackets]

```
const x = [1, 2, 3, 4, 5];
```

```
const x = ['hello', 'world', 'goodbye'];
```

```
// arrays can hold values of multiple types
const x = [12345, 42, 'bob'];
```

```
// no more trying to do this:
let x = 12345;
x = 42; // replaces 12345
x = 'bob'; // replaces 42
```

Declare arrays using [brackets]

```
const x = ['hello', 'world', 'goodbye'];
```

Get array contents using [brackets]

```
const x = ['hello', 'world', 'goodbye'];
const y = x[0];
const z = x[1];
const w = x[2];
const err = x[3];
```

Modify the array's contents using [brackets] (even if it's declared as 'const')

```
const x = ['hello', 'world', 'goodbye'];
x[0] = 'HELLO THERE!';
x[1] = 'earthlings';
x.push(3.14159);
x.pop();
```

Why can we still modify the contents of 'const' x?

What does 'const' mean then?!?

it means you can't reassign x to something else using '='

```
const x = ['hello', 'world', 'goodbye'];
x.push(3.14159); // OK
x = [1, 2, 3, 4, 5]; // nope!
```

```
let x = ['hello', 'world', 'goodbye'];
x.push(3.14159); // OK
x = [1, 2, 3, 4, 5]; // OK
// Question: what happens to first array?
```

What does the assignment operator '=' do for arrays, versus for primitive values?

```
const x = ['hello', 'world', 'goodbye'];
const y = x;
x[0] = 'HELLO THERE!';
console.log(y[0]); // what happens here?
```

```
const x = ['hello', 'world', 'goodbye'];
const y = ['hello', 'world', 'goodbye'];
x[0] = 'HELLO THERE!';
console.log(y[0]); // what happens here?
```

What does the assignment operator '=' do for arrays, versus for primitive values?

```
const x = 'hello';
const y = x;
```

```
const x = 'hello';
const y = 'hello';
```

It doesn't matter which you do here, since primitive values are *immutable* - you can't change what's inside of them

What does the assignment operator '=' do for arrays, versus for primitive values?

```
const x = ['hello', 'world', 'goodbye'];
const y = x;
```

```
const x = ['hello', 'world', 'goodbye'];
const y = ['hello', 'world', 'goodbye'];
```

This distinction matters for compound values like arrays (and objects, as we'll see later)

Use 'for-of' loops to iterate thru arrays
(according to Google style guide)

```
const x = ['hello', 'world', 'goodbye'];
for (const e of x) {
  console.log('ELEMENT: ' + e);
}
```

Arrays hold an ordered collection of items indexed by position, but what if we want to associate pairs of items?

e.g., a person's name to their age
a zip code to its city
a piece of merchandise to its price
a food item to its nutritional value

Objects hold associations between *keys* and *values*. Declare using { braces }

```
const ages = {'joe smith': 25, 'pamela z': 30};

// no quotes needed if no spaces:
const ages = {joe: 25, pamela: 30};

// values can be of any type
const ages = {joe: [1,2,3], pamela: [4,5,6]};
```

Modify objects using [brackets] or dot

```
const ages = {'joe smith': 25, 'pamela z': 30};


ages['joe smith'] = 50;
ages['bobby bob'] = 100;
```

```
const ages = {joe: 25, pamela: 30};


ages.joe = 50;
ages.bobby = 100;
ages['bobby bob'] = 200;
```

What does the assignment operator '=' do for objects? (similar to arrays)

```
const x = {joe: 25, pamela: 30};
const y = x;
x.bobby = 100;
console.log(y.bobby);
```

```
const x = {joe: 25, pamela: 30};
const y = {joe: 25, pamela: 30};
x.bobby = 100;
console.log(y.bobby);
```

Use 'for-of' loops to iterate thru the *keys* of objects (according to Google style guide)

```
const ages = {joe: 25, pamela: 30, bobby: 100};
const myKeys = Object.keys(ages);
for (const e of myKeys) {
  console.log(e, '->', ages[e]);
}
```

```
const ages = {joe: 25, pamela: 30, bobby: 100};
for (const e of Object.keys(ages)) { // don't need 'myKeys'
  console.log(e, '->', ages[e]);
}
```

You might sometimes see 'for-in' loops for objects, but those suffer from subtle problems. Use Object.keys() and 'for-of'.

Arrays and objects can be nested

```
const myData = [
  {name: 'Philip G', age: 25},
  {name: 'Jane D', age: 30},
  {name: 'Bobby B', age: 48},
  {name: 'Carol C', age: [60, 70, 80, 90]},
]
```

An array of objects; one object even has an array inside of it.

Recap: primitive and compound values

```
// primitive values:
const x = 12345;     // number
const y = 'hello';   // string
const z = true;      // boolean


// compound values:
// array
const stuff = [1, 2, 3, false, 4, 'world', true];
// object
const ages = {joe: 25, pamela: 30, bobby: 100};
```

Recap of variables, primitive values, and compound values:

Use '**const**' as much as you can.

Use an array or object to hold a group of related items.

Then use [ ] to get individual items and '**for-of**' loops to iterate thru all items.

Be careful when using '=' on arrays/objects. 'x=y' doesn't copy the actual array/object.

Functions

I'm assuming you're all familiar with defining and calling functions, so I'll dive into more JavaScript-specific parts.

Functions define new scopes

```
const x = 123;
const y = 456;

function doStuff(x) {
  x = 'joe';
  let y = 'jane';
  console.log(x, y); // what happens here?
  y = 'JANE!';
}

doStuff(x);
console.log(x, y);   // what happens here?
```

Passing parameters is like assignment with '='

```
const x = 123;
const y = 456;

function doStuff(x) {
  x = 'joe';
  let y = 'jane';
  console.log(x, y); // what happens here?
  y = 'JANE!';
}

doStuff(x);
console.log(x, y);   // what happens here?
```

Passing parameters is like assignment with '='

```
const x = [1, 2, 3];
const y = [4, 5, 6];

function doStuff(x) { // what happens here?
  x.push('hello');
  x[0] = 3.14159;
  console.log(x);
}

doStuff(x);
doStuff(y);
console.log(x, y);
```

... so be careful with arrays/objects!

```
const x = [1, 2, 3];

function doStuff(x) { // doesn't copy array
  x = ['hello', 'world'];
  console.log(x); // what happens here?
}

doStuff(x);
console.log(x); // what happens here?
```

Functions are just objects

```
function square(x) {return x * x;}
function cubed(x) {return x * x * x;}

const mySquare = square;
const funcs = [mySquare, cubed];

const a = square(3);
const b = mySquare(3);
const c = cubed(10);
const d = funcs[1](10);
square = "hello";
square(3); // what happens here?
```

Functions can be declared in 2 ways:

```
function max1(a, b) {
  if (a > b) return a;
  else return b;
}

const max2 = (a, b) => {
  if (a > b) return a;
  else return b;
};
```

(There are other ways, but style guides recommend these two)

Use the 'function' keyword for top-level functions that are NOT inside of any other { braces }

```
function max1(a, b) { // good - top-level!
  if (a > b) return a;
  else return b;
}
```

Use the 'function' keyword for top-level functions that are NOT inside of any other { braces }

```
function max1(a, b) { // GOOD - top-level!
  if (a > b) return a;
  else return b;
}

let x = 100;
if (x > 10) {
  function maxbad(a, b) { // BAD - inside braces!
    if (a > b) return a;
    else return b;
  }
}
```

Use the arrow keyword '=>' to define functions inside of { braces }

```
// a top-level function using 'function'
function myBigFunction() {

  // define a function NESTED inside
  // of myBigFunction
  const max2 = (a, b) => {
    if (a > b) return a;
    else return b;
  }
}
```

Why do this? Why not simply use 'function' for nested functions too? Because of subtle issues with the JavaScript '**this**' variable (don't worry about this slide for exams)

```
// a top-level function using 'function'
function myBigFunction() {

  // define a function NESTED inside
  // of myBigFunction
  const max2 = (a, b) => {
    if (a > b) return a;
    else return b;
  }
}
```

You can pass functions as arguments to function calls.
Why would you ever want to do this?!?

Remember those 'for-of' loops?

```
const x = ['hello', 'world', 'goodbye'];
for (const e of x) {
  console.log('ELEMENT: ' + e);
}
```

You can pass functions as arguments to function calls. Why would you ever want to do this?!?

Remember those 'for-of' loops?

Google recommends using for-of loops to iterate over array/object items, but Airbnb recommends ***NOT*** using loops. Instead, it wants you to use functions such as forEach(), map(), etc. to iterate. This is known as a ***functional programming style.***
(you can do whatever you want, as long as it works!)

Use forEach() to iterate over array/object items

```
const x = ['hello', 'world', 'goodbye'];
for (const e of x) { // for-of loop
  console.log('ELEMENT: ' + e);
}
```

```
const x = ['hello', 'world', 'goodbye'];
function printMe(e) {
  console.log('ELEMENT: ' + e);
}
x.forEach(printMe); // no loop!
```

Does printMe really need a name? It's used only once

```
const x = ['hello', 'world', 'goodbye'];
function printMe(e) {
  console.log('ELEMENT: ' + e);
}
x.forEach(printMe); // no loop!
```

```
const x = ['hello', 'world', 'goodbye'];
x.forEach((e) => {
  console.log('ELEMENT: ' + e);
}); // no loop, no named function
```

This is called an ***anonymous function***, and it's very common in JavaScript. Define these using arrow '=>' and ***never*** by using the 'function' keyword. Anonymous functions usually occur inside of (parentheses) since they are passed as arguments to other functions.

```
const x = ['hello', 'world', 'goodbye'];
x.forEach((e) => {
  console.log('ELEMENT: ' + e);
});
```

```
const x = ['hello', 'world', 'goodbye'];
// forEach can also take a 2-argument function
x.forEach((e, i) => {
  console.log(i, 'ELEMENT: ' + e);
});
```

The map() function also takes a function as an argument. It returns a *new* array with each element processed by that function. Example:

```
const x = ['hello', 'world', 'goodbye'];
const y = [];
for (const e of x) { // use for-of loop
  y.push(e.toUpperCase());
}
```

What are the advantage of map versus for-of?!?

```
const x = ['hello', 'world', 'goodbye'];
const y = x.map((e) => {
  return e.toUpperCase();
}); // visualizer doesn't show this well :/
```

for-of loop says: "*create a new empty array y, now iterate through each element of the x array, assign e to it, then uppercase e, then push a new element onto the y array.*"

```
const x = ['hello', 'world', 'goodbye'];
const y = [];
for (const e of x) { // use for-of loop
  y.push(e.toUpperCase());
}
```

map() says: "*y should be an uppercased version of x.*"

```
const x = ['hello', 'world', 'goodbye'];
const y = x.map((e) => {
  return e.toUpperCase();
});
```

```
const x = ['hello', 'world', 'goodbye'];
const y = x.map((e) => {
  return e.toUpperCase();
});
```

If you have only one argument and one return statement, you can simplify the syntax to make this declaration even smaller.

```
const x = ['hello', 'world', 'goodbye'];
const y = x.map(e => e.toUpperCase());
```

map() says: *"y should be an uppercased version of x."*

Iterating through items in objects

```
const ages = {joe: 25, pamela: 30, bobby: 100};
for (const e of Object.keys(ages)) { // for-of loop
  console.log(e, '->', ages[e]);
}
```

```
const ages = {joe: 25, pamela: 30, bobby: 100};
Object.keys(ages).forEach((e) => { // no loop
  console.log(e, '->', ages[e]);
});
```

Recap of functions:

Use 'function' to declare top-level functions

Use the '=>' arrow syntax to declare functions inside of { braces } and to declare anonymous functions usually inside of (parentheses)

Functions can be assigned to variables and passed as arguments to function calls

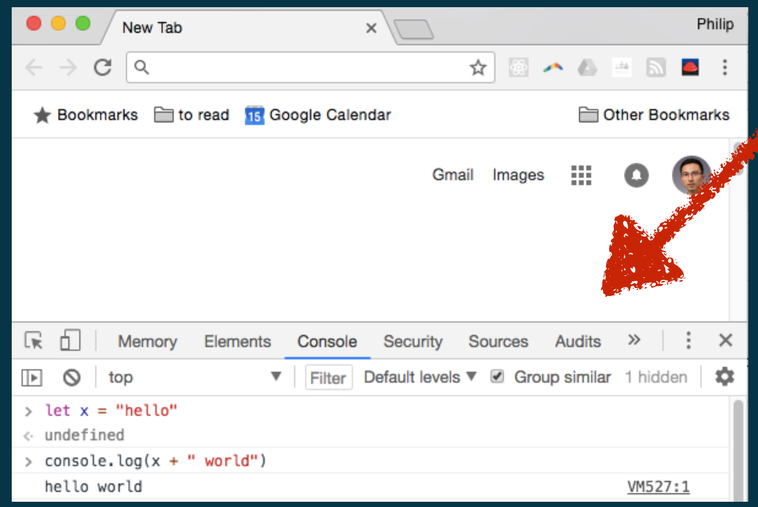
To iterate over arrays/objects, use functions such as forEach() and map() instead of loops

Asynchronous Programming

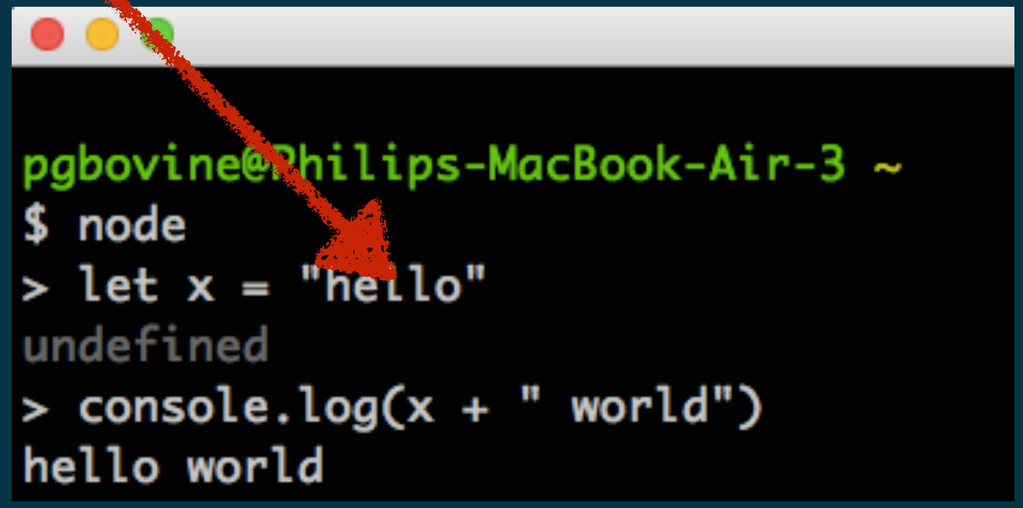
So far, all of your code has executed sequentially one line after another.

Your program ends when the final line of executed code finishes.

JavaScript programming language



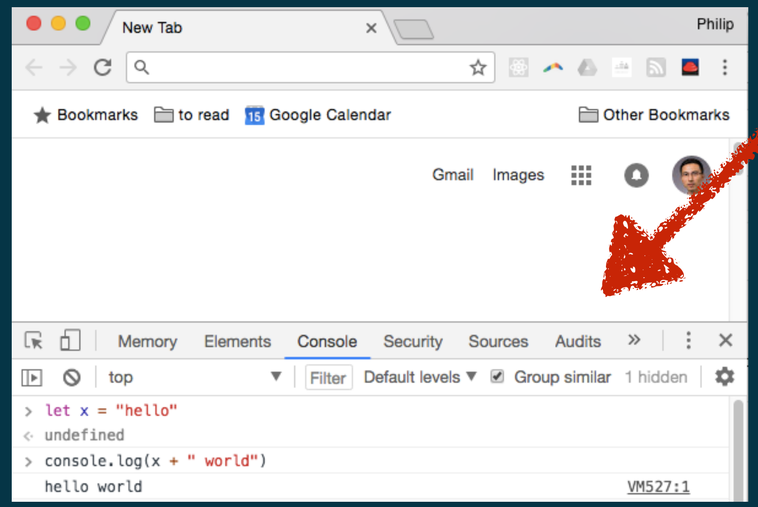
Runs in a web browser



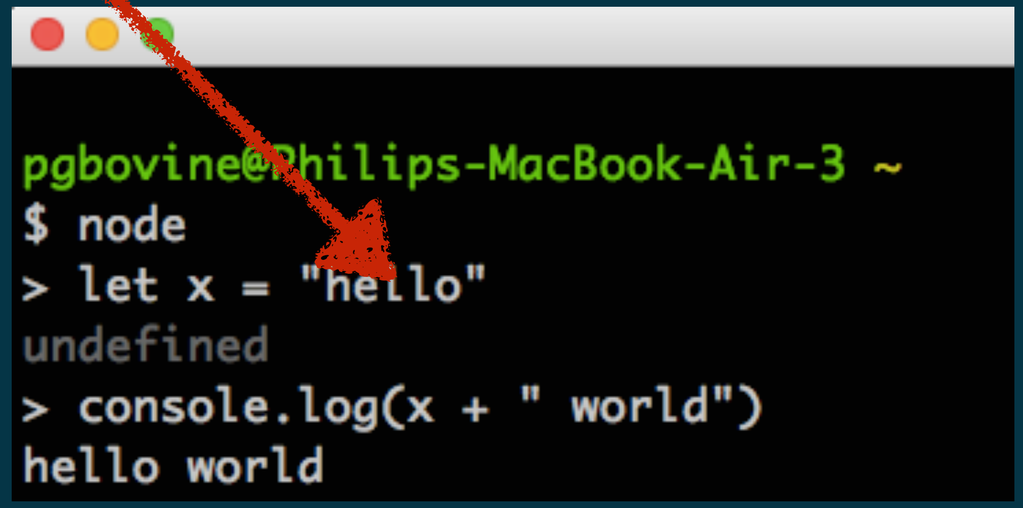
Runs on a web server

So far, all of your code has executed sequentially one line after another. Your program ends when the final line of executed code finishes.

JavaScript programming language



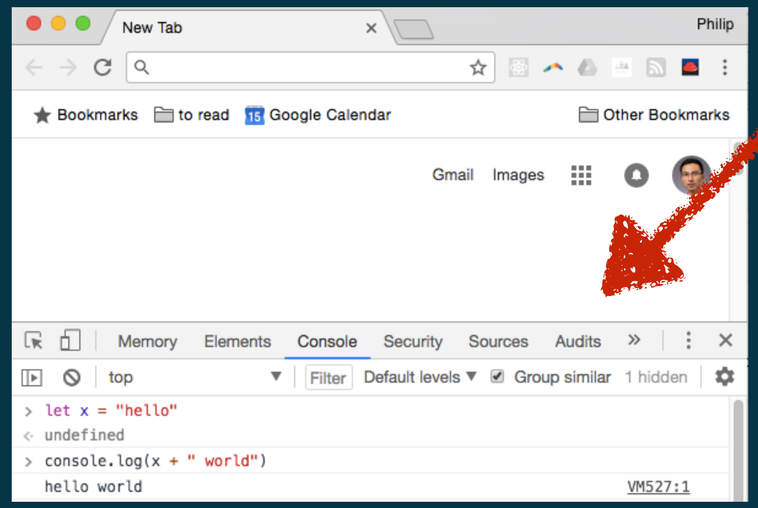
Runs in a web browser



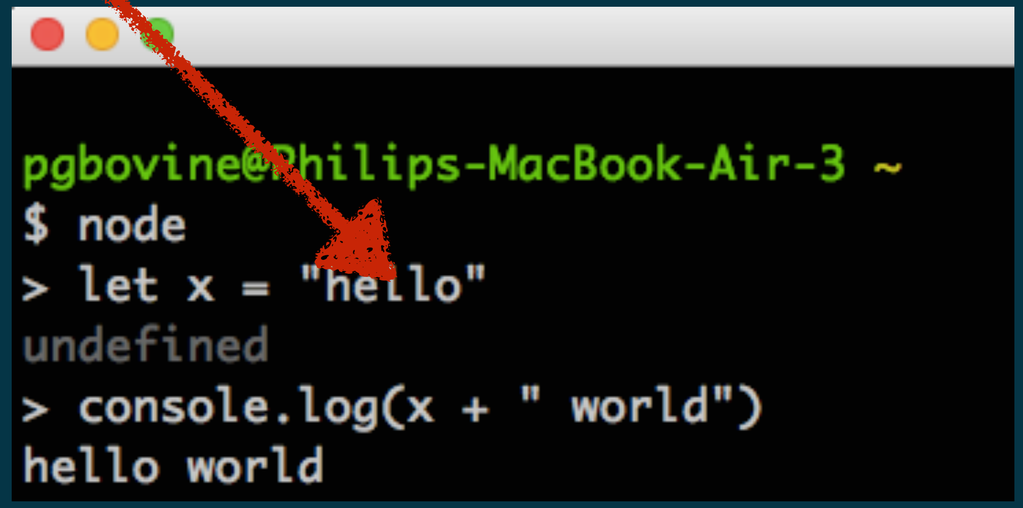
Runs on a web server

~~*So far, all of your code has executed sequentially one line after another. Your program ends when the final line of executed code finishes.*~~

JavaScript programming language



A web browser's JavaScript code must initialize the webpage and then sit idle to wait for user input, such as mouse or keyboard events.



A web server's JavaScript code must open a connection to the internet and then sit idle to wait for requests to be sent to it.

```
// pseudocode (not real!)
initializeWebpage();
while (true) {
  e = checkForUserInput();
  if (e.type == "mouse") {
    handleMouseEvent(e);
  } else if (e.type == "key") {
    handleKeyboardEvent(e);
  }
  ...
}
```

A web browser's JavaScript code must initialize the webpage and then sit idle to wait for user input, such as mouse or keyboard events.

```
// pseudocode (not real!)
initializeWebserver();
while (true) {
  r = checkForHttpRequest();
  if (r) {
    handleHttpRequest(r);
  }
  ...
}
```

A web server's JavaScript code must open a connection to the internet and then sit idle to wait for requests to be sent to it.

```
// pseudocode (not real!)
initializeWebpage();
while (true) {
  e = checkForUserInput();
  if (e.type == "mouse") {
    handleMouseEvent(e);
  } else if (e.type == "key") {
    handleKeyboardEvent(e);
  }
  ...
}
```

```
// pseudocode (not real!)
initializeWebserver();
while (true) {
  r = checkForHttpRequest();
  if (r) {
    handleHttpRequest(r);
  }
  ...
}
```

For efficiency and reliability reasons, these are implemented as a single thread of execution. It's *conceptually* a single infinite loop that goes on forever until you exit the program.

```
// pseudocode (not real!)
handleHttpRequest(r) {
  query = r.parameters[0];
  result = databaseLookup(query);
  sendDataToClient(result);
}
```

```
// pseudocode (not real!)
initializeWebserver();
while (true) {
  r = checkForHttpRequest();
  if (r) {
    handleHttpRequest(r);
  }
  ...
}
```

Typical task: client (e.g., web browser) sends an HTTP request to server, server looks at its URL parameters, makes a database query, then sends the data back to the client.

e.g., http://facebook.com/getProfile?name=Alice&location=SanDiego

```
// pseudocode (not real!)
handleHttpRequest(r) {
  query = r.parameters[0];
  result = databaseLookup(query);
  sendDataToClient(result);
}
```

```
// pseudocode (not real!)
initializeWebserver();
while (true) {
  r = checkForHttpRequest();
  if (r) {
    handleHttpRequest(r);
  }
  ...
}
```

What if **databaseLookup(query)** is really slow? e.g., it takes 5 seconds. What happens to the server if other requests now come in? (there's only a single thread)

e.g., http://facebook.com/getProfile?name=Alice&location=SanDiego

```
// pseudocode (not real!)
handleHttpRequest(r) {
  query = r.parameters[0];          // 0.001 seconds
  result = databaseLookup(query);   // 5 seconds
  sendDataToClient(result);         // 0.01 seconds
}
```

This function takes 5.011 seconds to run.
The server can't handle incoming requests
for the next 5 seconds!!! That's terrible.

```
// pseudocode (not real!)
handleHttpRequest(r) {
  query = r.parameters[0];          // 0.001 seconds
  result = databaseLookup(query);   // 5 seconds
  sendDataToClient(result);         // 0.01 seconds
}
```

How much time would this function call take if it didn't need to wait for the database lookup to finish?

0.001 + 0.01 = **0.011 seconds.**

What if we could let the server handle other requests while it waited for the database to finish its lookup?

```
// pseudocode (not real!)
handleHttpRequest(r) {
  query = r.parameters[0];          // 0.001 seconds
  result = databaseLookup(query);   // 5 seconds
  sendDataToClient(result);         // 0.01 seconds
}
```

What if we could let the server handle other requests while it waited for the database to finish its lookup?

Original:

```
handleHttpRequest(r) {
  query = r.parameters[0];          // 0.001 seconds
  result = databaseLookup(query); // 5 seconds
  sendDataToClient(result);         // 0.01 seconds
}
```

Asynchronous Programming:

```
handleHttpRequest(r) {
  query = r.parameters[0];                // 0.001 seconds
  databaseLookupAsync(query, finisher); // 0.001 seconds
}
// runs 5 seconds later ... in the meantime,
// the server can handle other requests
function finisher(result) {
  sendDataToClient(result);               // 0.01 seconds
}
```

What does databaseLookupAsync() look like? You don't have to worry about it; the library creator writes it.

```
databaseLookupAsync(query, finisher) {
  // magically dispatch 'query' to the database and
  // tell it to call 'finisher' when the database
  // finishes its job after 5 seconds
  return; // returns RIGHT AWAY to caller in 0.001 seconds
}
```

Asynchronous Programming:

```
handleHttpRequest(r) {
  query = r.parameters[0];                  // 0.001 seconds
  databaseLookupAsync(query, finisher); // 0.001 seconds
}
// runs 5 seconds later ... in the meantime,
// the server can handle other requests
function finisher(result) {
  sendDataToClient(result);                 // 0.01 seconds
}
```

Both versions take 5.011 total seconds to run. So why bother with asynchronous programming?

Original:

```
handleHttpRequest(r) {
  query = r.parameters[0];          // 0.001 seconds
  result = databaseLookup(query); // 5 seconds
  sendDataToClient(result);         // 0.01 seconds
}
```

Asynchronous Programming:

```
handleHttpRequest(r) {
  query = r.parameters[0];                // 0.001 seconds
  databaseLookupAsync(query, finisher); // 0.001 seconds
}
// runs 5 seconds later ... in the meantime,
// the server can handle other requests
function finisher(result) {
  sendDataToClient(result);               // 0.01 seconds
}
```

Let's say there are 3 nearly-simultaneous HTTP requests from clients A, B, and C:

Original:

B has to wait ~10 seconds,
C has to wait ~15 sec!!!

Asynchronous:

Everyone is done after ~5 seconds!

```
handleHttpRequest(r) {
  query = r.parameters[0];                // 0.001 seconds
  databaseLookupAsync(query, finisher); // 0.001 seconds
}
// runs 5 seconds later ... in the meantime,
// the server can handle other requests
function finisher(result) {
  sendDataToClient(result);               // 0.01 seconds
}
```

Let's say there are 3 nearly-simultaneous HTTP requests from clients A, B, and C:

Original:

B has to wait ~10 seconds,
C has to wait ~15 sec!!!

Asynchronous:

Everyone is done after ~5 seconds!

Asynchronous programming allows the server to process 3 requests concurrently with only a single thread of execution.

This frequently happens in the web browser too. Imagine A, B, and C are three user input events in the browser ... imagine A takes a long time (e.g., fetching Twitter data from API) but B and C are *FAST*.

Original:

B and C both have to wait 5 seconds!

Browser freezes for 5 seconds while handling A!!!

Asynchronous:

B and C finish instantly

A finishes after 5 seconds, but that's unavoidable

Browser always feels responsive to user inputs.

To perform asynchronous programming, pass a ***callback function*** into special functions that were designed for this style of programming.

Regular named callback function

```
handleHttpRequest(r) {
  query = r.parameters[0];                  // 0.001 seconds
  databaseLookupAsync(query, finisher); // 0.001 seconds
}

// runs 5 seconds later ...
function finisher(result) {
  sendDataToClient(result);                 // 0.01 seconds
}
```

Anonymous callback function *(very common!)*

```
handleHttpRequest(r) {
  query = r.parameters[0];                    // 0.001 seconds
  databaseLookupAsync(query, (result) => { // 0.001 seconds
    // runs 5 seconds later ...
    sendDataToClient(result); // 0.01 seconds
  });
}
```

Recap: Asynchronous programming is widely used in web servers and browsers to try to do more than one thing at once.

It allows servers to handle concurrent requests faster and browsers to be more responsive to user inputs.

It relies on passing a *callback function* into specialized functions that were designed with asynchronous programming in mind.

Unfortunately my JavaScript visualizer doesn't work with asynchronous programming, so we won't be able to do live coding demos. But we will see this programming style MANY times throughout the remaining lectures.

Other major JavaScript concepts that we don't have time to cover in these lectures:

- Comparison operators and equality
- Defining nested functions and closures
- Errors and exception handling
- JavaScript 'this' keyword
- Object-oriented programming: defining your own classes
- Defining and using modules
- Promises and async / await syntax as more advanced ways of doing asynchronous programming

Lots more info in Mozilla's JavaScript reference docs

Learning Objective

to give you a practical understanding of JavaScript as a programming language so that you can more easily learn specialized frameworks and libraries later as needed.

TODOs after class

- Make sure you're on Piazza
- Think about form team of 3-4 members