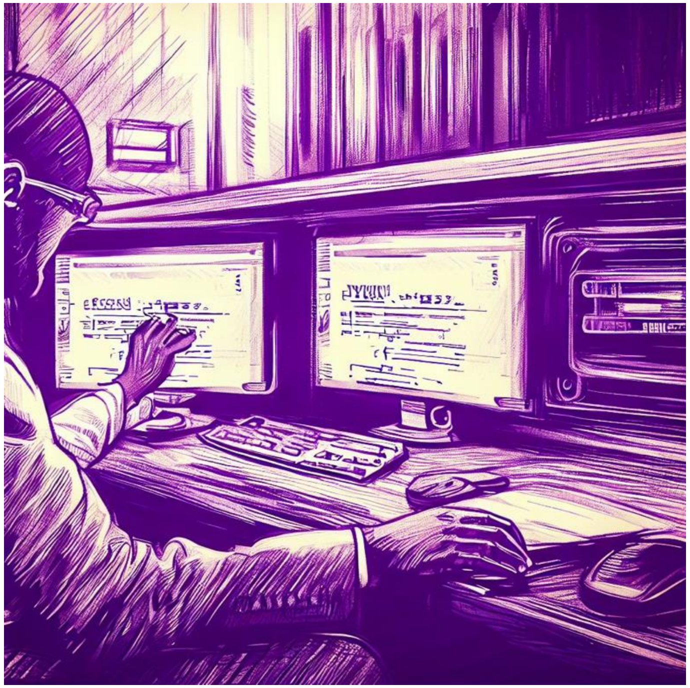


CSE 331

Software Design & Implementation

Intro to JavaScript

James Wilcox and Kevin Zatloukal

Your instructors

- Professional programmers with 30+ years experience

James Wilcox

- Built a wide range of systems and applications

Systems

- compilers
- operating systems
- distributed systems
- networking systems
- database systems
- graphics
- …

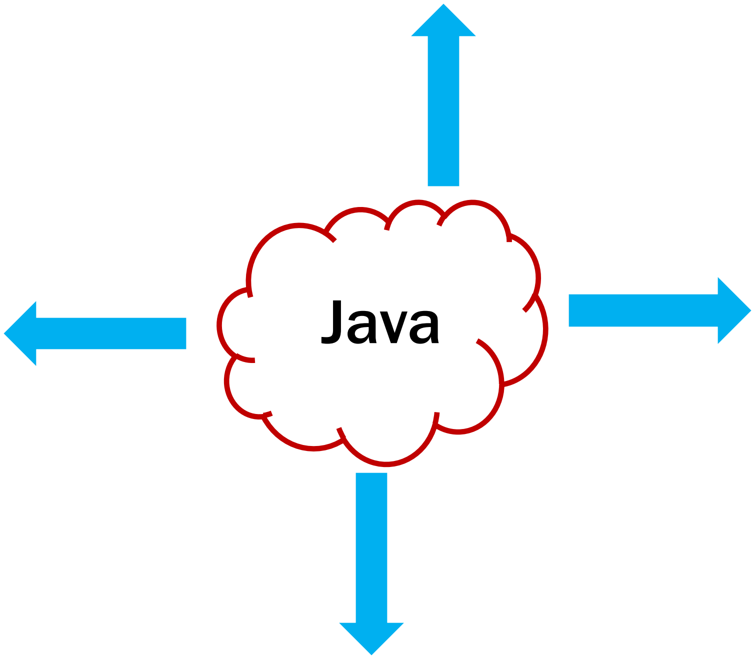
Applications

- desktop apps
- web apps
- phone apps
- IDE
- games
- …

Kevin Zatloukal

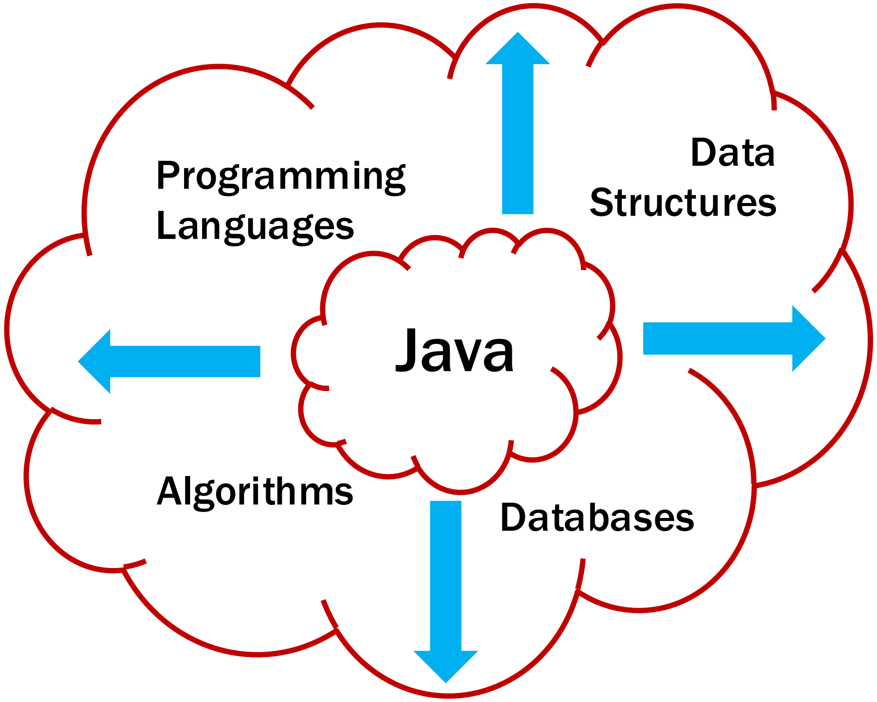
Learning Computer Science

- You already know Java
  - some basic data structures and algorithms
- Working on expanding your knowledge

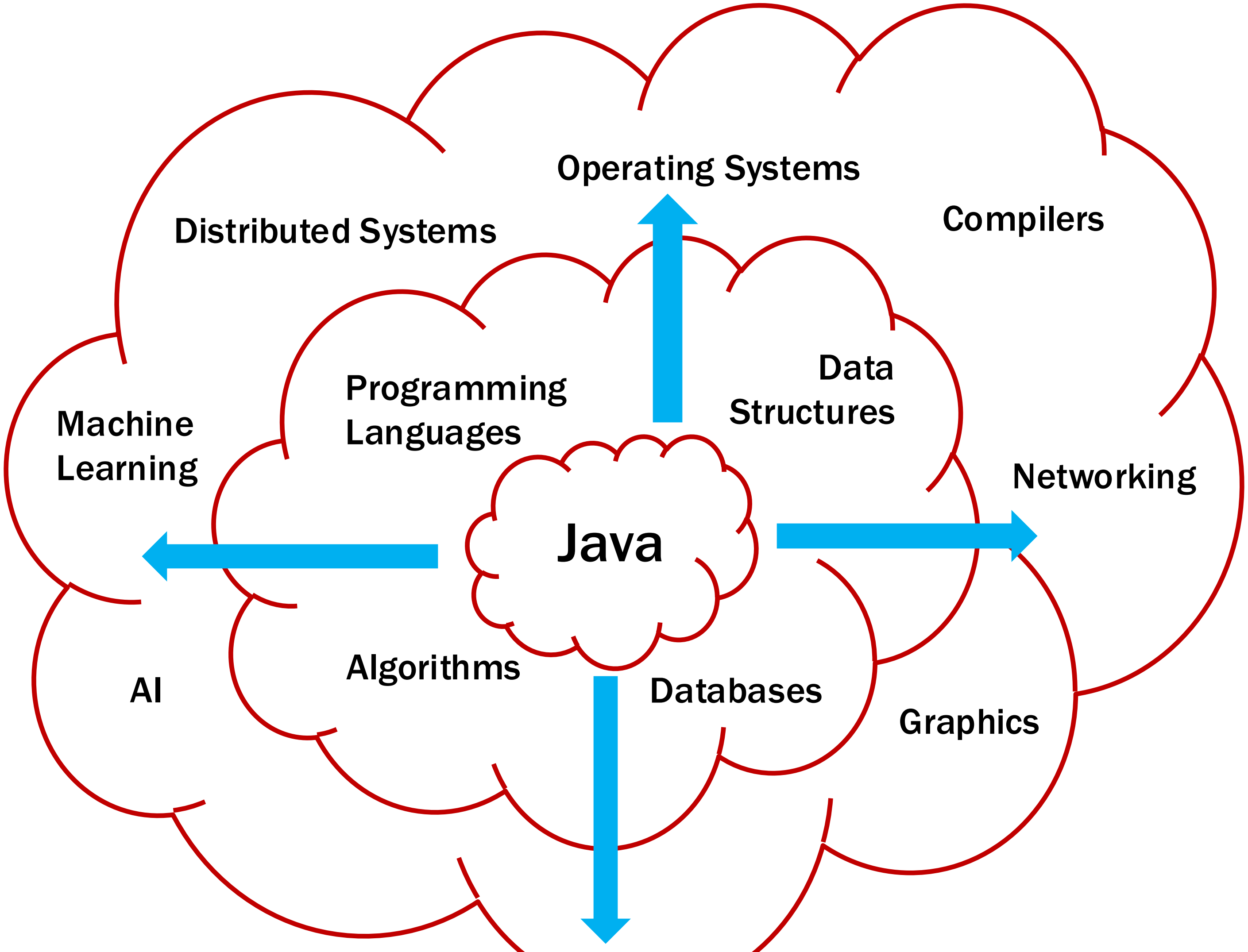


Learning Computer Science

- You already know Java
  - – some basic data structures and algorithms
- Working on expanding your knowledge



Learning Computer Science



Learning Computer Science

1. **First time solving this kind of problem**

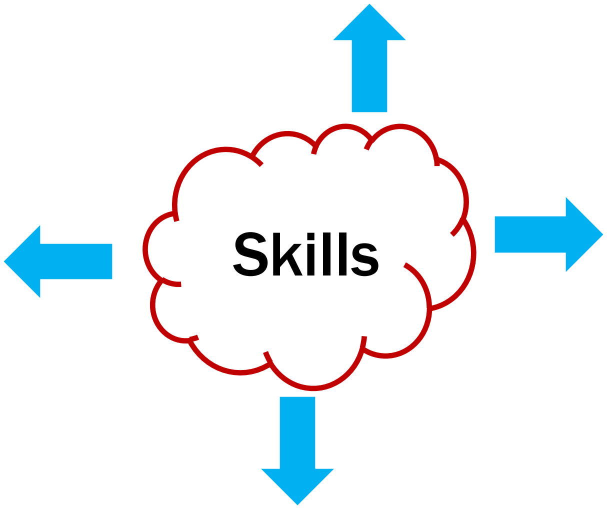
2. **Given lots of help**

   will often tell you if it's right

3. **Expected to make mistakes**

   90% is an "A"!

All of these are
<u>different</u> in industry

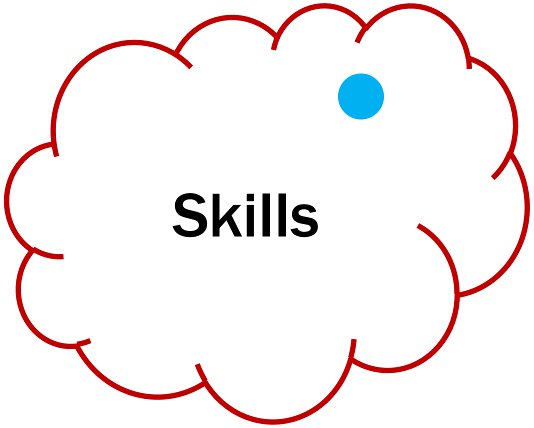


Practicing Computer Science

1. **Not the first time solving this kind of problem** 😀

   normal to hire someone with prior experience

   learn new skills in class or in spare time



Practicing Computer Science

1. **Not the <u>first time</u> solving this kind of problem** 😀

   normal to hire someone with prior experience

   learn new skills in class or in spare time

2. **<u>No one</u> to tell you if your code is right**

   That's your job!

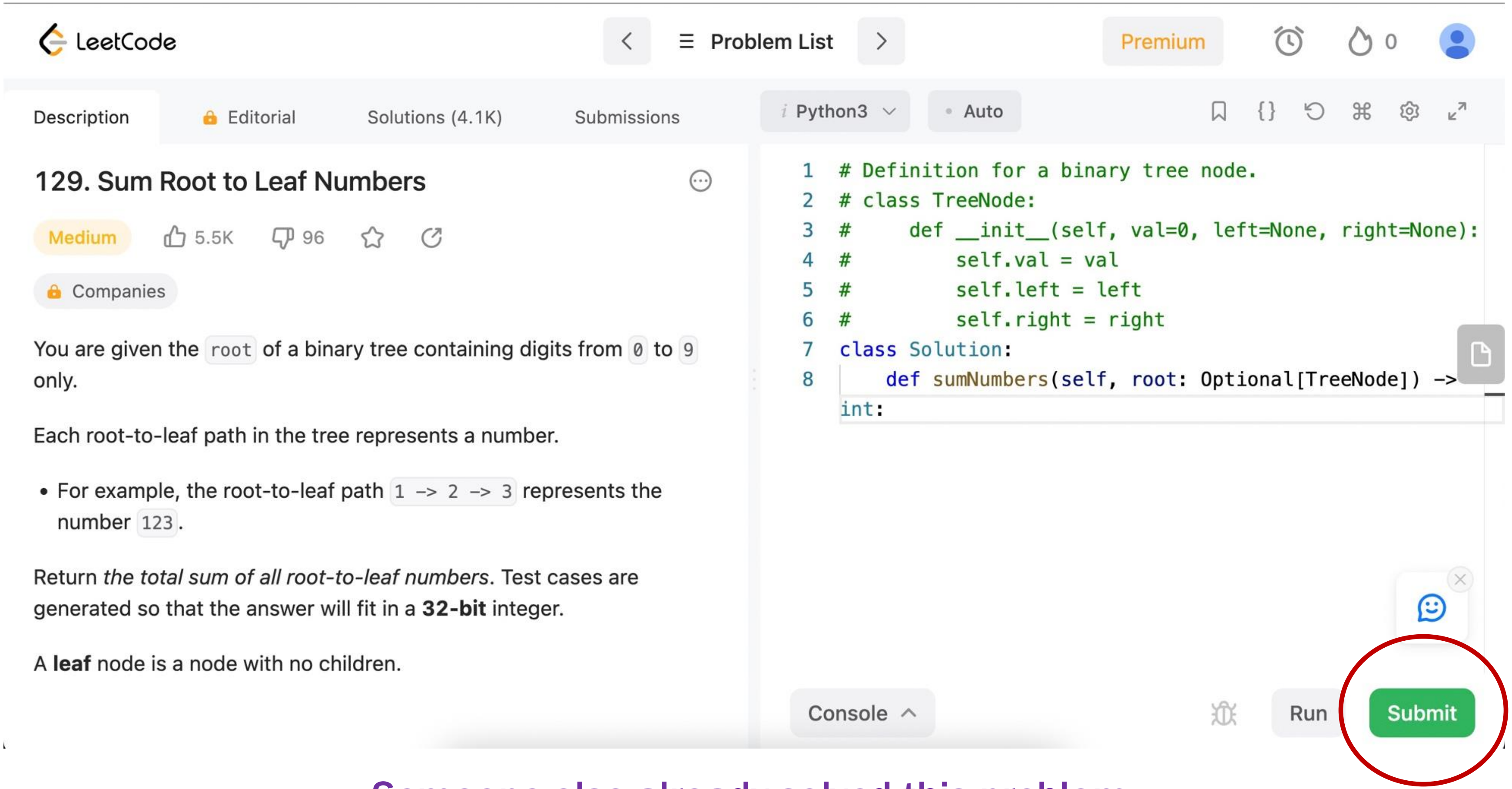
   (senior engineers will *double check* your work, but they expect it to be right)

   you will almost never be given tests

Skills

Least “Real World” Setting Possible

Would give you a button to click to see if it’s right…



129. Sum Root to Leaf Numbers

You are given the `root` of a binary tree containing digits from `0` to `9` only.

Each root-to-leaf path in the tree represents a number.

- For example, the root-to-leaf path `1 -> 2 -> 3` represents the number `123`.

Return *the total sum of all root-to-leaf numbers*. Test cases are generated so that the answer will fit in a **32-bit** integer.

A **leaf** node is a node with no children.

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def sumNumbers(self, root: Optional[TreeNode]) -> int:
```

Someone else already solved this problem.
They only need you for new problems.

Practicing Computer Science

1. **Not the first time solving this kind of problem** 😃

   normal to hire someone with prior experience

   learn new skills in class or in spare time

2. **No one to tell you if your code is right** ☹

   That’s your job!

   (senior engineers will *double check* your work, but they expect it to be right)

   you will almost never be given tests

Skills

Practicing Computer Science

1. Not the first time solving this kind of problem 😃

normal to hire someone with prior experience

learn new skills in class or in spare time

2. No one to tell you if your code is right ☹

That's your job!

(senior engineers will *double check* your work, but they expect it to be right)

you will almost never be given tests

3. Mistakes are not acceptable (to users)

90% is not an "A"

10% of 1m users is 100k users calling customer service

1% of 1m users is 10k users calling customer service

Skills

What This Class is About

- Learning what engineers do to make sure their code is correct <u>before</u> sending it to users

- Learn a toolkit for being 100% sure it is right
  - any “computer scientist” must know this

- Learn when to use the toolkit
  - not every problem requires it

We Will Ask You to Write Code Differently

- Our goal is **not** to teach you to write code that looks exactly like what you will see in industry
  - nor is it to use the libraries most common in industry

    the most popular languages and libraries change all the time

- Our goal is to teach you to **think** through your code and to **understand** how all the parts work

- That is best served by writing slowly and carefully

- We will force that by
  1. changing programming languages to something *unfamiliar*
  2. having *unusual* coding conventions at times

Homework

- CSE 331 is a **hard** class
  - because coding & debugging are hard!

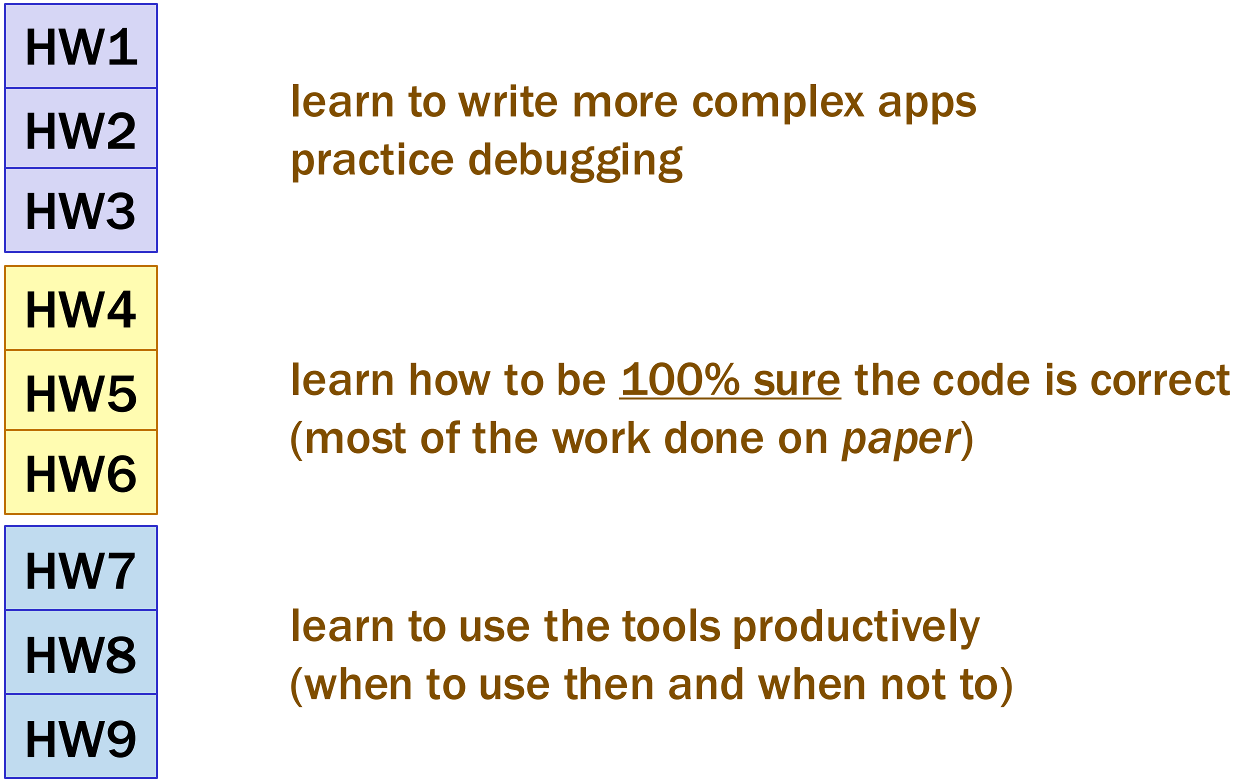
- Most of the work is done **outside of class**
  - university policy is 2 hours per hour of class time
  - plan for 8 hours per week, but…

- Wide variation in time required
  - some students will average 10-15 hours

    but this is not expected!
    be sure to get help if you are averaging over 15 hours

Homework Assignments

- Nine assignments split into these groups:



HW1
HW2
HW3

learn to write more complex apps
practice debugging

HW4
HW5
HW6

learn how to be 100% sure the code is correct
(most of the work done on *paper*)

HW7
HW8
HW9

learn to use the tools productively
(when to use then and when not to)

Learning a New Language

- We’re going to learn some JavaScript

- The second language can be the hardest to learn!
  - some things you took for granted no longer hold
  - must slow down think about think about every step

- We will move slowly
  - we won’t use all the language this quarter

    will not learn every feature of the language
  - comparison with Java will be useful

Running JavaScript

- Can be run in different environments
  - command line (like Java)

    instead of "`java MyClass`", it is "`node mycode.js`"
  - inside the browser

- Primarily interesting because of the browser
  - neither language would be used much otherwise
  - command line provided so you can use one language for both

- In both environments, print output with `console.log(..)`
  - prints to command line or "Developer Console" in the browser

Programming for the Browser

- JavaScript is the lingua franca of web browsers

- Previously, other languages were tried in the browser
  - Java was used but is no longer supported
  - Flash was used but is no longer supported
  - Google’s “dart” language is still around (probably)

- Now, other languages used by compiling into JavaScript
  - will see an example of this next week
  - Java can be compiled to JS (but it’s not great)

    you can’t really get around needing to learn JS

JavaScript

History of JavaScript

- Incredibly simple language
  - created in 10 days by Brendan Eich in 1995
  - often difficult to use because it is so simple

- Features added later to fix problem areas
  - imports (ES6)
  - classes (ES6)
  - integers (ES2020)

Relationship to Java

- Initially had no relation to Java
  - picked the name because Java was popular then
  - added Java's Math library to JS also

    e.g., `Math.sqrt` is available in JS, just like Java
  - copied *some* of Java's String functions to JS string

- Both are in the "C family" of languages
  - much of the syntax is the same
  - more differences in data types

- We will discuss syntax (code) first and then data…

JavaScript Syntax

- Both are in the “C family” of languages

- Much of the syntax is the same
  - most expressions (+, -, *, /, ?:, function calls, etc.)
  - if, for, while, break, continue, return
  - comments with // or /* .. */

- Different syntax for a few things
  - declaring variables
  - declaring functions
  - equality (===)

Java vs JavaScript Syntax

- The following code is legal in <u>both</u> languages:
  - assume “`s`” and “`j`” are already declared

```
s = 0;
j = 0;
while (j < 10) {               OR for (j = 0; j < 10; j++)
  s += j;
  j++;
}

// Now s == 45
```

Differences from Java: Type Declarations

- JavaScript variables have no declared types
  - this is a problem… (we will get them back later)

- Declare variables in one of these ways:

```
const x = 1;
let y = "foo";
```

  - “`const`” cannot be changed; “`let`” can be changed
  - use “`const`” whenever possible!

Basic Data Types of JavaScript

- **JavaScript includes the following <u>runtime</u> types**

`number`	
`bigint`	
`string`	
`boolean`	
`undefined`	
`null`	(another undefined)
`Object`	
`Array`	(special subtype of Object)

Differences from Java: “`===`” operator

- **JavaScript’s “`==`” is problematic**
  - **tries to convert objects to the same type**

    e.g., `3 == "3"` and even `0 == ""` are… true?!?

- **We will use “`===`” (and “`!==`”) instead:**
  - **no type conversion will be performed**

    e.g., `3 === "3"` is false

- **Mostly same as Java**
  - **compares *values* on primitives, *references* on objects**
  - **but strings are primitive in JS (no `.equals` needed)**

    `==` on strings common source of bugs in Java

Checking Types at Run Time

Condition	Code
x is undefined	`x === undefined`
x is null	`x === null`
x is a number	`typeof x === "number"`
x is an integer	`typeof x === "bigint"`
x is a string	`typeof x === "string"`
x is an object or array (or null)	`typeof x === "object"`
x is an array	`Array.isArray(x)`

Numbers

`bigint`	integers
`number`	floating point (like Java `double`)

- **By default, JS uses `number` not `bigint`**
  - `0`**,** `1`**,** `2` **are numbers not integers**
  - **add an "`n`" at the end for integers (e.g.,** `2n`**)**

- **All the usual operators:** `+ - * / ++ -- +=` **…**
  - **division is different with `number` and `bigint`**
  - **we will prefer `bigint` because correctness is more important**

- **Math library largely copied from Java**
  - **e.g.,** `Math.sqrt` **returns the square root**

Strings

- Mostly the same as Java
  - immutable
  - string concatenation with “`+`”

- A few improvements
  - string comparison with “`===`” and “`<`”

    no need for `s.equals(t)`… just write `s === t`
  - use either `’..’` or `”..”` (single or double quotes)
  - new string literals that support variable substitution:

```
const name = "Fred";
console.log(`Hi, ${name}!`);  // prints “Hi, Fred!”
```

Boolean

- All the usual operators: `&&` `||` `!`

- "`if`" can be used with any value
  - "falsey" things: `false`, `0`, `NaN`, `""`, `null`, `undefined`
  - "truthy" things: everything else

- A common source of bugs…
  - stick to boolean values for <u>all</u> conditions

Record Types

- JavaScript “Object” is something with “fields”

- JavaScript has special syntax for creating them

```
const p = {x: 1n, y: 2n};
console.log(p.x);  // prints 1n
```

- The term “object” is potentially confusing
  - used for many things
  - I prefer it as shorthand for “mathematical object”

- Will refer to things with fields as “records”
  - normal name in programming languages

Record Types

- Quotes are optional around field names

```
const p = {x: 1n, y: 2n};
console.log(p.x);  // prints 1n

const q = {"x": 1n, "y": 2n};
console.log(q.x);  // also prints 1n
```

- Field names are literal strings, not expressions!

```
const x = "foo";
console.log({x: x});  // prints {″x″: ″foo″}
```

Record Types

- Retrieving a non-existent field returns “`undefined`”

```
const p = {x: 1n, y: 2n};
console.log(p.z);  // prints undefined
```

- Can also check for presence with “`in`”

```
console.log("x" in p);  // prints true
console.log("z" in p);  // prints false
```

- Be careful: all records have hidden properties

```
console.log("toString" in p);  // prints true!
```

Maps and Sets

- Do not try to use a record as a map!
  - usually why reason people use “`in`” and `p["name"]`

- Just use `Map` instead:

```
const M = new Map([["a", 1], ["b", 5]]);
console.log(M.get("a"));        // prints 1
console.log(M.get("a"));        // prints 5
console.log(M.get("toString")); // prints undefined

M.set("a", 2);
M.set("c", 3);
console.log(M.get("a"));        // prints 2
console.log(M.get("c"));        // prints 3
```

Maps and Sets

- **JavaScript also provides `Set`:**

```
const S = new Set(["a", "b"]);
console.log(S.has("a")); // prints true
console.log(S.has("c")); // prints false

S.add("c");
console.log(S.has("c")); // prints true
```

- **Constructor takes an (optional) list of initial values**
  - **constructor of Map takes a list of pairs**

Arrays

- Simpler syntax for literals:

```
const A = [1, 2, "foo"]; // no type restriction!
console.log(A[2]);       // prints “foo”
```

- Add and remove using push and pop:

```
A.pop();
console.log(A);  // prints [1, 2]
A.push(3);
console.log(A);  // prints [1, 2, 3]
```

Arrays

- Length field stores the length of the array

```
const A = [1, 2, "foo"];
console.log(A.length);  // prints 3
A.pop();
console.log(A.length);  // prints 2
```

- Arrays are a special type of object:

```
console.log(typeof A);  // prints ”object”

console.log(Array.isArray(A));         // prints true
console.log(Array.isArray({x: 1}));    // prints false
```

Functions

- Functions are first class objects
  - “arrow” expressions creates functions
  - store these into a variable to use it later

```
const add2 = (x, y) => x + y;
console.log(add2(1n, 2n));   // prints 3n

const add3 = (x, y, z) => {
  return x + y + z;
};
console.log(add3(1n, 2n, 3n));  // prints 6n
```

Functions

- We will declare functions like this

```
const add = (x, y) => {
  return x + y;
};

// add(2n, 3n) == 5n
```

- Functions can be passed around
  - “functional” programming language
  - but we won’t do that (much) this quarter

    we will pass functions to buttons to tell them what to do when clicked

    see CSE 341 for more on that topic

Classes

- Class syntax is similar to Java but no types:

```
class Pair {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
}

const p = new Pair(1, 2);
const q = new Pair(2, 2);
```

  – fields are not declared (because there are no types)
  – constructor is called “constructor” not class name

Classes

- **We will declare methods like this:**

```
class Pair {
  …
  distTo = (p) => {
    const dx = this.x - p.x;
    const dy = this.y - p.y;
    return Math.sqrt(dx*dx + dy*dy);
  };
}

console.log(p.distTo(q));  // prints 1
```

  - **this assignment is executed as part of the constructor**
  - **there is *another* syntax for method declarations but avoid it**

    leads to big problems when we are writing UI shortly

JavaScript Summary

- Most of the syntax is the same
  - even has `Map` and `Set` like Java

- Main difference is no <u>declared</u> types

- That means new syntax for
  - declaring variables, functions, and classes
  - checking type a runtime with `typeof`

- That means you can mix types in expressions
  - but you don't want to! avoid this!
  - use explicit type conversions (e.g. `Number(..)`) if necc.

JavaScript Summary

- A few new features that are useful…

- Strings are primitive types
  - can use "`===`" and "`<`" on them
  - simpler syntax for accessing characters: "`s[1]`"

- Integers have their own type
  - literals use an "`n`" suffix, e.g., "`3n`"
  - "`/`" is then integer division

- New syntax for string literals: `` `Hi, ${name}` ``

Modules

Imports

- Originally, all JavaScript lived in the same "*namespace*"
  - problems if two programmers use the same function name
  - tools would rename functions to avoid conflicts (e.g., webpack)

- Now, by default, declarations are hidden outside the file

- Add the keyword “export” to make it visible

```
export const MAX_NUMBER = 15;          // in src/foo.js
```

- Use the “import” statement to bring into another file

```
import { MAX_NUMBER } from './foo.js';  // in src/bar.js
```

  - ‘`./foo.js`’ is relative path from this file to `foo.js`

Imports

```
export const MAX_NUMBER = 15;              // in src/foo.js

import { MAX_NUMBER } from ‘./foo.js’;   // in src/bar.js
```

- For code you write, you will only need this syntax
- JS includes other ways of importing things
  - full explanation is very complicated
  - don’t worry about it...
- Starter code will include some that look different, e.g.:

```
import express from 'express';

import './foo.png';  // include a file along with the code
```

Put Code in Multiple Files

- Each file is a separate namespace ("module")
  - names can be shared (exported) or kept private

- Use `npm` (package manager) to enable this behavior
  - file called `package.json` contains project setup
  - scripts run node with module system enabled

```
{
  "name": "my-project",
  "type": "module",
  "scripts": {
    "exec": "node src/index.js"
  }
}
```

Packages

```
import express from 'express';
```

- **This imports from a package called "express"**
  - **use package name not a relative path (like "`./foo.js`")**

- **Use `npm` to download libraries**
  - **in `package.json`:**

```
"dependencies": {
  "express": "^4.2.1"
}
```

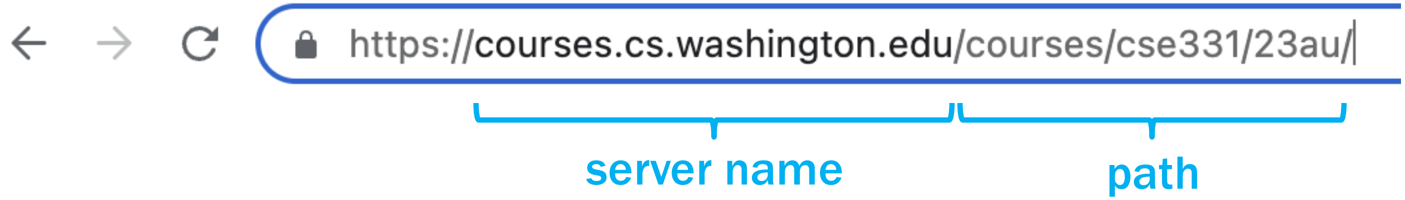
  - **second part is the version number we want to use**

    getting the wrong version can make things break, so be specific
  - **"`npm install`" downloads all libraries listed here**

HTTP Servers

Browser Operation

- Browser reads the URL to find what HTML to load

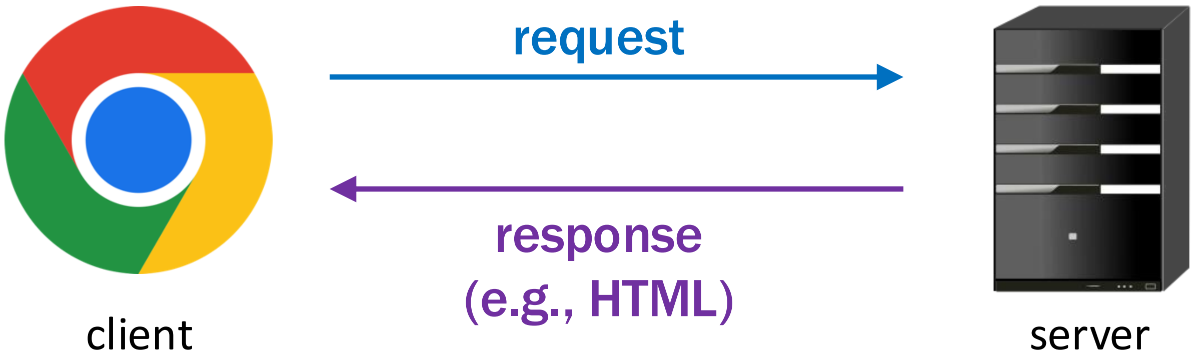


https://courses.cs.washington.edu/courses/cse331/23au/

server name

path

- Contacts the given server and asks for the given path



request

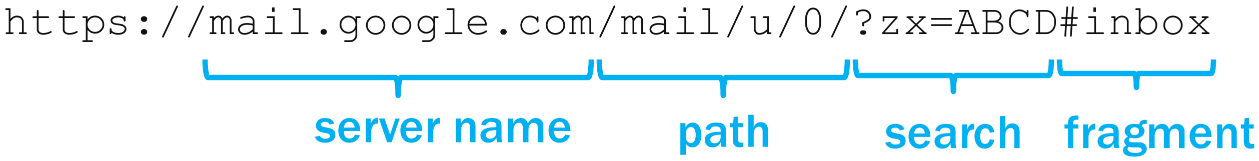
response (e.g., HTML)

client

server

URL Parts

- URLs have more parts than just server and path:



```
https://mail.google.com/mail/u/0/?zx=ABCD#inbox
```

- Server name identifies the computer to talk to
  - uses the HTTP(S) protocol

- Conceptually:
  - path identifies code to execute on the server
  - search string is input passed to that file when run
  - (fragment will not be important for us)

Query Parameters

- Search string can pass multiple values at once
  - we call these “query parameters”
- Each parameter is of the form “`name=value`”
  - no spaces around the “`=`“
- Multiple values are placed together with “`&`”s in between

```
?a=3&b=foo&c=Kevin
```

  - encodes three query parameters: a is “3”, b is “foo”, c is “Kevin”

Query Parameters

```
?a=3&b=foo&c=Kevin%20Z
```

- All values are **strings**
- Special characters (like spaces) are encoded
  - the `encodeURIComponent` function does this for us
- Will <u>not</u> need to write code to parse query params
  - have libraries that do this for us

Custom Server with Express

- Use "express" library to write a custom server:

```
const F = (req, res) => {
 …
}

const app = express();
app.get(“/foo”, F);
app.listen(8080);
```

  - request for `http://localhost:8080/foo` will call `F`
  - mapping from “`/foo`” to `F` is called a “route”
  - can have as many routes as we want (with different URLs)

HTTP Terminology

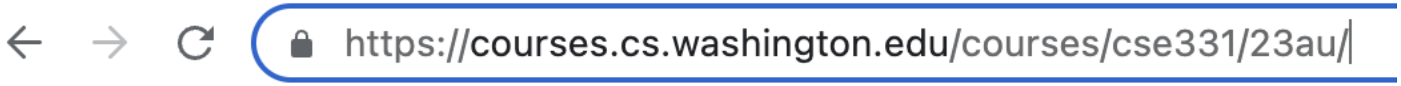
- HTTP request includes
  - **URL:** path and query parameters
  - **method:** GET or POST

    GET is used to *read* data stored on the server (cacheable)

    POST is used to *change* data stored on the server
  - **body (for POST only)**

    useful for sending large or **non-string** data with the request

- Browser issues a GET request when you type URL



https://courses.cs.washington.edu/courses/cse331/23au/

HTTP Terminology

- HTTP response includes
  - **status code**: 200 (ok), 400-99 (client error),
    or 500-99 (server error)

    was the server able to respond
  - **content type**: text/HTML or application/JSON (for us)

    what sort of data did the server send back
  - **content**

    in format described by the Content Type

- Browser expects HTML to display in the page
  - we will always send JSON or text to the browser

Custom Server

- **Query parameters (e.g., `?name=Fred`) in `req`**

```
const F = (req, res) => {
 if (req.query.name === undefined) {
   res.status(400).send(“Missing ‘name’”);
  return;
 }
  …  // name was provided
};
```

  - set status to 400 to indicate a client error (Bad Request)
  - set status to 500 to indicate a server error
  - default status is 200 (OK)

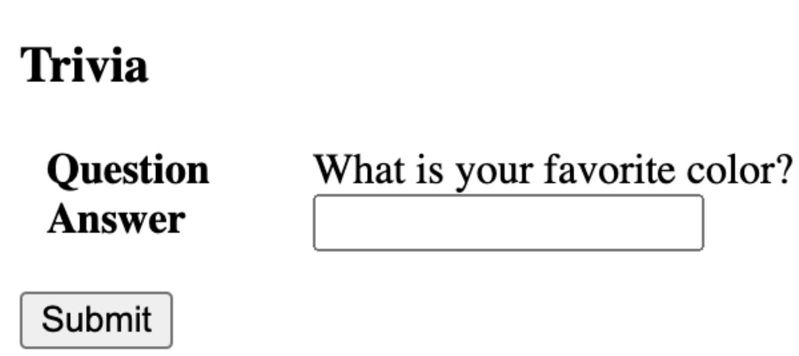
Custom Server

- **Query parameters (e.g.,** `?name=Fred`**) in** `req`

```
const F = (req, res) => {
  if (req.query.name === undefined) {
      res.status(400).send(“Missing ‘name’”);
    return;
  }
  res.send(`Hi, ${req.query.name}`);  // sent as text
};
```

- **Content type will be set automatically:**
  - `send` **of string returned as text/HTML**
  - `send` **of record returned as application/JSON**
  - **use this coding convention rather than explicit content type**

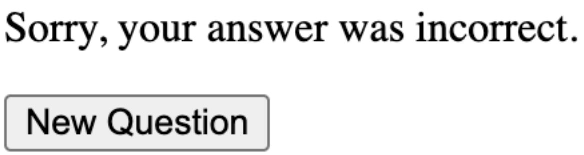
Example App



Trivia

Question	What is your favorite color?
Answer	

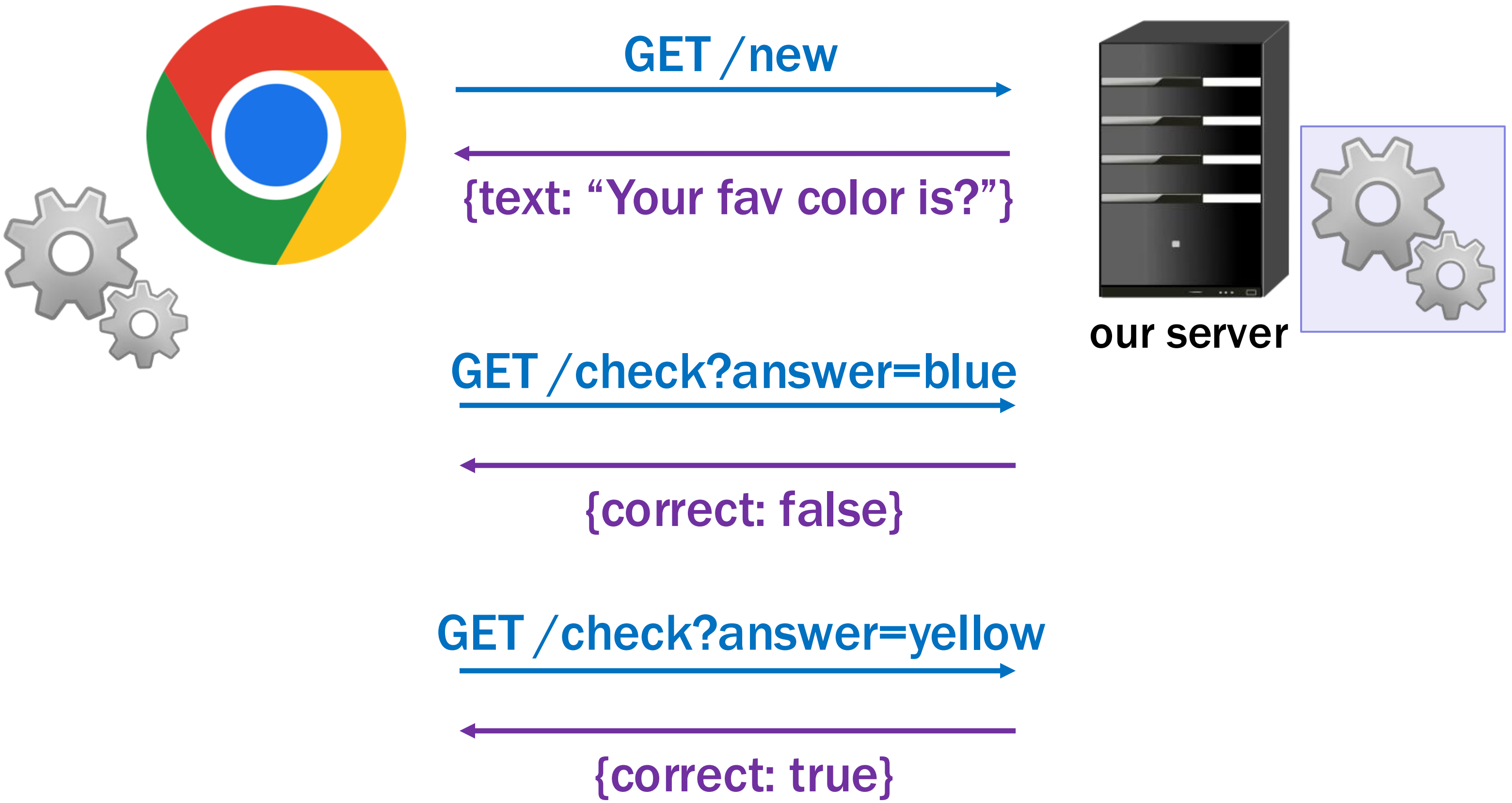
User types “blue” and presses “Submit”…



Sorry, your answer was incorrect.

Example App

- Apps will make sequence of requests to server



GET /new

{text: “Your fav color is?”}

our server

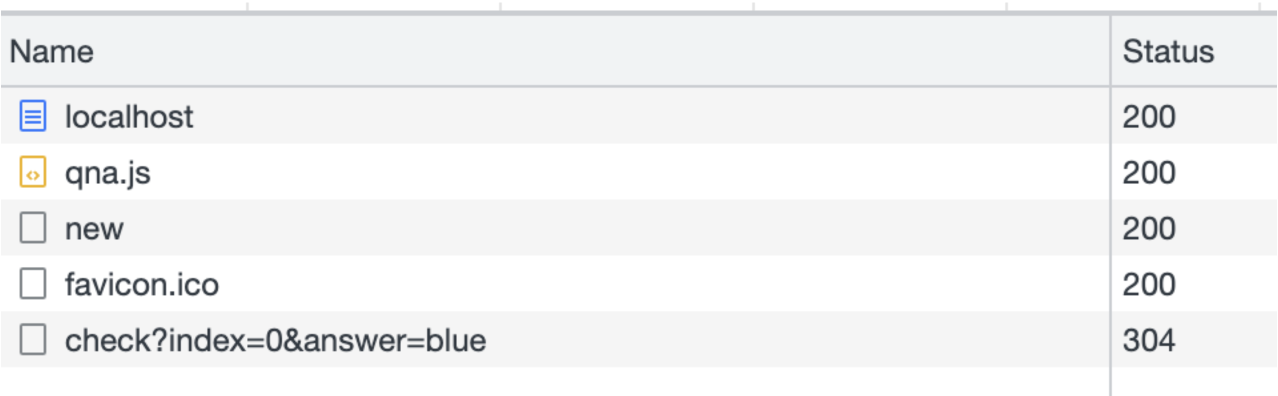
GET /check?answer=blue

{correct: false}

GET /check?answer=yellow

{correct: true}

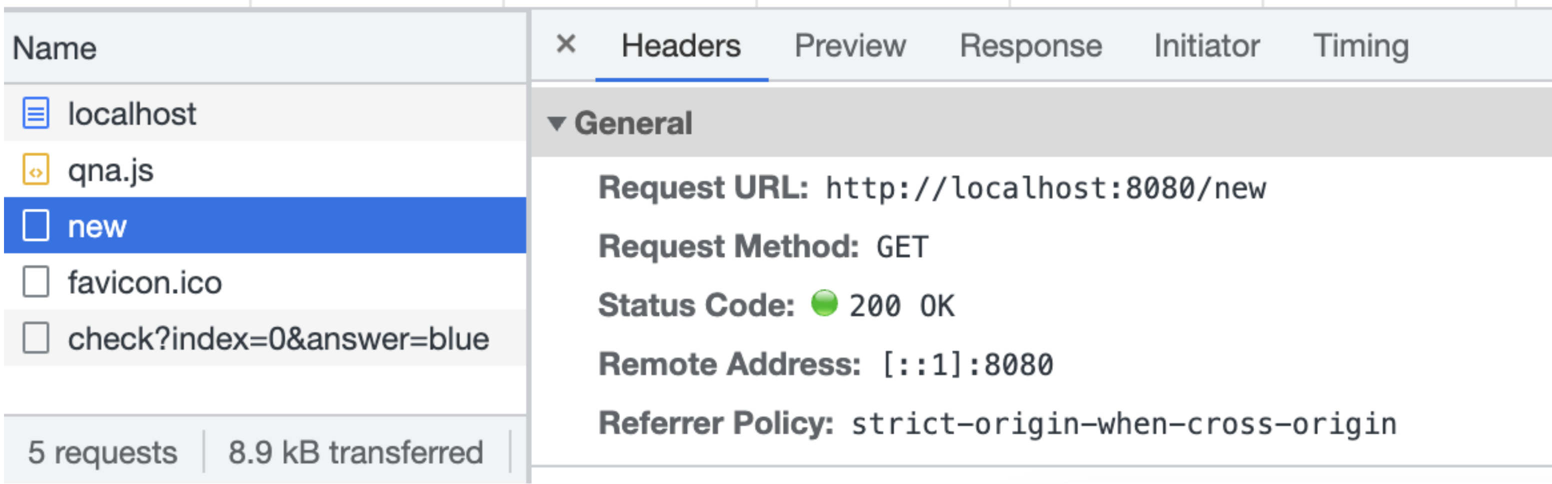
“Network” Tab Shows Requests



Name	Status
localhost	200
qna.js	200
new	200
favicon.ico	200
check?index=0&answer=blue	304

- Shows every request to the server
  - first request loads the app (as usual)
  - “`new`” is a request to get a question
  - “`check?index=0&answer=blue`” is a request to check answer
- Click on a request to see details…

“Network” Tab Shows Request & Response



Name

- localhost
- qna.js
- new
- favicon.ico
- check?index=0&answer=blue

5 requests | 8.9 kB transferred

▼ General

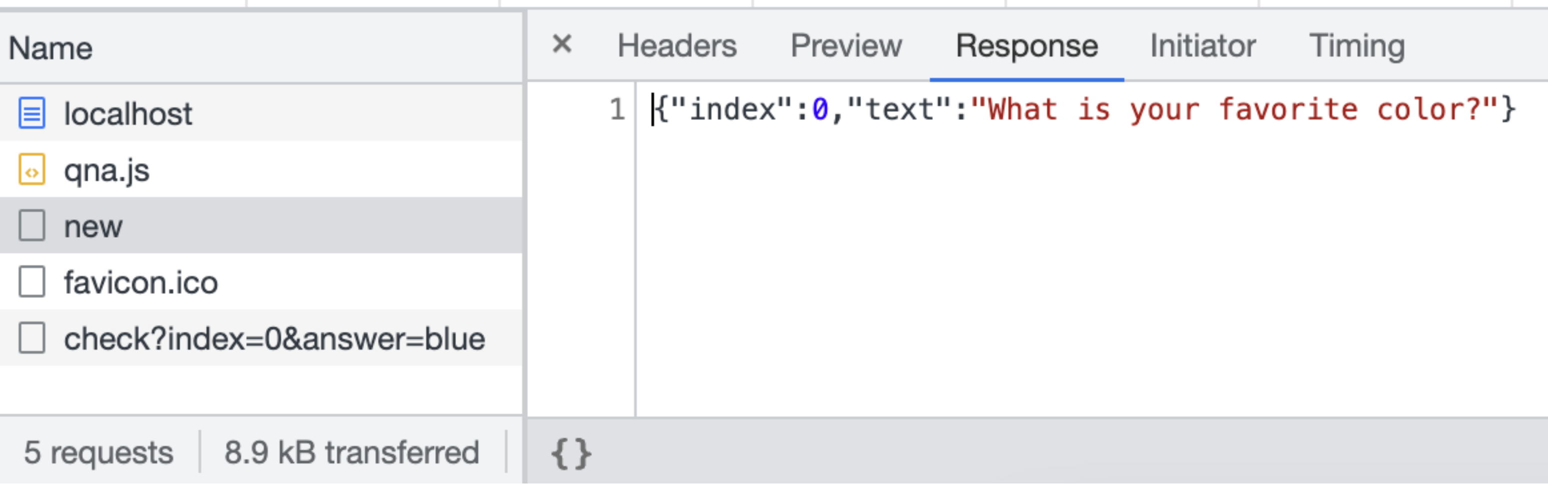
Request URL: `http://localhost:8080/new`

Request Method: `GET`

Status Code: ● `200 OK`

Remote Address: `[::1]:8080`

Referrer Policy: `strict-origin-when-cross-origin`



Name

- localhost
- qna.js
- new
- favicon.ico
- check?index=0&answer=blue

5 requests | 8.9 kB transferred

```
{"index":0,"text":"What is your favorite color?"}
```

Summary of Last Time

- Split code into multiple files with `import` & `export`
  - requires using `npm` to call `node` for us
    - node normally run all code in a single namespace
- NPM also allows us to use existing packages
  - will download them for us and let us import then
  - example: "express" is a library for writing HTTP servers
- Wrote our first HTTP server
  - GET requests take input in `req.query` (record of strings)
  - POST requests take input in `req.body` (record of anything)

JSON

- **JavaScript Object Notation**
  - **text description of JavaScript object**
  - **allows strings, numbers, null, arrays, and records**

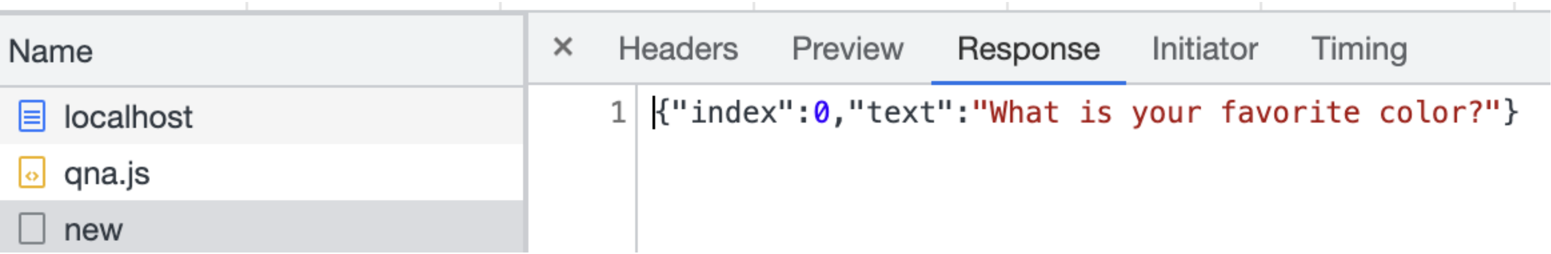
    no undefined and no instances of classes

    no ‘..’ (single quotes), only “..”

    requires quotes around keys in records

- **Translation into string done *automatically* by send**

```
res.send({index: 0, text: ’What is your …?’});
```



```
{"index":0,"text":"What is your favorite color?"}
```

POST Body

- Sent in request as JSON
  - parsed into a JS object by express library

- POST body available in `req.body`
  - e.g., if POST body is `{"a": 3, "b": 5}`

```
const getAvg = (req, res) => {
  const avg = (req.body.a + req.body.b) / 2;
  res.send({avg: avg});  // sent as JSON
};
```

  - note that `req.body.a` is a <u>number</u>, not a string

Servers

```
app.get(“/foo”, F);
app.listen(8080);
```

- Program does **not** exit at the end of the file
  - call to listen tells it to run forever
  - runs until forcibly stopped (Ctrl-C)
- Does work only when request "events" occur
  - called "event-driven" programs
- This is how most real-world programs work
  - client applications wait for user interaction
  - servers wait for new requests from clients

Debugging Event-Driven Programs

- When command-line program fails…
  - know the exact inputs that caused it
  - can re-run it over and over until you understand the cause
- When event-driven program fails…
  - might know the *last* event that occurred (e.g., that request)
  - don't know the full sequence of events
  - don't know the state of all the variables in the program
  - usually unclear how to reproduce the failure
- Debugging real-world programs is <u>hard</u>
  - in some settings, it is nearly impossible