Debugging

All programming involves debugging

- An art, not science (yet)
- But NOT random!

Much of this advice is true for all programming

- Not just web

Poor Debugging tactics

- Stare at it
- Changing things until it works
- Asking for help because "it doesn't work"

Debugging goal

What line isn't working as intended

- Most of the time the problem is identified
- Rest of the time you can get real help
- Avoids chaos in the rest of your code

Good Tactics

- Verify when inputs are good
    - Avoid Garbage In-Garbage Out (GIGO)
- Simplify the problem
    - Narrow the scope
    - Reduce the complexity of data
- Comment out code
    - Shows it is not part of the problem
- Rubber Duck Debugging

All towards finding that **one line of code**

Write and confirm in small steps

Avoid writing a lot code

- Write small chunks
- Confirm those chunks work before writing more

Keep code RUNNABLE

- Don't wait on everything being done

It worked 1 minute ago

- Code from the last minute has a bug
    - Probably only 1, there isn't much code
    - Easier to find
    - Easier to confirm the bug is fixed

It worked 3 hours ago (last test)

- Code from the last 3 hours has a bug
    - Possibly MANY
    - Bugs are really common
    - Hard to find
        - a lot of code
    - Hard to confirm a fix
        - another bug could be there

Seniors+ write bugs all the time!

Senior/Staff/Principal/Architects still write bugs

Seniors+ are just better at finding/fixing

- We seen it before
- Don't get upset you make bugs
- Get good at finding them
    - Finding does help to avoid future ones too

Bugs aren't fixed randomly

- They are communication errors
- We fix them with correct language and logic

Using `console` to debug

- Perfectly valid technique!

Goal: Find that **one line of code**

- Identify when values are expected
    - and when not
- Identify if code executes

I heard this expression

(paraphrased because terrible memory)

As a poor developer, I debugged using the console
As a good developer, I debugged using the debugger
As a great developer, I debugged using the console

Use the right tool, there is no BEST tool

- console is great for checking values
    - across multiple iterations

Clarity

When logging out a value

- important to know WHAT value represents
- and WHEN

Getting several `undefined` doesn't tell you much

Clean up

If a console statement isn't needed anymore

- remove it immediately

Don't want to lose the valuable outputs in the noise

All console statements and commented out code removed before submit!

- Clean as you go

Console options

`.log()` is not the only option on `console`

- `.dir()`
- `.table()`
- `.assert()`
- `.info()`, `.warn()`, `.error()`
- `.count()`
- `.timeLog()`, `.time()`, `.timeEnd()`

See MDN for info

Personal Tips

- Use numbers if just checking "did I get here"
    - Handles multiple checks quickly
    - No quoting
- Create shorthand objects to label output
    - instead of `console.log(word);`
    - use `console.log({word});`
    - labels output variable
    - Minimal typing, no quoting

Remember WHY you are using console

Find that one line of code

- Is the bug before or after this line of code?
- Am I passing garbage into a function or good data?
- Did I get garbage or expected data?

A Debugger

An interactive display of code

- Allows you to pause execution
    - Inspect variable values
    - Even modify them
- Can pause on certain situations
- Can move forward in execution
    - Big or Small steps

Debugger

Pro:

- Granular control over flow
- Don't need to type out console.log in advance
    - Can avoid a lot of repeated checks
- No console.log() clean up

Con:

- Have to go through all the code
- Can be heavier to connect
- Can lose track of where you are in frameworks

How to get a Debugger for Node JS

- Minimal command-line debugger in node
    - I avoid
- Connect to Debugger in IDE
    - Example: VS Code extension
- Connect via Chrome Debugger
    - Also used once we get to browser JS

For all, need to tell Node to expose debugging

- Only do during development

```
node --inspect server.js
```

Connecting to browser debugger

- Run node process with `--inspect` flag
- Load `chrome://inspect` in browser
- Click on 'inspect' next to file name
    - Opens new window
    - Be in "Sources" Tab of that window
- Set any breakpoints
    - Make any requests
- Fight with having multiple windows

Breakpoints

Pause the code at **breakpoints**

- Click on line number to set/unset
- System may use a different line
    - if statement spans lines
- Function defined and function called are different
- See **Scope** for list of variables in scope
    - You can **Watch** variables

Interactive Execution

- Press "Play" button to resume execution
  - Until next breakpoint (if any)
- "Step Over", "Step Into", etc statements
  - Go one statement at a time
- Hold down Play to play and skip breakpoints

Automated Unit Tests

This is a huge and vital subject

- But we don't have room!
- Could be a class by itself
- Expected by all coding jobs
    - web / not-web
    - frontend / backend
- Will give a VERY BRIEF intro

Testing "Units"

A "unit" is a piece of code with a purpose

- Usually a function, object, or module

We test the input/output

- We want no outside interactions
    - No "side-effects"
- No browser
- No database
- No server (!)
    - test require() code, but not server.js
- Just input/output

Simple test

We ran `compare()` with some sample input?

- Automating that would be something like

```
let isPassing = true;
if( compare('BOO', 'FOO') !== 2 ) {
  isPassing = false;
}
if( compare('GEESE', 'FREED') !== 2 ) {
  isPassing = false;
}

if( !isPassing ) {
  console.error('Testing Failed!');
}
```

More Automation

That would only work for this specific function

- And doesn't provide much info about what failed
- We could improve the interface to the test
- And improve the output when it failed
- And make it more generic so we could test other code

People have done that

An example unit test

```
const compare = require("./compare");

describe("compare", () => {
  it("counts the letters of exact matches", () => {
    expect( compare("EAT", "EAT") ).toBe(3);
    expect( compare("GEESE", "GEESE") ).toBe(5);
  });

  it("counts the letters of anagrams", () => {
    expect( compare("EAT", "TEA") ).toBe(3);
  });

  it("Ignores case", () => {
    expect( compare("EAT", "tea") ).toBe(3);
  });

  //....
});
```

Compare Unit Test Results

After `npm install --save-dev jest`

```
swiftone@shiny:$ jest compare.test.js
 PASS  ./compare.test.js
  compare
    ✓ counts the letters of exact matches (2 ms)
    ✓ counts the letters of anagrams
    ✓ Ignores case (1 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        0.296 s, estimated 1 s
Ran all test suites matching /compare.test.js/i
```

Why Unit Tests?

- Fast!
    - Much faster than manual testing
    - Easy to run after changes
    - Computer won't forget to test
- Confirms behavior of code
    - Outside the complexity of context
- Can find bugs BEFORE app has them
- Documents the unit at the I/O level!

Additional Automated Tests

- Integration and End-to-End/UI tests exist
    - Tend to be fewer of them
    - They "break" more often
        - Means work to fix test, not to fix code
- More on this we find room to squeeze it in
- No required automated tests yet
    - You are welcome to include it