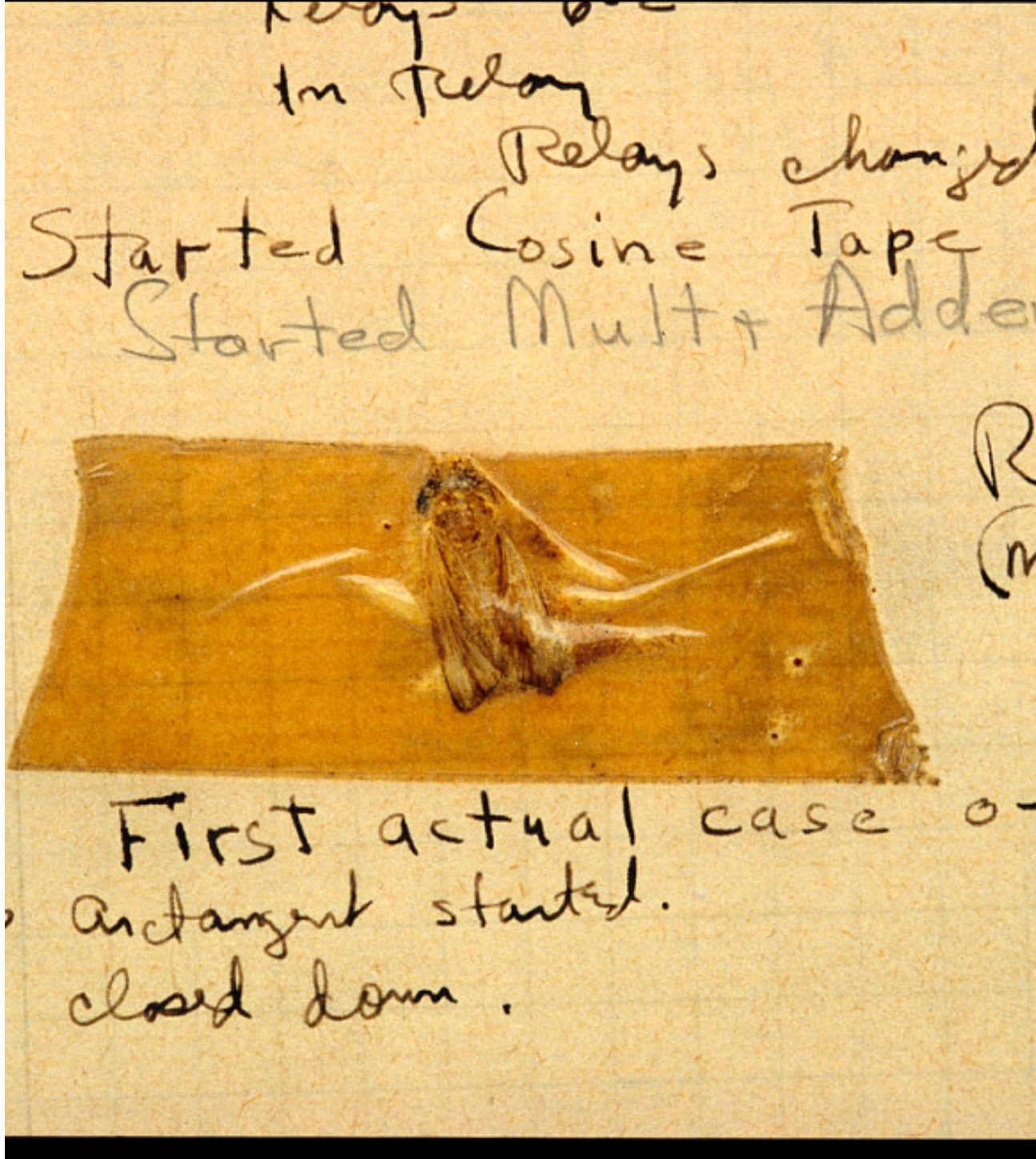
Debugging in JavaScript

What is a bug?

A bug is an unintended consequence of running code.

Debugging is the process of finding and fixing bugs in code.

The first bug![1] ->



[1] The Smithsonian National Museum of American History (https://americanhistory.si.edu/collections/search/object/nmah_334663)

Categories of bugs

— *Syntax error*: a programming language operates according to certain structural rules. If you violate what the interpreter expects (e.g., you omit a parenthesis or you put a character in the wrong place), your program won't run.
— *Runtime error*: Also called exceptions. The program runs and the syntax is correct, but encounters something unexpected and therefore bad -- something that was unforeseen and not accounted for by the programmer (e.g., you call a function that does not exist).
— *Logic error*: This is the hardest one to debug, because it does not produce an actual error. The outcome is just not what you had intended: it is correct in terms of programming, but it doesn't do what you want it to do.

How to think about debugging

- Isolate the problem so that you can focus on one thing at a time.
- Be *curious* about what is happening!
- Be *patient*. Take your time to understand, and to *learn from what is going on*.
- Maintain an open mind and *don't make any assumptions*.
- Be clear on what you expect to happen/to be true, and compare it to what actually does happen.

1. Identify the bug. What exactly is happening?

We are trying to get to the source of an issue. To do that, we need to isolate the problem. We need to deal with one problem at at time -- the issue we are seeing might be caused by more than one thing.

- What is happening, exactly? Not the cause, just what you are observing.
- When does it happen?
  - What happens right before the problematic behavior or error? What seems to trigger it?
- *Try to reproduce the bug* and observe it in action until you begin to understand what is going on.
  - Observe the behavior repeatedly, trying to gather information each time.
  - Can you cause it to happen by doing something?

2. Find the source of the bug. What is the cause?

Somewhere in your code is a line that is causing this issue.

- If you have an error message, start at the line that it is pointing to.
- If you do not have an error message, find the part of your code that you think is relevant.
- Read though each line of code and verify your assumptions about what is happening by testing parts of the code in the console. Isolate as much as you can.
  - Take one tiny piece of code that can be executed -- for instance, not the whole `if` statement at once, but the predicate/condition (the code in the parentheses after `if`) only.
  - Run that in the console. Does that give you what you expect? If so, move on to the next thing. If not, then YAY 🎉that's at least not part of your bug!
  - Confirm your expectations at each step by testing in the console.
- Be methodical. Do not jump around or give up. Follow one line of inquiry though to the end.
- Verify *even the thing that you are 100% sure is right*.

3. Fix the bug. Only change ONE thing at a time.

A common beginner mistake is to change a number of things all at once and then reload the browser. If you change more than one thing at a time, you won't know for certain what change you made that may have caused any differences in behavior.

Make one change that you think may fix things. Test it in the console. Then rerun through the whole scenario in which the bug occurred. Try to make the bug happen again, more than once.

- If you make a change and the original bug doesn't happen again...
  - Are you getting a different error or other problems? Take a look at that; you may need to revert the change you made, or you may not.
  - If everything looks good, test it several more times. Still good? 🎉You fixed it! COMMIT THAT!
- If the same error does happen again, revert the change and resume your methodical checking.

Linters

VS Code and other editors can automatically check your code for errors or other potential problems. The way linters work is that you have the basic functionality and then you add a set of linting rules that you choose. These rules are configurable.

— StandardJS
— StandardJS extension for VS Code

The linter is what highlights something in your code with a squiggly line or some other indicator and warns you about a potential problem.

Linters can be really helpful, but can also get in the way if they are overly sensitive. You still have to make the judgment about whether something is ok or not, by manually checking what is going on.

Other VS Code tools that can help you

— Bracket Pair Colorizer extension
— indent-rainbow extension
— Path Intellisense
— webhint

The `console`

Errors that happen when the JavaScript runs are displayed in the console.

— type of error
— message (optional part of the error; meant to be human-readable)
— the file and line number where the error occurred

`console.log()` prints information to the browser's debugging console

Breakpoints & Chrome Dev Tools

You can set a *breakpoint* to force your program to stop running at a certain place in your code, so that you can examine variables and values at that point in the program.

- Set a breakpoint
  - in Chrome DevTools on Sources panel
  - You can also use the `debugger` keyword in your js file
- Step through your code line by line (use arrow icons)
- `esc` key to open a console that you can type in
- Play button will resume execution

Some common bugs in JS

Look out for these!

- Using an assignment operator = instead of a comparison operator ===
- Comparing mismatched data types (using == instead of ===)
- Missing brackets and parentheses, or a misplaced return statement
- Unescaped quotes in a string (closing the string too early)
- Typos, misspellings, capitalization errors
- Not returning a value; returning the wrong value; returning too early
- Mistakenly using a variable or function not in scope

TypeError

A value is not the type that the JS interpreter expected it to be, based on what you tried to do

- TypeError: cannot read property <something> of undefined
- TypeError: cannot read property <something> of null
- TypeError: <something> is not a function

```
let arrayOfNumbers
arrayOfNumbers.length
// Uncaught TypeError: Cannot read property 'length' of undefined
```

How to fix it

— You may have misspelled something
— You may not have defined a variable that you thought you did (check it in the console)
— Check your types. For instance, are you calling an array method on something that is not an array?

SyntaxError

You have invalid syntax that stops execution. It's often a typo!

```
if 2 > 0 {
  console.log('this is true!')
}
// SyntaxError: Unexpected number
```

☝️ in this example, we have forgotten the parentheses around the condition after the `if` keyword

How to fix it

Look for misspellings, unclosed parentheses, or extra or missing characters.

ReferenceError

`Reference Error: [something] is not defined`

JS evaluated something that looked like a variable or function name, but it did not find it

How to fix it

— Check for misspellings or typos.
— Look for where the variable is defined. Is it missing? Is it defined somewhere else? Make sure that you have access to a function or variable in the scope where you use it.

Exceptions

Exceptions are errors that developers create on purpose. It's another way to control the flow of a program.

We can *throw* or *raise* an exception in order to handle specific cases where we anticipate something might go wrong.

- An exception will stop the execution of the program
  - It either ends the program or passes execution to the first catch block it encounters
  - In that catch block, we can run code!
  - We might log information about what happened, or call another function, or whatever we want.

try...catch

If we anticipate that an error or exception might be thrown, we can set up our code to catch that error.

```
try {
  // code that might result in an error or exception
} catch (error) {
  // what you want to do when an error happens
}
```

Handling the error prevents it from stopping execution of the rest of the program!

try...catch example

```
// This function is designed to throw an error in certain cases:
function calculateDiscount(percentage) {
  if (percentage) {
    console.log("Carry on!")
  } else {
    // the throw keyword is used to raise an exception
    throw new Error("Missing percentage to calculate the discount");
  }
}

// Here is a function that calls the above function within a try block
function applyDiscount(discountPercent) {
  try {
    calculateDiscount(discountPercent)
  } catch (error) {
    console.log(error)
    doSomethingElseInstead()
  }
}
```

try...catch...finally

You can also add a `finally` block to ensure code that always executes (if you do, you won't be able to return values from the try or catch blocks).

```
function doStuff(arg) {
  try {
    calculateDiscount(arg)
  } catch (error) {
    console.log(error)
    doSomethingElseInstead()
  } finally {
    console.log("this code always runs no matter what!")
    doTheThingWeAlwaysWantToDo()
  }
}
```