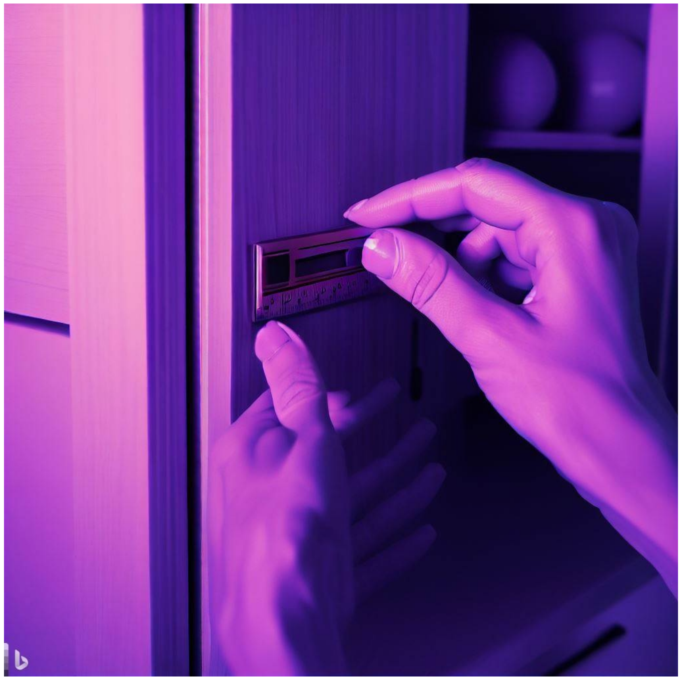


# CSE 331

## Correctness

Kevin Zatloukal

# Recall: Shipping Software

- Building shippable version is ~10x harder than demo
  - demo version needs to work when *used properly*
  - shipped version needs to work properly no matter what

- 1m users will try millions of cases that you didn’t
  - needs to work properly on all cases, even ones you didn’t try

- How is this achieved in practice?

# Standard Techniques for Correctness

Standard practice uses three techniques:

- **Testing**: try it on a well-chosen set of examples
- **Tools**: type checker, libraries, etc.
- **Reasoning**: think through your code carefully
  - have another person do the same ("code review")

Each removes ~2/3rd bugs but of different kinds

Combination removes >97% of bugs

# Which Ones and How Much

- The first question to ask yourself:

  How much of each is needed for my program?

- Correctness is easier for some programs vs others

- Personally, I break this into 5 cases…
  - “levels” of difficulty

    warning: I made this terminology up

# Correctness Levels

| Level | Description | Testing | Tools | Reasoning |
|---|---|---|---|---|
| -1 | small # of inputs | exhaustive | | |
| 0 | ? | | | |
| 1 | ? | | | |
| 2 | ? | | | |
| 3 | ? | | | |

# Level -1

- Small number of inputs / configurations

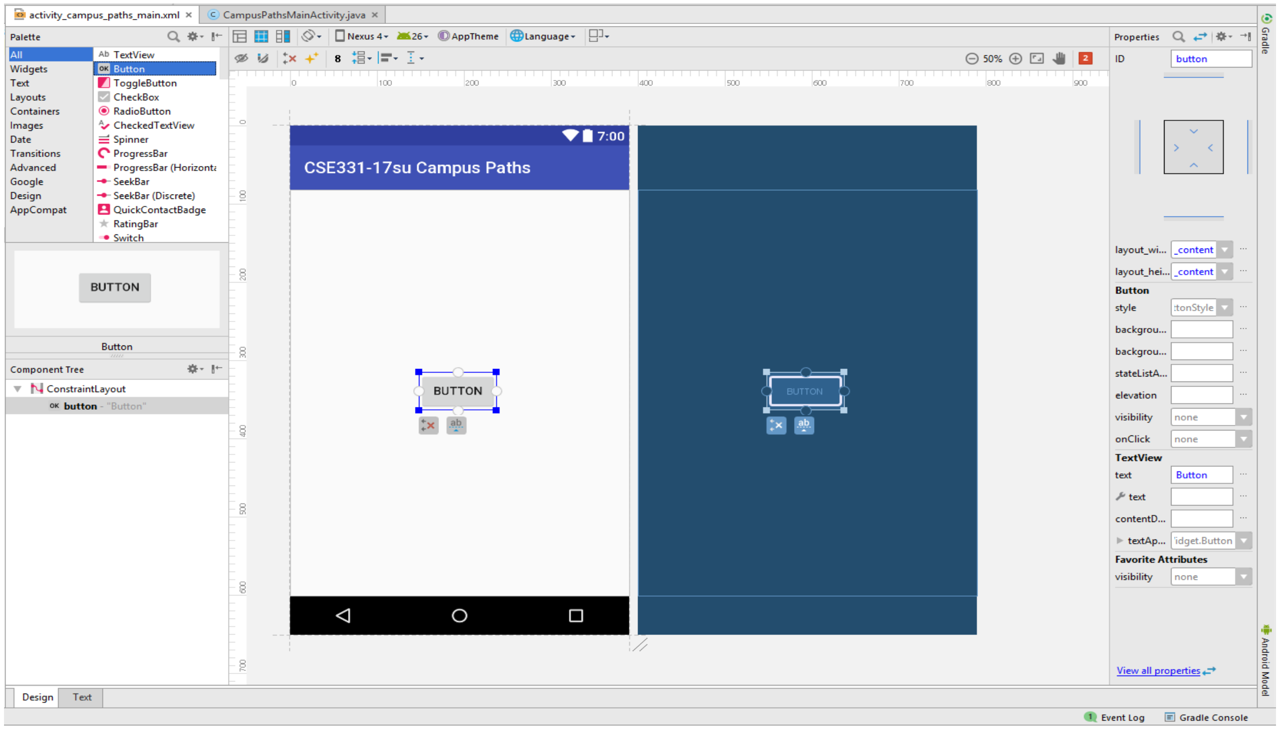
- Just check them all!
  - this is the right answer

- This category does not require a programmer
  - anyone can check the answer
  - programming is hard, so skip it when you can

# Level -1

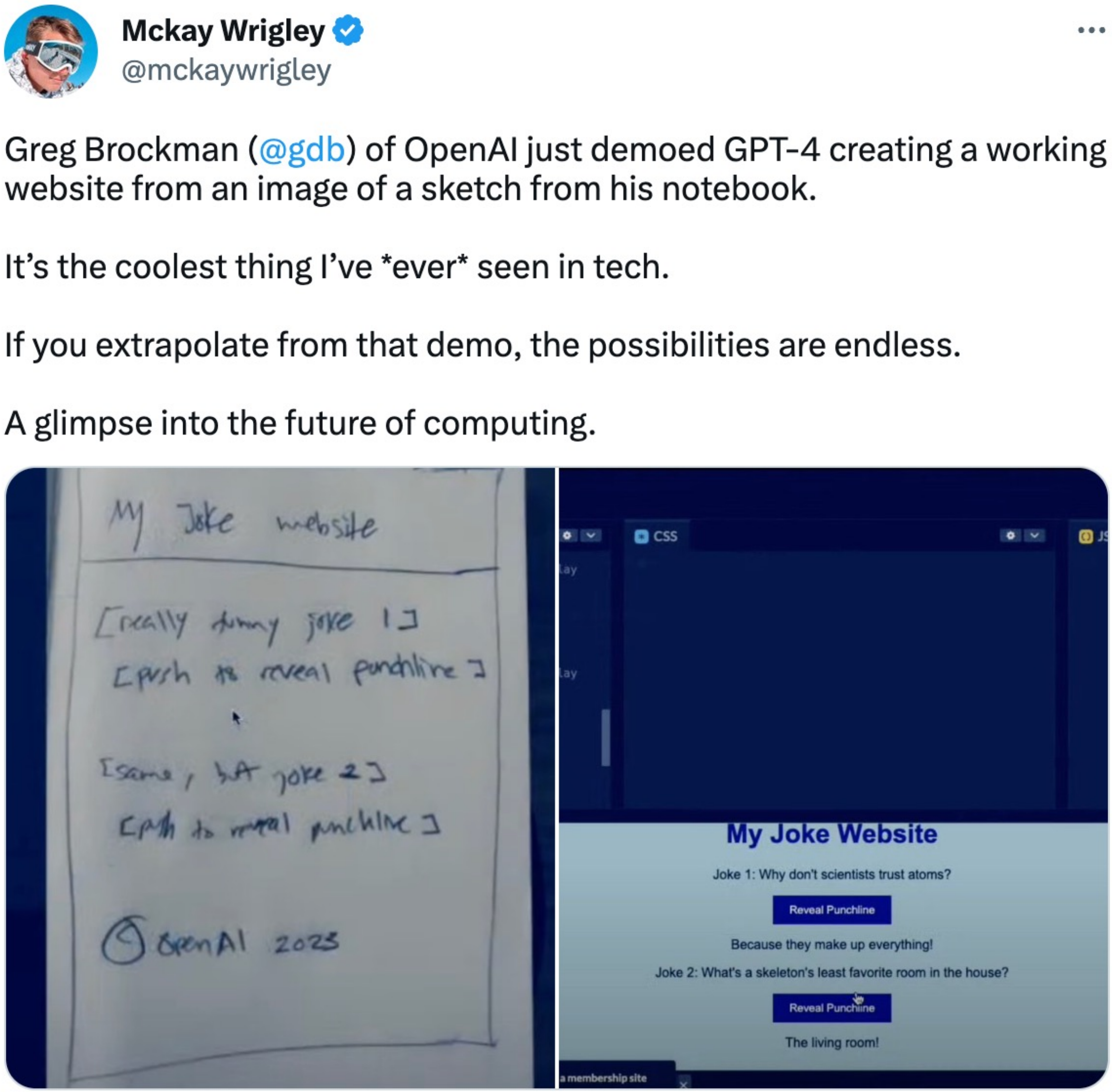
- Coding is the wrong tool for this job
  - can happen in part of a larger application

- iPhone development lets you draw the UI:



ID: button
layout_wi...: _content
layout_hei...: _content
style: :tonStyle
visibility: none
onClick: none
text: Button
textAp...: idget.Button
visibility: none

# Level -1



**Mckay Wrigley**
@mckaywrigley

Greg Brockman (@gdb) of OpenAI just demoed GPT-4 creating a working website from an image of a sketch from his notebook.

It’s the coolest thing I’ve *ever* seen in tech.

If you extrapolate from that demo, the possibilities are endless.

A glimpse into the future of computing.

My Joke website

[really funny joke 1]
[push to reveal punchline]

[same, but joke 2]
[push to reveal punchline]

© OpenAI 2023

My Joke Website

Joke 1: Why don't scientists trust atoms?

Reveal Punchline

Because they make up everything!

Joke 2: What's a skeleton's least favorite room in the house?

Reveal Punchline

The living room!

# Level -1

- Can happen as part of a larger application
  - may require code but not reasoning

- Happens more often than you might think
  - individual function can be level -1
    - e.g., two boolean inputs (only 4 configurations)
  - quite common with UI
    - e.g., when I click the button, it should say “hi”

- Be on the lookout for these cases
  - save yourself work by spotting them

# Correctness Levels

| Level | Description | Testing | Tools | Reasoning |
|---|---|---|---|---|
| -1 | small # of inputs | exhaustive | | |
| 0 | ? | | | |
| 1 | ? | | | |
| 2 | ? | | | |
| 3 | ? | | | |

# Correctness Levels

| Level | Description | Testing | Tools | Reasoning |
|---|---|---|---|---|
| -1 | small # of inputs | exhaustive | | |
| 0 | straight from spec | heuristics | | |
| 1 | no mutation | “ | | |
| 2 | local variable mutation | “ | | |
| 3 | array / object mutation | “ | | |

# Correctness Levels

| Level | Description | Testing | Tools | Reasoning |
|---|---|---|---|---|
| -1 | small # of inputs | exhaustive | | |
| 0 | straight from spec | heuristics | type checking | |
| 1 | no mutation | “ | libraries | |
| 2 | local variable mutation | “ | “ | |
| 3 | array / object mutation | “ | “ | |

# Correctness Levels

| Level | Description | Testing | Tools | Reasoning |
|---|---|---|---|---|
| -1 | small # of inputs | exhaustive | | |
| 0 | straight from spec | heuristics | type checking | code reviews |
| 1 | no mutation | “ | libraries | calculation<br>induction |
| 2 | local variable mutation | “ | “ | Floyd logic |
| 3 | array / object mutation | “ | “ | rep invariants |

# Reminders

- We will set an **extremely high bar** for correctness
- Now is the time to practice proper technique
  - much harder to learn technique on harder problems
- Reasoning is not optional
  - “either reason now or debug **and then** reason”
  - debugging can be **painful**

# Specifications

# Specifications

- Correctness requires a description of the correct answer
  - true at any level of correctness
- Description must be <u>precise</u>
  - can't have disagreement about what is correct
- Informal descriptions (English) are usually imprecise
  - necessary to "formalize" the English
    - turn the English into a precise *mathematical* definition
  - professionals are *extremely* good at this
    - usually just give English definitions
  - important skill to practice

# Kinds of Specifications

- **Imperative** specification says <u>how</u> to calculate the answer
  - lays out the exact steps to perform to get the answer

- **Declarative** specification says <u>what</u> the answer looks like
  - does not say how to calculate it
  - future: prove our calculation meets the spec

- Can implement a *different* imperative specification
  - future: prove ours is equivalent to the original specification

# Example: Imperative Specification

- Absolute value $|x| = x$ if $x \geq 0$ and $-x$ otherwise
  - definition is an “if” statement

```
function abs(x: number): number {
  if (x >= 0) {
    return x;
  } else {
    return -x;
  }
}
```

just translating math to TypeScript

Level 0

# Example: Declarative Specification

- Absolute value $|x|$ is a number $y$ such that
  - $y \geq x$
  - $y \geq -x$
  - $y = x$ or $y = -x$

```
function abs(x: number): number {
  if (x >= 0) {
    return x;
  } else {
    return -x;
  }
}
```

requires some thinking to make sure this code returns a number with the properties above

**Level 1+**

(in fact, Level 1)

# Correctness Levels

| Level | Description | Testing | Tools | Reasoning |
|---|---|---|---|---|
| -1 | small # of inputs | exhaustive | | |
| 0 | direct from spec | ? | ? | ? |
| 1 | ? | | | |
| 2 | ? | | | |
| 3 | ? | | | |

# Level 0

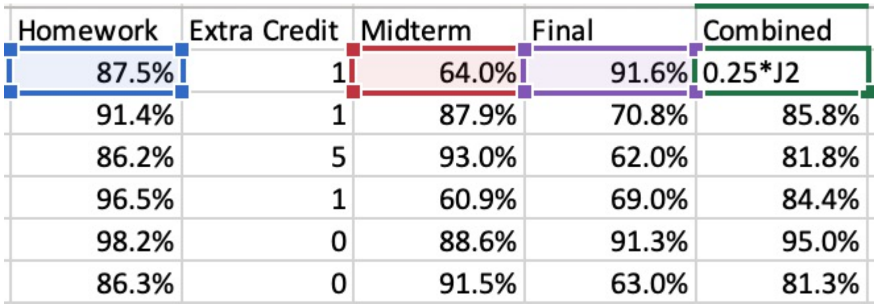
---

- Instructions say exactly how to calculate answer
  - given an imperative specification
  - we are just translating math into code
- Still easy to make mistakes!
  - too many inputs to test them all
  - need to additional ways of checking for bugs
- Still important to get it right!

# Non-programming Example

- Important to calculate grades correctly!

fx =0.6*G4+0.15*I4+0.25*J4



| Homework | Extra Credit | Midterm | Final | Combined |
|---|---|---|---|---|
| 87.5% | 1 | 64.0% | 91.6% | 0.25*J2 |
| 91.4% | 1 | 87.9% | 70.8% | 85.8% |
| 86.2% | 5 | 93.0% | 62.0% | 81.8% |
| 96.5% | 1 | 60.9% | 69.0% | 84.4% |
| 98.2% | 0 | 88.6% | 91.3% | 95.0% |
| 86.3% | 0 | 91.5% | 63.0% | 81.3% |

- The syllabus says the formula
  - ask someone else to double-check ("code review")
  - spot check some of them

# Correctness Levels

| Level | Description | Testing | Tools | Reasoning |
|---|---|---|---|---|
| -1 | small # of inputs | exhaustive | | |
| 0 | direct from spec | heuristics | type checking | code reviews |
| 1 | ? | | | |
| 2 | ? | | | |
| 3 | ? | | | |

# Correctness at Level 0

**Correctness at Level 0 requires these elements:**

- **Code review**
  - **second set of eyes**
- **Type checker**
  - **third set of eyes (so to speak)**

    (tends to find *different* mistakes than human reviewers)
- **Good set of tests**
  - **can’t test every case… need to pick the the right ones**

    (more on this next lecture…)

# Type Checkers

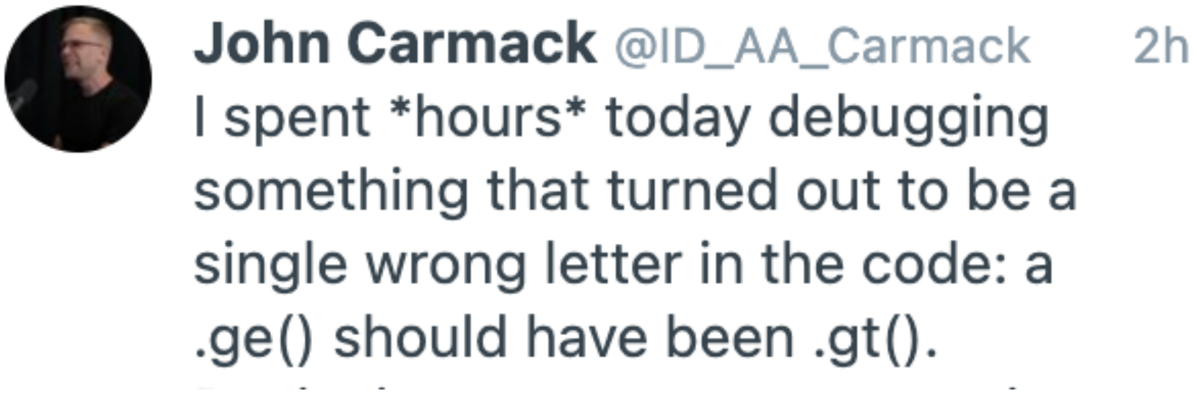
- The main part of “Tools” is the type checker
  - libraries are the other important part

- Type Checkers are very useful for finding bugs
  - another set of “eyes” helping us find them
  - you have probably learned this already

# Type Checkers

- **TypeScript and Java have different type systems**
  - **they can catch different bugs for us**
    - TypeScript ensures references are not null (Java does not)
    - Java ensures that numbers are integers (TypeScript does not)
    - (more examples coming soon…)

- **Critical to understand what the tools will miss**
  - **can ignore issues the tools would catch**
  - **must carefully think about issues the tools would miss**



**John Carmack** @ID_AA_Carmack 2h

I spent *hours* today debugging something that turned out to be a single wrong letter in the code: a .ge() should have been .gt().

# How-To For Level 0

- Level 0 = “direct from spec”
  - translate math into our programming language
  - TypeScript here, but could also be Java

- Rest of this lecture:
  - define math for data and code
  - describe how to translate those into TypeScript

    try to make the translations as *straightforward* as possible (fewer mistakes)
  - mention new TypeScript features when related

# Math Notation

# Math Notation

- Define a language for clear, precise specifications

- Will use a very small math toolkit
  - almost all of it describable in one lecture

    most of it today, but one key tool coming later
  - full description is just 3 pages

- Split this into two parts: data and code
  - data types: our math for **data**
  - functions: our math for **code**

    (can’t talk about code until we describe input and output types)

# Basic Data Types in Math

- In math, the basic data types are “sets”
  - sets are collections of objects called **elements**
  - write $x \in S$ to say that “x” is an element of set “S”, and $x \notin S$ to say that it is not.

- Examples:

| | | |
|---|---|---|
| $x \in \mathbb{Z}$ | x is an integer | |
| $x \in \mathbb{N}$ | x is a non-negative integer (natural) | |
| $x \in \mathbb{R}$ | x is a real number | |
| $x \in \mathbb{B}$ | x is T or F (boolean) | non-standard names |
| $x \in \mathbb{S}$ | x is a character | |
| $x \in \mathbb{S}^*$ | x is a string | |

# Basic Data Types in TypeScript

| Condition | Math | TypeScript | Up to Us |
|---|---|---|---|
| integer | $x \in \mathbb{Z}$ | `number` | no fractional part |
| natural | $x \in \mathbb{N}$ | `number` | non-negative |
| real | $x \in \mathbb{R}$ | `number` | |
| boolean | $x \in \mathbb{B}$ | `boolean` | |
| character | $x \in \mathbb{S}$ | `string` | length 1 |
| string | $x \in \mathbb{S}^*$ | `string` | |

we will often write
$x : \mathbb{Z}$ instead of $x \in \mathbb{Z}$

– **only division on integers can produce non-integer**

– **only subtraction on non-negative can produce negative**

# Ways to Create New Types In Math

- **Union Types** $\mathbb{S}^* \cup \mathbb{N}$
  - contains every object in either (or both) of those sets
  - e.g., all strings and natural numbers

- If $x \in \mathbb{N} \cup \mathbb{S}^*$, then $x$ could be a natural or string

- Two sets can contain common elements
  - in this case, the sets are disjoint

# Ways to Create New Types in TypeScript

- **Union Types** `string | number`
  - can be either one of these
- How do we work with this code?

```
const x: string | number = …;

// can I call Math.abs(x)?
```

- We can check the type of `x` using "`typeof`"
  - TypeScript understands these expressions
  - will "narrow" the type of `x` to reflect that information

# Ways to Create New Types in TypeScript

- **Union Types** `string | number`
  - can be either one of these

- **How do we work with this code?**

```
const x: string | number = …;

if (typeof x === "number") {
  console.log(Math.abs(x))   // okay! x is a number
} else {
  …                          // x is a string
}
```