Please open **pollev.com/comp110** and respond to the opening poll question! Not registered? Do so before next week!

Lecture 01

Started from the Bottom

Statements and Variables

Getting Started Open House Friday 12p-5p

- Run into issues getting software installed?
- Can't run the code coming up in today's lecture?
- Come by the Getting Started Open House!
- **TOMORROW: Sitterson 008 from 12pm to 5pm!**

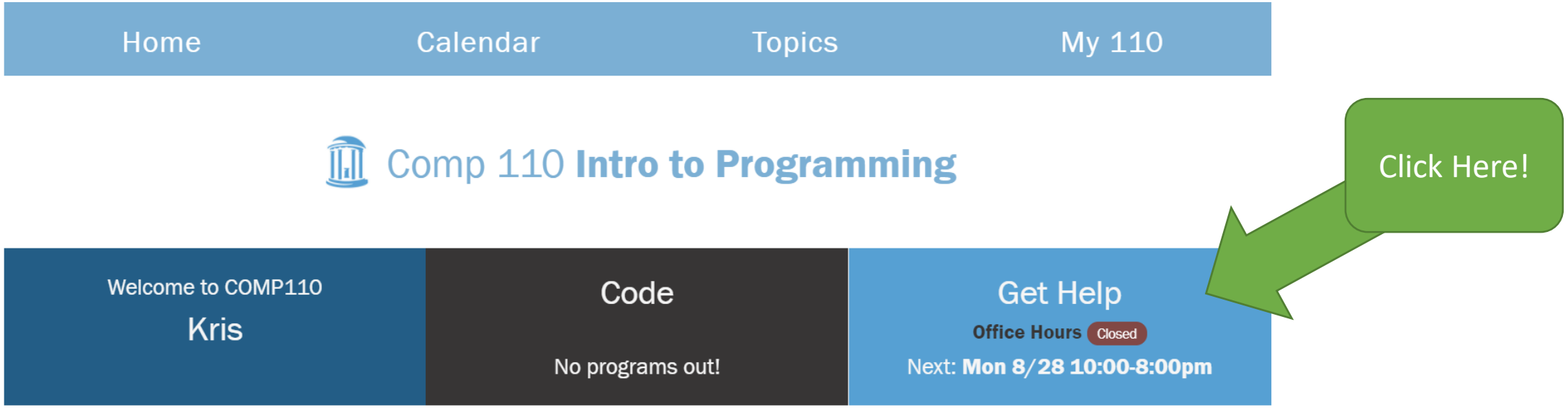
Review Sessions

- Lecture-format review sessions will run Wednesdays
  - Time and Location TBA next week
- Small Group Review Sessions will cover the same material as Lecture-format and run Thursdays & Fridays from 2-6pm starting next week.
  - Location TBA
  - No appointments needed. Come on in!
- These are for reviewing general concepts and questions.
- For help with Problem Sets and Worksheets use Office Hours…

Office Hours Check-in Process

- Regular Office Hours begin Monday at 10am through 8pm!
- You must "check-in" to get help in Office Hours… it's like calling an Uber
- Let's walk through the process…

Office Hours Check-in Process



Welcome to COMP110

Wednesday, July 12th

Welcome to Carolina! This Fall we'll go from zero-to-programmers with no prior programming experience expected. Comp 110 is open to and taken by all majors, not just those thinking about concentrating in Computer Science. Learning computer programming is a rewarding, challenging, and increasingly valuable skill. If you are a curious soul who enjoys creative problem solving: you're in the right place.

While this course does not currently have a Math prerequisite, we have found students

Course Info

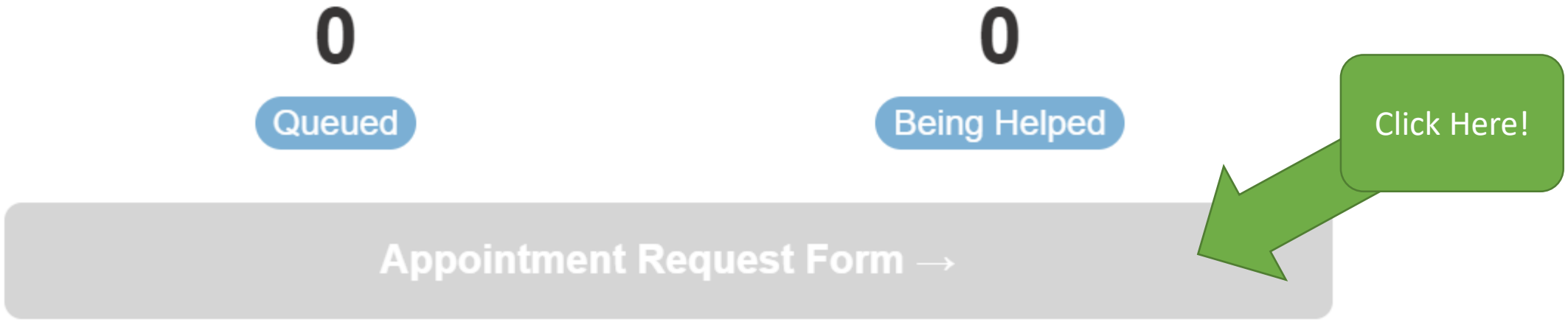
Teaching Staff

- **Kris Jordan**
- **The Comp110 Team**

Sections

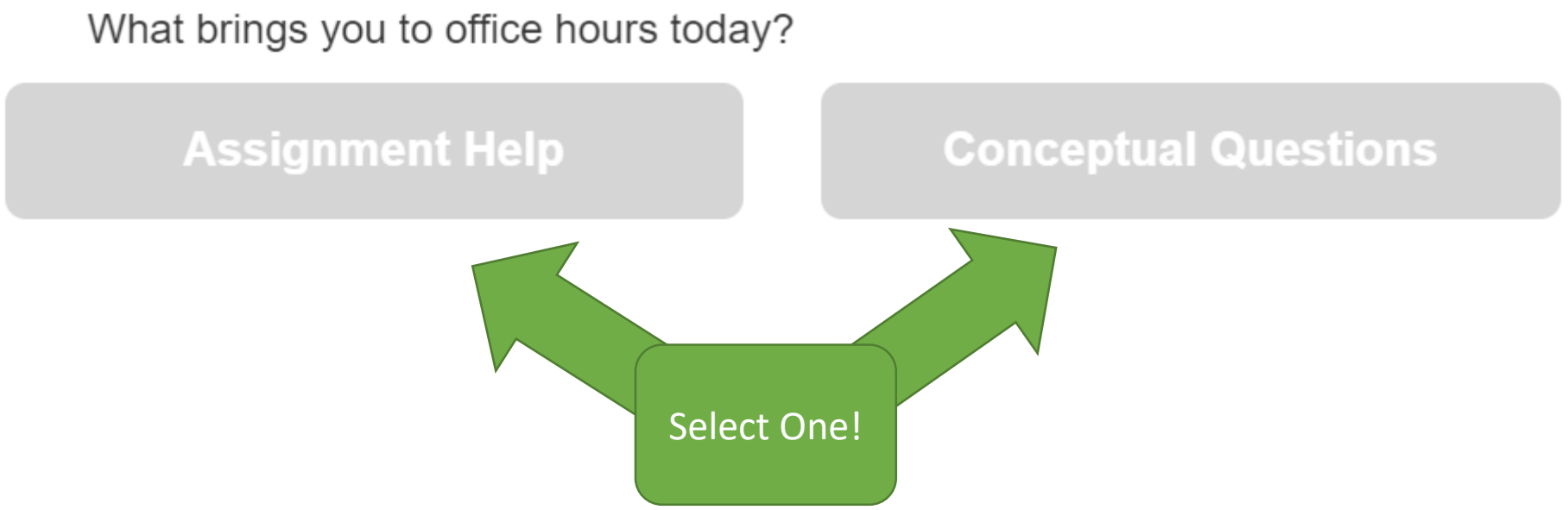
Honors - M/W 11:15 - 12:30pm
Section 2 - T/Th - 12:30 - 1:45pm
Section 3 - T/Th 3:30 - 4:45pm

Office Hours Check-in Process

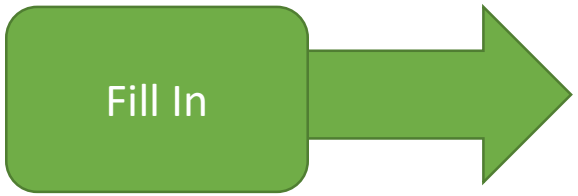


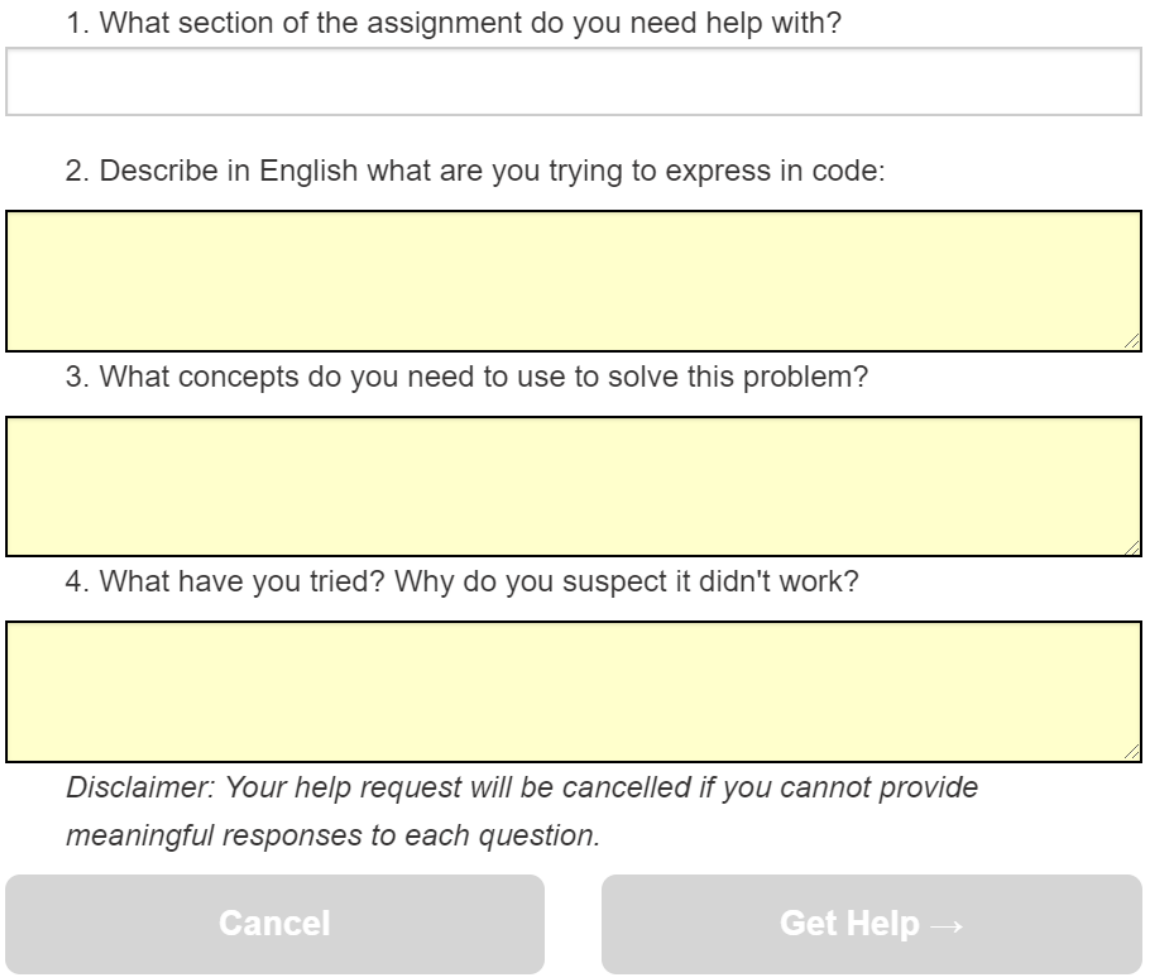
You can see how many people are currently waiting to be helped and currently being helped ahead of you.

Office Hours Check-in Process



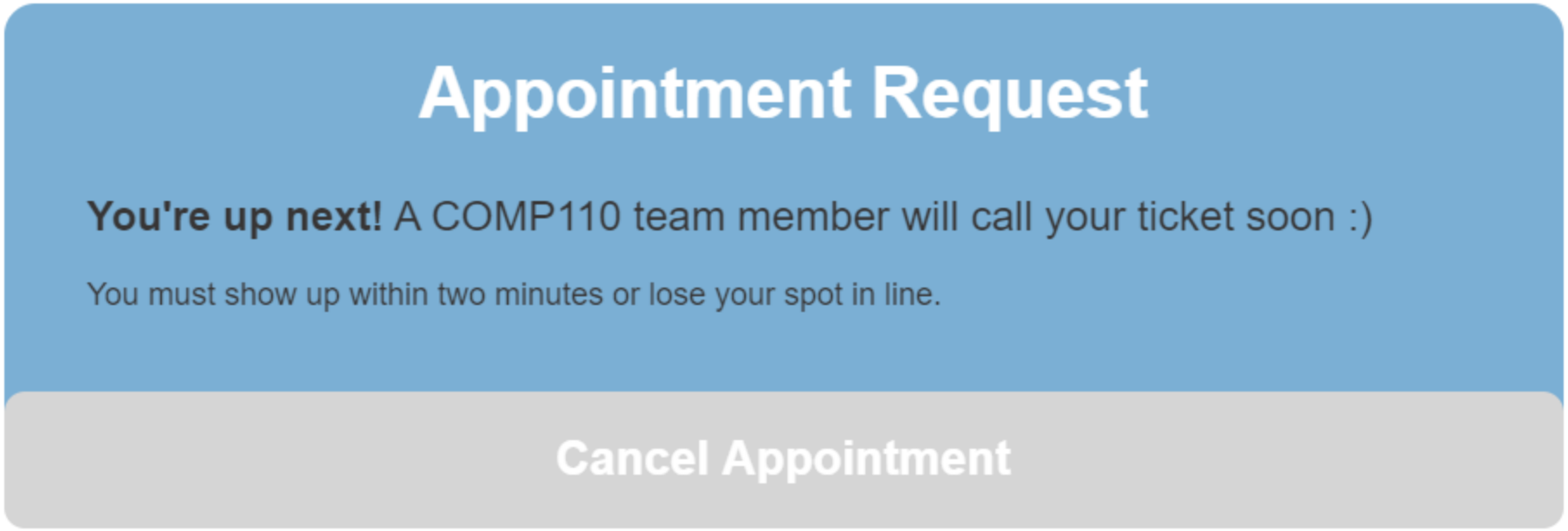
Office Hours Check-in Process





IMPORTANT: You must demonstrate **effort and thought** in these fields. If you do not, the TAs are instructed to **cancel** your request so you can try again.

Office Hours Check-in Process



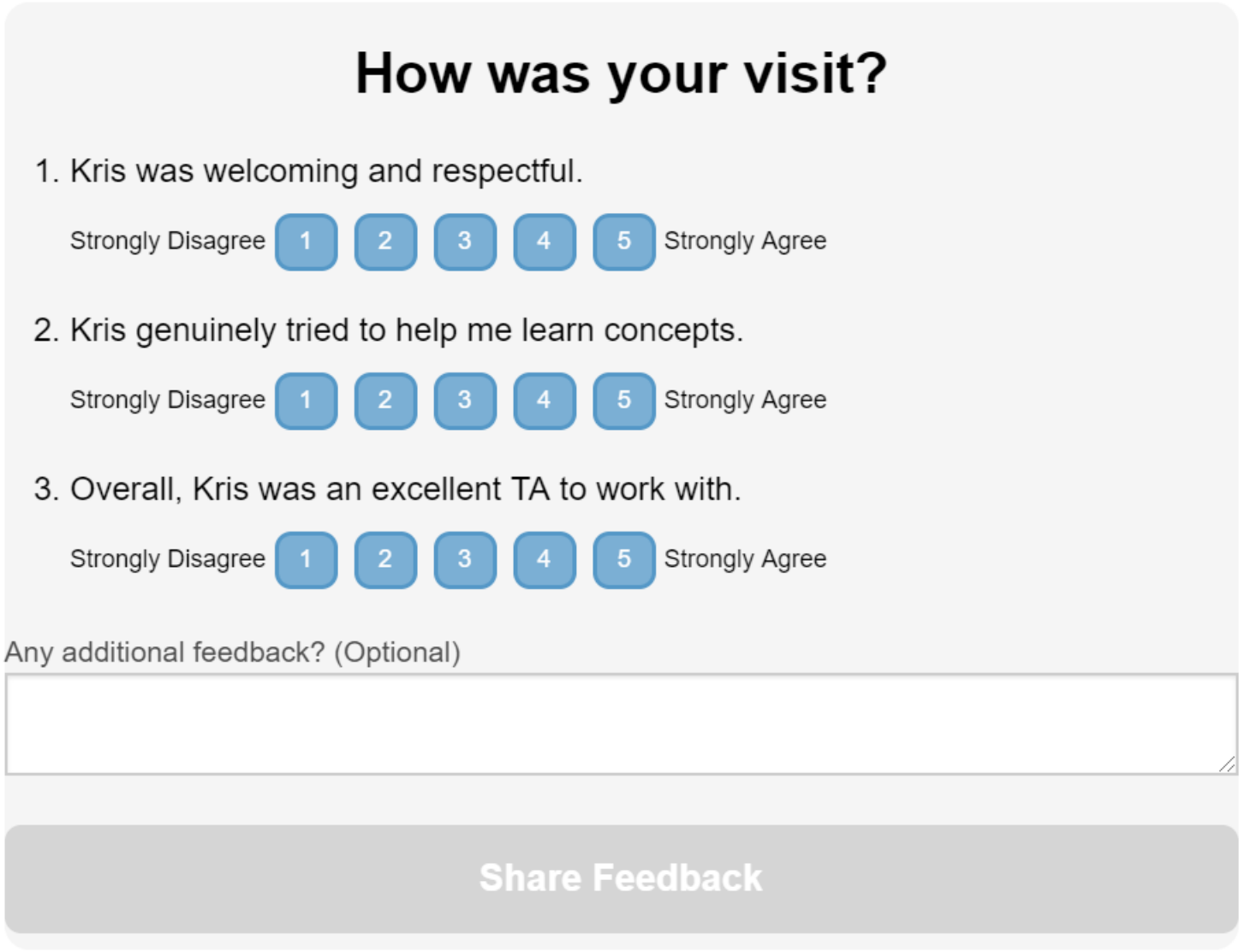
Office Hours Check-in Process

Kris is ready for you!

Come on in to SN008! You must show up within two minutes or lose your spot in line.

Cancel Appointment

Office Hours Check-in Process – **TA Feedback**



Like AirBnB & Uber: TA Feedback is 2-way

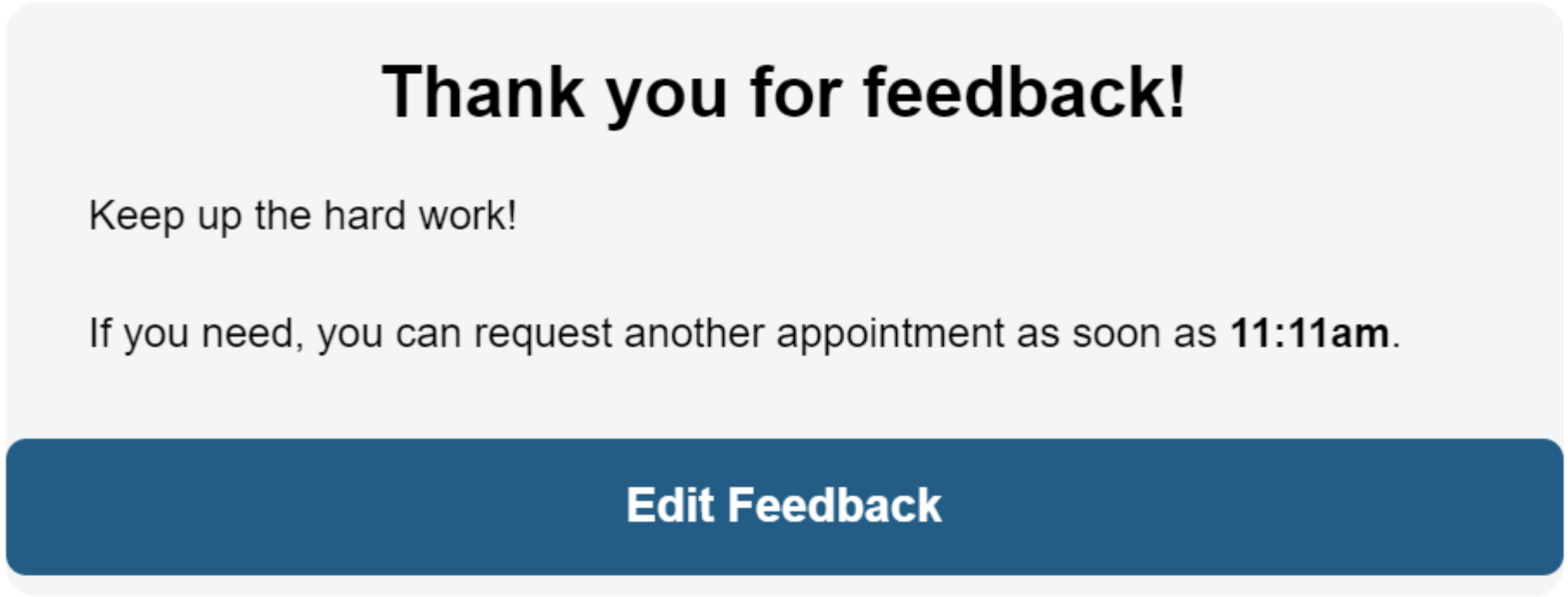
TAs will asses the appointment, as well:

1. Did you come to office hours prepared and able to demonstrate effort?
   - Come in ready to describe what you have tried so far!

2. Were you respectful and genuinely engaged in learning?
   - Don't come just looking for an answer. This strategy will bite you on exams!

3. Are we concerned over your progress in the course?

Office Hours Check-in Process – **Waiting Period**



Office Hours

- When other students are waiting, meetings are for up to 15 minutes
  - You can see how many people are currently waiting for help.
  - Come *before* due dates!
- Last semester, the average wait time was *less than five minutes*
  - Wait times grow longer as deadlines approach.
  - Start early!
- TAs are here to help you learn and work through misunderstanding
  - They are not here to tell you what to do nor give you answers

Use Office Hours!!!!! … *with Moderation*

- Overdosing on "office hours" leads to bad outcomes.
- Median student used office hours for one to two appointments / week.
- The max student had 96 appointments in the Spring.
  That's an average of **6 per week** for *24 hours* spent in office hours!
  - Did not enjoy the course (understandably!)
  - Did not do well in the course (understandably!)
- "Living" in office hours will be detrimental to your performance on exams.

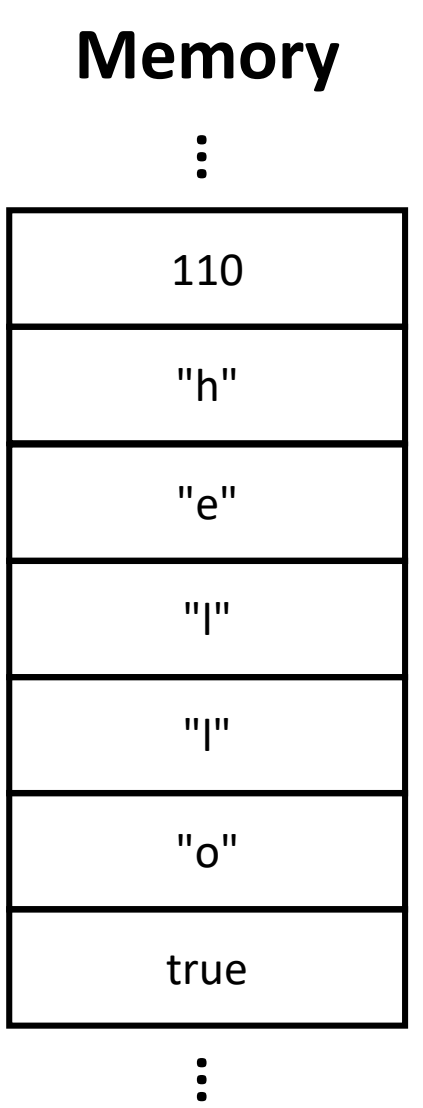
And now, back to Computer Science…

Fundamental Components of a Computer

1. **Memory**
   - Storage space for data.
   - **Millions to billions of tiny spaces** for storing numbers, letters, etc.

2. **Processor** *(Central Processing Unit)*
   - Has an instruction set that determines its capabilities.
   - Far simpler and fewer built-in capabilities than you'd think!
     - Add, subtract, load, store, jump, compare, etc.
   - Can retrieve and store values to and from memory.

3. **Program**
   - A sequence of instructions the processor follows one-by-one.
   - As a programmers, *you* are authoring these instructions!



What is a program?

- Computers run ***programs***
- A program is a series of ***instructions*** that *load*, *change*, and *store* data in ***memory***
- Computer chips understand ***machine code*** instructions in a ***binary*** code format
  - It's *very* painful for humans to author directly
- So humans can write machine-level code in a *slightly* nicer language called ***assembly code*** that is ***compiled***, or translated, into machine code.
  - It turns out assembly is painful to work with, too.
  - You can work with it in COMP411.

```
add $t1, $s3, $s3    # Temp reg $t1 = 2 * i
add $t1, $t1, $t1    # Temp reg $t1 = 4 * i
add $t1, $t1, $s6    # $t1 = address of save[i]
lw $t0, 0($t1)       # Temp reg $t0 = save[i]
bne $t0, $s5, Exit   # go to Exit if save[i] ≠ k
 add $s3, $s3, $s4   # i = i + j
add $t1, $s3, $s3    # Temp reg $t1 = 2 * i
add $t1, $t1, $t1    # Temp reg $t1 = 4 * i
add $t1, $t1, $s6    # $t1 = address of save[i]
lw $t0, 0($t1)       # Temp reg $t0 = save[i]
beq $t0, $s5, Loop   # go to Loop if save[i] = k
```

What is a programming language?

- Most programmers use high-level **programming languages**
  - First programming language invented in the 1952 by Grace Hopper
- Programming languages make reading and writing programs *much* more pleasant
- Like written languages, each has **syntax** and **grammar**
- Programs are ***compiled*** *into* or ***interpreted*** *as* machine code
  - Through a series of translations your TypeScript code becomes machine code instructions
  - We will not get into the details of this in COMP110
  - In COMP520 you can build your own compiler from scratch!

Grace Hopper

Lecture Readiness - "Pulling" Class Materials

- When you come into lecture each day, the routine we'll get into is:

1. Open PollEv.com/comp110

2. Open VSCode and its Integrated Terminal Tab

3. In the Integrated Terminal, first run: **npm run pull**
   - This downloads the latest lecture materials.

4. Then run: **npm start**
   - This starts the development compiler and server allowing us to see the output of our code.

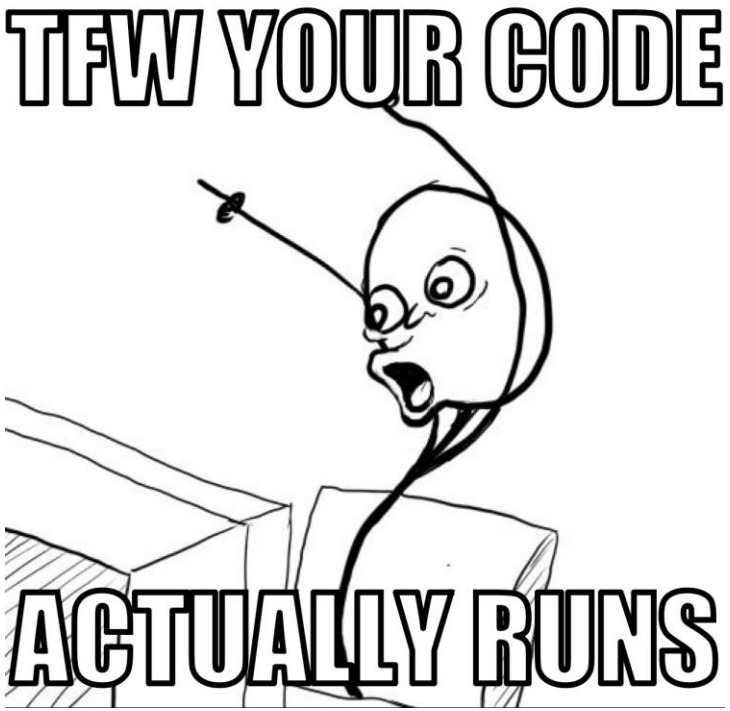
COMP110's Source Code Structure

- The folder "src" stands for "Source Code"

- Each subfolder groups related "apps"
  - Each Lecture and Problem Set will have its own folder
  - You can create your own subfolders for tinkering around

- Each "app" is a TypeScript file (.ts extension) that ends in -app
  - For example: variables-app.ts, hello-world-app.ts
  - You can create a new app by right clicking in a subfolder, selecting "New File", and naming the file something that ends in "-app.ts"

Follow-along: the classical “hello, world”



// Code Comments (1/2)

- We can add notes for ourselves in code using **comments**
- The computer ignores comments when running your program
- Single line comments begin with a //

```
// This is a single line comment
```

- Multi-line comments are surrounded by an opening **/*** and closing ***/**

```
/* This
   is
   a multi-line comment
*/
```

- Comments are also useful for ignoring code you've written without deleting it.

// Code Comments (2/2)

- In the early days of programming, we recommend writing comments liberally in your code to explaining what your program is doing in English.

- Comments are *free* so use them liberally.

- Comments are the easiest way to "take notes" when we are working on examples in lecture.

Statements (1/2)

- **Statements** are equivalent to an English sentence
- Statements *usually* end with a semi-colon ;
  - Like sentences end with a period!
- Each statement is an instruction you are giving to the computer.

```
print("hello, world");
```


That's a **statement**!

"Print 'hello, world' to the screen."

Statements (2/2)

- The computer will not carry out our instructions until we **run** the code
  - When you *write* programs, it is like you are writing a recipe down
  - When you *run* a programs, the computer is like the chef *following* your recipe
- Before the first statement runs, your program is a barren, empty world
  - *Your* code builds up its own little world piece-by-piece

```
print("hello, world");
```


That's a **statement**!

"Print 'hello, world' to the screen."

Data Types

- *Every **value***, or piece of data, in TypeScript has a specific ***type***

- 3 kinds of *simple data types*
  1. **Numerical**
  2. **Textual**
  3. **Boolean** (`true` or `false`)

- 2 kinds of *composite data types* composed of other data types
  1. **Arrays** are a series of values of a single type
  2. **Objects** can hold many values of many different types

Let's *print* some **literal values**!

- You should write: print( );
- Place your literal values within the blank space

- // number literals

10

3.14

- // string literal

"one"

"hello"

- // boolean literals

true

false

```
// String literal
print("These are literal values being printed out");

// number literals
print(20);
print(2.0);

// boolean literals
print(true);
print(false);
```

Numerical Type - `number`

- Literal examples: **`0, 1, 2, 3.14, 110.110`**

- We tend to use numerical data in two ways:
  - As **integers**, which are useful for counting
  - As **decimals**, which are useful in simulations, modeling, and so on

- Lower-level programming languages like Java and C have specific types for integer (i.e. `int`) and decimal data (i.e. `double`). In TypeScript, it's just **number**.

Textual Type - `string`

- **`string`** is short for "**string of characters**"
- Literal examples: **`"abc", "123", "~() @#z2"`**
- Useful for **all textual data**.

Logical Type - `boolean`

- Literal examples: **`true, false`**

- A **`boolean`** can only be **one of two possible values**, either `true` or `false`.

Block Statements

- A **block** is a special kind of statement that groups multiple, related statements.

- Blocks are **enclosing curly braces** that "contain" its statements.

```
{
   // This is a block statement
   print("Statement one");
   print("Statement two");
}
```

- Anywhere you can write a statement, you can also write a block statement.

- **Important** formatting rule: **each statement inside of a block is indented**!

- Blocks do not end with a semicolon after the closing curly brace.

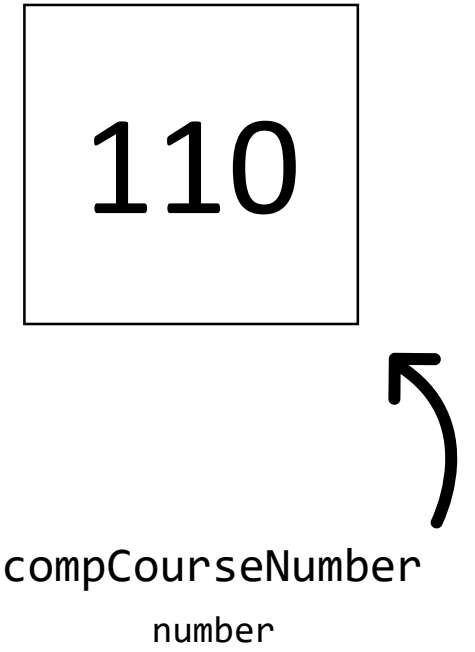
How do we work with values in memory?

- If a program is *"a series of instructions that load, change, and store data in memory",* then how do we work with memory?

- With **variables**!

- In programs, variables are used to *load*, *change*, and *store* values to and from memory.

- *Every* variable has a **name** and holds a specific **data type**.



Variable Declaration Syntax (1/3)

- When you **declare** a variable, you are proclaiming...
  "henceforth, within the nearest surrounding set of curly braces, the identifier <insert name> shall refer a(n) <insert type> value stored in memory"

```
let compCourseNumber: number;
```

  - "Let `compCourseNumber` refer to a `number` value stored in memory."

- General form:

```
let <name>: <type>;
```

- The type can be: **`number`, `string`, `boolean` (and more to come)**

Variable Declaration Semantics (2/3)

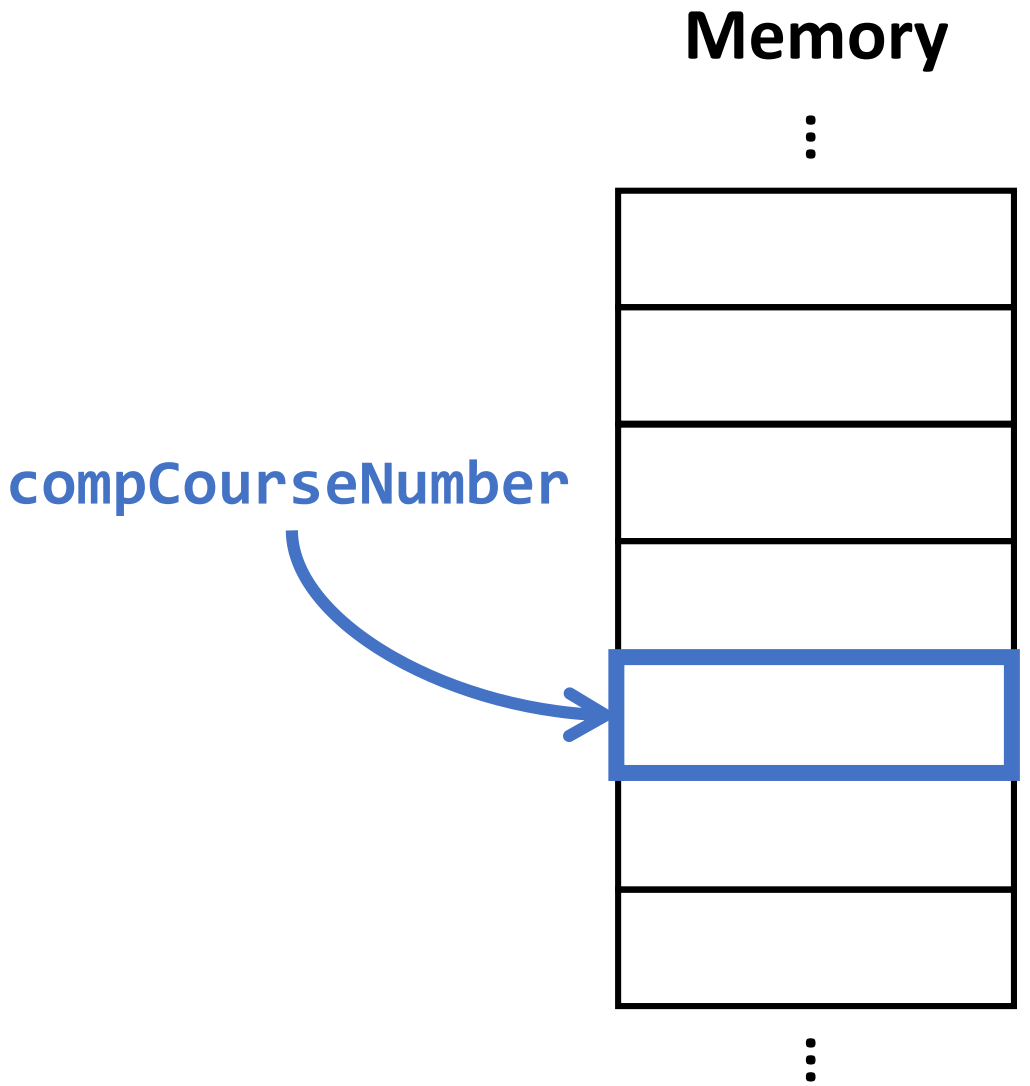
When this statement runs:

```
let compCourseNumber: number;
```

You are reserving a space in your computer's memory that can store a number value.

When you later refer to the word **compCourseNumber**, the program knows which exact location in memory it reserved.

Using this name, you can **store, access, and reassign** data in memory!



Variable Declaration Rules (3/3)

Variable names must be a single word, begin with a letter*, and contain only letters, numbers, and underscores.

In COMP110, we use "camel casing" for multiword names:

```
thisIsACamelCasedVariableName
```

You cannot refer to a variable until *after* its declaration.

Variables declared in block statements are only accessible inside of the block declared.

You cannot declare the same variable name twice in the same curly brace block.

Variable Assignment Syntax (1/3)

- The assignment statement **stores** a value in a variable

```
compCourseNumber = 110;
```

  - "`compCourseNumber` is assigned a value of `110`"
  - "`compCourseNumber` takes the value of `110`"
  - "compCourseNumber is now 110"
  - *Notice:* ***None of these readings uses the word "equals"!***

- General form:

```
<name> = <value>;
```

Variable Assignment Semantics (2/3)

When this line of code runs:

```
compCourseNumber = 110;
```

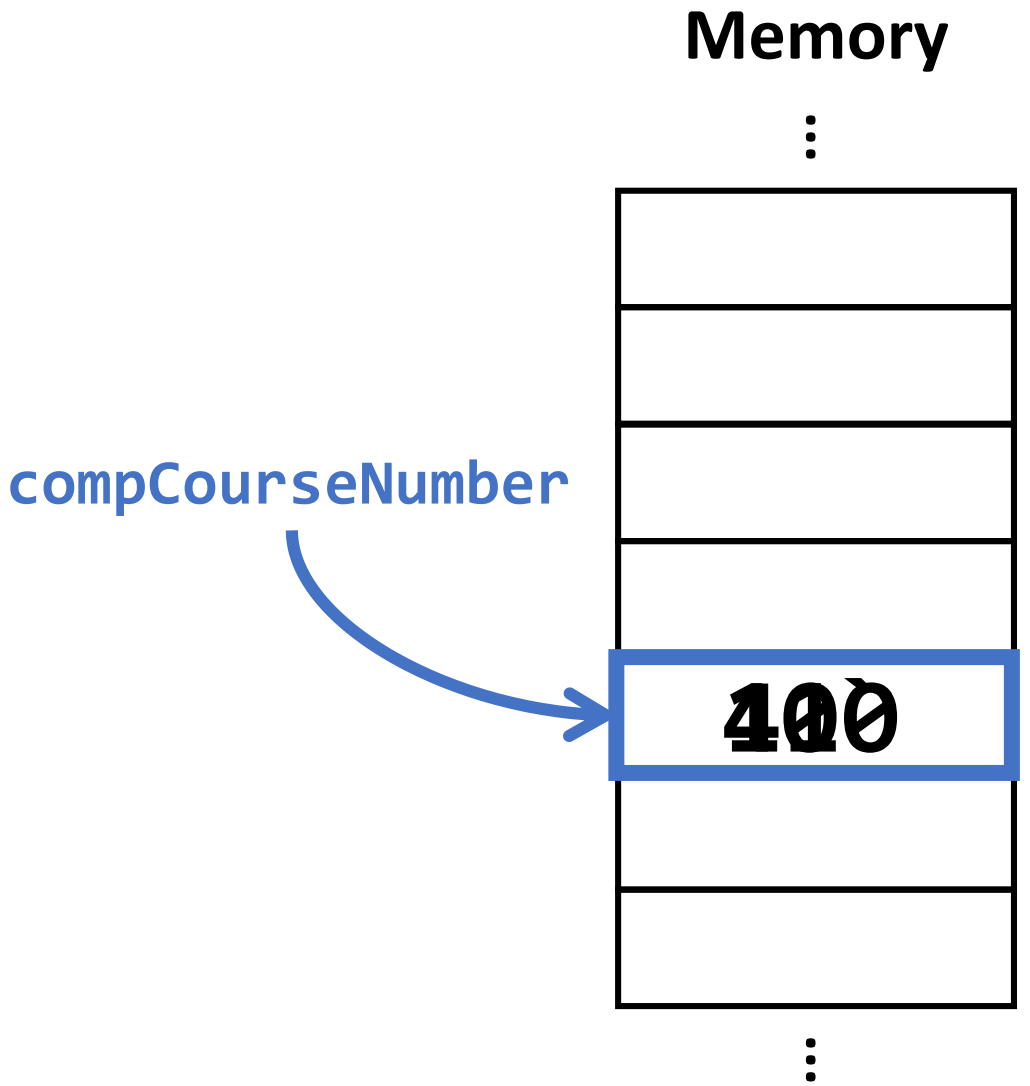
The value 110 is stored in the memory space reserved for the `compCourseNumber` variable.

Later, if the following line ran:

```
compCourseNumber = 401;
```

The value 401 is stored in the memory space it reserved for the `compCourseNumber` variable.

Assignment is *not* equality!



Variable Assignment Rules (3/3)

- **The value's type must match the variable's declared type**

- Variable Assignment Rules:
  - The *first* time you assign a value to a variable has a special name: **initialization**

  - It is very important to **always initialize your variables**
    - In other words, *always* assign a starting value to your variables

  - **A variable's value can change** as the program runs
    - Just assign a another value to the same variable!
    - After an assignment statement runs, when subsequent lines of code run the variable will have the most recently assigned value.

Variable Access – "Read" - Syntax (1/3)

- *After* you have *declared* a variable and *initialized* it by assigning a starting value…

- You can **access** (“read”, “load”) a variable’s value from memory **by its name**

```
print(compCourseNumber);
```

  - “Read *the last value assigned* to `compCourseNumber` and print it out.”

Variable Access – "Read" - Example (2/3)

- There are a *many* places where you can use variable access statements in code.

- For example, in assignment statements:

```
compCourseNumber = compCourseNumber + 291;
```

"`compCourseNumber` is now the last value assigned to compCourseNumber plus 291."

Steps:

1. current value of `compCourseNumber` is read
2. 291 is added to it
3. Result is assigned to `compCourseNumber`

Variable Access – "Read" - Rules (3/3)

- **Variable Scope:** Variables declared inside a block are only accessible in the same block.

```
{
   let x: number = 0;
   print(x); // OK!
   {
      print(x); // OK!
   }
}
print(x); // ERROR! x does not exist outside of its block
```

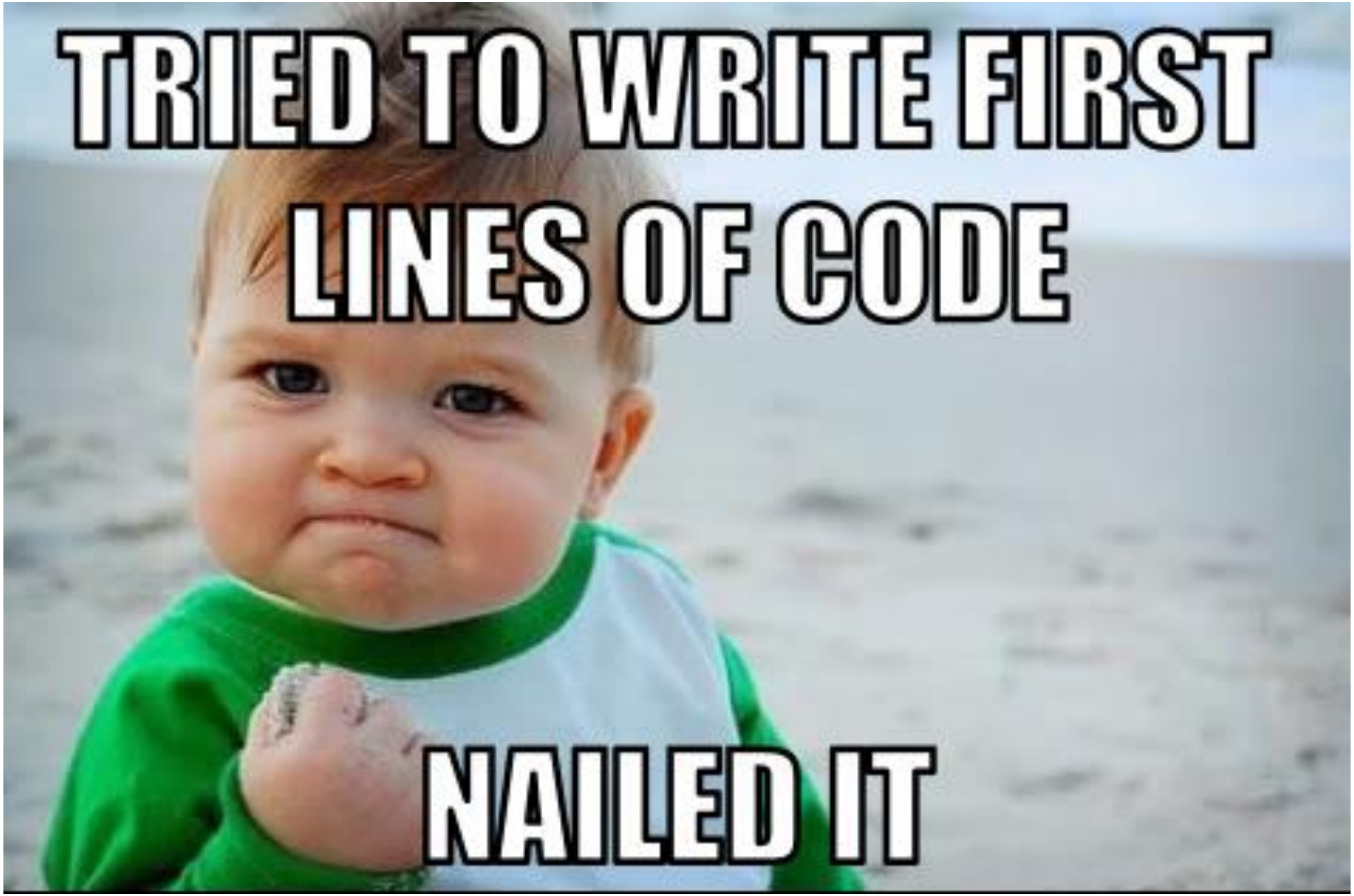
Variable Assignment *is not Equality*

- Imagine the following program:

```
1.  print("Variable Assignment");

2.  let compCourseNumber: number;

3.  compCourseNumber = 110;

4.  print(compCourseNumber);

5.  // Some time passes…

6.  compCourseNumber = compCourseNumber + 291;

7.  print(compCourseNumber);
```

compCourseNumber's value in memory

```
1.  undeclared

2.  uninitialized

3.  110

4.  110

5.  110

6.  401

7.  401
```


TRIED TO WRITE FIRST
LINES OF CODE
NAILED IT

Couldn't get code running today?

- **Come to the open house tomorrow from 12p-5p!**

- **Sitterson 008!**

Problem Set 0 – A Card for Someone Special

- Make a digital card in code for your Mom, Dad, Lover, Professor, *whoever (and send it to them!)*

- You will receive an e-mail from me over the weekend with instructions to get started on **Problem Set 0**.

- It will be due **next Friday** by 11:59pm.
  - We will go over hand-in instructions on Tuesday

- Very short, simple problem set to make sure everyone is setup!