CISC 181: DIGITAL SOCIETIES

UNIT 5: PROGRAMMING PART 1

ALGORITHM → PROGRAM

- An algorithm is an abstract (though precise) description of a process.
- A program is a computer implementation of an algorithm. No longer abstract, it exists as – or can be translated into – machine language and executed on a computer.

PROGRAMS

- Because a program is not an abstraction, it must deal with the realities of physical existence, including finite RAM, finite secondary storage, and an operating system that limits its share of these and other resources, notably
  - CPU time
  - Access to devices (mice, trackpads, touchscreens, printers, …)
  - Access to a network

PROGRAMS

- Of course, an operating system (Windows, macOS, Linux, Android, iOS, etc.) is a program or a collection of programs that must manage
  - an unpredictable mix of *application programs* (*apps*) all vying for its attention, and
  - orderly access for those programs to various system resources (RAM, disk access, space on the monitor, printing time, network access, …).

SOFTWARE

- The term *software* is used to refer to programs **and** the data they operate on.

ASSEMBLERS AND PORTABILITY

- As we have seen previously, assemblers – programs that translate assembly language text into machine language – are not difficult to create, because much of what they must do is straightforward translation of specific human-readable opcodes and memory address labels into their machine-readable numeric equivalents.
- The "grammar" used in an assembly language – the way instructions are structured – is modeled on the "grammar" of the targeted machine language, so that, too, makes for easy translation.

ASSEMBLERS AND PORTABILITY

- But, as we have also seen, since an assembly language is closely tied to a particular machine language, an assembler or an assembly language program written for one family of CPU won't be of use on a computer with a CPU from a different family, since the two computers will have different machine languages.
- We say that assembly languages and assembly language programs are not *portable* between different machine families/architectures.

ASSEMBLERS AND PORTABILITY

- Assembly language programmers, too, are not "portable," since people trained to write programs for one type of machine must start all over again on the introduction of a second type of machine. Thus, assembly language programming is highly-specialized work.

ASSEMBLERS AND PORTABILITY

- Assembly language programming is still done, but rarely, and only in very specialized computer engineering applications. It's easy to find debates online about its relevance for the typical computer programmer, many of whom will never see assembly language programming once they leave school.

HIGH-LEVEL PROGRAMMING LANGUAGES

PROGRAMMING AT A HIGH LEVEL

- To circumvent the assembly language portability problem, computer scientists in the late 1950s and early 1960s started developing the first so-called *high-level programming languages* or *high-level languages* (*HLLs*).
- The new HLLs were – and still are – generic, in that a program written in an HLL is (mostly) portable between machine types.

US Navy Admiral Grace Hopper (1906-1992) was a pioneer in the development of HLLs.
Source.

COMPILERS

- The way this was accomplished was to write translation programs, called *compilers*, that went far beyond what assemblers would do. In an HLL, a single instruction, called a *statement*, typically translates to a sequence of several assembly language or machine language instructions.

COMPILERS

- For example, a single HLL statement, such as this:

```
total = num1 + num2 + num3;
```

might, if translated by a compiler into an assembly language like the Toy's, turn into this series of instructions:

```
load num1
add num2
add num3
store total
```

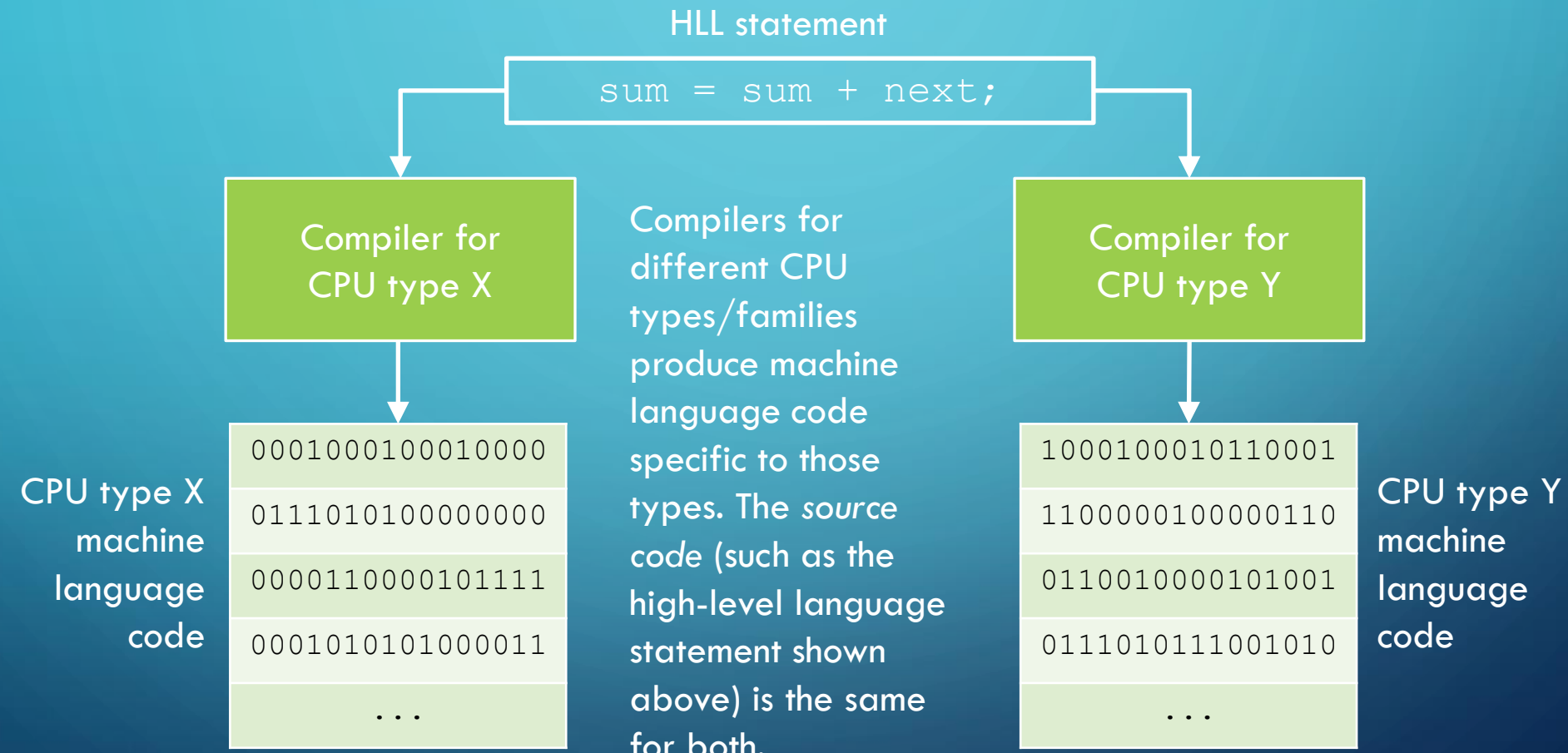
HLL READABILITY AND PORTABILITY

- If you found the HLL statement on the previous slide easier to read and understand than the Toy assembly language equivalent, you've spotted a huge advantage of programs written in an HLL over programs written in assembly language: They're easier for human programmers to write and to understand.
- This doesn't make them more portable, however.

HLL READABILITY AND PORTABILITY

- The portability comes from having a compiler for a specific HLL written for each new type of CPU that's released to the market. This is not an easy job, but it must be done only once for any given high-level language and any given CPU design. After that, programmers write their code in the HLL, and the compiler takes care of producing CPU-appropriate machine language code.
- The program – and the programmer – will be portable, since the "vocabulary" and "grammar" of the HLL will be the same, or very nearly the same, for every machine type. Only the machine language output from the compilers will be different.

HLL READABILITY AND PORTABILITY



HLL READABILITY AND PORTABILITY

- Though creating a compiler isn't easy (they teach courses on it in university computer science departments), a further shortcut is frequently taken to make the job easier.
- Many HLLs are like one another, enough so that translating a program written in one language into a somewhat similar language can be automated.

HLL READABILITY AND PORTABILITY

- For example, the first HLL compiler created for a new type of CPU is usually for a programming language called C.

```
#include <stdio.h>
int main() {
    printf("Hello, world!");
    return 0;
}
```

Source code for a complete C program.

HLL READABILITY AND PORTABILITY

- C has features that make code written in it easier to translate into various machine languages than code written in most other HLLs. This has led some to suggest that C is at the low end of the high-level languages (see, for example, this online discussion) being "closer to the metal" than other HLLs, and others to complain about how easy it is to overlook errors when programming in C (some types of which are examined in this article).

HLL READABILITY AND PORTABILITY

- If a C compiler has already been written for a new family of CPU, then creators of other HLLs can take advantage of that fact in two ways:
  - They can create compilers for other HLLs using C instead of assembly language and
  - They can create compilers that output C code as intermediate expressions of programs. The C code can then be translated to machine language by a C compiler.

HLL READABILITY AND PORTABILITY

- This means that HLL compilers can be made portable between CPU types.

COMPILERS AND INTERPRETERS

- As we've seen, a compiler translates a program written in an HLL into machine language (or maybe into an intermediate language, like C).
- Compilers convert whole source code programs – or at least large sections of source code programs – into potentially quite large binary files (binaries) that are placed into secondary storage, then loaded back into RAM and executed as needed.

COMPILERS AND INTERPRETERS

- However, many programming languages are designed to work differently.
- These languages don't rely on compilers to produce binaries; they instead are distributed with execution environments which are themselves compiled programs called *interpreters*.

COMPILERS AND INTERPRETERS

- Interpreters alternate between translating (interpreting) the source code statements of programs into machine language and executing them one at a time.
- Source code programs written for interpreters are frequently called *scripts* to distinguish them from programs designed to be compiled to disk. Hence, for example, the name of the programming language Java*Script* that we'll be using as an example HLL in this course.

COMPILERS AND INTERPRETERS

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

→ interpret → ignore
→ interpret → execute
→ interpret → execute
→ interpret → execute
→ interpret → execute
→ interpret → execute

Interpreting and executing a JavaScript program one statement at a time. The usual flow of interpreting/execution is from the top down, but this can be interrupted with looping and other techniques.

COMPILERS AND INTERPRETERS

- The term *just-in-time* (*JIT*) *compiler* is applied to a sophisticated HLL interpreter that analyses the first-time execution of an interpreted script to optimize – and thus speed up – some of its operations in subsequent runs. Many modern interpreters fall into this category.

COMPILERS AND INTERPRETERS

- Some languages, C for example, are generally viewed as compiled languages while others, like JavaScript, are typically thought of as interpreted languages, but compilers for JavaScript and interpreters for C have been written, so a language choice does not necessarily determine how a program will be translated and executed.

PROGRAMMING PARADIGMS

- C and JavaScript are members of a large family of HLLs called *imperative programming languages*, and the style of programming they support is sometimes called the *imperative paradigm*.

PROGRAMMING PARADIGMS

- The great majority of programming code in existence was written using imperative languages.
- Though high-level imperative languages are sufficiently generalized to make them portable – and relatively easy to program in compared with assembly language – they are still conceptually close to the underlying machine model with respect to such things as sequential execution, branching, and memory allocation.

PROGRAMMING PARADIGMS

- Other ways of thinking about computation have led to the development of a different set of HLLs that are broadly described as *declarative*. When working in the *declarative paradigm*, a programmer isn't concerned with the steps required to solve a problem. Rather, the programmer may describe a computation as a set of functions using a *functional programming language*, or as a set of logical relationships using a *logical programming language*.
- This Wikipedia article has more information about programming paradigms.

EXAMPLES OF (MOSTLY IMPERATIVE) HLLs

- Fortran – An early HLL well-suited to engineering applications. It is still around but used mostly by elderly engineers, I suspect.
- COBOL – An early business applications HLL. Lots of COBOL "legacy" code is still in use because it works, and no one dares turn off the systems – notably the banking systems – that use it long enough to replace it all.
- BASIC – An interpretive language built into many early desktop computers. It was used for years as a teaching language in high schools. It's still used in an embedded form (Visual Basic for Applications, or VBA) in Microsoft Office programs to automate tasks and to add functionality.

EXAMPLES OF (MOSTLY IMPERATIVE) HLLs

- C – Very widely used from its release in the early 1970s onward. (As an aside, Canadian Brian Kernighan, inventor of the Toy, co-wrote the definitive C language programmer's reference guide with Dennis Ritchie, the inventor of the language.)
- C++ – (Pronounced "C plus plus.") Developed as a superset of C to facilitate large programming projects.

EXAMPLES OF (MOSTLY IMPERATIVE) LANGUAGES

- Java – A lot like C++ but with automatic memory management.
- C# – (pronounced "C sharp.") Microsoft's response to Java. Much or most application development for Microsoft Windows is done in C#, and it is also used in programming for the web.
- JavaScript – Developed for use in web pages. JavaScript interpreters are built into web browsers (Firefox, Safari, Chrome, Edge, etc.). **Not to be confused with Java, despite the similarity in names.** The Toy computer simulator used in Lab 3 was written in JavaScript. A popular interpreter for JavaScript, node.js, allows JavaScript programs to be interpreted and executed outside of the control of browsers.

EXAMPLES OF (MOSTLY IMPERATIVE) LANGUAGES

- PHP – In widespread use on web servers. PHP is used to build web pages from various files on servers. PHP has routines built into it for allowing efficient access to online databases, making it suitable for use in online banking, shopping, and hotel or travel reservation sites.
- PHP is also used in many systems that allow people to manage websites even if they have limited knowledge of how websites work. An underlying PHP program, or a set of PHP programs, manages all the tricky web stuff for them. If you've worked on or visited a site that uses a web content management system like WordPress or Drupal, then you've interacted with an elaborate example of such a site.

EXAMPLES OF (MOSTLY IMPERATIVE) LANGUAGES

- Python – Widely used as a teaching language, but also used in industry and on web servers in the same way that PHP is used. It was named in honour of the UK comedy troupe, Monty Python.
- TypeScript – Very much like JavaScript, and compiles to ordinary JavaScript, but provides more rigorous control over data.
- Ruby – A multi-paradigm language in wide use. Software written in Ruby, called *Ruby on Rails*, is an important web app development tool.

AN EXAMPLE JAVASCRIPT PROGRAM

- Here's a JavaScript program shown on a previous slide. It is designed to collect a series of numbers from the user (via the keyboard) and display the average of those numbers.

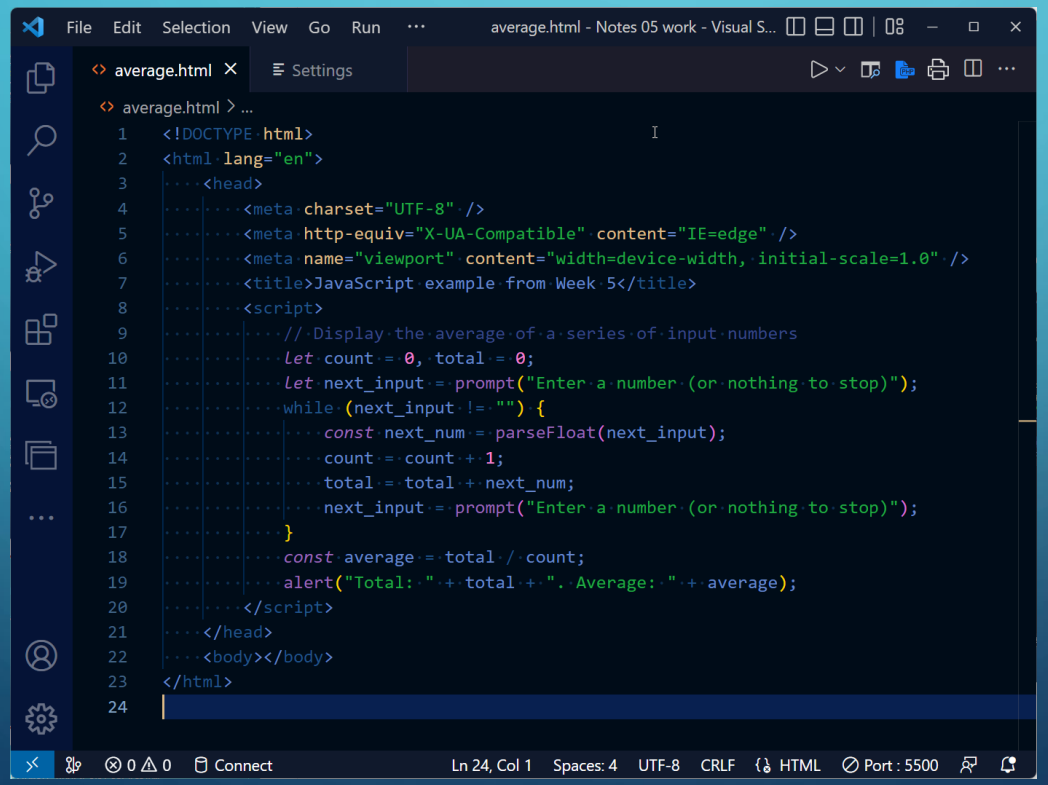
```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

AN EXAMPLE JAVASCRIPT PROGRAM

- Most JavaScript programs are written for execution in the context of web pages and are therefore embedded in the Hypertext Markup Language (HTML) documents from which web pages are constructed.
- JavaScript programs typically interact with the various elements – buttons, text areas, and so on – that comprise a web page and they rely on services provided by the browser to get input from the user and to display output.
- Our example program, like the Toy simulator, uses browser-provided pop-up windows to get keyboard input and another to display results.

AN EXAMPLE JAVASCRIPT PROGRAM

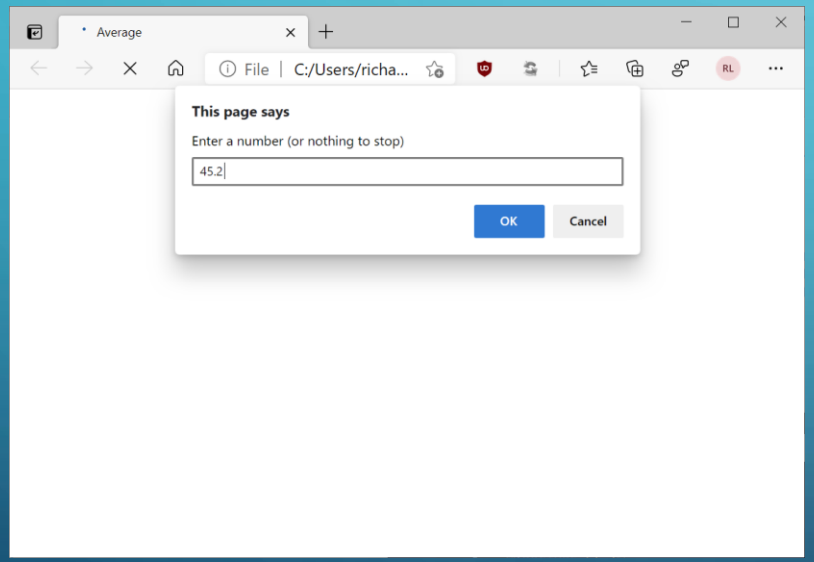
- Here's how the program, in its HTML file, appears in Visual Studio Code.

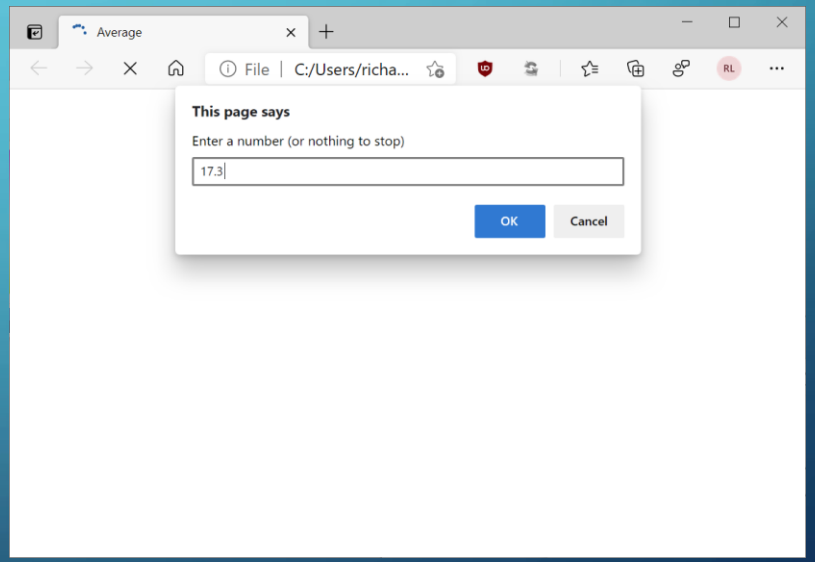


Here's the JavaScript program. Note that it sits inside `<script>` and `</script>` HTML tags. These tags tell the web browser to expect JavaScript code.

AN EXAMPLE JAVASCRIPT PROGRAM

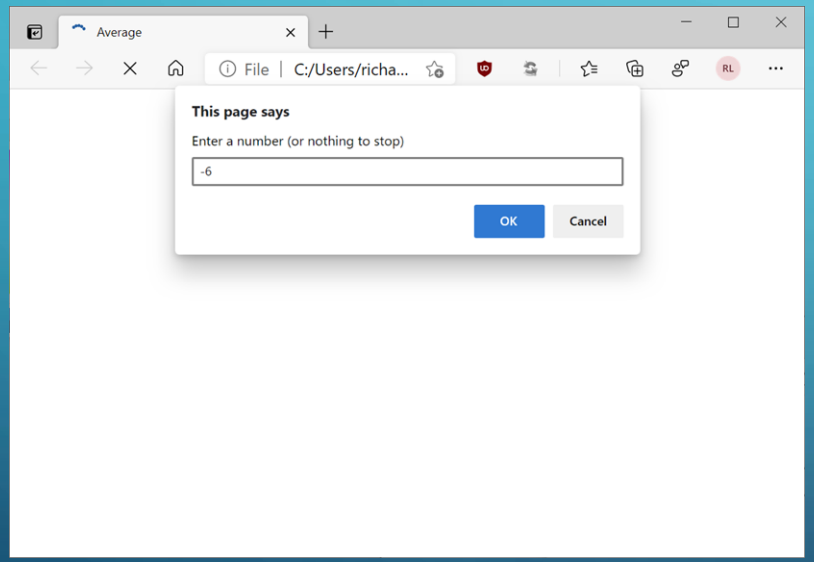
- When that web page is loaded into a web browser, the program is interpreted and executed automatically. Its execution might look something like this…

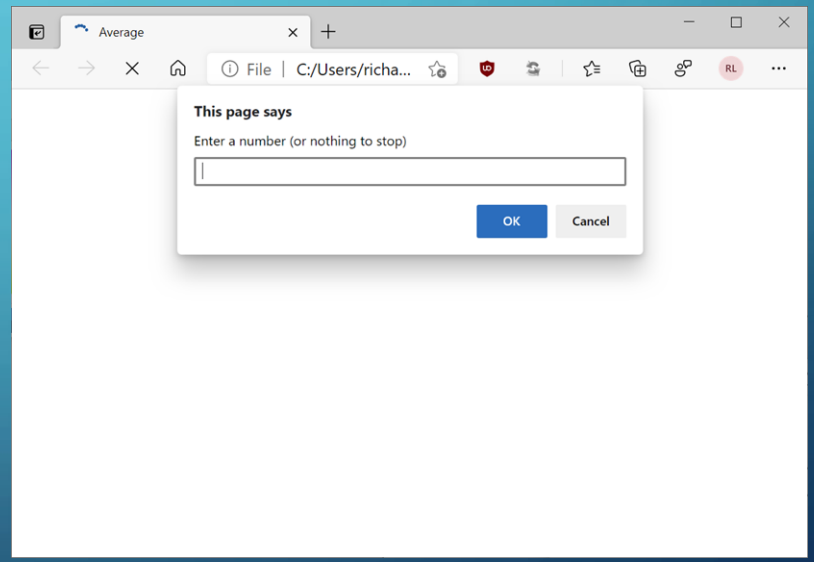




AN EXAMPLE JAVASCRIPT PROGRAM

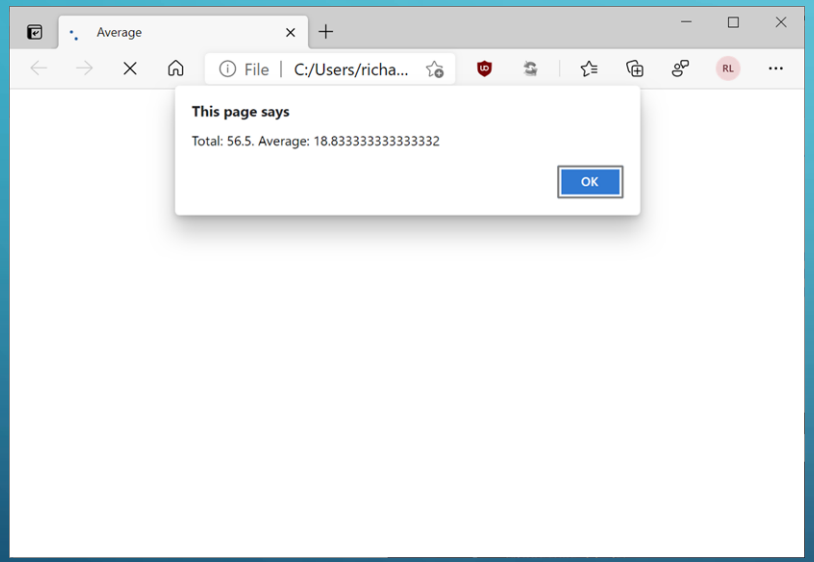
- Execution continues…

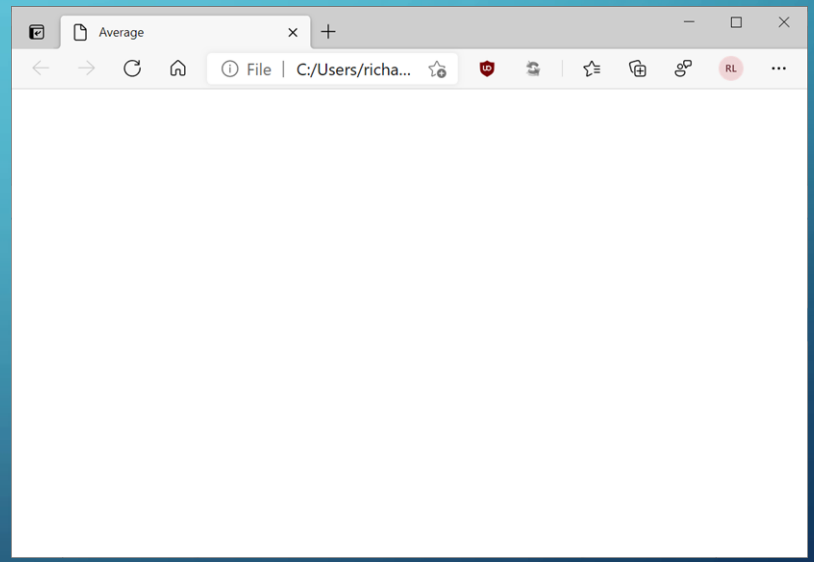




AN EXAMPLE JAVASCRIPT PROGRAM

- Execution continues… and stops.





DISSECTING THE EXAMPLE PROGRAM

AND INTRODUCING SOME PROGRAMMING VOCABULARY IN THE PROCESS

VALUES

- Before we start this detailed look at the example, note that when I use the word *value* it applies not only to numbers but to data items of any type, including strings of characters like "cat" or "encyclopedia".

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

COMMENTS

- The highlighted line here, preceded by two (forward) slashes, is a *comment*. It's a note from the programmer to other programmers that describes what the code is intended to do. It is ignored by the interpreter.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

KEYWORDS

- **`let`**, **`const`**, and **`while`** are examples of ***keywords***. As with the opcodes (LOAD, STORE, GET, etc.) in the Toy assembly language, they are part of the language definition for JavaScript and may not be used for other purposes.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

EXPRESSIONS

- An expression is something that either is a fixed value (like `0` or `"Enter a number (or nothing to stop)"`), has a value associated with it (such as a variable like `count` in this program), or computes to a value (like `count + 1`).

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

VARIABLES

- The highlighted items are examples of ***variables***. Variables are names (*identifiers*) of convenience given to data locations (addresses) in RAM. Two types of variables (strings and numbers) are used in this program.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

VARIABLES

- Variables in JavaScript are rather like the labeled memory locations at which we stored data in the Toy, but the JavaScript interpreter does a lot more work behind the scenes to manage a program's variables and the space they occupy in RAM.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

VARIABLES: `const`

- Two of the variables in this program are introduced using the **`const`** keyword. This signals to the interpreter that the variable will only receive a value once in a specific context and will not have that value changed.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

VARIABLES: `let`

- The other variables in this program are introduced with the **`let`** keyword. This tells the JavaScript interpreter that the values stored in those variables ***may*** change during the program's execution.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

VARIABLES: DECLARATION

- Bringing a variable into existence with a `let` or a `const` is called a ***declaration***. In many programming languages, a variable is declared along with an indication of its data type (integer, string, whatever). That's not always the case in JavaScript.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

VARIABLES: DECLARATION

- In many languages, including JavaScript, multiple variables can be declared in a single statement by placing commas between the declarations. This is just a programmer's convenience.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

VARIABLES: ASSIGNMENT

- The most common way to give a variable a value in most HLLs is to make what is called an ***assignment*** to it, and the equals sign, =, is used for this purpose in JavaScript. We call it the ***assignment operator*** for this reason.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

VARIABLES: ASSIGNMENT

- Variables are (usually) assigned values by placing the name (*identifier*) of the variable on the left side of the assignment operator, and an expression on the right.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

VARIABLES: ASSIGNMENT

- After an assignment statement is executed, the value of the variable will be the computed value of the expression.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

VARIABLES: INITIALIZATION

- Note that JavaScript allows assignment to happen in the context of a `let` declaration, useful for when the ***initial*** (first) value of a variable is known. To give a variable its first value is to ***initialize*** it. Variables declared with `let` do not need to be initialized right away but may be.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

VARIABLES: INITIALIZATION

- A variable declared with **`const` must always** be initialized (given a value) as part of its declaration.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

COMPARING VALUES

- This program makes use of the ***inequality operator***, **!=** (an exclamation mark immediately followed by an equals sign). **!=** is read as "is not equal to.". The ***equality operator*** in JavaScript is two equals signs, **==**. It's not used in this short program.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

COMPARING VALUES

- Here, the value (content) of a string variable, **`next_input`**, is compared with a string containing no characters (called *the empty string*). If **`next_input`** contains any characters, the expression **`next_input != ""`** computes to **`true`**. Otherwise, it's **`false`**.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

COMPARING VALUES

- Values computed using comparison operators like `==` and `!=` are of a type called *Boolean* (after George Boole, mentioned in Week 1). All such values are either **`true`** or **`false`**.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

COMPARING VALUES

- Here are some other Boolean operators used for comparing values in JavaScript. Interestingly, the Boolean operators can be used for comparing strings of characters for alphabetical order as well as comparing numbers.

Operator	Meaning	`true` Examples
<	Less than	`4 < 5` `"dog" < "elephant"`

<=	Less than or equal to	`4 <= 5`
		`5 <= 5`
		`"cat" <= "cat"`

>	Greater than	`5.2 > 5.1` `"cow" > "cat"`
>=	Greater than or equal to	Well, you get the idea.

STATEMENT TERMINATOR

- The semicolon used at the end of most of these lines of code is JavaScript's ***statement terminator***.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

STATEMENT TERMINATOR

- Where an English sentence expresses a complete thought and ends in a period, a JavaScript statement expresses a complete step in the implementation of an algorithm and ends in a semicolon.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

COMPOUND STATEMENTS

- A ***compound statement*** is one that contains a grouping of statements, such as the four statements constituting the body of this **`while`** statement.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

BLOCKS

- A group of statements like this is usually called a ***block***, or sometimes a ***statement block***. In JavaScript, blocks are surrounded by curly brackets, **{** and **}**, which are sometimes called "brace brackets".

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

BLOCKS

- Blocks in JavaScript and many languages introduce a new context for variables. Any variables declared above the block are available inside the block, but any variables declared inside the block, like **`next_num`** in this example, are only available for use inside the block.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

BLOCKS

- Moreover, any variable declared inside a block only exists for one iteration of the block, after which its memory is freed for reuse. That's why **`const`** is an appropriate way to declare **`next_num`** here.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

BLOCKS

- Programmers also indent blocks to emphasize structure – the body of a loop in this example – though most languages don't require it.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

LITERALS

- *Literals* are self-evident/as-they-appear values of any type. Their intended values are what they appear to be in the context of a program's code. Here we have number literals, 0 and 1, and several string literals in quotation marks.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

STRING LITERALS

- String literals are generally used in messages for the user or to initialize string variables. The quotation marks are only used in the source code, they are not displayed for the user. The programmer may use either double or single quotation marks to delimit a string literal.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

THE EMPTY STRING

- As previously noted, a string variable or string literal containing no characters is called the ***empty string***. The string literal highlighted here, two quotation marks with no space between them, represents the empty string.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

STRING LITERALS AS/IN EXPRESSIONS

- String literals may form expressions by themselves or be used with + to build longer string expressions. More on that later.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

OPERATORS

- ***Operators*** are those parts of an assignment or expression that tell the compiler the nature of a computation.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

OPERATORS

- We've already looked at the assignment operator, =, and the inequality operator, !=. The ***division operator***, / also appears here, as does the plus sign operator, +. The plus sign is a special operator, however.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

OPERATORS

- The two plus signs highlighted here are ***addition operators***. The first causes the value 1 to be added to the value stored in variable **`count`**, the second to increase the value stored in **`total`** by the value stored in **`next_num`**.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

OPERATORS

- These plus signs, however, are serving an entirely different purpose. They are stitching together two string literals and the values of two number variables into a single string.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

OPERATORS

- Used this way, + becomes the ***concatenation operator***. Concatenation is used to join strings together or to combine strings with non-string values.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

OPERATORS

- The interpreter decides on the meaning of + from the context in which it is used. This can be confusing for programmers at times. When an operator has two meanings in a programming language, we say it is ***overloaded***.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

OPERANDS

- The parts of an expression that aren't operators are called ***operands***. They form the data parts of an expression, while the operators tell the compiler what is to be done with the operand data.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

FUNCTION CALLS

- The highlighted parts here are examples of ***function calls***. A function call causes some external block of code (called a *function*) to execute. A function call consists, at minimum, of the function's name followed by a pair of round brackets.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

FUNCTION CALLS

- Unlike a GOTO, IFZERO, or IFPOS instruction in the Toy, a function call executes the code inside the named function, then automatically returns execution to the calling part of the program so it may continue running.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

FUNCTION ARGUMENTS

- If a function call is made with its round brackets containing one or more expressions, we refer to those expressions as the ***arguments*** of the function. You may also hear them called *parameters*.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

FUNCTION ARGUMENTS

- The function calls in this example all have single arguments which happen to be strings. Multiple function arguments are allowed in JavaScript, but the arguments must be separated with commas. Arguments are used to convey custom data values to a function.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

FUNCTIONS: `prompt`

- **`prompt`** is a function implemented in the user's web browser which creates a pop-up window containing a message (the string argument provided by the programmer) on the user's display and waits to accept keyboard input in response to the message.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

FUNCTIONS: `prompt`

- When **`prompt`** has done its work, its window closes, and execution returns to the statement that was executing when the function call was made. Functions may, optionally, resolve to some value that can be used in an expression or assigned to a variable, in this case, **`next_input`**.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

FUNCTIONS: `prompt`

- **`prompt`** resolves to a string of characters representing whatever the user typed in response to the message they were presented with. We say this is the value *returned* by the function which, in this program, is then assigned to **`next_input`**.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

FUNCTIONS: `prompt`

- But wait, if **`prompt`** returns a string of characters, making **`next_input`** a string variable, that presents a problem in a program that requires numeric input to compute a sum and an average.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

FUNCTIONS: `parseFloat`

- That problem is solved in this program by running **`next_input`** through the **`parseFloat`** function.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

DISSECTING THE EXAMPLE PROGRAM

- The value returned by **`parseFloat`** is a number computed from the numeric characters in its argument. (For example, **`parseFloat("1234.5")`** returns the number 1234.5). In this program, that means **`next_num`** is assigned (and becomes) a number and not a string.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

DISSECTING THE EXAMPLE PROGRAM

- The **`alert`** function causes the web browser to pop up a message (its string argument) on the user's display. Unlike the **`prompt`** function, **`alert`** doesn't accept input from the user, except for a button click to close the pop-up.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

TRACING THE EXAMPLE PROGRAM'S EXECUTION

- As mentioned earlier, program execution begins at the top and proceeds to the bottom but may be interrupted by branching instructions and loops. Here, the first line of code isn't a statement, it's a comment, so results in no computation.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

TRACING THE EXAMPLE PROGRAM'S EXECUTION

- Next, we have two variables being declared and initialized as numbers.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

TRACING THE EXAMPLE PROGRAM'S EXECUTION

- This statement executes the **`prompt`** function. It causes an input window to appear on the user's screen displaying the message, "Enter a number (or nothing to stop)." The intention is that the user takes the hint, types in a number, and enters it.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

TRACING THE EXAMPLE PROGRAM'S EXECUTION

- However, the user is under no obligation to do what the prompt message says and may type in a non-number before clicking on OK or pressing Enter/Return. A thorough programmer would create extra code to protect against this. This simple program offers no such protection.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

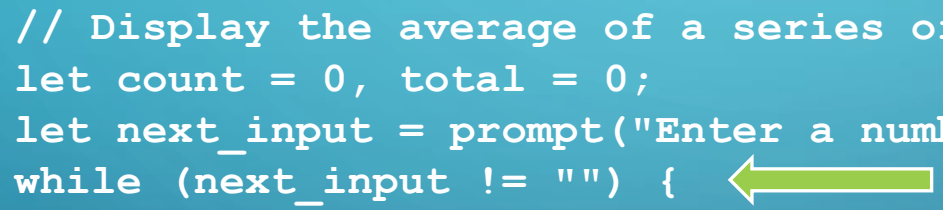
TRACING THE EXAMPLE PROGRAM'S EXECUTION

- All being well, the **`next_input`** variable will be created and initialized either with a string that can be converted to a number **or** with the empty string if the user clicked OK or pressed Enter/Return without typing in anything.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

TRACING THE EXAMPLE PROGRAM'S EXECUTION

- Execution has reached the **`while`** loop. Inside its round brackets is a Boolean expression that will compute to either **`true`** or **`false`**, depending on the string value stored in **`next_input`**.



```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

TRACING THE EXAMPLE PROGRAM'S EXECUTION

- If the string **`next_input`** isn't empty, then the expression **`next_input != ""`** (next input value not equal to the empty string) will be true, and the block of the **`while`** loop will execute.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

`while` loop block

TRACING THE EXAMPLE PROGRAM'S EXECUTION

- In this statement, **`parseFloat`** will take whatever it was the user typed and (if possible) convert it to an actual number which is then stored in a newly declared **`next_num`**.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

TRACING THE EXAMPLE PROGRAM'S EXECUTION

- Now the number variable **`count`** will have the value it contains increased by 1 (*incremented*).

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

TRACING THE EXAMPLE PROGRAM'S EXECUTION

- Number variable **`total`** will have its value increased by the value stored in **`next_num`**.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

TRACING THE EXAMPLE PROGRAM'S EXECUTION

- Then, **`next_input`** will have its string value overwritten by another string entered by the user in response to another pop-up **`prompt`**.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

TRACING THE EXAMPLE PROGRAM'S EXECUTION

- Execution now branches back to here where the user's input is again compared with the empty string. Depending on the computed value of the Boolean expression, the loop will either execute again or be skipped over.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

TRACING THE EXAMPLE PROGRAM'S EXECUTION

- We anticipate that, at some point, the user will stop entering numbers (or strings that look like numbers) and the loop will terminate. At this point, the **`average`** variable is created, and its value computed as the value of **`total`** divided by the value of **`count`**.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

TRACING THE EXAMPLE PROGRAM'S EXECUTION

- Finally, the values stored in **`total`** and in **`average`** are concatenated along with a couple of string literals into a message to the user displayed in an **`alert`** pop-up window. The user reads the message, then clicks on OK or Enter/Return, upon which the program stops.

```
// Display the average of a series of input numbers
let count = 0, total = 0;
let next_input = prompt("Enter a number (or nothing to stop)");
while (next_input != "") {
    const next_num = parseFloat(next_input);
    count = count + 1;
    total = total + next_num;
    next_input = prompt("Enter a number (or nothing to stop)");
}
const average = total / count;
alert("Total: " + total + ". Average: " + average);
```

SUMMARY

IN THIS UNIT WE LEARNED

- Programs are machine implementations of algorithms
- High-level (programming) languages (HLLs) are
  - easier to work with than assembly languages
  - portable between CPU families
- Compilers turn HLL code into assembly or machine languages
- HLLs are classified by paradigm
  - The imperative paradigm is most in use
  - Others are used sometimes in industry and academia

IN THIS UNIT WE LEARNED

- There are a great many HLLs, including a number that were described
- Interpreters are statement-by-statement compilers and executors of scripts

WE ALSO SAW

- A very short example JavaScript program that was used
  - to introduce (a lot of) programming terminology, and
  - to show the program works when it is executed.

Reminder: Friday is a quiz day.

CISC 181: DIGITAL SOCIETIES

UNIT 6: PROGRAMMING PART 2

GOALS FOR THIS UNIT

- This material will not turn you into an expert, nor even a novice programmer.
- It will (particularly if you do this unit's lab!) help you to understand some programming concepts beyond those introduced previously.
- It will perhaps give you the confidence to create a simple program in JavaScript based on an algorithm expressed in pseudocode.

GETTING STARTED

- Once again, our examples will be written in JavaScript code placed into HTML documents.
- Recall that whenever the HTML tags

  ```
  <script>
  ```

  and

  ```
  </script>
  ```

  are encountered in a web page, your web browser assumes that what lies between them is JavaScript source code that is to be interpreted and executed.

GETTING STARTED

- Recall, too, that execution of JavaScript in the context of an HTML document is tightly controlled by the browser's JavaScript interpreter, which acts as a sort of *virtual machine* protecting your actual machine, its operating system, and its file system from poorly written or malicious code.

INPUT AND OUTPUT (I/O)

- Most older high-level languages were created with good support for the type of application program that used a simple, text-only *console* for input and output.
- Historically, the word "console" was used to mean a combination of a keyboard for input and a monitor or printer for output.

I/O

- These older HLLs came with built-in functions for
  - getting user input from the keyboard, with characters appearing on the monitor as the user typed them;
  - sending output (messages, summaries, or lists of results of some computation) to a monitor, a printer, a disk file, a tape, etc.;
  - getting input in the form of a collection of information – called a *batch* – from a disk file or a tape, or – going way back – a stack of punched cards;

  among other things.

I/O WITH JAVASCRIPT

- JavaScript was developed with the idea that it would be interpreted in a web browser that would handle mundane chores like input and output (I/O) for it. For that reason, it relies on features of the browser for input and output services.

I/O WITH JAVASCRIPT

- These I/O services may be categorized as being either
  - easy to write into a program and ugly for the user, or
  - elegant for the user but requiring of the programmer an extensive knowledge of how to create and style web pages with on-screen data entry forms in HTML and how to interact programmatically with those forms through a programmer's interface called the *Document Object Model* (*DOM*).
  - Lab 05 made use of the DOM in its somewhat elaborate user interface.

I/O WITH JAVASCRIPT

- The examples I've given you in lecture – and will continue to give you for now – use the easy to write/horrible to look at I/O functions. To review…

I/O WITH JAVASCRIPT

- The **`alert`** function…

  `alert(`*string_expression*`);`

  The **`alert`** function is used for output. It causes a small pop-up window to appear on the screen. The *string_expression* argument is presented as a message in that window. The appearance of the window varies from browser to browser, but it will likely have an "OK" button on it that, when clicked or tapped on, closes the window.

I/O WITH JAVASCRIPT

- For example:

```
alert("Hello, world!");
```

  is a complete JavaScript statement that causes the message "Hello, world!" to pop up in a window. Try it by clicking here.

I/O WITH JAVASCRIPT

- The **`prompt`** function…

  `prompt` (*string_expression*)

  The **`prompt`** function is used for both output and input. It, too, causes a pop-up window to appear on the screen containing a message consisting of the *string_expression* argument, but it then waits for the user to type something in response. In addition to an "OK" button, it will likely have a "Cancel" button. When the user types something in response to the message and clicks on "OK" or presses enter, the response becomes available for use by the JavaScript program.

I/O WITH JAVASCRIPT

- For example:

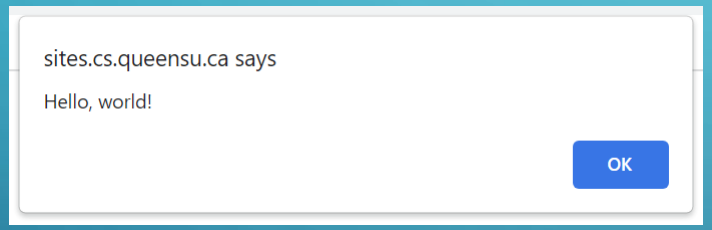
```
const userName = prompt("What is your name?");
```

  is a complete JavaScript statement that causes the message "What is your name?" to pop up in a window, waits for the user to type something and press Enter or click on "OK", then stores whatever the user typed in the variable named **`userName`**. Try it by clicking here.

  As pointed out in our first look at the **`prompt`** function, the type of data it produces is a string of text characters. It's up to the programmer to include code that converts this string to a number if numeric input is required.

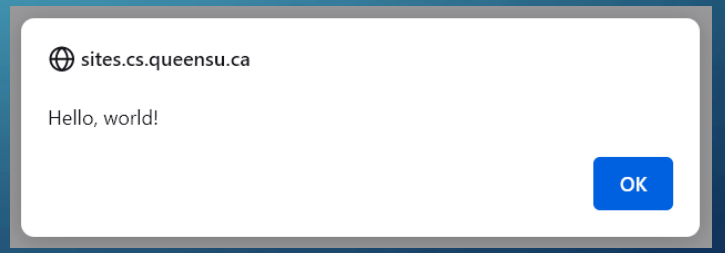
I/O WITH JAVASCRIPT

- The precise behaviour of both **`alert`** and **`prompt`** are determined by the browser, so although their purpose is the same across browsers, what their pop-up windows look like in different browsers is likely to vary slightly.



Output of the "Hello, world!" program in Chrome.

Output of the "Hello, world!" program in Firefox.



BY THE WAY…

- As mentioned in our last unit, JavaScript *does* exist outside of web browsers.
- Some application programs from Adobe, for example, have built-in JavaScript interpreters.
- A previously mentioned programming and execution environment, Node.js, is a free JavaScript interpreter that works in a console-style window rather than in a web browser. In this JavaScript variant, the **`alert`** and **`prompt`** functions don't exist. Because it is widely supported and operates independent of a web browser, Node.js is a popular choice for writing programs that run on web servers.

BY THE WAY…

- Unlike browser-embedded JavaScript programs, Node.js programs are free to create, modify, and erase files on secondary storage devices. Node.js has functions available to it for doing these things that are not available in standard JavaScript.
- Similarly, Node.js has facilities for creating and manipulating databases, making it more like PHP in terms of usefulness for web development.
- Though Node.js is an interpreted language, compilers have been created for it. See, for example, https://www.section.io/engineering-education/compile-your-nodejs-application-into-a-exe-file/.

BUILDING A JAVASCRIPT FUNCTION

THE GOAL

- This example is much less rudimentary than the averaging program I created for the previous unit, but it's a useful example of adapting an algorithm in pseudocode for use in a real program.
- My object is to build a JavaScript function that implements the pseudocode exchange sort studied in the Algorithms unit.
- My intention is to demonstrate a creation process for a small program. It's important that I have a clear idea about how a problem might be solved before I start to code it.

THE GOAL

- A function, you may recall from the previous unit and from our brief look at blockchain, is a sort of software factory that typically takes data of some sort and manipulates it to produce a useful result.

THE GOAL

- We are not restricted to the functions that come with JavaScript and your web browser, however, which is why I can build my own.
- When a function such as the one I'll be creating is written and tested, it can be placed in its own file or into a file with other functions I might write and incorporated into any programs that might need it with a single line of HTML code.

A SKELETON FUNCTION

- This is what a JavaScript function that performs an exchange sort on a list (array) of data might look like when it is first created, and before it does anything useful. We may call this early, empty form of a function a *skeleton*.

```
function sort(a) {
    // Put elements of array a in ascending order.
}
```

A SKELETON FUNCTION

- **`function`** is a keyword that tells the JavaScript interpreter to expect a function definition. Since JavaScript is case-sensitive, it, like all other keywords, must be typed in lowercase.

```
function sort(a) {
    // Put elements of array a in ascending order.
}
```

A SKELETON FUNCTION

- **`sort`** is an identifier; that is, it's a meaningful name I made up for my function. **`prompt`** and **`alert`** are also function identifiers, but they are predefined.

```
function sort(a) {
    // Put elements of array a in ascending order.
}
```

A SKELETON FUNCTION

- These parentheses (round brackets) are a requirement of the JavaScript language for a function definition…

```
function sort(a) {
    // Put elements of array a in ascending order.
}
```

A SKELETON FUNCTION

- … as are these brace (curly, squiggly) brackets.

```
function sort(a) {
    // Put elements of array a in ascending order.
}
```

A SKELETON FUNCTION

- This is what's called a *parameter*. Within the function code, it will be the identifier used for the array that is being sorted. **a** will become an alias of whatever array is passed as an argument when this function is called.

```
function sort(a) {
    // Put elements of array a in ascending order.
}
```

A SKELETON FUNCTION

- Recall that in my pseudocode examples in the Algorithms unit, I referred to the elements of a list or collection called `a` by subscripting them as a_0, a_1, a_2, a_3, and so on. The elements of an array called `a` in JavaScript would be subscripted as `a[0]`, `a[1]`, `a[2]`, `a[3]`, and so on.

```
function sort(a) {
    // Put elements of array a in ascending order.
}
```

A SKELETON FUNCTION

- Finally, recall that two (forward) slashes in a line tell the interpreter that what follows is a comment and not executable code. A comment at the top of a function serves to tell programmers examining the code what the function is meant to do.

```
function sort(a) {
    // Put elements of array a in ascending order.
}
```

A BUILDING AND TESTING FRAMEWORK

- I've built the preceding skeleton function into an HTML and JavaScript framework where I can continue to build it and test it to my satisfaction. I used Visual Studio Code as my development environment, and a Chrome-like browser as my testing environment.

A BUILDING AND TESTING FRAMEWORK

- As much of this stuff is beyond the scope of this course, I've shared my HTML and JavaScript framework here, https://sites.cs.queensu.ca/courses/cisc181/exchange_sort_demo.zip, rather than suggesting you build it for yourself. Download the file to your computer and unzip it to somewhere convenient, like your Desktop. You can then edit the unzipped exchange_sort_demo.html file with a tool like Visual Studio Code and open it in your web browser to make its JavaScript execute.
- You can then copy what I do next to make your own working copy of the function.

A BUILDING AND TESTING FRAMEWORK

- This is what the unedited file looks like in VS Code:

exchange_sort_demo.html > ...

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Exchange Sort Demo</title>
        <script>
            // Test lists
            const empty_list = [];
            const word_list = ["tree", "duck", "frog", "rock", "orange"];
            const number_list = [17, -2.1, 8, 32, -15, 6.25];

            function sort (a) {
                // Put elements of array a in ascending order.
            }

            // Test code (should be more thorough)
            alert("Empty list (before): " + empty_list.join(", "));
            sort(empty_list);
            alert("Empty list (after): " + empty_list.join(", "));
            alert("Word list (before): " + word_list.join(", "));
            sort(word_list);
            alert("Word list (after): " + word_list.join(", "));
            alert("Number list (before): " + number_list.join(", "));
            sort(number_list);
            alert("Number list (after): " + number_list.join(", "));
        </script>
    </head>
    <body>
    </body>
</html>
```

← There's the skeleton function.

A BUILDING AND TESTING FRAMEWORK

- Up at the top of the JavaScript, I have my unsorted test arrays defined. Note that one, initialized with `[]`, is empty. It's always good to test a list-handling function on a list with nothing in it.

```
<script>
    // Test lists
    const empty_list = [];
    const word_list = ["tree", "duck", "frog", "rock", "orange"];
    const number_list = [17, -2.1, 8, 32, -15, 6.25];
```

A BUILDING AND TESTING FRAMEWORK

- Underneath the function, I have the code that displays the content of the lists before and after calls to my **`sort`** function. Note that each call to **`sort`** specifies, as its argument, the identifier of the list to be sorted.

```
    // Test code (should be more thorough)
    alert("Empty list (before): " + empty_list.join(", "));
    sort(empty_list);
    alert("Empty list (after): " + empty_list.join(", "));
    alert("Word list (before): " + word_list.join(", "));
    sort(word_list);
    alert("Word list (after): " + word_list.join(", "));
    alert("Number list (before): " + number_list.join(", "));
    sort(number_list);
    alert("Number list (after): " + number_list.join(", "));
</script>
```

A BUILDING AND TESTING FRAMEWORK

- Executed as is (by loading the file into a browser), the three "after" output (`alert`) messages just show the names and content of the unsorted lists, one after the other. This will change when I incorporate my exchange sort code into the **`sort`** function.

THE PSEUDOCODE

- This is the exchange sort pseudocode from the Algorithms unit's slides. This will be my guide to building a JavaScript equivalent.

```
let n be the number of items in a collection
let a be the collection with positions a0, a1, a2, … an-1
let x be 0
while x < n – 1
    let y be x + 1
    while y < n
        if ax > ay
            swap ax, ay
        add 1 to y
    add 1 to x
```

THE PSEUDOCODE

- In the following slides, as I turn this pseudocode into actual code, I'll include statements from the pseudocode surrounded by green borders so you can better see what I'm translating.

```
let n be the number of items in a collection
let a be the collection with positions a0, a1, a2, … an-1
let x be 0
while x < n – 1
    let y be x + 1
    while y < n
        if ax > ay
            swap ax, ay
        add 1 to y
    add 1 to x
```

THE NUMBER OF ITEMS IN A COLLECTION

- As already mentioned, a collection of items such as those used in our examples is called an array in JavaScript. The number of items in an array – its *length* – can be determined at runtime (while the program is being executed) by putting "`.length`" directly after the array's name. In programming terminology, `.length` is said to be a *property* of an array. Thus, the length of an array called `a` can be determined by a reference to `a.length`.

THE NUMBER OF ITEMS IN A COLLECTION

- Our skeleton function has already named the array parameter **a**, so that takes care of the second line of the pseudocode. We can use **a.length** throughout our function instead of creating a variable called **n** to hold the value, but there's no real advantage in doing that (and it involves more typing), so we can start our translation with this:

```
function sort(a) {
    // Put elements of array a in ascending order.
    const n = a.length;
}
```

```
let n be the number of items in a collection
let a be the collection with positions a0, a1, a2, … an-1
```

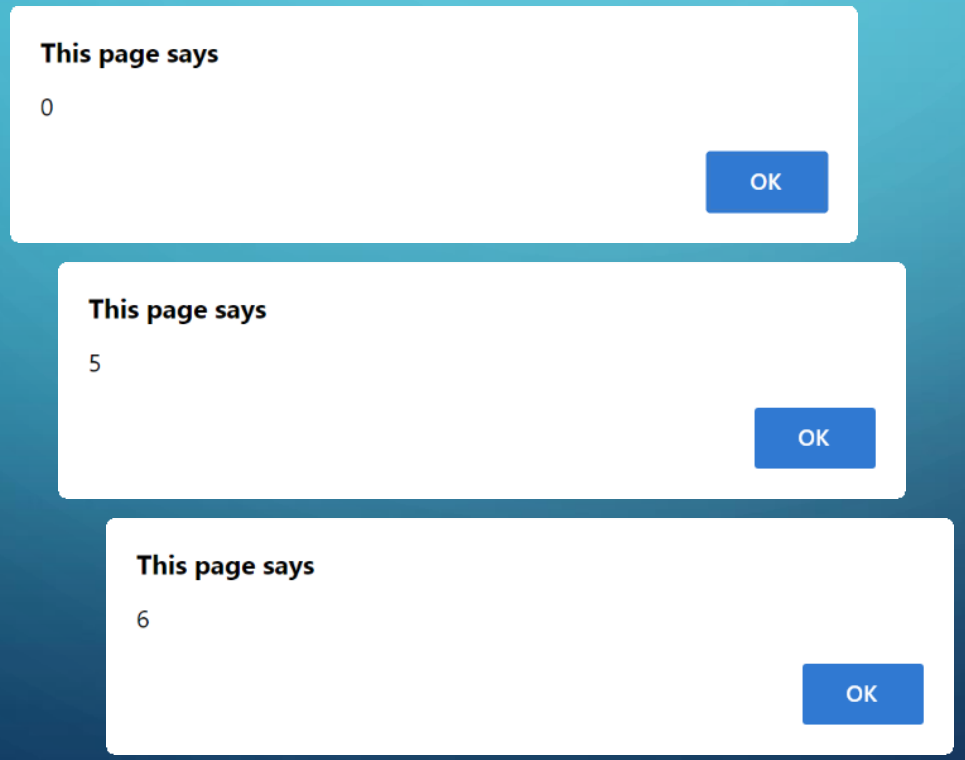
THE NUMBER OF ITEMS IN A COLLECTION

- There are many good tools available to programmers to check that variables are getting their expected values when their programs are run, but one of the easiest to use is the **temporary** insertion of an output statement that displays the variable's value. We can do that here with an **`alert`** message.

```
function sort(a) {
    // Put elements of array a in ascending order.
    const n = a.length;
    alert(n);
}
```

THE NUMBER OF ITEMS IN A COLLECTION

- Now when I save and run the program in my browser, I get three additional messages interspersed with the messages generated by the test code.



THE NUMBER OF ITEMS IN A COLLECTION

- Having determined that the function is displaying the correct array lengths, I can now remove the `alert` statement.

CREATING AND INITIALIZING `x`

- The pseudocode's "`let x be 0`" becomes a `let` variable declaration.

```
function sort(a) {
    // Put elements of array a in ascending order.
    var n = a.length;
    let x = 0;
}
```

```
let x be 0
```

BUILDING THE STRUCTURE OF THE OUTER LOOP

- We've seen JavaScript's **`while`** loop previously. Its requirements for bracketing makes it different from our pseudocode's loops. I put in the closing brace bracket right away, so I don't forget to add it later.

```
function sort(a) {
    // Put elements of array a in ascending order.
    const n = a.length;
    let x = 0;
    while (x < n - 1) {

    }
}
```

```
while x < n - 1
```

CREATING AND INITIALIZING y

- As with **x**, so with **y**, further indenting its declaration as in the pseudocode to show it is part of the body of the while loop.

```
function sort(a) {
    // Put elements of array a in ascending order.
    const n = a.length;
    let x = 0;
    while (x < n - 1) {
        let y = x + 1;
    }
}
```

```
let y be x + 1
```

BUILDING THE STRUCTURE OF THE INNER LOOP

- We introduce the nested loop…

```
function sort(a) {
    // Put elements of array a in ascending order.
    const n = a.length;
    let x = 0;
    while (x < n - 1) {
        let y = x + 1;
        while (y < n) {

        }
    }
}
```

```
while y < n
```

BUILDING THE STRUCTURE OF THE `if` STATEMENT

- As with our pseudocode, JavaScript has a *conditional* compound statement, **`if`**, that allows a program to choose to do something or not. **`if`** is a JavaScript keyword. The bracketing rules for an **`if`** statement are like those for **`while`**, but **`if`** is NOT a looping statement.

```
function sort(a) {
    // Put elements of array a in ascending order.
    const n = a.length;
    let x = 0;
    while (x < n - 1) {
        let y = x + 1;
        while (y < n) {
            if (a[x] > a[y]) {

            }
        }
    }
}
```

if $a_x > a_y$

BUILDING THE STRUCTURE OF THE `if` STATEMENT

- Recall that since text editors used for programming can't subscript array elements like the pseudocode's a_x or a_y, JavaScript uses square brackets around array positions instead.

```
function sort(a) {
    // Put elements of array a in ascending order.
    const n = a.length;
    let x = 0;
    while (x < n - 1) {
        let y = x + 1;
        while (y < n) {
            if (a[x] > a[y]) {

            }
        }
    }
}
```

`if` $a_x > a_y$

SWAPPING?

- Our pseudocode calls for a swap of list elements inside its **`if`** block. There are a couple of ways to achieve this in JavaScript. Here's the classic: Using another variable created for the sole purpose of storing one of the array values so it isn't lost in the exchange. I can make a separate function that does this and then add it to my code outside the **`sort`** function definition.

```
function swap(a, posX, posY) {
    // Exchange the values at posX and posY in array a.
    const temp = a[posX];
    a[posX] = a[posY];
    a[posY] = temp;
}
```

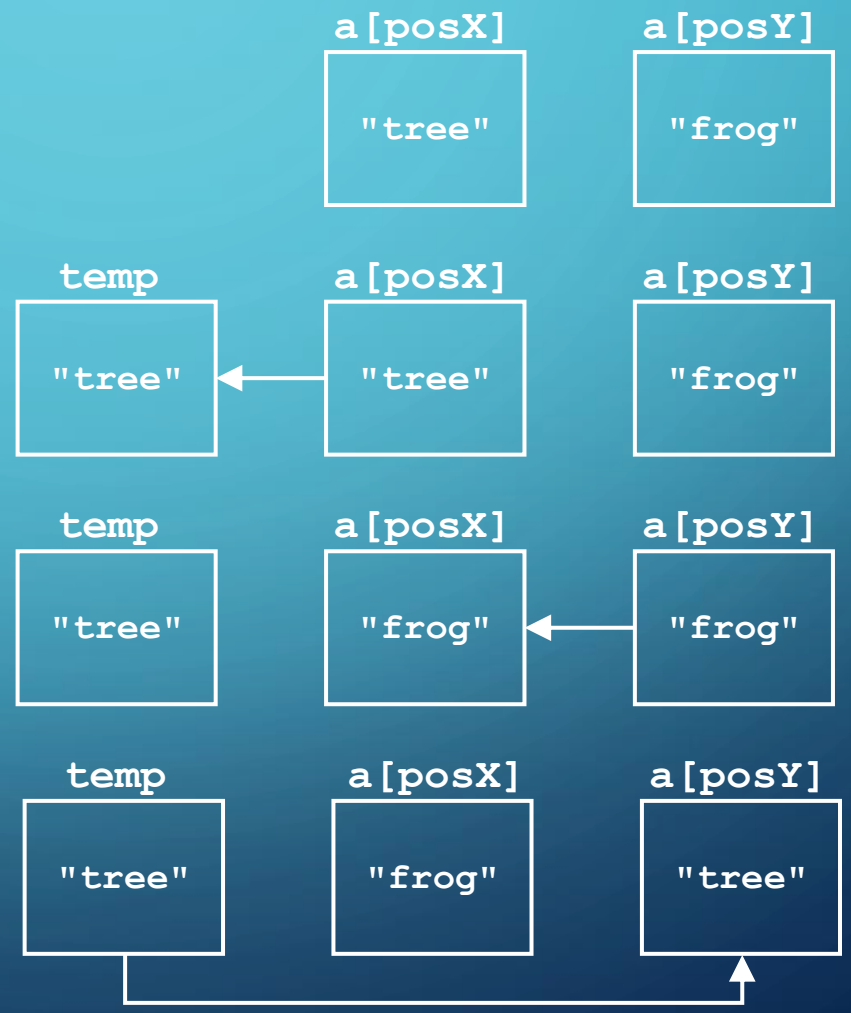
SWAPPING?

An illustration of an exchange of two string array element values as done by my **swap** function.

```
const temp = a[posX];
```

```
a[posX] = a[posY];
```

```
a[posY] = temp;
```



SWAPPING

- I can then call (execute) my **swap** function from my sort function like this:

```
function sort(a) {
    // Put elements of array a in ascending order.
    const n = a.length;
    let x = 0;
    while (x < n - 1) {
        let y = x + 1;
        while (y < n) {
            if (a[x] > a[y]) {
                swap(a, x, y);
            }
        }
    }
}
```

SWAPPING

- Alternatively, I can make use of a strange-looking 2015 addition to the JavaScript language. Ugly as it is, it does work, so I'll use this way of swapping and skip writing a swap function.

```
function sort(a) {
    // Put elements of array a in ascending order.
    const n = a.length;
    let x = 0;
    while (x < n - 1) {
        let y = x + 1;
        while (y < n) {
            if (a[x] > a[y]) {
                [a[x], a[y]] = [a[y], a[x]];
            }
        }
    }
}
```

swap a_x, a_y

INCREMENTING y

- Two things left to do: Incrementing **y** at the end of the **inner** loop…

```
function sort(a) {
    // Put elements of array a in ascending order.
    const n = a.length;
    let x = 0;
    while (x < n - 1) {
        let y = x + 1;
        while (y < n) {
            if (a[x] > a[y]) {
                [a[x], a[y]] = [a[y], a[x]]
            }
            y = y + 1;
        }
    }
}
```

add 1 to y

INCREMENTING y

- … and then do the same for **x** at the end of the **outer** loop.

```
function sort(a) {
    // Put elements of array a in ascending order.
    const n = a.length;
    let x = 0;
    while (x < n - 1) {
        let y = x + 1;
        while (y < n) {
            if (a[x] > a[y]) {
                [a[x], a[y]] = [a[y], a[x]]
            }
            y = y + 1;
        }
        x = x + 1;
    }
}
```

add 1 to x

AND WE'RE DONE!

- Here's a link to the completed program now showing the array contents before and after they've been sorted. Obviously, there's no change for the empty list!

SEMANTICS, SYNTAX, AND RUNTIME ERRORS

SEMANTICS, SYNTAX, AND RUNTIME ERRORS

- Now we'll look at a few terms associated with programming that crop up regardless of the choice of programming language.

SEMANTICS

- The *meaning* of an expression, a statement, a function, or a whole program is its *semantics*.
- For example, the meaning of the statement

```
let i = 0;
```

is well defined in the context of a JavaScript program. It creates an integer `let` variable called `i` and initializes it to 0. Those are its semantics.

SEMANTICS

- On the other hand, the plus sign operator, +, is overloaded in JavaScript. **Its semantics is determined by the context in which it's used.** When used exclusively with numbers, the interpreter recognizes + as the addition operator, but if strings are involved in the expression, it becomes the concatenation operator.

SEMANTICS

- We judge the semantics of a whole statement by what it does and what, if any, variables it affects.
- The same applies for a function.

SEMANTICS

- As programmers, we occasionally make errors in semantics. In these cases, our program executes, but does not behave as intended. For example:
  - Original statement:

```
energy = mass * speedOfLight;
```

  - Correction:

```
energy = mass * speedOfLight * speedOfLight;
```

- Such errors are often hard to detect because they don't stop program execution.

SYNTAX

- To write programming code that meets the requirements of a given high-level language is to use correct *syntax*.
- If correct syntax is used in a program, that program can be compiled, or interpreted, depending on the language used.
- A syntax error would prevent compilation for compiled languages (like C) or the interpretation and execution of a statement in an interpreted language (like JavaScript).

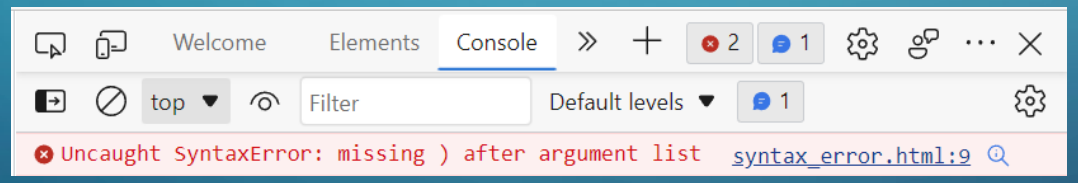
SYNTAX

- For example, this JavaScript statement contains a syntax error. It is missing the closing quotation mark from its string literal:

```
const userName = prompt("What is your name?);
```

```
const userName = prompt(What is your name?);
```

VS Code has spotted that this statement contains invalid JavaScript.



The same error identified using the Edge browser's debugging tool.

The line number containing the error.

RUNTIME ERRORS

- A runtime error occurs when an unanticipated situation causes a program to crash (that is, to stop unexpectedly).
- Examples:
  - A program attempts a division by zero. This causes a runtime error in most languages, but JavaScript simply renders a result of `infinity`.
  - A program attempts to read elements beyond the end of an array. Again, JavaScript just says that's `undefined`.
  - An interpreted language program attempts to execute a function that doesn't exist. (In a compiled language, such an error would prevent compilation.)

RUNTIME ERRORS

- An *exception* is an event that occurs during a program's execution that triggers a runtime error.
- *Exception handling* is done by a programmer who adds code to a program to anticipate and work around an exception.
- Special exception handling keywords and syntax for using them are common features of most programming languages today.

RUNTIME ERRORS

- JavaScript allows exception handling by way of several keywords including `try` and `catch`. I've used them in an example program on the next slide.

```
<html>
    <head>
        <title>Exception handling demo</title>
        <script>
            function fubar() {
                alert("Hello, world!");
            }

            try {
                farbu();
            }
            catch (e) {
                alert("Error: " + e.message);
            }
        </script>
    </head>
    <body>
    </body>
</html>
```

This example program defines a function called **fubar** but tries to call a function called **farbu**. Instead of crashing, it displays the message, "Error: farbu is not defined."

CODE LIBRARIES

CODE LIBRARIES: DEFINITION AND USE

- A *code library* is a collection of programming code (functions, special values, data structures) external to any program.
- A code library may be
  - in another file on the user's computer, or
  - elsewhere in the world on a remote server.

CODE LIBRARIES: DEFINITION AND USE

- A code library may be imported for use by a program at runtime, meaning that the functions and other elements it provides can be used by multiple programmers who had no hand in its creation and no knowledge of how it works. This makes code libraries invaluable in software development.

USING CODE LIBRARIES IN JAVASCRIPT

- Here are HTML opening and closing script tags that link to a JavaScript library stored in a file called my_lib.js. In this (very simple) example, the library file is in the same folder as the HTML file. The imported script contains only JavaScript.

```
<script src="my_lib.js"></script>
```

- Once such an import has been included, any other JavaScript may make use of functions defined in my_lib.js. For example, since JavaScript doesn't have a built-in **`hello`** function, we can assume from the following that it is a function defined in my_lib.js.

```
<script src="my_lib.js"></script>
<script>hello();</script>
```

USING CODE LIBRARIES IN JAVASCRIPT

- And here's an example that links to a library file called bar.js on some remote server:

```
<script src="https:/fu.com/lib/bar.js"></script>
```

- Note that there is no such remote library as that named above, but real examples include
  - remote math libraries for JavaScript and other languages,
  - effects libraries for websites for doing things like slide shows, and
  - maps services (like Google's) built into websites.

APPLICATION PROGRAMMING INTERFACES

- An *application programming interface* (*API*) is that part of a code library that is available for use by program developers.
- For example, a code library likely has a list of documented functions that programmers using the library can invoke from their programs. Those functions are part of the code library's API.

APPLICATION PROGRAMMING INTERFACES

- On the other hand, the code library will likely also contain functions (and other things) that it uses internally to do its work but that are *not* available to the remote programmer. These are not part of the API.
- You may think of the API as representing the visible or *public* part of a code library and the rest as hidden or *private*.

SUMMARY

IN THIS UNIT WE LOOKED AT

- How an algorithm expressed in pseudocode might be turned into high-level language code function (using a JavaScript example)
- How that function might be tested using a program framework
- More JavaScript:
  - Arrays and array subscripting
  - Conditional (`if`) statements
- Semantics and semantics errors
- Syntax and syntax errors

IN THIS UNIT WE LOOKED AT

- Runtime errors
- Exceptions and exception handling
- Code libraries (with examples in JavaScript)
- Application programming interfaces (APIs)