CSE 331
Intro to the Browser
Katherine Murphy
“Engineers are paid to think and understand.”
— Class slogan #1

“Reasoning is not optional: either reason up front or debug and then reason.”
— Class slogan #2

“Programmers overestimate the importance of efficiency and underestimate the difficulty of correctness.”
— Class slogan #3
Learning a New Language

• We’re going to learn JavaScript and TypeScript

• The second language can be the hardest to learn!
  – some things you took for granted no longer hold
  – must slow down think about think about every step

• We want to move slowly and deliberately
  – understand exactly what the type system does and does not do
    “Engineers are paid to think and understand”
  – comparison with Java will be useful
Programming for the Browser

• Will give an overview of JavaScript (JS) & TypeScript (TS)

• Both languages can be run in different environments
  – command line (like Java)
  – inside the browser

• Primarily interesting because of the browser
  – neither language would be used much otherwise
  – command line provided so you can use one language for both

• In both environments, print output with `console.log(..)`
  – prints to command line or “Developer Console” in the browser
Programming for the Browser

- JavaScript is the lingua franca of web browsers

- Previously, other languages were tried in the browser
  - Java was used but is no longer supported
  - Flash was used but is no longer supported
  - Google’s “dart” language is still around (probably)

- Now, other languages used by compiling into JavaScript
  - TypeScript used this way
  - Java can be compiled to JS (but it’s not great)
    you can’t really get around needing to learn JS
Learning a New Language

• No rush to learn the whole language
  – we will start with a small subset of its features
  – we won’t use all the language this quarter
  – have all quarter to get more familiar with JS and TS

• The second language can be the hardest to learn!
  – some things you took for granted no longer hold
  – must slow down think about think about every step

• We want to move slowly and deliberately
  – understand exactly what the type system does and does not do
    “Engineers are paid to think and understand”
  – comparison with Java will be useful
JavaScript
History of JavaScript

• Incredibly simple language
  – created in 10 days by Brendan Eich in 1995
  – often difficult to use because it is so simple

• Features added later to fix problem areas
  – imports (ES6)
  – classes (ES6)
  – integers (ES2020)
Relationship to Java

• Initially had no relation to Java
  – picked the name because Java was popular then
  – added Java’s Math library to JS also
    e.g., Math.sqrt is available in JS, just like Java
  – copied some of Java’s String functions to JS string
    e.g., s.charCodeAt(3) is available in JS, just like Java

• Both are in the “C family” of languages
  – much of the syntax is the same
  – more differences in data types

• We will discuss syntax (code) first and then data...
JavaScript Syntax

• Both are in the “C family” of languages

• Much of the syntax is the same
  – most expressions (+, -, *, /, ?, :, function calls, etc.)
  – control flow (if, for, while, break, continue, return)
  – comments with // or /* .. */

• Different syntax for a few things
  – declaring variables
  – declaring functions
  – equality (===)
Java vs JavaScript Syntax

• The following code is legal in both languages:
  – assume “s” and “j” are already declared

```
s = 0;
j = 0;
while (j < 10) {
    s += j;
    j++;
}

// Now s == 45
```
Differences from Java: Type Declarations

• JavaScript variables have no declared types
  – this is a problem... we will get them back in TypeScript

• Declare variables in one of these ways:

  ```javascript
  const x = 1;
  let y = "foo";
  ```

  – “const” cannot be changed; “let” can be changed
  – use “const” whenever possible!

• Also affects function argument declarations
  – more on this later...
Differences from Java: “===” operator

• JavaScript’s “==” is problematic
  – tries to convert objects to the same type
e.g., 3 == “3” and even 0 == “” are... true?!?

• We will use “===” (and “!==”) instead:
  – no type conversion will be performed
e.g., 3 === “3” is false

• Mostly same as Java
  – compares values on primitives, references on objects
  – but strings are primitive in JS (no .equals needed)
    == on strings common source of bugs in Java
Basic Data Types of JavaScript

- JavaScript includes the following runtime types

  - `number`
  - `bigint`
  - `string`
  - `boolean`
  - `null`
  - `undefined` (another null)
  - `Object`
  - `Array` (special subtype of Object)

  we won’t use them until week 5/6
Numbers

- bigint
- integers
- number
- floating point (like Java double)

- By default, JS uses number not bigint
  - 0, 1, 2 are numbers not integers
  - add an “n” at the end for integers (e.g., 2n)

- All the usual operators: + - * / ++ -- += ...

- Math library largely copied from Java
  - e.g., Math.sqrt returns the square root
Strings

- Mostly the same as Java
  - immutable
  - string concatenation with “+”

- A few improvements
  - string comparison with “===” and “<”
    no need for `s.equals(t)`... just write `s === t`
  - use either ‘...’ or ”...” (single or double quotes)
  - new string literals that support variable substitution:

```
const name = "Fred";
console.log(`Hi, ${name}!`);    // prints “Hi, Fred!”
```
Boolean

• All the usual operators: `&&` `||` `!`

• “if” can be used with any value
  – “falsey” things: `false`, `0`, `NaN`, `""`, `null`, `undefined`
  – “truthy” things: everything else

• A common source of bugs...
  – best to stick to boolean values
Record Types

• JavaScript “Object” is something with “fields”

• JavaScript has special syntax for creating them

```javascript
const p = {x: 1n, y: 2n};
console.log(p.x); // prints 1n
```

• The term “object” is potentially confusing
  – used for many things
  – I prefer it as shorthand for “mathematical object”

• Will refer to things with fields as “records”
Record Types

- Quotes are **optional** around field names
  
  ```javascript
  const p = {x: 1n, y: 2n};
  console.log(p.x); // prints 1n
  
  const q = {"x": 1n, "y": 2n};
  console.log(q.x); // also prints 1n
  ```

- Field names are literal strings, not expressions!
  
  ```javascript
  const x = "foo";
  console.log({x: x}); // prints {"x": "foo"}
  ```
## Checking Types at Run Time

<table>
<thead>
<tr>
<th>Condition</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>x is undefined</td>
<td><code>x === undefined</code></td>
</tr>
<tr>
<td>x is null</td>
<td><code>x === null</code></td>
</tr>
<tr>
<td>x is a number</td>
<td><code>typeof x === &quot;number&quot;</code></td>
</tr>
<tr>
<td>x is an integer</td>
<td><code>typeof x === &quot;bigint&quot;</code></td>
</tr>
<tr>
<td>x is a string</td>
<td><code>typeof x === &quot;string&quot;</code></td>
</tr>
<tr>
<td>x is an object or array (or null)</td>
<td><code>typeof x === &quot;object&quot;</code></td>
</tr>
<tr>
<td>x is an array</td>
<td><code>Array.isArray(x)</code></td>
</tr>
</tbody>
</table>

Hard to check if \( x \) is a specific record type at runtime. Much easier to let the type checker do this!

We will need to check types ourselves later in the course... when we start sending data between client and server.
Functions

- Functions are first class objects
  - “arrow” expressions creates functions
  - store these into a variable to use it later

```javascript
const add2 = (x, y) => x + y;
console.log(add2(1n, 2n)); // prints 3n

const add3 = (x, y, z) => {
    return x + y + z;
};
console.log(add3(1n, 2n, 3n)); // prints 6n
```
Functions

• We will declare functions like this

```javascript
const add = (x, y) => {
    return x + y;
};
```

// add(2n, 3n) == 5n

• Functions can be passed around
  – “functional” programming language
  – but we won’t do that (much) this quarter
    we will pass functions to buttons to tell them what to do when clicked
    see CSE 341 for more on that topic
Ternary Operator

- JS includes C’s ternary operator ("if" expression):

```javascript
console.log(x >= 0 ? x : y);  // prints x if x >= 0
//        y if x < 0
```

- Can be replaced with an “if” + function call:

```javascript
const f = (x, y) => {
    if (x >= 0) {
        return x;
    } else {
        return y;
    }
};

console.log(f(x, y));
```
TypeScript
TypeScript Adds Declared Types

• TypeScript includes declared types for variables

• Compiler checks that the types are valid
  – extremely useful!
  – produces JS just by removing the types

• Critical to understand what the type system does
  – not enough just to make the type errors go away
  – need to understand exactly what bugs the type system catches
    you do not need to worry about those
    but you are responsible for everything else
TypeScript Adds Declared Types

• Type is declared after the variable name:

```typescript
const u: bigint = 3n;
const v: bigint = 4n;

const add = (x: bigint, y: bigint): bigint => {
    return x + y;
};
```

```javascript
console.log(add(u, v)); // prints 7n
```

– return type is declared after the argument list (…) and before =>

• “Where types go” is the main syntax difference vs Java
  – other key differences are functions (=>) and equality (===)
Basic Data Types of TypeScript

• JavaScript includes the following types

  - number
  - bigint
  - string
  - boolean
  - null
  - undefined
  - Object (record types)
  - Array (e.g., string[] as in Java)

• TypeScript has these and also...

  - unknown (could be anything)
  - any (turns off type checking — do not use!)
Literal Types

• Any literal value is also a type:

```javascript
let x: "foo" = "foo";
let y: 16n = 16n;
```

• Variable can only hold that specific value!
  – can assign it again, but only with the same value

• Seems silly, but turns out to be useful...
Ways to Create New Types in TypeScript

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

• **Not possible in Java!**  
  - TS can describe types of code that Java cannot

• **Unknown type is (essentially) a union**
  
  ```typescript
  type unknown = number | bigint | string | boolean | ...
  ```
Enumerations

- Use **unions of literals** are “enums”

```typescript
const dist = (dir: "left"|"right", amt: bigint): bigint => {
  if (dir === "right") {
    return amt;
  } else {
    return -amt;
  }
};
```

- TypeScript ensures that callers will only pass one of those two strings (“left” or “right”)
  - impossible to do this in Java
    (must fake it with the enumeration **design pattern**
Ways to Create New Types in TypeScript

• Can create **compound** types in multiple ways
  – put multiple types together into one larger type

• **Record Types**  
  
  
  `{x: bigint, s: string}`
  – anything with *at least* fields “x” and “s”

```javascript
const p: {x: bigint, s: string} = {x: 1n, s: 'hi'};
console.log(p.x);  // prints 1n
```
Ways to Create New Types In TypeScript

- Can create *compound* types in multiple ways
  - put multiple types together into one larger type

- Tuple Types  
  - create them like this

```javascript
const p: [bigint, string] = [1n, 'hi'];
```

- give names to the parts (“destructuring”) to use them

```javascript
const [x, y] = p;
console.log(x); // prints 1n
```

- at runtime, this is an array of length 2 (JS has no tuples)
  but do not actually use them like arrays, e.g., no array indexing
Records vs Tuples

• Records and tuples provide the same functionality
  – both allow you to put parts together into one object
  – conceptually interchangeable

• They differ in who names the parts and when
  – record: creator picks the names
    everyone must use the same name
  – tuple: user of the tuple picks the names
    each user can pick their own names
Type Aliases

- TypeScript lets you give shorthand names for types

```typescript
type Point = {x: bigint, y: bigint};

const p: Point = {x: 1n, y: 2n};
console.log(p.x); // prints 1n
```

- Usually nicer but not necessary
  - e.g., this does the same thing

```typescript
const p: {x: bigint, y: bigint} = {x: 1n, y: 2n};
console.log(x); // prints 1n
```
Structural vs Nominal Typing

• Deeper difference between TypeScript and Java
  – records aren’t just a quick way to describe a class

• TypeScript uses “structural typing”
  – sometimes called “duck typing”
    “if it walks like a duck and quacks like a duck, it’s a duck”

```typescript
type T1 = {a: bigint, b: string};
type T2 = {a: bigint, b: string};

const x: T1 = {a: 1n, b: "two"};
```

– can pass “x” to a function expecting a “T2”!
Structural vs Nominal Typing

• Java uses “nominal typing”

```java
class T1 { int a; int b; }
class T2 { int a; int b; }

T1 x = new T1();
```

– cannot pass “x” to a function expecting a “T2”

• Libraries do not interoperate unless it was pre-planned
  – create “adapters” to work around this
    example of a design pattern used to work around language limitations
Type Inference

• If you leave off the type, TS will try to guess it
  – often, but not always, it guesses correctly

• This will work fine

  ```javascript
  const p = {x: ln, y: 2n};
  console.log(p.x);  // prints 1n
  ```
  
  – compiler should correctly guess `{x: bigint, y: bigint}`
  – can see in VSCode by hovering over “p”
Type Inference

• If you leave off the type, TS will try to guess it
  – often, but not always, it guesses correctly

• In 331, type declarations are required on
  – function arguments and return values
  – variables declared outside of any function (“top-level”) these could be exported, so types should be explicit

• We do not require declarations on local variables
  – but it is fine to include them
  – if TS guesses wrong, you will need to include it
**Imports**

- **JS / TS code can now be split into multiple files**
  - JS didn’t initially have that feature before ES6

- By default, declarations are hidden outside the file

- Add the keyword “export” to make it visible
  ```
  export const MAX_NUMBER = 15; // in src/foo.ts
  ```

- Use the “import” statement to bring into another file
  ```
  import { MAX_NUMBER } from './foo'; // in src/bar.ts
  ```
  - ‘./foo’ is relative path from this file to foo.ts
  - extension (.ts) is not included
Imports

```ts
export const MAX_NUMBER = 15; // in src/foo.ts

import { MAX_NUMBER } from './foo'; // in src/bar.ts
```

• For code you write, you will only need this syntax

• JS / TS includes other ways of importing things
  – full explanation is very complicated
  – don’t worry about it...

• Starter code will include some that look different, e.g.:

```ts
import React, { Component } from 'react';

import './foo.png'; // include a file along with the code
```
Linters
Linters

- Linters are like type checkers
  - try to find potential bugs in the program
  - as well as poor style / design issues

- In 331, we have our own linter ("comfy-tslint")
  - tries to check that our coding conventions are followed
  - e.g., types are declared except local vars in functions

- They can be overzealous
  - can flag issues that aren’t really problems
  - (happens with type checkers also, but less frequently)
JS Linting

• Linter will complain about unused variables

```javascript
const f = (a: bigint, b: bigint): bigint => {
    return b;
};
```

– linter will complain that `a` is unused
  this looks suspicious, doesn’t it?

• This ignores variables whose names start with “_”
  – the underscore indicates you know it is unused
  – change the variable name to get rid of the error
Browsers
Browser Operation

- Browser reads the URL to find what HTML to load

![URL component diagram]

- Contacts the given server and asks for the given path

![Request and response diagram]
• **Page can include JavaScript to execute**
  – each reload of the page re-executes the code

• **Simple demo...**
  – *ex1.html* invokes *ex1.js*
Running Code in the Browser

- Our code runs each time the page loads

- We can produce output with `console.log`
  - we will do fancier output shortly

- What about input?
  - easiest way to start is to pass input in the URL
URL Parts

• URLs have more parts than just server and path:

https://mail.google.com/mail/u/0/?zx=ABCD#inbox

server name  path  search  fragment

• Path identifies the file to load from the server

• Search string is the input passed to that file when run

• (Fragment will not be important for us.)
URL Parts

https://mail.google.com/mail/u/0/?zx=ABCD#inbox

- server name: mail.google.com
- path: /mail/u/0/
- search: ?zx=ABCD
- fragment: #inbox

- Get access to the URLs parts in `window.location`

  - `window.location.hostname` = "mail.google.com"
  - `window.location.pathname` = "/mail/u/0"
  - `window.location.search` = "?zx=ABCD"
  - `window.location.hash` = "#inbox"
Query Parameters

• Search string can pass multiple values at once
  – we call these “query parameters”

• Each parameter is of the form “name=value”
  – no spaces around the “=”

• Multiple values are placed together with “&”s in between

  ?a=3&b=foo&c=Kevin%20Z

  – encodes three query parameters: a is “3”, b is “foo”, c is “Kevin Z”
  – note that all values are strings
  – note that special characters (like spaces) are encoded
    the `encodeURIComponent` function does this for us
### Query Parameters

?a=3&b=foo&c=Kevin%20Z

- **Not necessary to write code to parse query params**
  - take advantage of the **built-in** `URLSearchParams` class
    (important in general to take advantage of provided libraries)

- **Read query parameters from the URL like this:**

  ```javascript
  const params = new URLSearchParams(window.location.search);
  console.log(params.get("a"));  // prints “3” or null
  ```

- **Another demo...**
  - `ex2.ts` takes name and age as inputs
HTML
HTML

• **HTML = Hyper Text Markup Language**
  – text format for describing a document / UI
  – text describes what you want *drawn* in the browser

• **HTML text consists primarily of “tags” and text**
HTML Tags

```
<p> Some Text </p>

```

```
<p id="firstParagraph"> Some Text </p>

```
Elements Form a Tree

- Elements can have children (text or elements)
  - text is always a leaf in the tree

```html
<div>
  <p id="firstParagraph"> Some Text </p>
  <br>
  <div>
    <p>Hello</p>
  </div>
</div>
```
More on HTML

- HTML is a text format that describes a tree
  - nodes are elements or text

```html
<div>
  <p>Some text</p>
  <p>More text</p>
</div>
```

- HTML text is parsed into a tree (“DOM”)
- JS can access the tree in the variable “document”
  our code lives in the world on the right side
More on HTML

- Browser window displays an HTML document
  - tree is turned into drawing in the page

- browser displays (renders) the HTML in the window
  browsers *parse* and *render* very quickly

- **JS** has *limited* access to display information
Our code runs in the **middle** part. We can make changes to the tree. The browser will then re-render.
Web App UI (From Section)

- Browser will render any HTML included in server response

- Our servers will send a page that just executes our code
  - page is mostly *empty*

- How do we display HTML from our code?
  - need to make HTML
  - need to tell the browser to render it
Web App UI (From Section)

- Initial page has a placeholder in the HTML:

  ```html
  <div id="main"></div>
  ```

  (empty DIV in index.html)

- Put HTML into it from code like this:

  ```javascript
  const elem: HTMLElement | null = document.getElementById("main");
  if (elem !== null) {
    const root: Root = createRoot(elem);
    root.render(... /* some HTML */);
  }
  ```

  - `createRoot` is a function provided by the React library
    (more details on this later on...)
  - how do we create the HTML?
HTML Literals

• Extension of JS / TS allows HTML expressions
  – file extension must be .jsx (or .tsx for TS)

    ```
    const x = <p>Hi there!</p>;
    ```

• TypeScript will make sure the HTML is valid
  – will complain if it has unknown tags or attributes
  – will complain if attribute values have the wrong type
  – these checks are very useful!
HTML Literals

• Supports substitution like `...` string literals,
  – but uses `{...}` not `${...}`

```javascript
const name = "Fred";
return <p>Hi {name} </p>;
```

• Can also substitute the value of an attribute:

```javascript
const rows = 3;
return (<textarea rows={rows} cols="25">
  initial text here
</textarea>);
```

initial text here
Styling

• The “style” attribute controls appearance details
  – margins, padding, width, fonts, etc.
  – see an HTML reference for details (when necessary)

• Attribute value can include many properties
  – each is “name: value”
  – separate multiple using “;”

```html
<p>Hi,
   <span style="color: red; margin-left: 15px">Bob</span>!
</p>
```

Hi,  Bob!

– we will generally not worry much about looks in this class...
Calculating the Style

• How do we calculate part of the style in code?
  – you might think this would work

```javascript
const n = 15;
...
<p>Hi,
  <span style={`color: red; margin-left: ${n}px`}>Bob</span>
</p>
```

– but it does not type check!
– the type of the “style” attribute is not string
Style Attribute in JSX

• The type of the style attribute is a record!
  – more accurate type checking this way (string would have none!)

```javascript
const r = {color: "red", marginLeft: `${n}px`};
return <p> Hi, <span style={r}>Bob</span>!</p>;
```

• Field names differ slightly from HTML
  – JS doesn’t allow “–” in a field name
  – JS uses camelCapNames instead of camel-caps-names

• Looks weird, but record can be written in-line:

```javascript
return (<p> Hi,
  <span style={{color: "red"}}>&gt;Bob</span>!</p>);
```
JSX Gotchas

• Must wrap multi-line HTML literals with (...) 

```jsx
return (<p>This is some HTML that spans multiple lines</p>);
```

• Must have a single root tag (i.e., must be a tree) 
  – e.g., cannot do this 

```jsx
return <p>one</p><p>two</p>;
```
  – instead, wrap in a `<div>` or just `<> . . . </>` (“fragment”)

• Replacements for attributes matching keywords 
  – use “className=” instead of “class=” 
  – use “htmlFor=” instead of “for=”