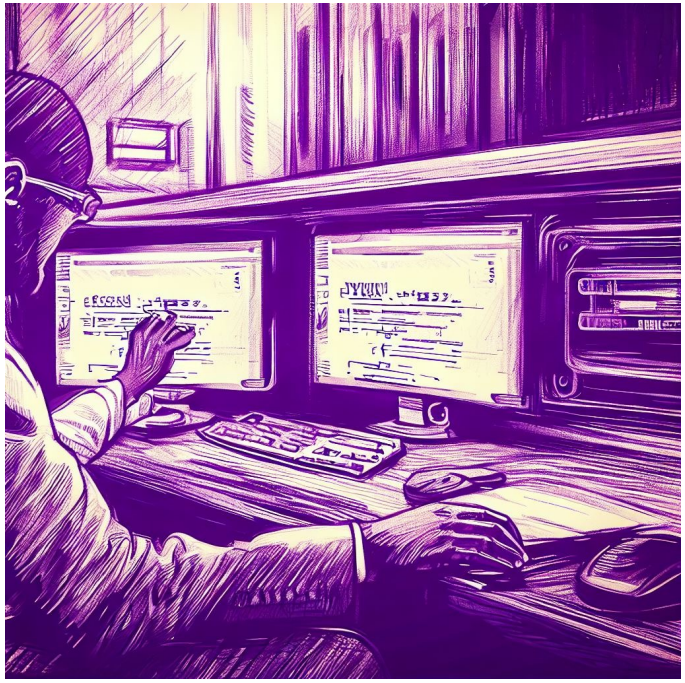


CSE 331

Intro to the Browser

Kevin Zatloukal

More TypeScript

Last Time: Ways to Create New Types

- **Union Types** `string | number`
  - 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

```
type unknown = number | string | boolean | null | …
```

Last Time: Ways to Create New Types

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

- **Record Types** `{x: number, s: string}`
  - anything with *at least* fields "x" and "s"

```
const p: {x: number, s: string} = {x: 1, s: 'hi'};
console.log(p.x);  // prints 1
```

Last Time: Ways to Create New Types

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

- Tuple Types `[number, string]`
  - at runtime, this is an array of length 2
  - create them like this

```
const p: [number, string] = [1, 'hi'];
```

  - give names to the parts to use them

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

Last Time: Type Aliases

- TypeScript lets you give shorthand names for types

```
type Point = {x: number, y: number};

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

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

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

Last Time: Structural vs Nominal Typing

- Java uses "**nominal typing**"

```
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

Last Time: 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"

```
type T1 = {a: number, b: number};
type T2 = {a: number, b: number};

const x: T1 = {a: 1, b: 2};
```

  - can pass " `x` " to a function expecting a " `T2` "!

Type Inference

- If you leave off the type, TS will try to <u>guess</u> it
  - often, but not always, it guesses correctly

- This will work fine

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

  - compiler should correctly guess `{x: number, y: number}`
  - can see in VS Code by <u>hovering</u> over "`p`"

Type Inference

- If you leave off the type, TS will try to <u>guess</u> 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")

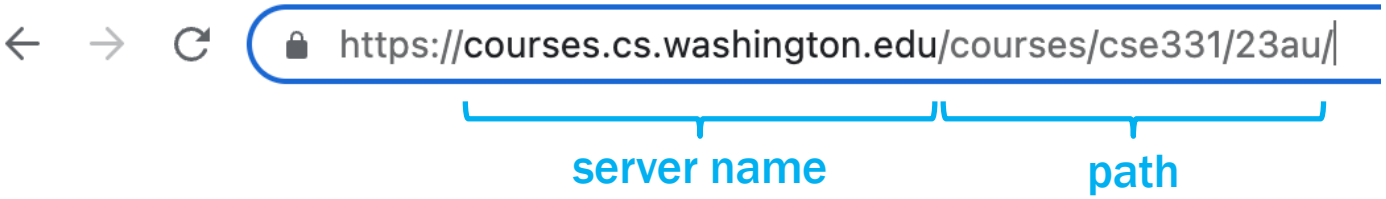
    these could be exported, so types should be explicit

- 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

Browsers

Last Time (section): Browser Operation

- Browser reads the URL to find what HTML to load



- Contacts the given server and asks for the given path

Last Time (section): Browser Operation

- **Tools come with its own server**
  - `npm run start` **starts that server for us**
  - **available at** `http://localhost:8080/`
  - **compiles the code and returns an HTML page**

Last Time (section): Query Parameters

- Talked about the query parameters in the URL
  - encoded in the "search string" in the form "`?a=b&c=d…`"
  - primary way we will provide input to our apps

- Read query parameters from the URL like this:

```
const params = new URLSearchParams(window.location.search);
console.log(params.get("a"));  // prints "b"
```

- `URLSearchParams` class is built into JavaScript

Last Time (section): Summary

Key points to understand for now

- Must execute `npm run start` to use your app
  - starts a server that will give your code to the browser
- Run your app in browser at "`http://localhost:8080/`"
  - browser gets the code from the server
- Code running in the browser gets its input from the URL
  - input is provided in the URL as query parameters "`?a=b&…`"
  - use `URLSearchParams` to read the query parameters

More details later in the quarter…

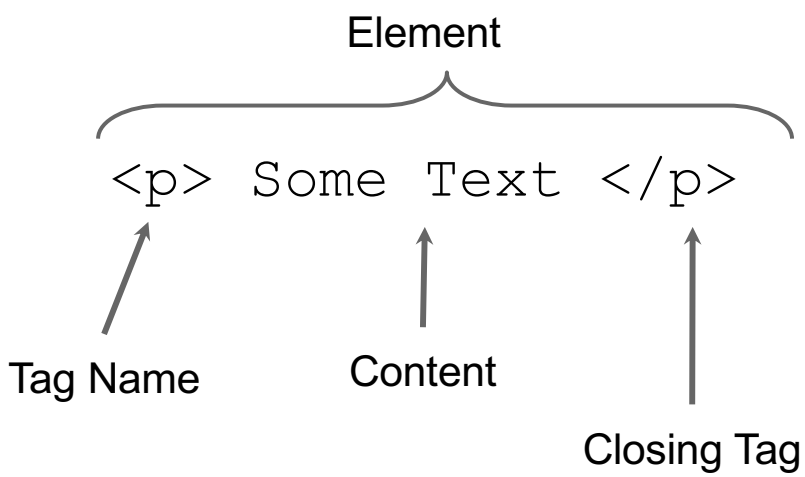
HTML

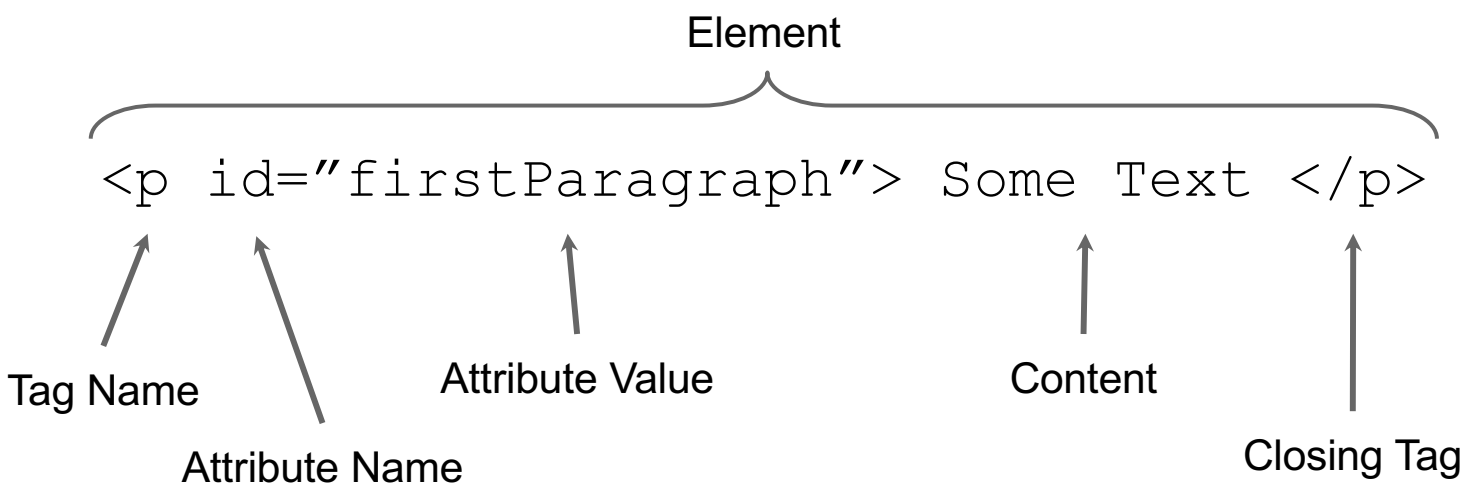
HTML

- 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

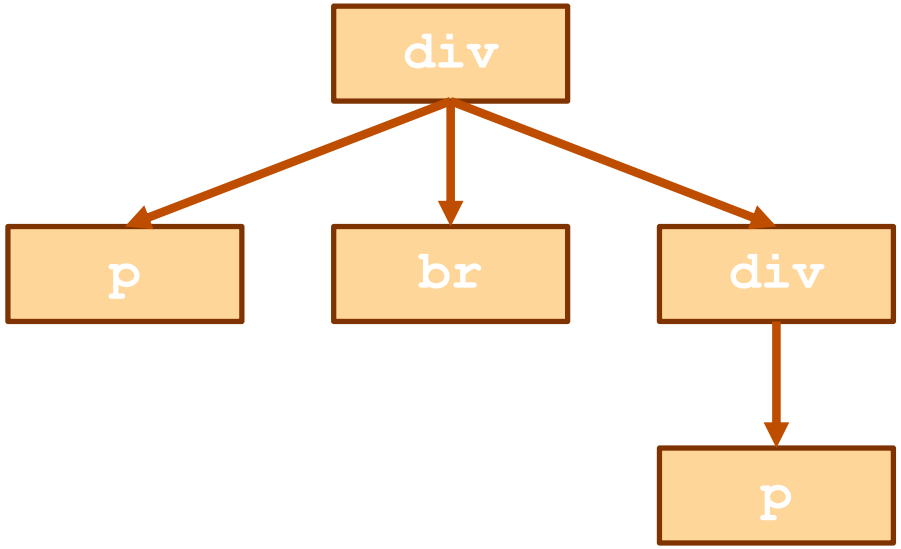

Element
 Some Text
Tag Name
Content
Closing Tag


Element
 Some Text
Tag Name
Attribute Name
Attribute Value
Content
Closing Tag

Elements Form a Tree

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

```
<div>
    <p id="firstParagraph"> Some Text </p>
    <br>
    <div>
        <p>Hello</p>
    </div>
</div>
```

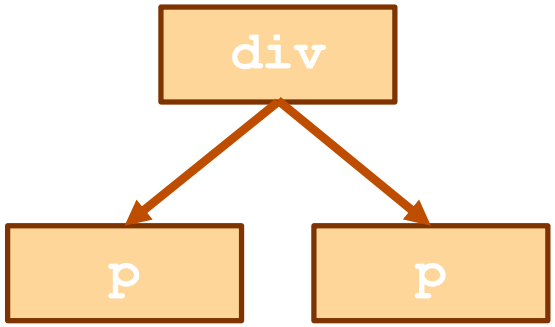


More on HTML

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

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

HTML text

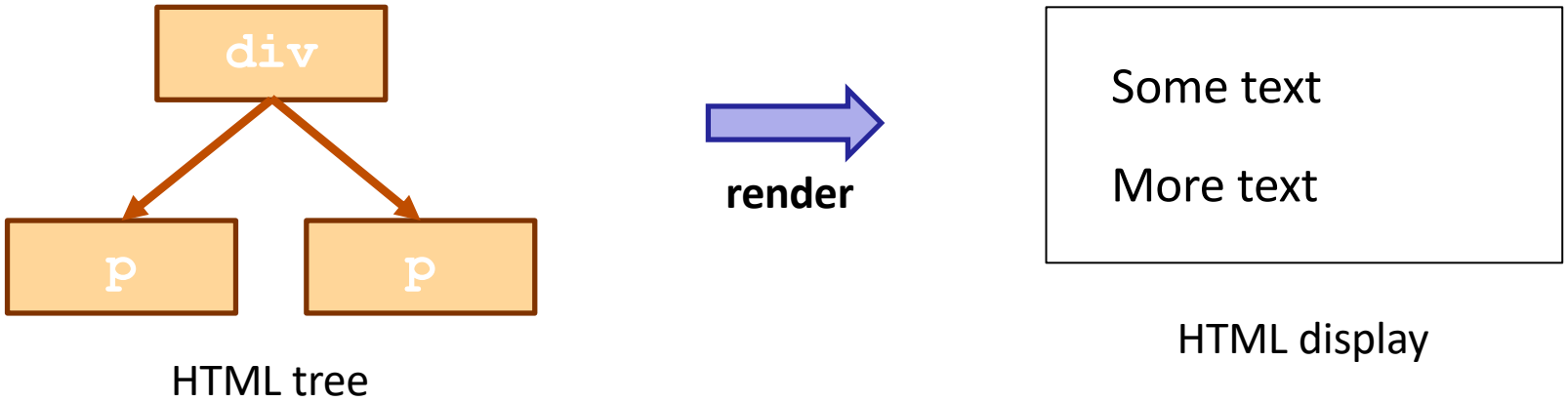


HTML tree

  - HTML text is <u>parsed</u> into a tree
  - JS can access the tree in the variable "`document`"

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**

Web App UI

- Browser will render any HTML included in server response
- Our server sends 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

- **Initial page has a placeholder in the HTML:**

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

(empty DIV in `index.html`)

- **Put HTML into it from code like this:**

```
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

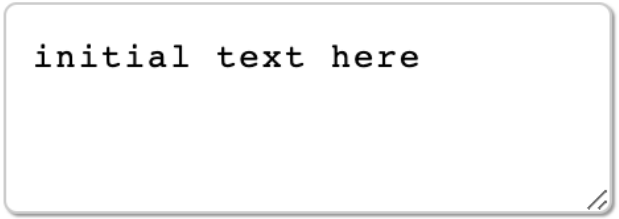
HTML Literals

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

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

- Can also substitute the value of an attribute:

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



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 “;”

```
<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

```
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!**

```
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:**

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

Cascading Style Sheets (CSS)

- Commonly used styles can be named
  - association of names to styles goes in a `.css` file

```
// foo.css
span.fancy { color: red; margin-left: 15px }

// foo.html
… <p>Hi, <span class="fancy">Bob</span></p> …
```

- Useful to avoid repetition of styling
  - makes it easier to change

Cascading Style Sheets (CSS)

- CSS styling can be used in JSX / TSX as well

```
// foo.css
span.fancy { color: red; margin-left: 15px }

// foo.tsx
import ’./foo.css’;  // another weird import
…
return <p>Hi, <span className=“fancy”>Bob</span>!</p>;
```

- Nice to get this out of the source code
  - usually not the programmers who need to change it

JSX Gotchas

- Must wrap multi-line HTML literals with `( . . )`
- Must have a single root (a tree)
  - e.g., cannot do this

```
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=`"