EECS 368
Programming Language Paradigms

David O. Johnson

Fall 2022

Reminders

- Assignment 4 due: 11:59 PM, Monday, October 17
- Assignment 5 due: 11:59 PM, Monday, October 31

Any Questions?

In-Class Problem Solution

- 18-(10-5) In-Class Problem Solution.pptx

Any Questions?

Chapter 18 - HTTP and Forms

- ~~The protocol~~
- ~~Browsers and HTTP~~
- ~~Fetch~~
- ~~HTTP sandboxing~~
- ~~Appreciating HTTP~~
- ~~Security and HTTPS~~
- ~~Form fields~~
- ~~Focus~~
- ~~Disabled fields~~
- ~~The form as a whole~~
- ~~Text fields~~
- ~~Checkboxes and radio buttons~~
- ~~Select fields~~
- File fields
- Storing data client-side

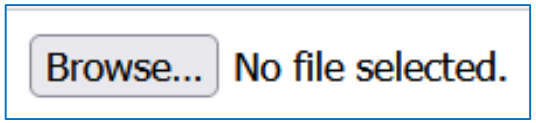
File Fields

- File fields were originally designed as a way to upload files from the user's machine through a form.
- In modern browsers, they also provide a way to read such files from JavaScript programs.
- The field acts as a kind of gatekeeper.
- The script cannot simply start reading private files from the user's computer, …
- but if the user selects a file in such a field, …
- the browser interprets that action to mean that the script may read the file.

File Field

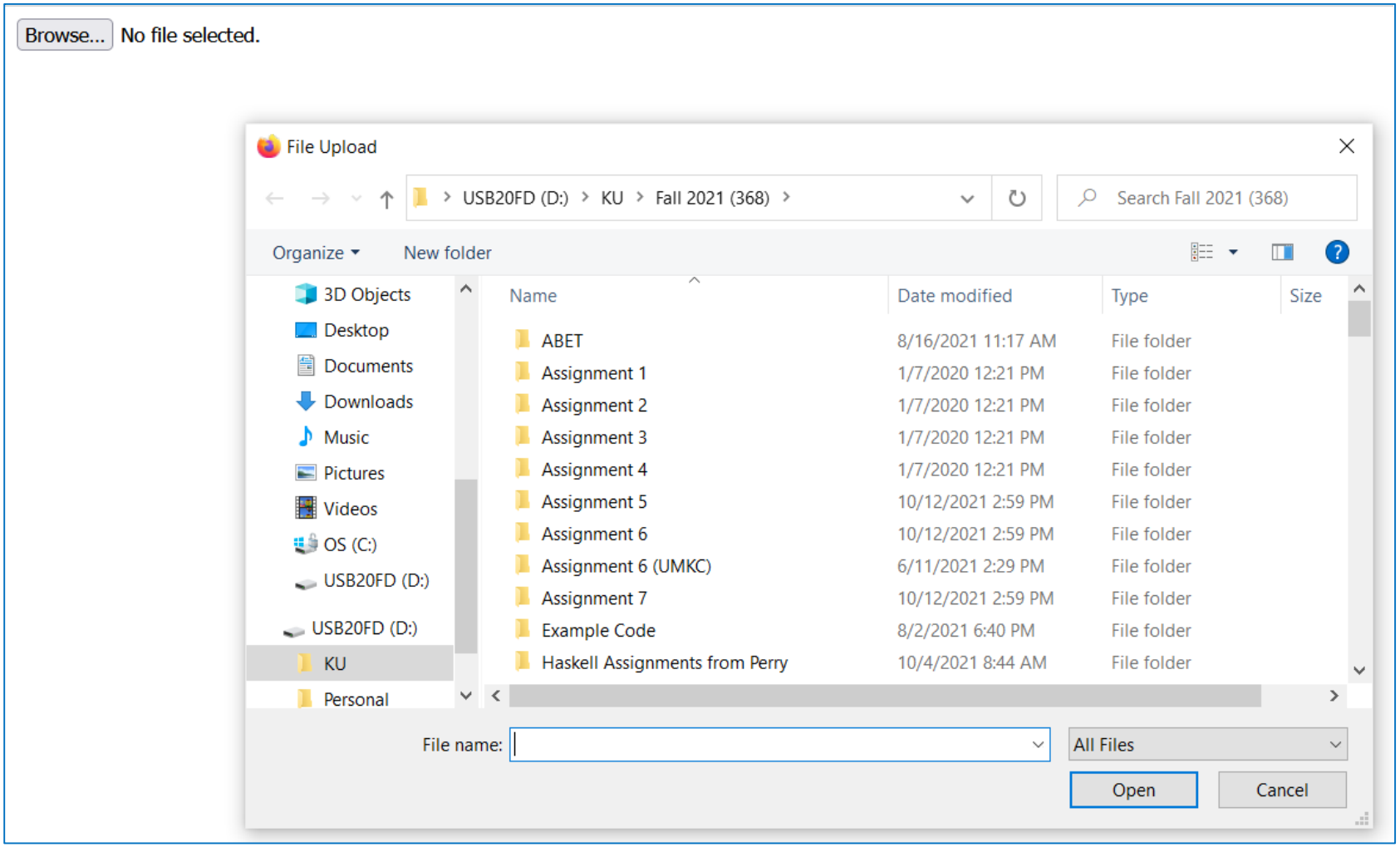
```
<input type="file">
<script>
let input = document.querySelector("input");
input.addEventListener("change", () => {
if (input.files.length > 0) {
let file = input.files[0];
console.log("You chose", file.name);
if (file.type) console.log("It has type", file.type);
}
});
</script>
```

- A file field usually looks like a button labeled with something like:
- “choose file” or…
- “browse” …



File Field

When you click on the button a window opens which lets you browse your files.



File Field

```
<input type="file">
<script>
let input = document.querySelector("input");
input.addEventListener("change", () => {
if (input.files.length > 0) {
let file = input.files[0];
console.log("You chose", file.name);
if (file.type) console.log("It has type", file.type);
}
});
</script>
```

- And then shows you which file you selected.

Browse... 36-(11-19) More JavaScript & HTML Forms.pptx

files Property

```
<input type="file">
<script>
let input = document.querySelector("input");
input.addEventListener("change", () => {
if (input.files.length > 0) {
let file = input.files[0]; //get first file in files
console.log("You chose", file.name);
if (file.type) console.log("It has type", file.type);
}
});
</script>
```

- The files property of a file field element is an array-like object (again, not a real array) containing the files chosen in the field.
- It is initially empty.
- The reason there isn't simply a file property is that file fields also support a multiple attribute, which makes it possible to select multiple files at the same time.
- Objects in the files object have properties such as:
  - name (the filename)
  - size (the file's size in bytes)
  - type (the media type of the file, such as text/plain or image/jpeg)

Console Output:

```
You chose 36-(11-19) In-Class Problem Solution.pptx
It has type application/vnd.openxmlformats-officedocument.presentationml.presentation
```

Any Questions?

Reading a File

- What the files property does not have is a property that contains the content of the file.
- Getting at that is a little more involved.
- Since reading a file from disk can take time, the interface must be asynchronous to avoid freezing the client's browser.

Reading a File

- Reading a file is done by creating a FileReader object, …
- registering a "load" event handler for it, …
- calling its readAsText method, and …
- giving it the file we want to read.
- Once loading finishes, the reader's result property (reader.result) contains the file's content.

```
<input type="file" multiple>
<script>
  let input = document.querySelector("input");
  input.addEventListener("change", () => {
    for (let file of Array.from(input.files)) {
      let reader = new FileReader();
      reader.addEventListener("load", () => {
        console.log("File", file.name, "starts with", reader.result.slice(0, 20));
      });
      reader.readAsText(file);
    }
  });
</script>
```

slice Method

```
console.log("File", file.name, "starts with", reader.result.slice(0, 20));
```

slice(start, end)

- **start** - Optional
  - Zero-based index at which to start extraction.
  - A negative index can be used, indicating an offset from the end of the sequence.
    - slice(-2) extracts the last two elements in the sequence.
  - If start is undefined, slice starts from the index 0.
  - If start is greater than the index range of the sequence, an empty array is returned.
- **end** - Optional
  - Zero-based index **before** which to end extraction.
    - **slice extracts up to but not including end**.
    - For example, slice(1,4) extracts the second element through the fourth element (elements indexed 1, 2, and 3).
  - A negative index can be used, indicating an offset from the end of the sequence.
    - slice(2,-1) extracts the third element through the second-to-last element in the sequence.
  - If end is omitted, slice extracts through the end of the sequence (arr.length).
  - If end is greater than the length of the sequence, slice extracts through to the end of the sequence (arr.length).

In the example above it extracts bytes 0-19.

Reading a File

```
<input type="file" multiple>
<script>
  let input = document.querySelector("input");
  input.addEventListener("change", () => {
    for (let file of Array.from(input.files)) {
      let reader = new FileReader();
      reader.addEventListener("load", () => {
        console.log("File", file.name, "starts with", reader.result.slice(0, 20));
      });
      reader.readAsText(file);
    }
  });
</script>
```

Browse... Chapter 18 - Reading a File.html

Console Output:

File Chapter 18 - Reading a File.html starts with
<!doctype html>
<htm

Note:

- .slice extracts 20 bytes
- (including \n after <!doctype html>)
- with indices 0-19

Reading a File Error

- FileReaders also fire an "error" event when reading the file fails for any reason.
- The error object itself will end up in the reader's error property.
- This interface was designed before promises became part of the language.
- You could wrap it in a promise like this:

```
function readFileText(file) {
  return new Promise((resolve, reject) => {
    let reader = new FileReader();
    reader.addEventListener(
      "load", () => resolve(reader.result));
    reader.addEventListener(
      "error", () => reject(reader.error));
    reader.readAsText(file);
  });
}
```

Any Questions?

Storing Data Client-Side

- Simple HTML pages with a bit of JavaScript can be a great format for “mini applications”.
- Small helper programs that automate basic tasks.
- By connecting a few form fields with event handlers, …
- you can do anything from converting between centimeters and inches …
- to computing passwords from a master password and a website name.

- When such an application needs to remember something between sessions you cannot use JavaScript bindings.
- Those are thrown away every time the page is closed.
- You could set up a server, connect it to the Internet, and have your application store something there.
- We will see how to do that later with Node.js.
- But that’s a lot of extra work and complexity.
- Sometimes it is enough to just keep the data in the browser.

Storing Data Client-Side

- The localStorage object can be used to store data in a way that survives page reloads.
- This object allows you to file string values under names.

```
localStorage.setItem("username", "marijn");
console.log(localStorage.getItem("username"));
// → marijn
localStorage.removeItem("username");
```

- A value in localStorage sticks around until it is overwritten, …
- it is removed with removeItem, or …
- the user clears their local data.

- Sites from different domains get different storage compartments.
- That means data stored in localStorage by a given website can, in principle, be read (and overwritten) only by scripts on that same site.

- Browsers do enforce a limit on the size of the data a site can store in localStorage.
- That restriction, along with the fact that filling up people's hard drives with junk is not really profitable, prevents the feature from eating up too much space.

Storing Data Client-Side

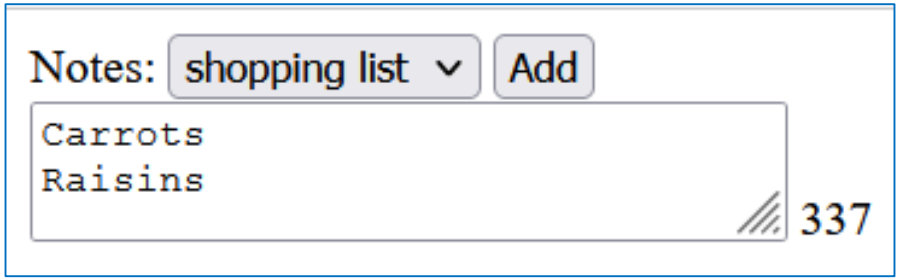
- There is another object, similar to localStorage, called sessionStorage.
- The difference between the two is that the content of sessionStorage is forgotten at the end of each session, …
- which for most browsers means whenever the browser is closed.

Storing Data Client-Side Example

- The following example implements a crude note-taking application.
- It keeps a set of named notes and allows the user to edit notes and create new ones.

Here's the HTML to create the form:

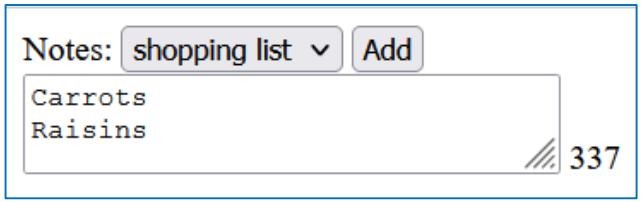
```
Notes: <select></select>
<button>Add</button><br>
<textarea style="width: 100%"></textarea>
```



Storing Data Client-Side Example

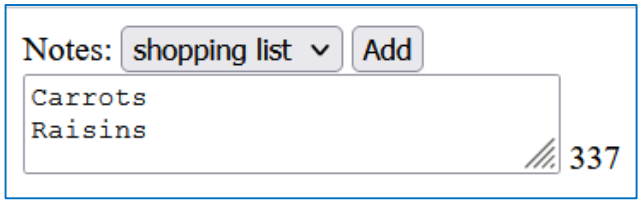
- The setState function makes sure the DOM is showing a given state and stores the new state to localStorage.

```
<script>
let list = document.querySelector("select");
let note = document.querySelector("textarea");
let state;
function setState(newState) {
   list.textContent = "";
   for (let name of Object.keys(newState.notes)) {
      let option = document.createElement("option");
      option.textContent = name;
      if (newState.selected == name) option.selected = true;
      list.appendChild(option);
   }
   note.value = newState.notes[newState.selected];
   localStorage.setItem("Notes", JSON.stringify(newState));
   state = newState;
}
setState (JSON.parse(localStorage.getItem("Notes")) || {
   notes: {"shopping list": "Carrots\nRaisins"},
   selected: "shopping list"
});
…
```



Storing Data Client-Side Example

- The script gets its starting state from the "Notes" value stored in localStorage.
- If that is missing, it creates an example state that has only a shopping list in it.
- Reading a field that does not exist from localStorage will yield null.
- Passing null to JSON.parse will make it parse the string "null" and return null.
- Thus, the || operator can be used to provide a default value in a situation like this.




```
<script>
let list = document.querySelector("select");
let note = document.querySelector("textarea");
let state;
function setState(newState) {
  list.textContent = "";
  for (let name of Object.keys(newState.notes)) {
    let option = document.createElement("option");
    option.textContent = name;
    if (newState.selected == name) option.selected = true;
    list.appendChild(option);
  }
  note.value = newState.notes[newState.selected];
  localStorage.setItem("Notes", JSON.stringify(newState));
  state = newState;
}
setState (JSON.parse(localStorage.getItem("Notes")) || {
  notes: {"shopping list": "Carrots\nRaisins"},
  selected: "shopping list"
});
...
```

Storing Data Client-Side Example

- Event handlers call setState to move to a new state.

```
...
list.addEventListener("change", () => {
  setState({notes: state.notes, selected: list.value});
});
note.addEventListener("change", () => {
  setState({
    notes: Object.assign({}, state.notes,
                         {[state.selected]: note.value}),
    selected: state.selected
  });
});
...
```

Notes: shopping list Add

```
Carrots
Raisins
```

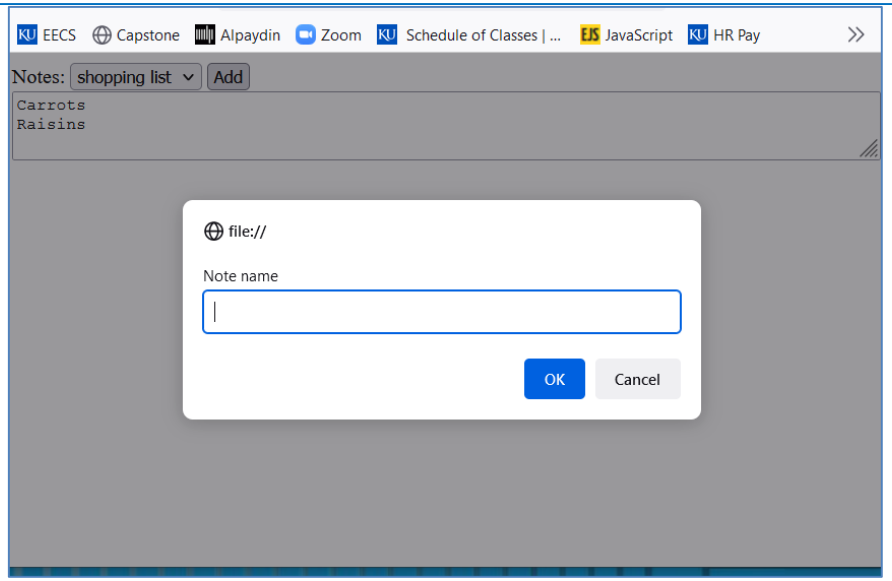
337

Storing Data Client-Side Example

```
...
document.querySelector("button")
  .addEventListener("click", () => {
    let name = prompt("Note name");
    if (name) setState({
      notes: Object.assign({}, state.notes, {[name]: ""}),
      selected: name
    });
  });
</script>
```

- Clicking the Add button causes a prompt …
- Which causes a new note to be created called name.

```
Notes: <select></select>
<button>Add</button><br>
<textarea style="width: 100%"></textarea>
```

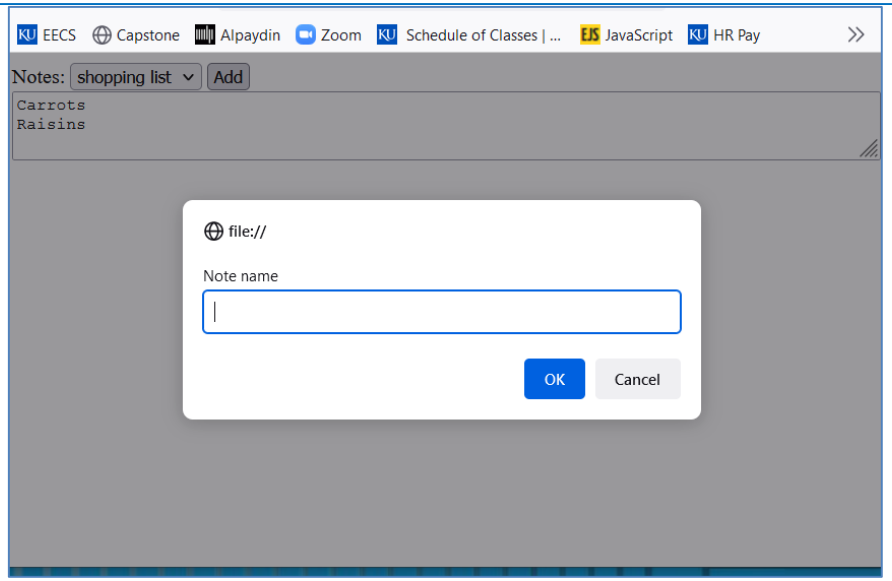


Storing Data Client-Side Example

- The use of Object.assign in the example is intended to create a new object that is a clone of the old state.notes, …
- but with one property added or overwritten.
- Object.assign takes its first argument and adds all properties from any further arguments to it.
- Thus, giving it an empty object will cause it to fill a fresh object.
- The square brackets notation in the third argument is used to create a property whose name is based on some dynamic value.

```
…
document.querySelector("button")
  .addEventListener("click", () => {
    let name = prompt("Note name");
    if (name) setState({
      notes: Object.assign({}, state.notes, {[name]: ""}),
      selected: name
    });
  });
</script>
```

```
Notes: <select></select>
<button>Add</button><br>
<textarea style="width: 100%"></textarea>
```



Any Questions?

Cookies vs. Local Storage

- Cookies = 4K
- Local Storage = 5M

- Cookies go back to server
- Local Storage stays local

- Important when developing apps for countries (e.g., EU) which have stricter data privacy laws than the US

Any Questions?

HTTP and Forms Summary

- In this chapter, we discussed how the HTTP protocol works.
- A client sends a request, which contains a method (usually GET) and a path that identifies a resource.
- The server then decides what to do with the request and responds with a status code and a response body.
- Both requests and responses may contain headers that provide additional information.

Request from client

```
GET /18_http.html HTTP/1.1
Host: eloquentjavascript.net
User-Agent: Your browser's name
```

Response from server

```
HTTP/1.1 200 OK
Content-Length: 65585
Content-Type: text/html
Last-Modified: Mon, 08 Jan 2018 10:29:45 GMT

<!doctype html>
... the rest of the document
```

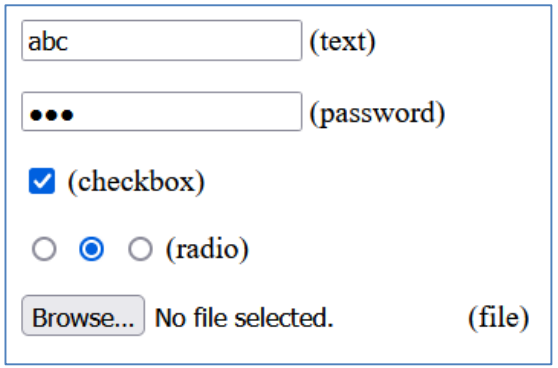
HTTP and Forms Summary

- The interface through which browser JavaScript can make HTTP requests is called fetch.
- Making a request looks like this:

```
fetch("/18_http.html").then(r => r.text()).then(text => {
console.log(`The page starts with ${text.slice(0, 15)}`);
});
```

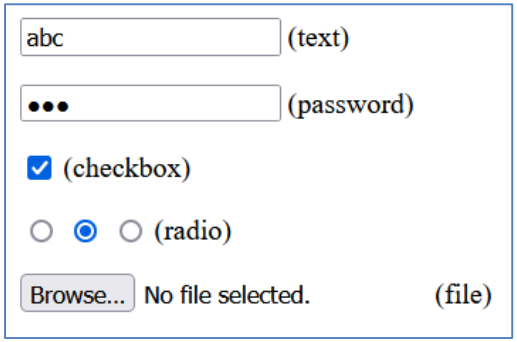
- Browsers make GET requests to fetch the resources needed to display a web page.
- A page may also contain forms, which allow information entered by the user to be sent as a request for a new page when the form is submitted.

- HTML can represent various types of form fields, such as text fields, checkboxes, multiple-choice fields, and file pickers.



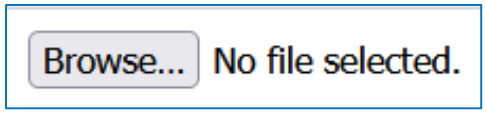
HTTP and Forms Summary

- Such fields can be inspected and manipulated with JavaScript.
- They fire the "change" event when changed, …
- fire the "input" event when text is typed, and …
- receive keyboard events when they have keyboard focus.
- Properties like value (for text and select fields) or checked (for checkboxes and radio buttons) are used to read or set the field's content.



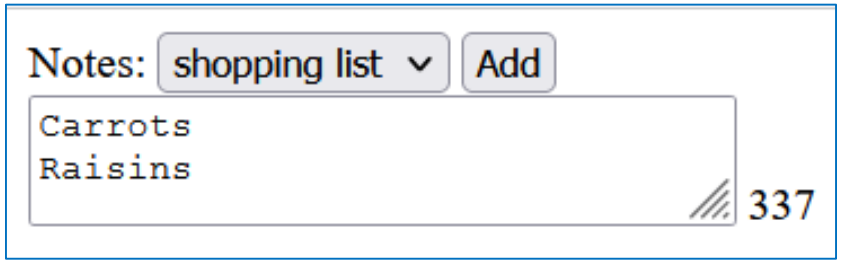
- When a form is submitted, a "submit" event is fired on it.
- A JavaScript handler can call preventDefault on that event to disable the browser's default behavior.
- Form field elements may also occur outside of a form tag.

- When the user has selected a file from their local file system in a file picker field, …
- the FileReader interface can be used to access the content of this file from a JavaScript program.



HTTP and Forms Summary

- The localStorage and sessionStorage objects can be used to save information in a way that survives page reloads.
- localStorage saves the data forever (or until the user decides to clear it).
- sessionStorage saves it until the browser is closed.



Any Questions?

In-Class Problem

1. Draw what the code below would display on the client's browser.
2. Assume the code is in a file named "EECS368InClassProblem2.html", draw what the code below would display on the client's browser if the user selected this file name.
3. What would the console output be?

```
<input type="file" multiple>
<script>
  let input = document.querySelector("input");
  input.addEventListener("change", () => {
    for (let file of Array.from(input.files)) {
      let reader = new FileReader();
      reader.addEventListener("load", () => {
        console.log("File", file.name, "starts with", reader.result.slice(0, 20));
      });
      reader.readAsText(file);
    }
  });
</script>
```