CSE 331
Software Design & Implementation

James Wilcox
Autumn 2021
User Interfaces & JavaScript
Graphical User Interfaces (GUls)

- Large and important class of event-driven programs
  - waits for user-interaction events
  - mouse clicks, button presses, etc.

- Java, Android, Web, etc. provide libraries to write these
  - each of these use the Observer pattern

- Using these libraries decreases bugs
  - also gives users a familiar experience
GUI terminology

**window**: A first-class citizen of the graphical desktop
- also called a *top-level container*
- Examples: *frame* (window), dialog box

**component**: A GUI widget that resides in a window
- called *controls* in many other languages
- Examples: button, text box, label

**container**: A component that hosts (holds) & lays out components
- Examples: frame, *panel*, box
More components...
Structure of GUI Application

- **Model**: data and invariants
- **GUI Components**: presentation (provided by library)
- **Listeners**: updates, events
Improvements in GUI Libraries

• Core parts of the applications are:
  – data and invariants (model)
  – mapping from model into components (view)
  – updates performed in response to events (controller / model)

• Early libraries required a lot of code to implement these

• More recent improvements have made this easier
  – highly valuable
    • your time is important
  – less code (usually) means fewer bugs
This lecture

• Brief survey of Java, Android, & Web
  – discuss how each handles issues below
  – (no need to memorize anything)

• Introductory discussion of JavaScript
AWT / Swing Example 1

SimpleFieldDemo.java
Containers and layout

- Container needs to position (lay out) the child components
- You need to tell it how you want them arranged
- In AWT / Swing, each container has a layout manager
AWT / Swing Examples

• Default is a flow layout
  – components placed next to each other
  – wrap around when out of space on the line

• Can change to a 2 x 2 grid layout
AWT / Swing Example 2

SimpleFieldDemo2.java
AWT / Swing Examples

- Does not look natural

- Instead try 2 rows (2 x 1 grid) and flow layout *within* the rows
SimpleFieldDemo3.java
Easier Layout Idea #1: Just Say No

• Much of the difficulty here has to do with resizing…

• Do we really need to support resizing?

• Two platforms restrict resizing in some ways:
  – Android / iPhone
  – Bootstrap (HTML)
iPhone / Android Layout

- iPhone and iPad come in fixed sizes
- Just give a fixed layout for each possible size
Bootstrap (HTML)

- Width is restricted to one of 5 values (phone up to huge screen)
  - library automatically switches to best match for screen width
  - can use the same design for multiple sizes if you wish

- Still allows arbitrary height for the content
Bootstrap Example

BootstrapDemo.html
Easier Layout Idea #2: Declarative UI

- How much of layout needs to be code?
  - does this really require forward / backward reasoning?

- iPhone / Android show that this can be done
  - only for fixed sized screens

- HTML can be used as a more declarative language for UI
  - (.NET and other frameworks have comparable toolkits)
HTML, Formally

- **Hyper-Text Markup Language**

- Consists of tags and their content
  - components become tags
    - input fields, buttons, etc.
  - containers have start and end tags
    - tags placed in between are children
  - additional information provided to the tag with “attributes”
Anatomy of a Tag

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

- Tag Name
- Content
- Closing Tag

Element
Anatomy of a Tag

```html
<p id="firstParagraph"> Some Text </p>
```
Tags form a Tree

This tree is called the Document Tree

When used from JS, known as the "DOM" – Document Object Model
A Few Useful Tags

• See the HTML Living Standard (linked from Resources tab) for a complete list, along with all their supported attributes.

• Some worth knowing:
  • `<p>` - Paragraph tag, surrounds text with whitespace/line breaks.
  • `<div>` - “The curly braces of HTML” - used for grouping other tags. Surrounds its content with whitespace/line breaks.
  • `<span>` - Like `<div>`, but no whitespace/line breaks.
  • `<br />` - Forces a new line (like “\n”). Has no content.
  • `<html>` and `<head>` and `<body>` - Used to organize a basic HTML document.
HTML + JS

- To make an app we also need **code**

- Code is provided inside a `<script>` tag
  - all browsers support the JavaScript language
  - more in a moment…
HTML + JS UI Example

HtmlFieldDemo.html
HTML + JS + CSS

• HTML removes the need for `panel.add` calls
  – parent / child relationship implied by structure

• Cascading Style Sheets allow separation of styling from rest
  – *styling* is colors, margins, etc.
  – separates concerns
    • code produces document tree
    • CSS styles the tree
    • any changes to tree require agreement by both parties
JAVASCRIPT
JavaScript (formally EcmaScript)

- Created in 1995 by Brendan Eich as a “scripting language” for Mozilla’s browser
  - done in 10 days!

- No relation to Java other than trying to piggyback on all the Java hype at that time

- Tricky due to its minimalism…
  - examples coming shortly…
Syntax and variables

• Syntax similar to Java, C, etc.
• /* comments */ or // comments (prefer //)
• Variables have no type constraints:
  ```
  let x = 42;
  x = "ima string now!";
  ```
  – introduced into program with let
  – use const for constants

• Semicolons are optional at ends of lines and often omitted, but also encouraged 😊
Control flow – just like Java

• Conditionals
  
  ```java
  if (condition) {
    statements
  }
  else if (condition) {
    statements
  }
  else {
    statements
  }
  ```

• Loops
  
  ```java
  while (condition) {
    statements
  }
  ```
  
  ```java
  for (init; condition; update) {
    statements
  }
  ```

  – Also for-of and for-in loops

  • Be careful with these. They have “interesting” semantics and differences that you need to get right if you use them.
Types

• Values do have types, but just 6 of them:
  – number
  – string
  – boolean
  – null & undefined
  – Object (including arrays / lists)
Number Type

• All numbers are floating point! Even here:

```javascript
for (let i = 0; i < 10; i++) {
    ...
}
```

• Usual numeric operations:
  - `+` `-` `*` `/`
  - `++` `--`
  - `+=`
  - `...`

• Math methods much the same as in Java
String type

- Immutable as in Java
- Most of the same methods as in Java
- String concatenation with +

- But also string comparison with <

- Better string literals: `Hi, ${name}!`
  - `${name}` replaced by value of `name`
Boolean type

- Any value can be used in an “if”
  - “falsey” values: `false`, `0`, `NaN`, `“”`, `null`, `undefined`
  - “truthy” values: everything else (including `true`!)

- As expected: `&&`, `||`, `!`
Arrays

```javascript
let empty = [ ];
let names = [ "bart", "lisa" ];
let stuff = [ "wookie", 17, false ];
stuff[6] = 331  // in-between undefined
```

- Access elements with subscripts as usual
- push, pop, shift, unshift, length, …
Objects

• Everything other than number, string, boolean, null, and undefined are mutable Objects

• A JavaScript object is a set of name/value pairs:

```javascript
character = { name: "Lisa Simpson", age: 30 }
```

• Reference properties in two different ways:

```javascript
character.age = 7
character["age"] = 7
```
Objects

- Objects are basically HashMaps (well, almost)
- Add and remove properties as you like

```python
character.instrument = "horn"
delete character.age
```
Objects

• Quotes are optional in object literals:

```javascript
let obj = {a: 1, "b": 2};
```

• But be careful:

```javascript
let x = "foo";
console.log({x: x}); // {"x": "foo"}
```
Equality

• Equality is complicated in any language

• JS has two versions: `===` (strict); `==` (loose)
  – `===`, `!==` check both types and values
  – `==` and `!=` can surprise you with conversions
    • `7 == "7"` is true!

• Object equality is reference equality
  – have to compare arrays yourself
Functions

• Named functions:

```javascript
function average(x, y) {
    return (x + y) / 2;
}
```

• Anonymous (lambda) functions:

```javascript
let f = function (x) { return x+1; }
let g = (x) => { return x+1; }
let h = (x, y) => (x + y) / 2;
```
No type constraints!

```javascript
function average(x, y) {
    return (x + y) / 2;
}
```

• No surprise

```javascript
let result = average(6,7); // 6.5
```

• But then…

```javascript
let answer = average("6","7"); // 33.5!
```
Functions are values

- JavaScript is a functional language
  - functions can be stored as values of variables, passed as parameters, and so on
  - lots of powerful techniques (cf CSE 341); we won’t cover for the most part

```javascript
let f = average
let result = f(6, 7)  // result is 6.5
f = Math.max
result = f(6,7)      // result is 7
```
Higher-order Functions

• Functions can be passed as parameters

```javascript
function compute(f) {
    return f(2,3);
}

compute(((a,b) => a+b));  // 5
compute(((a,b) => a*b));  // 6
```
Every browser has developer tools including the console, details about web pages and objects, etc.

A JS program can use `console.log("message");` to write a message to the console for debugging, recording, etc.

- "printf debugging" for JavaScript programs

In Chrome, right-click on a web page and select Inspect or pick View > Developer > Developer Tools from the menu. Click the console tab and you can see output that’s been written there, plus you can enter JavaScript expressions and evaluate them. Super useful for trying things out.
Resources

• Lectures will (try to) point out key things
• For more: start with Mozilla (MDN) JavaScript tutorial:  
• CodeAcademy has a good, free JavaScript basics course

• Be real careful about web searches – the JavaScript/ webapp ecosystem has way too many somewhat-to-totally incompatible or current vs. obsolete ways of doing similar things. Code snippets from the web may lead you way off.
CLASSES IN JS
Classes

• JavaScript (until recently) has *no classes!*
  – but it is still an object-oriented language

• We can do some of what we need already:

  ```javascript
  let obj = {f: (x) => x + 1};
  console.log(obj.f(2));  // 3
  ```

• **Problem**: how would a method update the state (other fields) of the object?
In expression `obj.method(...)`:  
- `obj` is secretly passed to `method`  
- can be accessed using keyword `this`

So this works properly:

```javascript
let obj = {
    a: 3,
    f: function (x) { return x + this.a }
};
console.log(obj.f(2)); // 5
```
You can explicitly pass the “this” argument using the `call` method on a function object:

```javascript
let obj = {
    a: 3,
    f: function (x) { return x + this.a; }
};
console.log(obj.f.call(obj, 2)); // 5
console.log(obj.f.call({a:4}, 2)); // 6
```
**this**

- **this** only passed if you have "obj." before `method(...)"

```javascript
function compute(f) {
    return f(2);  // no "obj." so no "this"
}
let obj = {
    a: 3,
    f: function (x) { return x + this.a; } // NaN!
};
console.log(compute(obj.f));  // NaN!
```
Why should I care about this?

• We can add listener functions to components, but they don’t know to pass this!

• This will not work:

```javascript
btn.addEventListener("click", obj.foo)
```

• `obj.foo` produces a function, which it calls, but at that point it’s a regular function
  – `this` is only passed if you put (...) right after it
How to fix this

• You can produce a function with this already set by using the bind method of a function object:

```javascript
btn.addEventListener("click",
    foo.bind(obj))
```

• Inside of another method (where this is already set) the => lambda syntax does this automatically:

```javascript
btn.addEventListener("click",
    evt => this.onClick(evt))
```