CS106AJ Final Review Session

December 8, 2018

Ryan Eberhardt

Game plan

- Run through course material in syllabus order
  - If you see material you are uncomfortable with, make a note of it and we can do some practice problems
  - You don’t have to have all of this *memorized*. (In fact, that’s not a good use of time.) However, you should feel familiar with it such that you can remember what you need, find it in the book, and use it to solve a problem.
- Talk tips for taking the final
- Work through practice problems

Structure of the final

1) Short answer (trace problems)
2) “Simple” graphics
3) Interactive graphics
4) Strings
5) Arrays
6) Working with data structures (combining arrays and objects)
7) Reading data structures from embedded XML

Functions

- (Optionally) takes some input, does something, and (optionally) returns some output
- Syntax:

```
function calcHypotenuse(a, b) {
    return Math.sqrt(a * a + b * b);
}
```

Or:

```
let calcHypotenuse = function(a, b) {
    return Math.sqrt(a * a + b * b);
}
```

Functions

- Important things to know
  - Variables in one function are *not* accessible in a different function (as long as the functions aren't nested)! If you want to share variables, you need to pass them as parameters

```
function main() {
    let str = "hello world";
    print();
}
function print() {
    console.log(str);          // error!
}
```

  - Parameters are passed by order, not by name
  - (Most) parameters get copied when you pass them. If you want to modify a parameter, you need to `return` it to the code that called the function

Functions

- Important things to know
  - Variables in one function are *not* accessible in a different function (as long as the functions aren't nested)! If you want to share variables, you need to pass them as parameters
  - Parameters are passed by order, not by name:

```
function main() {
    let a = "a";
    let b = "b";
    print(b);
}
function print(a) {
    console.log(a);
}
```

  - (Most) parameters get copied when you pass them. If you want to modify a parameter, you need to `return` it to the code that called the function

Functions

- Important things to know
  - Variables in one function are *not* accessible in a different function (as long as the functions aren't nested)! If you want to share variables, you need to pass them as parameters
  - Parameters are passed by order, not by name:

```
function main() {
    let a = "a";
    let b = "b";
    print(b);
}
function print(a) {
    console.log(a);  // prints "b"
}
```

  - (Most) parameters get copied when you pass them. If you want to modify a parameter, you need to `return` it to the code that called the function

Functions

- Important things to know
  - Variables in one function are *not* accessible in a different function (as long as the functions aren't nested)! If you want to share variables, you need to pass them as parameters
  - Parameters are passed by order, not by name
  - (Most) parameters get copied when you pass them. If you want to modify a parameter, you need to `return` it to the code that called the function

Functions

- Important things to know
  - Variables in one function are *not* accessible in a different function (as long as the functions aren't nested)! If you want to share variables, you need to pass them as parameters
  - Parameters are passed by order, not by name
  - (Most) parameters get copied when you pass them. If you want to modify a parameter, you need to `return` it to the code that called the function

```
function main() {
    let a = 0;
    addTwo(a);
    console.log(a);
}
function addTwo(num) {
    num = num + 2;
}
```

Functions

- Important things to know
  - Variables in one function are *not* accessible in a different function (as long as the functions aren't nested)! If you want to share variables, you need to pass them as parameters
  - Parameters are passed by order, not by name
  - (Most) parameters get copied when you pass them. If you want to modify a parameter, you need to `return` it to the code that called the function

```
function main() {
    let a = 0;
    addTwo(a);
    console.log(a);  // prints 0
}
function addTwo(num) {
    num = num + 2;
}
```

Functions

- Important things to know
  - Variables in one function are *not* accessible in a different function (as long as the functions aren't nested)! If you want to share variables, you need to pass them as parameters
  - Parameters are passed by order, not by name
  - (Most) parameters get copied when you pass them. If you want to modify a parameter, you need to `return` it to the code that called the function

```
function main() {
    let a = 0;
    a = addTwo(a);
    console.log(a);  // prints 2
}
function addTwo(num) {
    num = num + 2;
    return num;
}
```

Functions

- Important things to know
  - Variables in one function are *not* accessible in a different function (as long as the functions aren't nested)! If you want to share variables, you need to pass them as parameters
  - Parameters are passed by order, not by name
  - (Most) parameters get copied when you pass them. If you want to modify a parameter, you need to `return` it to the code that called the function

```
function main() {
     let point = {x: 1, y: 2};
     reset(point);
     console.log(point);
}
function reset(point) {
     point = {x: 0, y: 0};
}
```

Functions

- Important things to know
  - Variables in one function are *not* accessible in a different function (as long as the functions aren't nested)! If you want to share variables, you need to pass them as parameters
  - Parameters are passed by order, not by name
  - (Most) parameters get copied when you pass them. If you want to modify a parameter, you need to `return` it to the code that called the function

```
function main() {
    let point = {x: 1, y: 2};
    reset(point);
    console.log(point); // prints {x: 1, y: 2}
}
function reset(point) {
    point = {x: 0, y: 0};
}
```

Functions

- Important things to know
  - Variables in one function are *not* accessible in a different function (as long as the functions aren't nested)! If you want to share variables, you need to pass them as parameters
  - Parameters are passed by order, not by name
  - (Most) parameters get copied when you pass them. If you want to modify a parameter, you need to `return` it to the code that called the function

```
function main() {
    let point = {x: 1, y: 2};
    point = reset(point);
    console.log(point); // prints {x: 0, y: 0}
}
function reset(point) {
    point = {x: 0, y: 0};
    return point;
}
```

Functions

- Important things to know
  - Variables in one function are *not* accessible in a different function (as long as the functions aren't nested)! If you want to share variables, you need to pass them as parameters
  - Parameters are passed by order, not by name
  - (Most) parameters get copied when you pass them. If you want to modify a parameter, you need to `return` it to the code that called the function

```
function main() {
     let point = {x: 1, y: 2};
     scale(point);
     console.log(point);
}
function scale(point) {
     point.x *= 2;
     point.y *= 2;
}
```

Functions

- Important things to know
  - Variables in one function are *not* accessible in a different function (as long as the functions aren't nested)! If you want to share variables, you need to pass them as parameters
  - Parameters are passed by order, not by name
  - (Most) parameters get copied when you pass them. If you want to modify a parameter, you need to `return` it to the code that called the function

```
function main() {
    let point = {x: 1, y: 2};
    scale(point);
    console.log(point); // prints {x: 2, y: 4}
}
function scale(point) {
    point.x *= 2;
    point.y *= 2;
}
```

Functions

- Important things to know
  - Variables in one function are *not* accessible in a different function (as long as the functions aren't nested)! If you want to share variables, you need to pass them as parameters
  - Parameters are passed by order, not by name
  - (Most) parameters get copied when you pass them. If you want to modify a parameter, you need to `return` it to the code that called the function
    - Caveat: objects/arrays can be *modified*, but they can't be *reassigned*. (The reference gets copied, not the object/array itself.)
- Tip for trace problems: draw "stack cards" to illustrate the value of variables in each function, and draw out the values of arrays/objects

Closures

- A nested function gets access to all of its parent's variables
  - ```
    function main() {
        let str = "hello world";
        function print() {
            console.log(str);    // works!
        }
        print();
    }
    ```
- This works for functions nested arbitrarily deep (although stylistically, you shouldn't do that)
- Important for timers (via `setTimeout` or `setInterval`) and mouse event handlers
- Style note (not important for final): don't abuse/overuse closures!
    ```

Graphics

- Remember that coordinates for most GObjects specify the top-left of the object.
- ```
  let gw = GWindow(width, height);
  let line = GLine(x0, y0, x1, y1);
  let oval = GOval(x, y, diameterX, diameterY);
  let rect = GRect(x, y, width, height);
  ```
- The coordinates for a GLabel specify the left point on the baseline for the text
  ```

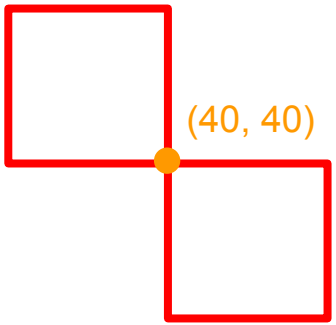
Graphics

- Remember that coordinates for GArc, GCompound, and GPolygon specify the origin that you defined when creating the object. This might be better understood through example.

```
let diamond = GPolygon();

diamond.addVertex(-DIAMOND_WIDTH / 2, 0);
diamond.addVertex(0, DIAMOND_HEIGHT / 2);
diamond.addVertex(DIAMOND_WIDTH / 2, 0);
diamond.addVertex(0, -DIAMOND_HEIGHT / 2);

gw.add(diamond, gw.getWidth() / 2, gw.getHeight() / 2);
```

Graphics

```
const SIZE = 30;

let compound = GCompound();

compound.add(GRect(-SIZE, -SIZE, SIZE, SIZE));
compound.add(GRect(0, 0, SIZE, SIZE));
gw.add(compound, 40, 40);
```



Graphics

- Tips for graphics problems:
  - Draw it out! Draw what the screen should look like. Then figure out the coordinates that are necessary for the screen to look like that
  - If you're dealing with many shapes (like the pyramid problem), it doesn't hurt to draw an example situation (e.g. BRICKS_IN_BASE = 3) and manually figure out the coordinates for each individual brick. Then, try to figure out a general formula that applies for any brick.
  - If you are dealing with animations, figure out what variables you will need ahead of time. Leave extra room. Be careful of where you define your variables:
    - Variables defined in a `step` function will be reset on every step
    - Variables defined in one closure function will not be available to a different closure function

Graphics

- Mouse events:

```
function listenerFunction(e) { … }
gw.addEventListener("click", listenerFunction);
```

- Know how to use gw.getElementAt(x, y) to get a reference to a GObject

Mouse Events

- The following table shows the different mouse-event types:

`"click"`	The user clicks the mouse in the window.
`"dblclk"`	The user double-clicks the mouse.
`"mousedown"`	The user presses the mouse button.
`"mouseup"`	The user releases the mouse button.
`"mousemove"`	The user moves the mouse with the button up.
`"drag"`	The user drags the mouse with the button down.

- Certain user actions can generate more than one mouse event. For example, clicking the mouse generates a `"mousedown"` event, a `"mouseup"` event, and a `"click"` event, in that order.
- Events trigger no action unless a client is listening for that event type. The `DrawDots.js` program listens only for the `"click"` event and is therefore never notified about any of the other event types that occur.

Graphics

- Timer events
  - Events that occur after a specific time interval
  - Allows you to add animation to a JavaScript program
- Timer functions
  - `let timer = setTimeout(func, delay)`
    - “One-shot” timer
  - `let timer = setInterval(func, delay)`
    - Repeated timer
  - `clearTimeout(timer)`

RandomLib.js

- `<script type='text/javascript' src='http://cs106aj.stanford.edu/jslib/RandomLib.js'> </script>`
- See pg 123 of course reader
- `randomInteger(low, high); // [low, high] inclusive`
- `randomReal(low, high); // [low, high) inclusive, exclusive`
- `randomChance(probability);`
- `randomColor();`

Strings

- Ordered collection of characters
- Represented in quotes
  - Example: "CS106J is awesome!"
  - Example: ""
- Character positions in a string are identified by an index
  - Indices begin with 0, not 1
  - `let exam = "The final"`
  - `exam.charAt(0) -> "T"`
  - `exam.charAt(5) -> "i"`
  - `exam.length -> 9`
  - `exam.indexOf("f") -> 4`

Strings

- Concatenation
  - Fancy word for combining strings together
  - Ex: `"Jerry Cain and " + "Ryan Eberhardt"`
    `-> "Jerry Cain and Ryan Eberhardt"`
- Substrings
  - Extract parts of a string
  - `str.substring(p1, p2)`
    - `p1` is first index position in desired substring
    - `p2` is index immediately following the last index you want
- Comparison
  - `a === b` to check if strings `a` and `b` are equal
  - if `a < b`, `a` comes before `b` in dictionary
  - if `a > b`, comes after `b` in dictionary

Strings

Other Methods in the `String` Class

`String.fromCharCode(`*code*`)`** Returns the one-character string whose Unicode value is *code*.
`charCodeAt(`*index*`)`** Returns the Unicode value of the character at the specified index.
`toLowerCase()` Returns a copy of this string converted to lower case.
`toUpperCase()` Returns a copy of this string converted to upper case.
`startsWith(`*prefix*`)`** Returns **`true`** if this string starts with *prefix*.
`endsWith(`*suffix*`)`** Returns **`true`** if this string starts with *suffix*.
`trim()` Returns a copy of this string with leading and trailing spaces removed.

Strings

- Strings are immutable
  - ```
    let s = "hello!";
    s.toUpperCase();
    console.log(s);
    ```
    ```

Strings

- Strings are immutable
  - ```
    let s = "hello!";
    s.toUpperCase();
    console.log(s);  // prints "hello!"
    ```
    ```

Strings

- Strings are immutable
  - ```
    let s = "hello!";
    s = s.toUpperCase();
    console.log(s);  // prints "HELLO!"
    ```
- In most string problems, we take some existing string, loop over its characters, and build up a new string from scratch
- Try to come up with an approach in your head before you think about any code
    ```

Arrays

- Arrays are *ordered* collections of elements
- Like strings, indices start from 0 and go to `arr.length - 1`
- ```
  let arr = ["a", "b", "c"];
  console.log(arr.length);  // prints 3
  console.log(arr[1]);      // prints b
  ```
- Array “iteration”:
  ```
  for (let i = 0; i < arr.length; i++) {
      // do something with arr[i]
      console.log(arr[i]);
  }
  ```
- Reverse iteration?
  ```

Arrays

- Arrays are *ordered* collections of elements
- Like strings, indices start from 0 and go to `arr.length - 1`
- ```
  let arr = ["a", "b", "c"];
  console.log(arr.length);  // prints 3
  console.log(arr[1]);      // prints b
  ```
- Array “iteration”:
- Reverse iteration?

```
for (let i = arr.length - 1; i >= 0; i--) {
    // do something with arr[i]
    console.log(arr[i]);
}
```
```

Arrays

- Arrays are *ordered* collections of elements
- Like strings, indices start from 0 and go to `arr.length - 1`
- ```
  let arr = ["a", "b", "c"];
  console.log(arr.length);  // prints 3
  console.log(arr[1]);      // prints b
  ```
- Array “iteration”:
- Reverse iteration?

```
for (let i = 0; i < arr.length; i++) {
    // do something with arr[arr.length - 1 - i]
    console.log(arr[arr.length - 1 - i]);
}
```
```

Arrays

- Add one or more elements:
  `arr.push(element, …)`
- Remove and return the first element:
  `arr.shift()`
- Remove and return the last element:
  `arr.pop()`
- Remove the element at index `i`:
  `arr.splice(i, 1)`
- Find an element:
  `["a", "b", "c"].indexOf("b") -> 1`

Arrays for tabulation

- If we have an array of digits (0-9), how can we find the most common number (the mode)?

```
let digits = [3, 2, 6, 8, 0, 6, 2, 4, 4, 6, 7, 5, 6, 4, 9, 2, 3, 1, 3, 3];
```

Arrays for tabulation

- If we have an array of digits (0-9), how can we find the most common number (the mode)?

```
let digits = [3, 2, 6, 8, 0, 6, 2, 4, 4, 6, 7, 5, 6, 4, 9, 2, 3, 1, 3, 3];
let counts = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
for (let i = 0; i < digits.length; i++) {
     let digit = digits[i];
     counts[digit]++;
}
// find largest digit
let largestCount = 0;
let largestIndex = 0;
for (let i = 0; i < counts.length; i++) {
     if (counts[i] > largestCount) {
          largestCount = counts[i];
          largestIndex = i;
     }
}
console.log("Mode: " + largestIndex);
```

Arrays for tabulation

- If we have an array of digits (0-9), how can we find the most common number (the mode)?

```
let digits = [3, 2, 6, 8, 0, 6, 2, 4, 4, 6, 7, 5, 6, 4, 9, 2, 3, 1, 3, 3];
```

- Semi-related: How would you do this if we weren't limited to numbers 0 to 9?

Arrays for tabulation

- If we have an array of digits (0-9), how can we find the most common number (the mode)?

```
let digits = [3, 2, 6, 8, 0, 6, 2, 4, 4, 6, 7, 5, 6, 4, 9, 2, 3, 1, 3, 3];
```

- Semi-related: How would you do this if we weren't limited to numbers 0 to 9?
  (use a map!)

2D Arrays

- Literally an array of arrays:

```
let 2dArr = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
```

- `2dArr.length` is the number of rows (i.e. the height) of the matrix
- `2dArr[0].length` is the number of columns (i.e. the width) of the matrix
- `console.log(2dArr[1][2]); // prints 6`
- You can get the pixels from an image as a 2D array via `img.getPixelArray()`
- You can create an image from a pixel array as `let img = GImage(2dArr);`

Reading from a file

```
let callback = function(text) {
    let lines = JSFile.convertToLineArray(text);
    while (lines.length > 0) {
        let line = lines.shift();
        // Do something with line
    }
};
JSFile.chooseTextFile(callback);
```

Reading from embedded XML

- The XML data from `index.html` is stored internally in the DOM.
  Access it using the following three methods:

`document.getElementById(id)`	Returns the element with the specified id attribute
`element.getElementsByTagName(name)`	Returns an array of the elements with the specified tag name
`element.getAttribute(name)`	Returns the value of the named attribute
`element.innerHTML` (no parentheses!)	Returns the literal contents of the HTML tag as a string

Reading from embedded XML

- Recall how the Teaching Machine reads the question data into an internal form:
-
```
<course id="CourseData" title="JavaScript">
    <question name="Q1">
        True or false: Numbers can have fractional parts.
        <answer response="true" nextQuestion="Q3" />
        <answer response="false" nextQuestion="Q2" />
    </question>
    <question name="Q2">
        That's incorrect.
        True or false: Numbers can be negative.
        <answer response="true" nextQuestion="Q3" />
        <answer response="false" nextQuestion="Q1" />
    </question>
    …
</course>
```

Reading from embedded XML

- Recall how the Teaching Machine reads the question data into an internal form:
- ```
  let courseXML = document.getElementById("CourseData");
  let questionElements = courseXML.getElementsByTagName("question");
  for (let i = 0; i < questionElements.length; i++) {
      let questionXML = questionElements[i];
      let name = questionXML.getAttribute("name");
      let lines = questionXML.innerHTML.split("\n");
      let answerElements =
                    questionXML.getElementsByTagName("answer”);
      for (let i = 0; i < answerElements.length; i++) {
          let answerXML = answerElements[i];
          let nextQuestion = answerXML.getAttribute("nextQuestion");
          // ...
      }
  }
  ```
  ```

Requesting Input

- ```
  let callback = function(input) {
      // Do something with input…

      // Request input a second time:
      console.requestInput("> ", callback);
  };
  // Request input the first time:
  console.requestInput("> ", callback);
  ```
  ```

Objects

- ```
  let obj = {
      key: "value"
  };
  obj.key2 = "value2";
  console.log(obj.key);
  console.log(obj["key"]);
  ```
- Objects are super flexible. Many ways to use them
  - Aggregates
  - OOP objects
  - Maps
  ```

Objects

- Aggregates are the most “primitive” way to use objects. Just a collection of variables
  - `let point = {x: 1, y: 2};`
- Use object-oriented design to design more complex objects in order to safeguard yourself from mistakes
- Use maps when the *keys* are also considered to be “data” (i.e. the keys aren’t known while you’re writing your program)

Objects

- Aggregates are the most “primitive” way to use objects. Just a collection of variables
- Use object-oriented design to design more complex objects in order to safeguard yourself from mistakes
  - ```
    function Employee() {
        let salary = 100;
        return {
            getSalary: function() { return salary; },
            setSalary: function(newSalary) {
                if (newSalary > 0) salary = newSalary;
                else console.log("Salary must be positive!");
            }
        };
    }
    ```
- Use maps when the *keys* are also considered to be “data” (i.e. the keys aren’t known while you’re writing your program)
    ```

Objects

- Aggregates are the most “primitive” way to use objects. Just a collection of variables
- Use object-oriented design to design more complex objects in order to safeguard yourself from mistakes
  - ```
    let label = GLabel("Hello!");
    console.log(label.getLabel());
    console.log(label["getLabel"]);
    ```
- Use maps when the *keys* are also considered to be “data” (i.e. the keys aren’t known while you’re writing your program)
    ```

Objects

- Aggregates are the most “primitive” way to use objects. Just a collection of variables
- Use object-oriented design to design more complex objects in order to safeguard yourself from mistakes
  - ```
    let label = GLabel("Hello!");
    console.log(label.getLabel());
    console.log(label["getLabel"]());    // this actually works!
    ```
- Use maps when the *keys* are also considered to be “data” (i.e. the keys aren’t known while you’re writing your program)
    ```

Objects

- Aggregates are the most “primitive” way to use objects. Just a collection of variables
- Use object-oriented design to design more complex objects in order to safeguard yourself from mistakes
- Use maps when the *keys* are also considered to be “data” (i.e. the keys aren’t known while you’re writing your program)

Objects

- Say we have XML like this:

```
<directory id="Directory">
    <person name="Ryan"
            number="(847) 220-8476" />
    <person name="Jerry"
            number="(123) 456-4587" />
</directory>
```

- We can write a phonebook simulator:

Objects

- Say we have XML like this:

```
<directory id="Directory">
    <person name="Ryan"
            number="(847) 220-8476" />
    <person name="Jerry"
            number="(123) 456-4587" />
</directory>
```

- We can write a phonebook simulator:

```
let directory = document.getElementById("Directory");
let people = directory.getElementsByTagName("person");
let phoneNumbers = {};
for (let i = 0; i < people.length; i++) {
    let name = people[i].getAttribute("name");
    let number = people[i].getAttribute("number");
    phoneNumbers[name] = number;
}
let lookUpPerson = function(name) {
    console.log(phoneNumbers[name]);
}
console.requestInput("Enter a person to look up: ", lookUpPerson);
```

Objects

- Aggregates are the most “primitive” way to use objects. Just a collection of variables
- Use object-oriented design to design more complex objects in order to safeguard yourself from mistakes
- Use maps when the *keys* are also considered to be “data” (i.e. the keys aren’t known while you’re writing your program)

Objects

- Iterating through maps:

```
for (let key in map) {
    console.log(key + ": " + map[key]);
}
```

- Remember, objects are unordered collections!
- Keys are unique; values are not necessarily unique

Objects

- ```
  let obj = {a: null};
  console.log(obj[a]);
  ```
  ```

Objects

- ```
  let obj = {a: null};
  console.log(obj[a]);
  console.log(obj.a);
  ```
  ```

Objects

- ```
  let obj = {a: null};
  console.log(obj[a]);
  console.log(obj.a);
  console.log(obj["a"]);
  ```
  ```

Objects

- ```
  let obj = {a: null};
  console.log(obj[a]);       // error! a is undefined
  console.log(obj.a);    // prints null
  console.log(obj["a"]);     // prints null
  let a = "b";
  console.log(obj[a]);
  ```
  ```

Objects

- ```
  let obj = {a: null};
  console.log(obj[a]);       // error! a is undefined
  console.log(obj.a);    // prints null
  console.log(obj["a"]);     // prints null
  let a = "b";
  console.log(obj[a]);       // prints undefined
  let label = GLabel("hello");
  console.log(label.getLabel);
  ```
  ```

Objects

- ```
  let obj = {a: null};
  console.log(obj[a]);       // error! a is undefined
  console.log(obj.a);    // prints null
  console.log(obj["a"]);     // prints null
  let a = "b";
  console.log(obj[a]);       // prints undefined
  let label = GLabel("hello");
  console.log(label.getLabel);  // prints a function
  ```
  ```

Structure of the final

1) Short answer (trace problems)
    - Understand scoping rules
    - Understand the idea of functions as objects
2) “Simple” graphics
    - Simple as in “no animation or interactivity”
    - Draw things on paper
3) Interactive graphics
    - Be familiar with event handlers and timers
4) Strings
5) Arrays
6) Working with data structures (combining arrays and objects)
7) Reading data structures from embedded XML

Tips for taking the final

- Don’t panic!
  - You *can* do this. Try writing out different things or try thinking through different approaches. Don’t sit and stare; move on and come back if you’re stuck
- Go in with a plan (e.g. write pseudocode or write your approach)
- Leave extra space between your lines
- Make sure you’re familiar with the book or with your notes
  - Be able to look things up quickly
- Commenting is optional but can be a really good idea
  - Commenting helps your grader figure out what you were doing and can help us give you partial credit
- Tackle the problem in chunks
  - If you figure out a high level decomposition strategy first, it will be much easier than trying to take the problem one line at a time