Game plan

- Quickly run through course material
  - 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 probably 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 midterm
- Work through practice problems
Topics on the midterm

1) Short answer (trace problems)
   - Understand scoping rules
   - Understand the idea of functions as objects

2) Graphics
   - Draw things on paper
   - Be familiar with event handlers and timers

3) Strings

4) Arrays

5) Working with data structures (combining arrays and objects)
Functions

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

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

```javascript
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:
    ```javascript
    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:
    ```javascript
    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.

```javascript
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.

```javascript
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

```javascript
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

```javascript
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.

```javascript
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.

```javascript
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

```javascript
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

```javascript
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
  
```javascript
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 (bottom-left) for the text
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);

bw.add(diamond, bw.getWidth() / 2, bw.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);
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.
Mouse events:

```javascript
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:

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;click&quot;</td>
<td>The user clicks the mouse in the window.</td>
</tr>
<tr>
<td>&quot;dblclk&quot;</td>
<td>The user double-clicks the mouse.</td>
</tr>
<tr>
<td>&quot;mousedown&quot;</td>
<td>The user presses the mouse button.</td>
</tr>
<tr>
<td>&quot;mouseup&quot;</td>
<td>The user releases the mouse button.</td>
</tr>
<tr>
<td>&quot;mousemove&quot;</td>
<td>The user moves the mouse with the button up.</td>
</tr>
<tr>
<td>&quot;drag&quot;</td>
<td>The user drags the mouse with the button down.</td>
</tr>
</tbody>
</table>

- Certain user actions can generate more than one mouse event. For example, clicking the mouse generates a "mousedown" event, a "mousemove" 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)`
  - `clearInterval(timer)`
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: "CS106X 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, a comes after b in dictionary
Other Methods in the `String` Class

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>String.fromCharCode(code)</code></td>
<td>Returns the one-character string whose Unicode value is <code>code</code>.</td>
</tr>
<tr>
<td><code>charAt(index)</code></td>
<td>Returns the Unicode value of the character at the specified index.</td>
</tr>
<tr>
<td><code>toLowerCase()</code></td>
<td>Returns a copy of this string converted to lower case.</td>
</tr>
<tr>
<td><code>toUpperCase()</code></td>
<td>Returns a copy of this string converted to upper case.</td>
</tr>
<tr>
<td><code>startsWith(prefix)</code></td>
<td>Returns <code>true</code> if this string starts with <code>prefix</code>.</td>
</tr>
<tr>
<td><code>endsWith(suffix)</code></td>
<td>Returns <code>true</code> if this string starts with <code>suffix</code>.</td>
</tr>
<tr>
<td><code>trim()</code></td>
<td>Returns a copy of this string with leading and trailing spaces removed.</td>
</tr>
</tbody>
</table>
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”:
  ```javascript
  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 \texttt{arr.length - 1}
- let \texttt{arr} = ["a", "b", "c"];  
  \texttt{console.log(arr.length); // prints 3}  
  \texttt{console.log(arr[1]); // prints b}
- Array “iteration”:
- Reverse iteration?
  for (let \texttt{i} = arr.length - 1; i >= 0; i--) {
    // do something with arr[i]
    \texttt{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"];

```javascript
console.log(arr.length); // prints 3
console.log(arr[1]);      // prints b
```

- Array “iteration”:
- Reverse iteration?

```javascript
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:
  \[ \text{arr.push(element, ...)} \]
- Remove and return the first element:
  \[ \text{arr.shift()} \]
- Remove and return the last element:
  \[ \text{arr.pop()} \]
- Remove the element at index \( i \):
  \[ \text{arr.splice(i, 1)} \]
- Find an element:
  \[ ["a", "b", "c"].indexOf("b") \rightarrow 1 \]
Arrays for tabulation

- If we have an array of digits (0–9), how can we find the most common number (the mode)?
  ```javascript
  let digits = [3, 2, 6, 8, 0, 6, 2, 4, 4, 6, 7, 5, 6, 4, 9, 2, 3, 1, 3, 3];
  ```
If we have an array of digits (0-9), how can we find the most common number (the mode)?

```javascript
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, 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)?
  
  ```javascript
  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!)
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
  - Maps
Objects

- **Aggregates** are the most “primitive” way to use objects. Just a collection of variables
  - `let point = {x: 1, y: 2};`
- 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

- Iterating through maps:
  ```javascript
  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
Topics on the midterm

1) Short answer (trace problems)
   - Understand scoping rules
   - Understand the idea of functions as objects

2) Graphics
   - Draw things on paper
   - Be familiar with event handlers and timers

3) Strings

4) Arrays

5) Working with data structures (combining arrays and objects)
Tips for taking your first CS Midterm

- 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