CS106AJ Midterm Review Session

October 28, 2018

Ryan Eberhardt

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) Karel
2) Code tracing:
    a) JavaScript Expressions
    b) Functions and closures
3) Simple JavaScript Programs (loops)
4) Graphics and animation
5) String manipulation

* One question pulled directly from course reader

Karel

- Functions
  - `move()`
  - `turnLeft()`
  - `pickBeeper()`
  - `putBeeper()`
- Control flow
  - `repeat (count) {}`
  - `while (condition) {}`
  - `if (condition) {}`

Karel

Conditions in Karel

- Karel can test the following conditions:

positive condition	*negative condition*
`frontIsClear()`	`frontIsBlocked()`
`leftIsClear()`	`leftIsBlocked()`
`rightIsClear()`	`rightIsBlocked()`
`beepersPresent()`	`noBeepersPresent()`
`beepersInBag()`	`noBeepersInBag()`
`facingNorth()`	`notFacingNorth()`
`facingEast()`	`notFacingEast()`
`facingSouth()`	`notFacingSouth()`
`facingWest()`	`notFacingWest()`

Karel

- Remember that you cannot use variables, `break`, `return`, etc!
- Karel problems will mostly be algorithmic and decomposition challenges. Have a plan for how you are going to structure your program before you write it!
  - We aren't grading you on style (so decomposition and comments won't affect your grade). However, good decomposition will make it much easier to solve the problem.
  - Be sure to write out preconditions and postconditions for your functions. Make sure that given the precondition, your function will always satisfy the postcondition. This will make it much easier to trace through the code in your head
  - Work through some test worlds visually to make sure your algorithm works. (Use an eraser to represent Karel's position)

Expressions and Operators

- Operators
  - `*, /, %, +, -`
  - Order of operations matters!
- Variables
  - `let` vs `const`

Expressions and Operators

Useful Functions in the `Math` Class

`Math.PI`	The mathematical constant π
`Math.E`	The mathematical constant e
`Math.abs(`x`)`	The absolute value of x
`Math.max(`x`, `y`, ...)`	The largest of the arguments
`Math.min(`x`, `y`, ...)`	The smallest of the arguments
`Math.round(`x`)`	The closest integer to x
`Math.floor(`x`)`	The largest integer not exceeding x
`Math.log(`x`)`	The natural logarithm of x
`Math.exp(`x`)`	The inverse logarithm (e^x)
`Math.pow(`x`, `y`)`	The value x raised to the y power (x^y)
`Math.sin(`θ`)`	The sine of θ, measured in radians
`Math.cos(`θ`)`	The cosine of θ, measured in radians
`Math.random(`x`)`	A random number between 0 and 1

Loops

- `for` loop
  - for (initialization; condition; do something on every iteration) {
    // do stuff
    }
  - Print 0, 2, 4, 6, 8:

```
for (let i = 0; i < 10; i += 2) {
    console.log(i);
}
```

- `while` loop
- Use a `for` loop when you know how many times you will be looping

Loops

- `for` loop
- `while` loop
  - while(condition) {
    // do stuff
    }
  - ```
    while (n != 1) {
        // do stuff
    }
    ```
- Use a `for` loop when you know how many times you will be looping
    ```

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
    - Caveat: GObjects can be *modified*, but they can't be *reassigned*.
- Tip for trace problems: draw "stack cards" to illustrate the value of variables in each function

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 midterm): don't abuse/overuse closures!
    ```

Graphics

- `<script type='text/javascript' src='http://cs106aj.stanford.edu/jslib/JSGraphics.js'></script>`
- Remember that coordinates specify the top-left of an 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);
  ```
  ```

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.

```
var 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

```
let compound = GCompound();

compound.add(GRect(-width/2, -height/2, width, height));
compound.add(GRect(-width/2, -height/2, width/2, height/2));
compound.add(GRect(-width/2, 0, width/2, height/2));
```

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

The `GWindow` Class Revisited

The following expanded set of methods are available in the `GWindow` class:

`add``(`*object*`)`	Adds the object to the canvas at the front of the stack
`add``(`*object*`, `*x*`, `*y*`)`	Moves the object to (*x*, *y*) and then adds it to the canvas
`remove``(`*object*`)`	Removes the object from the canvas
`removeAll``()`	Removes all objects from the canvas
`getElementAt``(`*x*`, `*y*`)`	Returns the frontmost object at (*x*, *y*), or **`null`** if none
`getWidth``()`	Returns the width in pixels of the entire canvas
`getHeight``()`	Returns the height in pixels of the entire canvas
`setBackground``(`*c*`)`	Sets the background color of the canvas to *c*.

Graphics

Methods Common to All `GObjects`

`setLocation(`*x*`, `*y*`)`	Resets the location of the object to the specified point
`move(`*dx*`, `*dy*`)`	Moves the object *dx* and *dy* pixels from its current position
`movePolar(`*r*`, `*theta*`)`	Moves the object *r* pixel units in direction *theta*
`getX()`	Returns the *x* coordinate of the object
`getY()`	Returns the *y* coordinate of the object
`getWidth()`	Returns the horizontal width of the object in pixels
`getHeight()`	Returns the vertical height of the object in pixels
`contains(`*x*`, `*y*`)`	Returns `true` if the object contains the specified point
`setColor(`*c*`)`	Sets the color of the object to *c*
`getColor()`	Returns the color currently assigned to the object
`scale(`*sf*`)`	Scales the shape by the scale factor *sf*
`rotate(`*theta*`)`	Rotates the shape counterclockwise by *theta* degrees
`sendToFront()`	Sends the object to the front of the stacking order
`sendToBack()`	Sends the object to the back of the stacking order
`sendForward()`	Sends the object forward one position in the stacking order
`sendBackward()`	Sends the object backward one position in the stacking order

Graphics

Additional Methods for `GOval` and `GRect`

Fillable shapes (`GOval` and `GRect` [and later `GArc` and `GPolygon`])

`setFilled(`*flag*`)`	Sets the fill state for the object (`false`=outlined, `true`=filled)
`isFilled()`	Returns the fill state for the object
`setFillColor(`*c*`)`	Sets the color used to fill the interior of the object to *c*
`getFillColor()`	Returns the fill color

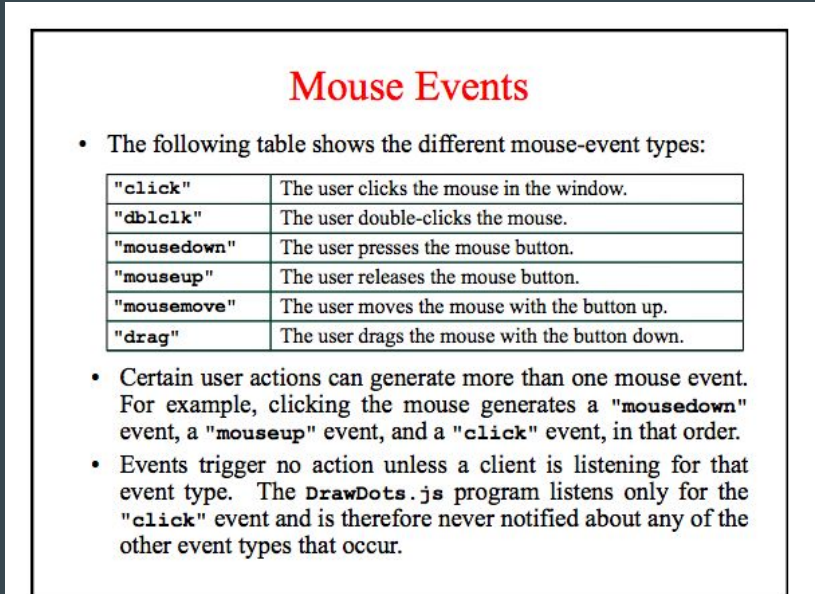
Resizable shapes (`GOval` and `GRect` [and later `GImage`])

`setSize(`*width*`,` *height*`)`	Sets the dimensions of the object as specified
`setBounds(`*x*`,` *y*`,` *width*`,` *height*`)`	Sets the location and dimensions together

Graphics

- Mouse events:

```
function listenerFunction(e) {
    console.log(e.getX());
}
gw.addEventListener("click", listenerFunction);
```



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

A Simple Line-Drawing Program

In all likelihood, you have at some point used an application that allows you to draw lines with the mouse. In JavaScript, the necessary code fits easily on a single slide.

```
import "graphics";

const GWINDOW_WIDTH = 500;
const GWINDOW_HEIGHT = 300;

function DrawLines() {
   var gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT);
   var line = null;
   var mousedownAction = function(e) {
      line = GLine(e.getX(), e.getY(), e.getX(), e.getY());
      gw.add(line);
   };
   var dragAction = function(e) {
      line.setEndPoint(e.getX(), e.getY());
   };
   gw.addEventListener("mousedown", mousedownAction);
   gw.addEventListener("drag", dragAction);
}
```

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 midterm"`
  - `exam.charAt(0) -> "T"`
  - `exam.charAt(5) -> "i"`
  - `exam.length -> 11`
  - `exam.indexOf("m") -> 4`

Strings

- Concatenation
  - Fancy word for combining strings together
  - Ex: `"Jerry Cain and " + "Eric Roberts"` -> `"Jerry Cain and Eric Roberts"`
- 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
    ```

Tips for 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