Week 1: HTML, CSS, JavaScript, Output, Variables, Functions, & Creating Animations w/Canvas

Dr. Jason Schanker

HTML

- **HTML** stands for Hypertext Markup Language; it's the language used to declare the **structure** of web pages
- In HTML, you Markup text by using tags, which define the structure and meaning of your text:
- Text is surrounded by (possibly multiple) opening and closing tags (e.g., `<title>`Title goes here`</title>`)
- Include `<!doctype html>` at top to let the browser know that you're using HTML5; surround rest of HTML by `<html>` & `</html>`

HTML (cont.)

- In between `<html>` and `</html>`, include header information between `<head>` and `</head>` (information *about* the web page such as title, language, where to look for styling information, etc.) and between `<body>` and `</body>` , include the *content and structure* of the page.
- Every time you open a tag (with a closing one), increase the indentation by one level of each subsequent line, and every time you close a tag, decrease the indentation by one level of each subsequent tag. Although such spacing is ignored by the browser, it makes it easier for you to read and make sure you know which content is marked up by which tags and that you close the tags that need to be.

HTML Starter Code

```
<!doctype html>
<html>
  <head>
    <title>Flappy Bird Game!</title>
    <meta charset = "utf-8">
    <meta name= "viewport" content = "width=device-width,
                                      initial-scale=1">
    <link rel = "stylesheet" href = "index.css">
  </head>
  <body>
    <canvas width = "320" height = "480" id = "game"></canvas>
    <script src = "index.js"></script>
  </body>
</html>
```

HTML Elements

- An element consists of an opening tag (generally) followed by content (generally), followed by a closing tag.
- Some elements called void elements, however do *not* mark up any content, and therefore should not have a separate tag (although they may be self-closing).
  - Example: The line break element make the text flow to the next line and do not mark up any content; use `<br>` or the self-closing `<br />`
- The content of an element *can contain* other elements. In such cases, the elements are said to be *nested*.
  - Example: The body element is nested in the html element.

Meta tags and attributes

- Attributes provide additional information about elements and are written in the form *name*="*value*" , placed *inside* the opening tag, separated by spaces.
- Meta tags provide information about the web page:
  - The charset expresses the character encoding (numbers used to represent letters, digits, spaces, punctuation, symbols, etc.); UTF-8 is a standard encoding for the web. Other character encodings may use different sequences of numbers to represent different symbols.
  - The meta tag specifying the viewport tells the browser that the width of the page is exactly the width of the device and that it should display it at 100% of its width. In other words, a mobile device should display the page as is instead of zooming in or out to try to compensate for a smaller or larger screen.

Canvas Element/Linking in Stylesheets

- The `<canvas>` element is used for allocating a space on a web page to draw shapes and images; the width and height attributes specify its dimensions in pixels, which can be thought of as tiny 1 X 1 rectangles for displaying colors. The better the resolution, the more pixels per sq. inch.
- While HTML is for specifying structure, **CSS (Cascading Style Sheets)** is for specifying the **presentation style**.
- You can link in external resources such as stylesheets, scripts, etc. using the `<link>` element. The `rel` attribute specifies the relationship between the HTML file and the thing we're linking in. In this case, we're linking in the stylesheet for the page and the `href` attribute specifies its location (if in the same directory, can just use its name: `index.css`).

CSS (Cascading Style Sheets)

- You tell how elements on your page should be styled via *rules* with *selectors* selecting the element(s) to style and property-value pair(s) which specify how the properties of the selected element(s) should be styled:

```
canvas {
  background-color: cyan;
}
```

- Note the form of the rule: selector first (`canvas` in this case) followed by a { followed by a series of property-value pairs in the form `property:value`, each terminated by a semicolon concluded with a closing }; this gives a cyan background color to *all* canvas elements
- Alternatively, you could use `#game` in place of `canvas` (# denotes id)

Viewing what you have

- Open any text editor such as Notepad (on Windows), TextEdit (on Mac), use a cool Text Editor such as Sublime Text (https://www.sublimetext.com).
- Create the file index.html (save it in a easily accessible directory) and enter the HTML from slide 4. Create a file index.css and save it in the same directory. Enter the CSS from the previous slide.
- Go to the directory where you saved the files and open the index.html file in Chrome (Right mouse click index.html and select the Open with option)
- For quick testing, you can also use https://repl.it/languages/web_project .

Introducing JavaScript (Statements, Output, Comments)

- You can display messages by using console.log. You can separate these statements with semicolons.

```
console.log("Welcome to the Wonderful World of JavaScript!");
console.log("The meaning of life is 42.");
```

- A message to be displayed as is needs to be placed in between a pair of double quotes so that it's not interpreted as a variable name (if bar is not declared, you get the following (`ReferenceError: bar is not defined`)

```
console.log(bar);
```

- Lines starting with `//` are known as single line comments. They can be used as notes which can provide context for your code. Comments are optional but useful; removing them will have no effect on how your program runs.

```
const a = 9.8; // a is approximate acceleration due to gravity in meters per second
```

Variables

- **Quick Exercise 1.1**: Write a statement to print a message to the console. It could be anything you want. Also, include a comment: https://repl.it/I9Xc/0
- Use a variable to name something stored in memory so that you can refer to it later.
- Use descriptive variable names in camel case: e.g., `firstName`, `bestGameScore`
- Assign variables with let statements:

```
let x = 7;                  // x gets assigned 7
let y = x + 5;              // y gets assigned 5 more than x (12)
let name = "Jane Doe";      // variables can be assigned words
```

- Assign with const for constants. Constants are sometimes expressed in all CAPS:

```
const PI = 3.141592653589793;
```

Variables (cont.)

- You can print the values of variables just as you do text using console.log, but they need to be placed outside of the quotes so that they're not interpreted literally:

```
console.log("x =", x, "and y =", y);
console.log("Hi,", name, ".");
console.log("PI =", PI);
```

- Quick Exercise 1.2: Define a variable meaningOfLife and set it to 42. Then write a statement to print the message, "The meaning of life is: 42." Do this in such a way so that when you change the value of the variable to 4 without changing anything else and run the program again, the message changes to "The meaning of life is: 4.": https://repl.it/I9Yh/0

Math Operators/Basic Math Functions

- You can use math operators with specific numbers or variables that refer to numbers: `+` for Addition, – for Subtraction, `*` for Multiplication, `/` for Division
- `Math.trunc(# / #)` : Integer Division (do the division and remove the decimal point and all digits that come after it; truncate it) - See Math.trunc()
- `%` for remainder (e.g., `14 % 5` will be 4, i.e., remainder when dividing 14 by 5)
- `**`: Exponentiation (NOT `^`, which is bitwise XOR); note this was standardized in 2016 so it may not work on older browsers; on these browsers you can use `Math.pow(#, #)` instead: e.g., **`2**10`** or **`Math.pow(2, 10)`** will be 2^{10} or 1024.
- Operations performed in PEMDAS order: (): Parentheses first; then Exponentiation (**), then Multiplication/Division (*, /, and % for remainder), then Addition/Subtraction (+/-)

Math Operators/Basic Math Functions (cont.)

```
//                                                    _2_R4
console.log(Math.trunc(14 / 5)); // prints 2 : 5|14

console.log(14 % 5);                  // prints remainder of 4
```

- **Quick Exercise 1.3**: Define a variable n and set it to 54. Then define a variable d, and set it to n - 44. Have the program compute the quotient of n divided by d (integer division) and output the result. Then have it compute the remainder when you divide n by d and output the result. Before running the program, determine what you think the output will be. Change d to be n + 44 instead. How do you think this will affect the output? Run it to find out if you were correct. Finally, change d to be n - 54 instead. How do you think this will affect the output? Run it to find out if you were correct: https://repl.it/I944/0

Defining/Calling Functions

- You can use variables to refer to functions as well. Define functions using arrow notation:

```
let double = x => 2*x;
```

  - The variable double refers to a function that takes x as an input and outputs two times it.
  - The variable `x` in the above expression is called a parameter variable. It is a placeholder for the input.
  - `2*x` is the return value or output of the function.
  - Note the `x` can be replaced with other variable names such as `t` or even `foo`. In those cases, you would use `t => 2*t` and `foo => 2*foo`, respectively.

Defining/Calling Functions (cont.)

- Evaluate functions at specific inputs by writing them as you would in your PreCalculus class. Use `console.log` to display the result:

```
console.log(double(0));      // outputs 0 : 0 => 2*0
console.log(double(5));      // outputs 10 : 5 => 2*5
console.log(double(-1.57)); // outputs -3.14 : -1.57 => 2*-1.57
```

- The inputs 0, 5, and -1.57 are called **arguments** and are said to be *passed* to the `double` function. To get the output, we *substitute* the argument for the parameter variable x and calculate what's to the right of the arrow. (E.g., for 5, we substitute 5 for x and calculate `2*5`, which is 10.)

Defining/Calling Functions (cont.)

- Functions that perform a computation such as `double` without changing anything are known as pure functions. `console.log` is a function but not a pure one. It does change something, mainly the console where you will see output displayed. When a function changes the state of something, it is said to have *side effects*. A pure function by definition has no side effects. If we wrote `double(5);` without using `console.log`, we would not notice that any computation was done.
- Functions can have more than one input. In these cases, you must use parentheses to group the parameter variables together and a comma in between them to separate them.

```
let rectangleArea = (length, width) => length * width;
console.log(rectangleArea(5, 2)); // logs 10 (5 * 2)
```

No Implicit Multiplication

- **IMPORTANT NOTE**: The `*` is required so you couldn't for example write `lengthwidth` or the computer would get confused. Did you mean one long variable `lengthwidth`? More generally, there is no implicit multiplication; to calculate 2x, you must explicitly include the `*` symbol so that would be `2*x`.

Quick Exercises

- Fill in the blank to define square as the squaring function (REMEMBER ^ is used for bitwise XOR, not for raising a number to a power). Do NOT use **; use multiplication instead:

```
let square = x => _________;
// Test it with the inputs 4, -5, 7, and 3.14; the ouputs
// should be 16, 25, 49, and 9.8596, respectively.
```

- Fill in the blank to define distance as the function that takes an input t representing the time in seconds and outputs the distance in meters that an object starting from rest would fall in t seconds. The formula is $d = 1/2\ at^2$ where a = 9.8 is the acceleration due to gravity.

```
const a = 9.8;
let distance = t => __________;
// Test it with the inputs 1, 2.4, and 0; the outputs
// should be 4.9, 28.224, and 0, respectively.
```

Quick Exercises (cont.)

- There are 1000 milliseconds in a second. Fill in the blank to define the function msToSec that takes a number of milliseconds as an input and returns the **whole** number of seconds:

```
let msToSec = (milliseconds) => _____________; // should return 5 for 5247
```

- Fill in the blank to define the distance travelled by a car going at a constant speed of v meters per second for t seconds in a single direction.

```
let distance = (v, t) => _________;
```

- Fill in the blank to define daysAfterSunday that takes an integer greater than or equal to 0, representing the days after Sunday and returns the day of the week it is (i.e., 0 = Sunday, 1 = Monday, 2 = Tuesday, …, 6 = Saturday)

```
let daysAfterSunday = days => __________;
// Test it with the inputs 3, 30, 14; the outputs
// should be 3, 2, and 0, respectively.
```

Objects, Methods, Creating and Drawing to the Canvas

- You can access the HTML Canvas element with the id game using `document.getElementById("game")`. A `<canvas>` element has a `getContext` method, which returns an object that provides methods for you to draw on it.
- An object is simply some virtual entity (a specific button on a page, your very first Farmville cow, if you purchased one, or in this case the game `<canvas>` element). In JavaScript, a method is simply a function that belongs to an object. When calling a method on the object, it may access or change it in some way.
  - Two cows `c1` and `c2` may exist as does a canvas element `gameCanvas`. Typing `c1.moo()` calls moo on `c1` and may make it moo while leaving `c2` silent. Typing `c2.moo()` to call moo on `c2` would do the opposite. Calling `gameCanvas.moo()` would result in an error; canvases don't have a `moo` method :-(, but you can add one.

Drawing to the Canvas

- Once you have a context object as follows, you may draw a 50 x 20 rectangle to your canvas at position (5, 10) as follows (Note: y *increases* as you go down):

```
let gameContext = document.getElementById("game").getContext("2d");
gameContext.fillStyle = "red"; // replace red with desired color
gameContext.fillRect(5, 10, 50, 20);
```

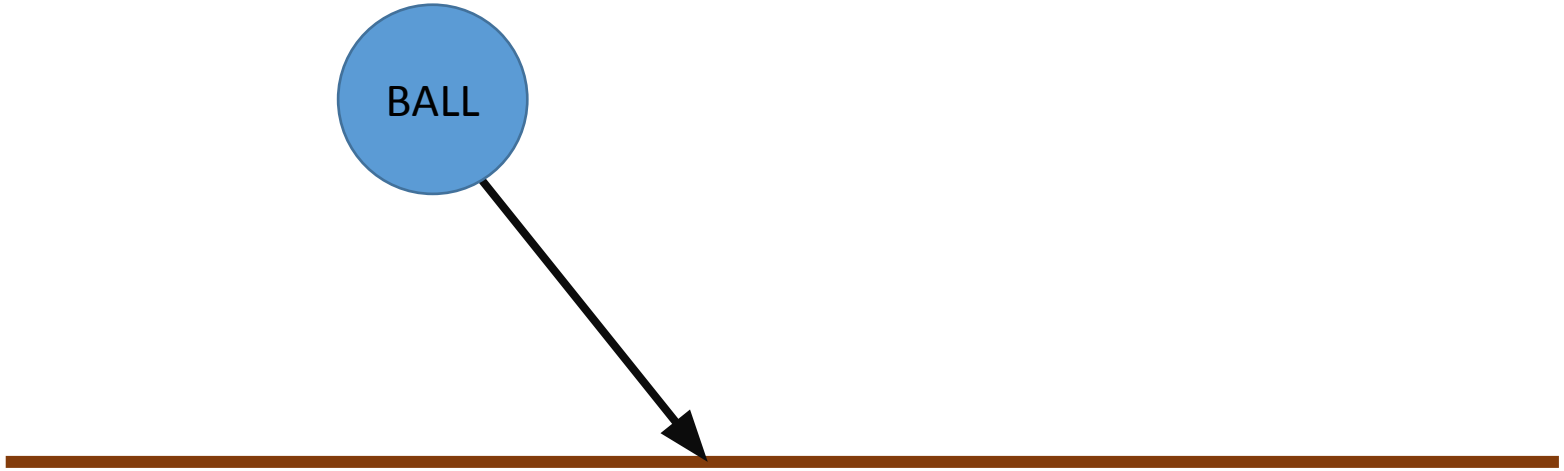
- You can clear the rectangle in the same way by calling the context's clear method as follows:

```
gameContext.clearRect(5, 10, 50, 20);
```

- Of course, if you fill it and clear it immediately, it'll occur so quickly, it'll be as if nothing happened. However, this is the key to making an animation (read on).
- For more on the methods you can call on a 2 dimensional context object, see: https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D

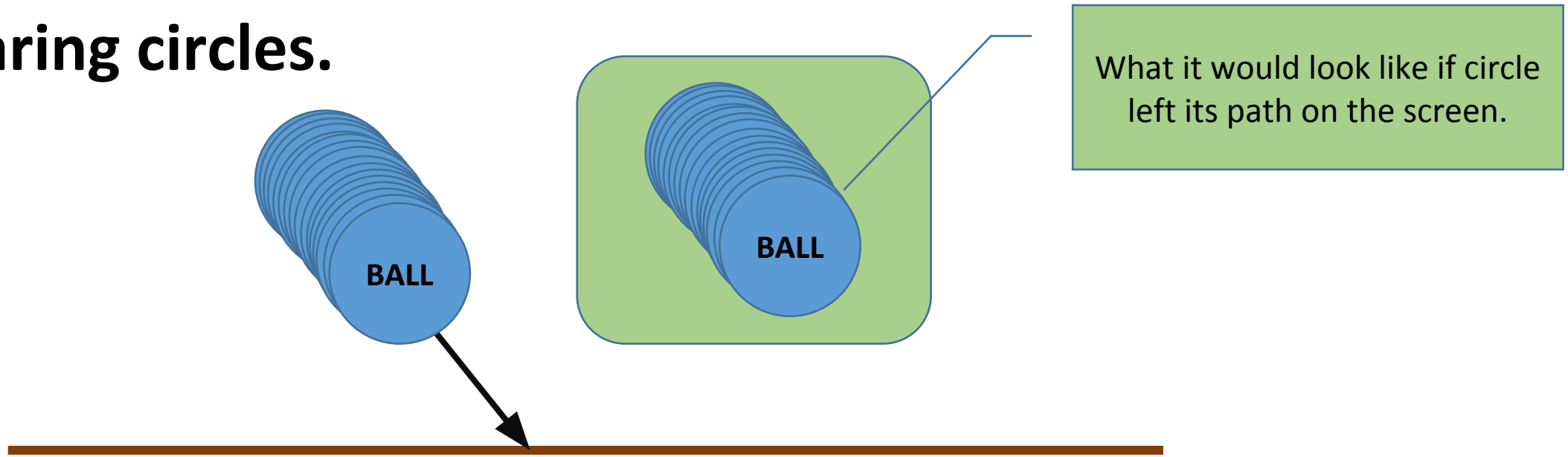
Illusion of Ball Movement

- **Q: How can we create ball movement in 2D Arkanoid Game?**



Creating Appearance of Movement

- **A: It's an illusion: x and y coordinates of circle (ball in game) change at regularly timed intervals (animation) by drawing and clearing circles.**



Successive still frames

Scheduling Functions at Repaints

- Although it might not look like it, the browser can be quite busy responding to all sorts of things such as button clicks, timers, keyboard presses, mouse moves, etc. With some exceptions such as when it gets a so-called Web Worker involved, the browser often responds to these events sequentially, one at a time in the order in which they occurred. If one event requires a long calculation to take place, it can freeze the browser just as one person with 500 items and 1000 coupons can hold up a check-up line when there's only one cashier. (Don't shop after 1am!)
- **Don't be that demanding function!** Animations require repeated updates of the screen to simulate movement, but rather than freeze the browser with repeated requests w/o breaks, we can repeatedly ask the browser to schedule a change whenever it'd naturally repaint the screen using `requestAnimationFrame`.

Using `requestAnimationFrame`

- The requestAnimationFrame function takes a reference to a function as an input and calls it with the number of milliseconds since the user visited this page. To see this in action, try entering the following in https://repl.it/babel :

```
let render = milliseconds => {
  console.log(milliseconds);
  requestAnimationFrame(render);
};
requestAnimationFrame(render);
```

- **Observation # 1**: Passing the reference to `render` in `requestAnimationFrame` at the end of the render function after displaying the number of milliseconds repeatedly schedules this function for the next time the browser repaints the screen. This means you'll see a stream of increasing numbers!

Using `requestAnimationFrame` (cont.)

- To see this in action, try entering the following in https://repl.it/babel :

```
let render = milliseconds => {
  console.log(milliseconds);
  requestAnimationFrame(render);
};
requestAnimationFrame(render);
```

- **Observation # 2**: The function definition requires an opening and closing brace; this is because the function consists of more than one statement.
- **Observation # 3**: The render function doesn't calculate anything; it's used purely for its side effects (i.e., printing to the screen and scheduling itself to be called again).

Cumulative Exercises: Successive Stills for Animation

- Modify your `render` function to show a rectangle moving horizontally at approximately 20 pixels per second. It'll eventually travel off the canvas. **Hint**: You may want to define a variable above your render function to keep track of the current x position and modify it in your render function. You may also want to call one or more of the functions you defined for a previous exercise.
- Now modify it so that it returns (approximately) to its starting position after travelling off the canvas. **Hint**: You may want to use `%`.

Cumulative Exercises: Flappy Bird in the Clouds

- Download the cloud-background.jpg file from Canvas and place it in the same directory as your other files. You can draw this image to the canvas at position (5, 10) as follows:

```
var clouds = new Image();
clouds.src = "cloud-background.jpg";
context.drawImage(clouds, 5, 10);
```

- Fill the canvas with clouds, animating the background so it looks as if you are flying forward (as the clouds are moving backward or left).

Post to GitHub

- Register for a GitHub account (https://www.github.com , create a repository with a README file, and drag all of the files you created from the folder, but not the folder itself, to your repository (index.html, index.css, index.js, and any image files).
- Then enable GitHub pages in Settings to show off your Animation Prowess! It'll be at https://*username*.github.io/*repositoryname*