Bonus: JavaScript Object-ball

- Due No Due Date
- Points 1
- Submitting a website url

 (https://github.com/learn-co-curriculum/phase-1-object-ball)  (https://github.com/learn-co-curriculum/phase-1-object-ball/issues/new)

Learning Goals

- Practice building nested objects
- Practice iterating over nested objects

Instructions

Great news! You're going to an NBA game. The only catch is that you've been volunteered to keep stats at the game.

Fork and clone this lab and open the `index.html` file in the browser. You'll be coding your solution in `src/00-objectball.js`. There are no tests for this lab — you'll need to use the browser's developer tools to check your work. There are instructions throughout this lesson on how to use `console.log` and debugger to test your code.

Once you're done, be sure to commit and push your code up to GitHub, then submit the assignment using CodeGrade. Even though this bonus lab does not have tests, it must still be submitted through CodeGrade in order to be marked as complete in Canvas.

Let's get started!

Part 1: Building the Object

The first function you will define is called `gameObject`. This function contains and returns an object nested in the following manner:

- The top level of the object has two keys: `"home"`, for the home team, and `"away"`, for the away team.
- The values of the `"home"` and `"away"` keys are objects too. These objects have the following keys:
  - `"teamName"`

    - `"colors"`
    - `"players"`
- The `teamName` key points to a string of the team name.
- The `colors` key points to an array of strings that are that team's colors.
- The `players` key points to an object of players whose names (as strings) are the keys to an object containing their stats. The values for each player's names and their stats can be found in the table below. The stats keys should be formatted like this:
    - `"number"`
    - `"shoe"`
    - `"points"`
    - `"rebounds"`
    - `"assists"`
    - `"steals"`
    - `"blocks"`
    - `"slamDunks"`

Use the following data to populate your `gameObject` as outlined above.

Home Team:

- Team name: Brooklyn Nets
- Colors: Black, White
- Players:

Stat	Info	Info	Info	Info	Info
Player Name	Alan Anderson	Reggie Evans	Brook Lopez	Mason Plumlee	Jason Terry
Number	0	30	11	1	31
Shoe	16	14	17	19	15
Points	22	12	17	26	19
Rebounds	12	12	19	12	2
Assists	12	12	10	6	2

Stat	Info	Info	Info	Info	Info
Steals	3	12	3	3	4
Blocks	1	12	1	8	11
Slam Dunks	1	7	15	5	1

Away Team:

- Team name: Charlotte Hornets
- Colors: Turquoise, Purple
- Players:

Stat	Info	Info	Info	Info	Info
Player Name	Jeff Adrien	Bismak Biyombo	DeSagna Diop	Ben Gordon	Brendan Haywood
Number	4	0	2	8	33
Shoe	18	16	14	15	15
Points	10	12	24	33	6
Rebounds	1	4	12	3	12
Assists	1	7	12	2	12
Steals	2	7	4	1	22
Blocks	7	15	5	1	5
Slam Dunks	2	10	5	0	12

To check your work, call your `gameObject` function and log its output:

```
console.log(gameObject());
/*
{
  home: {
    teamName: "",
```

```
      colors: [...],
      players: {
        "Alan Anderson": {...},
        "Reggie Evans": {...}
      }
    },
    away: {
      ...
    }
}
*/
```

Step 2: Building Functions

Calling Functions Within Functions

You'll be building a series of functions that operate on the object returned from the `gameObject` function. Each function will call the `gameObject` function and then return properties accessed off it.

For example, let's say we want to build a function, `homeTeamName`, that returns the name of the home team, `"Brooklyn Nets"`. We can call the `gameObject` function inside of our `homeTeamName` function and operate on the object:

```
function homeTeamName() {
  let object = gameObject();
  return object["home"]["teamName"];
}

console.log(homeTeamName());
// logs "Brooklyn Nets"
```

It's possible to access the properties of the object directly off the function call too. We don't necessarily need to save the result of the `gameObject` function into a variable. That being said, it's often a good idea to save results from functions into variables because it makes it easier to debug our

programs later. One-liners are not always better!

```
function homeTeamName() {
  return gameObject()["home"]["teamName"];
}

console.log(homeTeamName());
// logs "Brooklyn Nets"
```

Now that we understand how we are going to operate on the object that's returned from `gameObject` inside of other functions we're building, let's make sure we know how to debug our programs. Then we can start building those functions.

Debugging JavaScript

We have two options to debug our JavaScript programs:

1. Use `console.log()` to print out parts of our program.
2. Use a debug tool to pause our program, step through it and inspect values.

Using `console.log()` is an absolutely legitimate debugging technique. It can be nice because you can see lots of output from your program without it stopping and you can see how values change over time as you print them out. Using `console.log()` means we have to manually print out exactly what we want to see and we can't really change our minds in the middle of a debug session. Using `console.log()` also litters our program with lots of print statements so you'll find yourself deleting them once you're done debugging.

Using a debug tool will allow us to pause our program and interact in more powerful ways in the middle of a debugging session. We can set more breakpoints while we're debugging, type new code into the console to see what happens, and interactively investigate the value of different parts of our program.

There are two debug tools. One is built into `node` and it's simple. The other is built into Chrome and it's fantastically interactive.

When we run our programs in the command line using `node` we won't have access to a fancy debugger. Node does have it's own built-in debugger, but it's just not the best thing in the world. It's not intuitive. It's a bit hard to use. Check out the docs and see if you like it!

- **Node's debugger documentation (https://nodejs.org/api/debugger.html)**

When we're running command line applications in our terminal it would be natural to run our code with `node`. This means we won't have access to Chrome's fancy debugger.

Instead of using `node` to debug or command line JavaScript programs let's attach our JavaScript to an HTML page, open that HTML page in Chrome and leverage Chrome's amazing developer tools.

Here's a sample web page that doesn't have much content and primarily just attaches a JavaScript file. Chrome will execute the JavaScript and we can open Chrome's dev tools to debug our program.

```
<h2>JavaScript Objectball</h2>
<p>
  Open your Chrome dev tools to see console output and trigger the debugger to
  catch.
</p>

<script src="./src/00-objectball.js"></script>
<script src="./src/01-simple-debug.js"></script>
```

Open up the `01-simple-debug.js` file in the `src` folder and check out its content. We'll be debugging this file in the Chrome debugger.

We can include the `debugger` keyword to stop our program and inspect it with Chrome's dev tools.

Follow these steps to start the debugger:

- Open the `index.html` file.
- Press F12 to open the dev tools panel.
- Refresh the page with the dev tools panel open (press `CMD + R` on a Mac or `CTRL + R` on Windows or Linux to refresh).

Notice, the program will only ever stop if we have the dev tools open.

```
// dev tools must be open for the debugger to trigger
let x = 99;
let y = 42;
```

```
debugger;
console.log("x:", x);
```

- Look at the Sources pane to see your source code and see where the debugger has paused your program.
- Observe the Scope section to the right of your source code to see the current value of different variables in your program.
- Use your mouse to hover over variables in your source code to see their current value.
- Press ESC to toggle having the console appear at the bottom of your sources tab.
- Use the console to type in variable names and see their values.

Accessing Key Values and Iterating Through Objects

There are three ways to iterate through objects in JavaScript:

- Use a key you know and type the key as a string manually `oo['some_key']`
- Use a key you know and type it after a dot manually `oo.some_key`
- Notice that you can only access keys via the `.` if they have a name like a legal JavaScript variable (basically it can't have spaces, you'd have to use the string way)

```
let oo = { foo: 42, bar: 83, "key w/ spaces": true };
console.log(oo["foo"]);
console.log(oo["bar"]);
console.log(oo["key w/ spaces"]);

console.log(oo.foo);
console.log(oo.bar);

// you can't use the dot to access this key because it has spaces
// console.log(oo.key w/ spaces)
```

Use a `for..in` loop to iterate over all of the keys and access their value through brackets:

```
let oo = { foo: 42, bar: 83, baz: 79 };
for (let key in oo) {
```

```
  let value = oo[key];
  console.log("key:", key, "value:", value);
}
```

- Use built-in functions attached to the `Object` class to access keys, values, or entries:
  - `Object.keys(oo)` returns an array of all keys
  - `Object.values(oo)` returns an array of all values
  - `Object.entries(oo)` returns an array of arrays containing `[key, value]` together

```
let oo = { foo: 42, bar: 83, baz: 79 };
console.log("   Object.keys(oo) =>", Object.keys(oo));
console.log(" Object.values(oo) =>", Object.values(oo));
console.log("Object.entries(oo) =>", Object.entries(oo));
```

```
   Object.keys(oo) => [ 'foo', 'bar', 'baz' ]
 Object.values(oo) => [ 42, 83, 79 ]
Object.entries(oo) => [ [ 'foo', 42 ], [ 'bar', 83 ], [ 'baz', 79 ] ]
```

Iterating Through Deeply Nested Objects

Go to the `index.html` file and uncomment the third `<script>` element `02-advanced-debug.js`. Do this after you've built your `gameObject` function that returns an object with all the properties described in the beginning of this lab.

Open `index.html` in Chrome, open Chrome's developer tools and inspect the flow of this program. Get used to using Chrome's debug buttons like "play", "step over." These buttons control the debugger. Pressing "play" will have the program continue running until it hits another `debugger` keyword. Pressing "step over" will make the program execute code one statement at a time so you can step through your program bit by bit without adding `debugger` keywords everywhere.

```
// src/02-advanced-debug.js
function goodPractices() {
  let game = gameObject();
  for (let gameKey in game) {
```

```
      // are you ABSOLUTELY SURE what 'gameKey' is?
      // use the debugger to find out!
      debugger;
      let teamObj = game[gameKey];
      for (let teamKey in teamObj) {
        // are you ABSOLUTELY SURE what 'teamKey' is?
        // use debugger to find out!
        debugger;

        // what is 'data' at each loop through out this block?
        // when will the following line of code work and when will it break?
        let data = teamObj.player;
        for (let key in data) {
          debugger;
        }
      }
    }
  }
```

Play around with the debug tools around each `debugger` keyword until you get comfortable with the iteration. This should give you a stronger sense of how we iterate through so many levels of a nested object and what happens on each level. **Use this strategy of placing LOTS of `debugger` keywords when you iterate over things in order to investigate your program and solve this lab.**

Okay, *now* we're ready to build out functions:

Function Building

- Build a function, `numPointsScored` that takes in an argument of a player's name and returns the number of points scored for that player.
  - Think about where in the object you will find a player's `points`. How can you iterate down into that level? Think about the return value of your function.
- Build a function, `shoeSize`, that takes in an argument of a player's name and returns the shoe size for that player.
  - Think about how you will find the shoe size of the correct player. How can you check and see if a player's name matches the name that has been passed into the function as an argument?

- Build a function, `teamColors`, that takes in an argument of the team name and returns an array of that teams colors.
- Build a function, `teamNames`, that operates on the game object to return an array of the team names.
- Build a function, `playerNumbers`, that takes in an argument of a team name and returns an array of the jersey numbers for that team.
- Build a function, `playerStats`, that takes in an argument of a player's name and returns an object of that player's stats. Check out the following example of the expected return value of the `playerStats` function:

```
playerStats("Alan Anderson")
// returns:
{
  number: 0,
  shoe: 16,
  points: 22,
  rebounds: 12,
  assists: 12,
  steals: 3,
  blocks: 1,
  slamDunks: 1
}
```

- Build a function, `bigShoeRebounds`, that will return the number of rebounds associated with the player that has the largest shoe size. Break this one down into steps:
  - First, find the player with the largest shoe size
  - Then, return that player's number of rebounds
  - Remember to think about return values here. Use `debugger` to drop into your function and understand what it is returning and why.

Bonus Questions

Define functions to return the answer to the following questions:

1. Which player has the most points? Call the function `mostPointsScored`.
2. Which team has the most points? Call the function `winningTeam`.
3. Which player has the longest name? Call the function `playerWithLongestName`.

Super Bonus

1. Write a function that returns true if the player with the longest name had the most steals. Call the function `doesLongNameStealATon`.