

# JAVASCRIPT DEVELOPMENT

*Sasha Vodnik, Instructor*

# HELLO!

1. Pull changes from the `svodnik/JS-SF-12-resources` repo to your computer
2. Open the `14-closures-module-pattern` folder in your code editor

**JAVASCRIPT DEVELOPMENT**

---

# **CLOSURES & THE MODULE PATTERN**

# LEARNING OBJECTIVES

At the end of this class, you will be able to

- › Understand and explain closures.
- › Instantly invoke functions.
- › Implement the module pattern in your code.
- › Understand and explain Javascript context.

# AGENDA

- Closures
- IIFEs
- Module pattern
- `this`

---

## CLOSURES & THE MODULE PATTERN

---

# WEEKLY OVERVIEW

**WEEK 8**

Closures & the module pattern / CRUD & Firebase

**WEEK 9**

Deploying your app / Final project lab

**WEEK 10**

(holiday) / React

---

**CLOSURES & THE MODULE PATTERN**

---

# **HOMEWORK REVIEW**

---

# ACTIVITY

---



## **KEY OBJECTIVE**

---

- › Check in on Feedr project

## **TYPE OF EXERCISE**

---

- › Groups of 3-4

## **TIMING**

---

*6 min*

1. Take turns checking in on your Feedr progress — what do you have done? what's your next step?
2. If you're currently wrestling with a challenge, share it with your group members and brainstorm together how you might move forward. If you've hit a challenge and worked through it, describe the issue and how you resolved it.



---

# ACTIVITY

---



## **KEY OBJECTIVE**

---

- › Check in on final projects

## **TYPE OF EXERCISE**

---

- › Groups of 3-4

## **TIMING**

---

*6 min*

1. Describe your idea for your final project:
  - What API(s) will you use?
  - How will users interact with your app?
  - What will the DOM manipulation look like?
2. Share your next step. If you're not sure, share where you are right now and brainstorm with your group what next steps might look like.

## **Exit Ticket Questions**

1. What are 'mixins'?
2. You mentioned that classes are used in React... do most new frameworks use classes?
3. Is closure the same as scope?

---

## CLOSURES & THE MODULE PATTERN

---

# THE MODULE PATTERN



**OBJECT-  
ORIENTED  
CODE**



**CLOSURES**



**IIFES**

# **CLOSURES**

---

## CLOSURES & THE MODULE PATTERN

---

# THE MODULE PATTERN



**OBJECT-  
ORIENTED  
CODE**



**CLOSURES**



**IIFES**


# SCOPE

- › Describes the set of variables you have access to

## GLOBAL SCOPE

- ▶ A variable declared outside of a function is accessible everywhere, even within functions. Such a variable is said to have **global scope**.

a variable declared outside of the function is in the global scope




```
let temp = 75;
function predict() {
  console.log(temp); // 75
}
console.log(temp); // 75
```

## LOCAL SCOPE

- ▶ A variable declared within a function is not accessible outside of that function. Such a variable is said to have **local scope**.

```
let temp = 75;
function predict() {
  let forecast = 'Sun';
  console.log(temp + " and " + forecast); // 75 and Sun
}
```

a variable declared within a function is in the local scope of that function



```
console.log(temp + " and " + forecast);
// 'forecast' is undefined
```

a local variable is not accessible outside of its local scope





## BLOCK SCOPE

- ▶ A variable created with `let` or `const` creates local scope within any block, including blocks that are part of loops and conditionals.
- ▶ This is known as **block scope**.

`let` creates a local variable within any block, such as an `if` statement

```
let temp = 75;
if (temp > 70) {
  let forecast = 'It's gonna be warm!';
  console.log(temp + "! " + forecast); // 75! It's gonna be warm!
}
console.log(temp + "! " + forecast); // 'forecast' is undefined
```

a variable with block scope is not accessible outside of its block

# CLOSURES

- ▶ A **closure** is an inner function that has access to the outer (enclosing) function's variables.
- ▶ You create a closure by nesting a function inside another function.

## CLOSURES

- ▶ A **closure** is an inner function that has access to the outer (enclosing) function's variables.

```
function getTemp() {  
  let temp = 75;  
  let tempAccess = function() {  
    console.log(temp);  
  }  
  return tempAccess;  
}
```

the tempAccess()  
function is a  
closure

outer function  
getTemp() returns  
a reference to the  
inner function  
tempAccess()

# **BUILDING BLOCKS OF CLOSURES**

## **1. nested functions**

## **2. scope**

inner function has access to outer function's variables

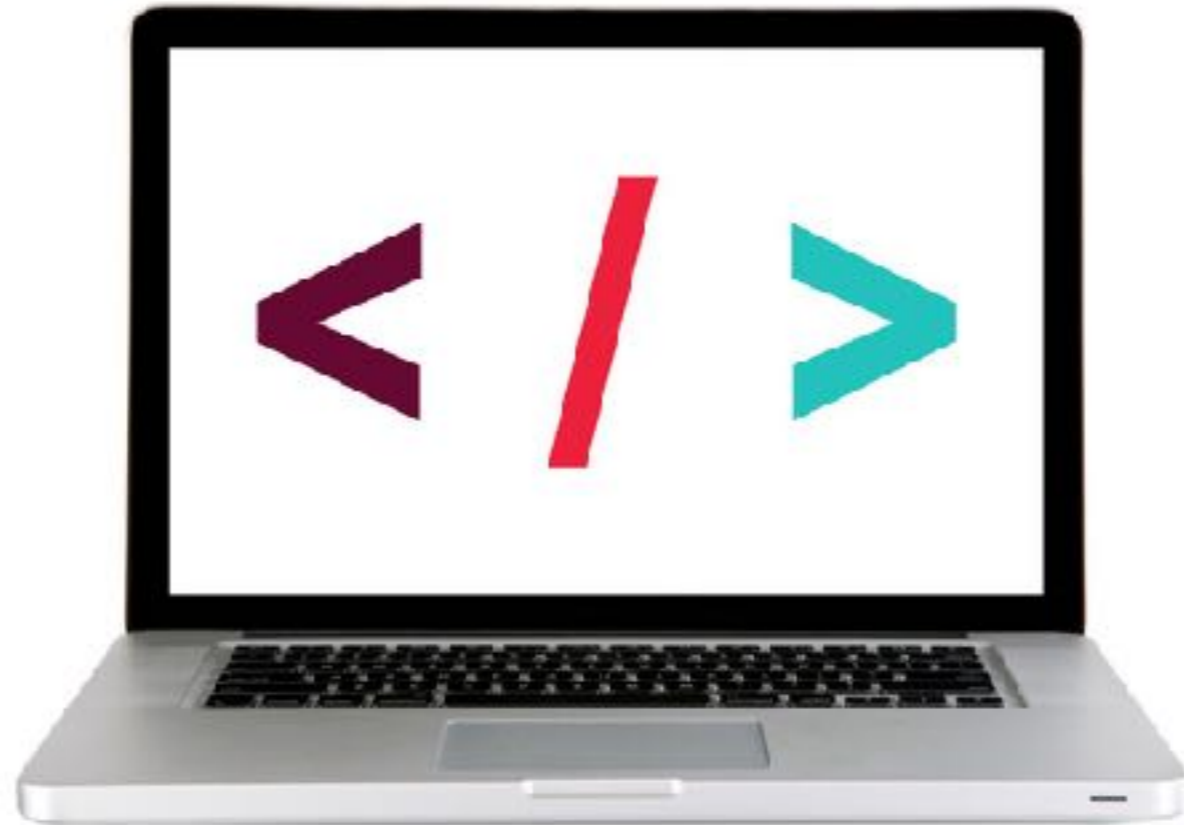
## **3. return statements**

outer function returns reference to inner function

---

**LET'S TAKE A CLOSER LOOK**

---



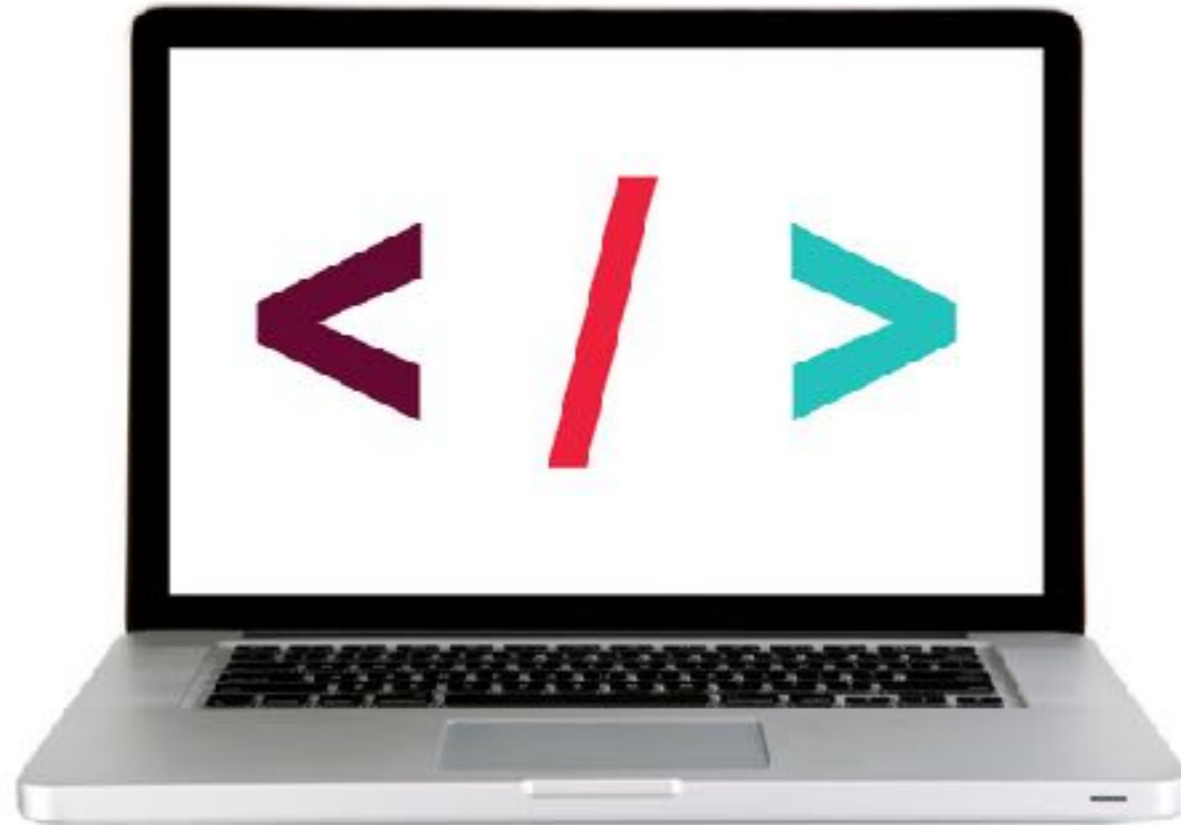
## **CLOSURES — KEY POINTS**

- ▶ Closures have access to the outer function's variables (including parameters) **even after the outer function returns.**
- ▶ Closures store **references** to the outer function's variables, not the actual values.

---

**LET'S TAKE A CLOSER LOOK**

---



## **WHAT ARE CLOSURES USED FOR?**

- Turning an outer variable into a private variable
- Namespacing private functions



# LAB — CLOSURES

---



## **KEY OBJECTIVE**

---

- ▶ Understand and explain closures

## **TYPE OF EXERCISE**

---

- ▶ Pairs

## **LOCATION**

---

- ▶ `starter-code > 1-closures-lab`

## **EXECUTION**

---

*15 min*

1. Follow the instructions in `app.js` to build and test code that uses a closure.

# Immediately-invoked function expressions

---

## CLOSURES & THE MODULE PATTERN

---

# THE MODULE PATTERN



**OBJECT-  
ORIENTED  
CODE**



**CLOSURES**



**IIFES**

# **Immediately-invoked function expression (IIFE)**

- A function expression that is executed as soon as it is declared
- Pronounced “iffy”

## IIFE based on a function expression

- ▶ Make a function expression into an IIFE by adding `()` to the end (before the semicolon)

```
let countdown = function() {  
  let counter;  
  for(counter = 3; counter > 0; counter--) {  
    console.log(counter);  
  }  
}();
```

## IIFE based on a function expression

- ▶ Make a function expression into an IIFE by adding `()` to the end (before the semicolon)

```
let countdown = function() {  
  let counter;  
  for(counter = 3; counter > 0; counter--) {  
    console.log(counter);  
  }  
}();
```

## IIFE based on a function declaration

- Make a function declaration into an IIFE by adding  
( at the start and  
)(); to the end

```
(function countdown() {  
  let counter;  
  for(counter = 3; counter > 0; counter--) {  
    console.log(counter);  
  }  
})();
```

## IIFE based on a function declaration

- Make a function declaration into an IIFE by adding  
( at the start and  
)(); to the end

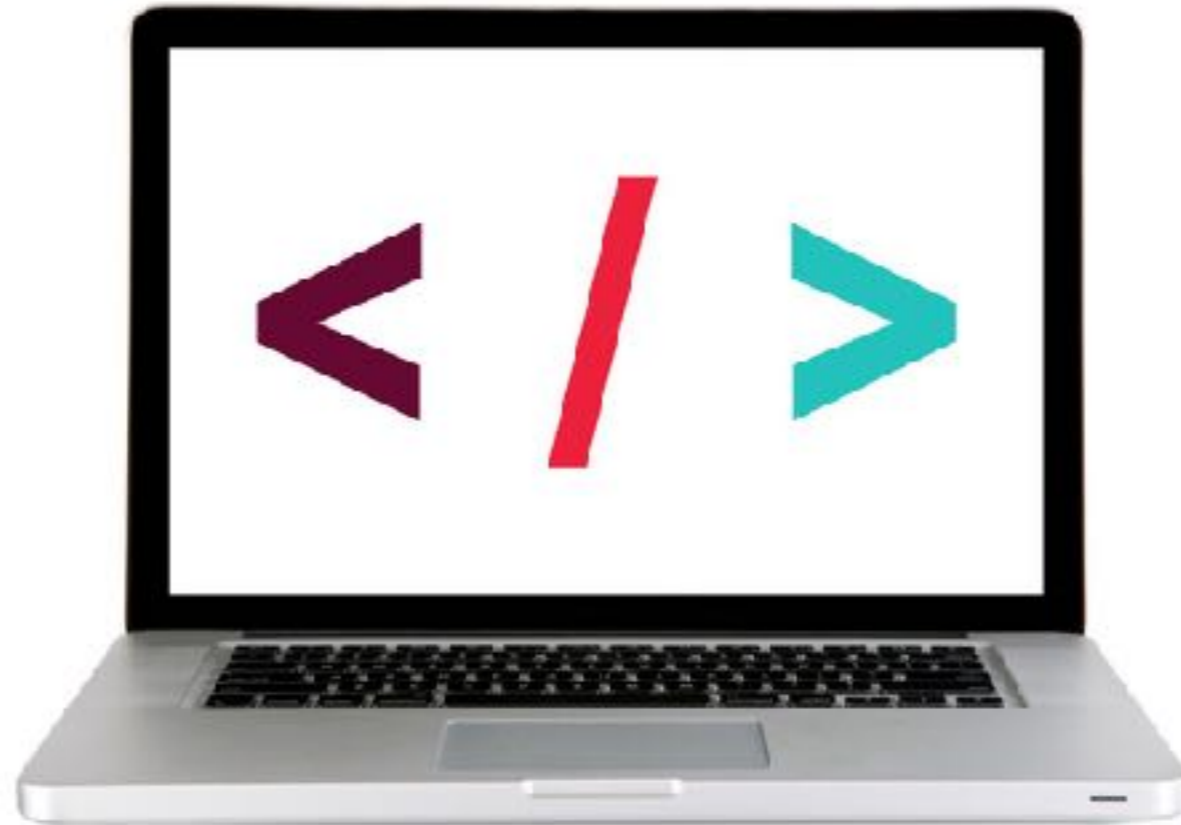
```
(function countdown() {  
  let counter;  
  for(counter = 3; counter > 0; counter--) {  
    console.log(counter);  
  }  
})();
```



---

**LET'S TAKE A CLOSER LOOK**

---



# **THE MODULE PATTERN**

---

## CLOSURES & THE MODULE PATTERN

---

# PUTTING IT ALL TOGETHER!



**OBJECT-  
ORIENTED  
CODE**



**CLOSURES**



**IIFES**

## **THE MODULE PATTERN**

- Using an IIFE to return an object literal
- The methods of the returned object can access the private properties and methods of the IIFE (closures!), but other code cannot do this
- This means specific parts of the IIFE are not available in the global scope

## BUILDING A MODULE

```
let counter = function() {  
  let count = 0;  
  return {  
    reset: function() {  
      count = 0;  
    },  
    get: function() {  
      return count;  
    },  
    increment: function() {  
      count++;  
    }  
  };  
}();
```

returning an object literal

from an IIFE

containing closures

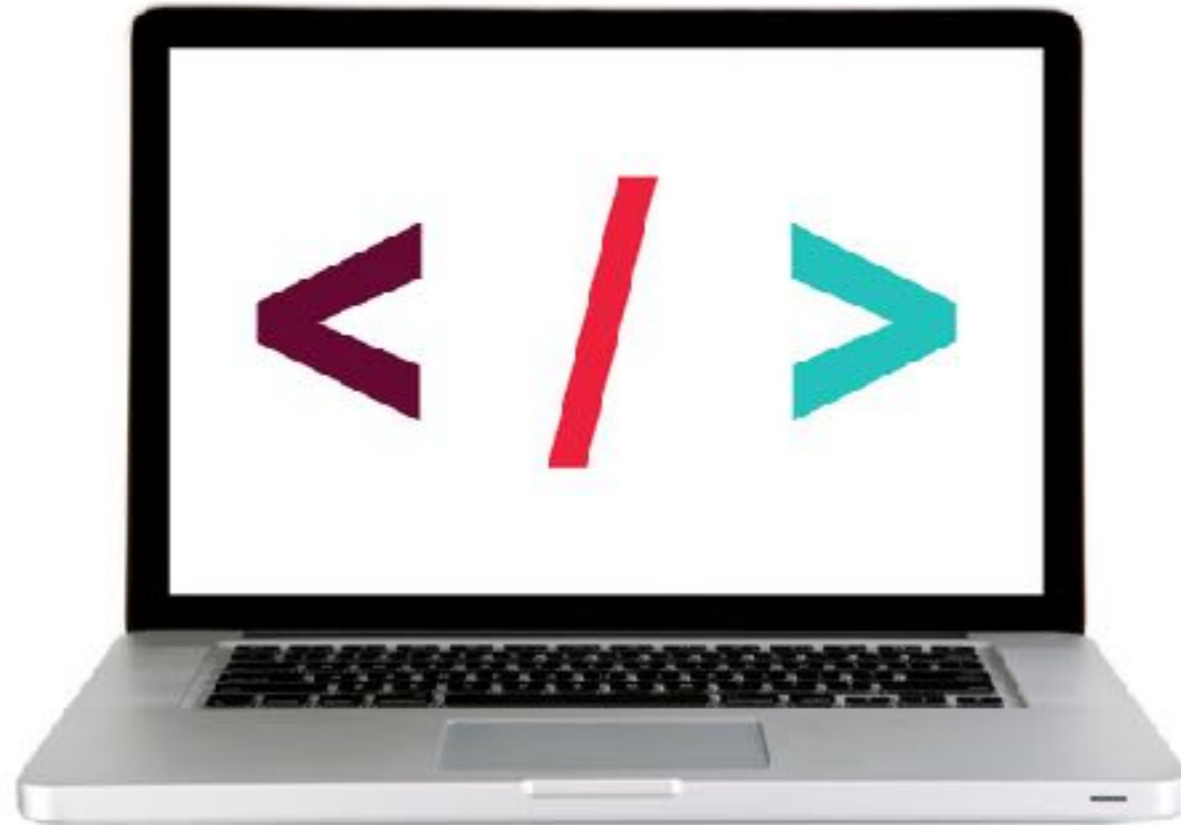
## **BENEFITS OF THE MODULE PATTERN**

- Keeps some functions and variables private
- Avoids polluting the global scope
- Organizes code into objects

---

**LET'S TAKE A CLOSER LOOK**

---



---

# EXERCISE — CREATE A MODULE

---



EXERCISE

## **TYPE OF EXERCISE**

---

▸ Pair

## **LOCATION**

---

▸ start files > 4-modules-exercise

## **TIMING**

---

*12 min*

1. In `app.js`, complete the module so it exports methods for the behaviors described in the comment at the top of the file.
2. When your code is complete and works properly, the statements at the bottom of the file should all return the expected values in the console.
3. **BONUS:** Add a "tradeIn" method that lets you change the make of the car and refuels it. Be sure the `getMake` method still works after doing a `tradeIn`.



**this**

## **CONTEXT AND THIS**

- Functions are always executed in relation to some object
- **Context** refers to whatever object is responsible for executing a function
- This object can be referenced using the keyword `this`
- In other words, `this` represents whatever object is in context when a function runs

## **HOW IS CONTEXT DECIDED?**

- At runtime
- Based on how the function is called

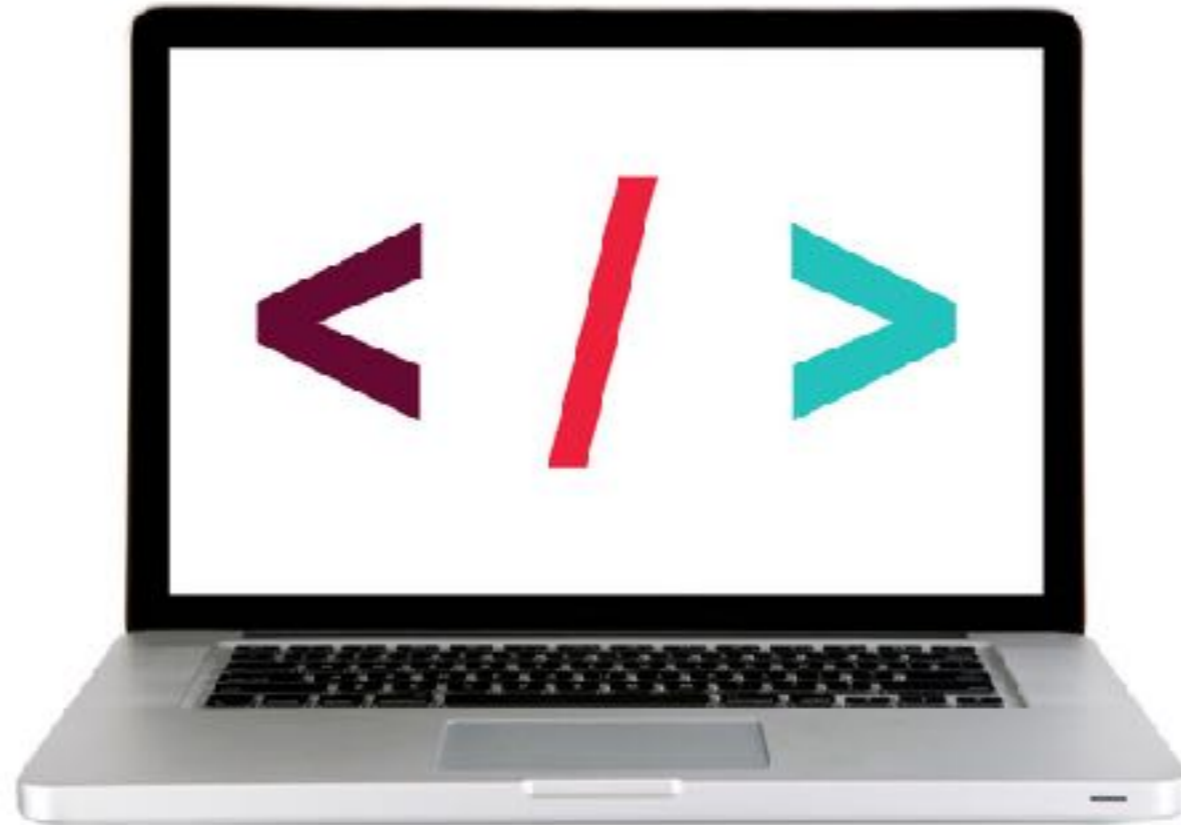
# CONTEXT RULES

situation	what <b>this</b> maps to
method invocation	the object that owns the method
constructor function	the newly created object
event handler	the element that the event was fired from
function invocation	default:       the global object (window) strict mode: undefined
arrow function	the context of the caller

---

**LET'S TAKE A CLOSER LOOK**

---



---

# EXERCISE — PREDICT CONTEXT

---



EXERCISE

## **TYPE OF EXERCISE**

---

- ▶ Groups of 2 or 3

## **LOCATION**

---

- ▶ `start files > 8-this-exercise`

## **TIMING**

---

*5 min*

1. In `app.js`, read through the code without running it.
2. Predict the results of the two `console.log` statements.
3. Run the code and check the results against your predictions. If the results were different, identify why.

# **Exit Tickets!**

**(Class #14)**

---

# **LEARNING OBJECTIVES – REVIEW**

- Understand and explain closures.
- Instantly invoke functions.
- Implement the module pattern in your code.
- Understand and explain Javascript context.



## **NEXT CLASS PREVIEW**

### **In-class lab: Intro to CRUD and Firebase**

- Explain what CRUD is. (**Preview:** Create, Read, Update, Delete)
- Explain the HTTP methods associated with CRUD.
- Implement Firebase in an application.
- Build a full-stack app.

# **Q&A**