

INFO 2124 JavaScript



```
1  const createSlideShow = () => {
2      const title = "Module 9";
3      const subTitle = "Functions, Closures, & Modules"
4      return {
5          getTitle () {
6              return `

# ${title}</h1> 7 <h2>${subTitle}</h2>`; 8 } 9 } 10 } 11 12 $(document).ready(() => { 13 const slideShow = createSlideShow(); 14 $('#container').html(slideShow.getTitle()); 15 });


```


Module Objectives

- A. Describe how default parameters work.
- B. Distinguish between the rest operator and the spread operator.
- C. Describe how the scope chain works in JavaScript.
- D. Describe the use of the module pattern.
- E. Describe the creation and use of namespaces.
- F. Use default parameters and rest parameters in your functions.
- G. Use closures, immediately invoked function expressions (IIFE), the module pattern, and ECMAScript modules in your applications.



A cartoon illustration of Kenny McCormick from South Park, wearing a brown suit and tie, standing in front of a window at night. A blue banner is overlaid on the image.

Objective 9A

Describe how default parameters work.

THE ORION PARAMETERS ARE

Let's Review . . .

- A *function* contains a block of code that can be *called* (or *invoked*) by other statements in the program.
- A *function declaration* is a function that starts with the *function* keyword.
 - e.g., **function myFunction() { ... }**
- A *function expression* uses the function keyword to create a function that can be assigned to a constant.
 - e.g., **const myFunction = function() { ... }**
- An *arrow function* uses the arrow operator (`=>`) to create a function.
 - e.g., **const myFunction = () => { ... }**



Let's Review . . .

- When JavaScript is executed, function declarations are *hoisted*
 - This means that function declarations can be coded after statements that call them, though this is bad practice.
- Remember, function expressions and arrow functions, in contrast to function declarations, are **not** hoisted.
 - So, function expressions and arrow functions must be coded prior to code that invokes them.



```
1 myFunction()  
2 function myFunction() {  
3   //do something  
4 }
```



```
1 //myFunction() //this no longer works  
2 const myFunction = function {  
3   //do something  
4 }  
5  
6 myFunction();
```



"Default" Parameters

- Up to this point, when we've declared functions, we either declare the functions with or without parameters.
- We have a third option, which is to declare a function with *default* parameters. Essentially, you can assign a default value to a function parameters. Then, if the code that invokes the function doesn't provide a value for that parameter, the function uses the default value. 🔥
- The default value can be a literal value or an expression. Parameters that come later in the parameter list can use the value of earlier parameters as their default value.
- If you pass a *null* value to a function, it is used instead of the default value. By contrast, if you pass an undefined value, it isn't used instead of the default value.
- In an arrow function, a single parameter without a default value must be in parentheses.
- A function can define multiple parameters with default values. These parameters aren't required to be coded at the end of the parameter list, though they usually are.
- To preserve the default values of earlier parameters, you can pass *undefined*.



A Function With a Default Parameter

```
1 //function definition -- note the default
2 //value for "taxRate"
3 const calculateTax = (subtotal, taxRate = 0.074) => {
4     const tax = subtotal * taxRate;
5     return tax.toFixed(2);
6 }
7
8 //Code that calls the function
9 const tax1 = calculateTax(100);
10 // tax1 is 7.40
11
12 const tax2 = calculateTax(100, 0.087);
13 // tax2 is 8.70
14
15 const tax3 = calculateTax(100, null);
16 // tax3 is 0.00
17
18 const tax4 = calculateTax(100, undefined);
19 // tax4 is 7.40
```



An Arrow Function With A Single Parameter And A Default Value

```
1 // note the default value for "places"
2 const getPI = (places = 15) => {
3     return Math.round(Math.PI * (10 ** places) ) /
4         (10 ** places);
5 };
6
7 //Code that calls the function
8 const pi = getPI();           // pi is 3.141592653589793
9 const pi2 = getPI(4);        // pi2 is 3.1416
```



A Function With Two Parameters That Have Default Values

```
1 //note the default values for
2 // "places" and "type"
3 const toPlaces = (num, places = 2, type = "round") => {
4   if (type == "floor") {
5     return Math.floor(num * (10 ** places) ) /
6       (10 ** places);
7   } else if (type == "ceil") {
8     return Math.ceil(num * (10 ** places) ) / (10 ** places);
9   } else {
10    return Math.round(num * (10 ** places) ) /
11      (10 ** places);
12   }
13 };
14
15 //Code that calls the function
16 const num1 = toPlaces(5.22873);           // num1 is 5.23
17 const num2 = toPlaces(5.22873, 4);        // num2 is 5.2287
18 const num3 = toPlaces(5.22873, 4, "ceil"); // num3 is 5.2288
19 const num4 = toPlaces(5.22873, "floor");  // num4 is NaN
20 const num5 = toPlaces(5.22873, undefined, "floor");
21                                           // num5 is 5.22
```



Additional Notes

- JavaScript does not require you to code default parameters at the end of the parameter list; however many other languages do require default parameters to be coded at the end of the parameter list.
- Your book notes that it's considered a "best practice" to code your default parameters at the end of the parameter list. I agree.

You can do
this...

```
1 function calculateTotal (taxRate = 0.03, subTotal, state = "undefined") {  
2   const total = (subTotal * taxRate) + subTotal;  
3   return `The total in ${state} is ${total}.`  
4 }
```

...but coding your
default parameters
at the end of the
parameter list is
best practice.

```
1 function calculateTotal (subTotal, taxRate = 0.03, state = "undefined") {  
2   const total = (subTotal * taxRate) + subTotal;  
3   return `The total in ${state} is ${total}.`  
4 }
```



A hand is holding a small, round, brown object, possibly a piece of wood or a nut, against a dark, textured background. The object has a rough, fibrous texture and a small hole in the center. The hand is positioned at the top of the frame, with the thumb and index finger visible. The background is a dark, mottled greenish-brown color.

Objective 9B

Distinguish between the rest operator and the spread operator.

The Rest Operator

- You can use the *rest operator* (`...`) to accept a variable number of parameters and put them into an array.
 - The parameter with the rest operator is called a *rest parameter*.
- A function can only define **one** rest parameter and it must be the **last** parameter in the parameter list.

```
1 // An Arrow Function With A Rest Parameter
2
3 const calculateTaxAll = (taxRate, ...subtotals) => {
4     let tax = 0;
5     for (let subtotal of subtotals) {
6         tax += subtotal * taxRate;
7     }
8     return tax.toFixed(2);
9 }
```



The Spread Operator

- You can use the *spread operator* (`...`) to pass the elements in an array to a *variadic function* as an individual parameters. A function call can use more than one spread operator, and the spread operator doesn't need to be the last argument.
 - *variadic function* is a function that accepts a varying number of parameters via the "rest operator."

```
1 // An Arrow Function With A Rest Parameter
2
3 const calculateTaxAll = (taxRate, ...subtotals) => {
4   let tax = 0;
5   for (let subtotal of subtotals) {
6     tax += subtotal * taxRate;
7   }
8   return tax.toFixed(2);
9 }
10
11 //invoking with a spread operator
12 const taxRateAndAmounts = [0.074, 100, 200, 400, 500];
13 const tax4 = calculateTaxAll(...taxRateAndAmounts);    // tax4 is 88.80
```



Rest vs. Spread

- The *rest* and *spread* operators look the same (...); however:
 - A *rest* operator is used within a function definition to define a variable number of items.
 - While a *spread* operator is used in the statement that invokes a function coded with a *rest* operator. The *spread* operator passes a variable list of items into the *rest* parameter.

Rest

Spread

```
1 // An Arrow Function With A Rest Parameter
2
3 const calculateTaxAll = (taxRate, ...subtotals) => {
4   let tax = 0;
5   for (let subtotal of subtotals) {
6     tax += subtotal * taxRate;
7   }
8   return tax.toFixed(2);
9 }
10
11 //invoking with a Spread operator
12 const taxRateAndAmounts = [0.074, 100, 200, 400, 500];
13 const tax4 = calculateTaxAll(...taxRateAndAmounts); // tax4 is 88.80
```





Objective 9C

Describe how the scope chain works in JavaScript.

Closures

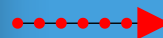
- In JavaScript, the *scope chain* refers to what is "*in scope*," or what can be seen and used by an object.
 - *global scope* - code can be "seen" anywhere
 - *local scope* - code is visible only inside the function where it is defined
- An object that is created within another object has access to its own scope as well as the scope of the object that contains it.
- An object in the scope chain is available, or "alive", as long as something is referring to it. This is true even if the object that contains it has finished executing and is *out of scope*.
- A *closure* is created when an inner function refers to one or more objects in the scope of the outer function that contains it.



A Closure

- In the snippet below, the *closure* is the *clickCounter* function, which has been created *inside* the *createClickCounter* function. Note how *clickCounter* refers to the variable *count*, which is in the scope of the outer function (*createClickCounter*).

```
1 // An example that illustrates a closure
2
3 const createClickCounter = () => { // outer function
4   let count = 0;                  // local variable
5
6   const clickCounter = evt => { // inner function
7     count++;
8     console.log(evt.currentTarget.id +
9       " count is " + count);
10  };
11  return clickCounter;
12 };
13
14 $(document).ready( () => {
15   $("#first_button").click(createClickCounter());
16   $("#second_button").click(createClickCounter());
17 });
```



first_button count is 1
second_button count is 1
second_button count is 2
second_button count is 3
first_button count is 2



Closures and Private State

- All variables, constants, and functions in global scope are publicly available.
- All the properties and methods of a JavaScript object that's in global scope are publicly available. That means that critical objects can be overwritten.
- You may recall from our discussion on JavaScript objects, unlike other languages, JavaScript does not provide a direct way to hide information in objects, except...
- You can use *closures* to create "private state", which can protect/hide variables, constants, and functions of an object.
- You create a closure by returning an inner function that refers to objects in the outer function.
- To return multiple inner function, you can return an object that contains the inner functions as its methods.



Closures and Private State



```
1 // A function that creates a closure with private state
2
3 const createSlideShow = () => {
4   // private variables and constants
5   let timer = null;
6   let play = true;
7   let speed = 2000;
8   const nodes = { image: null, caption: null };
9   const img = { cache: [], counter: 0 };
10
11   // private functions
12   const stopSlideShow = () => { ... };
13   const displayNextImage = () => { ... };
14
15   return {                                     // a public object
16     loadImages(slides) { ... },                // public method #1
17     startSlideShow(image, caption) { ... },    // public method #2
18     getToggleHandler(){ ... }                  // public method #3
19   };
20 };
```



Closures and Private State



```
1 // Code that creates and uses the slide show object
2 // returned by the function
3
4 // create the slideShow object
5 const slideShow = createSlideShow();
6
7 // call a public method from it
8 slideShow.loadImages(slides);
```



Closures and Private State



```
1 const SomeClosure = () => {  
2   let myVar = 4;           //private  
3 }  
4  
5 const sample = SomeClosure();  
6 console.log(sample.myVar);  //undefined
```

JavaScript + No-Library (pure JS) ▼

```
1 ▾ const SomeClosure = () => {  
2   let myVar = 4;  
3 }  
4  
5 const sample = SomeClosure();  
6 console.log(sample.myVar); //undefined
```

>_ Console (beta) Clear console Minimize

☁ "Running fiddle"

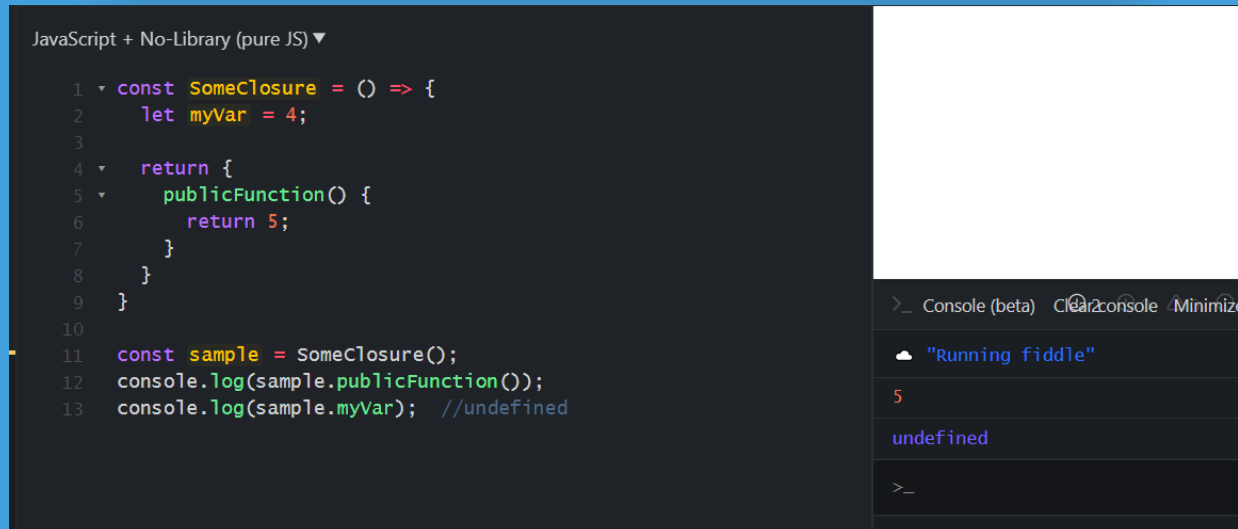
undefined

>_



Closures and Private State

```
1  const SomeClosure = () => {  
2    let myVar = 4;                //private  
3  
4    return {  
5      publicFunction() {          //public  
6        return 5;  
7      }  
8    }  
9  }  
10  
11 const sample = SomeClosure();  
12 console.log(sample.publicFunction());  
13 console.log(sample.myVar);        //undefined
```



JavaScript + No-Library (pure JS) ▼

```
1  const SomeClosure = () => {  
2    let myVar = 4;  
3  
4    return {  
5      publicFunction() {  
6        return 5;  
7      }  
8    }  
9  }  
10  
11 const sample = SomeClosure();  
12 console.log(sample.publicFunction());  
13 console.log(sample.myVar); //undefined
```

>_ Console (beta) Clear console Minimize

Running fiddle

5

undefined

>_





this

This just needs to be discussed.

"this" Is a Mess

- JavaScript's *this* keyword operates differently than the same keyword operates in other languages ("surprise"). JavaScript's *this* keyword has different values depending on where it is used:
 - In a method, *this* refers to the owner object.
 - Alone, *this* refers to the global object.
 - In a function, *this* refers to the global object.
 - In a function, in strict mode, *this* is "undefined".
 - In an event, *this* refers to the element that received the event.
 - An arrow function doesn't have its own *this* keyword. Instead, it always uses *this* to refer to its containing environment.



this in a Method

- In an object method, *this* refers to the "owner" of the method.
In the example below, *this* refers to the object literal "person".
The person object is the owner of the fullName method.

```
1  const person = {
2    firstName: "John",
3    lastName : "Doe",
4    id       : 5566,
5    fullName : function() {
6      return this.firstName + " " + this.lastName;
7    },
8    getThis : function() {
9      return this;
10   }
11 };
12
13 console.log(person.fullName());
14 console.log(person.getThis());           //outputs the person object
```



this Alone

- When used alone, the *owner* is the Global object (Window), so *this* refers to the Global object in non-strict mode.

In a browser window the Global object is [object Window]:



```
1 let x = this;  
2 console.log(x);
```

```
▶ Window {window: Window, self: Window, document: document, name: '', location: Location, ...} script.js:2
```



this Alone (Strict)

- In strict mode, when used alone, *this* also refers to the Global object [object Window]:

```
1 "use strict";  
2 let x = this;  
3 console.log(x);
```

```
▶ Window {window: Window, self: Window, document: document, name: '', location: Location, ...} script.js:2
```



this In a Function (Default)

- In a JavaScript function, the owner of the function is the default binding for *this*. So, in a function, *this* refers to the Global object [object Window].

```
1 function myFunction() {  
2     console.log(this);  
3 }  
4  
5 myFunction();
```

```
▶ Window {window: Window, self: Window, document: document, name: '', location: Location, ...} script.js:2
```



this In a Function (Strict)

- JavaScript **strict mode** does not allow default binding.
So, when used in a function, in strict mode, *this* is undefined.

```
1 "use strict";  
2 function myFunction() {  
3     console.log(this);  
4 }  
5  
6 myFunction();
```

undefined



this In Event Handlers

- In HTML event handlers, *this* refers to the HTML element that received the event:

```
1  /* HTML --
2  *  <button id="myButton">
3  *    Click to Remove Me!
4  *  </button>
5  */
6  $(document).ready(function() {
7      const buttons = document.querySelectorAll('button');
8      for (let button of buttons) {
9          button.addEventListener('click', function() {
10              this.style.display = 'none';
11          });
12      }
13 });
```

- This would cause the button defined on the HTML side to disappear when clicked ... but it comes with a caveat ...



this In Event Handlers

- REMEMBER an arrow function doesn't have its own *this* keyword. Instead, it always uses *this* to refer to its containing environment.
 - So, rewriting the event handler functions to use arrow functions, instead of function declarations, will cause the code below to fail because "this" will refer to the window, as opposed to the element that fired the event.

```
1  /* HTML --
2  *  <button id="myButton">
3  *    Click to Remove Me!
4  *  </button>
5  */
6  $(document).ready(() => {
7      const buttons = document.querySelectorAll('button');
8      for (let button of buttons) {
9          button.addEventListener('click', () => {
10              this.style.display = 'none';
11          });
12      }
13 });
```



this In Event Handlers

- *this* within event handlers defined as function expressions refers to the object that raised the event (so it works like a function declaration). Arrow functions are the only bad kid on the block.

```
1  /* HTML --
2  *  <button id="myButton">
3  *    Click to Remove Me!
4  *  </button>
5  */
6  $(document).ready(function() {
7      const clickHandler = function() {
8          this.style.display = 'none';
9      }
10     const buttons = document.querySelectorAll('button');
11     for (let button of buttons) {
12         button.addEventListener('click', clickHandler);
13     }
14 });
```



"this" In Closures

- Remember an arrow function doesn't have its own *this* keyword. Rather, when the *this* keyword is used inside an arrow function, it refers to the arrow function's containing environment (which could be the Global object itself, or "window").
- A regular function *does* have its own *this* keyword. Because of this, an inner regular function cannot access the out function's *this* keyword.
- When working with event handlers in closures, you could get around some of these issues by just using the *currentTarget* property of the *event* object to determine which object raised the event.
 - The same method can be used to determine what object raised an event when using arrow functions to write event handlers.



Button
module

Objective 9D

Describe the use of the module pattern.

Software Design Pattern

- "In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system."*



IIEFs

- An *immediately invoked function expression* (or *IIFE*) is a function that is defined and invoked in a single statement.
 - *IIFEs* provide a way to define and invoke a function in a single statement and they also provide a way to keep variables, constants, and functions out of scope and prevent name conflicts (stay tuned -- name conflict avoidance is an important concept).
- To define the function that's immediately invoked, you can use a function expression or an arrow function.
- If an *IIFE* is coded within a statement, you don't need to code parentheses around it.
 - It's generally considered best practice to code an *IIFE* within parentheses to help other programmers recognize the *IIFE*.
- Like a regular function call, an *IIFE* ends with opening and closing parentheses that can include arguments.



IIFEs ...

- Normally, we define functions, then invoke them ...

```
1  const sayHello = function() { // define function
2      console.log("Hello");
3  };
4  sayHello();                    // invoke function
```

- An *IIFE*, however, is defined and invoked in a single statement ...

```
1  (function() {                  // define and invoke function
2      console.log("Hello");
3  })();
```

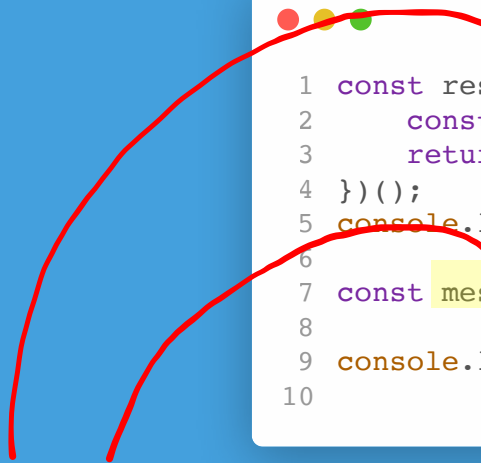
- ... this can also be done with an arrow function ...

```
1  ( () => {                      // define and invoke function
2      console.log("Hello");
3  })();
```



IIFEs ...

- ... help keep variables, constants, and functions out of global scope.
- ... help prevent name conflicts
- ... can be used to create private state for objects.
- Here's an *IIFE* that prevents a name conflict:



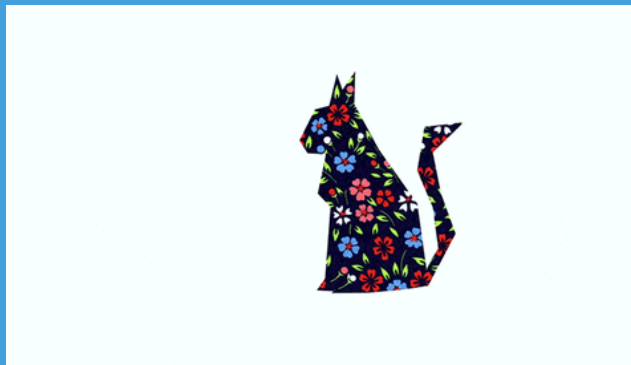
```
1  const result = ( () => {  
2      const message = "Thanks!";  
3      return message;  
4  })();  
5  console.log(result); // displays "Thanks!"  
6  
7  const message = "Have a great weekend!";  
8                      // no name conflict  
9  console.log(message); // displays "Have a great weekend!"  
10
```

The *message* constant is
in a different scope in
each of these instances!



The Module Pattern

- A *singleton* is a pattern that creates a single instance of an object. The easiest way to create a singleton in JavaScript is to use an object literal.
 - Object literals are problematic because there's no easy way to create private state.
 - Remember, you can use a closure to solve the issue of private state; however, closures allow for the creation of multiple instances.
 - This is problematic if you're looking to implement the singleton pattern.
- The *module pattern* uses an *Immediately Invoked Function* (IIFE) to create a single instance of the object, thus creating a *module*, that's returned by the function.
 - This gives you the benefits of an object literal, while also providing the private state of a closure.



A Module Pattern That Creates An Object With Private State

```
1  const slideshow = ( () => {
2    // private variables and constants
3    let timer = null;
4    let play = true;
5    let speed = 2000;
6    const nodes = { image: null, caption: null };
7    const img = { cache: [], counter: 0 };
8
9    // private methods
10   const stopSlideshow = () => { ... }
11   const displayNextImage = () => { ... }
12
13   // public methods
14   return {
15     loadImages(slides) { ... },
16     startSlideshow(image, caption) { ... },
17     getToggleHandler(){ ... }
18   };
19 })(); // Invoke the IIFE to create the object
```





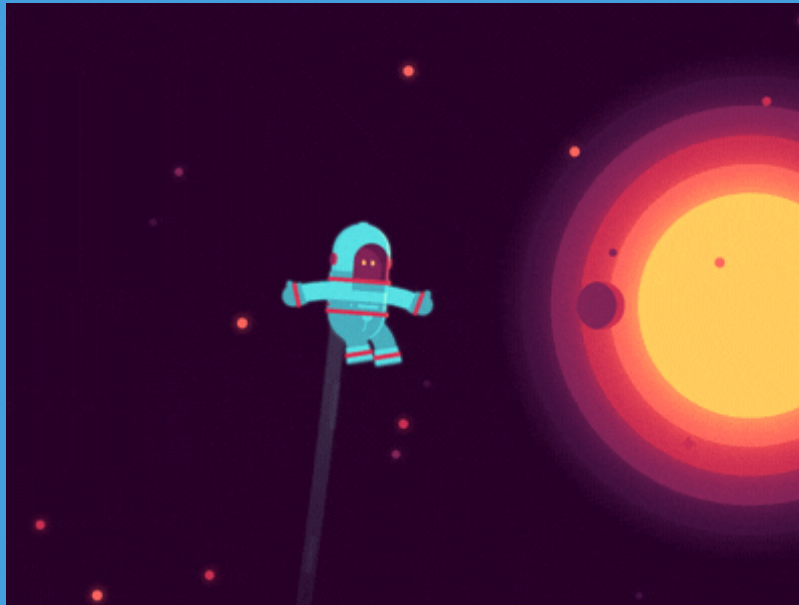
Objective 9E

Describe the creation and use of namespaces.

1h Reply

Namespaces

- When working with the *module* pattern, it's common to also use a *namespace*.
 - A namespace is a way to keep your objects out of the global namespace. This helps you avoid name conflicts. For example, what if you create a variable or constant that has the same name as another variable or constant defined in a library? Namespaces provide the means to avoid this collision/conflict.
- To create a namespace, you assign an object literal to a constant or to a variable.
- Once the namespace has been created, you can add one or more modules to the namespace.



Creating a Namespace

- To create a namespace, you assign an object literal to a constant or to a variable.

```
1 const myapp = {};
```

The *myapp* Namespace



Adding a Module to a Namespace

- Once the namespace has been created, you can add one or more modules to the namespace.

```
1  const myapp = {}; //create the namespace
2  myapp.moduleName = ( () => {
3
4      //define the IIFE
5
6
7  }) (); //invoke the IIFE
```

Adding a Module to
the *myapp* Namespace



Avoiding Overwriting Existing Namespaces

- If you want to avoid a namespace collision, you can use the snippet of code show below (where *myapp* is the name of the namespace).
 - This test requires the use of the *var* keyword and it is the only time in this course use of the *var* keyword will be permitted.

```
1 var myapp = myapp || {};
```

Checking for the Presence of the
myapp Namespace



Fin

Make sure to review the lecture videos and lab demos before starting your homework. The videos will provide more information on topics discussed in this presentation.

