CSCI-4208: Developing Advanced Web Applications

Week 3: Lecture 5

JavaScript - Basics

Overview: JavaScript - Basics

1. Programming Paradigms
2. Comments & Console Logs
3. Data Types
4. Variables & Scoping Rules
5. Operators & Processing operations
6. Type checking & Casting
7. Control Flow: loops & selections
8. Functions: declarations, expressions, arrow
9. Advanced Scoping: Closures & Hoisting

1. Programming Paradigms

JS is an interpreted language that supports many different paradigms

- Interpreter, executes an instruction at a time, not compiled

Common Approaches to Structure your Codebase:

- Imperative - Sequential list of commands that execute in order.
- Procedural -Split instructions into procedures (or functions).
- Object Oriented - Split instructions into Objects that contain properties & behaviors.
- Functional - Procedural calls that avoid any global state.

2. Comments & Console Logs

Comments

Inline comment

```
// this is an inline comment
```

Multi-line comment

```
/*
    this is a multi-line comment
*/
```

2. Comments & Console Logs

Console Logs

Console log (single parameter)

```
> console.log("Hello World");
  Hello World
```

Console log (multiple parameters)

```
> console.log("Number:", 10, "Boolean:", true );
  Number: 10 Boolean: true
```

3. Data Types (Primitive)

- **Number** or BigInt (primitive)
- **String** or Symbol (primitive)
- **Boolean** (primitive)

- Object (reference)
- Function (reference)

- Undefined (Non-Type)
- Null (Non-Type)

Everything in JavaScript is an object, including functions.

3. Data Types: Non-Types

Undefined vs. Null vs. NaN

`Null`	A null value represents a reference that points to a nonexistent or invalid object or address. Even though it points to something non-existing, nothing, it's a global object.
`Undefined`	Represents the primitive value of undefined. It tells us that something isn't defined. You get this e.g. by displaying a value of variable which don't have assigned value.
`NaN`	The global NaN property is a value representing Not-A-Number

Similarities	Differences
• both when negated are giving true (falsy values), but none of them equals true or false • they represent something non-existing…	• null represents "nothing". undefined something that isn't defined • undefined has its own data type, null is an object • null is treated as 0 in basic arithmetic operations, undefined returns NaN

3. Data Types: Strings

The String object is used to represent and manipulate a sequence of characters.

Basic String

```
"This is an example of a string literal"
```

```
'This is also an example of a string literal'
```

```
'nesting "double" quotes into a string'
```

```
"nesting 'single' quotes into a string"
```

3. Data Types: Strings

The String object is used to represent and manipulate a sequence of characters.

String Template

```
`nesting "double" quotes & 'single' quotes into a string`
```

```
`embed values/variables/expressions into strings: ${10*3}`
```

```
`string templates also maintain all original formatting:
new lines, tabs, and spaces for everything between tics`
```

3. Data Types: Numbers

The Number object is used to represent and manipulate numbers.

Representation	Example
float	`3.14159`
decimal	`9999999`
binary	`0b1011` → `11`
octal	`0o77` → `63`
hexadecimal	`0xff` → `255`
scientific notation	`1e+10` → `10000000000`, `1e-5` → `0.00001`
NaN	`Number('A')` → `NaN`, Not a Number,

3. Data Types: Numbers

The Number object is used to represent and manipulate numbers.

Arbitrarily large integers

```
> 1000000000000000000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000000000000000000000
  000000000000000000000000000000000000000000000000000000000000000000000000 * 1.79
< 1.79e+308
```

```
> 1000000000000000000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000000000000000000000
  000000000000000000000000000000000000000000000000000000000000000000000000 * 1.8
< Infinity
```

3. Data Types: Booleans

The Boolean object is used to represent and manipulate true/false values

Any value *(Strings, Numbers, Objects)* may be cast into Boolean values:

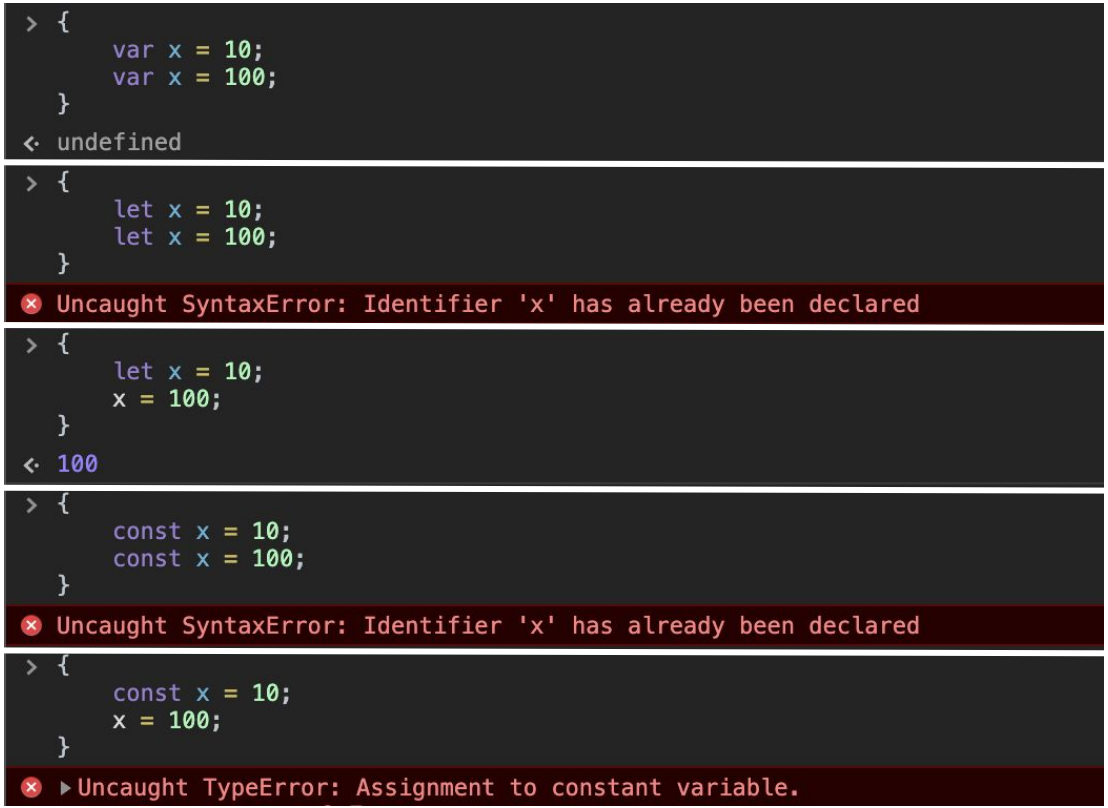
`false`	0, -0, null, false, NaN, undefined, or the empty string ("")
`true`	All other values, including any object, an empty array ([]), or the string "false"

4. Variables & Scoping Rules: var, let, const

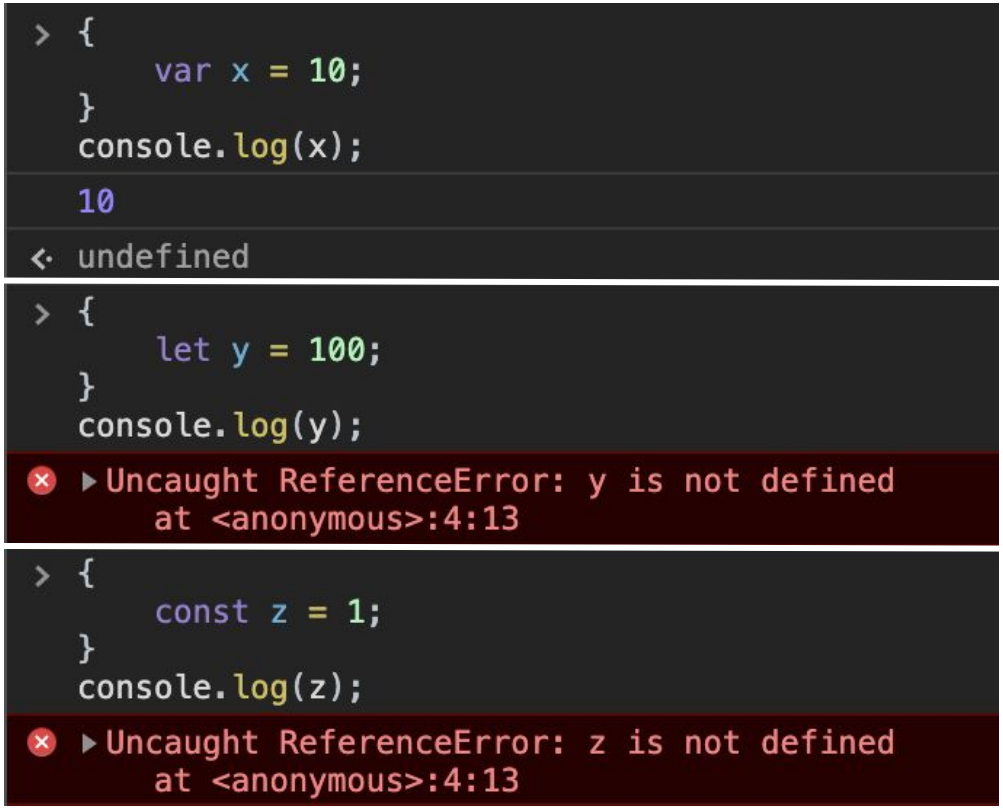
Variable Types (Keywords)

`var`	function-scope, allows redeclaration, allows overwrite
`let`	block-scope, no redeclaration, allows overwrite
`const`	block-scope, no redeclaration, no overwrite

4. Variables & Scoping Rules: re-declarations



4. Variables & Scoping Rules: block scopes



4. Variables & Scoping Rules: scopes

What is Scope?
Scope determines the visibility or accessibility of a variable or other resource in the area of your code.

JavaScript scope defined by curly braces { }

Default scope for Browser Interpreter is within Window

```
> this
Window
```

Note: Try declaring a variable at global scope and check the Window object

4. Variables & Scoping Rules: scopes

Global Scope

There's only one Global scope. The area outside all the functions is consider the global scope and the variables defined inside the global scope can be accessed and altered in any other scopes.

```
//global scope
var fruit = 'apple'
console.log(fruit);          //apple

function getFruit(){
    console.log(fruit);      //fruit is accessible here
}

getFruit();                  //apple
```

4. Variables & Scoping Rules: scopes

Function Scope

Variables declared inside the functions become Local to the function and are considered in the corresponding local scope. Every Functions has its own scope. Same variable can be used in different functions because they are bound to the respective functions and are not mutual visible.

```
//global scope
function foo1(){
    //local scope 1
    function foo2(){
        //local scope 2
    }
}

//global scope
function foo3(){
    //local scope 3
}

//global scope
```

4. Variables & Scoping Rules: scopes

Function Scope

Whenever you declare a variable in a function, the variable is visible only within the function. You can't access it outside the function. var is the keyword to define variable for a function-scope accessibility.

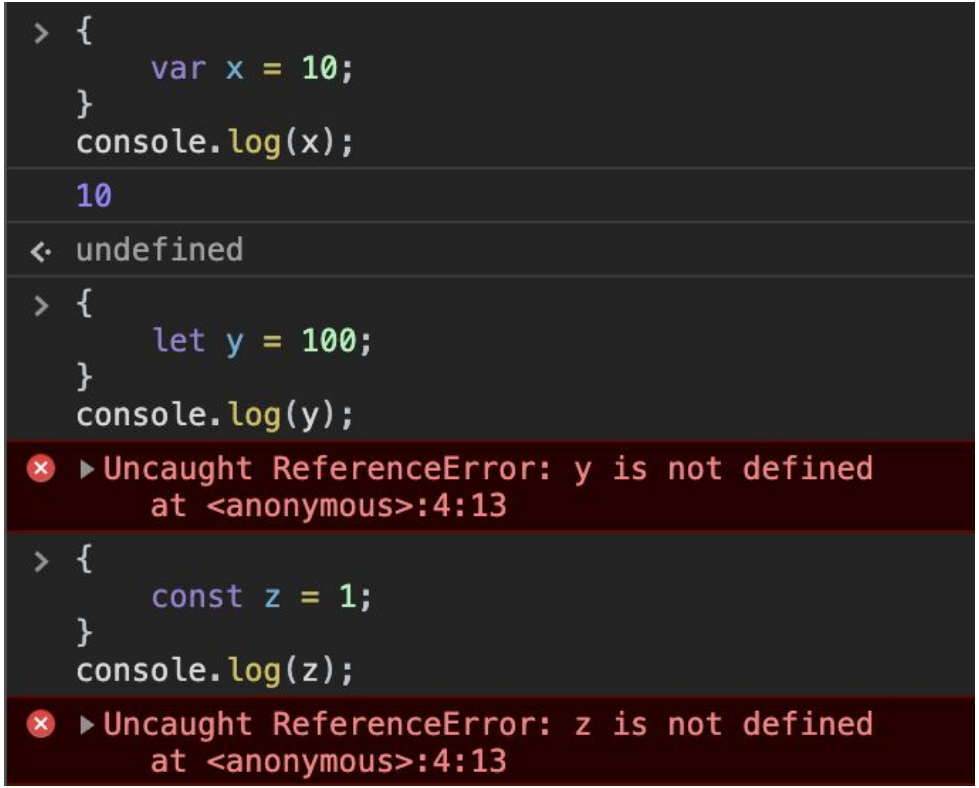
```
function foo(){
    var fruit ='apple';
    console.log('inside function: ',fruit);
}

foo();                  //inside function: apple
console.log(fruit);     //error: fruit is not defined
```

4. Variables & Scoping Rules: scopes

Block Scope

Blocks are defined by {curly brackets}. const and let keywords allow developers to declare variables in the block scope, which means those variables exist only within the corresponding block.



5. Operators & Processing operations

- Arithmetic
- Relational
- Equality
- Logical
- Bitwise
- Others

5. Operators: Arithmetic

Arithmetic operators take numerical values (either literals or variables) as their operands and return a single numerical value. Boolean values also works!

Name	Symbol	Data In	Data Out
Addition	**+**	**Number**	**Number**
Subtraction	**-**	**Number**	**Number**
Multiplication	*****	**Number**	**Number**
Division	**/**	**Number**	**Number**
Modulus	**%**	**Number**	**Number**
Exponentiation	******	**Number**	**Number**

5. Operators: Relational

A comparison operator compares its operands and returns a Boolean value based on whether the comparison is true. Note: You may chain relational operations.

Name	Symbol	Data In	Data Out
less than	**<**	**Number, String, Boolean**	**Boolean**
greater than	**>**	**Number, String, Boolean**	**Boolean**
less than or equal	**<=**	**Number, String, Boolean**	**Boolean**
greater than or equal	**>=**	**Number, String, Boolean**	**Boolean**

5. Operators: Equality

The result of evaluating an equality operator is always of type Boolean based on whether the comparison is true. Note: You may chain equality operations.

Name	Symbol	Data In	Data Out
equality	**==**	**Any**	**Boolean**
identity	**===**	**Any**	**Boolean**
inequality	**!=**	**Any**	**Boolean**
nonidentity	**!==**	**Any**	**Boolean**

5. Operators: Logical

Logical operators are typically used with boolean (logical) values, and when they are, they return a boolean value. Note: You may chain logical operations.

Name	Symbol	Data In	Data Out
and	**&&**	**Boolean**	**Boolean**
or	**||**	**Boolean**	**Boolean**
xor	**^**	**Boolean**	**Boolean**
not	**!**	**Boolean**	**Boolean**
Nullish Coalescing	**??**	**Boolean**	**Boolean**

5. Operators: Bitwise

Bitwise operators treat their operands as a set of 32 bits (zeros and ones) and return standard JavaScript numerical values. You may chain bitwise operations.

Name	Symbol	Data In	Data Out
Bitwise left shift	**<<**	**Number**	**Number**
Bitwise right shift	**>>**	**Number**	**Number**
Bitwise unsigned right shift	**>>>**	**Number**	**Number**
Bitwise AND	**&**	**Number**	**Number**
Bitwise OR	**|**	**Number**	**Number**
Bitwise XOR	**^**	**Number**	**Number**

5. Operators: Others

The conditional operator returns one of two values based on the logical value of the condition. The concatenation operator merges NaNs into a String

Name	Symbol	Data In	Data Out
Concatenation	**+**	**NaN**	**String**
Conditional	**condition ? t : f**	**Boolean**	**Any**

6. Type Checking & Casting

- typeof
- instanceof
- casting

6. Type Checking & Casting: typeof

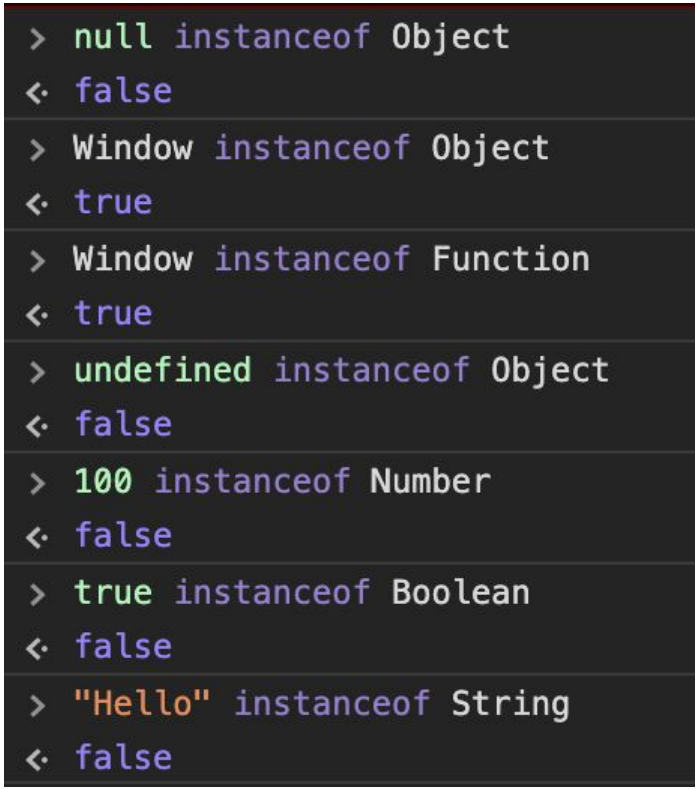
```
> typeof 1
<· "number"
> typeof 'Hello World'
<· "string"
> typeof true
<· "boolean"
> typeof Math
<· "object"
> typeof Window
<· "function"
```

```
> typeof null
<· "object"
> typeof undefined
<· "undefined"
> typeof NaN
<· "number"
```

6. Type Checking & Casting: instanceof

instanceof returns a boolean after checking the reference types. Not for primitive types.

We'll talk more about this once we get to Classes & OOP



6. Type Checking & Casting: Casting

Show Examples:

- Number() or +
- Boolean() or !!
- String() or `${}`

7. Control Flow: loops & selections

- single selection
  - if
- double selection
  - if, else
- multi-selection
  - nested if, else
  - switch
- while, do...while
- for
  - basic
  - for in
  - for of
- continue, break

7. Control Flow: Single Selection

```
> if (true){
      console.log('A');
  }
  A
```

7. Control Flow: Double Selection

```
> if (false){
      console.log('A');
  } else{
      console.log('B');
  }
  B
```

7. Control Flow: Multi-Selection - Nested if...else

```
> if (false){
      console.log('A');
  } else if (true){
      console.log('B');
  } else {
      console.log('C');
  }

  B
```

7. Control Flow: Multi-Selection - Switch

```
> let result = "";
  switch ('option2'){
      case 'option1':
          result += 'A';
          break;
      case 'option2':
          result += 'B';
          break;
      default:
          result += 'C';
      }
  console.log(result);
  B
```

```
> let result = "";
  switch (1){
      case 1:
          result += 'A';
       case 2:
          result += 'B';
      default:
          result += 'C';
  }
  console.log(result);
  ABC
```

7. Control Flow: counter-controlled loop - for

```
> let result = "";
  for (let i=0; i< 10; i++){
      result += 'Na';
  }
  console.log(result, 'Batman!');
  NaNaNaNaNaNaNaNaNaNa Batman!
```

7. Control Flow: counter-controlled loop - enhanced for...in

```
> let result = "", arr = ['a','b','c'];
  for (let i in arr){
      result += i;
  }
  console.log(result);
  012
```

7. Control Flow: counter-controlled loop - enhanced for...of

```
> let result = "", arr = ['a','b','c'];
  for (let i of arr){
      result += i;
  }
  console.log(result);

  abc
```

7. Control Flow: sentinel-controlled loop - while

```
> let result = "";
  while (result.length < 20){
      result += 'Na';
  }
  console.log(result, 'Batman!');
  NaNaNaNaNaNaNaNaNaNaNa Batman!
```

7. Control Flow: sentinel-controlled loop - do...while

```
> let result = "";
  do{
      result = 'A';
  } while (false);
  console.log(result);
  A
```

7. Control Flow: continue & break

```
> let result = "";
  for (let i=0; i< 10; i++){
      if ( i === 5){
          break;
      }
      result += i;
  }
  console.log(result);
  01234
```

7. Control Flow: continue & break

```
> let result = "";
  for (let i=0; i< 10; i++){
      if ( i % 3){
          continue;
      }
      result += i;
  }
  console.log(result);
  0369
```

8. Functions: declarations, expressions, arrow

1. function declarations
2. function expressions
3. arrow functions

8. Functions: function declarations

```
> function foo(){
      console.log("Hello");
  }
<· undefined
> foo()
  Hello
<· undefined
```

```
> function foo(x){
      console.log(x);
  }
<· undefined
> foo("Bye!")
  Bye!
<· undefined
```

```
> function foo(x){
      return x*5;
  }
<· undefined
> x= foo(100)
<· 500
```

8. Functions: function expressions

```
> const foo = function(){
      console.log("Hello");
  }
<· undefined
> foo()
  Hello
<· undefined
```

```
> const foo = function(x){
      console.log(x);
  }
<· undefined
> foo("Hello")
  Hello
<· undefined
```

```
> const foo = function(x){
      return x*5;
  }
<· undefined
> x = foo(10)
<· 50
```

8. Functions: arrow function

```
> const foo = () => console.log("hello")
  foo()
  hello
<· undefined
```

```
> const foo = (x) => console.log(x)
  foo("Bye")
  Bye
<· undefined
```

```
> const foo = (x) => x * 5
  x = foo(10)
<· 50
```

8. Functions: default values versus overloading *(not in JS)*

```
> function foo(x="Hello"){
    console.log(x);
  }
<· undefined
> foo()
  Hello
<· undefined
> foo("Bye")
  Bye
<· undefined
```

```
> function foo(){
    console.log("Hello");
  }

  function foo(x){
    console.log(x);
  }
<· undefined
> foo()
  undefined
<· undefined
```

9. Closures, & Hoisting: Hoisting defined

Hoisting is JavaScript's default behavior of moving all ***declarations*** to the top of the current scope (to the top of the current script or the current function).

JavaScript Initializations are Not Hoisted

- global vars/functions are hoisted to top of global scope.
- vars are hoisted to top of function scope.
- let, const are not hoisted.

9. Closures, & Hoisting: Example of Variable Hoisting

```
> function foo(){
      console.log(z);
      var z = 10;
  }
< undefined
> foo()
  undefined
< undefined
> function foo(){
      z = 10;
      console.log(z);
      var z;
  }
< undefined
> foo()
  10
```

```
> {
      x = 10;
      console.log(x);
      let x;
  }
✖ ▸Uncaught ReferenceError: Cannot access 'x' before initialization
```

- var is hoisted to nearest function, if not defined in a function, its within the global scope.
- let, const are not hoisted

9. Closures, & Hoisting: Example of Function Hoisting

```
> foo()

  function foo(){
      console.log("hello world");
  }

  hello world
```

```
> bar()

  const bar = function(){
    console.log("Goodbye world");
  }

✕ ▶Uncaught ReferenceError: Cannot access 'bar' before initialization
```

9. Closures, & Hoisting: Closures defined

Closures are **nested functions**, possible since functions are objects!

Closures are useful because they let you associate data (the lexical environment) with a function that operates on that data. This has obvious parallels to object-oriented programming, where objects allow you to associate data (the object's properties) with one or more methods.

Consequently, you can use a closure anywhere that you might normally use an object with only a single method.

9. Closures, & Hoisting: Examples of Closures

```
> const adder = function(x) {
    return function(y) {
      return x + y;
    };
  }

  const add5 = adder(5);
  const add10 = adder(10);
<· undefined
> add5(2)
<· 7
> add10(2)
<· 12
```

```
> const adder = x => y => x + y;
  const add5 = adder(5);
  const add10 = adder(10);
<· undefined
> add5(2)
<· 7
> add10(2)
<· 12
```

The End.

Coming Soon... to a Moodle near you!

Quiz	'JavaScript - Basics' Quiz
Homework 4	JavaScript Basics - Coding Problems