FUNCTION EXPRESSIONS

Wholeness

This short lesson introduces several features of JavaScript functions that will be important as we start to use more advanced functional programming techniques in the remainder of the course. *Science of Consciousness: This knowledge will put us in a good position for diving deeply into functional programming. Take the correct angle and let go.*

Lesson Objectives

- Understand when, where, and how to declare and use function declarations, function expressions, and arrow functions
- Understand the concept of first class objects, callback functions, and anonymous functions
- Understand the concept of the runtime execution stack and be able to diagram instances of the runtime stack

Main Point Preview: JS functions are first class objects

Functions are first class objects in JavaScript, which means they can be treated as values, parameters, and return values. Whenever functions are followed by parentheses it means they should be executed.

Functions as values

- Functions are first-class objects in JavaScript
  - stored in a variable, object, or array.
  - passed as an argument to a function.
  - returned from a function

```
function sayHi() {
  alert( "Hello" );
}
const myHi = sayHi;
alert( sayHi ); // shows the function code
function higherOrder() { return sayHi; }
```

- ( ) after function name is an important syntactic construct
  - means that there is a function and it should be executed

Function expressions & Anonymous Functions

- syntax used before is a *Function Declaration*
- another syntax is a *Function Expression*.
  - function keyword can be used to define a function inside an expression

```
// function expression
let sayHi = function(){console.log("Hi");};
sayHi();
```

  - In JavaScript, a function is a value, so we can deal with it as a value.

- Function without a name is called *anonymous* function.

Semicolon rules

- why does Function Expression end with semicolon ; but not Function Declaration

```
function sayHi() {
  // ...
}

let sayHi = function() {
  // ...
};
```

- ; not needed at the end of code blocks and syntax structures that use code blocks
  - like if { ... }, for { }, function f { } etc.

- A Function Expression is used as part the statement: let sayHi = ...;, as a value.
  - It’s not a code block, but rather an assignment statement
  - semicolon ; is recommended at the end of statements.
  - semicolon here is not related to the Function Expression itself, it just terminates the statement

Callback functions

- more examples of passing functions as values and using function expressions
- arguments showOk and showCancel of ask are called callback functions or callbacks
- idea is that we pass a function and expect it to be “called back” later
  - showOk becomes callback for “yes” answer
  - showCancel for “no” answer

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}
function showOk() {
 console.log( "You agreed." );
}
function showCancel() {
  console.log( "You canceled the execution." );
}
ask("Do you agree?", showOk, showCancel);
```

```
//more succinct function expression version (anonymous functions)
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask( "Do you agree?",
  function() {console.log("You agreed."); },
  function() {console.log("You canceled the execution."); }
```

Exercise

- Write a function, myCallback(func, arg). myCallback will call func with the given arg and then log the return value to the console.
  - Test myCallback by calling it with a function that takes a number and returns the cube of the argument. First write it as "cube" a normal named function declaration, then as an anonymous function expression.

Function Expression vs Function Declaration

- *Function declaration:* declared as a separate statement, in the main code flow
- *Function expression*: created inside an expression or inside another syntax construct.
  - E.g., at the right side of the “assignment expression” =:

- when created by JS engine?
  - A Function Expression is created when the execution reaches it and is usable only from that moment.
  - A Function Declaration can be called earlier than it is defined.

- When to choose Function Declaration versus Function Expression?
  - first consider Function Declaration
    - more freedom in how to organize our code, because can call such functions before they are declared.
    - better for readability, Function Declarations are more “eye-catching”
  - if a Function Declaration does not suit us for some reason, then function expression
    - anonymous functions (e.g., DOM event handlers often are not reusable)

Arrow functions

- Can be used in the same way as function expressions
  - More succinct, advantageous for short anonymous callbacks
  - Fix language issue involving inner functions (will discuss with objects)

```
//function expression
let sum = function(a, b) {
  return a + b;
};

//equivalent arrow function
let sum = (a, b) => a + b;

//only one argument, then parentheses around parameters can be omitted
let double = x => 2 * x;

//no arguments, parentheses should be empty (but they should be present):
let sayHi = () => alert("Hello!");

//if body has { } brackets then must use return
let sum = (a, b) => { return a + b; };
```

Arrow function

- New syntax in ES6 to write a function expression in concise way

```
let isEven = (a) => {return a%2===0;}
console.log(isEven(4));

let isOdd = (a) => a%2 !== 0;
console.log(isOdd(7));

let sayHello = () => console.log('HI');
sayHello();
```

```
(arguments) => { return statement }; // general syntax
  argument => { return statement }; // one parameter
      argument => statement; // implicit return
          () => statement; // no parameter
```

- Exercise: rewrite isEven as a function expression

Main Point: JS functions are first class objects

Functions are first class objects in JavaScript, which means they can be treated as values, parameters, and return values. Whenever functions are followed by parentheses it means they should be executed.

Main Point Preview: Runtime stack

Functions can call other functions. Synchronous calls are managed in the runtime execution stack. *Science of consciousness: Programmers do not directly manage the runtime stack but understanding how it works will allow us to spontaneously write correct programs that work with the full support of the underlying laws and rules of the runtime engine.*

The execution context and stack

- The information about the process of execution of a running function is stored in its *execution context*.
- The execution context is an internal data structure that contains details about the execution of a function:
  - current variables the function is using
  - Location in code to resume execution
- One function call has exactly one execution context associated with it.
- When a function makes call to another function, the following happens
  - The current function is paused
  - The execution context associated with it is remembered in a special data structure called *execution context stack*.
  - Called function executes
  - After it ends, the calling function is resumed with prior saved *execution context*.

Function calling another function

```
// Output?

function A(){
    console.log("A is called");
    console.log("Before B is called");
    B();
    console.log("After B is called")
}

function B(){
    console.log("B is called");
    console.log("Before C is called");
    C();
    console.log("After C is called");
}

function C(){
    console.log("C is called");
}
A();
console.log("After A is called");
```

Example: Lets draw a stack

```
function funA(a,n){
    let something;
    something = "something.";
    funB(something, n);
}

function funB(a,b){
    let thing;
    thing = "a thing.";
    console.log("What is on the stack when we're here?");
}

function main(){
    let test;
    let n;
    test = "Hello";
    n = 5;
    funA(n, 10);
}

main();
```

Exercise: Draw the stack

```
function funX(a, b) {
    let c;
    c = 5;
    funY(a * c, "yes");
}

function funY(x, y) {
    let z;
    z = "I can see the sea";
    console.log("What is on the stack here?");
}

function main() {
    let a;
    let b;
    a = "Hello";
    funX(3, a);
    b = "World";
}

main();
```

Software design principles for functions

- avoid globals
- avoid side effects
- a pure function takes arguments and returns a value
  - does not change arguments
  - returns a value
  - does not change any variables or state outside the function
- a function should be a command or a query, not both
  - tendency is for people to reuse functions that return values to get the value
  - if it also has a side effect (updating a database, printing, etc) can be unexpected and produce a bug
- 30 second rule: should take 30 seconds or less to read and understand, else too long
- functions should be self contained and only require user to know signature and return value

Avoid premature optimization

- avoid break and continue
- "premature optimization is the root of all evil (in programming)“
- 3 laws of optimization
  - don’t
  - later
  - only after profile

Main Point: Runtime stack

Functions can call other functions. Synchronous calls are managed in the runtime execution stack. *Science of consciousness: Programmers do not directly manage the runtime stack but understanding how it works will allow us to spontaneously write correct programs that work with the full support of the underlying laws and rules of the runtime engine.*

Review of JavaScript Syntax, Functions, Arrays

Data Types

- All programing languages have data types, some are strict on data types, and others are loose.
- In JavaScript, variable's type determined by value it is currently holding.
- 6 primitive data types: **`string`**, **`number`**, `bigint`, **`boolean`**, **`undefined`**, and `symbol`.
  - **`null`**, which appears primitive, but is a special case of **`Object`**.
- can put any type in a variable.

```
// no error
let message = "hello";
message = 123456;
```

- Programming languages like this are called *"dynamically typed"*
  - *Also called "loosely typed" or "weakly typed"*

null and undefined

- The special values `null` and `undefined` are special types as well.
- In JavaScript, `null` is not a “reference to a non-existing object” or a “null pointer” like in Java
  - special value which represents “nothing”, “empty” or “value unknown”.
- The meaning of `undefined` is “value is not assigned”.
  - If a variable is declared, but not assigned, then its value is `undefined`.
- Programmers assign null; the compiler assigns undefined

```
let name = null;
let age;
console.log(name,age) // null, undefined
```

Type Conversions

- Most of the time, operators and functions automatically convert the values given to them to the right type.
  - For example, `alert` automatically converts any value to a string to show it.
  - Mathematical operations convert values to numbers.

- There are also cases when we need to explicitly convert a value to the expected type.

Comparing different types

- When comparing values of different types, JavaScript converts the values to numbers.

```
alert( '2' > 1 ); // true, string '2' becomes a number 2
alert( '01' == 1 ); // true, string '01' becomes a number 1
```

- A strict equality operator === checks the equality without type conversion.
  - In other words, if a and b are of different types, then a === b immediately returns false without an attempt to convert them.
  - Always use === instead of ==

- Generally, comparison of different types is a mistake or poor design

Truthy & Falsy

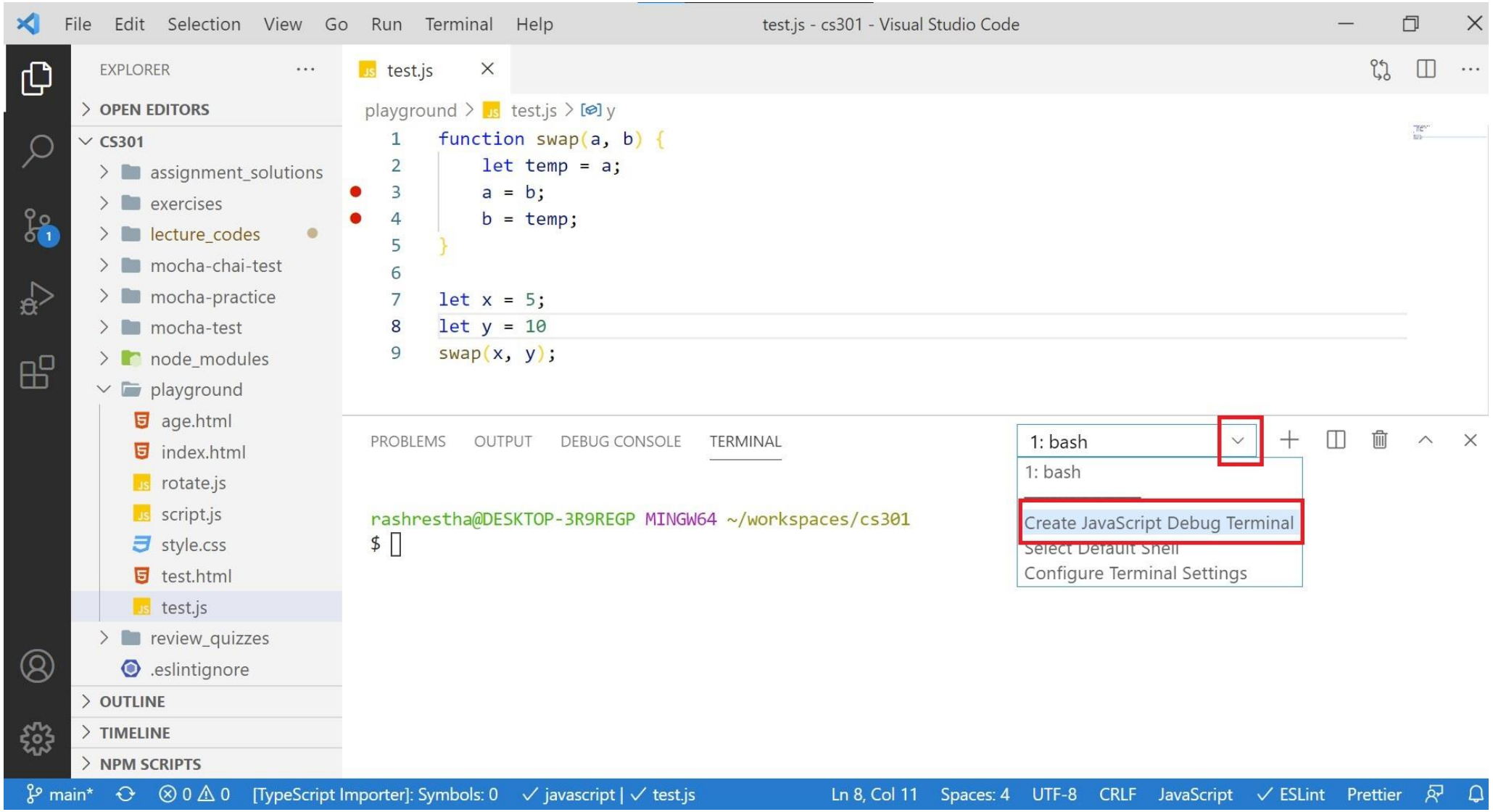
- In JavaScript, any value can be used as a Boolean
  - "**falsy**" values: 0, 0.0, NaN, "", null, and undefined
  - "**truthy**" values: anything else

Returning a value

- A function can return a value to the calling code
- The directive `return` can be any place of the function.
  - When the execution reaches it, the function stops, and the value is returned to the calling code.
  - There may be many occurrences of `return` in a single function.
  - It is also possible to use `return` without a value. That causes the function to exit immediately.
- A function with an empty `return` or without it returns `undefined`

```
function oddEven(num){
  if (!num) return;
  if(num%2==0) return "Even";
  else return "Odd"
}
```

Debugging in Node



JS doc

- common development problem:
  - you have written JavaScript code that is to be used by others and need a nice-looking HTML documentation of its API.
  - standard tool in the JavaScript world is JSDoc.
  - It is modeled after its Java analog, JavaDoc.

- JSDoc takes JavaScript code with /** */ comments (normal block comments that start with an asterisk) and produces HTML documentation for it

- For functions and methods, you should document parameters and return values,
  - and exceptions they may throw:

```
/**
*
 * @param {number} x The number to raise.
 * @param {number} n The power, must be a natural number.
 * @return {number} x raised to the n-th power.
 */
function pow(x, n) {
  ...
}
```

Using an Array

- Array elements are numbered, starting with zero.
- We can get an element by its number in square brackets:

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum
```

- We can replace an element:

```
fruits[2] = 'Pear'; // now ["Apple", "Orange", "Pear"]
```

- Or add a new one to the array:

```
fruits[3] = 'Lemon'; // now ["Apple", "Orange", "Pear", "Lemon"]
```

Looping through an array elements

- One of the oldest ways to cycle array items is the for loop over indexes:

```
let arr = ["Apple", "Orange", "Pear"];

for (let i = 0; i < arr.length; i++) {
  alert( arr[i] );
}
```

- But for arrays there is another form of loop, `for..of`:

```
for (let fruit of fruits) {
  alert( fruit );
}
```

- The for..of doesn't give access to the index of the current element, just its value, but in most cases that's enough.
  - And it's shorter.
  - And avoids bugs that often occur from index errors at the end points
  - Favor for..of as default loop over arrays unless really need index

Array comparison

- Arrays are type Object
- When == or === operators are used on JavaScript objects, their references are compared

- **If array comparison is needed compare them item-by-item in a loop**.
  - Mocha has a very convenient assert.deepStrictEqual
  - [1, 2, 3] === [1, 2, 3] → false

Add/Remove elements To/From the end

- Array in JavaScript has inbuilt methods that allow you to add/remove elements to/from the end of the array.
  - `pop`: extracts the last element of the array and return it.

```
let fruits = ["Apple", "Orange", "Pear"];
console.log( fruits.pop() ); // remove "Pear" and log it
console.log( fruits ); // Apple, Orange
```

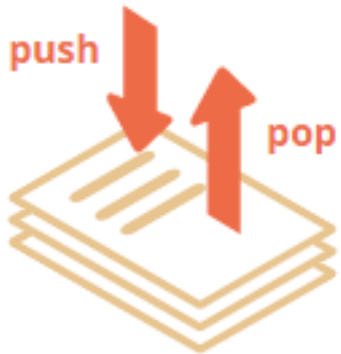
  - `push`: append element to the end of the array.

```
let fruits = ["Apple", "Orange"];
fruits.push("Pear");
console.log( fruits ); // Apple, Orange, Pear
```

  - The call `fruits.push(...)` is equal to `fruits[fruits.length] = ...`

Array as a stack

- There’s another use case for arrays – the data structure named stack.
- It supports two operations:
  - push adds an element to the end.
  - pop takes an element from the end.
- So new elements are added or taken always from the “end”.



- For stacks, the latest pushed item is received first, that’s also called LIFO (Last-In-First-Out) principle

Accessing elements

```
let matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]];

console.log(matrix);

for (let i = 0; i < matrix.length; i++) {
    for (let j = 0; j < matrix[i].length; j++) {
        console.log(matrix[i][j]);
    }
}
```