**FUNCTIONS:**

**ARROW FUNCTIONS:**

\_The arrow function does NOT have the RETURN keyword or the semicolon like this:

ex:

const upperizedNames = ['Farrin', 'Kagure', 'Asser'].map(function(name) {

return name.toUpperCase();

});

Instead:

const upperizedNames = ['Farrin', 'Kagure', 'Asser'].map(

name => name.toUpperCase()

);

Q: Function declaration vs Function expression

A: Function Declaration:

function foo() { return 5; }

Function expression:

var foo = function foo() { return 5; }

Difference: Function declarations load before any code is executed.

Function expressions load only when the interpreter reaches that line of code

* Arrow function is function expression

**WEIRD SYNTAX:**

Arrow function is stored in a variable:

Const greet = name => `Hello ${name}!`;

greet(‘Bryan’);

Returns: Hello Bryan

BUT if there are more than 1 items in the parameter list:

const orderIceCream = (flavor, cone) => console.log(`Here's your ${flavor} ice cream in a ${cone} cone.`);

orderIceCream('chocolate', 'waffle');

Prints: Here's your chocolate ice cream in a waffle cone.

Multiple way to write arrow functions:

Each of these is correct:

setTimeout(() => {

console.log('starting the test');

test.start();

}, 2000);

setTimeout( \_ => {

console.log('starting the test');

test.start();

}, 2000);

* **Either () or \_ is fined. Underscore never gets used so it’s undefined**

const vowels = 'aeiou'.split('');

const bigVowels = vowels.map( (letter) => letter.toUpperCase() );

const vowels = 'aeiou'.split('');

const bigVowels = vowels.map( letter => letter.toUpperCase() );

* If there’s only 1 paramter, then without the parenthesis is fine but not wrong

Q: Concise and Block body syntax with Arrow function

A: 1) Concise body syntax:

+) Has no curly braces surrounding the function body

+) Automatically returns the expression

+) NO return statement, no curly braces

ex:

const upperizedNames = ['Farrin', 'Kagure', 'Asser'].map(

name => name.toUpperCase()

);

2)Block body syntax

+) Uses curly braces to wrap the function body

+) And when used with the curly braces, it needs to be used to actually return something from the function

+) Without either the curly braces or the “return” statement, it wouldn’t work

+) Used when there’s more than 1 line of code in the arrow function’s body

Ex:

const upperizedNames = ['Farrin', 'Kagure', 'Asser'].map( name => {

name = name.toUpperCase();

return `${name} has ${name.length} characters in their name`;

});

ex 2:

const squares = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].map((square) => {

square = square \* square

return square;

});

console.log(...squares);

Q: What do I notice about that?

A: 1) {} always go with “return”. It doesn’t stop the function. Also .map already has loop built inside it

2) When you print out …squares, it’s like you print out everything (spread operator)

Q: Can I use arrow function instead of normal function from now on?

A: 1) When there’s the keyword “this”, can’t

2) Arrow functions are only expression, not declaration

**“THIS” KEYWORD:**

Q: Method vs Function?

A: 1) Method is a function associated with an object ( a constructor function or class)

2) Method has access to the data present in its Construction function

ex: console.log(‘print something’)

log() is a method

console is an object

ex2: array.map() is a method

Q: What’s a constructor?  
A: A function that creates an object

Ex: function Tree(name) {

this.name = name;

}

var theTree = new Tree('Banana Tree');

console.log(`theTree.constructor is ${theTree.constructor}`);

* Will print:

theTree.constructor is function Tree(name) {

this.name = name;

}

Q: Values of ‘this’ in different cases?

A:

1. A new object:

If a function is called with “new”, then the value of “this” is a new object

Ex:

Const mySundae = new Sundae(‘Chocolate’, [‘Sprinkles’, ‘Hot Fudge’]);

1. A specified object:

If the function is invoked with “call/apply”, it’ll refer to the parameter it’s set to:

Ex:

Const result = obj1.printName.call(obj2);

“this” is set to obj2

1. A context object:

If a function is a method of an object, the value of “this” will refer to the object:

Ex: const redTrain = new Train('red');

redTrain.increaseSpeed(25);

* “this” will be the redTrain object

1. If a function is called without context:

teleport();

“this” is bound to the global object, or if in strict mode, it’s “undefined”

* Therefore, we can say that the value of “this” is set based on how the function is called

**“this” and ARROW FUNCTIONS:**

Q: .call(), .apply() vs .bind()?

A:

1. .**bind() vs .call()/ .apply():**

+)bind() attaches “this” into function but only if you want that function to be later called with a certain context. .call() & .apply() is when you want to invoke the function immediately, and modify the context

ex1:

function MyObject(element) {

this.elm = element;

element.addEventListener('click', **this.onClick.bind(this**), false);

};

ex2:

var person = {

name: "James Smith",

hello: function(thing) {

console.log(this.name + " says hello " + thing);

}

}

person.hello("world"); // output: "James Smith says hello world"

var helloFunc = person.hello.bind({ name: "Jim Smith" });

helloFunc("world"); // output: Jim Smith says hello world"

or like this:

...

var helloFunc = person.hello.bind({ name: "Jim Smith" }, "world");

helloFunc(); // output: Jim Smith says hello world"

1. **Apply() vs .Call():**

+) Similar, except .apply() takes arguments as an array; .call() requires you to set the arguments explicitly.

Ex:

function theFunction(name, profession) {

console.log("My name is " + name + " and I am a " + profession + ".");

}

theFunction("John", "fireman");

theFunction.apply(undefined, ["Susan", "school teacher"]);

theFunction.call(null, "Claude", "mathematician");

theFunction.call(undefined, ...["Matthew", "physicist"]); // used with the spread operator

// Output:

// My name is John and I am a fireman.

// My name is Susan and I am a school teacher.

// My name is Claude and I am a mathematician.

// My name is Matthew and I am a physicist.

* First arguments as “undefined/ null” as first argument to indicate that “this” is the global object

Ex2: clearer example:

function product(name, value)

{

this.name = name;

if (value >= 1000)

this.value = 999;

else

this.value = value;

}

function prod\_dept(name, value, dept)

{

this.dept = dept;

product.apply(this, arguments);

}

prod\_dept.prototype = new product();

// since 5 is less than 1000 value is set

var cheese = new prod\_dept("feta", 5, "food");

// since 5000 is above 1000, value will be 999

var car = new prod\_dept("honda", 5000, "auto");

Q: If you set “this” inside an arrow function, what value does “this” take?

A: “this” will be set to the function’s surrounding context

Q: How is it different from the regular function?

A: Because in regular function, the value of “this” is set to how the function is called

Q: example 1:

A:

// constructor

function IceCream() {

this.scoops = 0;

}

// adds scoop to ice cream

IceCream.prototype.addScoop = function() {

setTimeout(function() {

this.scoops++;

console.log('scoop added!');

}, 500);

};

const dessert = new IceCream();

dessert.addScoop();

console.log(dessert.scoops);

* Print: 0

Q: Why?

A: Because “this” is passed into setTimeOut() without “new”, “call()”, “apply” and without a context object

* So “this” value is the global object
* So “scoops” is undefined

Q: How to fix that?

A: Put closure in

// constructor

function IceCream() {

this.scoops = 0;

}

// adds scoop to ice cream

IceCream.prototype.addScoop = function() {

const cone = this; // sets `this` to the `cone` variable

setTimeout(function() {

cone.scoops++; // references the `cone` variable

console.log('scoop added!');

}, 0.5);

};

const dessert = new IceCream();

dessert.addScoop();

console.log(dessert.scoops);

Prints:

1

Set const cone to “this” because now it uses “this” outside the setTimeOut function and instead sets it equal to value of “cone”

* That’s the same thing with arrow function. Setting “this” from the surrounding context

Ex:

// constructor

function IceCream() {

this.scoops = 0;

}

// adds scoop to ice cream

IceCream.prototype.addScoop = function() {

setTimeout(() => { // an arrow function is passed to setTimeout

this.scoops++;

console.log('scoop added!');

}, 0.5);

};

const dessert = new IceCream();

dessert.addScoop();

console.log(dessert.scoops);

Prints:

1

Q: But this doesn’t work, why?

// constructor

function IceCream() {

this.scoops = 0;

}

// adds scoop to ice cream

IceCream.prototype.addScoop = () => { // addScoop is now an arrow function

setTimeout(() => {

this.scoops++;

console.log('scoop added!');

}, 0.5);

};

const dessert = new IceCream();

dessert.addScoop();

A: Because here, value of “this” is set to be outside of the outer most arrow function, which is the global variable

**DEFAULT FUNCTION PARAMETERS:**

Q: Why is it important?

A: function greet(name, greeting) {

name = (typeof name !== 'undefined') ? name : 'Student';

greeting = (typeof greeting !== 'undefined') ? greeting : 'Welcome';

return `${greeting} ${name}!`;

}

greet(); // Welcome Student!

greet('James'); // Welcome James!

greet('Richard', 'Howdy'); // Howdy Richard!

* That is a mess

So to establish default function parameters:

function greet(name = 'Student', greeting = 'Welcome') {

return `${greeting} ${name}!`;

}

greet(); // Welcome Student!

greet('James'); // Welcome James!

greet('Richard', 'Howdy'); // Howdy Richard!

Returns:

Welcome Student!

Welcome James!

Howdy Richard!

* Add an equal sign and then whatever you want the parameter to default to if an argument is not provided

**DEFAULT AND RESTRUCTURING:**

**DESTRUCTURING WITH ARRAY:**

Q: what’s join()?

A: To join elements of an array into a string

Ex: ‘white’, ‘gray’, ‘pink’

NOT to join strings together

Q: Example 1

A:

function createGrid([width = 5, height = 5]) {

return `Generates a ${width} x ${height} grid`;

}

createGrid([]); // Generates a 5 x 5 grid

createGrid([2]); // Generates a 2 x 5 grid

createGrid([2, 3]); // Generates a 2 x 3 grid

createGrid([undefined, 3]); // Generates a 5 x 3 grid

BUT:

createGrid(); // throws an error

Q: How to fix that ^?

A: function createGrid([width = 5, height = 5] = **[])** {

return `Generating a grid of ${width} by ${height}`;

}

**DESTRUCTURING WITH OBJECTS:**

Q: Example 2?

A: function createSundae({scoops = 1, toppings = ['Hot Fudge']} = **{}**) {

const scoopText = scoops === 1 ? 'scoop' : 'scoops';

return `Your sundae has ${scoops} ${scoopText} with ${toppings.join(' and ')} toppings.`;

}

createSundae(); // Your sundae has 1 scoop with Hot Fudge toppings.

**DIFFERENCES BETWEEN DESTRUCTURING WITH OBJECTS AND ARRAY:**

Q: With objects?

A:

function createSundae({scoops = 1, toppings = ['Hot Fudge']} = {}) { … }

* If I want to use the default value for scoops but change the toppings, then it’s OK to just pass the toppings:

createSundae({toppings: ['Hot Fudge', 'Sprinkles', 'Caramel']});

Q: With arrays?

A: To do the same thing, you must do:

createSundae([undefined, toppings: ['Hot Fudge', 'Sprinkles', 'Caramel']]);

Why undefined? So that we can “skip” over the first argument and accept the default to get to the second argument

* So just use object with object destructuring, and not array destructuring

Q: What if I want to use the arrow function?

A:

const buildHouse = (floors = 1, color= 'red', walls='brick') => { return `Your house has ${floors} floor(s) with ${color} ${walls} walls` }

OR:

const buildHouse = (floors = 1, color= 'red', walls='brick') => (`Your house has ${floors} floor(s) with ${color} ${walls} walls`)

Either {return} or none

Also have to declare it equal to var/let/const, or annonymous

Ex: variable\_ = () => { return ...} or \_variable\_ = () => ()

**CLASS IN JAVASCRIPT:**

Q: What is Class in other languages?

A: It’s used to create objects

Q: Class in Javascript?

A: Not a class at all but an illusion

+) Used functions to create object

+) Link objects together by prototypal inheritance

Q: prototypal inheritance vs classical inheritance?

A:

1. Prototypal inheritance:

+) How JS implements inheritance

+) Children inherit from parents

Ex: If parent’s array.prototype.slice = null, then children won’t have the slice method

+) You have an object and you use that objects to model your other children objects based on it

1. Classical inheritance:

+) Use class to inherit objects

+) Is a blue print, and you use that blue print to create other objects

Ex: Car doesn’t’ exist, but you create it

\_The constructor is a function that creates object

**ES6 & ES5 “CLASS” DIFFERENCE:**

\_They’re the same but look different (because one tries to look more classical):

\_This is a syntactic sugar to entice people who use to object oriented languages to use JS

In ES5:

**function** **Plane**(numEngines) {

**this**.numEngines = numEngines;

**this**.enginesActive = false;

}

*// methods "inherited" by all instances*

Plane.prototype.startEngines = **function** () {

console.log('starting engines...');

**this**.enginesActive = true;

};

**const** richardsPlane = **new** Plane(1);

richardsPlane.startEngines();

**const** jamesPlane = **new** Plane(4);

jamesPlane.startEngines();

* What it looks like: You create a parent object first, then you create children objects based on that parent object. The constructor function here is called with the “new” keyword (starts with a capital letter). “Inherited” methods are placed on the constructor function’s prototype object

In ES6:

**class** **Plane** {

constructor(numEngines) {

**this**.numEngines = numEngines;

**this**.enginesActive = false;

}

startEngines() {

console.log('starting engines…');

**this**.enginesActive = true;

}

}

Here: You don’t have a parent object, only a constructor and use that to create all other objects. You create the object and all prototype methods in 1 go