**JS 129 – OOP**

* JS functions are first class objects. Gives us the ability to:
  + Use callbacks
  + Function expressions
  + Initially invoked function expressions
* Concept execution context
* Dealing with content loss
* How to bind functions to specific context objects
* Cornerstone OOP concepts:
  + Prototypal inheritance
  + Encapsulation
  + Polymorphism
* Major topics:
  + What are objects
  + How inheritance works in JavaScript
  + How JavaScript deal with multiple inheritance
  + Object properties and methods
  + Working with object oriented code
  + Design considerations when working with OOP
  + Function expressions
  + Higher order functions
  + The global object
  + Execution context and dealing with context loss
  + Factory functions
  + Object prototypes
  + Constructors
  + Classes
  + Projects
* Encapsulation – bundling state (data) and behavior (operations) to form an object
  + In most OOP languages, encapsulation also refers to restricting access to the state and certain behaviors. Objects expose a **public interface** for interacting with other objects and keep their implementation details hidden. So other objects can’t change the data of an object without going through the proper interface. Unfortunately, JS doesn’t support access restrictions

Questions

1. What is OOP
   1. OOP is the style of organizing operations and data in objects.
2. Describe some advantages and disadvantages
   1. Advantages of OOP:
      1. makes organization clear, flexible, and easier to change
      2. Allows programmers to write programs that reduce dependencies in a program, makes maintenance easier
      3. Can make use of restrictions to prevent mutation
   2. Disadvantages:
      1. Complicated, often much larger than the equivalent procedural program
      2. May lead to less efficient code, OO programs may require more memory, disk space, and computing power
3. What does encapsulation refer to in JS?
   1. It refers to the idea of ***bundling*** state (data) and behavior (operations) to form an object
4. In JS how does encapsulation differ?
   1. Other languages allow encapsulation to hide the details of an object from code that uses the object. An object should only expose the methods and properties that other objects need. However JS does not directly provide the means to limit the exposure of methods and properties

Functions as object’s property values

* Object properties that are function values are methods
* We’ve seen before that JS functions are first-class values, which means we can treat them as we would any JS value.
* Advantage is if we want to operate on a car, we don’t need to look for both the function and the data. We can see at a glance what you can do with a car merely by looking at the object
* For the car example, note that JS won’t stop you from changing the fuelLevel property directly instead of calling the refuel method – this is the encapsulation limitation

Compact Method Syntax

* startEngine: function () {} 🡪 startEngine() {}

**This** keyword

* we refer to the object from the inside the methods by directly using the variable name, racecar
* If we change the variable name or pass the object to a function that refers to its arguments by a different name it would throw a referenceError

Collaborators

* Objects that are used to store state within another object are called collaborator objects, collaborators
* For example, the VSCode file intro.JS: pete object has a collaborator object stored in its pet property
* Collaborator objects let you chop up and modularize the problem domain into cohesive pieces
* They do not have to be custom objects, they can be built-in objects like arrays and dates

Functions as Object Factories – Automate the process of creating objects

Class Inheritance

* Child types inherit common properties and methods from a parent

Review

* Encapsulation is the idea of bundling data and operations related to that data in a unit called an object. In other languages encapsulation also refers to the idea of restricting access to state and some behavior, but JS objects don’t support that
* Simplest way to create an object is to use the object literal syntax: pair of opening and closing curly braces. Adding methods to an object is as simple as adding a function to the value of a property
* You can access the properties and methods of an object from within a method using the `this` keyword
* Objects collaborate with other objects and primitive values by using them as part of their state. We say two items have a collaborator relationship if one of them is part of the state of the other
* One way to automate the creation of objects is to use the factory function pattern. A factory function returns an object with a particular set of methods and properties. The methods remain the same across the objects, while the property values can be customized by providing them as arguments
* One object factory can reuse another object factory by mixing the object returned by another factory function into itself using `Object.assign`
* Interface of an object are the state and behaviors exposed by the object for other objects to use
* OOP is a programming paradigm in which we think about a problem in terms of objects
  + It’s a style of programming that uses objects that interact with each other
* Benefits of OOP:
  + Large complex programs can be difficult to maintain due to dependencies throughout the program. OOP lets programmers write programs in a manner that reduces dependencies and makes maintenance easier
  + Complex coding problems are often difficult to break down and solve clearly and systematically. Using OOP to model objects and using real-world nouns to represent objects lets programmers think at a higher level of abstraction. That in turn helps them break down and solve problems
* Encapsulation
  + Refers to how we can bundle state (data) and behavior (operations relevant to that data) into a single entity (an object)
* Object interface:
  + Refers to the state and behaviors exposed by the object for other objects to use
* Collaborator objects:
  + A primitive value that is part of the state of another object
  + An object that is part of the state of another object
* Collaborator objects:
  + Represent the connections between various actors in your program
  + Are at the core of OOP
* Object factory benefits:
  + Let you avoid most of the tedium and errors that result from copying and pasting to create multiple objects of the same type
  + Help reduce code duplication
* Classical approach to OOP:
  + Organize and associate the verbs and nouns
  + Write a textual description of the problem or exercise
  + Extract all ***significant*** nouns and verbs from the description
* Facts about OOP approaches:
  + Don’t worry about finding the optimal OOP design for a problem
  + Choosing an approach for an OO problem always comes down to making tradeoffs
  + There is no single correct approaches for any given OO problem

**LESSON 2**

2 major topics: Execution Content and Object Prototypes

Objects are one of the eight fundamental types in JavaScript:

* String
* Number
* Boolean
* Null
* Undefined
* Object
* BigInt & Symbol (don’t need to know)

Objects are a collection of properties where each property has a key and value. While values can be any of the JS types, property keys are always strings. If you define a property with a non-key string, it will first be converted to a string

* + Let myObject = {}
  + myObject[false] = 0
  + myObject[7] = 1
  + myObject[[1,2,3]] = 0
  + Object.keys(myObject) 🡪 [“false”, “7”, “1,2,3”]

Property Access

* When dealing with objects we are doing one of two things: setting a property or accessing a property. We do both through the property key by using the bracket notation or the dot notation
* Brackets can take any UTF-compatible string as the key, whereas dot notation requires valid variable names
  + myObject.a-key //SyntaxError (a-key is not valid variable name)

Property Existence

* Trying to access a non-existent property on an object returns undefined
* Accessing a property explicity set to undefined returns the same value, undefined
* Two ways to distinguish:
  + in
    - ‘false” in myObject // true
    - “true in myObject // false
  + .hasOwnProperty()
    - myObject.hasOwnProperty(‘7’) // true
    - myObject.hasOwnProperty(‘8’) // false
* 2 other solutions:
  + Object.keys(myObject)
  + Object.getOwnPropertyNames(myObject)

Object Prototypes

Object factory serves two purposes:

* It returns objects that represent data of a specific type
* It reuses code

Prototypes

* Objects can inherit properties from other objects
* JavaScript objects use **prototypal** **inheritance.** The object that you inherit properties and methods from is called the prototype. The function Object.create creates a new object that inherits properties from an existing object
  + This simply gives the new object access to the properties of the prototype
  + It doesn’t have any of its own
* JS objects use internal [[prototype]] property to keep track. So when you create an object with Object.create, the new object’s [[prototype]] property gets assigned to the prototype object
* [[prototype]] is an internal property, which means you cannot access it directly in your code. However you can access and replace its value with Object functions.
  + Object.getPrototypeOf()
  + Ex. obj = Object.getPrototypeOf(obj)
    - allows you to go through the chain of prototypes!
* [[prototype]] sets reference to their prototype objects. So if the prototype changes, these changes are observable in inheriting objects as well

The Default Prototype

* All JS objects have access to .hasOwnProperty() method
* We don’t have to add our own. JS obtains the method from the object’s prototype
* All JS objects inherit from a prototype
  + Object.getPrototypeOf(a) // returns {}

Iterating Over Objects

* For/in will iterate over prototype’s keys as well
  + Use .hasOwnProperty() to skip inherited properties
* Object.keys() returns an object’s own properties, you do not need to use .hasOwnProperty()
* For/in and object.keys deal with enumerable properties
  + Some properties, methods and the length property of an array, are not enumerable

Prototype Chain

* Object.setPrototypeOf(c, b);
* Object.setPrototypeOf(b, a);
  + c inherits from b, which inherits from a
  + b is the prototype of c, a is the protype of b
  + b and a are part of the prototype chain of c
  + the prototype chain also includes the default prototype
  + which is the prototype of object a
  + since the prototype of Object.prototype is null the complete prototype
  + chain looks like this:
  + c --> b --> a --> Object.prototype --> null

Prototype Look-Up in the Prototype Chain

* when you access a property on an object, JS first looks for an “own” property with that name on the object. If the object does not define the specified property, JS looks for it in the object’s prototype. If not there, it’ll continue until it finds the property or reaches Object.prototype. If Object.property also doesn’t define the property, the property access evaluates to undefined
* Basically if there is a prototype property with the same name, the object that’s closer to the calling object takes precedence

Assigning Properties with Chains

* When assigning a property on a JS object, it always treats the property as an “own” property. This means it will not mutate the value of a prototype’s property

Methods from Object.prototype (aka methods on Object.prototype)

* Object.prototype.toString()
* Object.prototype.isPrototypeOf(obj)
* Object.prototype.hasOwnProperty(prop)

Objects Without Prototypes

* You can create objects that don’t have Object.prototype in the prototype chain
  + Let a = Object.create(null)
  + Object.getPrototypeOf(a) // returns null

Set the prototype

* Two ways:
  + Let obj = Object.create(proto)
  + Object.setPrototypeOf(obj, proto)

Function Expressions

* Function declarations performs **hoisting** which means the JS engine moves function declarations to the top of the scope in which they’re defined
* Function definitions that are the first thing on the line are **function declarations.** Whereas **function expressions** are function definitions that are part of an expression
  + (function foo() {}) 🡪 function expression
* Normally we assign function expressions to a variable or object property
  + Ex. let prompt = function () {}
  + Function expressions can be defined without being given a name
  + Above example we have assigned an unnamed function to the prompt variable
  + These are referred to as **anonymous functions**
  + This is commonplace in the callback functions for array methods i.e. forEach & map
* Main advantage of naming function expressions is that when the function throws an error, the stack trace uses that name to help you determine where the error occurred

Arrow Functions

* They are always function expressions
* They are always anonymous
* They’re either: immediately invoked, assigned to variables or properties, or passed around as arguments

First Class Functions

* **First-class functions** means functions in JS are values that we can assign to variables and properties, pass them to other functions, or return them from another function
* **MAIN ADVANTAGE: You should not invoke functions when you want to use them as values!!**
  + Example1:
    - function say(words) {console.log(words)}
    - let speak = say;
    - speak(‘howdy’) // logs “howdy”
  + Example2
    - function logNum(num) {console.log(“Number: “ + num);}
    - [1,2,3].forEach(logNum)

Type of a Function Value

* Since all functions are first-class values in JS and all values in JS have a type, functions also have a type
* Example:
  + Let myFunc = function () {}
  + typeOf myFunc // returns ‘function’
* functions are objects. They’re a compound type that has its own properties and methods

Summary

* Remember functions in JS are first-class values. You can use them any place you can use an expression. To use a function as an expression, write its name without the parenthesis of invocation. All functions have a type of `function`, which is a kind of object with properties and methods

Higher Order Functions

* Because JS treats functions as values, means we can have a special kind of function in our programs: a higher-order function. Higher order function has at least one of the following properties:
  + Takes a function as an argument
  + Returns a function

Functions That Accept Functions as Arguments

* Array methods! Callback functions make them higher order functions

Functions that Return a Function

* Aka a function factory

The Global Object

* JS creates a global object when it starts running. It serves as the **implicit execution context**
* In Node.js it’s called `global` and in a browser it’s `window`
* Global object is available everywhere.
* Global values like infinity and NaN & global functions like isNaN and parseInt are properties of the global object
  + Don’t use .isNaN in your code 🡪 instead use Number.isNaN

Global Object and Undeclared Variables

* Whenever you assign a variable to a value without using let, const, or var, it gets added to the global object as a property
  + Foo = ‘bar’
  + global.foo // returns ‘bar’
  + window.foo // returns ‘bar’ (in a browser)
* You can even access these variables without using the global object as the caller

Implicit and Explicit Execution Context

* Execution context – earlier we said `this` refers to the object that contains the method. The execution context refers to the environment where a function executes. In JS it refers to the current value of `this`. When we talk about the execution context of a function or method call, we’re talking about the value of `this` when that code executes. This context depends on how the function or method was invoked, not on where the function was defined. Two invocations of the same function can have very different contexts depending on ***how*** you make those calls!!!
* Two basic ways to set context when calling a function or method:
  + **Explicit**: The execution context that you set explicitly
  + **Implicit**: The execution context that JavaScript sets implicitly when your code doesn’t provide an explicit context
* Setting the context, aka **binding** `this` aka **setting the binding.** 
  + A binding is something that ties two things together. Here it means a call binds `this` to a specific object when the function or method is called

Function Execution Context – Implicit

* Every JS function call has an execution context. This means the `this` keyword is available to every function in your JS program
  + Every time you call that function, JS binds some object to `this`
  + Function foo() { console.log(‘this refers to’: + this)}
  + Foo() // prints: this refers to: [object global]
* So with a normal function call, JS binds `this` to the global object
* If you use `this` to access or modify properties, you will access or modify properties on the global object

Strict Mode & Implicit Context

* “use strict” – binds `this` to undefined

Method Execution Context – Explicit

* When you call a method that belongs to an object, the execution context inside that method call is the object used to call the method. When you call a method on a specific object, it’s explicit
* Example:
  + let foo = { bar: function() { console.log(this)}}
  + foo.bar() // { bar: [Function: bar]}
  + the method call on the object shows the execution context is the object used to call the method
* Be careful!
  + let baz = foo.bar;
  + baz() // Object [global] {…}
  + this assigns the foo.bar method to the baz variable. So they now refer to the same function object. Since we’re calling baz as a standalone function, its execution context is the *global object* ***and not*** the foo object
* When you invoke a function with () – global
* When you invoke a method – object that you used to call the method
* Although there are ways to subvert this by providing explicit context:
  + call
  + apply

Explicit Execution Context with `call`

* Remember all JS are objects. One method that all JS functions have is the `call` method
* See /Users/caseyrudick/LaunchSchool/JS120/lesson2.js
* Using the call method will not mutate the object
* You can also use the `call` method on methods, not just functions. See VS notes
* If you want to pass arguments to a function, the call method allows with comma-separated values

Explicit Execution Context with Apply

* The `apply` method works in the same way as `call`
* Only difference is apply uses an array to pass any arguments to the function
  + someObject.someMethod.apply(context, [arg1, arg2, arg3])
* Although with modern JS, (ES6 and higher) apply isn’t needed since you can use call in conjunction with the spread operator to do the same thing
  + Let args = [arg1, arg2, arg3]
  + someObject.someMethod.call(context, ..args)

Summary

* All JS functions and method execute within an execution context. Sometimes called `this` binding.
* How `this` gets bound depends entirely on how the function is invoked
* You must examine the invocation itself, cannot just look at the the how and where
* Regular function calls use the global object as their execution context
* Method calls use the calling object as their context
* You can override this by setting the execution context with either call or apply
* Remember just because a method is inside an object, this scope doesn’t mean the context at invocation is set to the object!!!

Hard Binding Functions with Contexts

* **Bind method -** `bind` method works differently than call and apply
* Bind returns a new function that is permanently bound to the object passed as bind’s first argument
* See /Users/caseyrudick/LaunchSchool/JS120/lesson2.js
* You cannot change the execution context after using the bind method
* Remember the bind method returns a new function. The original function is not changed!

Dealing with Context Loss I

* Here we will study what happens when a method is copied out of an object and used elsewhere

Method Copied from Object

let john = {

firstName: 'John',

lastName: 'Doe',

greetings() {

console.log('hello, ' + this.firstName + ' ' + this.lastName);

},

};

john.greetings(); // context is john

let foo = john.greetings; // Strips context

foo(); // context is now the global object

* The stripping can be fixed by foo.call(john) to restore the obj context
* Versus
* Let foo = john.greetings()
* Console.log(foo) // returns the method
* This can be

Passing Context as Argument

* See /Users/caseyrudick/LaunchSchool/JS120/notes.js
* Passing a context to a function or method can be solved:
  + Passing the context as an argument
    - function passContext(func, context){func.call(context)}
  + Calling the function with the method bound to the context
    - passContext(obj.func.bind(context))

Dealing with Context Loss II

* Nested functions also suffer from context loss
* See /Users/caseyrudick/LaunchSchool/JS120/notes.js
* Remember not to rely on arrow functions as methods to prevent context loss!
* They turn to the global object rather than surrounding object
* THIS IS AN EXCEPTION TO THE RULE OF CONTEXT IS DETERMINED BY HOW FUNCTIONS/METHODS ARE INVOKED! SO DON’T USE ARROW FUNCTIONS AS METHODS FOR CONTEXT PRESERVATION

Dealing with Context Loss III

* Function as argument losing surrounding context
* Passing a function as an argument to another function strips it of its execution context
* That means the function argument gets invoked with the context set to the global object
* This problem is identical to the problem with copying a method from an object and using it as a bare function. The following 2 code snippets do the same thing
  + array.forEach(obj.logData);
  + let logData = obj.logData;
  + array.forEach(logData);