**SECTION 14**

* Object-Oriented Programming
* Programming paradigm based on the concept of objects.
* It is style of code, 'how' we write and organize codes.
* We use objects to model (describe) real-world (ex. user or todo list items) or abstract (HTML component or data structure) features.
* Object may contain data (properties) and code (methods). By using objects, we pack data and the corresponding behavior into one block.
* In OOP, objects are self-contained pieces/blocks of codes
* Objects are building blocks of applications, and interact with one another
* Interactions happen through a public interface (API): methods that the code outside of the object can access and use to communicate with the object
* OOP was developed with the goal of organizing code, to make it more flexible and easier to maintain
* We need a way to create objects
  + Traditional: classes and objects were used
    - Class is like a blue print for creating objects
    - Does not contain real world data
    - Objects created from a class -> instances of the class
* Principles of OOP (GENERAL, not just javascript)
  + Abstraction: ignoring or hiding details that don’t matter, allowing us to get an overview of perspective of the thing we are implementing.
  + Encapsulation: keeps properties and methods private inside the class, so they are not accessible outside the class. Some methods can be exposed as a public interface (API)
    - Prevents external code from accidentally manipulating internal properties/state
  + Inheritance: when we have more than 1 class that are closely related to each other, we can simply have one class inherit another
    - Child class extends the parent class
    - Making all properties and methods of a certain class available to child classes, forming hierarchical relationship between classes.
  + Polymorphism: a child class can overwrite a method it inherited from a parent class.
    - By simply writing the same code again, differently inside
* OOP in javascript
  + Prototypes (instead of classes)
    - Contains methods
    - Objects are linked to a prototype object -> can access methods (behaviors)
      * Prototypical inheritance (delegation): the prototype contains methods that are accessible to all objects linked to that prototype
    - Array methods can be used thanks to prototypes
      * Ex. Array.prototype.map()
      * Array.prototype is the prototype of all array objects we create in javascript.
        + All arrays have access to the map method.
  + CLASS vs PROTOTYPE
    - Classes: methods are copied from class to object
    - Prototypes: behavior is delegated to the linked prototype object
* Prototypes
  + Each and every function in javascript has a property called prototype (including the constructor function)
    - All the object created by the constructor function will get access to all the properties and methods we defined on the constructor’s prototype properties.
* How to create prototypes in javascript, how to link objects to prototypes?
  1. Constructor functions
     + Technique to create objects from a function
     + It is how built-in objects like arrays, maps, or sets are implemented
     + Constructor function always starts with a capital letter
     + Arrow function can’t be used here (as we need ‘this’ keyword for the constructor to work)
     + THE ONLY DIFFERENCE between a regular function, and a constructor function:
       - We call a constructor using ‘new’ keyword
     + We created an empty object, which is then linked to prototype.
       - The constructor automatically returns whatever that is added within the constructor. (properties, methods)
       - We don’t define a method inside a constructor
         * Instead, we do something like:
         * ConstructorName.prototype.functionName = function () {};
         * This is possible, because all object has a property called “\_\_proto\_\_”
     + This is essentially a pattern (developed for programmers’ usage), instead of a feature in javascript
     + Diagram

       Description automatically generated
     + Arr.prototype === customary.\_\_proto\_\_
       - Meaning, ‘.prototype’ property of top level object(constructor) is the same one that exists and is usable as prototype for all its instances
       - \_\_proto\_\_ it self is an object. Any instance/objects has access to it.
  2. ES6 classes
     + Modern alternative to constructor function syntax
     + ES6 classes work exactly like the constructor functions
     + The classes do not behave like classes in classical OOP!
     + just nicer syntax to make javascript OOP easier for the beginners, but actually implemented using constructor functions (they use prototype inheritance).
     + It is basically a function, works the same way.
       - We have both definition and declaration for example
     + New object creation works the same way as we use the ‘new’ operator.
     + Rules
       - Classes are NOT hoisted
         * Unlike with functions, we cannot use declaration before they are defined (?)
       - Classes are treated as function too (can be passed in as parameters, return values)
       - Classes are executed in strict mode
  3. Object.create()
     + Easiest, straight-forward

const steven = **Object**.**create**(PersonProto);

*// we pass into the function what we want to be a prototype of the new object*

* + - To define the properties, we make a function like a constructor, where syntax almost looks identical. However it is different from a constructor as it just a normal property function that we can call over and over even outside of the function.
* Inheritance between classes/prototypes (classes don’t exist in js, but we use the term to make it easier)

CLOSURES – we don’t create closures manually, they are explicit

*// execution context, call stack, scope chain*

*// call stack: order in which functions were called*

*// scope chain: order in which functions are written in the code*

const **secureBooking** = function () {

  let passengerCount = 0;

  return function () {

    passengerCount++;

    console.**log**(`${passengerCount} passengers`);

  };

};

const **booker** = **secureBooking**();

*// this booker is a variable that exists in the global environment*

**booker**();

**booker**();

**booker**();

* we can see that the passengerCount increments
* !! closures make the functins remember all the variables the function used in the history
* secure Booking leaves the call stack after its execution
* and then we run booker function in the global scope
* new empty execution context for booker() is then created on top of the call stack
* in scope chain, booker() scope then becomes the child of the global scope, just like secureBooking() scope
* the empty booker function needs access to passsenger count!

/\* ANY function always has access to the variable environment of the execution context

where the function was created\*/

* closure: Variable environment attached to the function, exactly as it was at the time and place the function was created
* booker function was born in the execution context of secureBooking
* therefore booker function has access to the passengerCount