JavaScript Class – July 1, 2024

1. Object-oriented Programming
   1. Uses types of objects as the unit of program organization
   2. Provides a way to think about a program’s structure and enforce discipline that prevents everything from becoming entangled
   3. Abstraction – electric mixer
      1. Abstract data type, or object class, may contain arbitrarily complicated code but exposes a limited set of methods and properties that people working with it are supposed to see
         1. Allows large programs to be built out of a number of object types, limiting the degree to which these different parts are entangled by requiring them to only interact with each other in specific ways
         2. Allows an object to be repaired or rewritten without impacting the rest of the program
         3. Allows object classes to be reused in multiple different programs without recreating functionality from scratch
      2. Each abstract data type has an interface
         1. Collection of operations that external code can perform on it
      3. Any details beyond that are encapsulate d
         1. Treated as internal and of no concern to the rest of the program
   4. this()
      1. When a function is called as a method (looked up as a property and immediately called), ‘this’ is a binding that points at the specific object upon which it was called

function speak(line) {

console.log(`The ${this.type} rabbit says ‘${line}’`);

}

let whiteRabbit = {type: “white”, speak};

let hungryRabbit = {type: “hungry”, speak};

whiteRabbit.speak(“Oh my fur and whiskers”);

// the white rabbit says ‘Oh my fur and whiskers’

hungryRabbit.speak(“Got any carrots?”);

// the hungry rabbit says ‘Got any carrots?’

* + 1. this is like an extra parameter that is passed in a different way then regular parameters
    2. to provide it explicitly, you can use the call() method, which takes the value of as its first argument and treats further arguments as normal parameters
    3. each function has its own this binding whose value depends on the way it is called
       1. means you can’t refer to the this of the wrapping scope in a regular function defined with the function keyword
    4. arrow functions do not bind their own this but can see the this binding of the scope around them
       1. allows the following code, which references this from inside a local function

let finder = {

find(array) {  
 return array.some(v => v == this.value);  
 },

value: 5

};

console.log(finder.find([4, 5])); // true

* code wouldn’t work if function keyword was used
  1. Prototypes
     1. Problem with rabbit code – have to add speak method to each rabbit object
        1. Would be better to create helper function that has rabbit type as a parameter and returns an object holding that as its type property and the speak function in a speak property
     2. Objects can be linked to other objects to get all the properties the other object has
        1. Useful for keeping an object type’s methods in a single place without having to add them to each object individually
        2. This can be done with prototypes
     3. Objects created with {} notation are linked to an object called Object.prototype
     4. Objects are collections of key-value pairs
     5. Can create objects two ways:
        1. Object literals – use curly braces to enclose the key-value pairs
        2. Use the new keyword to create new instances of an object

// creating object using object literal

let person = {  
 name: “John”,

age: 30,  
 occupation: “Developer”  
};

// creating object using constructor function  
function Person(name, age, occupation) {  
 this.name = name;  
 this.age = age;  
 this.occupation = occupation;  
}  
let person2 = new Person(“Bill”, 25, “Bricklayer”);

* + 1. Purpose of prototypes – when we want to create a new object that shares some or all of the properties and methods of an existing object
       1. Instead of copying all of the properties and methods of the existing object, we can create a new object that inherits from the existing object’s prototype
          1. Avoids duplicating code
          2. Simplifies object hierarchy
    2. Every object has a prototype, which will have its own prototype, and so on, until we reach a prototype that has null for its own prototype
    3. Standard way to access object’s prototype is Object.getPrototypeOf()
    4. When you try to access a property of an object, if the property cannot be found in the object itself, the prototype is searched, then the prototype’s prototype, and so on until the property is found or the end of the prototype chain is reached, in which case undefined is returned
       1. Ex: let numberObject = {  
           firstNum = 1,  
           secondNum = 2  
          };  
          console.log(numberObject.firstNum); // 1  
          console.log(numberObject.firstNum.toString()); // “1”  
           - numberObject doesn’t have a method toString() on it  
           - accesses toString() from Object.prototype
    5. Object.prototype is the most basic prototype – all objects have by default
       1. End of prototype chain
       2. Functions derive from Function.prototype and arrays derive from Array.prototype, but Function.prototype and Array.prototype have Object.prototype as their prototype  
          console.log(Object.getPrototypeOf(Math.max) == Function.prototype); // true  
          console.log(Object.getPrototypeOf(Object.getPrototypeOf(Math.max)) == Object.prototype); // true
    6. Can create an object inheriting from existing object’s prototype in two ways:
       1. Object.create() method
          1. Creates a new object with the specified prototype object  
             let protoRabbit = {  
              speak(line) {  
              console.log(`The ${this.type} rabbit says ${line}`);  
              }  
             };  
             let blackRabbit = Object.create(protoRabbit);  
             blackRabbit.type = “black”;  
             blackRabbit.speak(“I am fear and darkness.”);  
             console.log(Object.getPrototypeOf(blackRabbit) – protoRabbit); // true
          2. protoRabbit is container for properties shared by all rabbits
          3. blackRabbit is an individual rabbit object that contains properties that only apply to itself (type) and derives shared properties from its prototype
  1. Classes
     1. Prototype system is a somewhat free-form take on abstract data types or classes
     2. Class defines the shape of a type of object – what methods and properties it has
        1. Such an object is an instance of the class
     3. To create an instance of a given class, you have to make an object that derives from the proper prototype, but you also have to make sure it itself has the properties that instances of the class are supposed to have
     4. This is what a constructor function does  
          
        function makeRabbit(type) {  
         let rabbit = Object.create(protoRabbit);  
         rabbit.type = type;  
         return rabbit;  
        }
     5. Class notation makes it easier to define this type of function, along with a prototype object  
          
        class Rabbit {  
         constructor(type) {  
         this.type = type;  
         }  
         speak(line) {  
         console.log(`The ${this.type} rabbit says ${line}.`);  
         }  
        }
     6. class keyword allows us to define a constructor and a set of methods together
        1. any number of methods may be written inside the declaration’s braces
        2. code above has the effect of defining a binding called rabbit, which holds the function that runs the code in constructor and has a prototype property that holds the speak method
     7. constructors have to be called using the keyword new in front of them
        1. creates a fresh instance object whose prototype is the object from the function’s prototype property, then runs the function with ‘this’ bound to the new object, then finally returns the object
     8. should understand the distinction between the way a prototype is associated with a constructor (through its prototype property) and the way objects have a prototype (which can be found with Object.getPrototypeOf)
        1. actual prototype of a constructor is Function.prototype since constructors are functions
        2. the constructor functions prototype property holds the prototype used for instances created through it
        3. when a class does not define a constructor, it automatically gets an empty one
     9. You can declare properties directly in the class declaration
        1. Unlike methods, those properties are added to instance objects and not the prototype  
             
           class Particle {  
            speed = 0;  
            constructor(position) {  
            this.position = position;  
            }  
           }
     10. Like function, class can be used in both statements and expressions
         1. When used as an expression, it doesn’t define a binding but just produces the constructor as a value
            1. Means you can omit the class name in a class expression  
                 
               let object = new class { getWord() { return “hello”; } };  
               console.log(object.getWord()); // hello
     11. Classes can have private properties – properties and methods that are for internal use and not part of their interface
         1. All other properties are public properties, which are part of the object’s external interface
         2. To declare a private method, put a hashtag # in front of its name
            1. Such methods can only be called from the class declaration that defines them  
                 
               class SecretiveObject {  
                #getSecret() {  
                return “I ate all the plums”;  
                }  
                interrogate() {  
                let shallISayIt = this.#getSecret();  
                return “never”;  
                }  
               }
         3. To use private instance properties you must declare them in the class to be available at all  
              
            class RandomSource {  
             #max;  
             constructor(max) {  
             this.#max = max;  
             }  
            getNumber() {  
             return Math.floor(Math.random() \* this.#max);  
            }  
            }
  2. Overriding derived properties
     1. When you add a property to an object, whether it is present in the prototype or not, the property is added to the object itself
        1. If there was already a property with the same name in the prototype, the prototype property will no longer affect the object but will be hidden behind the object’s own property

Rabbit.prototype.teeth = “small”;  
console.log(killerRabbit.teeth); // small  
killerRabbit.teeth = “long, sharp, and bloody”;  
console.log(killerRabbit.teeth); // long, sharp and bloody  
console.log((new Rabbit(“basic)).teeth); // small  
console.log(Rabbit.prototype.teeth); // small

* + 1. Overriding properties is useful to express exceptional properties in instances of a more generic class of properties while letting the nonexceptional objects take a standard value from their prototype
  1. Maps
     1. Map as a noun is a data structure that associates values (the keys) with other values and remembers the original insertion order of the keys
        1. A key in the Map my only occur once – it is unique in the Map’s collection
        2. A Map object is iterated by key-value pairs
           1. A for … of loop returns a 2-member array of [key, value] for each iteration
        3. Iteration happens in insertion order - the order in which each key-value pair was first inserted into the map by the set() method
     2. Differences between object and map:
        1. Key types: A map’s keys can be of any data type (Including Object), but an Object’s key can only be a String or a Symbol
        2. Accidental keys: A Map has no default keys and can only contain the keys you explicitly put into it, but an Object prototype has default keys that could clash with your keys
        3. Size: You can quickly get the number of items in a Map using its size property, but you have to count manually in an Object (length property of the array returned from Object.keys())
        4. Performance: it is much easier to remove or add key value pairs in a Map than in an Object
        5. Iteration: a Map is an iterable, so it can be directly iterated (for…of) whereas an Object can use for…in to iterate over the enumerable properties of an object or can get an iterable for the object using Object.keys or Object.entries