JavaScript Class – June 24, 2024

Topics:

Date Objects

Call Stack

Optional Arguments

Closures

1. Date objects
   1. Allows us to manipulate dates
      1. By default, will display a date as a full text string using the browsers time zone
   2. The new Date() constructor is used to build date objects
   3. A new date object can be made in four ways:
      1. new Date()
      2. new Date(year, month, day, hours, minutes, seconds, milliseconds)
      3. new Date(milliseconds)
      4. new Date(date string)
   4. new Date() creates a date object that contains the current date and time
   5. date objects are always the same – do not tick like the computer clock
      1. new Date(year, month, …)  
         new Date(year, month,…) generates a new date object with the date and time supplied
      2. year, month, day, hour, minute, second and millisecond (in that sequence) are specified by seven numbers:
         1. let d = new Date(2024, 11, 24, 10, 33, 30, 0);
      3. months are counted from 0 to 11, not 1 to 12
   6. JavaScript stores dates as milliseconds from January 1, 1970 at 00:00:00 UTC (Universal Time Coordinated)
      1. The current date and time is January 1, 1970, milliseconds ago
   7. The values of date objects can be retrieved and set using either local or UTC/GMT time
   8. By default, JavaScript outputs dates in full string text format:
   9. Formatting date objects(“MMMM yyyy or MMM yyyy or MM yy, etc);
2. Call Stack
   1. function greet(who) {  
       console.log(“Hello “ + who);  
      }  
      greet(“Harry”);  
      console.log(“Bye”);
      1. call to greet causes control to jump to start of that function
      2. function calls console.log
      3. console.log takes control, does its job, returns control to function line
      4. control reaches end of function
      5. control calls console.log again for “Bye”
      6. program reaches end
      7. because computer has to jump back to where it was called when it returns, computer must remember context from which call happened
   2. Call stack – where computer stores context
      1. Tracks the execution of the program
      2. Tracks all of the functions that are currently running
      3. Last in – first out
         1. When function is called, it’s placed on top of stack
         2. When function returns, it removes top context from the stack and uses to continue execution
      4. When stack grows too big, computer fails – blows the stack
         1. Happens with infinite calls  
              
            function chicken() {  
             return egg();  
            }  
            function egg() {  
             return chicken();  
            }  
            console.log(chicken() + “ came first.”);
3. Optional arguments
   1. If more arguments passed than there are parameters in the function, computer will ignore extra arguments and just use what it has parameters for  
        
      function square(num) { return num \* num; }  
      console.log(square(4, true, “Ford”); // 16
   2. If not enough arguments are passed for the number of parameters in the function, the missing ones are assigned undefined
   3. These make it possible to pass wrong number of arguments without any warning
   4. Also allows you to call a function with different number of arguments on purpose  
        
      function minus(a, b) {  
       if (b === undefined) return -a;  
       else return a – b;  
      }  
        
      console.log(minus(10)); // -10  
      console.log(minus(10, 5)); // 5
   5. If equal sign after a parameter, value of expression will replace argument which is not given – makes argument optional  
        
      function roundTo(n, step = 1) {  
       let remainder = n % step;  
       return n – remainder + (remainder < step / 2 ? 0 : step);  
      };  
        
      console.log(roundTo(4.5)); // 5  
      console.log(roundTo(4.5, 2)); // 4
4. Closure
   1. function returnPlusTwo(num) {  
       let sum = num + 2;  
       return () => sum;  
      }  
        
      let threePlusTwo = returnPlusTwo(3);  
      let fivePlusTwo = returnPlusTwo(5);  
        
      console.log(threePlusTwo()); // 5  
      console.log(fivePlusTwo()); // 7  
      1. returnPlusTwo creates local binding (sum)
      2. returnPlusTwo returns another function that accesses and returns sum
      3. both instances of binding ( 3 and 5 ) are accessible
         1. proves local bindings are created anew for every call and different calls don’t affect each other’s local bindings
   2. being able to reference a specific instance of a local binding in an enclosing scope is called closure
   3. a function that references bindings from local scopes around it is called a closure
   4. Ex:  
      function multiplier(factor) {  
       return number => number \* factor;  
      }  
        
      let twice = multiplier(2);  
      console.log(twice(5));  
      1. multiplier is called and creates an environment in which its factor parameter is bound to 2
      2. function value it returns is stored in twice, which remembers the environment so that when it is called, it multiplies its argument by 2
   5. described another way – closure gives inner functions access to variables declared in the outer function scope, even after the outer function has returned
      1. applies to nested functions
      2. closure gives the function multiplier access to the argument used in the call to twice, which it uses for the variable number
   6. const createSecret = (secret) => {  
       return {  
       getSecret: () => secret,  
       setSecret: (newSecret) => {  
       secret = newSecret;  
       };  
       };  
      };  
        
      const mySecret = createSecret(“My secret”);  
      console.log(mySecret.getSecret()); // My secret  
        
      mySecret.setSecret(“My new secret”);  
      console.log(mySecret.getSecret()); // My new secret
   7. closure variables are live references to the outer-scoped variable, not a copy
      1. if you change outer-scoped variable, the change is reflected in the closure variable and vice versa
         1. means other functions declared in the same outer function will have access to those changes
   8. uses for closures:
      1. Data Privacy  
           
         const createCounter = () => {  
          let count = 0;  
          return {  
          increment: () => ++count,  
          decrement: () => --count,  
          getCount: () => count  
          };  
         };
      2. Curried functions and partial applications
         1. A curried function is one that takes multiple arguments one at a time  
              
            const add = (a) => (b) => a + b;
         2. A partial application is a function that has been applied to some but not yet all of its arguments  
              
            const increment = add(1) // partial application  
            increment(2); // 3
5. Destructuring arrays and objects
   1. Destructuring allows you to extract elements and properties from arrays and objects and assign them to variables in a concise and readable way

Object - without destructuring:  
const person = {  
 firstName: ‘John’,  
 lastName: ‘Doe’,  
 age: 30  
}  
const firstName = person.firstName, lastName = person.lastName, age = person.age;

console.log(`My name is ${firstName} ${lastName}, and I am ${age} years old.`);  
  
Object - with destructuring:  
const { firstName, lastName, age } = person;

console.log(`My name is ${firstName} ${lastName}, and I am ${age} years old.`);

Array – without destructuring  
const colors = [‘red’, ‘green’, ‘blue’];  
const firstColor = colors[0], secondColor = colors[1], thirdColor = colors[2];

console.log(`RGB stands for ${firstColor}, ${secondColor}, ${thirdColor};  
  
Array – with destructuring  
const [ firstColor, secondColor, thirdColor ] = colors;  
console.log(`RGB stands for ${firstColor}, ${secondColor}, ${thirdColor};

const inventory = [  
 { name: “apples”, quantity: 2 },  
 { name: “bananas”, quantity: 0 },  
 { name: “cherries”, quantity: 5 }  
];  
const result = inventory.find(({ name} ) => name === “cherries”);  
console.log(result); // { name: ‘cherries’, quantity: 5 }

* 1. Works for nested objects and arrays  
       
     const user = {  
      name: ‘Alice’,  
      age: 30,  
      address: {  
      city: ‘New York’,  
      country: ‘USA’  
      }  
     };  
     const { name, age, address: { city, country } } = user;  
     console.log(name, age, city, country);  
       
     const nestedNumberArray = [ 1, [ 2, 3 ], 4 ];  
     const [a, [ b, c ], d] = nestedNumberArray  
     console.log(a, b, c, d);
  2. Can assign default values to variables in case the property or element being destructured is undefined  
       
     const { firstName, lastName, age = 25, gender = ‘male’ } = person;  
     console.log(`My name is ${firstName} ${lastName}, and I am a ${age} year old ${gender}.`); // My name is John Doe and I am a 30 year old male.
  3. Rest syntax (…) allows you to gather remaining elements into a new array or object  
       
     const colors2 = [‘red’, ‘green’, ‘blue’, ‘yellow’, ‘orange’];  
     const [firstColor, secondColor, …otherColors] = colors2;  
     console.log(firstColor); // red  
     console.log(firstColor); // green  
     console.log(firstColor); // [‘blue’, ‘yellow’, ‘orange’]
  4. Can assign destructured object properties to variables with different names by using a colon followed by the new variable name within the object destructuring pattern  
       
     const { firstName: givenName, lastName: familyName, age: personage } = person;  
     console.log(`My name is ${ givenName } ${ familyName }, and I am ${ personage } years old.}’);
  5. Can use destructuring in function parameters in order to pass parameters as objects or arrays  
       
     const printPerson = ({name, age }) => {  
      console.log(`Name: ${name}, Age: ${age}`};  
     };  
       
     const thatPerson = { name: ‘John’, age: 30 };  
     printPerson(thatPerson); // Name: John, Age: 30

1. Rest parameters
   1. Allows you to represent an indefinite number of arguments as an array
      1. Means you can use any of the Array methods on it  
           
         double = (x) => x \* 2;  
         sum = (x, y) => x + y;  
         doubleAndSum = (…numbers) => numbers  
          .map(double)  
          .reduce(sum, 0)  
         doubleAndSum(1, 2, 3) // 12  
         doubleAndSum(1, 2, 3, 4) // 20  
         doubleAndSum(1, 2, 3, 4, 5) // 30
   2. You can name as many parameters as you want in your function before using rest, but rest has to be the last one specified, since it captures the rest of your arguments  
        
      someFunction = (a, b, c, …others) => {  
       console.log(a, b, c, others)  
      }  
      someFunction(‘one’, ‘two’, ‘three’, ‘four’, ‘five’, ‘six’);   
      // one two three [“four”, “five”, “six”]  
        
      \*\*\* pizzaOrder recursion uses rest parameter \*\*\*
2. Recursion
   1. A method that calls itself
   2. Two parts
      1. The recursion – where function is called
      2. The base case – the stopping condition
   3. Iterative vs recursive  
        
      // iterative factorial  
      function iterativeFactorial(n) {  
       let res = 1;  
       for (let i = n; i > 0; i++) {  
       res = res \* I;  
       }  
       return res;  
      }  
      console.log(iterativeFactorial(5)); // 120  
        
      // recursive factorial  
      function recursiveFactorial(n) {  
       if (n === 1) {  
       return 1;  
       }  
       return n \* recursiveFactorial(n – 1);  
      }  
      console.log(recursiveFactorial(5)); // 120
   4. Function can be called in multiple places, as well as multiple times in the same expression with likely different arguments
      1. That could make program crash with a stack overflow error
   5. There can be as many base cases as the algorithm requires
3. 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 this value 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
    5. Object.prototype is the most basic prototype – all objects have by default
       1. End of prototype chain
    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