JavaScript class – Jun 12 2024

1. Anonymous Functions
   1. When a function is created without using a function name
      1. Done with arrow functions and with assigning functions to variables
      2. Use if a function will only be used once in project
      3. done to avoid polluting namespace and avoid namespace collisions
      4. named function:  
         function functionName([arguments]) {  
          JavaScript statements  
         }
      5. function without a name:  
         function ([arguments]) {  
          JavaScript statements  
         }
      6. call by:
         1. assigning the function to a variable
         2. replacing a function call with the function itself
   2. Example:  
        
      function calculateTip(preTip, tipPercent) {  
       const tipResult = preTip \* tipPercent;  
       return tipResult;  
      }  
        
      const preTipTotal = 100.00;  
      const tipPercentage = 0.15;  
      const tipCost = calculateTip(preTipTotal, tipPercentage);  
      const totalBill = preTipTotal + tipCost;  
      console.log(“Your total bill is $” + totalBill):
   3. Rewrite to use anonymous function:  
        
      const preTipTotal = 100.00;  
      const tipPercentage = 0.15;  
        
      // Declare tipCost using an anonymous function  
      const tipCost = function(preTip, tipPercent) {  
       const tipResult = preTip \* tipPercent;  
       return (tipResult);  
      }  
        
      const totalBill = preTipTotal + tipCost(preTipTotal, tipPercentage);  
      console.log(“Your total bill is $” + totalBill);
2. Arrow Functions
   1. Uses fat arrow (=>) instead of function keyword  
      ([arguments] => {  
       JavaScript Statements  
      }  
        
      // Declare tipCost using an arrow function  
      const tipCost = (preTip, tipPercent) {  
       const tipResult = preTip \* tipPercent;  
       return (tipResult);  
      }  
        
      const roundTo = (n, step) => {  
       let remainder = n % step;  
       return n – remainder + (remainder < step / 2 ? 0 : step);  
      }
      1. Arrow comes after list of parameters
      2. Arrow followed by function’s body
   2. Arrow functions have implicit return feature
      1. ([arguments]) => statement
      2. If function body consists of a single expression, you can omit the return keyword
         1. JavaScript assumes that a single-statement function means function returns right after executing the statement – can leave out braces and return keyword  
              
            () => ‘foo’ is an arrow function that takes no parameters and returns the string ‘foo’
      3. To use implicit return for an object literal, you have to wrap the object in parentheses  
           
         const noop = () => { foo: ‘bar’ };  
         console.log(noop()); // undefined  
           
         const createFoo = () => ( { foo: ‘bar’ } );  
         console.log(createFoo()); // { foo: ‘bar’ }
   3. When only one parameter name, you can omit parentheses around the parameter list
   4. If the body is a single expression rather than a block in braces, the expression will be returned from the function
      1. These two definitions of square do the same thing:  
           
         const square1 = (x) => { return x \* x };  
         const square2 = x => x \* x;
   5. When no parameters at all, parameter list is just an empty set of parentheses
3. Objects
   1. Non-primitive data types that are programmable
   2. Objects are like real life objects such as houses, cars, people, animals or any other subjects
   3. Objects have traits that identify the object and actions that the object can take
      1. The traits are called properties, which are named values
      2. The actions are called methods, which are functions stored as properties
   4. Example  
      A real-life car can have a name (“Fiat”), a model (“500”), a weight (850kg), and a color(white) - properties  
      A real-life car can start, drive, brake and stop – methods
   5. Example of this object in code:  
      const car = {  
       name: "Fiat",  
       model: "500",  
       weight: 850,  
       color: "white",  
       start(speed) {  
       console.log("Starting car");  
       for (let i = 0; i <= speed; I += 5) {  
       console.log('accelerating at ' + i + " mph");  
       }  
       console.log("Reached desired speed");  
       },  
       makeAndModel: function() {  
       return this.name + “ “ + this.model; // “this” refers to the car object  
       }  
      };  
      console.log("the type of car is a " + car.name);   
      console.log(car.start(55));
   6. Almost everything in JavaScript is an object: objects, maths, functions, dates, arrays, maps, sets, all JavaScript values except primitives are objects
   7. An object literal is a list of name:value pairs inside curly braces
      1. Can also be called key:value pairs
   8. Line spaces and breaks are not important
      1. const person = {firstName: “Bob”, lastName: “Ross”, age: 50, eyeColor: “blue”);
   9. Can create an empty object two ways:
      1. const duck = {};
      2. const duck = new Object();
         1. better to use the first way for readability, simplicity and execution speed
   10. Ways to access properties:
       1. objectName.propertyName
          1. let firstGuestName = person.firstName;
       2. objectName[“propertyName”]
          1. let firstGuestName = person[“firstName”];
       3. objectName[expression]”
          1. let x = “firstName”;  
             let firstGuestName = person[x];
       4. difference between dot notation and bracket notation
          1. when using a dot, the word after the dot is the literal name of the property
          2. when using brackets, the expression between the brackets is evaluated to get the property name
          3. value.x fetches the value of the property named x
          4. value[x] takes the value of the variable named x and uses that, converted to a string, as the property name
   11. property names are strings
       1. dot notation only works with names that look like valid binding names – starting with letter or underscore and containing only letters, numbers and underscores
   12. elements in an array are stored as the array’s properties, using numbers as property names
       1. have to use bracket notation because you can’t use dot notation with numbers
   13. const person = {  
        name: "Bill",  
        age: 54,   
        2: "Thompson",  
        "Joe Cool": "Snoopy"  
       };  
       let i = "Joe Cool";  
       console.log(person["name"]);  
       person.ethnicity = "Asian";  
       console.log(person["ethnicity"]);  
       person["country"] = "USA";  
       console.log(person["country"]);  
       console.log(person);  
       person.name = "John";  
       console.log(person);  
       person["age"] = 49;   
       console.log(person);  
       person[i] = "Spike";  
       console.log(person);  
       delete person.name;  
       console.log(person.name);  
       console.log(person);  
       console.log("name" in person);  
       console.log("age" in person);  
       console.log(Object.keys(person));  
       /\* console.log(person.2); \*/  
       console.log(person[2]);   
       /\*person.2 = "Johnson"; \*/  
       person[2] = "Johnson";  
       console.log(person);  
       /\* person.'2' = "Baird"; \*/  
       /\* console.log(person); \*/  
       person['2'] = "Baird";  
       console.log(person);  
       person[1 + 1] = "Smith";  
       console.log(person);  
       /\* person.Joe Cool = "Sonny"; \*/  
       /\* console.log(person); \*/  
       /\* person."Joe Cool"] = "Sonny"; \*/  
       /\* console.log(person); \*/  
       /\* person[Joe Cool] = 'Sonny'; \*/  
       /\* console.log(person); \*/  
       person["Joe Cool"] = 'Sonny';  
       console.log(person);
   14. Objects can contain arrays and other objects  
         
       const myObj = {  
        name: “John”,  
        age: 30,  
        myCars: {  
        car1: “Ford”,  
        car2: “BMW”,  
        car3: “Fiat”  
        },  
        myClasses: [  
        {  
        className: “Science”,  
        roomNumber: “102”,  
        teacher: “O’Grady”  
        },  
        {  
        className: “History”,  
        roomNumber: “303”,  
        teacher: “Hanon”  
        }  
        ]  
       }
4. Auto-boxing
   1. Primitives have no methods but behave as if they do
   2. When properties are accessed on primitives, JavaScript auto-boxes the value into a wrapper object and accesses the property on that object instead
      1. Ex: toUpperCase() – string method
      2. toFixed() – number method
   3. The basis of object-oriented programming – using types of objects as the unit of program organization
   4. Idea is to set up program as a number of strictly separated object types
   5. Abstract data type, or object class, is a subprogram that may contain
5. 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. 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
      2. 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.”);
6. 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
7. 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
8. Objects
   1. JSON.stringify to print nested objects
9. Array destructuring
   1. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment>
10. Arrays
    1. slice
    2. Array.from() -- ???
    3. map() -- ???
    4. filter()
    5. Discuss difference between methods that alter existing array (like sort) vs methods that create a new array