香港中文大學
The Chinese University of Hong Kong

CSCI2720 - Building Web Applications

Lecture 9: JavaScript Functions and Arrays

Dr Colin Tsang

Outline

- More on functions
- Functions parameters
- Arrow functions
- Invoking functions
- Nested functions
- Callback functions
- Generator functions
- Object methods

- More on arrays
- Creating arrays
- Destructuring arrays
- Modifying arrays
- Iterating arrays
- Searching in arrays
- Transforming arrays

JS functions

- A JS function has the *function* keyword, an optional function name, optional function parameters, and an optional return
  - *Parameters*: the list of input names in function definition
  - *Arguments*: the actual values being passed at function call

```
function func(para1, para2, …) {
    // function body
    // optional return statement
    }
```

Function parameters

- Default values for parameters can be supplied, and *missing arguments* will begiven the undefined value

```
function f1(x, y=2, z) {
    console.log("x = " + x);
    console.log("y = " + y);
    console.log("z = " + z);
    }
```

```
f1(5);  // 2nd and 3rd arguments missing
"x = 5"
"y = 2"
"z = undefined"
```

- The function arguments can also be found in an *arguments* object without a parameter list
  - Note: this is not for arrow functions

```
function f2() {
    for (i of arguments) {
    console.log(i);
    }
    }
```

```
f2(1,2,3); // but f2() has no parameters
"1"
"2"
"3"
```

Function parameters

- A new way to obtain an unknown number of arguments: *rest operator* ...
  - The rest parameters must be the last item in the parameter list

```
function f3(x, y, ...more) {
    console.log("x is " + x);
    console.log("y is " + y);
    console.log(more);
    console.log(typeof more);
    }
```

```
f3(2,4,6,8,10);
"x is 2"
"y is 4"
// [object Array] (3)
[6,8,10]
"object"
```

Function parameters

- The rest operator can be used in arrow function syntax

```
let f4 = (a, ...b) =>
console.log(b);
```

```
f4(1, 3, 5);
// [object Array] (2)
[3,5]
```

- More about three dots: https://dev.to/sagar/three-dots---in-javascript-26ci

Function declaration vs expression

- In JavaScript, function codes are stored as plain values
  - Function declarations are *hoisted to the top* of the scope, i.e., used before being declared.

```
// function declaration
function f1(text) {
console.log("This is the f1 input: " + text);
}
// function expression with anonymous function
let f2 = function (text) {
console.log("This is the f2 input: " + text);
}
// arrow (anonymous) function in expression
let f3 = text => console.log("This is the f3 input: " + text);
```

```
console.log(f1);          // shows f1() code
function f1(text) {
 console.log("This is the f1 input: " + text);
}
console.log(typeof f1);
"function"
f1("a");                  // executes f1()
"This is the f1 input: a"
```

Arrow functions

- `(para1, para2, …) => { statements; }`
  - Brackets () for parameter list can be omitted for single parameter
  - Single-line: braces *{}* and *return* can be omitted.
    - `let square = num => num*num; // square(10) is 100`
  - Multi-line: braces *{}* and *return* must be present, like regular functions
  - Although similar, arrow functions are not the same as regular functions
    - No *this*, no *arguments*
    - No suitable as methods, constructors
  - See: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Invoking functions

- Functions are invoked (executed) using the parentheses () after the function/variable name
  - Without the parentheses, the function code is returned

- Why use immediately-invoked function expression (IIFE)?
  - Variables are not accessible from outside the scope, preventing potential naming conflicts.
  - One-time initialization or set up tasks (e.g., event listeners)

```
(function() {
  console.log("Hello there");
})();
(() => console.log("Hello again"))();
```

Invoking functions

- Common mistake for DOM events
  - A function should be used for events to invoke later!

```
<button id="btn1">Button 1</button>
<button id="btn2">Button 2</button>
<button onclick="alert('JS in HTML')">Button 3</button>

<script>
// without function, the statement is executed when loaded
document.querySelector("#btn1").onclick = alert("Wrong alert: too early");
// function code is executed only at onclick event
// either arrow function or regular function is okay
document.querySelector("#btn2").onclick = () => alert("Correct onclick alert");
</script>
```

Nested functions

- Functions in JS can be nested
  - Separation of variables in different scopes
    - Inner function can access variable of outer functions, but not vice versa
  - Multiple parentheses to invoke functions with function arguments

```
function f1(a) {
    function f2(b) {
    return a+b;
    }
    return f2;
    }
    console.log(f1(10)); // code of f2 is returned
    console.log(f1(10)(5)); // results of f2(5) is returned
```

Callback functions

- As functions are simply values, they can be passed as a function argument, and these are *callbacks*
  - More often used in asynchronous JS, where callbacks are called only after some waiting time or events

```
function f0(callback1, callback2) {
    let x = prompt("A number?");
    if (x%2)
    callback1();
    else
    callback2();
    }

    f0(
        ()=>alert("Odd"),
        ()=>alert("Even")
        );
```

Generator functions

- Generator functions can return a value and be re-entered later
  - Special keywords: *function** and *yield*
  - Re-entrance after the previous *yield* statement

```
<button onclick="alert(count.next().value)">Click</button>
    <script>
        function* g(inc) {
        let x = 0;
        while (1)
        yield x += inc;
        }
        let count = g(2);
    </script>
```

Object methods

- Object can contain functions, and they are called object methods

```
let human = {
    keyword: "Hello!",
    shout: function() { alert(this.keyword) }
    }
let human2 = {
    keyword: "Hello again!",
    shout() { alert(this.keyword) } // alternative shorter syntax for methods
    }
human.shout();
human2.shout();
```

- See: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Method_definitions

JS arrays

- A JS array is an ordered collection.
  - Data type is not limited
  - Can be functions, objects, and/or arrays
  - It is a special kind of object, i.e., it can be assessed like an object
  - Optimized with contiguous memory storage
- To verify if a variable/expression is an array, use **Array.isArray()**

JS arrays

- Arrays are copied by reference, like objects
  - See: https://javascript.info/object-copy

```
let x = [1,2,3];
console.log(Array.isArray(x)); // true

let y = x;
console.log(y); // [1,2,3]
y[1] = 0;
console.log(x); // [1,0,3]
console.log(y); // [1,0,3]
```

Creating arrays

- From an array-like object: with a length and indexed elements
  - See: https://javascript.info/iterable
- By combining other arrays
  - The *spread operator* ...
  - Note: the *rest operator* ... has the same syntax, but it is used in function

```
let s = "Hello World";
let array = Array.from(s);
console.log(array);
// ["H","e","l","l","o"," ","W","o","r","l","d"]
console.log(s.length); // 11
console.log(array.length); // 11
let a = [1,3,5];
let b = [2,4,6];
let c = [...a, 0, ...b];
console.log(c); // [1,3,5,0,2,4,6]
```

⇒ Otherwise, c = [a, 0, b]
Spread operator split the elements

Destructuring arrays

- An array can be destructured into separate variables.
- This makes it possible for a function to return multiple values.
- The rest operator is also supported

```
let a, b, restVar;
[a, b] = [10, 20];
console.log(a);
// 10
console.log(b);
// 20
[a, b, ...restVar] = [10, 20, 30, 40, 50];
console.log(restVar);
// [30,40,50]
```

Modifying arrays

- **array.slice(start, [end])**
  - Returns the new sliced array from *start* inclusive to *end* exclusive
  - Not modifying the original array / Remains unchanged

- **array.splice(start, [deleteCount, [itemsToAdd…]])**
  - Changes original and returns an array with deleted elements: *deleteCount* from *start* index.
  - Items are added to the original array from *start* index

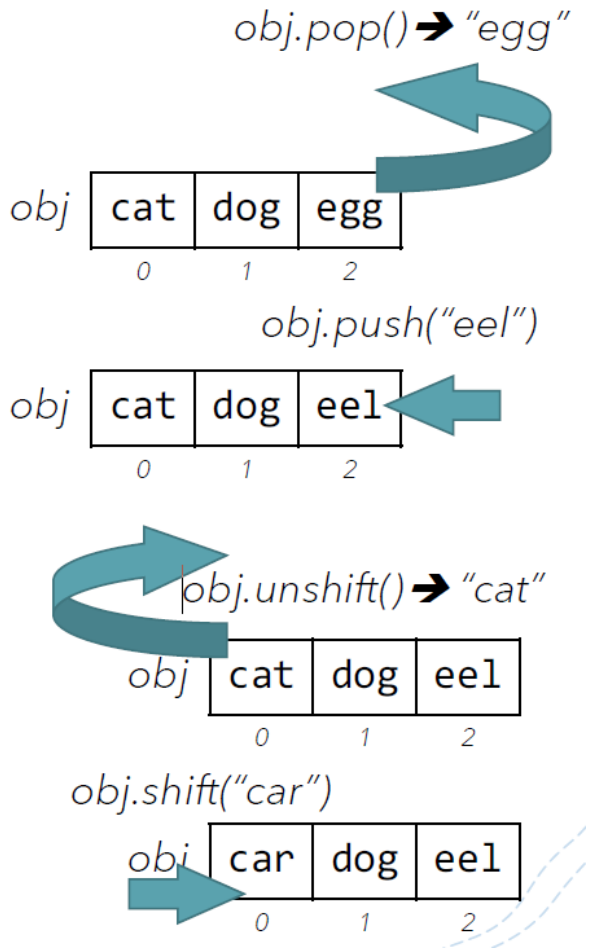
- Negative indices are accepted for *start* or *end*
  - -1 denotes last item, -2 second last, and so on.
  - Same as Python
- The array is updated.

Modifying arrays - example

```
let c = ["cyan","magenta","yellow","black"];
let c1 = c.slice(1,2);
console.log(c);
// ["cyan","magenta","yellow","black"]
console.log(c1);
// ["magenta"]
let c2 = c.splice(2,1,"red","green","blue");
console.log(c);
// ["cyan","magenta","red","green","blue","black"]
console.log(c2);
// ["yellow"]
```

Modifying arrays

- Array as stack (last-in-first-out LIFO)
  - **array.pop()** removes the last element and return it
  - **array.push(items)** adds items to the end of array
- Array as a queue (first-in-first-out FIFO)
  - **array.shift()** removes the first element and return it
  - **array.unshift(items)** adds items to the start of array
  - **push()** can be used to add items to the end of queue
- Original array is always modified
- Stack processes are faster since the array index is not affected
- See: https://javascript.info/array



Iterating arrays

- The traditional *for* loop allows flexible changes, and is fastest
- The *for…of* loop is handy for obtaining only a copy of the array elements (e.g., for displaying)
- The *forEach* loop takes a function as input with different levels of flexibility
  - *Callback* functions can also be used

Iterating arrays - example

```
let a = [1,3,5];
for (let i=0; i<a.length; i++) {
console.log(a[i]);
a[i] = a[i]+1;
}
console.log(a); // [2,4,6]
let b = [1,3,5];
for (let item of b) {
console.log(item);
item = item + 1;
}
console.log(b); // [1,3,5] not modified
let c = [1,3,5];
c.forEach(item=>item+1);
console.log(c); // [1,3,5] not modified
let d = [1,3,5];
d.forEach((item,i,d) => d[i]+=1);
console.log(d); // [2,4,6]
```

Searching in arrays

n%5 -> finding remainder -> 10%5 = 0 = False

```
let num = [10, 12, 13, 15, 20];
console.log( num.find(n => n%5) )
// 12
console.log( num.filter(n => n%5) )
// [12,13]
```

- **array.indexOf(item, start)**
- **array.lastIndexOf(item, start)**
  - Return the index if found (with === comparison) from start, or -1 if not found
- **array.includes(value)**
  - True if the array has the value
- **array.find(function(item, index, array))**
  - The way to match can be defined in the function
  - The first item returning true in function will be returned
- **array.filter(function(item,index.array))**
  - An array of matching items will be returned

Transform arrays

```
let num = [10, 12, 13, 15, 20];
console.log( num.map(n => n*2) )
// [20,24,26,30,40]
```

- **array.reverse()**
  - Reversing order of elements in array
- **array.split() / array.join()**
  - Converting a string to character array, or vice versa
- **array.map(function(item, index, array))**
  - a new array is returned with the transformation defined in function
- **sort([function(a,b)])**
  - Without the function, default sorting is comparing as string (e.g., 2>1000)
  - The function can decide how comparison should be done

HTMLCollection vs NodeList

getElementByID querySelector

- Both *HTMLCollection* and *NodeList* are *"array-like"* objects.
- An *"array-like"* object refers to an object that has some similarities to an array. It has numeric indices and a length property, allowing you to access its elements in a similar way to an array.
- However, it lacks the full set of methods, functions, and some other properties that arrays provide.
- The advantage of using *array-like objects* is that they are lighter in terms of memory usage and performance, which are important in web development.
- You can consider them as a *"cheap" version of the array*.
- You can turn them into an array by **Array.from()**. In this way, they turn from a "cheap" version to a fully functioning array.
- For *HTMLCollection*, it doesn't have built-in functions like **ForEach()**, **find()**, etc.
- *NodeList* reserves the above function, but it is still limited compared to an array. For example, *NodeList* cannot use **filter()**.

HTMLCollection vs NodeList

- *NodeList* is *static*, meaning that it won't update the property (e.g., length) even after we update the elements.
- *HTMLCollection* is *live*, in some sense similar to an array.
- In some cases, you can create a *live Nodelist*:
  - https://developer.mozilla.org/en-US/docs/Web/API/NodeList

Further readings

- MDN:
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions

- JavaScript.info array methods
  - https://javascript.info/array-methods