LESSON 1: REVIEW OF JAVASCRIPT FUNCTIONS AND ARRAYS

The Whole Is Greater than the Sum of the Parts

Wholeness: Programs are composed of many small functions that each have a singular purpose. JavaScript functions are first class meaning they can be passed and assigned as well as executed. JavaScript arrays are the most common data structure for collections of sequential data.

Main Point Preview: Programs are composed of functions

1. Programs are composed of functions
2. JavaScript functions are first class objects
3. Arrays

Main Point Preview: Programs are composed of functions

Functions are the fundamental components that do all the processing in JavaScript programs. Functions should be short and self-contained. Science of Consciousness: Just like clear self-contained functions promote successful programs, clear self-sufficient awareness promotes successful action in life.

Global variables (avoid)

- Variables declared outside of any function are called global.
  - visible from any function
  - Avoid globals

```
let userName = 'John';

function showMessage() {
  userName = "Bob"; // (1) changed the outer variable
  let message = 'Hello, ' + userName;
  alert(message);
}

alert( userName ); // John before the function call
showMessage();
alert( userName ); // Bob, the value was modified by the function
```

Shadowing (avoid)

- ‘shadowing’ is when a local variable is given the same name as a more global one
  - they are different variables
  - the global one is hidden or shadowed by the local one
  - can be confusing for humans reading code

```
function showMessage(from, text) {
  from = '*' + from + '*'; // make "from" look nicer
  console.log(from + ': ' + text);
}
let from = "Ann";
showMessage(from, "Hello"); // *Ann*: Hello
// the value of "from" is the same, the function modified a local copy
// this is called ‘shadowing’, best to avoid shadowing for clarity
console.log(from); // Ann  or * Ann * ?
```

- If a parameter is not provided, then its value becomes ***undefined***.
  - showMessage(‘Beth’) → ??

Default values

- a default value can be specified for a parameter
  - if no value is passed the parameter will get the default value instead of ‘undefined’

```
function showMessage(from, text = ‘no text given’) {
  from = '*' + from + '*’;
  console.log(from + ': ' + text);
}
```

Returning a value

- return can be in any place of the function. When the execution reaches it, the function stops, and the value is returned to the calling code (assigned to result above).
  - return without a value. That causes the function to exit immediately.
  - A function with an empty return or without it returns undefined

```
function checkAge(age) {
  if (age > 18) {
    return true;
  } else {
    return confirm('Do you have permission from your parents?');
  }
}

let age = prompt('How old are you?', 18);

if ( checkAge(age) ) {
  alert( 'Access granted' );
} else {
  alert( 'Access denied' );
}
```

Never add a newline between return and the value

- For a long expression in return, it might be tempting to put it on a separate line, like this:

```
return
 (some + long + expression + or + whatever * f(a) + f(b))
```

- That doesn't work, because JavaScript assumes a semicolon after return. That'll work the same as:
  - effectively becomes an empty return

```
return;
 (some + long + expression + or + whatever * f(a) + f(b))
```

- If we want the returned expression to wrap across multiple lines, we should start it at the same line as return. Or at least put the opening parentheses there as follows:

```
return (
  some + long + expression
  + or +
  whatever * f(a) + f(b)
  )
```

One function – one action (design guideline)

- A function should do exactly what is suggested by its name, no more.

- Two independent actions usually deserve two functions, even if they are usually called together (in that case we can make a 3rd function that calls those two).

- A few examples of breaking this rule:
  - getAge –bad if shows an alert with age
    - should only get
  - createForm –bad if modifies document, adding a form to it
    - should only create and return
  - checkPermission –bad if it displays access granted/denied message
    - should only perform the check and return the result

Command/query separation principle (design guideline)

- Principle: a function should be either a command or a query but not both
  - a command has a side effect, makes some state change external to the function
    - console.log
    - write to a database
    - modify a global variable
  - a query function returns a value and has no side effect
  - pure query functions are more modular and reusable
    - functions that return values and have side effects often lead to surprising bugs
      - developers see the return value and want to reuse
      - easy to overlook the side effect, which causes unexpected outcomes in reuse

- A few examples of breaking this rule:
  - getAge –bad if it shows an alert with the age (should only get).
  - createForm –bad if modifies document, adding a form to it (should only create and return).
  - checkPermission –bad if displays access granted/denied message (should only perform check and return result).

Functions == Comments

- Functions should be short and do exactly one thing.
  - 30-second rule: if it takes more than 30 seconds to read and understand a function, then it should be split up
- A separate function is not only easier to test and debug – its very existence is a great comment!
- For instance, compare the two functions showPrimes(n) below. Each one outputs prime numbers up to n.

```
function showPrimes(n) {
  nextPrime: for (let i = 2; i < n; i++) {

    for (let j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }

    alert( i ); // a prime
  }
}
```

```
function showPrimes(n) {
  for (let i = 2; i < n; i++) {
    if (!isPrime(i)) continue;
    alert(i);  // a prime
  }
}
function isPrime(n) {
  for (let i = 2; i < n; i++) {
    if ( n % i == 0) return false;
  }
  return true;
}
```

Tasks

- Complete the following tasks from The JavaScript Language > JavaScript Fundamentals > Functions
  - Is "else" required?
  - Rewrite the function using '?' or '||'
  - Function min(a, b)
  - Function pow(x,n)

Main Point: Programs are composed of functions

Functions are the fundamental components that do all the processing in JavaScript programs. Functions should be short and self-contained. Science of Consciousness: Just like clear self-contained functions promote successful programs, clear self-sufficient awareness promotes successful action in life.

Main Point Preview: JS functions are first class objects

JavaScript functions are executed if they end with parentheses. They can also be treated as values, parameters, and return values.

Functions as values

- Functions are first-class objects in JavaScript
  - stored in a variable, object, or array.
  - passed as an argument to a function.
  - returned from a function

```
function sayHi() {
  alert( "Hello" );
}
const myHi = sayHi;
alert( sayHi ); // shows the function code
function higherOrder() { return sayHi; }
```

- ( ) after function name is an important syntactic construct
  - means that there is a function and it should be executed

Semicolon rules

- why does Function Expression end with semicolon ; but not Function Declaration

```
function sayHi() {
  // ...
}

let sayHi = function() {
  // ...
};
```

- ; not needed at the end of code blocks and syntax structures that use them
  - like if { ... }, for { }, function f { } etc.

- A Function Expression is used as part the statement: let sayHi = ...;, as a value.
  - It's not a code block, but rather an assignment statement
  - semicolon ; is recommended at the end of statements.
  - semicolon here is not related to the Function Expression itself, it just terminates the statement

Callback functions

- more examples of passing functions as values and using function expressions
- arguments showOk and showCancel of ask are called callback functions or callbacks
- idea is that we pass a function and expect it to be “called back” later
  - showOk becomes callback for “yes” answer
  - showCancel for “no” answer

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}
function showOk() {
  alert( "You agreed." );
}
function showCancel() {
  alert( "You canceled the execution." );
}
ask("Do you agree?", showOk, showCancel);
```

```
//more succinct function expression version (anonymous functions)
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}

ask( "Do you agree?",
  function() { alert("You agreed."); },
  function() { alert("You canceled the execution."); }
```

Function Expression vs Function Declaration

- *Function declaration:* declared as a separate statement, in the main code flow
- *Function expression*: created inside an expression or inside another syntax construct.
  - E.g., at the right side of the “assignment expression” =:

- when created by JS engine?
  - A Function Expression is created when the execution reaches it and is usable only from that moment.
  - A Function Declaration can be called earlier than it is defined.

- When to choose Function Declaration versus Function Expression?
  - first consider Function Declaration
    - more freedom in how to organize our code, because can call such functions before they are declared.
    - better for readability, Function Declarations are more “eye-catching”
  - if a Function Declaration does not suit us for some reason, then function expression
    - anonymous functions (e.g., DOM event handlers often are not reusable)
    - E.g., need a conditional declaration (jsfiddle link)

Arrow functions

- Can be used in the same way as function expressions
  - More succinct, advantageous for short anonymous callbacks
  - Fix language issue involving inner functions (will discuss with objects)

```
//function expression
let sum = function(a, b) {
  return a + b;
};

//equivalent arrow function
let sum = (a, b) => a + b;

//only one argument, then parentheses around parameters can be omitted
let double = x => 2 * x;

//no arguments, parentheses should be empty (but they should be present):
let sayHi = () => alert("Hello!");

//if body has { } brackets then must use return
let sum = (a, b) => { return a + b; }
```

- Exercise: **Rewrite with arrow functions**

Semicolon rules

- Most codestyle guides agree that we should put a semicolon after each statement.

- Semicolons are not required after code blocks {...} and syntax constructs with them like loops:

```
function f() {
  // no semicolon needed after function declaration
}

for(;;) {
  // no semicolon needed after the loop
}
```

- …But even if we can put an “extra” semicolon somewhere, that’s not an error. It will be ignored.

Strict mode

- To fully enable all features of modern JavaScript, we should start scripts with "use strict“

```
'use strict';
...
```

- The directive must be at the top of a script or at the beginning of a function body.

Main Point: JS functions are first class objects

JavaScript functions are executed if they end with parentheses. They can also be treated as values, parameters, and return values.

Main Point Preview: Arrays

Arrays are fundamental data structures for ordered collections, where we have a 1st, a 2nd, a 3rd element and so on.

8/9 data types

1) number for numbers of any kind: integer or floating-point, integers are limited by ±253.
2) bigint is for integer numbers of arbitrary length.
3) string A string may have zero or more characters,
    1) there's no separate single-character type.
4) boolean for true/false.
5) null for unknown values – a standalone type that has a single value null.
6) undefined for unassigned values – a standalone type that has a single value undefined.
7) object for more complex data structures.
8) symbol for unique identifiers.
9) function : a non-data structure, though it also answers for typeof operator: typeof instance === "function".

- The typeof operator returns the type for a value, with two exceptions:

```
typeof null == "object" // error in the language
typeof function(){} == "function" // functions are treated specially
```

Operators (review)

Arithmetic

- Regular: * + - /, also % for the remainder and ** for power of a number.
- The binary plus + concatenates strings. And if any of the operands is a string, the other one is converted to string too:

```
alert( '1' + 2 ); // '12', string
alert( 1 + '2' ); // '12', string
```

Assignments

- There is a simple assignment: a = b and combined ones like a *= 2.
- Conditional
- The only operator with three parameters: cond ? resultA : resultB. If cond is truthy, returns resultA, otherwise resultB.

Operators2 (review)

- Logical operators
  - Logical AND && and OR || perform short-circuit evaluation and then return the value where it stopped
  - Logical NOT ! converts the operand to boolean type and returns the inverse value.

- Nullish coalescing operator
  - The ?? operator provides a way to choose a defined value from a list of variables.
  - The result of a ?? b is a unless it's null/undefined, then b.

- Comparisons
  - Equality check == for values of different types converts to a number

```
alert( 0 == false ); // true
alert( 0 == “”); // true
```

  - Other comparisons convert to a number as well.
  - The strict equality operator === doesn't do the conversion: different types always mean different values for it.
  - Values null and undefined are special: they equal == each other and don't equal anything else.
  - Greater/less comparisons compare strings character-by-character, other types are converted to a number.

Interaction

When using a browser as a working environment, basic UI functions will be:

prompt(question, [default])
Ask a question, and return either what the visitor entered or null if they clicked “cancel”.

confirm(question)
Ask a question and suggest to choose between Ok and Cancel. The choice is returned as true/false.

alert(message)
Output a message.

Arrays

```
let fruits = ["Apple", "Orange", "Plum"];

alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum

fruits[2] = 'Pear'; // now ["Apple", "Orange", "Pear"]
fruits[3] = 'Lemon'; // now ["Apple", "Orange", "Pear", "Lemon"]
```

Stacks and Queues

Queue: ordered collection of elements which supports two operations: (FIFO)

push appends an element to the end.

shift get an element from the beginning, advancing the queue, so that the 2nd element becomes the 1st

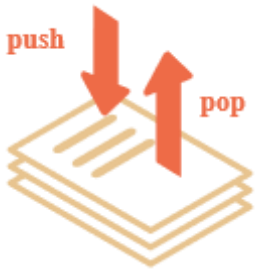
Stack: (LIFO)

push adds an element to the end.

pop takes an element from the end.

So new elements are added or taken always from the “end”.

A stack is usually illustrated as a pack of cards: new cards are added to the top or taken from the top:



Arrays in JavaScript can work both as a queue and as a stack.

Methods that work with the end of the array:

pop Extracts the last element of the array and returns it:

```
let fruits = ["Apple", "Orange", "Pear"];
alert( fruits.pop() ); // remove "Pear" and alert it
alert( fruits ); // Apple, Orange
```

push Append the element to the end of the array:

```
let fruits = ["Apple", "Orange"];
fruits.push("Pear");
alert( fruits ); // Apple, Orange, Pear
```

The call fruits.push(...) is equal to fruits[fruits.length] = ...

Methods that work with the beginning of the array:

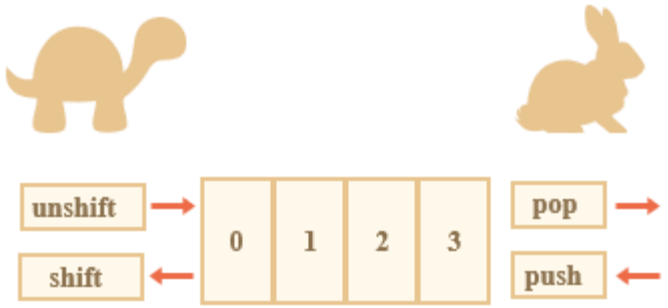
shift Extracts the first element of the array and returns it:

```
let fruits = ["Apple", "Orange", "Pear"];
alert( fruits.shift() ); // remove Apple and alert it
alert( fruits ); // Orange, Pear
```

unshift Add the element to the beginning of the array:

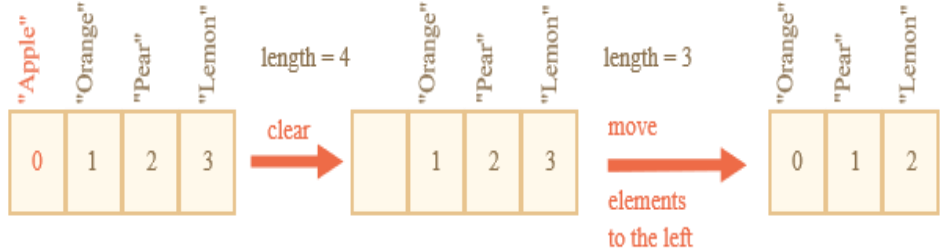
```
let fruits = ["Orange", "Pear"];
fruits.unshift('Apple');
alert( fruits ); // Apple, Orange, Pear
```

Methods push/pop run fast, while shift/unshift are slow:

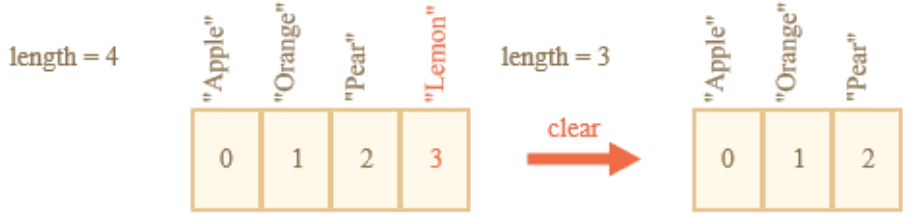


The shift operation must do 3 things:

1. Remove the element with the index 0.
2. Move all elements to the left, renumber them from the index 1 to 0, from 2 to 1 and so on.
3. Update the length property.



push/pop? They do not need to move anything. To extract an element from the end, the pop method cleans the index and shortens length.



Loops

One of the oldest ways to cycle array items is the for loop over indexes:

```
let arr = ["Apple", "Orange", "Pear"];
for (let i = 0; i < arr.length; i++) {
  alert( arr[i] );}
```

But for arrays there is another form of loop, for..of:

```
let fruits = ["Apple", "Orange", "Plum"];
// iterates over array elements
for (let fruit of fruits) {
  alert( fruit );}
```

- Favor for..of if do not need indices
  - Less opportunities for bugs
  - Problems with beginning or end of array indices, etc

Multi-dimensional arrays and toString

Arrays can have items that are also arrays. We can use it for multidimensional arrays, for example to store matrices

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];
alert( matrix[1][1] ); // 5, the central element
```

Arrays have their own implementation of toString method that returns a comma-separated list of elements

```
let arr = [1, 2, 3];
alert( arr ); // 1,2,3
alert( String(arr) === '1,2,3' ); // true
```

Exercises

- Is array copied?
- Array operations.
- Calling in an array context
- Sum input numbers
- A maximal subarray

Main Point: Arrays

Arrays are fundamental data structures for ordered collections, where we have a 1st, a 2nd, a 3rd element and so on.

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

The Whole Is Greater than the Sum of the Parts

1. Key components of JavaScript programs are functions and arrays.

2. Functions can be values as well as executable instructions.

---

3. **Transcendental consciousness** is the experience of the wholeness that is our own awareness and the field in which all other experiences live.

4. **Impulses within the transcendental field:** Thoughts from this level are naturally integrated and coherent and infused with the wholeness of pure awareness.

5. **Wholeness moving within itself:** In unity consciousness we appreciate the value of wholeness in all the components of daily life.