CS472 WAP
Lecture 3:Introduction to JavaScript

Maharishi International University - Fairfield, Iowa

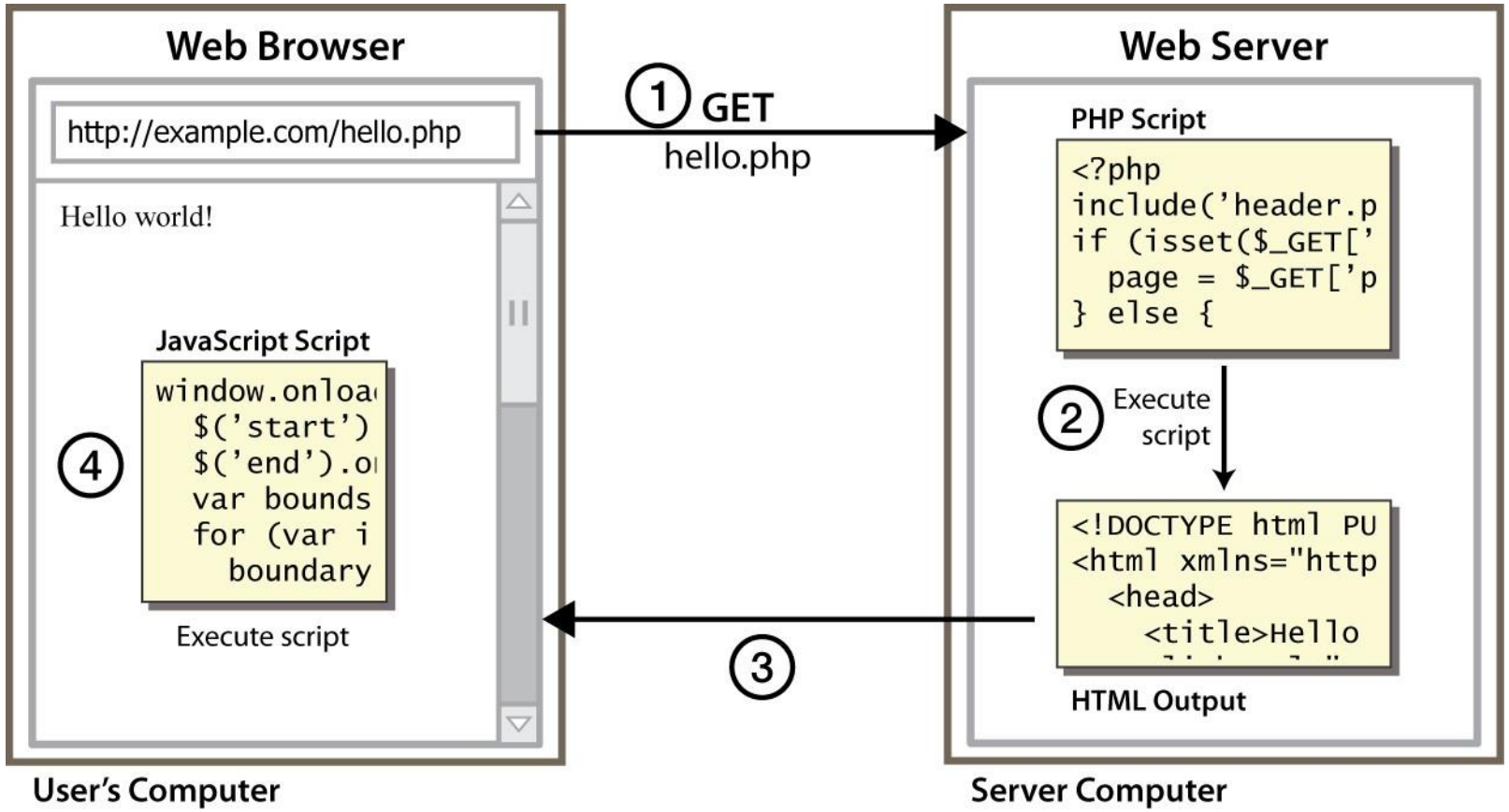
Main Point Preview

- JavaScript is a loosely typed language. It has types but does no compile time type checking. Programmers must be cautious of automatic type conversions, including conversions to Boolean types. It has a flexible and powerful array type as well as distinct types of null and undefined.

- **Science of Consciousness:** To be an effective JavaScript programmer one needs to understand the principles and details of the language. If our awareness is established in the source of all the laws of nature, then our actions will spontaneously be in accord with the laws of nature for a particular environment.

Client-side Scripting

- client-side script: code runs in browser *after* page is sent back from server
  - often this code manipulates the page or responds to user actions



Why use client-side programming?

- client-side scripting (JavaScript) benefits:
  - usability: can modify a page without having to post back to the server (faster UI)
  - efficiency: can make small, quick changes to page without waiting for server
  - event-driven: can respond to user actions like clicks and key presses

What is JavaScript?

- a lightweight programming language ("scripting language")

- used to make web pages interactive
  - insert dynamic text into HTML (ex: username)
  - react to events (ex: page load user click)
  - get information about a user's computer (ex: browser type)
  - perform calculations on user's computer (ex: form validation)

- NOT related to Java other than by name and some syntactic similarities

Your first JS file

To get JS Engine starts in your browser, all you need is to add the following code:

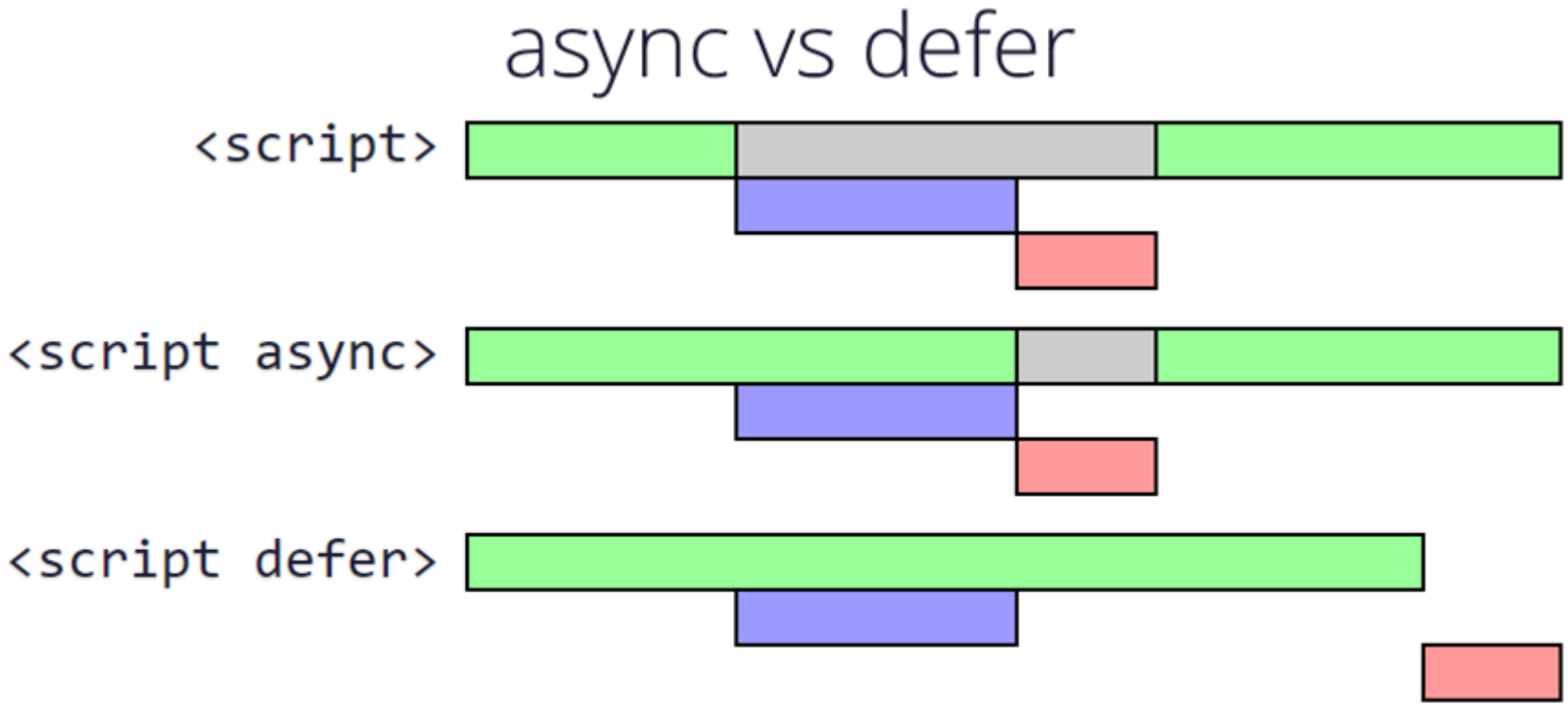
```
<script src="script.js" type="text/javascript"></script>
```

- The JS Engine will create all the **global objects** along with "**this**"
- All your code (variables and functions) will be attached to the global object **window**

What will happened if we have two or more scripts included? They all run as one single file!

```
<script src="script1.js" type="text/javascript"></script>
<script src="script2.js" type="text/javascript"></script>
```

Your first JS file



The modern mode, "use strict"

- Strict mode makes it easier to write "secure" JavaScript.
- Strict mode changes previously accepted "bad syntax" into real errors.

- Examples:

  The strict mode in JavaScript does not allow following things:

  1. Use of undefined variables
  2. Use of reserved keywords as variable or function name
  3. Duplicate properties of an object
  4. Duplicate parameters of function
  5. Assign values to read-only properties
  6. Modifying arguments object
  7. Octal numeric literals
  8. with statement

- Ref: https://www.tutorialsteacher.com/javascript/javascript-strict

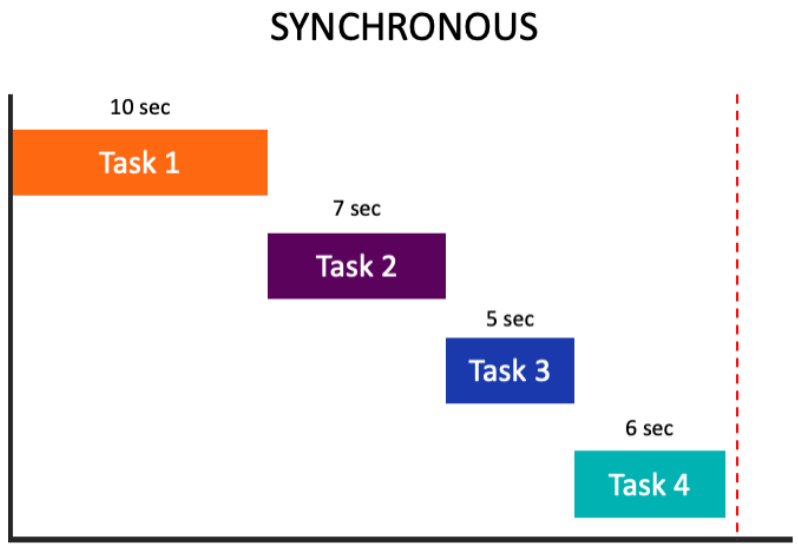
JS Engine

- An implementation of ECAM specifications. A program that converts JS code into something that a computer processor understands (machine code that runs in the CPU), should follow the ECMAScript standard of how the language should work and what features it should have.
- **Google**: V8
- **Mozilla**: Spider Monkey
- **Microsoft**: Chakra Core
- **Apple**: JavaScript Core

JS is a Synchronous Language

JavaScript is a **Synchronous language**, which means the code is interpreted and compiled line-by-line.

There is one thread running the code, if one line takes a long time to execute (blocking), the next line has to wait until the one before completely finishes.



Variables and types

```
var name = expression;
let name = expression; (ES6)
const name = expression; (ES6)

var age = 32;
let weight = 127.4;
const clientName = "Connie Client";
```

- variables are declared with the `var/let/const` keyword (case sensitive)
- types are not specified, but JS does have types ("loosely typed")

8 basic data types in JavaScript

Seven primitive data types:

1. **number** for numbers of any kind: integer or floating-point.
2. **string** for strings.
   A string may have one or more characters, there's no separate single-character type.
3. **boolean** for true/false.
4. **null** for unknown values – a standalone type that has a single value null.
5. **undefined** for unassigned values – a standalone type that has a single value undefined.
6. **symbol** for unique identifiers.
7. **bigInt** for integers larger than 2**53

Non-primitive data type:

- **object** for more complex data structures.
- Arrays and functions are objects

- The **typeof** operator allows us to see which type is stored in a variable.
  - Two forms: typeof x or typeof(x).

JavaScript Primitives

Type	**`typeof` return value**	**Object wrapper**
null	**`"object"`**	N/A
undefined	**`"undefined"`**	N/A
boolean	**`"boolean"`**	**Boolean**
number	**`"number"`**	**Number**
bigint	**`"bigint"`**	**BigInt**
string	**`"string"`**	**String**
symbol	**`"symbol"`**	**Symbol**

The JS engine assigns a value of **`undefined`** for a variable that is declared but is not assigned a value yet. Developers assign the **`null`** value to a variable that don't have an initial value.

dynamic (loose) typing

- Dynamic typing

- JavaScript is a loosely typed or a dynamic language. Variables in JavaScript are not directly associated with a specific value type, and any variable can be assigned (and re-assigned) values of all types:

```
let foo = 42;          // foo is now a number
foo = 'bar';           // foo is now a string
foo = true;            // foo is now a boolean
```

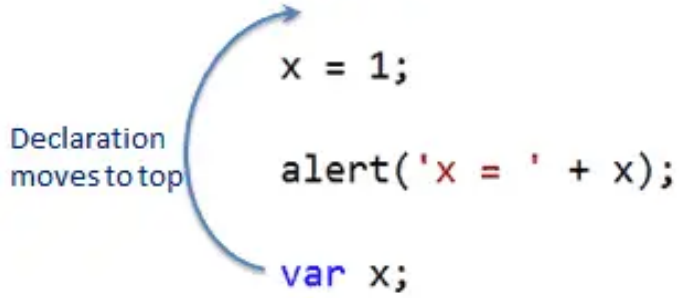
Scope of variables

- The scope of a variable determines how long and where a variable can be used.
- When const or let keywords are used, scope is within nearest the block {}

```
let x = 5;
console.log(x);
if(x==5){
        let y = 2*x;
        console.log(y);
        console.log(x); // x is accessible here.
}
console.log(x);
console.log(y); // y is not accessible here.
```

- Declare y using `var` keyword in above code and see the change in output.

Hoisting



- Hoisting is JavaScript's default behavior of moving all declarations to the top of the current scope.

- Variables defined with **var** have function-scoped or global-scoped if declared outside function.
- Variables defined with **const, let** have block-scoped.

- Variables defined with **var** are hoisted to the top and can be initialized at any time.
- Variables defined with **let**, **const** are also hoisted to the top of the block, but not initialized.
  - ✓ Meaning: Using a let variable before it is declared will result in a ReferenceError.
  - ✓ The block of code is aware of the variable, but it cannot be used until it has been declared.

Number

- The *number* type represents both integer and floating-point numbers.
- Besides regular numbers, there are so-called “special numeric values” which also belong to this data type: Infinity, -Infinity, and NaN.

```
let n = 123;
n = 12.345;
console.log( 1 / 0 ); // Infinity
console.log( "not a number" / 2 ); // NaN
```

- BigInt type was added to the language to represent integers of arbitrary length.
- integers and real numbers are the same type (no int vs. double)
- same operators: + - * / % ++ -- = += -= *= /= %=
- similar precedence to Java
- many operators auto-convert types: "2" * 3 is 6

Numeric Conversion

- Numeric conversion happens in mathematical functions and expressions automatically.
  - For example, when division / is applied to non-numbers:

```
alert( "6" / "2" ); // 3, strings are converted to numbers
```

  - We can use the Number(value) function to explicitly convert a value to a number:

```
let str = "123.33";
let num = Number(str); // becomes a number 123.33
num = parseFloat(str); // becomes a number 123.33
num = parseInt(str); // becomes a number 123
```

Numeric conversion rules

Value	Becomes…
undefined	NaN
null	0
true and false	1 and 0
string	Whitespaces from the start and end are removed. If the remaining string is empty, the result is 0. Otherwise, the number is “read” from the string. An error gives NaN.

```
console.log( Number("   123   ") ); // 123
console.log( Number("123z") );      // NaN (error reading a number at "z")


console.log( Number(true) );        // 1
console.log( Number(false) );       // 0
```

Boolean Type

The `boolean` type has only two values: `true` and `false`.

```
let isLazy = false;
let isHealthy = true;
```

```
let iLikeWebApps = true;
if ("web dev is great") { /* true */ }
if (0) { /* false */ }
```

- Any value can be used as a Boolean
  - "falsey" values: false, 0, 0.0, NaN, empty String "", null, and undefined
  - "truthy" values: anything else, include objects

- Boolean Conversion:

```
console.log( Boolean(1) ); // true
console.log( Boolean(0) ); // false
console.log( Boolean("hello") ); // true
console.log( Boolean("") ); // false
```

Logical Operators

- >, <, >=, <=, &&, ||, ! ==, !=, ===, !==
- most logical operators automatically convert types:
  - 5 < "7" is true
  - 42 == 42.0 is true
  - "5.0" == 5 is true
- === and !== are **strict** equality tests; checks both type and value
  - "5.0" === 5 is false

- Always use **strict** equality

Comments (same as Java)

```
// single-line comment
/* multi-line comment */
```

- identical to Java's comment syntax
- recall: 4 comment syntaxes
  - HTML: <!-- *comment* -->
  - CSS/JS/PHP: /* *comment* */
  - Java/JS/PHP: // *comment*
  - Python: # *comment*

JavaScript String

You may create a string literal with single quotation `''`, double quotation `""`, or back-ticks `` ` ` ``.
Back-tick strings are multiple lines and allow variables.

```
let str = "Hello";
let str2 = 'Single quotes are ok too (but we prefer double)';
let phrase = `can embed another ${str} in a backtick quote`;      //backtick
```

- methods: charAt, at, indexOf, lastIndexOf, split, substring, toLowerCase, toUpperCase, includes, startsWith, localeCompare, …
  - charAt returns a one-letter String (there is no char type)

- length is property
- concatenation with + : 1 + 1 is 2, but "1" + 1 is "11"

The `Try-Catch` Statement

Wrap a block of code in a try block and catch potential errors in the catch block, you can prevent your program from crashing when an error occurs. Inside catch blocks: Access message, name, stack trace, etc.

```
try {
  // Code that may throw an error
  const result = undefinedVar + 10;
} catch (error) {
   // Output: An error occurred: undefinedVar is not defined
   console.log("An error occurred:", error.message);
}
```

Error Object

The **`Error`** object in JavaScript plays a crucial role in error handling.

It is an object created when an error occurs during runtime.

We can also create it manually using the `Error` constructor:
`new Error("message")`

It contains information about the error, including its message, name, stack trace, and other properties.

Throwing an Error

```
try {
  // Throw an error
  throw new Error("Something went wrong")
} catch (error) {
   // Output: An error occurred: Something went wrong
   console.log("An error occurred:", error.message);
}
```

JavaScript Functions

- Function Declaration **`function foo(){}`**
- Function Expression **`const foo = function(){}`**
- Arrow Function **`const foo = ()=>{}`**
- Anonymous Function (to be used once, mostly as callback)
- IIFE (Immediately Invoked Function Expression) **`(f)()`**

A function returns `undefined` without an explicit return statement.

JavaScript Array

Arrays are dynamic and may hold different types (not good practice), they have a length property that grows as needed when elements are added.

```
const items = ['plate', 'fork', 'spoon’];
//or const items = Array.of('plate', 'fork', 'spoon');
//or const items = new Array(['plate', 'fork', 'spoon’]);

console.log(items[0]) // plate
```

Demo Array API: at, push, concat, includes, splice, find, findIndex, map, filter, reduce, sort, Array.isArray, Array.from, Array.of, …

Demo Array Iteration: forEach, for-of, for-in

Array methods summary

- To add/remove elements:
  - push(...items) – adds items to the end,
  - pop() – extracts an item from the end,
  - splice(pos, deleteCount, ...items) – at index pos delete deleteCount elements and insert items.
  - slice(start, end) – creates a new array, copies elements from position start till end (not inclusive) into it.
  - concat(...items) – returns a new array: copies all members of the current one and adds items to it. If any of items is an array, then its elements are taken.
- To search among elements:
  - indexOf/lastIndexOf(item, pos) – look for item starting from position pos, return the index or -1 if not found.
  - includes(value) – returns true if the array has value, otherwise false.
  - find/filter(func) – filter elements through the function, return first/all values that make it return true.
  - findIndex(func) is like find, but returns the index instead of a value.
- To iterate over elements:
  - forEach(func) – calls func for every element, does not return anything.
- To transform the array:
  - map(func) – creates a new array from results of calling func for every element.
  - sort(func) – sorts the array in-place, then returns it.
  - reverse() – reverses the array in-place, then returns it.
  - split/join – convert a string to array and back.
  - reduce(func, initial) – calculate a single value over the array by calling func for each element and passing an intermediate result between the calls.

splice

- The arr.splice(str) method is a swiss army knife for arrays.
  - It can do everything: insert, remove and replace elements.

```
arr.splice(index [, deleteCount, elem1, ..., elemN])
```

- It starts from the position index:
  - removes deleteCount elements and then
  - inserts elem1, ..., elemN at their place.
  - Returns the array of removed elements.

deletion:

```
let arr = ["I", "study", "JavaScript"];
arr.splice(1, 1);         // from index 1 remove 1 element
console.log( arr );       // ["I", "JavaScript"]
```

splice (2)

remove 3 elements and replace them with the other two:

```
let arr = ["I", "study", "JavaScript", "right", "now"];
// remove 3 first elements and replace them with another
arr.splice(0, 3, "Let's", "dance");
console.log(arr) // now ["Let's", "dance", "right", "now"]
```

splice returns the array of removed elements:

```
let arr = ["I", "study", "JavaScript", "right", "now"];
// remove 2 first elements
let removed = arr.splice(0, 2);
console.log(removed); // "I", "study" <-- array of removed elements
```

insert the elements without any removals.

```
let arr = ["I", "study", "JavaScript"];
// from index 2
// delete 0
// then insert "complex" and "language"
arr.splice(2, 0, "complex", "language");
console.log(arr); // "I", "study", "complex", "language", "JavaScript"
```

concat

- returns new array that includes values from other arrays and additional items
  - accepts any number of arguments – either arrays or values.
  - result is a new array containing items from arr, then arg1, arg2 etc.
  - If an argument argN is an array, then all its elements are copied.
  - Otherwise, the argument itself is copied. arr.concat(arg1, arg2...)

```
let arr = [1, 2];

// create an array from: arr and [3,4]
alert(arr.concat([3, 4])); // 1,2,3,4

// create an array from: arr and [3,4] and [5,6]
alert(arr.concat([3, 4], [5, 6])); // 1,2,3,4,5,6

// create an array from: arr and [3,4], then add values 5 and 6
alert(arr.concat([3, 4], 5, 6)); // 1,2,3,4,5,6
```

map/filter/find/reduce are “pure” functions

- Important principle of “functional” programming

- Pure functions have no side effects
  - Do not change state information
  - Do not modify the input arguments

- Take arguments and return a new value

Iterate: forEach

- run a function for every element of the array.
  - result of the function (if it returns any) is thrown away and ignored
  - Intended for some side effect on each element of the array
    - print or alert or post to database

```
arr.forEach(function(item, index, array) {
                    // ... do something with item
});
```

- shows each element of the array

```
 // for each element call alert
["Bilbo", "Gandalf", "Nazgul"].forEach(function(element){console.log(element)} );

["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {
     console.log (`${item} is at index ${index} in ${array}`);
});
```

sort(fn)

- default sort order is ascending and converts all arguments to strings
- sorts the array in place, changing its element order.
- returns sorted array, but the returned value is usually ignored, as arr itself is modified.

```
let arr = [2, 1, 15];
// the method reorders the content of arr
arr.sort();
console.log(arr); // [1, 15, 2]
```

To use our own sorting order, we need to supply a (comparator) function as the argument of arr.sort().

```
function compareNumeric(a, b) {
    if (a > b) return 1;    //a comes after b if 1
    if (a == b) return 0;
    if (a < b) return -1;   //a comes before b if -1
}

let arr = [2, 1, 15];
arr.sort(compareNumeric);
console.log(arr);   // [1, 2, 15]
```

reduce

calculate a single value based on the array.

```
let value = arr.reduce(function(accumulator, item, index, array) {
              // ...
}, [initial]);
```

The function is applied to all array elements one after another and “carries on” its result to the next call.

accumulator – is the result of the previous function call, equals initial the first time (if initial is provided).

item – is the current array item.

index – is its position.

array – is the array.

- first argument is the “*accumulator*” that stores the combined result of all previous execution.
  - at the end it becomes the result of reduce.

JavaScript Object

An object is a collection of name/value pairs. Example: a car, which has properties like color, weight.. etc, and actions like honk(), move().. etc

One way of creating an object is by using the object-literal syntax:

```
const car = {
   color: "red",
   weight: 500,
   honk: function(){},
   move(){}
};
```

Primitive vs. Object

Primitive	Object
Passed **by value**	Passed **by reference**
Immutable	**Not immutable** by default
-	**Shallow** or **Deep** copy

```
// by value (primitives)
let a = 1;
let b = a;
a = 2;

console.log(a); // 2
console.log(b); // 1
```

```
// by reference (objects)
const a = { firstname: 'Theo' };
const b = a;
a.firstname = 'Asaad';

console.log(a.firstname); // Asaad
console.log(b.firstname); // Asaad
```

Spread operator (ES6)

The same … notation can be used to unpack iterable elements (array, string, object) rather than pack extra arguments into a function parameter.

```
alert(Math.max(3, 5, 1));          // 5

let arr = [3, 5, 1];
alert(Math.max(arr));              // NaN
Math.max(arr[0], arr[1], arr[2]) //5

let arr = [3, 5, 1];
alert(Math.max(...arr));  //5 (spread turns array into a list of arguments)
```

Spread Operator syntax – for array

```
let initialNumbers = [0, 1, 2];
let newNumber = 15;
let updatedNumbers = [...initialNumbers, newNumber];
console.log(updatedNumbers); //[0,1,2,15]

//Example Concatenate arrays
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
let arr3 = arr1.concat(arr2); // arr3: [1,2,3,4,5,6]

//Use spread syntax:
let arr4 = [...arr1, ...arr2]; // arr4: [1,2,3,4,5,6]
```

Rest and Spread Operators

```
// Spread Operator
const technologies = ['TypeScript', 'Node', 'React'];
const gainedKnowledge = ['HTML', 'CSS', ...technologies];

// Rest Operator
const [father, mother, ...children] = ['George', 'Angel', 'Mada', 'Asaad', 'Mike'];


function sum(x,y, ...more){
        let total = x + y;
        if(more.length){  // "more" is array of all extra passed params
           more.forEach( (extra)=> total += extra )
        }
        console.log(total);
}

sum(1,2,3,4,5,6); // 21
```

What is destructuring assignment?

- Special syntax that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

```
const numbers = [10, 20];
let [a, b] = numbers;
console.log(a);
console.log(b);
```

- Benefits:
  - ‘Syntactic sugar’ to replace the following:
    - `let a = numbers[0];`
    - `let b = numbers[1];`

Destructuring Arrays and Objects

```
// Destructuring Array (by order)
const [name, id, website] = ['Asaad', 123, 'miu.edu'];

// Destructuring Object (by key)
const { width, color: backgroundColor = 'white' } = { width: 300, color: 'black' }
```

Destructuring assignment

- Unwanted elements of the array can also be thrown away via an extra comma:

```
const [first, , third] = ["foo", "bar","baz","foo"];
console.log(first);
console.log(third);
```

Destructuring objects

- Destructuring on objects lets you bind variables to different properties of an object.
  - Order does not matter

```
let options = {
    title: "Menu",
    width: 100,
    height: 200
};
let {width, title, height} = options;
alert(title); // Menu
alert(width); // 100
alert(height); // 200
```

Destructure property to another name

- to assign a property to a variable with another name, set it using a colon

```
// { sourceProperty: targetVariable }
let { width: w, height: h = 10, title } = options;

// width -> w
// height -> h
// title -> title

alert(title); // Menu
alert(w); // 100
alert(h); // 200
```

Modules

Any file is considered a module, and everything inside the file is private, until we explicitly export it, we have two kinds of exports:

`export default` (can be used once) *default export*

`export` (can be used multiple times) *named export*

To import what is explicitly exported we use:

`import` defaultExport, **{**namedExport1, namedExport2**}** **`from`** './file.ts'

Creating Objects

- Object literals `{}`
- From another object **`Object.create(), Object.assign()`**
- From class **`new Course()`**

Key, Value from variable

The **key** is a string. The **`value`** could be of any type.

```
const dog = 'Theo';

const obj1 = { dog: dog };    // string key - { dog: 'Theo'}
const obj2 = { dog };         // shortcut - { dog: 'Theo'}

const obj3 = { [dog]: dog }; // dynamic key - { Theo: 'Theo'}
```

Inheritance

```
const asaad = { name: "Asaad Saad", height: 180 };

const theo = Object.create(asaad);

console.log(theo); // {}
console.log(theo.name); // Inheritance - Asaad Saad
console.log(theo.height); // Inheritance - 180

theo.name = "Theo Saad";
console.log(theo); // Own Property - { name: "Theo Saad" };
```

Check if an Object Key Exists

```
const key = 'name';
const obj1 = { name: "Asaad" };
const obj2 = Object.create(obj1);

console.log(key in obj1); // true
console.log(obj1.hasOwnProperty(key)); // true

console.log(key in obj2); // true, key is found by inheritance
console.log(obj2.hasOwnProperty(key)); // false 👍
```

Merge Objects

```
const asaad = { name: "Asaad", height: 180 };
const theo = { name: "Theo" };

const combine = Object.assign(asaad, theo); // destination, source

console.log(theo); // { name: 'Theo' }
console.log(asaad); // { name: 'Theo', height: 180 }
console.log(combine); // { name: 'Theo', height: 180 }
console.log(asaad === combine); // true
```

New Object

```
const asaad = { name: "Asaad Saad", height: 180 };

const obj1 = Object.assign({}, asaad);
const obj2 = { ...asaad };

console.log(obj1); // { name: "Asaad Saad", height: 180 }
console.log(obj2); // { name: "Asaad Saad", height: 180 }

console.log(obj1 === asaad); // false
console.log(obj2 === asaad); // false
```

Shallow Clone

```
const asaad = { name: "Asaad Saad", stats: { height: 180, weight: 70 } };
const obj1 = Object.assign({}, asaad);
const obj2 = { ...asaad };

obj1.name = 'John';
obj2.name = 'Anne';
console.log(asaad); // { name: 'Asaad Saad', stats: { height: 180, weight: 70 } }
console.log(obj1); // { name: 'John', stats: { height: 180, weight: 70 } }
console.log(obj2); // { name: 'Anne', stats: { height: 180, weight: 70 } }

obj1.stats.height = 185;
obj2.stats.weight = 75;
console.log(asaad); // { name: 'Asaad Saad', stats: { height: 185, weight: 75 } }
console.log(obj1); // { name: 'John', stats: { height: 185, weight: 75 } }
console.log(obj2); // { name: 'Anne', stats: { height: 185, weight: 75 } }
```

Deep Clone

```
const asaad = { name: "Asaad Saad", stats: { height: 180, weight: 70 } };
const obj1 = JSON.parse(JSON.stringify(asaad));
const obj2 = structuredClone(asaad);

obj1.name = 'John';
obj2.name = 'Anne';
console.log(asaad); // { name: 'Asaad Saad', stats: { height: 180, weight: 70 } }
console.log(obj1); // { name: 'John', stats: { height: 180, weight: 70 } }
console.log(obj2); // { name: 'Anne', stats: { height: 180, weight: 70 } }

obj1.stats.height = 185;
obj2.stats.weight = 75;
console.log(asaad); // { name: 'Asaad Saad', stats: { height: 180, weight: 70 } }
console.log(obj1); // { name: 'John', stats: { height: 185, weight: 70 } }
console.log(obj2); // { name: 'Anne', stats: { height: 180, weight: 75 } }
```

Keys, Values, Entries

```
const asaad = { name: "Asaad Saad", height: 180 };

const keys = Object.keys(asaad);
console.log(keys); // [ 'name', 'height' ]


const values = Object.values(asaad);

console.log(values); // [ 'Asaad Saad', 180 ]


const entries = Object.entries(asaad);
console.log(entries); // [ [ 'name', 'Asaad Saad' ], [ 'height', 180 ] ]
```

Object.freeze()

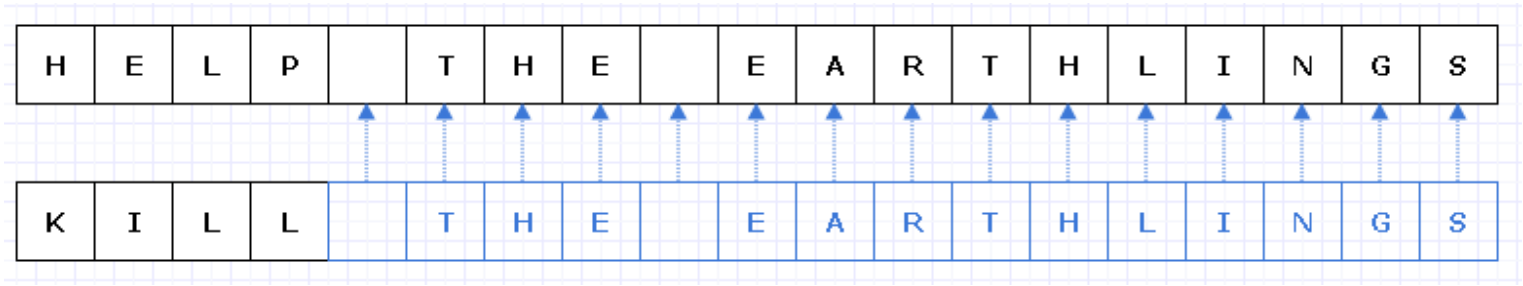
```
const asaad = { name: "Asaad Saad", height: 180 };

Object.freeze(asaad);

asaad.height = 185; // Throws an error in strict mode

console.log(asaad.height); // 180
```

Immutable Array and Object

An Immutable operation, for an array or object, will always return a new reference with the desired change and never mutate the original array or object.

Immutable Data Structure

In general, most of a data structure is not copied, but *shared*, and only the changed portions are copied. It is called a **persistent data structure**. Most implementations are able to take advantage of persistent data structures most of the time. The performance is close enough to mutable data structures that functional programmers generally consider it negligible.



Add to Immutable Array

```
const arr1 = Object.freeze(['a', 'b', 'c']);

const arr2 = arr1.concat('d');
const arr3 = [...arr1, 'd'];

console.log(arr1); // ['a', 'b', 'c']
console.log(arr2); // ['a', 'b', 'c', 'd']
console.log(arr3); // ['a', 'b', 'c', 'd']
```

Remove from Immutable Array

```
const arr1 = Object.freeze(['a', 'b', 'c', 'd', 'e']);
const arr2 = arr1.filter(item => item !== 'c');

console.log(arr1); // ['a', 'b', 'c', 'd', 'e']
console.log(arr2); // ['a', 'b', 'd', 'e']
```

Update an Immutable Array

```
const arr1 = Object.freeze(['a', 'b', 'c', 'd', 'e']);
const arr2 = arr1.map(item => item === 'c' ? 'Asaad' : item);

console.log(arr1); // ['a', 'b', 'c', 'd', 'e']
console.log(arr2); // ['a', 'b', 'Asaad', 'd', 'e']
```

Add to Immutable Object

```
const obj1 = Object.freeze({first: 'Asaad'});

const obj2 = Object.assign({}, obj1, { last: 'Saad' });
const obj3 = { ...obj1, last: 'Saad' };

console.log(obj1 ); // { first: "Asaad" }
console.log(obj2);  // { first: "Asaad", last: "Saad" }
console.log(obj3);  // { first: "Asaad", last: "Saad" }
```

Update an Immutable Object

```
const obj1 = Object.freeze({first: 'Asaad', last: 'Saad' });

const obj2 = Object.assign({}, obj1, { first : 'Theo' });
const obj3 = { ...obj1, first : 'Theo' };

console.log(obj1 ); // { first: "Asaad", last: "Saad" }
console.log(obj2);  // { first: "Theo", last: "Saad" }
console.log(obj3);  // { first: "Theo", last: "Saad" }
```