# Syntax, Functions and Statements

## What is JavaScript?

* JavaScript (**JS**) is a **high-level** programming language
  + One of the **core technologies** of the World Wide Web
  + Enables **interactive** web pages and applications
  + Can be **executed** on the **server** and on the **client**
* Features:
  + C-like **syntax** (curly-brackets, identifiers, operator)
  + **Multi-paradigm** (imperative, functional, OOP)
  + Dynamic **typing**

## Dynamic Programming Language

* JavaScript is a **dynamic programming language**
  + Operations otherwise done at **compile-time** can be done at **run-time**
* It is **possible** to change the **type** of a variable or add new properties or methods to an object **while** the program is **running**
* In **static** **programming** **languages**, such changes are normally **not** **possible**

## Node.js

* What is **Node.js**?
  + **Server-side** JavaScript runtime
  + Chrome V8 JavaScript engine
  + NPM **package manager**
  + Install node packages

## Data Types

* Seven **data types** that are **primitives**
  + **String -** used to represent textual data
  + **Number -** a numeric data type
  + **Boolean -** a logical data type
  + **Undefined** - automatically assigned to variables
  + **Null** - represents the **intentional** **absence** of any object value
  + **BigInt** -represent integers with **arbitrary** **precision**
  + **Symbol** - **unique** and **immutable** primitive value
* **Reference types – Object**

## Identifiers

* An**identifier** is a sequence of characters in the code that   
  identifies a **variable**, **function**, or **property**
* In JavaScript, identifiers are **case-sensitive** and can contain   
  Unicode **letters**, **$**, **\_**, and **digits** (0-9), but may **not** start with a   
  digit

## Variable Values

* Used to **store** data values
* Variables that are assigned a **non-primitive** value are   
  given a **reference** to that value
* **Undefined** - a variable that has been declared with a keyword, but not given a value

let a;

console.log(a) *//undefined*

* **Undeclared** - a **variable** that hasn't been declared at all

console.log(undeclaredVariable); *//ReferenceError: undeclaredVariable is not defined*

* **let**, **const** and **var** are used to declare variables
  + **let** - allows **reassignment**
  + **const** - once assigned it **cannot** be modified
  + **var** - defines a variable in the function scope **regardless** of block scope

## Legacy Variable Declaration

* You will see **var** used in old examples
* Using **var** to declare variables is a **legacy** technique
* Since **ES2015** keywords **let** and **const** are available
* **var** introduces function scope **hoisting**
  + Will be discussed later in the lesson
* There is no good reason to **ever** use **var**!

## Variable Scopes

* **Global scope –** Any variable that’s **NOT** inside any **function** or **block** (a pair of curly braces);
* **Functional scope –** Variable declared **inside a function** is inside the local scope;
* **Block scope – let** and **const** declares **block** scoped variables

## Dynamic Typing

* Variables in JavaScript are **not** directly **associated** with any particular **value** **type**
* Any variable **can** be assigned (and re-assigned) values of all types
* **NOTE: The use of dynamic typing is considered a bad practice!**

## Functions

* **Function** - named list of instructions (statements and expressions)
* Can take **parameters** and return **result**
  + Function names and parameters use **camel case**
  + The **{** stays at the same line

function printStars(count) {

    console.log("\*".repeat(count));

}

* + **Invoke** the function

printStars(10);

## Declaring Functions

* Function declaration

function walk() {

    console.log("walking");

}

* Function expression

const walk = function () {

    console.log("walking");

}

* Arrow functions

const walk = () => {

    console.log("walking");

}

## Parameters and Returned Value

* You can receive parameters with **no value**
* The **unused parameters** are ignored
* Functions can yield a value with the **return** operator

## Object Methods and Standard Library

* Any object may have **methods**
  + **Functions** that operate from the **context** of the object
  + Accessed as a **property** using the **dot-notation**

let myString = 'Hello, JavaScript!';

console.log(myString.toLowerCase()); *// hello, javascript!*

* JavaScript has a large **standard library**
  + **Math**, **Number**, **Date**, **RegExp**, **JSON** and more
  + For more information, [**visit MDN**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects)

## Default Function Parameter Values

* Functions can have **default parameter** values

function printStars(count = 5) {

    console.log("\*".repeat(count));

}

printStars(); *// \*\*\*\*\**

printStars(2); *// \*\**

printStars(3, 5, 8); *// \*\*\**

## Operators and Statements

### Arithmetic Operators

let a = 15;

let b = 5;

let c;

c = a + b; // 20

c = a - b; // 10

c = a \* b; // 75

c = a / b; // 3

c = a % b; // 0

c = a \*\* b;//155 = 759375

* **Arithmetic** **operators** - take numerical values (either   
  literals or variables) as their operands
  + Return a single numerical value
    - Addition (**+**)
    - Subtraction (**-**)
    - Multiplication (**\***)
    - Division (**/**)
    - Remainder (**%**)
    - Exponentiation (**\*\***)

### Assignment Operators

* **Assignment** **operators** - **assign** a value to its left operand based on the value of the right operand

### Comparison Operators

|  |  |
| --- | --- |
| **Operator** | **Notation in JS** |
| EQUAL value  EQUAL value and type | **==**  **===** |
| NOT EQUAL value  NOT EQUAL value or type | **!=**  **!==** |
| GREATER than | **>** |
| GREATER than OR EQUAL | **>=** |
| LESS than | **<** |
| LESS than OR EQUAL | **<=** |

### Conditional Statements

* The **if-else** statement:
  + Do action depending on condition

let a = 5;

if (a >= 5) {

console.log(a);

}

* + You can chain conditions

else {

console.log('no');

}

### Truthy and Falsy Values

* "**truthy**" - a value that **coerces** to **true** when **evaluated** in a boolean context
* The following values are "**falsy**" - **false**, **null**, **undefined**, **NaN**, **0, 0n** and **""**

logTruthiness (3.14); //Truthy!

logTruthiness ({}); //Truthy!

logTruthiness (NaN); //Falsy.

logTruthiness ("NaN"); //Truthy!

logTruthiness ([]); //Truthy!

logTruthiness (null); //Falsy.

logTruthiness (""); //Falsy.

logTruthiness (undefined); //Falsy.

logTruthiness (0); //Falsy.

### Logical Operators

* && (**logical AND**) - returns the leftmost "**false**" value or the **last truthy** value, if all are true.
* || (**logical OR**) - returns the leftmost "**true**" value or the **last falsy** value, if all are false.
* ! (**logical NOT**) - Returns **false** if its single operand can be converted to **true**; otherwise, returns **true**

const val = !true

console.log(val); *// false*

const val = !false;

console.log(val); *// true*

### Typeof Operator

* The **typeof** operator returns a string indicating the type of an operand

const val = 5;

console.log(typeof val);    // number

const str = 'hello';

console.log(typeof str);    // string

const obj = {name: 'Maria', age:18};

console.log(typeof obj);   // object

## Loops

* The **for** / **while** loops work as in C++, C# and Java
* Classical **for**-loop

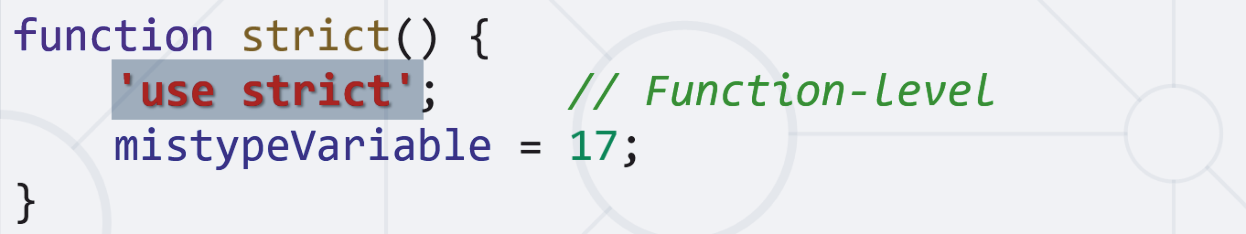
for (let i = 0; i <= 5; i++) { console.log(i); } *// 0 1 2 3 4 5*

* JavaScript supports two more variants of the **for**-loop:
  + **for-of** – used with arrays and iterators
  + **for-in** – used with objects and associative arrays
  + Both will be reviewed in **upcoming lessons**

## Debugging Techniques

* **Strict mode** limits certain "sloppy" language features
  + Silent errors will **throw** **Exception** instead





* + Enabled by default in **modules**

## Debugging in Visual Studio Code

* Visual Studio Code has a built-in **debugger**
* It provides:
  + **Breakpoints**
  + Ability to **trace** the code execution
  + Ability to **inspect** variables at runtime

## Using the Debugger in Visual Studio Code

* Start without Debugger: **[Ctrl+F5]**
* Start with Debugger: **[F5]**
* Toggle a breakpoint: **[F9]**
* Trace step by step: **[F10]**
* Force step into: **[F11]**

## Language Specifics

### First-class Functions

* First-class functions – a function can be passed as an **argument** to other functions
* Can be **returned** by another function and can be **assigned** as a value to a variable

function running() {

    return "Running";

}

function category(run, type) {

    console.log(run() + " " + type);

}

category(running, "sprint"); *//Running sprint*

## Nested Functions

* Functions can be **nested** - **hold other functions**
  + Inner functions have **access** to **variables** from **their parent**

function hypotenuse(m, n) { *// outer function*

    function square(num) {  *// inner function*

        return num \* num;

    }

    return Math.sqrt(square(m) + square(n));

}

## Hoisting

* Variable and function declarations are **put into memory** during the **compile** phase, but stay exactly where you **typed** them in your code
* **Only declarations are hoisted**

#### Hoisting Variables

console.log(num); *// Returns undefined*

var num;

num = 6;

num = 6;

console.log(num); *// returns 6*

var num;

num = 6;

console.log(num); *// ReferenceError: num is not defined*

let num;

console.log(num); *// ReferenceError: num is not defined*

num = 6;

#### Hoisting Functions

run(); // running

function run() {

    console.log("running");

};

walk(); // ReferenceError: walk is not defined

let walk = function () {

    console.log("walking");

};

console.log(walk); //undefined

walk(); // TypeError: walk is not a function

var walk = function () {

    console.log("walking");

};

# Working with Arrays of Elements

## What is an Array?

* Arrays are **list-like** **objects**
* Arrays are a **reference** **type**, the variable points to an   
  address in memory
* Elements are **numbered** from **0** to **length - 1**
* Creating an array using **an array literal**

let numbers = [10, 20, 30, 40, 50];

* Neither the **length** of a JavaScript array **nor** the **types** of its elements are **fixed**
* An array's **length** **can be changed** at any time
* Data can be stored at non-contiguous locations in the array
* JavaScript arrays are not guaranteed to be dens

## Accessing Array Elements

* Array elements are accessed using their **index**

let cars = ['BMW', 'Audi', 'Opel'];

let firstCar = cars[0];   *// BMW*

let lastCar = cars[cars.length - 1];  *// Opel*

* Accessing indexes that do not exist in the array returns **undefined**

console.log(cars[3]);   *// undefined*

console.log(cars[-1]);  *// undefined*

* Arrays can be **iterated** using **for-of** loop

for (let car of cars) { … }

## Arrays Indexation

* Setting values via **non-integers** using **bracket** **notation** (or dot notation) creates **object** **properties** instead of array elements (will be discussed in later lesson)

let arr = [];

arr[3.4] = 'Oranges';

arr[-1] = 'Apples';

console.log(arr.length);                *// 0*

console.log(arr.hasOwnProperty(3.4));   *// true*

arr["1"] = 'Grapes';

console.log(arr.length);                *// 2*

console.log(arr); *// [ <1 empty item>, 'Grapes', '3.4': 'Oranges', '-1': 'Apples' ]*

## Destructuring Syntax

* Expression that **unpacks values** from **arrays** or **objects**, into distinct **variables**

let numbers = [10, 20, 30, 40, 50];

let [a, b, ...elems] = numbers;

console.log(a) *// 10*

console.log(b) *// 20*

console.log(elems) *// [30, 40, 50]*

* The **rest** **operator** can also be used to collect function parameters into an array

## Mutator Methods

### Pop

* Removes the **last** **element** from an array and returns that element
* This method **changes** the **length** of the array

let nums = [10, 20, 30, 40, 50, 60, 70];

console.log(nums.length); *// 7*

console.log(nums.pop());  *// 70*

console.log(nums.length); *// 6*

console.log(nums);     *// [ 10, 20, 30, 40, 50, 60 ]*

### Push

* The **push()** method **adds one or more** elements to the **end** of an array and **returns** the new **length** of   
  the array

let nums = [10, 20, 30, 40, 50, 60, 70];

console.log(nums.length);  *// 7*

console.log(nums.push(80)); *// 8 (nums.length)*

console.log(nums); *// [ 10, 20, 30, 40, 50, 60, 70, 80 ]*

### Shift

* The **shift()** method **removes** the **first** **element** from an array and **returns** that **removed** **element**
* This method **changes** the **length** of the array

let nums = [10, 20, 30, 40, 50, 60, 70];

console.log(nums.length); // 7

console.log(nums.shift()); // 10 (removed element)

console.log(nums); // [ 20, 30, 40, 50, 60, 70 ]

### Unshift

* The **unshift()** method **adds** **one or more**   
  elements to the **beginning** of an array and **returns**   
  the new **length** of the array

let nums = [40, 50, 60];

console.log(nums.length);  *// 3*

console.log(nums.unshift(30));  *// 4 (nums.length)*

console.log(nums.unshift(10,20)); *// 6 (nums.length)*

console.log(nums);  *// [ 10, 20, 30, 40, 50, 60 ]*

### Splice

* Changes the contents of an array by **removing** or **replacing** existing **elements** and/or **adding** **new** elements

let nums = [1, 3, 4, 5, 6];

nums.splice(1, 0, 2); *// inserts at index 1*

console.log(nums); *// [ 1, 2, 3, 4, 5, 6 ]*

nums.splice(4, 1, 19); *// replaces 1 element at index 4*

console.log(nums); *// [ 1, 2, 3, 4, 19, 6 ]*

let el = nums.splice(2, 1); *// removes 1 element at index 2*

console.log(nums); *// [ 1, 2, 4, 19, 6 ]*

console.log(el); *// [ 3 ]*

### Fill

* Changes the contents of an array by **removing** or **replacing** existing **elements** and/or **adding** **new** elements
* Fills all the elements of an array from a **start** **index** to an **end** **index** with a **static** **value**

let arr = [1, 2, 3, 4];

*// fill with 0 from position 2 until position 4*

console.log(arr.fill(0, 2, 4)); *// [1, 2, 0, 0]*  
*// fill with 5 from position 1*

console.log(arr.fill(5, 1)); *// [1, 5, 5, 5]*  
console.log(arr.fill(6)); *// [6, 6, 6, 6]*

### Reverse

* Reverses the array
  + The **first** array **element** **becomes** the **last**, and the last array element becomes the first

let arr = [1, 2, 3, 4];

arr.reverse();

console.log(arr); *// [ 4, 3, 2, 1 ]*

## Sorting Arrays

* The **sort()** method sorts the items of an array
* Depending on the provided **compare function**, sorting can be **alphabetic** or **numeric**,  
  and either **ascending (up)** or **descending (down)**
* By default, the **sort()** function sorts the values as strings in **alphabetical** and **ascending** order
* If you want to sort numbers or other values, you need to provide the correct **compare function**!

## Compare Functions

* A **function** receiving **two parameters**, e.g. **a** and **b**
  + **Returns** either a **positive** number, a **negative** number, or **zero**
  + If **result < 0**, a is sorted **before** b
  + If **result > 0**, a is sorted **after** b
  + If **result = 0**, a and b are **equal** (no change)

let words = ['nest', 'Eggs', 'bite', 'Grip', 'jAw'];

words.sort((a, b) => a.localeCompare(b));

// ['bite', 'Eggs', 'Grip', 'jAw', 'nest']

## Accessor Methods

### Join

* + Creates and returns a **new** **string** by **concatenating** all of the elements in an array (or an array-like object), **separated** by commas or a **specified** **separator** string

let elements = ['Fire', 'Air', 'Water'];  
console.log(elements.join()); *// "Fire,Air,Water"*

console.log(elements.join('')); *// "FireAirWater"*

console.log(elements.join('-')); *// "Fire-Air-Water"*

console.log(['Fire'].join(".")); *// Fire*

### Concat

* The **concat()** method is used to **merge** two or more arrays
* This method **does** **not** **change** the **existing** **arrays**, but instead returns a new array

const num1 = [1, 2, 3];

const num2 = [4, 5, 6];

const num3 = [7, 8, 9];

const numbers = num1.concat(num2, num3);

console.log(numbers); *//  [1, 2, 3, 4, 5, 6, 7, 8, 9]*

### Slice

* The **slice()** method **returns** a shallow **copy** of a **portion** of an array into a **new** **array** object selected   
  from begin to end (end not included)
* The **original** **array** will **not** be **modified**

let fruits = ['Banana', 'Orange', 'Lemon', 'Apple', 'Mango'];

let citrus = fruits.slice(1, 3);

let fruitsCopy = fruits.slice();

*// fruits contains ['Banana', 'Orange', 'Lemon', 'Apple', 'Mango']*

*// citrus contains ['Orange', 'Lemon']*

### Includes

* Determines whether an array contains a certain element, returning **true** or **false** as appropriate

// array length is 3

// fromIndex is -100

// computed index is 3 + (-100) = -97

let arr = ['a', 'b', 'c'];

arr.includes('a', -100); // true

arr.includes('b', -100); // true

arr.includes('c', -100); // true

arr.includes('a', -2); // false

### IndexOf

* The **indexOf()** method **returns** the **first** **index** at which a given **element** can be **found** in the array
* Outputis **-1** if element is **not** **present**

const beasts = ['ant', 'bison', 'camel', 'duck', 'bison'];

console.log(beasts.indexOf('bison')); // 1  
// start from index 2

console.log(beasts.indexOf('bison', 2)); // 4  
console.log(beasts.indexOf('giraffe')); // -1

## Iteration Methods

### ForEach

* The **forEach()** method **executes a provided function**   
  once for each array element

Converting a for loop to forEach

const items = ['item1', 'item2', 'item3'];

const copy = [];

*// For loop*

for (let i = 0; i < items.length; i++) {

  copy.push(items[i]);

}

*// ForEach*

items.forEach(item => { copy.push(item); });

### Map

* **Creates a new array** with the results of calling a **provided** **function** on every element in the calling array

let numbers = [1, 4, 9];

let roots = numbers.map(function(num, i, arr) {

return Math.sqrt(num)

});

*// roots is now [1, 2, 3]*

*// numbers is still [1, 4, 9]*

### Some

* The **some()** method **tests** whether **at** **least** **one**   
  element in the array passes the test implemented by the **provided** **function**

It returns a **Boolean** value

let array = [1, 2, 3, 4, 5];

let isEven = function(element) {

  // checks whether an element is even

  return element % 2 === 0;

};

console.log(array.some(isEven)); //true

### Find

* Returns the **first** **found** **value** in the array, if an **element** in the array **satisfies** the **provided** testing **function** or **undefined** if not found

let array1 = [5, 12, 8, 130, 44];  
let found = array1.find(function(element) {

  return element > 10;

});

console.log(found); *// 12*

### Filter

* Creates a **new array** with **filtered elements** **only**
* Calls a **provided** callback **function** once for each element in an array
* **Does** **not** **mutate** the **array** on which it is called

let fruits = ['apple', 'banana', 'grapes', 'mango', 'orange'];  
 *// Filter array items based on search criteria (query)*

function filterItems(arr, query) {

  return arr.filter(function(el) {

      return el.toLowerCase().indexOf(query.toLowerCase()) !== -1;

  });

};  
console.log(filterItems(fruits, 'ap')); *// ['apple', 'grapes']*

## Reducing Arrays

### Reduce

* The **reduce()** method executes a reducer function on each element of the array, resulting in a **single** **output** **value**

const array1 = [1, 2, 3, 4];

const reducer =

(accumulator, currentValue) => accumulator + currentValue;  
console.log(array1.reduce(reducer)); *// 10*

console.log(array1.reduce(reducer, 5)); *// 15*

### **Reducer Function**

* The reducer function takes **four** arguments:
  + Accumulator
  + Current Value
  + Current Index (Optional)
  + Source Array (Optional)
* Your **reducer** **function's** returned value is **assigned** to the **accumulator**
* **Accumulator's** **value** - the **final**, **single** resulting **value**

## Array of Arrays

### Nested Arrays in JS

#### Looping Through a Nested Array

let arr = [[4, 5, 6],

           [6, 5, 4],

           [5, 5, 5]];

arr.forEach(printRow);

function printRow(row){

    console.log(row);

    row.forEach(printNumber);

}

function printNumber(num){

    console.log(num);

};

# Objects & Composition

## What is an Object?

* An object is a **collection** **of** **fields**, called **properties**
* A property is an association between a name (or **key**) and a **value**
* Objects are a **reference** **data** **type**
* In JavaScript they are created with an **object literal**:

const person = {

    firstName: "John",

    lastName: "Doe",

    age: 50

};

## Object Properties

* A **property** of an object can be explained as a **variable** that is **attached** to the object
* Object properties are the same as **ordinary** **variables**, and can hold **any data type** and be **reassigned**

|  |  |
| --- | --- |
| Property Name | Property Value |
| firstName | John |
| lastName | Doe |
| age | 50 |

## Assigning and Accessing Properties

* Simple **dot-notation**

const person = { name: 'Peter' };

console.log(person.name); // Peter

* **Bracket-notation** (indexing operator)
  + Required if the key contains a **special character**

person['job-title'] = 'Trainer';

console.log(person['job-title']) // Trainer

console.log(person.job-title) // ReferenceError

* Brackets can be used with keys as **string variables**
* Properties can be **added** during run-time

const person = { name: 'Peter' };

person.age = 21; // { name: 'Peter', age: 21 }

console.log(person.age); // 21

* **Unassigned** properties of an object are **undefined**

const person = { name: 'Peter' };

console.log(person.lastName); *// undefined*

## Destructuring Syntax

* "**Dive into**" an **object** and extract properties by name
* Can be used to get **multiple** property values

const department = {

  name: 'Engineering',

director: 'Ted Thompson',

  employeeCount: 25

};

const { name, employeeCount } = department;

console.log(name, employeeCount); // 'Engineering' 25

## Deleting Properties

const person = {

name: 'Peter',

age: 21,

['job-title']: 'Trainer'

}

*// Object {name: 'Peter', age: 21, 'job-title': 'Trainer' }*

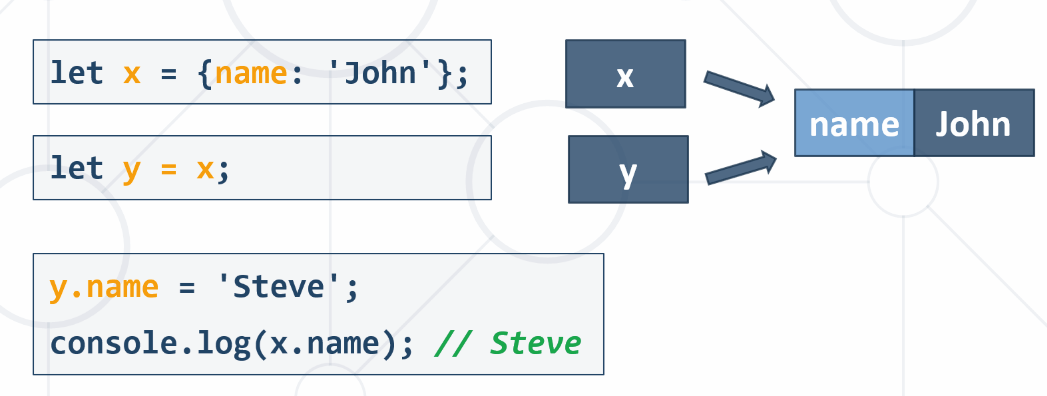
delete person.age;

*// Object {name: 'Peter', 'job-title': 'Trainer' }*

console.log(person.age) *// undefined*

# Object References

* Variables holding **reference** data types contain the **memory address** (reference) of the data
* **Copies** of the reference point to the **same data**



## Comparing Objects

* Two variables, **two distinct objects** with the same properties

const fruit = {name: 'apple'};

const fruitbear = {name: 'apple'};

fruit == fruitbear;  *// false*

fruit === fruitbear; *// false*

* Two variables, a **single object**

const fruit = { name: 'apple' };

const fruitbear = fruit;

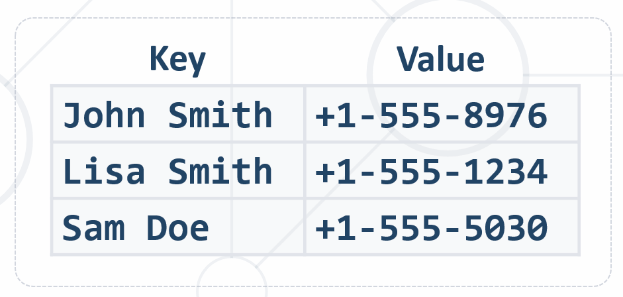
*// Assign a copy of the fruit reference to fruitbear*

fruit == fruitbear;  *// true*

fruit === fruitbear; *// true*

## Objects as Associative Arrays

* Objects can serve the role of **associative arrays** in JavaScript
  + The keys (property names) are **string indexes**
  + Values are **associated** to a key
  + All values should be of the **same type**



## For… in Loop

* **for** … **in** –iterates over all **enumerable** properties

const obj = {a: 1, b: 2, c: 3};

for (const key in obj) {

  console.log(`obj.${key} = ${obj[key]}`);

}

*// Output:*

*// "obj.a = 1"*

*// "obj.b = 2"*

*// "obj.c = 3"*

## Object Keys and Values

* Obtain an array of all keys or values in an object:

const phonebook = { 'Tim': '555-111',

'Bill': '555-333',

'Peter': '555-777' };

const keys = Object.keys(phonebook);

console.log(keys);

*// ['Tim', 'Bill', 'Peter']*

const values = Object.values(phonebook);

console.log(values);

*// ['555-111', '555-333', '555-777']*

* Get an array of **tuples** (array of two elements), representing each key and value pair
  + First tuple element is the **key**, the second is the **value**

const entries = Object.entries(phonebook);

console.log(entries);

*// [ ['Tim', '555-111'],*

*// ['Bill', '555-333'],*

*// ['Peter', '555-777'] ]*

* + This method is often used if we want to **sort** the contents

## Methods and Context

### Object Methods

* Objects can also have **methods**
* Methods are **actions** that can be performed on objects
* Methods are stored in **properties** as **function** definitions

let person = {

    firstName: "John",

    lastName: "Doe",

    age: function (myAge) {

        return `My age is ${myAge}!`    }

};

console.log(person.age(21)); *// My age is 21!*

### Objects as Function Libraries

* Related functions may be **grouped** in an object
* The object serves as a **function library**
  + Similar to built-in libraries like **Math**, **Object**, **Number**, etc.

// sorting helper

const compareNumbers = {

ascending: (a, b) => a - b;

descending: (a, b) => b - a;

};

* This technique is often used to **expose public API** in a module

### Objects as switch replacement

* You will **almost never** see **switch** used in JS code
* **Named** **methods** are used instead

|  |  |
| --- | --- |
| let count = 5;  switch (command) {  case 'increment':  count++;  break;  case 'decrement':  count--;  break;  case 'reset':  count = 0;  break;  } | let count = 5;  const parser = {  increment() { count++; },  decrement() { count--; },  reset() { count = 0; }  }  parser[command](); |

### Accessing Object Context

* Functions in JavaScript have **execution context**
  + Accessed with the keyword **this**
  + When executed as an **object method**, the context is a reference to the **parent object**

const person = {

firstName: 'Peter',

lastName: 'Johnson',

fullName() {

return this.firstName + ' ' + this.lastName;

}

};

console.log(person.fullName()); *// 'Peter Johnson'*

### **Function Execution Context**

* Execution context can be **changed** at run-time
* If a function is **executed outside** of its parent object, it will **no longer** have access to the object's content

const getFullName = person.fullName;

console.log(getFullName()); *// 'undefined undefined'*

const anotherPerson = { firstName: 'Bob',

lastName: 'Smith' };

anotherPerson.fullName = getFullName;

console.log(anotherPerson.fullName()); *// 'Bob Smith'*

* **Further lessons** will explore more **context** **features**!

## Object Composition

### What is Object Composition?

* **Combining** simple objects into more **complex** **ones**

let student = {

  firstName: 'Maria',

  lastName: 'Lopez',

  age: 22,

  location: { lat: 42.698, lng: 23.322 }

}

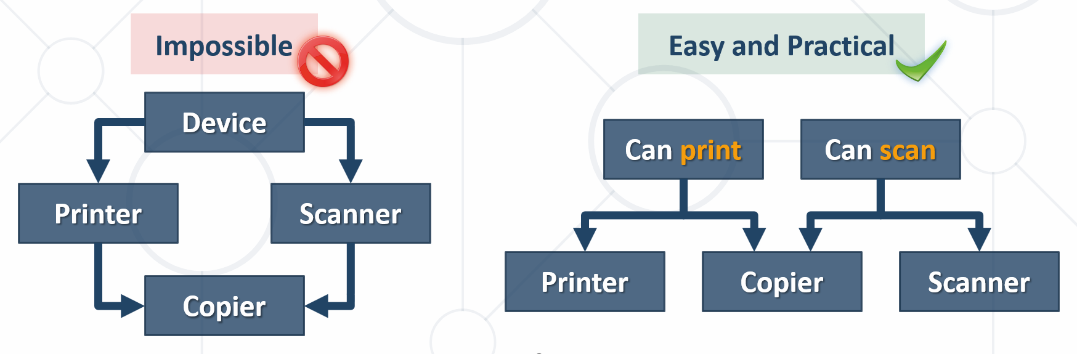
console.log(student);

console.log(student.location.lat);

* **Composition** is a powerful technique for **code** **reuse**
* **It can be considered superior to OOP inheritance**

### Composing Objects with Behavior

* We can **compose** **behavior** at run-time and **reuse** **functionality**
* **Describe** **objects** in terms of what they **do**, not what they **are**
* This solves a deeply rooted **problem** with **OOP inheritance**



function print() {

console.log(`${this.name} is printing a page`);

}

function scan() {

console.log(`${this.name} is scanning a document`);

}

const printer = { name: 'ACME Printer', print };

const scanner = { name: 'Initech Scanner', scan };

const copier = { name: 'ComTron Copier', print, scan };

### Factory Functions With Reference

* Functions that **compose** and **return** objects

function createRect(width, height) {

const rect = { width, height };

rect.getArea = () => {

return rect.width \* rect.height;

};

return rect;

}

* Creating methods with object references can **avoid** the pitfalls of using **this**

### Decorator Functions

* Functions that **add** new **data** and **behavior** to objects

function canPrint(device) {

device.print = () => {

console.log(`${device.name} is printing a page`);

}

}

const printer = { name: 'ACME Printer' };

canPrint(printer);

printer.print();

*// ACME Printer is printing a page*

* The object reference is **embedded** – using **this** is not required

## JSON

### JavaScript Object Notation

* It's a **data** interchange **format**
* It's **language independent** - syntax is like JavaScript object syntax, but the JSON format is text only
* Is **"self-describing"** and easy to understand:

{

    "employees": [

        { "firstName": "John", "lastName": "Doe" },

        { "firstName": "Anna", "lastName": "Smith" },

        { "firstName": "Peter", "lastName": "Jones" }

    ]

}

### Syntax Rules

* In JSON:
  + Data is in **name/value** pairs
  + Data is **separated by commas**
  + **Curly braces** hold **objects**
  + **Square brackets** hold **arrays**
  + **JSON only takes double quotes ""**

{

  "employees": [{ "firstName": "John", "lastName": "Doe" }]

}

### Converting to String

* Use **JSON.stringify()** to convert objects into a string:

let obj = { name: "John", age: 30, city: "New York" };

let myJSON = JSON.stringify(obj);

console.log(myJSON);*// {"name":"John","age":30,"city":"New York"}*

* You can do the same for **arrays**

let arr = [ "John", "Peter", "Sally", "Jane" ];

let myJSON = JSON.stringify(arr);

console.log(myJSON); *// ["John","Peter","Sally","Jane"]*

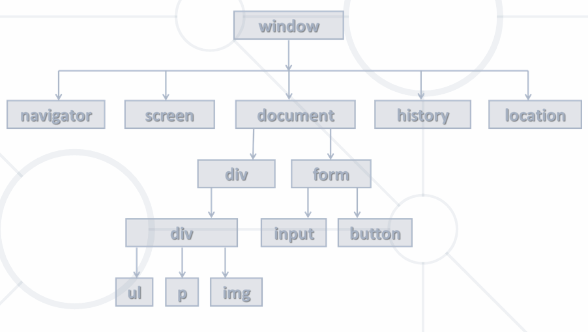
* **Format** the string with **indentation** for readability

let myJSON = JSON.stringify(arr, null, 2);

# DOM Introduction

## Browser API

### Browser Object Model

* ****Browsers expose some objects like **window**, **screen**, **navigator**, **history**, **location**, **document**, …

console.dir(window);

console.dir(navigator);

console.dir(screen);

console.dir(location);

console.dir(history);

console.dir(document);

* Most of this **API** will be examined in the **next course**

### Global Context in the Browser

* The **global object** in the browser is **window**

let b = 8;

console.log(this.b); *// undefined*

var a = 5;

console.log(this.a); // 5

function foo() {

console.log("Simple function call");

console.log(this === window);  // true

}  
foo();

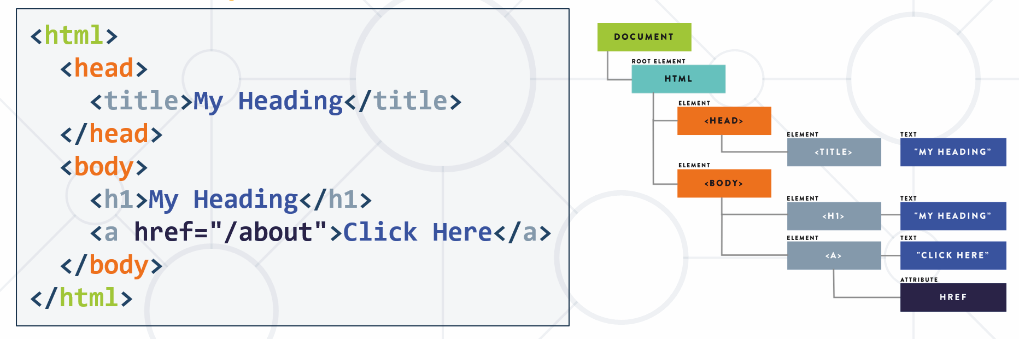
## Document Object Model (DOM)

### Document Object Model (DOM)

* The **DOM** represents the document as **nodes** and **objects**
  + That way, the programming languages **can connect** to the page
* The **HTML DOM** is an **Object Model** for **HTML**. It defines:
  + HTML elements as **objects**
  + **Properties**
  + **Methods**
  + **Events**

### From HTML to DOM Tree

* The browser **parses** HTML and creates a **DOM Tree**



* The elements are **nested** in each other and create a **hierarchy**
  + Like the hierarchy of a **street address** – Country, City, Street, etc.

### DOM Methods

* **DOM** **Methods** - **actions** you can perform on HTML elements
* **DOM** **Properties** - values of HTML elements that you can ss**set** or **change**
* HTML DOM **method** is an action you can do (like **add** or **delete** an HTML element)

### Using the DOM API

* JavaScript can **interact** with web pages via the **DOM API**:
  + Check the **contents** and **structure** of elements on the page
  + Modify element **style** and **properties**
  + Read **user input** and react to **events**
  + **Create** and **remove** elements
* Most actions are performed when an **event** occurs
  + Events are **"fired"** when something of interest happens
* All of this **and more** will be examined in upcoming lessons

### JavaScript in the Browser

* Code can be **executed in the page** in different ways:
  + Directly in the **developer console** – when **debugging**
  + As a page **event handler** – e.g., user **clicks** on a button
  + Via **inline** script, using **<script>** tags
  + By **importing** from external file – most **flexible method**

## HTML Elements

### Elements and Properties

* The DOM Tree is comprised of **HTML elements**
* Elements are **JS objects** with **properties** and **methods**
  + They can be **accessed** and **modified** like regular objects
* To change the contents of the page:
  + **Select** an element to obtain a **reference**
  + **Modify** its **properties**

### Attributes and Properties

* Attributes are defined by **HTML**
  + Attributes **initialize** DOM properties
  + **Property** values can **change** via the DOM API
* The HTML **attribute** and the DOM **property** are technically **not** **the** **same** **thing**
* Since the **outcome is the same**, in practice you will **almost never** encounter a difference!

### DOM Manipulations

* The **HTML DOM** allows JavaScript to change the   
  content of **HTML** **elements**
  + **innerHTML**
  + **textContent**
  + **value**
  + **style**
  + And many others to be discussed in upcoming lessons

### Accessing Element HTML

* To access raw HTML:

element.innerHTML = "<p>Welcome to the DOM</p>";

* This will be **parsed** – beware of **XSS attacks**!
* Changing **textContent** or **innerHTML** removes all child nodes

### Accessing Element Text

* The contents of HTML elements are stored in text nodes
  + To access the contents of an element:

let text = element.textContent; *//This is JavaScript!*

element.textContent = "Welcome to the DOM";

* + If the element has children, returns all text **concatenated**

### Accessing Element Values

* The **values** of input elements are **string** **properties** on them:

let num = Number(element.value);

element.value = 56;

## Targeting DOM Elements

### Targeting Elements

* There are a few ways to **find** a certain **HTML** **element** in the **DOM**:
  + By ID - **getElementById()**
  + By class name - **getElementsByClassName()**
  + By tag name - **getElementsByTagName()**
  + By CSS selector - **querySelector(), querySelectorAll()**
* These methods return a **reference** to the element, which can be **manipulated** with JavaScript

### Targeting by Tag and Class Names – Example

* The **tag name** specifies the **type** of element – **div**, **p**, **ul**, etc.

const elements = document.getElementsByTagName('p');

*// Select all paragraphs on the page*

* **Class names** are used for **styling** and easier **selection**

const elements = document.getElementsByClassName('list');

*// Select all elements having a class named 'list'*

* Both methods return a live **HTMLCollection**
  + **Even if** only **one** element is selected! This is a **common mistake**

### CSS Selectors

* **CSS selectors** are strings that follow CSS syntax for matching
* They allow very fast and powerful element matching, e.g.:
  + **"#main"** - returns the element with ID "main"
  + **"#content div"** - selects all **<div>**s inside **#content**
  + **".note, .alert"** - all elements with class "note" or "alert"
  + **"input[name='login']"** - **<input>** with name "login"
* Select the **first matching** element

const mainDiv = document.querySelector('#main'); *// Select the element with ID 'main'*

const element = document.querySelector('p'); *// Select the first paragraph on the page*

* Select **all** matching elements
  + Returns a **static NodeList**

const elements = document.querySelectorAll('article.list');

*// Select all <article> elements having a class named 'list'*

### NodeList vs. HTMLCollection

* Both interfaces are **collections** of **DOM nodes**
* **NodeList** can contain **any** node type, including **text** and **whitespace**
* **HTMLCollection** contains only **Element** **nodes**
* Both have **iteration** methods, **HTMLCollection** has an extra **namedItem** method
* **HTMLCollection** is **live**, while **NodeList** can be either **live** or **static**

### Iterating Element Collections

* **NodeList** and **HTMLCollection** are **NOT** arrays but can be **indexed** and **iterated**

const elements = document.querySelectorAll('p');

const first = elements[0];

*// Select the first paragraph on the page*

for (let p of elements) { */\* … \*/* }

*// Iterate over all entries*

* Both can be **explicitly converted** to an array

const elementArray = Array.from(elements);

const elementArr2 = [...elements]; *// Spread syntax*

### Parents and Child Elements

* Every DOM Element has a **parent**
  + Parents can be accessed by property **parentElement** or **parentNode**

Graphical user interface, text

Description automatically generated

let firstP = document.getElementsByTagName('p')[0];

console.log(firstP.parentElement);

Icon

Description automatically generated

## Using the DOM API

### External Page Scripts

* Page scripts can be **loaded** from an external file
  + Use the **src** attribute of the **script element**

<script src="app.js"></script>

* **Functions** from script files are in the **global scope**
  + Can be referenced and **executed** from **events** and **inline** scripts
  + **Multiple** script files in a page can see **each other**
* Pay attention to **load order**!

### Control Content via Visibility

* Content can be **hidden** or **revealed** by changing its **display** style
  + This is a **common technique** to display content dynamically
* To **hide** an element:

const element = document.getElementById('main');

element.style.display = 'none';

* To **reveal** an element, set **display** to anything that isn't **'none'** (including **empty string**)

element.style.display = ''; // Can be 'inline', 'block', etc.

### Match n-th Child

* Sometimes we need to target an element based on its **relation** to other **similar elements**
  + E.g., **row** or **column** in a table, **list item**, etc.
* Can be done either by index or with a CSS selector

const list = document.getElementsByTagName('ul')[0]; *// First <ul> on the page*

const thirdLi = list.getElementsByTagName('li')[2]; *// Third <li> inside the selected <ul>*

const thirdLi = document.querySelector('ul li:nth-child(3)'); *// Third <li> inside the first <ul> on the page*

# DOM Events

## DOM Manipulation

### DOM Manipulations

* We can **create**, **append** and **remove** HTML elements dynamically
  + **appendChild()**
  + **removeChild()**
  + **replaceChild()**

### Creating New DOM Elements

* HTML elements are created with **document.createElement**
  + This is called a **Factory Pattern**
* Variables holding HTML elements are **live**:
  + If you **modify** the contents of the variable, the DOM is **updated**
  + If you **insert** it somewhere in the DOM, the original is **moved**
* Text added to **textContent** will be **escaped**
* Text added to **innerHTML** will be **parsed** and turned into actual HTML elements 🡪 beware of **XSS attacks**!

### Manipulating Node Hierarchy

* **appendChild -** Adds a new child, as the **last child**

let p = document.createElement("p");

let li = document.createElement("li");

li.appendChild(p);

* **prepend -** Adds a new child, as the **first child**

let ul = document.getElementById("my-list");

let li = document.createElement("li");

ul.prepend(li);

### Deleting DOM Elements

Sunburst chart

Description automatically generated with medium confidence

A picture containing timeline

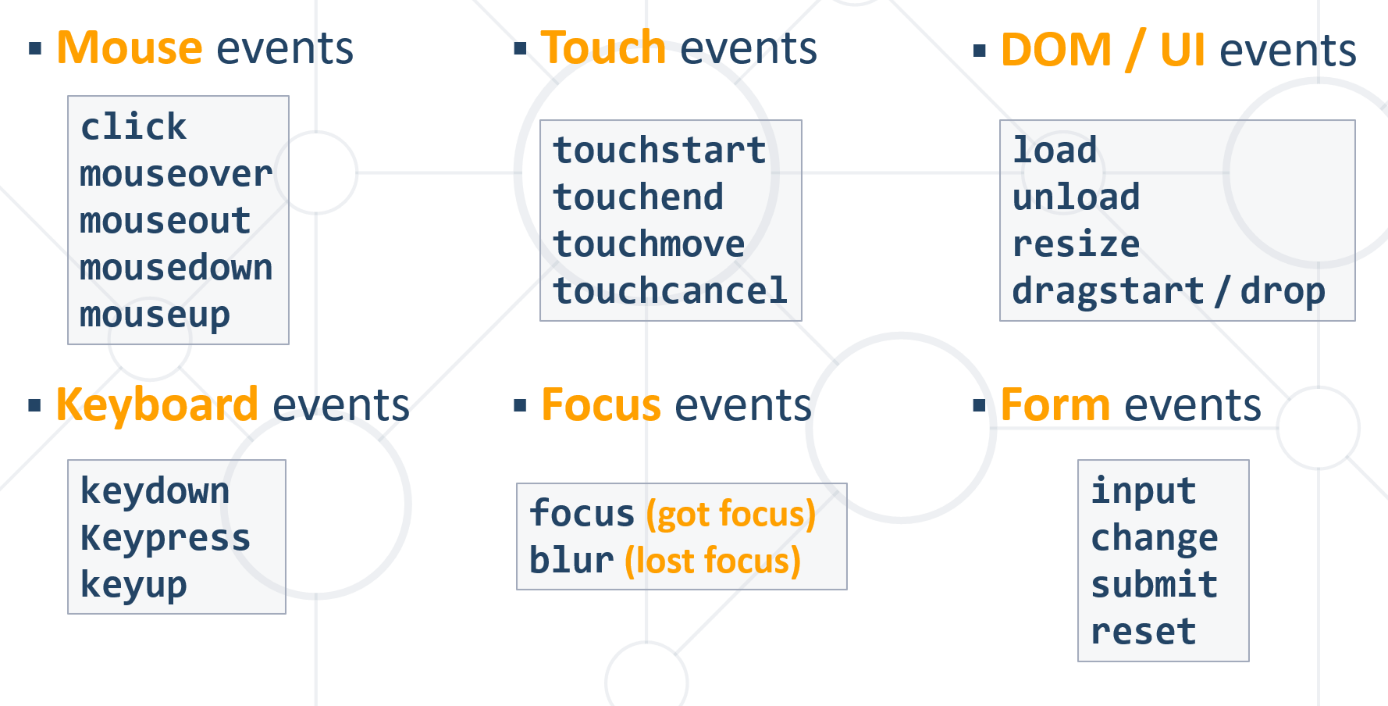
Description automatically generated

## The DOM Event

### Event Object

* Calls its **associated** **function**
* Passes a **single** **argument** to the function -   
  a **reference** to the event object
* Contains **properties** that describe the event
  + Which **element** triggered the event
  + Screen **coordinates** where it occurred
  + What is the **type** of the event
  + And more

### Event Types in DOM API



### Event Handling

* Event registration is done by providing a **callback function**
* Three ways to register for an event:
  + With **HTML Attributes**
  + Using **DOM element properties**
  + Using **DOM** **event** **handler** – preferred method

function handler(event){  
    // this --> object, html reference

    // event --> object, event configuration  
}

### Event Listener

* addEventListener();

htmlRef.addEventListener( 'click' , handler , false );

* removeEventListener();

htmlRef.removeEventListener( 'click' , handler);

### Attaching Click Handler

const button = document.getElementsByTagName('button')[0];

button.addEventListener('click', clickMe);

function clickMe(e) {

  const target = e.currentTarget;

  const targetText = target.textContent;

  target.textContent = Number(targetText) + 1;

}

### Events Handler Execution Context

* In event handlers, **this** refers to the event **source** **element**

element.addEventListener("click", function(e) {

console.log(this === e.currentTarget); *// Always true*

});

* Pay attention when using **object methods** as event listeners!
  + **this** may not behave as you expect with objects

### Attaching Hover Handler

const button = document.getElementsByTagName('div')[0];

button.addEventListener('mouseover', function (e) {

    const style = e.currentTarget;

    const { backgroundColor } = style;

    if(backgroundColor === 'white'){

        targetStyles.backgroundColor = '#234465';

        targetStyles.color = 'white';

    } else {

        targetStyles.backgroundColor = 'white';

        targetStyles.color = '#234465';

    }}

);

### Attaching Input Handler

const inputField = document.getElementsByTagName('input')[0];

const button = document.getElementsByTagName('button')[0];

inputField.addEventListener('input', function () {

    button.setAttribute('disabled', 'false')

});

### Remove Listeners

const password = document.querySelector('input[type="password"]');

const button = document.querySelector('button');

password.addEventListener('focus', focusEvent);

function focusEvent (){

    event.target.style.background = '#234465';

}

password.addEventListener('blur', (event) => {

    event.target.style.background = '';

});

button.addEventListener('click', () => {

    password.removeEventListener('focus', focusEvent);

});

### Multiple Listeners

* The **addEventListener()** method also allows you to add many listeners to the same element, without overwriting existing ones:

element.addEventListener("click", myFirstFunction);

element.addEventListener("click", mySecondFunction);

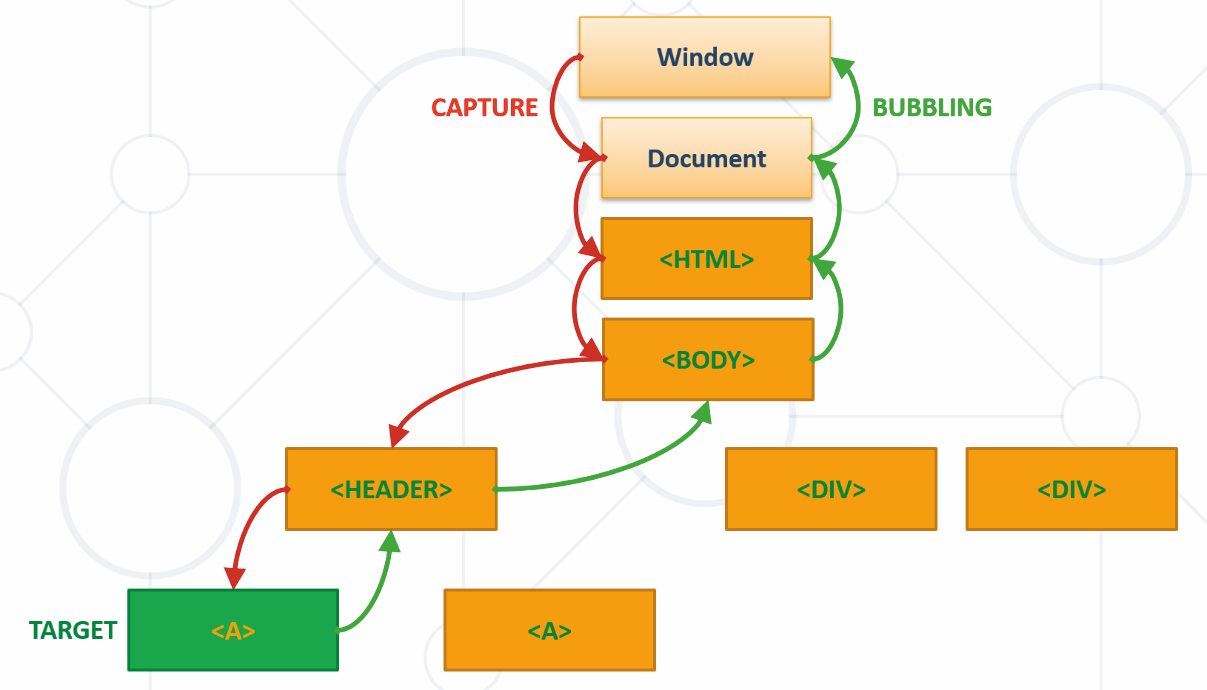
element.addEventListener("mouseover", myThirdFunction);

element.addEventListener("mouseout", myFourthFunction);

*Note that you don't use the "****on****" prefix for the event  
use "****click****" instead of "****onclick****"*

## Event Propagation

### Event Propagation

****

### DOM Event Delegation

* Allows you to **avoid** adding event listeners to specific nodes
* Event listener is assigned to a **single ancestor**

<ul id="parent-list">

    <li id="post-1">Item 1</li>

    <li id="post-2">Item 2</li>

</ul>

document.getElementById("parent-list")

.addEventListener("click", function(e) {

    if(e.target && e.target.nodeName == "LI") {

        console.log(

"List item ", e.target.id.replace("post-", ""),

" was clicked!");

    }

});

### Pros and Cons

* **Benefits**
  + Simplifies initialization
  + Saves memory
  + Less code
* **Limitations**
  + Event must be bubbling
  + May add CPU load

### Controlling Propagation and Behavior

* **stopPropagation** prevents further propagation of the event
  + If there are **multiple handlers** for the same event
* **preventDefault** stop the browser from executing default behavior, for example:
  + **Navigating** to a new page when <a> is clicked
  + Submitting **HTTP** **requests** via forms
  + Opening **context** **menus**

# Advanced Functions

## Execution Context

### Execution Context Review

* The **function context** is the object that **owns** the currently executed code
* Function context === **this** object
* Depends on how the function is invoked
  + Global invoke: **func()**
  + Object method: **object.function()**
  + DOM Event: **element.addEventListener()**
  + Using call() / apply() / bind()

### Inner Method Context

* **this** variable is **accessible** only by the **outer** **method**

const obj = {

name: 'Peter',

 outer() {

console.log(this); *// Object {name: "Peter"}*

function inner() { console.log(this); }

inner();

}

}

obj.outer(); *// Window*

### Arrow Function Context

* **this** retains the value of the **enclosing lexical context**

const obj = {

name: 'Peter',

 outer() {

console.log(this); *// Object {name: "Peter"}*

const inner = () => console.log(this);

inner();

}

}

obj.outer(); *// Object {name: "Peter"}*

### Explicit Binding

* Occurs when **call()**, **apply()**, or **bind()** are used on a function
* **Forces** a **function** call to **use** a particular **object** for this binding

function greet() {

console.log(this.name);

}

let person = { name:'Alex' };

greet.call(person, arg1, arg2, arg3, ...); *// Alex*

### Changing the Context: Call

* Calls a function with a given **this** value and **arguments** provided individually

const sharePersonalInfo = function (...activities) {

let info = `Hello, my name is ${this.name} and`+

`I'm a ${this.profession}.\n`;

info += activities.reduce((acc, curr) => {

let el  = `--- ${curr}\n`;

return acc + el;

}, "My hobbies are:\n").trim();  
return info;

}  
*const firstPerson = { name: "Peter", profession: "Fisherman" };*

*console.log(sharePersonalInfo.call(firstPerson, 'biking', 'swimming','football'));*

*// Hello, my name is Peter.*

*// I'm a Fisherman.*

*// My hobbies are:*

*// --- biking*

*// --- swimming*

*// --- football*

### Changing the Context: Apply

* Calls a function with a **given** **this** **value**, and **arguments** provided as an **array**
* **apply()** accepts a **single** **array** of arguments, while **call()** accepts an **argument** **list**
* If the first argument is **undefined** or **null** a similar outcome can be achieved using the array **spread** **syntax**

const firstPerson = {

name: "Peter",

prof: "Fisherman",

shareInfo: function () {

console.log(`${this.name} works as a ${this.prof}`);

}

};

const secondPerson = { name: "George", prof: "Manager" };

firstPerson.shareInfo.apply(secondPerson);

*// George works as a Manager*

### Changing the Context: Bind

* The **bind()** method creates a **new** **function**
* Has its **this** keyword **set** to the **provided** value, with a given sequence of arguments preceding any provided when the **new** **function** is called
* Calling the bound function generally results in the **execution** of its **wrapped** **function**

const x = 42;

const getX = function () {

  return this.x;

}

const module = {x , getX };

const unboundGetX = module.getX;

console.log(unboundGetX()); *// undefined*

const boundGetX = unboundGetX.bind(module);

console.log(boundGetX()); *// 42*

### Object Methods as Browser Event Handlers

const person = {

name: "Peter",

respond() {

alert(`${this.name} says hello!`);

}

}

const boundRespond = person.respond.bind(person);

documet.getElementById('callBtn').addEventListener('click', person.respond); //Unwanted result

documet.getElementById('callBtn').addEventListener('click', boundRespond); //Work as intended

## Functional Programming in JS

### First-Class Functions

* **First-class functions** are treated like any other variable
  + Passed as an **argument**
  + **Returned** by another function
  + Assigned as a **value** to a **variable**

The term "first-class" means that something is just a value. A first-class function is one that can go anywhere that any other value can go - there are few to no restrictions.

Michael Fogus, Functional Javascript

* Can be passed as an **argument** to another function

function sayHello() {

return "Hello, ";

}

function greeting(helloMessage, name) {

return helloMessage() + name;

}

console.log(greeting(sayHello, "JavaScript!")); *// Hello, JavaScript!*

* Can be **returned** by another function
  + We can do that, because we treated functions in JavaScript as a **value**

function sayHello() {

return function () {

console.log('Hello!');

}

}

* Can be assigned as a **value** to a **variable**

const write = function () {

return "Hello, world!";

}

console.log(write());*// Hello, world!*

### Higher-Order Functions

* Take other **functions** as an **argument** or **return a function** as a result

const sayHello = function () {

return function () {

console.log("Hello!");

}

}

const myFunc = sayHello();

myFunc(); *// Hello!*

### Predicates

* Any function that returns a **bool** **based** on evaluation of the **truth** of an **assertion**
* Predicates are often found in the form of **callbacks**

let found = array1.find(isFound);

function isFound(element) {

    return element > 10; *//True or false*

}

console.log(found); *// 12*

### Built-in Higher Order Functions

* **Array.prototype.map**
* **Array.prototype.filter**
* **Array.prototype.reduce**

users = [ { name: 'Tim', age: 25 },

{ name: 'Sam', age: 30 },

{ name: 'Bill', age: 20 } ];

getName = (user) => user.name;

usernames = users.map(getName);

console.log(usernames) *// ["Tim", "Sam", "Bill"]*

### Pure Functions

* Returns the **same result** given **same parameters**
* Execution is **independent** of the state of the system

// impure function:

let number = 1;

const increment = () => number += 1;

increment(); // 2

// pure function:

const increment = n => n + 1;

increment(1); // 2

### Referential Transparency

* An **expression** that can be **replaced** with its corresponding **value** without **changing** the program's behavior
* Expression is **pure** and its evaluation must have no **side effects**

function add(a, b) { return a + b };

function mult(a, b) { return a \* b};

let x = add(2, mult(3, 4)); *// mult(3, 4)) can be replaced with 12*

## Closure

### Closure

* One of the most **important** **features** in JavaScript
* The **scope** of an inner function **includes** the scope of the outer function
* An **inner** function retains **variables** being used from the **outer** function scope even after the parent function has **returned**

### Functions Returning Functions

* A **state** is preserved in the outer function (**closure**)

|  |  |
| --- | --- |
| const f = (function () {    let counter = 0;    return function () {  console.log(++counter);    }  })(); | f(); *// 1*  f(); *// 2*  f(); *// 3*  f(); *// 4*  f(); *// 5*  f(); *// 6*  f(); *// 7* |

### What is IIFE?

* **Immediately-Invoked Function Expressions (IIFE)**
  + Define **anonymous** function expression
  + Invoke it **immediately** after declaration

(function () { let name = "Peter"; })();

*// Variable name is not accessible from the outside scope*

console.log(name); *// ReferenceError*

let result = (function () {

    let name = "Peter";

    return name;

})();

*// Immediately creates the output:*

console.log(result); *// Peter*

## Function Decoration

### Partial Application

* **Set** some of the **arguments** of a function, **without executing** it
* Pass the **remaining arguments** when a result is needed
  + The partially applied function can be **used multiple times**
  + It will **retain** all fixed arguments, **regardless of context**

*f =* (x, y) => x + y => *g =* (x) => *f*(1, x)

Math.pow(x,y) => sqr = (x) => Math.pow(x,2)

### Currying

* Currying is a technique for **function** **decomposition**

function sum3(a) {

return (b) => {

return (c) => {

return a + b + c;

}

}

}

console.log(sum3(5)(6)(8)); *// 19*

* Supply arguments **one at a time**, instead of at once
  + They may come from **different sources**
  + Execution can be delayed until it's needed

### Currying Usage

* **Function Composition** - Building new function from old function by passing arguments
* **Memoization** - Functions that are called repeatedly with the same set of inputs but whose result is relatively expensive to produce
* **Handle** **Errors** - Throwing functions and exiting immediately after an error

### Currying vs Partial Application

* **Currying** alwaysproduces nested unary functions
* **Partial** application produces functions of arbitrary number of arguments
* Currying is **NOT** partial application It can be implemented using partial application

# Unit Testing and Error Handling

## Error Handling

### Error Handling

* The fundamental **principle** of error handling says that a function (method) should either:
  + Do what its **name** suggests
  + Indicate a **problem**
  + Any other behavior is incorrect
* A function failed to do what its name suggests should:
  + Return a special value (e.g. **undefined** / **false** / **-1**)
  + Throw an **exception** / **error**
  + Exceptions indicate **abnormal** execution **circumstances**

let str = "Hello, SoftUni";

console.log(str.indexOf("Sofia")); *// -1*

*// Special case returns a special value to indicate "not found"*

### Types of Errors

* There are **three types** of errors in programming:
  + **Syntax Errors** - during parsing
  + **Runtime Errors** - occur during execution
    - After compilation, when the application is running
  + **Logical Errors** - occur when a mistake has been made in the  
    logic of the script and the expected result is incorrect
    - Also known as bugs

### Error Handling – Exceptions (Errors)

let arr = new Array(-1);    *// RangeError*

let bigArr = new Array(9999999999); *// RangeError*

let index = undefined.indexOf("hi"); *// TypeError*

console.log(George);    *// ReferenceError*

console.print('hi');    *// TypeError*

### Error Handling – Special Values

let sqrt = Math.sqrt(-1); *// NaN (special value)*

let sub = "hello".substring(2, 1000); *// llo*

let sub = "hello".substring(-100, 100); *// hello*

*// Error avoidance – invalid ranges are adjusted*

let invalid = new Date("Christmas"); *// Invalid Date*

let date = invalid.getDate(); *// NaN*

### Throwing Errors (Exceptions)

* The **throw** statement lets you create custom errors
* **General** **Error**

throw new Error('Invalid state');

* **Range Error**

throw new RangeError("Invalid index")

* **Type Error**

throw new TypeError("String expected")

* **Reference Error**

throw new ReferenceError("Missing age")

### Try – Catch

* The **try** statement tests a block of code for **errors**
* The **catch** statement **handles** the error
* **Try** and **catch** come in pairs

try {  
  // Code that can throw an exception  
  // Some other code - not executed in case of error!

} catch (ex) {  
  // This code is executed in case of exception  
  // Ex holds the info about the exception  
}

### Exception Properties

* An **Error object** with properties is created

try {

    throw new RangeError("Invalid range.");

    console.log("This will not be executed.");

  } catch (ex) {

    console.log("Exception object: " + ex);

    console.log("Type: " + ex.name);

    console.log("Message: " + ex.message);

    console.log("Stack: " + ex.stack);  
}

## Unit Testing

### Unit Testing

* A **unit test** is a piece of code that checks whether certain functionality **works as expected**
* Allows developers to see **where** & **why** **errors** **occur**

function sortNums(arr) {  
 arr.sort((a,b) => a - b);  
}

let nums = [2, 15, -2, 4];  
sortNums(nums);

if (JSON.stringify(nums) === "[-2,2,4,15]") {

console.error("They are equal!");

}

* Testing enables the following:
  + **Easier** **maintenance** of the code base
    - Bugs are found ASAP
  + **Faster development**
    - The so called "Test-driven development"
    - Tests before code
  + **Automated way to find code wrongness**
    - If most of the features have tests, running them shows their correctness

### Unit Tests Structure

* The **AAA** Pattern: **Arrange**, **Act**, **Assert**

// Arrange all necessary preconditions and inputs

let nums = [2, 15, -2, 4];

// Act on the object or method under test

sortNums(nums);

// Assert that the obtained results are what we expect

if (JSON.stringify(nums) === "[-2,2,4,15]") {

    console.error("They are equal!");

}

### Unit Testing Frameworks

* JS Unit Testing:
  + [**Mocha**](https://mochajs.org/), [**QUnit**](https://qunitjs.com/), [**Unit.js**](http://unitjs.com/), [**Jasmine**](http://jasmine.github.io/) , **Jest (All in one)**
* Assertion frameworks (perform checks):
  + [**Chai**](http://chaijs.com/), [**Assert.js**](http://angular.github.io/assert/), [**Should.js**](https://github.com/shouldjs/should.js)
* Mocking frameworks (mocks and stubs):
  + [**Sinon**](http://sinonjs.org/), [**JMock**](http://www.jmock.org/), [**Mockito**](http://mockito.org/), [**Moq**](https://github.com/moq/moq4)

## Modules

### Modules

* + A **set of functions** to be included in applications
  + Group related behavior
  + Resolve naming collisions
    - **http.get(**url**)** and **students.get()**
  + Expose only public behavior
    - They do not populate the global scope with unnecessary objects

const loading = {

show() { },  
hide() { },

};

### Node.js Modules

* **require()** is used to **import** modules

const http = require('http'); *// For NPM packages*

const myModule = require('./myModule.js'); *// For internal modules*

* **Internal** modules need to be **exported** **before** being required
* In **Node.js** each file has its own scope
* Whatever value has **module.exports** will be the value when using **require**

const myModule = () => {...};

module.exports = myModule;

* To **export more than one** function, the value of **module.exports** will be an **object**

module.exports = {  
 toCamelCase: convertToCamelCase,  
 toLowerCase: convertToLowerCase  
};

## Unit Testing with Mocha and Chai

### What is Mocha?

* Feature-rich JS test framework
* Provides common testing functions including **it**, **describe** and the **main function** that runs tests

describe("title", function () {

  it("title", function () { … });

});

* Usually used together with **Chai**

### What is Chai?

* A library with many assertions
* Allows the usage of a lot of different assertions such as **assert.equal**

let assert = require("chai").assert;

describe("pow", function() {

it("2 raised to power 3 is 8", function() {

   assert.equal(pow(2, 3), 8);

});

});

### Installation

* To install **frameworks** and **libraries**, use the CMD
  + Installing **Mocha** and **Chai** through **npm**

npm init -y

npm install chai

npm install mocha

### Usage and Examples

* To load a library, we need to **require** it

const expect = require("chai").expect;

describe("Test group #1", function () {

    it("should… when…", function () {

        expect(actual).to.be.equal(expected);

    });

    it("should… when…", function () { … });

});

describe("Test group #2", function () {

    it("should… when…", function () {

        expect(actual).to.be.equal(expected);

    });

});

### Unit Testing Approaches

* "**Code First**" (code and test) approach
  + Classical approach
* "**Test First**" approach
  + **T**est-**d**riven **d**evelopment (**TDD**)

#### The Code and Test Approach

1. Write code
2. Write unit test
3. Run and succeed

#### The Test-Driven Development Approach

1. Write test
2. Run test and fail
3. Write code to pass test
4. Remove duplication

### Why TDD?

* TDD helps find design issues early
  + Avoids reworking
* Writing code to satisfy a test is a focused activity
  + Less chance of error
* Tests will be more comprehensive than if they are written after the code

# JavaScript Classes

## Defining Classes

### Class Definition

* **Structure** for objects
* Classes define:
  + **Data** (properties, attributes)
  + **Actions** (behavior)
* One class may have **many instances** (objects)
* Unlike functions, class declarations are **not hoisted**!

### Class Body

* The class body contains **method definitions**

class Circle {

constructor(r) {

this.r = r;

}

};

* The **constructor** is a special method for **creating** and **initializing** an object created with a class
* Instance **properties** are defined inside the **constructor**

### Class Methods

* A class may have **methods**, which will be available to its **instances**

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

*// Method*

calcArea() { return this.height \* this.width; }

}

const square = new Rectangle(10, 10);

console.log(square.calcArea()); *// 100*

### Instance Context

* **this** refers to the **instance** of the class

class Person {

constructor(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

displayName() {

console.log(`Name: ${this.firstName} ${this.lastName}`);

}

};

const person = new Person("John", "Doe");

person.displayName();  *// Name: John Doe*

### Instanceof Operator

* The **instanceof** operator returns **true** if the given object is an **instance** of the specified class

const circle = new Circle(5);

console.log(circle instanceof Circle); *// true*

console.log(circle instanceof Object); *// true*

console.log(circle instanceof String); *// false*

console.log(circle instanceof Number); *// false*

### Static Methods

* The **static** keyword defines a **static method** for a class

class MyClass {

static staticMethod() { return 'Static call'; }

}

* Static methods are **part of the class** and not of its instances

console.log(MyClass.staticMethod())

* They can **only** access other static methods via **this** context

static anotherStaticMethod() {

return this.staticMethod() + ' from another method';

}

### Accessor Properties

* Accessor properties are **methods** that **mimic values**
  + Keywords **get** and **set** with **matching identifiers**
  + They can be **accessed** and **assigned** to like properties

class Circle {

constructor(r) { this.radius = r; }

get area() {

return Math.PI \* (this.radius \*\* 2);

}

}

const circle = new Circle(5);

console.log(circle.area); *// 78.5398…*

### Accessor Properties Application

* Accessors are often used for **validation**
  + The **setter** can verify that a **given value** meets requirements

set diameter(value) {

if (value <= 0) {

throw new Error('Diameter must be positive');

}

this.radius = value / 2;

}

* Properties **without** a setter are **read-only** (cannot be assigned)
* Getters can be used for a **validated** or **calculated** property

## DOM Classes

### DOM Elements as Class Instances

* All DOM objects are **instances** of standard DOM classes
  + Always created via **factory functions**, instead of with **new**

const divElement = document.createElement('div');

console.log(divElement instanceof HTMLDivElement); *// true*

* They provide many useful methods and properties
  + Already seen: **addEventListener**, **appendChild**, **remove**, **children**, **parentNode**, **textContent**, **value**, etc.

### Additional DOM Methods

* **cloneNode(*deep*)** create a **duplicate** of the selected element
  + If ***deep*** is true, a **deep-copy** is created

const duplicate = divElement.cloneNode(true);

* **replaceWith()** **replaces** selected element with another

const span = document.createElement('span');

divElement.replaceWith(span);

* **before()** insert element before selected node

**after()** insert element after selected node

### Manipulate Element CSS Class

* **classList** - is a read-only property that returns a collection of the class attributes of specified element

<div id="myDiv" class="container div root"></div>

const element = document.getElementById('myDiv').classList;

*// DOMTokenList(3)  
["container", "div", "root", value: "container div root"]*

* **classList Methods**

<div id="myDiv" class="container div root"></div>

* + **add() -** Adds the specified class values

document.getElementById('myDiv').classList.add('testClass');

* + **remove() -** Removes the specified class values

document.getElementById('myDiv').classList.remove('container');

<div id="myDiv" class="div root testClass"></div>

### HTML Attributes and Methods

* **getAttribute() -** returns the value of attributes of specified HTML element

<input type="text" name="username"/>

<input type="password" name="password"/>

const inputEle = document.getElementsByTagName('input')[0];

inputEle.getAttribute('type'); *// text*

inputEle.getAttribute('name');*// username*

* **setAttribute()** - sets the value of an attribute on the specified HTML element

<input type="text" name="username"/>

<input type="password"/>

const inputPassEle = document.getElementsByTagName('input')[1];

inputPassEle.setAttribute('name', 'password');

<input type="text" name="username"/>

<input type="password" name="password"/>

* **removeAttribute()** - removes the attribute with the specified name from an HTML element

<input type="text" name="username" placeholder="Username..."/>

<input type="password" name="password" placeholder="Password..."/>

const inputPassEle = document.getElementsByTagName('input')[1];

inputPassEle.removeAttribute('placeholder');

<input type="text" name="username" placeholder="Username..."/>

<input type="password" name="password"/>

* **hasAttribute() -**  method returns true if the specified attribute exists, otherwise it returns false

<input type="text" name="username" placeholder="Username..."/>

<input type="password" name="password" id="password"/>

const passwordElement = document.getElementById('password');

passwordElement.hasAttribute('name'); *// true*

passwordElement.hasAttribute('placeholder'); *// false*

* **dataset** obtain **DOMStringMap** of custom **data attributes**

### Combining Elements and Behavior

* Classes can be used to **encapsulate** elements and behavior
  + Store **references** to DOM elements
  + Provide **event handlers**
  + **Methods** that **manipulate** the elements
* This is called the **Component Pattern**
  + Used in many **JS frameworks**, such as React, Vue, Angular
  + Used in the **Custom Web Component API**

## Build-in Collections

### What is a Map?

* A **Map** collection stores its elements in **insertion order**
* A for-of loop returns an array of **[key, value]** for each   
  iteration
* Pure **JavaScript objects** are like **Maps** in that both  
  let you:
  + Assign **values** to **keys**
  + Detect whether something is stored in a key
  + Delete keys

### Adding/Accessing Elements

* **.set**(key, value) – adds a new key-value pair

let map = new Map();

map.set(1, "one"); *// key - 1, value - one*

map.set(2, "two"); *// key - 2, value – two*

* **.get**(key) – returns the value of the given key

map.get(2); *// two*

map.get(1); *// one*

* **.size** – **property**, holding the number of stored entries

### Iterators

* **.entries()** - returns Iterator - array of **[key, value]**
* **.keys()** - returns Iterator with all the **keys**
* **.values()** - returns Iterator with all the **values**
* **let entries = Array.from(map.entries());   
  *// [ [1, 'one'], [2, 'two'] ]***
* **let keys = Array.from(map.keys()); *// [1, 2]***
* **let values = Array.from(map.values()); *// ['one', 'two']***

### Iterating a Map

* To print a map simply use one of the **iterators** inside a **for-of**

let iterable = phonebookMap.keys();

for(let key of iterable) {

console.log(`${key} => ${phonebookMap.get(key)}`);

}

for(let [key, value] of phonebookMap) {

console.log(`${key} => ${value}`);

}

### Map Sorting

* To **sort** a Map, first transform it into an **array**
* Then use the **sort()** method

let map = new Map();

map.set("one", 1);

map.set("eight", 8);

map.set("two", 2);

let sorted = Array.from(map.entries())

.sort((a, b) => a[1] - b[1]);

for (let kvp of sorted) {

console.log(`${kvp[0]} -> ${kvp[1]}`);

}

**What is a Set?**

* Store **unique values** of any type, whether **primitive** values or **object** references
* Set objects are **collections** of values

let set = new Set([1, 2, 2, 4, 5]);

// Set(4) { 1, 2, 4, 5 }

set.add(7)); // Add value

console.log(set.has(1));

// Expected output: true

* Can **iterate** through the elements of a set in **insertion** order

### WeakMap and WeakSet

* **Special variants** of Map and Set
* Their elements **do not** count as **active** references
  + Reference types visible (in **scope**) in the program stack are **active**
  + Active references **remain in memory**
  + **Out-of-scope** references are **removed** by the **garbage collector**
* These collections are used in **memory-intensive** applications

# Prototypes and Inheritance

## Internal Object Properties

### Internal Properties

* Every object field has **four** properties:
  + **Enumerable** - can access to all of them using a **for…in** loop
    - Enumerable property are returned using **Object.keys** method
  + **Configurable** - can **modify** the **behavior** of the property
    - You **can** **delete** only **configurable** properties
  + **Writable** - can **modify** their **values** and update a property just   
    assigning a new value to it
  + **Value**

### Object's Non-enumerable Properties

* They won't be in for…in iterations
* They won't appear using Object.keys function
* They are not serialized when using JSON.stringify

let ob = {a:1, b:2};

ob.c = 3;

Object.defineProperty(ob, 'd', { value: 4, enumerable: false });

ob.d; *// => 4*

for( let key in ob ) console.log( ob[key] ); *//1 2 3*

Object.keys( ob );  *// => ["a", "b", "c"]*

ob; *// => {a: 1, b: 2, c: 3, d: 4}*

ob.d; *// => 4*

### Object's Non-writable Properties

* Once its value is defined, it is **not possible to change** it using assignments

let ob = { a: 1 };

Object.defineProperty(ob, 'B', { value: 2, writable: false });

ob.B; // => 2

ob.B = 10;

ob.B; // => 2

* If the non-writable property **contains** an **object**, the **reference** to the object is what is **not** **writable**, but the **object** itself **can** **be** **modified**

### Object's Non-configurable Properties

* Once you have defined the property as **non-configurable**, there is only **one** **behavior** you **can** **change**
  + If the property is **writable**, you can convert it to non-writable
  + Any other try of definition update will **fail** throwing a TypeError

let ob = {};

Object.defineProperty(ob, 'a', { configurable: false, writable: true });

Object.defineProperty(ob, 'a', { enumerable: true }); *// throws a TypeError*

Object.defineProperty(ob, 'a', { value: 12 }); *// throws a TypeError*

Object.defineProperty(ob, 'a', { writable: false }); *// This is allowed!!*

Object.defineProperty(ob, 'a', { writable: true }); *// throws a TypeError*

delete ob.a; *// => false*

### Object Freeze and Seal

let cat = { name: 'Tom', age: 5 };

Object.freeze(cat);

cat.age = 10;          *// Error in strict mode*

cat.gender = 'male';  *// Error in strict mode*

console.log(cat);      *// { name: 'Tom', age: 5 }*

cat = { name: 'Tom', age: 5 };

Object.seal(cat);

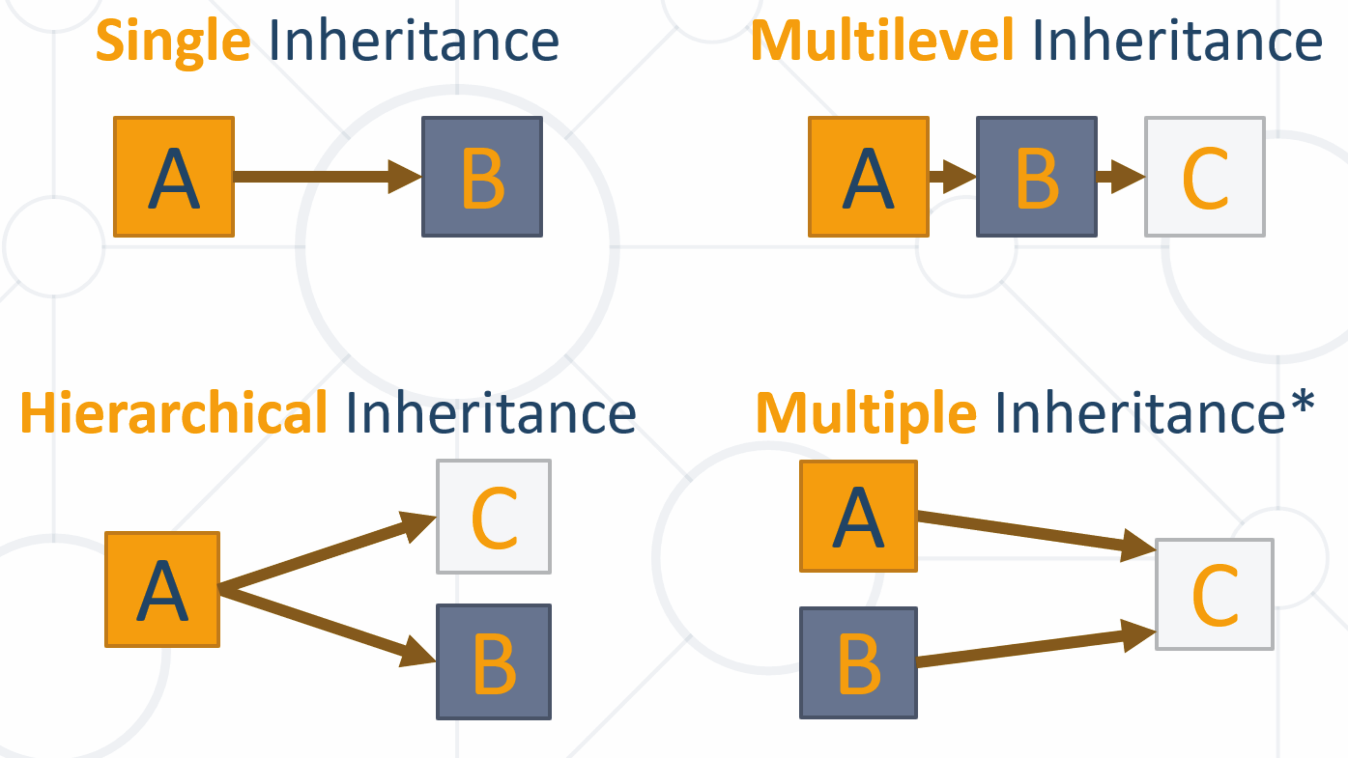
cat.age = 10;          // OK

delete cat.age;        // Error in strict mode

console.log(cat);     // { name: 'Tom', age: 10 }

## Inheritance

### Types of Inheritance



*\* Not supported in JS with* ***classes****, but works with* ***composition***

## The Prototype

### What is a Prototype?

* Every object in JS has a **prototype** (template)
  + Internally called **\_\_proto\_\_** in browsers and NodeJS
  + Properties lookup follows the **prototype chain**
* Obtained with **Object.getPrototypeOf(*obj*)**
* **Reference** to another objects
  + Objects are **not** separate and disconnected, but  **linked**

***Note****:* ***\_\_proto\_\_*** *is for debugging and should* ***never*** *be used in production code!*

### Prototype

* Objects **inherit** **properties** and **methods** from a **prototype**
* The **prototype property** allows you to add **new properties** to object **constructors**

function Person(first, last, age) {

this.firstName = first;

this.lastName = last;

this.age = age;

}

Person.prototype.nationality = "Bulgarian";

### Simulated Class Functionality

* Before ES6, **classes** were composed **manually**

function Rectangle(width, height) {

    this.width = width;

    this.height = height;

}

Rectangle.prototype.area = function () {

    return this.width \* this.height;

}

let rect = new Rectangle(3, 5);

### Object Creation

* **Literal** creation
* **Constructor** creation
  + Have an **implicit reference** (prototype) to the value of their constructor's "prototype" property
  + Gets an internal **\_\_proto\_\_ link** to the object

### JavaScript Objects

|  |  |
| --- | --- |
| * Literals   let bar = {  me: "I am b1",  speak: function() {  console.log("Hello, " + this.me + ".");  }  }; | * Constructed   function Bar(name) {  this.me = "I am " + name;  this.speak = function() {  console.log("Hello, " + this.me + ".");  };  };  let b1 = new Bar("b1"); |

### Object Create

* The **Object.create()** method creates a **new object**, using an existing object as **prototype**

const dog = {

name: 'Sparky',

printInfo: function() {console.log(`My name is ${this.name}`)}

};

const myDog = Object.create(dog);

myDog.name = 'Max'; *// inherited properties can be overwritten*

myDog.breed = 'shepherd'; *// breed is a property of myDog*

myDog.printInfo(); *// My name is Max*

### \_\_proto\_\_ vs Prototype Property

* **\_\_proto\_\_** 
  + Property of an objects that **points** at the prototype that has been **set**
  + Using **\_\_proto\_\_** directly is deprecated!
* prototype
  + Property of **a function** set if your object is created by a **constructor** **function**
  + Objects do not have **prototype** property

### Prototype Inheritance

function Foo(who) {

     this.me = who;

}

Foo.prototype.identify = function () { return "I am " + this.me; }

function Bar(who) { Foo.call(this, who); }

Bar.prototype = Object.create(Foo.prototype);

Bar.prototype.speak = function () {

     console.log("Hello, " + this.identify() + ".");

}

let b1 = new Bar("b1");

let b2 = new Bar("b2");

b1.speak(); b2.speak();

## Class Inheritance (ES6)

### Traditional Classes

* Classes are a **design pattern**
* Classes mean - creating **copies**
  + When **instantiated** – a **copy** from class to instance
  + When **inherited** – a **copy** from parent to child
* Class inheritance is a powerful tool, but has many **drawbacks** and **limitation**
  + **Composition** should be **preferred** whenever possible!

### Class Inheritance

* Classes can **inherit** (extend) other classes
  + Child class inherits data + methods from its parent
* **Child** **class** can:
  + Add **properties** (data)
  + Add / replace **methods**
  + Add / replace **access** properties
* Use the keyword **extends**

class Person {

constructor(name, email) {

this.name = name; this.email = email;

}

}

class Teacher extends Person {

constructor(name, email, subject) {

super(name, email); //Invoke the parent constructor

this.subject = subject;

}

}

### Classes in JavaScript

* **Prototypal** **inheritance** instead of classical inheritance
* **Does not automatically** create copies
* Common keys and values are shared by **reference**
* **Delegates not blueprints!**