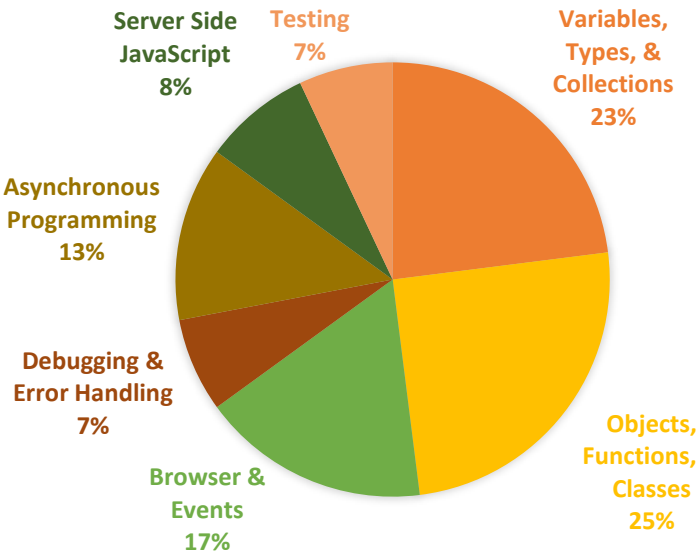
JAVASCRIPT DEVELOPER I EXAM CHEAT SHEET (UNOFFICIAL) – BY SANTANU BORAL

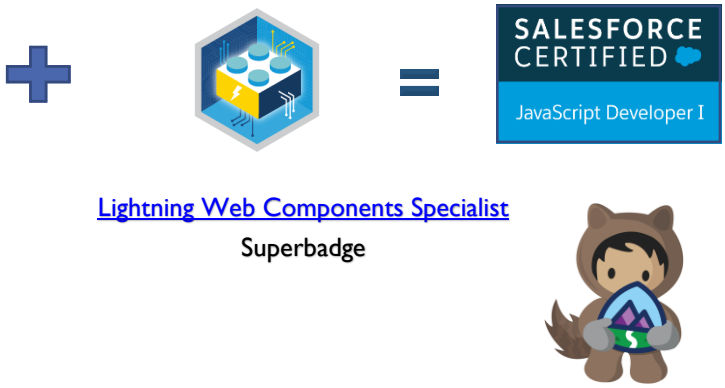
OVERVIEW

Multiple Choice Questions: 65 ques (5 unscored), pass: 65%, duration: 105 mins (No questions from LWC)



Trailmix: Prepare for your Salesforce JavaScript Developer I Credential

Trail: Study for the Salesforce JavaScript Developer I Exam



Lightning Web Components Specialist Superbadge

FEW IMPORTANT TOPICS FOR EXAM

DATATYPES & DECLARATIONS

Primitive	falsey	Object Wrapper & Type Casting
Boolean	false	Boolean b = new Boolean('false');

Number (holds decimal, float, integer, NaN)	0 and NaN	Number num = new Number(9.08);
		let num2 = 9.08;
		num === num2; //false
		num == num2; //false;
		Number ('9'); // 9

String	(""") or (')	Number.parseFloat('9.09'); Number.parseInt('2');
symbol		String str = new String('sun');
null	null	
undefined	undefined	

Falsey always returns false

```
if(0){
    //block never executes
}
```

Type Coercion & Operator Precedence

```
10 + '2' + undefined; // "102undefined"
true + 10 * 2; //21
'bat' + null; // "batnull"
"35" – 5; //30
```

```
typeof ('99') // "string" returns a string with a value

instanceof New String ('Bat');
//checks value is of object instance.

typeof(null); //"object"

const p = Number ('9');
p instanceof Number; //false

const q = new Number ('10');
q instanceof Number; //true
```

Refer: Grammar & Types.

Declaration	Usage	Initialization	Variable Hoisting (Use before declare)	Scope

var	var x; //undefined
	var x = 10;
		Optional	console.log(x); //undefined
			var x = 10;
				Function

let	let str = 'santanu';
	let name; //undefined
		Optional	console.log(x); //ReferenceError
			let x = 10;
				Block

const	const x = 10;
	x = 7; //can't reassign
		Mandatory	console.log(x); //ReferenceError
			const x = 10;
				Block

No declaration	x = 7; //same as below
	console.log(window.x)
	;
		Optional	console.log(x); //ReferenceError
			x = 10;
				Window or global

Primitive datatypes are **immutable**. Variables can be assigned to a new value but existing values cannot be changed.

Check for: &&, ||, ==, === , !=, !==, !! operators and usage and comparing value and object types.

Refer: Expressions & Operators

Example of Scopes

```
//block (trying to access let outside of block)
{
   let x=10;
}
console.log(x); //ReferenceError

//function (trying to access var outside of it)
const str =() => {
   var initialStr = 'tom';
   if (!!initialStr) {
      var changeStr = 'jerry';
   }
   console.log(initialStr); //tom
console.log(changeStr); //jerry
}
str();
console.log(changeStr); //ReferenceError
```

DATE FUNCTIONS

```
const myDt = new Date(); //today's date and time
dt.setDate(myDt + 10); //increment 10 days from today
dt = new Date (2008, 12, 30);
```

check getTime(), setTime(), Date.toString().

Refer: Date Functions

STRING METHODS AND USAGE

Few methods have been given for illustrations. Refer String Methods

concat
includes
indexOf
charAt
match
replace
slice
substring
substr
	"manu".concat("facture"); // "manufacture"
	
	const sentence = 'The quick brown fox jumps over the lazy dog.';
	const word = 'fox';
	const index = 4;
	
	console.log (sentence.indexOf(word)); //16
	console.log (sentence.includes(word)); //true
	console.log (`The character at ${index} is ${sentence.charAt(index)}`);
	//"The character at index 4 is q"
	console.log(sentence.substring(1,3)); //he
	console.log(sentence.slice(4,19)); // "quick brown fox"
		const str1 = '5';
		
		console.log (str1.padStart(2, '0')); //"05"
		const str2 = '200'; console.log(str2.padEnd(7,'ab'));
		//200abab
		
		var str = "The rain in Spain";
		var res = str.match(/ain/g);
		console.log(res); // Array ["ain", "ain"]

COLLECTIONS

ARRAYS [INDEXED COLLECTION] - Stores multiple values into single variable. Refer: Array

let fruits = ['Apple', 'Banana', 'Orange']; //single dimensional array
let fruits = new Array ('Apple', 'Banana', 'Orange');
let arr = [['a', 1], ['b', 2], ['c', 3]]; //multi-dimensional array
	//following creates array taking each character
	let fruits = Array.from ('Apple'); // ["A", "p", "p", "l", "e"],
	let arr = Array.of(5); //[5], here 5 is value of 0th index
	let arr2 = Array (3); //[undefined, undefined, undefined] , creates array with size 3
	Array.isArray(fruits); //true

Loop through an Array

for…in (index wise)	for…of (element wise)	Traditional for loop	for…each (operates on function)

let fruits = ['Apple', 'Banana', 'Orange'];
for (let x in fruits) {
console.log(fruits[x]);
}
// Apple, Banana, Orange
	let fruits = ['Apple', 'Banana', 'Orange'];
	for (let x of fruits) {
	console.log(x);
	}
	//Apple, Banana, Orange
		const arr = [1, 4, 9, 16];
		for (let i=0; i< arr.length; i++){
		console.log(arr[i]);
		}
		//1,4,9,16
			[2, 5, 9].forEach(logArrayElements);
			function logArrayElements(element, index, array) {
			console.log('a[' + index + '] = ' + element);
			}
			//a[0] = 2, a[1] = 5, a[2] = 9

Creating and returning new Array (original array content does not change)

map function – creates an array based on function's operations	**filter –** creates a new array with reduced number on the conditions applied.	**slice –** returns shallow copy portion of an array into new Array object.

const arr = [1, 4, 9, 16];
// pass a function to map
const mapA = arr.map(x => x * 2);
console.log(mapA);
// expected output: Array [2, 8, 18, 32]
	const arr = [1, 4, 9, 16];
	// pass a function to map
	const mapA = arr.filter(x => x % 2);
	console.log(mapA);
	// expected output: Array[4,16]
		const arr = [1, 4, 9, 16];
		console.log(arr.slice(1,3)); //final index omitted
		// expected output: Array[4,9]

Changing original array content

sort – returns sorted array	**splice** – changes the content by adding or removing elements	**reduce –** executes reducer function on each element resulting single output value.	**push –** add elements(s) at end.

const arr = [1, 4, 9, 16];
console.log(arr.sort());
//Array[1,16,4,9]
	const arr = [1, 4, 9, 16];
	//replaces first element with 5
	arr.splice(0,1,5);
	console.log(arr); //Array[5,4,9,10]
		const arr = [1, 4, 9, 16];
		const reducer = (acc, curr) => acc + curr;
		// 1 + 4 + 9 + 16
		console.log(arr.reduce(reducer)); //output: 30
			const arr = [1, 4, 9, 16];
			arr.push(25); //Array[1,4,9,16,25]
			arr.pop(); //removes last element
			refer shift, unshift functions

MAP AND SET

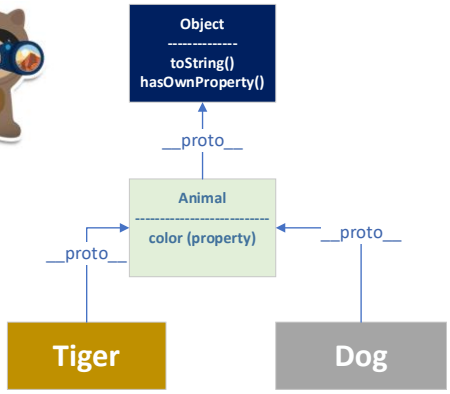
MAP – holds key/value pair. Refer: Map	**SET –** holds unique values (no duplicates) Refer: Set

let vmap = new Map ([['a', 1], ['b', 2], ['c', 3]]);
vmap.set('b',10); //assigns values based on key
vmap.get('c'); //get the values based on key
vmap.has('a'); //check existence
Refer: add, delete, keys, values, forEach functions on Map
	let pSet = new Set([1,4,9,4,16]);
	console.log(Array.from(pSet.values())); //Array[1,4,9,16]
	pSet.has(16); //check existence
	pSet.size(); //size of array, output 4
	Refer: add, delete, keys, values, forEach functions on Set

JSON – Serializing objects, arrays, numbers, strings, booleans and nulls.

JSON.parse – parse a JSON string and converts JavaScript value or object.	**JSON.stringify** – converts JavaScript object or value to JSON String.

const json = '{"result":true, "count":42}';
const obj = JSON.parse(json);
console.log(obj.result); //true
	console.log(JSON.stringify([new Number(3), new String('false'), new Boolean(false)]));
	//expected output: "[3,"false", false]"



- **Prototypical** object – defines a template from which new objects can get initial properties.
- **Class** based are Object oriented languages.
- Every object has __proto__ Object property which refers Parent object.
- Objects are mutable.
- If any property/method is reference on an object, then to find this existence system checks entire prototype chain
- Newly created Object which doesn't refer to a prototype should have toString() and hasOwnProperty().
- Objects are passed by reference; primitives are passed by value

CREATING OBJECTS

Using **new** operator from class	Using **literal**	Using **function**	Using **Prototype** with Object.create

class Employee {
constructor() {
this.name = '';
this.dept = 'general';
}
}
let emp = new Employee();
emp.name = 'Santanu';
	let emp = {
	name: "Santanu",
	dept: "IT"
	}
		function createEmp (name, dept){
		return {
		name: name,
		dept: dept
		}
		}
		let emp = createEmp('Santanu', 'IT');
			const employee = {
			name: '',
			dept: ''
			}
			const emp = Object.create (employee);
			emp.name = 'Santanu';
			emp.dept = 'IT';

DEFINING AND USING PROPERTIES

Key/value using semi-colon	Assigning **hardcoded** property	**Dynamic** assignment	Using **Object.defineProperty**	Using **getter/setter**

let emp = {
name: "Santanu",
dept: "IT"
}
//to delete property
delete emp.name;
	let emp = {
	name: "Santanu",
	dept: "IT"
	}
	emp.Id = "1001";
		emp [dynamicValue] = 'Kolkata';
		emp ['id'] = 1001;
			Object.defineProperty(emp, 'Dol', {
			value: new Date()
			});
			Refer: Enumerable, Configurable, Writable
				let emp = {
				sname: '',
				get name(){
				return this.sname;
				},
				set name(str){
				this.sname = str;
				}
				}
				emp.name ='Santanu';

FEW MORE IMPORANT STATIC METHODS

Object.**keys** – returns enumerable keys
Object.**values** – returns list of property values
Object.**assign** – copy objects/properties
Object.**freeze** – objects cannot be changed anymore
Object.**seal** – no new properties, existing properties will be non-configurable
	let emp = {
	name: "Santanu",
	dept: "IT"
	}
		console.log (Object.keys(emp)); // Array ["name", "dept"]
		console.log (Object.values(emp)); //Array ["Santanu", "IT"]
		const returnTarget = Object.assign(emp, {a:1, b:2});
		console.log(returnTarget); // Object { a:1,b:2,dept: "IT" ,name: "Santanu" }
		Object.seal(emp);
		delete emp.name; //cannot delete
		Object.freeze(returnTarget);
		returnTarget.dept = "Finance" //cannot change property value

FUNCTIONS

DEFINING FUNCTIONS

Using **function**	Using **expression**	Using **Arrow** (ES6)

function displayLog(str){
  console.log(str);
}
console.log(displayLog(3)); //3
 	
//with default parameter
function multiply (a, b = 1) {
  return a * b;
}
	const squareANumber = function square (number) {
	  return number * number;
	}
	console.log(squareANumber(3)); //Output: 9
		const squareANumber = (number) => number * number;
		//check no function and return keywords (clean writing)
		
		console.log(squareANumber(3)); //Output: 9
		
		const printLog = () => console.log('Hello'); //without parameters
		console.log(printLog());//Output: Hello

UNDERSTANDING **this**

- **this** is determined how function is called
- **this** cannot be set during execution
- It varies each time when function is called
- bind() to set regardless how it is called
- Arrow function retains **this** value of => context

this varies context wise

Global Context, refers to **window**	**Function with/out** strict mode	Function is called on an Object, **this** refers to **Object instance itself**	Function is used as **Constructor**; this refers to newly created **Object** instance

console.log (this === window);
a = 45;
console.log(window.a);
//45
	function f1() {
	  return this;
	}
	// In a browser:
	f1() === window; // true
	
	function f2() {
	  'use strict';
	  return this;
	}
	
	f2() === undefined; // true
		var o = {
		  prop: 10,
		
		  myFunc: function() {
		  return this.prop;
		  }
		};
		
		console.log(o.myFunc()); // 10
			function myFunction() {
			
			  this.num = 10;
			
			}
			
			var o = new myFunction();
			
			console.log(o.num); // 10

Arrow function holds this context	Example of Dynamic Binding	Using **call**: specify this and pass parameters individually	Using **apply**: this and array as parameters	Using **bind**: this and receive a reference

var globalObject = this;
 	
var foo = (() => this);
foo() === globalObject;
// true
	let product = {name: "Prod1"}
	
	function works(arg1, arg2){
	  console.log(`${this.name} has ${arg1} and ${arg2}`);
	
	}
		works.call(product, 'height', 'width');
		
		//Output: Prod1 has height and width
			works.apply(product, ['height', 'width']);
			
			//Output: Prod1 has height and width
				let prod = works.bind(product);
				prod('height', 'width');
				
				//Output: Prod1 has height and width

ASYNCHRONOUS PROGRAMMING

Not to prevent our applications to perform certain tasks that could be delayed due to other time-consuming operations, perform that in async way.

Callback function – is passed as param which is being called at any point of time. Like, setInterval is calling myCallback in every second up to thrice	**Promises** – Putting aside a long running function, when it is resolved or rejected and call stack is empty, we then use its value.	**Async** – Typically used with Promise which resolves to a value and await is also used with it.

var iCount = 1;
 	
var intervalID = setInterval (myCallback, 1000, 'Hello', 'World');
 	
function myCallback(a, b) {
 	
 console.log(a,b);
 	
 if (iCount ===3) clearInterval (intervalID);
 	
 iCount++;
}
//Output:
Hello World
Hello World
Hello World
	let myPromise = new Promise((resolve, reject)=>{
	 setTimeout(() => resolve("done"), 1000);
	 setTimeout(() => reject("error"), 2000);
	});
	
	myPromise.then(result => {console.log(result);}
	).catch(error => console.log(error)
	).finally(()=> console.log('finally'));
	
	//Output: done, finally [as resolve is returned first]
	if we change reject timeout to 500 then output will be error, finally
	Promise states – fulfilled, rejected, pending
	
	Methods – all, allSettled, race. Refer: Promise
		const promise1 = Promise.resolve('First')
		const promise2 = Promise.reject('Second')
		
		const runPromises = async () => {
		return await Promise.all([promise1, promise2])
		}
		
		runPromises()
		.then(res => console.log(res))
		.catch(err => console.log(err))
		
		//Output: Second, if we use Promise.race then First will be the output.
		For Promise.allSettled, output will Array with First and Second values.

CLASSES

Class is a template for an object and is a "syntactic sugar" on Prototype. It has properties and methods. JavaScript does not support multiple inheritance.

Create a class with height, width property and calculateArea method	Extend Shape class and call parent class' constructor and methods by **super** keyword

class Shape {
constructor (height, width) {
this.height = height;
this.width = width;
}
calculateArea(){
console.log('Calculate Area');
}
}
	class Square extends Shape {
	constructor (height, width, name) {
	super(height, width);
	this.name = name;
	}
	calculateArea(){
	super.calculateArea();
	console.log(`${this.name} area is`, this.height* this.width);
	}
	}
	//instantiate class and call its method
	let myShape = new Square(20,30,'Square Shape');
	myShape.calculateArea();
	//Output: Calculate Area, Square Shape area is 600

Difference with Prototype

1. Function created by class labelled by special internal property [FunctionKind]]:"classConstructor"
2. Unlike regular function, it must be called with new keyword.
3. Methods of a class are non-enumerable. A class definition sets enumerable flag to false to all the methods in the prototype.
4. Classes always run in strict mode. Refer Class

ERROR HANDLING

try-catch-finally	try-catch blocks can be **nested**, also below example of **throwing errors**

try {
//try to execute the code
} catch(e) {
//handle errors
} finally {
//execute always
}
try without catch and try without finally is possible. finally block is optional with try..catch block
	try {
	//try to execute the code
	try{
	console.log('inner try');
	throw err;
	}catch(e){
	console.log('inner catch');
	throw new Error('My Error Desc');
	}finally{
	console.log('inner finally');
	}
	} catch(e) {
	console.log('outer catch');
	} finally {
	console.log('outer finally');
	}
	//Output:
	inner try
	inner catch
	inner finally
	outer catch
	outer finally

Few important points

- try-catch only works for runtime errors
- try-catch works synchronously
- throw operator generates an error
- Variables are local inside try-catch-finally block
- Build-in constructors for standard errors: Error, SyntaxError, TypeError, ReferenceError
- catch only process those error which it knows and rethrow all other errors.

MODULES

Module is file which can be reused.

About a Module	Sample module script	Exporting a module	Importing a module

- Each module has its own scope
- Module automatically runs in strict mode
- Module is evaluated only first time when it is imported
- In a module, 'this' is undefined
- Module scripts are always deferred.
	<script type="module" src="hello.js">
	</script>
	<script type="module">
	// The variable is only visible in this module script
	let user = "Santanu";
	console.log('I am a module');
	</script>
		//myExport.js
		export default class User {
		constructor(name) {
		this.name = name;
		}
		}
		//myExport2.js
		export function sayHi(){ ... }
		export function sayBye() { ... }
			//main.js
			import User from './myExport.js';
			new User('Santanu');
			
			//main2.js
			import {myExport2} from './myExport.js';
			or,
			import * from as user from './myExport.js'; //to import all

TESTING WITH CONSOLE METHODS

console.log – outputs a message to web console
console.info - outputs an informational message to web console
console.error - outputs an error message to web console
console.warn - outputs a warning message to web console
console.table – displays data in tabular format
	console.assert – writes an error message when assertion is false, if assertion is true then nothing happens.
	console.assert(false, 'comparing values', !!true);
	//Output: Assertion failed {"comparing values"} true
	Refer: Console

TYPES OF TESTING

White-box testing	Black-box testing
High Complexity. Efficient in finding errors and problems. Helps to optimize code. Requires developer's introspection.	Testing Efficient for large segment of code. Code access is not required. Less complex. Separation between user and developer perspective.

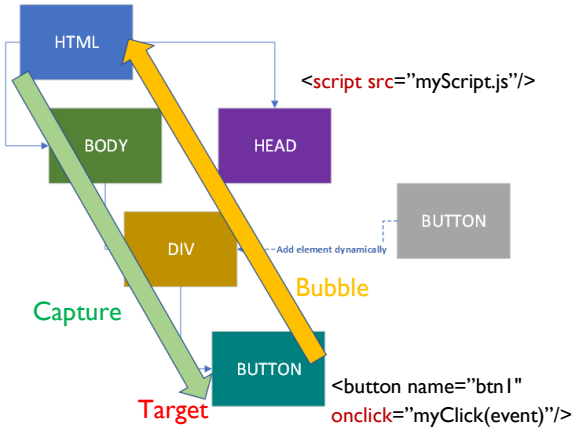
False positive – may not fail when we break application code

False negative – fail when we refactor code

DOCUMENT OBJECT MODEL (DOM) AND EVENTS

When webpage is loaded, browser create DOM on the page.

Right side screen
- Defining external script.
- Attach event listener on btn1
- Dynamically creating btn2 under DIV and attaching event listener dynamic with it.
- It shows event propagation during capture and bubble phase (default event firing phase).



Creating an element, adding event on the fly

```
<script>
  const btn2 = document.createElement('button');
  btn2.innerHTML = 'click on me';
  btn2.name = 'click on me';
  document.querySelector("#myDivId").appendChild(btn2);
  btn2.addEventListener('click', function(e){
                alert(e.target.name);
                 });
</script>
//to fire custom event from button, use following in addEventListner:
this.dispatchEvent(
new CustomEvent('myEvt', detail: {parameters})
);
```

Window Object	Use of selector	Important methods of Event	
Variables are global. Few important methods to read: location, history, open Refer: Window	**querySelector** – returns first element of matching selector **querySelectorAll** – returns all the elements. Refer: querySelector	**stopPropagation** – halt event propagation, let's say during bubble phase, you don't want event to be fired on outer element. Here event.currentTarget comes into play. Refer: stopPropagation	**preventDefault** – lets say, you want restrict user to view a site or performing certain action, you use this method. Refer: preventDefault

REFERENCES

https://trailhead.salesforce.com/en/content/learn/trails/study-for-the-salesforce-javascript-developer-i-exam
https://javascript.info/js
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference

FURTHER READING

Tips for passing Salesforce certified JavaScript Developer I Certification by Santanu Boral

ABOUT AUTHOR

Santanu Boral, Salesforce MVP, 23x certified (including Salesforce JavaScript Dev I certification)

Twitter: @santanuboral

Linkedin: Santanu Boral

My Blog: http://santanuboral.blogspot.com/

Date: 4th Sept 2020