Javascript written test

- This test will cover the following areas:
  - Primitive values, types, type coersion, immutability of primitive types versus mutability of compound types
  - Variable and function scopes, hoisting
  - Function declaration, function expressions, scoping
  - Object properties and mutation
  - Assignments and comparisons
  - Pure functions and side effects
  - Naming variables

Primitive values, types, type conversion

What this unit will test

- What primitive values are
- Which primitive values also have object equivalents
- How are implicit and explicit type coersions performed?

1. Which values are considered primitive in JS? And what makes them unique? Primitive data types include: boolean, null, undefined, number, and string. Primitive data types are immutable, unlike the object data types. No methods can be called on primitive data types either.

2. Which of the primitive values have object equivalents? boolean, string, and number all have object equivalents. They can be implicitly or explicitly converted to objects.

3. What does an implicit or explicit type conversion look like? Give examples. Implicit conversion can look like this: `'11' + 10` => 1110, with the + operator, this becomes `'11' + '10'` `11 * '10'` => 110, with the * operator, this becomes `11 * 10`

Explicit conversion can look like this:

```
aString = '123';
typeof aString; //  string
aNumber = Number(aString);
typeof aNumber; // number
```

4. You create a new primitive data type by declaring `var aString = 'string`, how would you declare an object data of the same type?

```
var anObject = new String('a string');
typeof anObject; // object
```

5. Supposed you have declared a variable that belongs to the primitive data type of string. You want to access the methods available to the `String` object data type. How would you go about doing it without changing the original variable data type?

JS allows for the temporary coersion of a data type, so when needed, the string primitive data type is temporary coersed into a String object data, and can thus access the methods available to a String object.

```
var stringObject = "This is a string";
typeof stringObject; // string
console.log(stringObject.toUpperCase()); // "THIS IS A STRING"
typeof stringObject; // string
```

Variable scopes and hoisting

6. How are variables scoped? Demonstrate with an example. Variables are accessible from within the scope it's created. Scoping occurs when a function is created, and variables can only be accessed from the scope from which it's created.

```
var globalScope = 'I belong to the global scope';
function innerFunction() {
  var innerScope = 'I belong to the inner scope';
}
console.log(globalScope); // I belong to the global scope
console.log(innerScope); // innerScope is not defined
```

7. What creates a scope? A scope can be created in two ways: either with the creation of a function as in the example above, or when a block is created {}. This is an example of block scoping, where variables declared with `var` is accessible from the global scope, but those created with `let` or `const` are not.

```
{
  var varScope = 'This is a var scope.'
  let letScope = 'This is a let scope.'
  const constScope = 'This is a const scope.'

  console.log("Inner scope " + varScope);
  console.log("Inner scope " + letScope);
  console.log("Inner scope" + constScope);
}

console.log("Outer scope  " + varScope);
```

```
console.log("Outer scope " + letScope); // letScope is not defined
console.log("Outer scope " + constScope);
```

8. Provide an example of variable scoping with various scopes.

In the below example, a global variable `yard` is declared and is available globally. The function `insideHouse` creates an inner scope, to where variable `garage` belongs to. A second function `insideHouse` is created that's within the first function, and the variable `livingRoom` is accessible only within this scope. As a result, calling the `insideHouse` function would access the `garage` and `yard` variables, but not the `livingRoom` variable.

```
var yard = 'A house has a yard.';

function insideHouse() {
  var garage = 'A house has a garage.'

  function insideHouse() {
    var livingRoom = 'A house has a living room.';
  }

  console.log(yard); // A house has a yard.
  console.log(garage); // A house has a garage.
  console.log(livingRoom); // livingRoom is not defined
}

insideHouse();
```

9. What is variable hoisting, and provide an example of it.

```
console.log(showVar);
var showVar = 12; // undefined
```

In this example, the variable declaration is hoisted to the top of the scope, but not the variable assignment. Therefore, logging the variable will show `undefined`, and not an unknow error nor the value assigned to the variable.

Function declaration, function expressions, scoping

10. What are the differences between a function declaration and a function expression? Accessibility: in a function expression, a function declaration is assigned to a variable. The function variable is accessible in the global scope, but the function name itself (anonymous or not) is not. Anonymous functions: a function expression can be anonymous, but a function declaration cannot be. Hoisting: rules are different for both of these. In a function expression, only the variable declaration is hoisted to the top of thes scope. In a function expression, the entire function is hoisted to the top of the scope. This means that function expressions should be defined before called

To demonstrate accessibility differences as a result of hoisting:

```
functionDeclaration(); // This is a function declaration
function funcDecalaration() {
  console.log('This is a function declaration');
}

console.log(functionExpressVar); // undefined
var functionExpressVar = function funcExpression() {
  console.log('This is a function expression');
}
```

*** Questions ***

```
var funcExpressionVar = function funcExpression() {
  console.log('This is a function expression');
}
funcExpressionVar; // should be funcExpressionVar()
console.log(typeof funcExpressionVar); // function
```

- Why does this not log anything? Why does the variable `funcExpressionVar` not pick up the return value of function declaration `funcExpression()`? Because it needs to be a function -> funcExpressionVar(), a function needs to be returned, and not a variable

11. What is scoping closure and demonstrate with an example. This is consistent with the concept behind lexical scoping, where scope is determined at function definition, and not function invocation. Therefore, the scope defined at function definition determines the scope at which vars are accessible. This means that functions can access variables within scope at method definition time, even if at method invocation time, it may appear out of scope.

```
function originalScope() {
  var innerScope = 'I am part of the original scope';
  console.log(innerScope);
}

function anotherScope() {
  var innerScope = 'I am part of another scope';
  originalScope();
}
```

```
anotherScope();
```

Object properties and mutation

What this unit will test

- Naming and manipulating built-in objects
- 3 ways of creating objects, and creating custom objects
- Creating, editing, and accessing object properties, and understanding the difference between object data and behaviour
- Four ways of enumerating over objects, editing enumerable flag
- Avoid unexpected behaviour when enumerating over object created based on prototype
- Array length

12. What are some built-in objects in JS? String, Array, Math, Date. Some built-in objects share the same name as their primitives counterparts. The primitives like `boolean`, `string`, and `number` can be temporarily coersed into their object counterparts and access the methods available to them. Note that `undefined` has no object counterpart.

13. How do you create new objects? Objects can be created in one of three ways: the first two use the built-in objects, the last one is a custom object created using the object literal notation.

- Using `new String('new string')`

```
var newString = new String('hello');
console.log(newString);
console.log(typeof newString);
```

- Using `Object.create()`

```
var dog = {
  name: 'Menno',
  type: 'Jack Russell'
}

myDog = Object.create(dog);

console.log(myDog.name);
console.log(myDog.type);
```

- Create a new custom object using the object literal

```
var newObject = {
  sides: 4,
  shape: 'rectangle',
  angles: [50, 100, 30],
}

console.log(newObject.sides);
console.log(newObject.shape);
console.log(newObject.angles);
```

14. What are the components that make up an object?

Some keywords are: properties (attributes, data), behaviour

An object consists of data, behaviour, and properties. Data describes various aspects of an object, e.g. a shape object will have size, number of sides, and angles, and the data is what fills up these various "attributes", or parts that describe an object. The behaviour tells us how an object will act, for example, an object has methods attached to it. The built-in objects of JS all have different methods that tells it how to act, e.g. `String.toUpperCase()`, these are the behaviours that govern an object. Properties are what we call the key-value pair of name and data, and is what we use to describe an object. For example, the same shape object will have `{ size: 10; sides: 4; angles: [50, 30, 100]}`, these key-value pairs are called properties.

15. Can any data types be used for an object property? String, number, boolean, object, function can all be used as an object property. The only thing to note is that number and multi-words strings can only be accessed using the bracket notation and not the dot notation.

```
var newObj = {
  property1: 5,
  property2: 'string',
  property3: true,
  func1: function() {},
  true: false;

  'property4': 10,
  5: 'property5',
  'property five': 'property 5',
  block1: {
    prop1: 'block1, prop1',
    prop2: 'block1, prop2',
  },
  func2: function () {
```

```
   console.log('function2')
  },
};

console.log(typeof newObj.property1); // number
console.log(typeof newObj.property2); // string
console.log(typeof newObj.property3); // boolean
console.log(typeof newObj.func1); // function
console.log(typeof newObj.true); // boolean

console.log(typeof newObj.property4); // number
// console.log(typeof newObj.'5'); // can't access a numbered attribute using the dot notation
console.log(typeof newObj[5]); // string
console.log(typeof newObj['property five']); // string, can't access two-words attribute using dot notation
console.log(typeof newObj.block1); // object
console.log(typeof newObj.func2); // function
```

16. Demonstrate how you can i) add and delete properties in an object, ii) how to step through the object, iii) how to print out all properties of the object.

```
var newClass = {
  boy1: 'Aaron',
  boy2: 'Bob',
  girl1: 'Cathy',
  girl2: 'Dora',
}

newClass.girl3 = 'Ellie';
console.log(newClass);
delete newClass.boy1;
console.log(newClass);

for (student in newClass) {
  console.log(newClass[student]);
}

console.log(Object.keys(newClass));
```

17. What are 4 ways of iterating over a collection `var newArray = [1, 2, 3, 4]`?

- Using a `for` loop

```
for (i = 0; i <= newArray.length; i++) {
  console.log(newArray[i]);
}
```

- Using a `for..in` loop, it loops over the properties of an object, it returns all the properties of an object, thus need the [] to get the values

```
for (var arr in newArray) {
  console.log(newArray[arr]);
}
```

- Using a `for..of` loop, where it loops over the values of an object, so it returns all the values of the properties in an object

```
for (arr of newArray) {
  console.log(arr);
}
```

- Using the built-in `Array.prototype.forEach`, using callback function to call value in an array

```
newArray.forEach(
  function(val) {
    console.log(val);
  }
);
```

18. What are instances where a collection is not iterable and instances where it is? If a collection is an array, then all 4 methods mentioned above can work. If a collection is an object with key-value pairs, then the `for` and `forEach` methods won't work.

19. What method is used to access properties on an object? The method `Object.getOwnPropertyDescriptors` is used to access the enumerability flag in an object. We can set an object property to `false` by setting `productDescriptors[configurable: false]`

In the below example, we can log values of each attribute based on their array position, or by attribute names.

Because you can't use `for...of` to retrieve the values, we pull out properties of each key (its value, writable status, enumerable status, configuratable status) by using the `Object.getOwnPropertyDescriptors` method and saving the output in an object.

```
var prices = [400, 80, 375, 870];
```

```
var products = {
  "widget": 400,
  "gear": 80,
  "crank": 375,
  "lever": 870,
};

var pricesDesc = Object.getOwnPropertyDescriptors(prices);
console.log(pricesDesc);

/*
{
  '0': { value: 400, writable: true, enumerable: true, configurable: true },
  '1': { value: 80, writable: true, enumerable: true, configurable: true },
  '2': { value: 375, writable: true, enumerable: true, configurable: true },
  '3': { value: 870, writable: true, enumerable: true, configurable: true },
  length: { value: 4, writable: true, enumerable: false, configurable: false }
}
*/

var proudctDesc = Object.getOwnPropertyDescriptors(products);
console.log(productDesc);

/*
{
  widget: { value: 400, writable: true, enumerable: true, configurable: true },
  gear: { value: 80, writable: true, enumerable: true, configurable: true },
  crank: { value: 375, writable: true, enumerable: true, configurable: true },
  lever: { value: 870, writable: true, enumerable: true, configurable: true }
}
*/
```

20. What can you do to skip over an property that's otherwise enumerable? How would this affect how the rest of the object is enumerated over?

A `for..in` loop can only iterate over properties with the flag `enumerable: true`. We can use `Object.defineProperty` to manually set the `enumerable: false`.

```
Object.defineProperty(prices, 1, {enumerable: false});

console.log(prices); //[ 400, 80, 375, 870 ], the array stays the same, there's no effect for the for ... of, for, or forEach method
console.log(Object.getOwnPropertyDescriptors(prices));
/*
{
  '0': { value: 400, writable: true, enumerable: false, configurable: true },
  '1': { value: 80, writable: true, enumerable: true, configurable: true },
  '2': { value: 375, writable: true, enumerable: true, configurable: true },
  '3': { value: 870, writable: true, enumerable: true, configurable: true },
  length: { value: 4, writable: true, enumerable: false, configurable: false }
}
*/
for (idx in prices) {
  console.log(prices[idx]);
}
 // 80 375 870, when enumerated over, it will print out only those that have the flag enumerable: false

console.log(Object.keys(products)); // [ 'widget', 'gear', 'crank', 'lever' ]
```

If we also explicitly skip over a specific property key, then the product object will skip over the property specified.

```
console.log(Object.keys(products)); // [ 'gear', 'crank', 'lever' ]
```

If we enumerate over the entire object and its key-value pair, the specified property will also be skipped over.

```
for (product in products) {
  console.log(product + ": " + products[product]);
}
```

21. What happens when you instantiate an object based on a pre-defined object and iterate over it?

Using the `for` loop, only the properties defined by the new object will be logged. But if we use the `for..in` loop, then all properties, including one from the prototype is logged.

```
var originalObject = {
  color: 'red',
  length: 10,
  weight: 50,
}
```

```
var newObject = Object.create(originalObject);

newObject['height'] = 20;
newObjectKeys = Object.keys(newObject);

for (i = 0; i < newObjectKeys.length; i++) {
  key = newObjectKeys[i];
  console.log(key + ": " + newObject[key]);
}
```

But using `for..in`, the results are a little different. The entire object, including those inherited from the prototype, shows up.

```
for (prop in newObject) {
  console.log(prop + ": " + newObject[prop]);
}
/*
height: 20
color: red
length: 10
weight: 50
*/
```

In order to avoid logging properties from the prototype, we need a conditional clause that only logs the properties of the object in question.

```
for (prop in newObject) {
  if (newObject.hasOwnProperty(prop)) {
    console.log(prop + ": " + newObject[prop]);
  }
} // logs only height: 20
```

22. What's the difference between `Array.length()` and `Object.keys(array).length`?

An array object is considered and object. When we log `Object.keys(array).length`, we count the number of keys in the object regardless of whether if has an index or not. But when we log `Array.length()`, only properties with an index gets counted.

```
var myArray = [];
console.log(typeof myArray); // object
myArray['foo'] = 'bar';
console.log(myArray); // [foo: 'bar']
myArray[0] = 0;
myArray[1] = 1;
console.log(myArray); // [0, 1, foo: 'bar']
console.log(myArray.length); // 2

for (arr in myArray) {
  console.log(arr); // 0 1 foo
}

console.log(myArray.indexOf('foo')); // -1
console.log(Object.keys(myArray).length); // 3
```

23. What happens when you manually set the length of the array to longer or shorter than what current exists? If an array's length is truncated by the `Array.length` method, then the array itself is shortened to the length specified. If it's lenghtened, then the array will simply have empty spots to take up the place.

Note that manually manipulating array lenghths will result in some differences when counting array length using the two different methods: `Array.length()` vs. `Object.keys().length`.

```
var newArr = [1, 2, 3];
newArr.length = 5;
console.log(newArr);

console.log(Object.keys(newArr).length); // 3
console.log(newArr.length); // 5
```

24. How are the following arithemic and comparison operators processed? What does a [] evaluate to, what about a {}, and if the {} is at the beginning of an expression?

[] evaluates to 0, {} evaluates to an object literal, which is `[object Object]`, if `{}` appears at the beginning of an expression, then `{}` is interpreted as a block of code instead of an object, and is usually ignored.

```
[] + {}; // '' + [object literal] = [object Object]
[] - {}; // '' - [object Object] = 0 - NaN = NaN
'[object Object]' == {}; // {} == {}, true
'' == {}; // 0 == {}, false
0 == {}; // 0 == {}, false

{} + []; // {} is ignored, so it becomes +[], which is 0
{}[0]; // the {} is again ignored, and it returns [0]
```

```
{ foo: 'bar' }['foo']; // {} is ignored, returns ['foo']
{} == '[object Object]' // =='[object Object]' evalautes to SyntaxError
'[object Object]' == {}; // true, [object Object] is {}
```

25. What's the difference between the == and === operators? == checks only for identical values, === checks for both identical type and value.

26. Demonstrate the immuatbility of a primitive type, and how a method can still temporarily return a mutated value of a primitive type without attempting to change the value of the type.

In the below example, the input string is temporarily changed as a return value when passed into a function, but the original string itself is not mutated.

```
function mutateGreeting(input) {
  return input.replace(/\.+/g, '!');
}

var originalGreeting = 'Hello.';
console.log(mutateGreeting(originalGreeting));
console.log(originalGreeting);
```

27. What would `typeof null` return? What about `typeof {}`, `typeof []`, `typeof function(){}`? `typeof null` is `object`, `typeof []` is `object`, as is `tyleof {}`. `typeof function()` is `function`.

28. What does it mean to say that JS is dynamically typed? It means that a `var`, `let` or `const` can be declared as one data type, and then dynamically changed to another throughout a program.

29. Does an array object equal to an object?

Both are objects. But an array object will also answer `true` to `Array.isArray(newArr)`, but an object literal, even though it will evaluate to an `object` like an array object, will evaluate `false` to `Array.isArray({})`.

```
var newArr = [];
console.log(typeof newArr); // object
console.log(Array.isArray(newArr)); // true
console.log(Array.isArray({})); // false
console.log(typeof {}); // object
console.log(typeof {} == typeof newArr); // true
console.log(Array.isArray({})); // false
```

30. What is strange about how `NaN` evaluates?

`NaN` will not evaluate to equal to itself, it evaluates `true` to the data type of `NaN`, but at the same time, it will also evaluate to `number` when ran against `typeof`.

Assignments and comparisons

31. What is the difference between attempting to mutate a primitive data type and re-assigning it? It's not possible to mutate a primitive data type, but re-assigning it (even to itself) will point to a different value (note the original value still exists, it just no longer points to the original variable name).

```
var stubbornVar = 'hello';

stubbornVar[0] = 'b';
console.log(stubbornVar);

stubbornVar = stubbornVar.concat('!');
console.log(stubbornVar);
```

32. Demonstrate through an example of implicit or explicit coersion.

If + is the operator and one of the arguments is a string, then the expression will act as a concatenation. If `*` is the operator, and one of the arguments is a number, then the expression will act as a `x` operator. If a boolean is joined by + and a `number` then it will be coersed into a number.

With explicit coersion, we can use `String()` or `Number()`.

Pure functions and side effects

33. What is a pure function, and what are different ways that a function is not pure? Pure function: the same return values are generated given the same arguments, and there will ALWAYS be a return value Functions with side effects: 1) when a higher scoped var is changed in value, 2) given the same arguments, different values can be returned, 3) returns the same value, but has side affects on argument passed in

Note that a function can be used to explicitly return values (pure function), or used to mutate arguments passed in (with side effects). It's fine if the intended effect is to change the argument passed in, but can always modify function so that it doesn't do so (e.g. it only returns the desired value without modifying the argument passed in).