JavaScript Theory Quiz

Take the quiz and find out how much you know about JavaScript!

A JavaScript interview often consists of fundamental JavaScript theory questions. Although it is not possible to cover everything about JavaScript, this post will give you an opportunity to test your knowledge.

You have probably seen that most tests contain questions that are a bit outdated. This test reflects 2017-18 standards, which means that you can, and sometimes you have to make use of your ES6 knowledge.

I suggest answering the questions one by one, so that you can grade yourself. Once you figure out your shortcomings, you can construct a learning plan for yourself.

Questions

Without looking at the solutions, you can access all ten questions at once here. I encourage you to solve the exercises on your own. Take out your favorite text editor or a piece of paper, and write down your answers. At first, do not use Google.

1. Explain scoping in JavaScript! Enumerate the different types of scopes you know. (6 points)
2. Explain hoisting with one or more examples including `var` and `let` variables. What is the temporal dead zone? (6 points)
3. Explain the role of the `prototype` property via an example! (5 points)
4. Extend your example from question 3 to demonstrate prototypal inheritance! (5 points)
5. Use the ES6 class syntax to rewrite the code you wrote in questions 3 and 4. (7 points)
6. Explain the `this` value in JavaScript! Illustrate your explanation with an example! (6 points)

7. Explain context binding using an example. (3 points)

8. Explain the difference between `==` and `===` in general. Determine the result of a comparison, when two values are:
   - of the same type (for all types)
   - `null` and `undefined`,
   - `NaN` to itself
   - 5 to `'5'` (6 points)
9. How can you check if a variable is an array? (2 points)
10. Suppose we would like to detect if a variable is an object using the following code. What can go wrong? How would you fix this error? (4 points)

```
if ( typeof x === 'object' ) {
    x.visited = true;
}
```

The maximum score is 50 points. Multiply your score by 2 to get your percentage. As you are not reading a meaningless horoscope test, I will not describe what each score range means to you. You can quickly figure it out yourself. Note though that this test is in strict mode, in most interviews, your interviewers are a lot more forgiving.

Once you are done with all the answers, start searching for relevant information that can improve your solutions. Make sure you write these enhancements with a different color, or in a separate document.

Verify your original solutions, the ones without using Google. Determine your score.

Verify your enhanced solutions. Have you missed anything?

Question 1: Scoping

Explain scoping in JavaScript! Enumerate the different types of scopes you know. (6 points)

Answer:

The scope of variables determines where you can access them in your code. (1 point)

Without considering ES6 modules, the scope can be global or local. (1 point)

In modules, a third scope is the module scope.

The global scope consists of global variables and constants that can be accessed from anywhere in your code. (1 point)

The local scope can be local to a function or a block. (1 point)

Variables defined using `var` inside a function have function scope. These variables are accessible/visible inside the function they are defined in. (1 point)

Variables and constants defined inside a block using `let` and `const` have block scope. They are accessible/visible inside the block they are defined in. (1 point)

When `var`, `let`, or `const` are used in global or module scope, their scope is going to be global or module.

Question 2: Hoisting

Explain hoisting with one or more examples including `var` and `let` variables. What is the temporal dead zone? (6 points)

Answer: In JavaScript, variable declarations are hoisted to the top of their scope. (1 point)

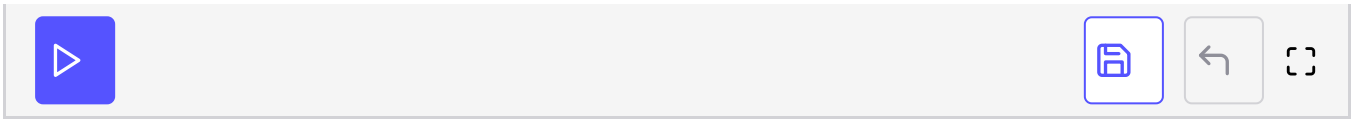
Both function and block-scoped variables are hoisted to the top of their scope. (1 point)

In the case of function-scoped variables, the value of a variable before its first assignment takes place is `undefined`. (1 point)

In case of a block scoped variable, its value is inaccessible before the intended location of its declaration. This is called the *temporal dead zone*. Accessing a variable in its temporal dead zone throws an error. (1 point)

Example for block scope: (1 point)

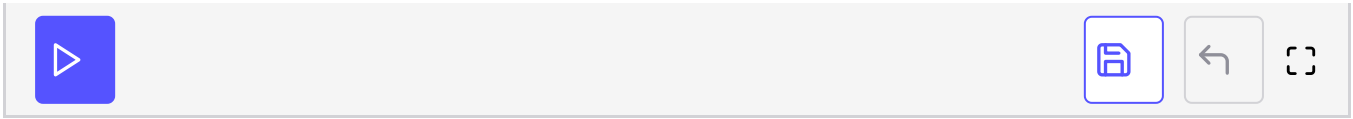
```
{
    console.log( x );
    let x = 3;
}
```



Uncaught ReferenceError: x is not defined

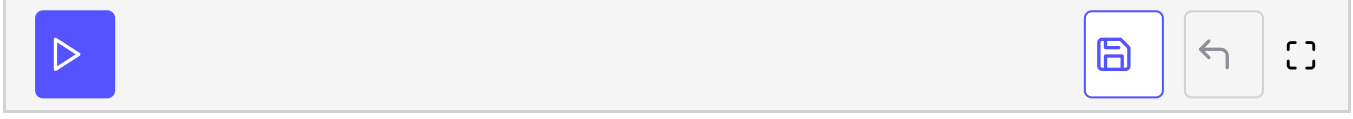
Reason:

```
{
    // let x; is hoisted to the top of the block
    console.log( x ); // temporal dead zone
    let x = 3;
}
```



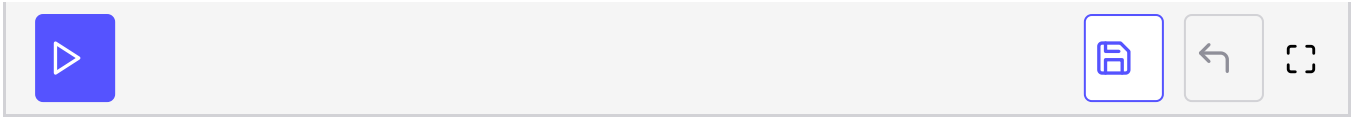
Tricky example:

```
{
    try { console.log( x ); } catch( _ ) { console.log( 'error' ); }
    let x;
    console.log( x );
    x = 3;
}
//> error
//> undefined
```



Reason:

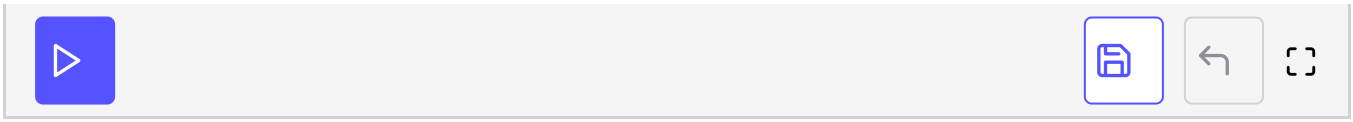
```
{
    // let x; // hoisting
    // start of temporal dead zone for x
    try { console.log( x ); } catch( _ ) { console.log( 'error' ); }
    // end of temporal dead zone for x
    // x = undefined; // at the point of its intended declaration
    console.log( x );
    x = 3;
}
```



Example for function scope: (1 point)

```
const f = function() {
    // var x; is hoisted to the top
    console.log( x );
    var x = 3;
}

f();
//> undefined
```



If you need some more information on these concepts, check out the first few chapters of this book.

Question 3: Prototypes

Explain the role of the `prototype` property via an example! (5 points)

Answer: JavaScript functions defined using the ES5 syntax have prototypes. (1 point)

Remark: ES6 arrow functions don't have prototypes. Methods defined using the concise method syntax don't have prototypes.

These prototypes become important once a function is used for instantiation. In JavaScript terminology, these functions are *constructor functions*. (1 point)

A prototype may contain functions that are available in every instance created by the constructor functions. (1 point)

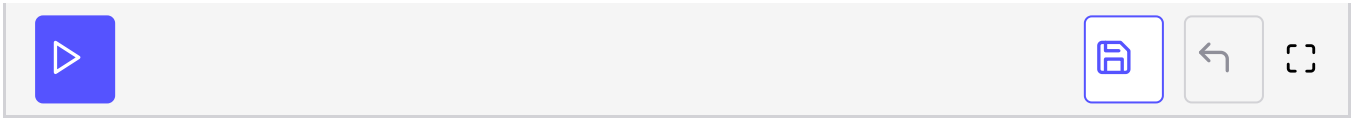
Example: (2 points)

```
function Wallet() {
    this.amount = 0;
}

Wallet.prototype.deposit = function( amount ) {
    this.amount += amount;
}
Wallet.prototype.withdraw = function( amount ) {
```

```
    if ( this.amount >= amount ) {
        this.amount -= amount;
    } else {
        throw 'Insufficient funds.';
    }
}

let myWallet = new Wallet();
myWallet.deposit( 100 );
console.log(myWallet.amount)
//> 100
```



Question 4: Prototypal inheritance

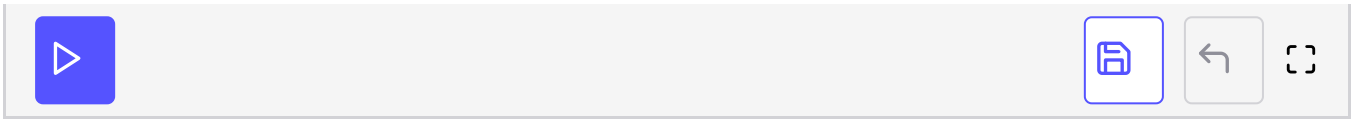
Extend your example from question 3 to demonstrate prototypal inheritance! (5 points)

Answer:

```
function BoundedWallet( maxAmount ) {
    Wallet.call( this );                // 1 point
    this.maxAmount = maxAmount;         // 1 point
}

BoundedWallet.prototype = Object.create( Wallet.prototype ); // 1 point
BoundedWallet.prototype.constructor = BoundedWallet;         // 1 point

BoundedWallet.prototype.deposit = function( amount ) {
    if ( this.amount + amount > this.maxAmount ) {
        throw 'Insufficient wallet capacity';
    }
    Wallet.prototype.deposit.call( this, amount );  // this.amount += amount;
} // 1 point
```



One construct worth mentioning is `Wallet.prototype.deposit.call( this, amount );`. Here, instead of writing `this.amount += amount;`, it is semantically more correct to reuse the `deposit` functionality of the base class. In ES5, this is the way to go.

Although the rest of the code is not self-explanatory, we have already covered a similar exercise in Chapter 4 on Classes.

Question 5: ES6 classes

Use the ES6 class syntax to rewrite the code you wrote in questions 3 and 4. (7 points)

Answer:

```
class Wallet {                              // 1 point
    constructor() { this.amount = 0; }      // 1 point
    deposit( amount ) { this.amount += amount; }
    withdraw( amount ) {
        if ( this.amount >= amount ) {
            this.amount -= amount;
        } else {
            throw 'Insufficient funds.';
        }
    }                                       // 1 point
}

class BoundedWallet extends Wallet {        // 1 point
    constructor( maxAmount ) {
        super();                            // 1 point
        this.maxAmount = maxAmount;
    }
    deposit( amount ) {
        if ( this.amount + amount > this.maxAmount ) {
            throw 'Insufficient wallet capacity';
        }
        super.deposit( amount );
    }                                       // 1 point
}

let myWallet = new Wallet();
myWallet.deposit( 100 );
console.log(myWallet.amount)                            // 1 point
//> 100
```

You need to demonstrate the following six items for a complete solution:

- `class` keyword, class name, and braces to define a constructor function
- a properly working constructor using the *concise method syntax*
- at least one method using the *concise method syntax*
- proper usage of the `extends` keyword
- proper usage of `super` in the constructor of the child class
- at least one redefined method. It is optional to access the shadowed base class method using super. In this example, `super.deposit(amount);`

accessed the deposit method of the `Wallet` class.

- some code demonstrating instantiation, which should be unchanged compared to the ES5 prototypal inheritance syntax

Question 6: this

Explain the `this` value in JavaScript! Illustrate your explanation with an example! (6 points)

Answer: In JavaScript, `this` is a global or function scoped variable. (1 point)

When used in global scope, `this` equals the global object, which is `window` in the browser. (1 point)

When used inside a function, the value of `this` is dynamically determined when the function is called and its value equals the context of the function. (2 points)

Example:

```
class C { f() { return this; } }
class D extends C {}

const d = new D();
console.log( d.f() === d );
//> true
```

Here, `f()` was inherited with prototypal inheritance from class `C`. When creating the `d` object using the class (constructor function) `D`, any method of `d` gets `this` assigned to `d`. (2 points)

The value of `this` can be changed using *context binding*. (1 point)

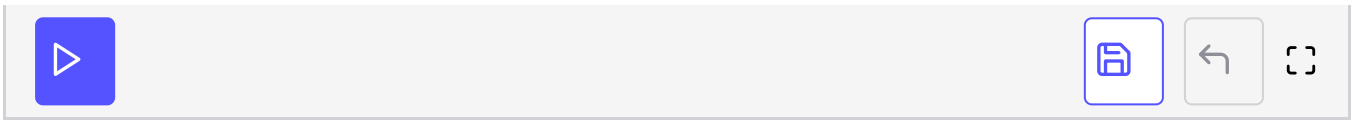
Question 7: context binding

Explain context binding using an example. (3 points)

Answer:

```
let f = function() {
    console.log( this );
}

f.bind( {a: 5} )();
//> {a: 5}
//( 2 points)
```



`f.bind({a: 5})` creates a function, where the value of `this` becomes `{a: 5}`. (1 point)

Question 8: Truthiness

Explain the difference between `==` and `===` in general. Determine the result of a comparison, when two values are:

- of the same type (for all types)
- `null` and `undefined`,
- `NaN` to itself
- 5 to `'5'`
- `Symbol.for( 'a' )` to `Symbol.for( 'a' )` (6 points)

Answer: `==` converts its operands to the same type, while `===` is type safe, and is only equal when the types and values are equal. (1 point)

There are six primitive datatypes in JavaScript: boolean, null, undefined, number, string, and symbol.

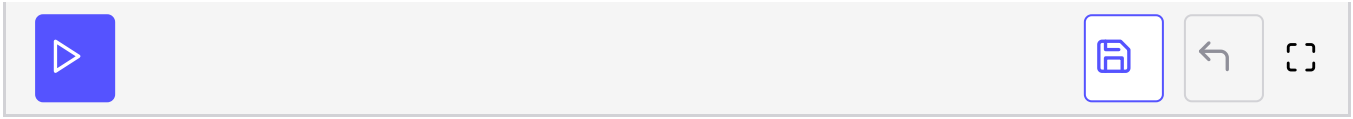
For null values, undefined values, booleans, strings, and numbers except `NaN`, `a == b` and `a === b` yield the same result, which is `true` whenever the values are the same. (1 point, you may miss the `NaN` exception)

For `NaN`, `NaN == NaN` and `NaN === NaN` is `false`. (1 point)

When it comes to arrays and objects, both `==` and `===` compare the references. So, for instance,

```
let a = [], b = [], c = a;
```

```
console.log(a == b)// or a === b
//false
console.log(a == c)   // or a === c
//true
//(1 point)
```

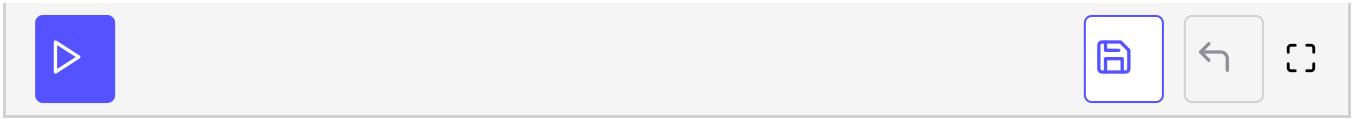


The `null == undefined` comparison is `true` by definition. `null === undefined` is false. You may argue why this is worth a point. In practice, you must have encountered this case during writing code. If you are unsure about this comparison, this may indicate a lack of hands-on experience. (1 point)

`5 === '5'` is false, because the types don't match. In case of `5 == '5'`, the string operand is converted to a number. As the number values match, the result is true. (1 point)

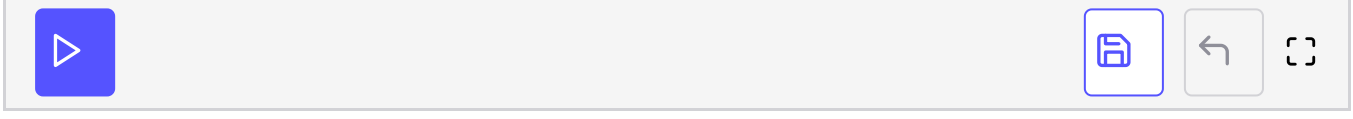
Whenever a `Symbol` is created, they are always unique. Therefore,

```
console.log(Symbol() == Symbol())
//false
```



When using the global symbol registry, we get the same symbol belonging to the string `'a'` whenever we call `Symbol.for( 'a' )`. Therefore,

```
console.log(Symbol.for( 'a' ) === Symbol.for( 'a' ))  // or ==
//true
//(1 point)
```



Don't confuse `Symbol.for( 'a' )` with `Symbol( 'a' )`. Only the former accesses the global symbol registry. The latter is a mere label associated with the symbol, and different symbols may have the same label:

```
console.log(Symbol( 'a' ) == Symbol( 'a' ))
//false
console.log(Symbol.for( 'a' ) == Symbol( 'a' ))
```

```
//false
```

Question 9: Arrays

How can you check if a variable is an array? (2 points)

Answer:

`Array.isArray( x )` gives you a `true` result if and only if `x` is an array. (2 points)

If you don't know the above check, you can still invent your own array type checker function. This function may not be as correct as `Array.isArray`, because it may not work in exceptional cases.

The check `x.constructor === Array` is incorrect and is worth zero points, because we could extend the Array class:

```
class MyArray extends Array {}
let x = new MyArray();

console.log(Array.isArray( x ))
//> true

console.log(x.constructor === Array)
//> false
```

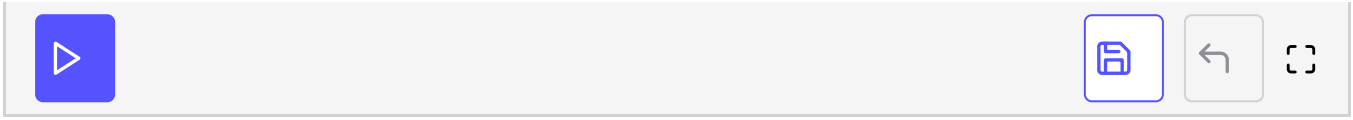
You might have used this check in your code before, so if you can recall the `'[object Array]'` value of the `Object.prototype.toString` function applied on an array; then you definitely deserve the points.

```
console.log(Object.prototype.toString.call( [] ))
//"[object Array]"

console.log(Object.prototype.toString.call( x ) === "[object Array]")
```

An almost correct check is the usage of the `instanceof` operator:

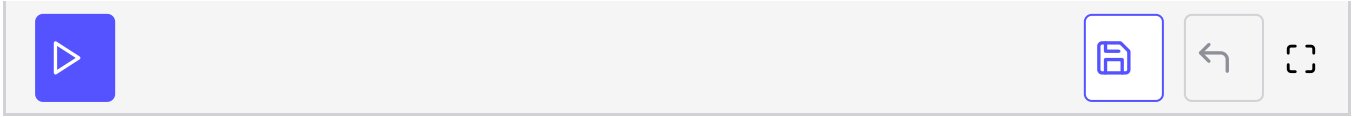
```
console.log(x instanceof Array)
//true
```



In an interview setting, I would accept the `instanceof` solution, because it is generally not expected to look up edge cases described by e.g. this StackOverflow article, where `instanceof` gives a different result than `Array.isArray`.

In case the solution mentions that it is not perfect, a thorough definition using the `typeof` operator is worth 1 point:

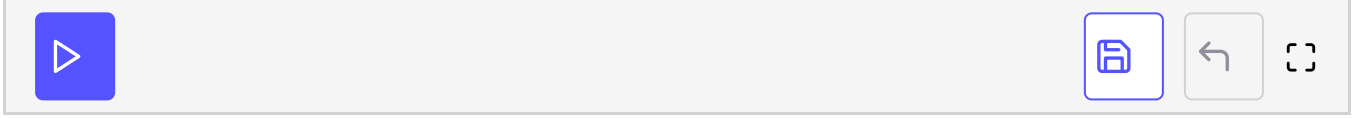
```
typeof x === 'object' &&
typeof x.length === 'number' &&
typeof x.push === 'function'
```



Question 10: Objects

Suppose we would like to detect if a variable is an object using the following code:

```
if ( typeof x === 'object' ) {
    x.visited = true;
}
```



What can go wrong? How would you fix this error? (4 points)

Answer:

`typeof null` is also an object. (1 point)

The value `null` cannot have properties; so the code will crash with an error. (1 point)

We have to add a null check in the condition. (1 point)

```
if ( x !== null && typeof x === 'object' ) {
    x.visited = true;
}
```

If you prefer not including arrays, you can use your array checker from question 8 to exclude arrays. (1 point)