
Session 16

Mohamed Emary

May 5, 2024

Important Note

This session discusses a lot of important and tricky concepts that gets asked frequently in interviews. So, make sure you understand the concepts well.

Implicit Conversion

Implicit conversion is the automatic conversion of a value from one data type to another. This is done by the JavaScript engine when the data type of the operands in an expression are different.

For example when you add a number and a string, JavaScript will convert the number to a string and concatenate the two strings.

```
1 var a = 20;
2 var b = "10";
3 var c = a + b;
4 console.log("c = " + c);
5 var f = a - b;
6 console.log("f = " + f);
7 var d = a * b;
8 console.log("d = " + d);
9 var e = a / b;
10 console.log("e = " + e);
```

```
c = 2010
f = 10
d = 200
e = 2
```

As you see the results below the code:

- **c** equals **2010** because **a** is converted to a string and concatenated with **b**.
- **f** equals **10** because **b** is converted to a number and subtracted from **a**.
- **d** equals **200** because **b** is converted to a number and multiplied by **a**.
- **e** equals **2** because **b** is converted to a number and divided by **a**.

But what if the value can't be converted to the desired data type? In this case, the result will be **NaN** (Not a Number).

IMPLICIT CONVERSION

```
1 var a = 20;
2 var b = "Hello";
3 var c = a - b;
4 console.log("c = " + c);
```

c = NaN

The result will be NaN because the string Hello can't be converted to a number to be subtracted from a.

Lets see the result of string to number conversion of some strings.

```
1 console.log( 'true = ' + Number(true) )
2 console.log( 'false = ' + Number(false) )
3 console.log( '"0" = ' + Number('0') )
4 console.log( 'null = ' + Number(null) )
5 console.log( 'undefined = ' + Number(undefined) )
6 console.log( '" " = ' + Number(' ') )
7 console.log( 'NaN = ' + Number(NaN) )
```

```
true = 1
false = 0
"0" = 0
null = 0
undefined = NaN
" " = 0
NaN = NaN
```

- true will be converted to 1 and false will be converted to 0.
- Since the string 0 can be converted to a number, the result will be 0.
- null, " " will be converted to 0.
- undefined will be converted to NaN.
- NaN stays as NaN.

What about adding these values to a number?

```
1 console.log( 'true + 1 = ' + (true + 1) )
2 console.log( 'false + 1 = ' + (false + 1) )
3 console.log( 'null + 1 = ' + (null + 1) )
4 console.log( 'undefined + 1 = ' + (undefined + 1) )
5 console.log( 'NaN + 1 = ' + (NaN + 1) )
```

```
true + 1 = 2
false + 1 = 1
null + 1 = 1
undefined + 1 = NaN
NaN + 1 = NaN
```

What about adding these values to a string?

```
1 console.log( 'true + "1" = ' + (true + "1") )
2 console.log( 'false + "1" = ' + (false + "1") )
3 console.log( 'null + "1" = ' + (null + "1") )
4 console.log( 'undefined + "1" = ' + (undefined + "1") )
5 console.log( 'NaN + "1" = ' + (NaN + 1) )
```

FUNCTION

```
true + "1" = true1
false + "1" = false1
null + "1" = null1
undefined + "1" = undefined1
NaN + "1" = NaN
```

Lets try some extra examples.

```
1 | console.log( 'true + true = ' + (true + true))
2 | console.log( 'true + false = ' + (true + false))
3 | console.log( 'false + false = ' + (false + false))
4 | console.log( 'true + null = ' + (true + null))
5 | console.log( 'true + undefined = ' + (true + undefined))
6 | console.log( 'true + NaN = ' + (true + NaN))
```

```
true + true = 2
true + false = 1
false + false = 0
true + null = 1
true + undefined = NaN
true + NaN = NaN
```

Some extra examples:

```
1 | console.log( '"3" * false = ' + ("3" * false))
2 | console.log( '"3" * true = ' + ("3" * true))
3 | console.log( '"3" / false = ' + ("3" / false))
4 | console.log( '"3" / true = ' + ("3" / true))
```

```
"3" * false = 0
"3" * true = 3
"3" / false = Infinity
"3" / true = 3
```

Function

Functions are reusable blocks of code that can be called multiple times. They can take parameters and return values.

Functions are defined using the `function` keyword followed by the function name and a list of parameters in parentheses. The function body is enclosed in curly braces `{}`.

Functions parameters are created without using `var` keyword.

```
1 | function FunctionName (parameter1, parameter2, ...) {
2 |     // function body
3 | }
```

And to use the function you have to call it, just write the function name followed by parentheses and any arguments you want to pass to the function.

The arguments are the values of the parameters that the function will use.

Example:

FUNCTION

```
1 function greet(name) {  
2     console.log("Hello " + name);  
3 }  
4 greet("World");
```

Hello World

The `greet("World")` is called a function call or invocation. The value inside the parentheses is called an argument.

For reusability purposes, it's preferred that each function does only one thing, this is called the single responsibility principle.

Functions can access already defined variables in the global scope.

```
1 var name = "World";  
2 function greet() {  
3     console.log("Hello " + name);  
4 }  
5 greet();
```

Hello World

Return Value

Functions can return a value using the `return` keyword followed by the value to return.

```
1 function add(a, b) {  
2     return a + b;  
3 }  
4 var result = add(10, 20);  
5 console.log("result = " + result);
```

result = 30

Any code after the `return` statement will not be executed.

The function that doesn't have a `return` statement will return `undefined`.

```
1 function greet(name) {  
2     console.log("Hello " + name);  
3 }  
4 var result = greet("World");  
5 console.log("result = " + result);
```

Hello World

result = undefined

The first output line here `Hello World` is because of the `console.log` inside the function, and the second output line `result = undefined` is because the function doesn't have a `return` statement.

A return function is the function that have a return statement.

Return is used when you want to get a value from a function to use it in another part of the code.

You can even return another function.

SOME INTERVIEW NOTES

Function Types

Declaration Function

This is the most common way to define a function. It is defined using the `function` keyword followed by the function name.

Declaration functions always start with the `function` keyword.

```
1 | function greet (name) {  
2 |     console.log("Hello " + name);  
3 | }  
4 | greet("World");
```

Hello World

Expression Function

This is when you assign a function to a variable.

```
1 | var greet = function (name) {  
2 |     console.log("Hello " + name);  
3 | }  
4 | greet("World");
```

Hello World

In this example, the function is assigned to the `greet` variable and then called using the variable.

When you print the value of the `greet` variable you will get the function definition.

```
1 | console.log(greet);
```

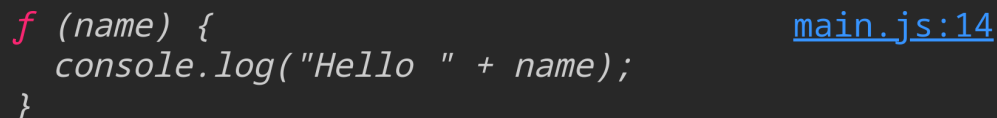


Figure 1: Console Output

User Input

To get a value from user you can use the `prompt` function. This function will show a dialog box to the user with a message and an input field, and it will return the value entered by the user.

```
1 | var name = prompt("Enter your name");  
2 | console.log("Hello " + name);
```

This code will make a dialog box appear with the message “Enter your name” and an input field. The value entered by the user will be stored in the `name` variable then it will be printed to the console.

Some Interview Notes

What will be the result of this code?

HOISTING WITH FUNCTION TYPES

```
1 function salaryBonus(salary) {  
  ↪  
2   console.log(salary + 100);  
3 }  
4 salaryBonus();  
5 salaryBonus(100);  
6 salaryBonus(100, 200);
```

The results will be:

1. NaN because salary is not defined so undefined + 100 is NaN.
2. 200 because salary is 100.
3. 200 because salary is 100 and 200 is ignored.

Hoisting

JavaScript hoisting is a mechanism where variable and function declarations are moved to the top of their containing scope during the compilation phase.

That is the reason why when you print the value of a variable before declaring it, you will get undefined instead of an error.

```
1 console.log(a);  
2 var a = 10;
```

undefined

But if you just print the value of a variable without declaring it, you will get an error.

```
1 console.log(a);
```

With the code above you will get: ReferenceError: a is not defined

Since hoisting also works with functions, you can call a function before declaring it.

```
1 greet("World");  
2 function greet(name) {  
3   console.log("Hello " + name);  
4 }
```

Hello World

These weird behaviors happen because of hoisting.

You should notice that only the declaration is hoisted not the initialization, so if you have var a = 10; the var a; part will be hoisted but the a = 10; part will not and that is why you get undefined when you print the value of a.

So these are equivalent and both will print undefined.

```
1 console.log(a);  
2 var a = 10;
```

undefined

```
1 var a;  
2 console.log(a);  
3 a = 10;
```

undefined

Hoisting With Function Types

Declaration Functions Hoisting

Declaration functions get hoisted so you can call the function before declaring it.

Expression Functions Hoisting

Expression functions don't get hoisted since only the variable declaration gets hoisted and not the initialization (which is the function definition)

For example in:

```
1 | var greet = function(name) {  
2 |     console.log("Hello " + name);  
3 | }
```

only `var greet;` gets hoisted.

so you can't call an expression function before the line it was assigned to the variable. In other words to make a function that can only be called after its definition use expression function.

Scope

Scope is the context in which a variable is defined. JavaScript has two types of scope: **global scope** and **local scope**.

Global Scope

A variable is in the global scope if it's declared outside of any function or block.

```
1 | var a = 10;  
2 | function getNum() {  
3 |     console.log(a);  
4 | }  
5 | getNum();
```

In this example, the variable `a` is declared in the global scope and **can be accessed** from the `getNum` function.

Local Scope

A variable is in the local scope if it's declared inside a function or block.

```
1 | function getNum() {  
2 |     var a = 10;  
3 | }  
4 | console.log(a);
```

In this example, the variable `a` is declared in the local scope of the `getNum` function and **can't be accessed** from outside the function, so you will get `ReferenceError: a is not defined`.

Inside a function you can access the variables of the global scope but in the global you can't access the variables of the function local scope.

The normal behavior assumes that each `{ }` creates a new scope but this is not the case when using `var` keyword, so if you create a loop or an if condition for example and declared variables inside their `{ }` using `var` keyword, you still can access these variables from outside the `{ }`, and the only exception is the function scope. The same happens with whole functions if you declared a function inside another function you can only access it from inside the function.

Questions

What will be the output of each of these codes?

Code 1

```
1 var a = 10;
2 function myFunction() {
3     var a = 20;
4     console.log(a);
5 }
6 myFunction();
7 console.log(a);
```

20

10

The output of the function call will be 20 and the output of the second `console.log` will be 10 because the function accessed the variable in the local scope which had the value 20 while the `console.log` accessed the variable in the global scope which had the value 10.

Code 2

```
1 var a = 10;
2 function myFunction() {
3     console.log(a);
4     a = 20;
5 }
6 myFunction();
7 console.log(a);
```

10

20

The output of the function call will be 10 and the output of the second `console.log` will be 20 because the function accessed the variable in the global scope which had the value 10 and changed it to 20 then the `console.log` accessed the same variable after it was changed.

Code 3

```
1 var a = 10;
2 function myFunction() {
3     console.log(a);
4     var a = 20;
5 }
6 myFunction()
7 console.log(a);
```

undefined

10

The output of the function call will be `undefined` and the output of the second `console.log` will be 10 because the function accessed the variable in the local scope which was hoisted to the top of the function but not initialized yet so it was `undefined` while the `console.log` accessed the variable in the global scope which had the value 10.

Notice that the variable inside the function gets hoisted to the top of the function not to the top of the global scope.

Code 4

```
1 function foo() {  
2     function bar() { return 3; }  
3     function bar() { return 8; }  
4     return bar();  
5 }  
6 console.log(foo())  
8
```

The output will be 8 because the second function definition will override the first one.

Code 5

```
1 function foo() {  
2     var bar = function() { return 3; }  
3     return bar();  
4     var bar = function() { return 8; }  
5 }  
6 console.log(foo())  
3
```

The output will be 3 because the second function definition will be ignored because it's an expression function that is not hoisted so only the variable declaration will be hoisted and the value of the variable will be the first function.

Code 6

```
1 function foo() {  
2     function bar() { return 3; }  
3     return bar();  
4     function bar() { return 8; }  
5 }  
6 console.log(foo())  
8
```

The output will be 8 because both functions are declaration functions so both will be hoisted (in the same order they were defined) and the second function will override the first one.

Its equivalent to this code:

```
1 function foo() {  
2     function bar() { return 3; }  
3     function bar() { return 8; }  
4     return bar();  
5 }  
6 console.log(foo())
```

Code 7

```
1 function foo() {  
2     return bar();
```

SELF INVOKED FUNCTION

```
3   function bar() { return 3; }
4   var bar = function() { return 8; }
5 }
6 console.log(foo())
```

3

The output will be 3 because the declaration function definition will be hoisted to the top before the return statement while in the expression function definition only the variable declaration will be hoisted but not the function assigned to it, you should also know that the value returned here is the result of the function call since we are using `()` after the function name but if we remove the `()` the value returned will be the function definition similar to 1.

The code above is equivalent to this code which will also print 3:

```
1 function foo() {
2     function bar() { return 3; }
3     var bar;
4     return bar();
5     bar = function() { return 8; }
6 }
7 console.log(foo())
```

Code 8

```
1 console.log(foo())
2 var foo = function() {
3     return bar();
4     function bar() { return 3; }
5     var bar = function() { return 8; }
6 }
```

The output will be `TypeError: foo is not a function` because the variable `foo` is hoisted to the top but not initialized yet so its value is `undefined` and you can't call `undefined` as a function. (if you `console.log(foo)` you will get `undefined`)

Self Invoked Function

It's also called IIFE (Immediately Invoked Function Expression). It's a function that is called immediately after it's defined.

Syntax:

```
1 (function() {
2     // function body
3 })();
```

We use IIFEs to create a new scope for our code so we can use variables without polluting the global scope.