JavaScript Day - 9

JSON, Rest parameter and Spread syntax

What is JSON ?

JSON is a data format that is easy to read and write, and it is commonly used for data exchange between applications and APIs.

JSON Methods:

1. JSON.stringify - to convert object to JSON
2. JSON.parse - to convert JSON back into object

JSON.stringify

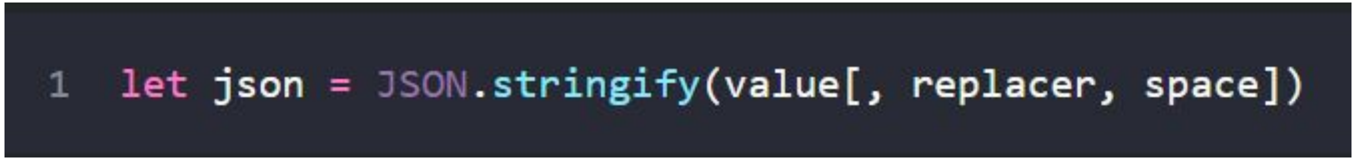
The method JSON.stringify(student) takes the object and converts it into a string.

The resulting json string is called a JSON-encoded or serialized or stringified or marshalled object. We are ready to send it over the wire or put into a plain data store.

JSON is data-only language-independent specification, so some JavaScript-specific object properties are skipped by JSON.stringify.

Namely:

- Function properties (methods).
- Symbolic keys and values.
- Properties that store undefined.



```
let json = JSON.stringify(value[, replacer, space])
```

JSON.stringify

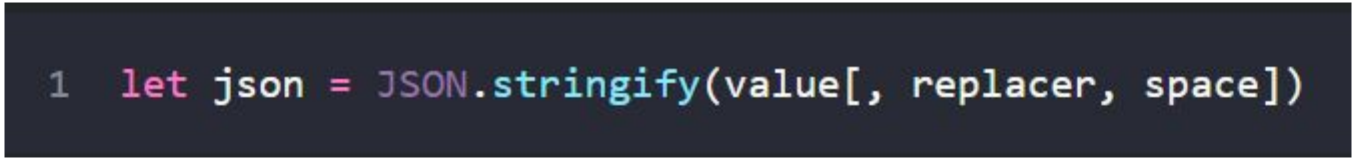
value: A value to encode.

replacer: Array of properties to encode or a mapping function function(key, value).

space: Amount of space to use for formatting

Most of the time, JSON.stringify is used with the first argument only. But if we need to fine-tune the replacement process, like to filter out properties we can use the second argument of JSON.stringify.

If we pass an array of properties to it, only these properties will be encoded.



```
let json = JSON.stringify(value[, replacer, space])
```

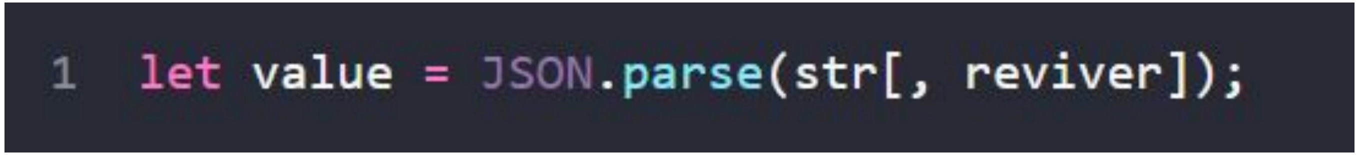
JSON.parse

To decode a JSON-string, we need another method named JSON.parse

str: JSON-string to parse.

reviver: Optional function(key,value) that will be called for each (key, value) pair and can transform the value.

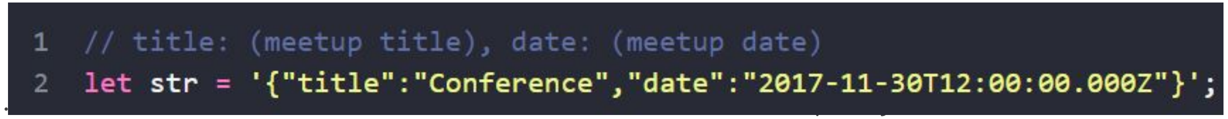
.



```
let value = JSON.parse(str[, reviver]);
```

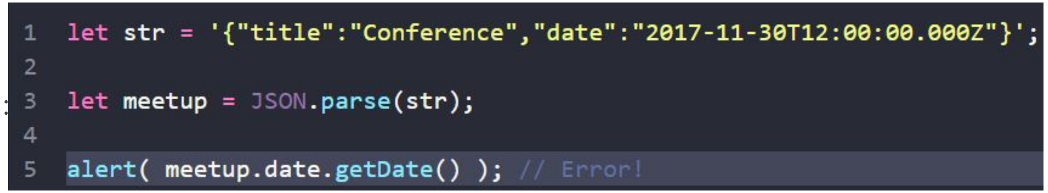
Using reviver

Imagine, we got a stringified meetup object from the server.



```
// title: (meetup title), date: (meetup date)
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';
```

Let’s do it by calling JSON.parse



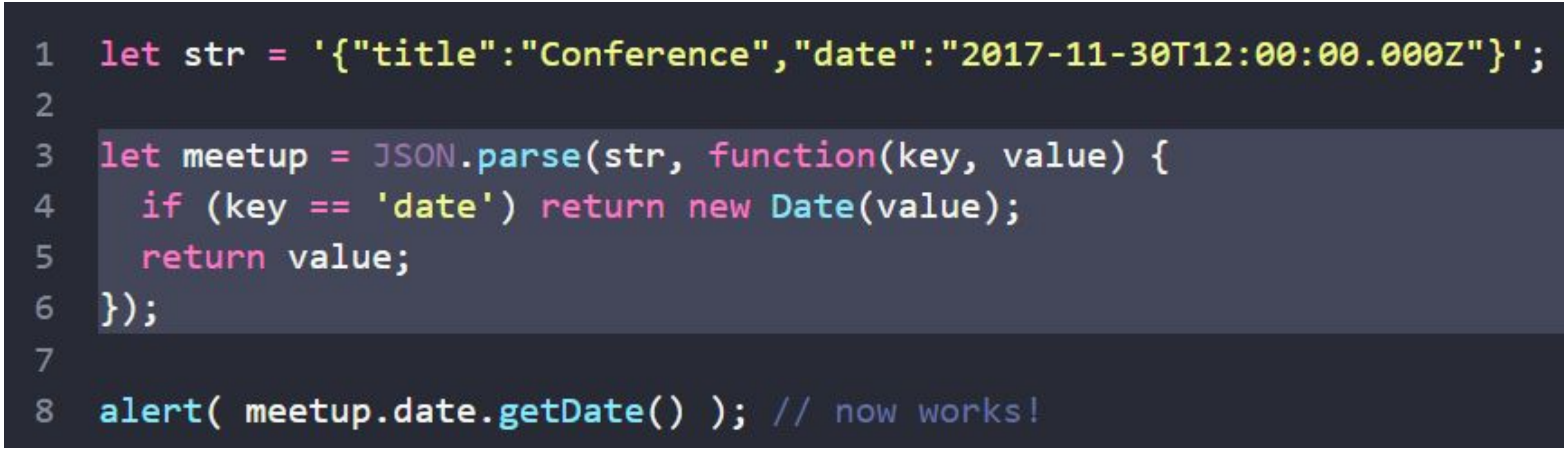
```
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str);

alert( meetup.date.getDate() ); // Error!
```

Whoops! An error!

The value of meetup.date is a string, not a Date object. How could JSON.parse know that it should transform that string into a Date?

Using reviver

Let’s pass to JSON.parse the reviving function as the second argument, that returns all values “as is”, but date will become a Date:



```javascript
let str = '{"title":"Conference","date":"2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});

alert( meetup.date.getDate() ); // now works!
```

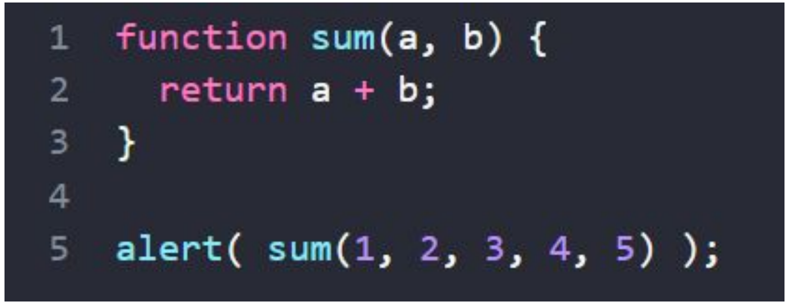
Summary

- JSON is a data format that has its own independent standard and libraries for most programming languages.
- JSON supports plain objects, arrays, strings, numbers, booleans, and null.
- JavaScript provides methods JSON.stringify to serialize into JSON and JSON.parse to read from JSON.
- Both methods support transformer functions for smart reading/writing.
- If an object has toJSON, then it is called by JSON.stringify.

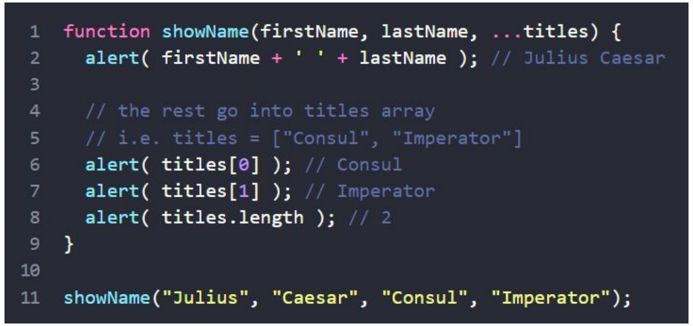
Rest Parameter and Spread Syntax

Rest parameter

A function can be called with any number of arguments, no matter how it is defined.



```
function sum(a, b) {
  return a + b;
}

alert( sum(1, 2, 3, 4, 5) );
```



```
function showName(firstName, lastName, ...titles) {
  alert( firstName + ' ' + lastName ); // Julius Caesar

  // the rest go into titles array
  // i.e. titles = ["Consul", "Imperator"]
  alert( titles[0] ); // Consul
  alert( titles[1] ); // Imperator
  alert( titles.length ); // 2
}

showName("Julius", "Caesar", "Consul", "Imperator");
```

The rest of the parameters can be included in the function definition by using three dots … followed by the name of the array that will contain them. The dots literally mean “gather the remaining parameters into an array”.

The rest parameter must be at the end

Spread Syntax

We’ve just seen how to get an array from the list of parameters.

But sometimes we need to do exactly the reverse.

For instance, there’s a built-in function Math.max that returns the greatest number from a list:

Now let’s say we have an array [3, 5, 1]. How do we call Math.max with it?

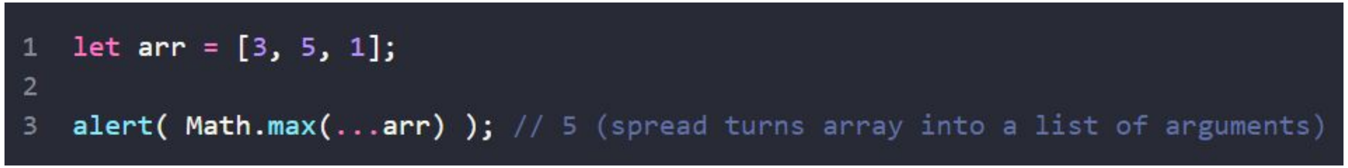
Passing it “as is” won’t work, because Math.max expects a list of numeric arguments, not a single array:

Spread syntax to the rescue! It looks similar to rest parameters, also using ..., but does quite the opposite.

Spread Syntax

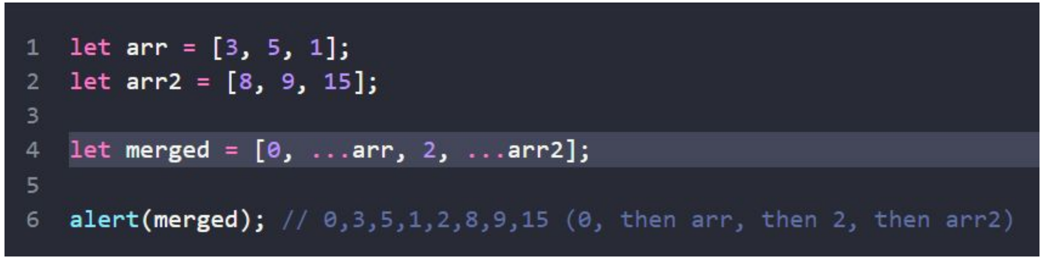
When ...arr is used in the function call, it “expands” an iterable object arr into the list of arguments.

For Math.max:



```javascript
let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5 (spread turns array into a list of arguments)
```

Also, the spread syntax can be used to merge arrays:



```javascript
let arr = [3, 5, 1];
let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];

alert(merged); // 0,3,5,1,2,8,9,15 (0, then arr, then 2, then arr2)
```

Summary

When we see "..." in the code, it is either rest parameters or the spread syntax.

There’s an easy way to distinguish between them:

- When ... is at the end of function parameters, it’s “rest parameters” and gathers the rest of the list of arguments into an array.
- When ... occurs in a function call or alike, it’s called a “spread syntax” and expands an array into a list.

Use patterns:

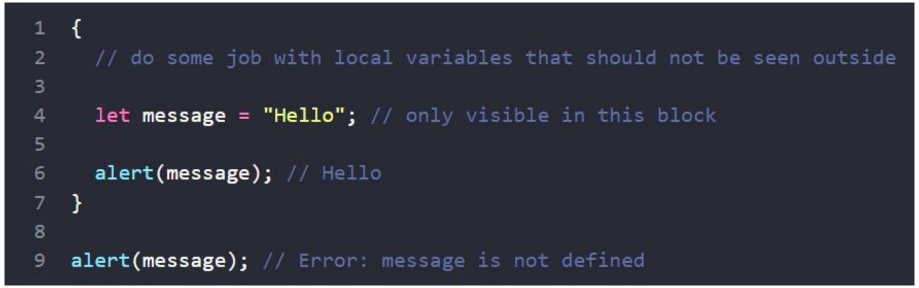
- Rest parameters are used to create functions that accept any number of arguments.
- The spread syntax is used to pass an array to functions that normally require a list of many arguments.

Together they help to travel between a list and an array of parameters with ease.

All arguments of a function call are also available in “old-style” arguments: array-like iterable object.

Variable scope and closure

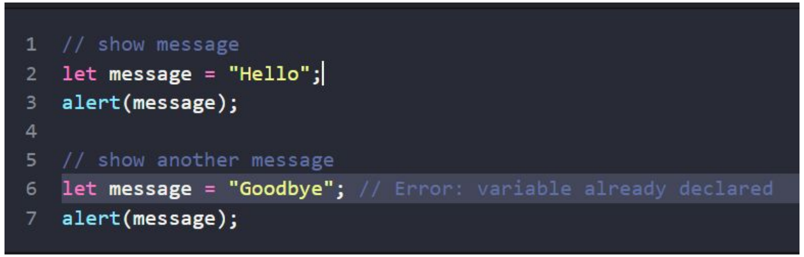
Code Block



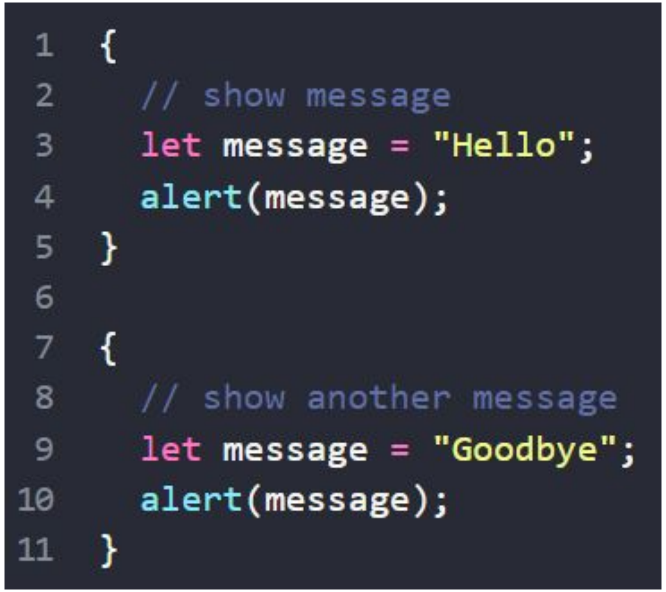
```
{
  // do some job with local variables that should not be seen outside

  let message = "Hello"; // only visible in this block

  alert(message); // Hello
}

alert(message); // Error: message is not defined
```

If a variable is declared inside a code block {…}, it's only visible inside that block.

We can use this to isolate a piece of code that does its own task, with variables that only belong to it:



```
// show message
let message = "Hello";
alert(message);

// show another message
let message = "Goodbye"; // Error: variable already declared
alert(message);
```



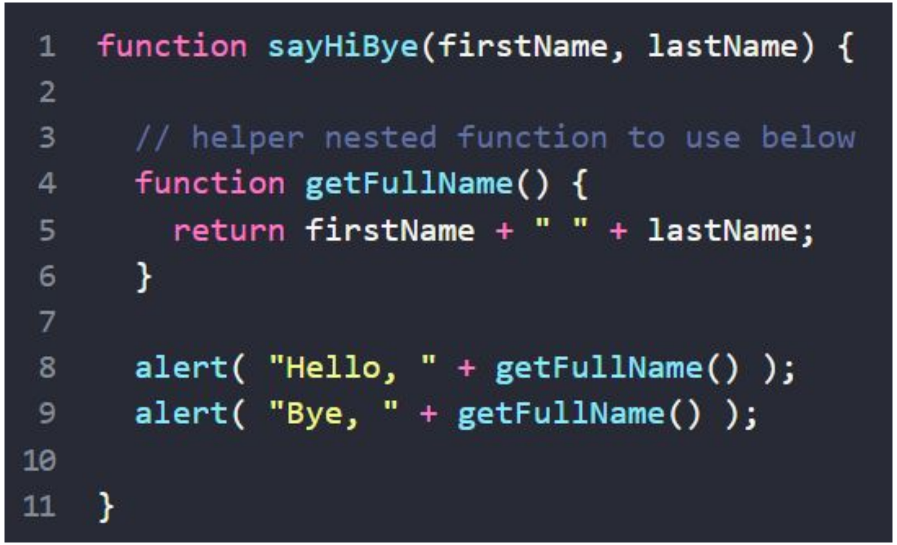
```
{
  // show message
  let message = "Hello";
  alert(message);
}

{
  // show another message
  let message = "Goodbye";
  alert(message);
}
```

Nested Function

A function is called “nested” when it is created inside another function.

It is easily possible to do this with JavaScript.

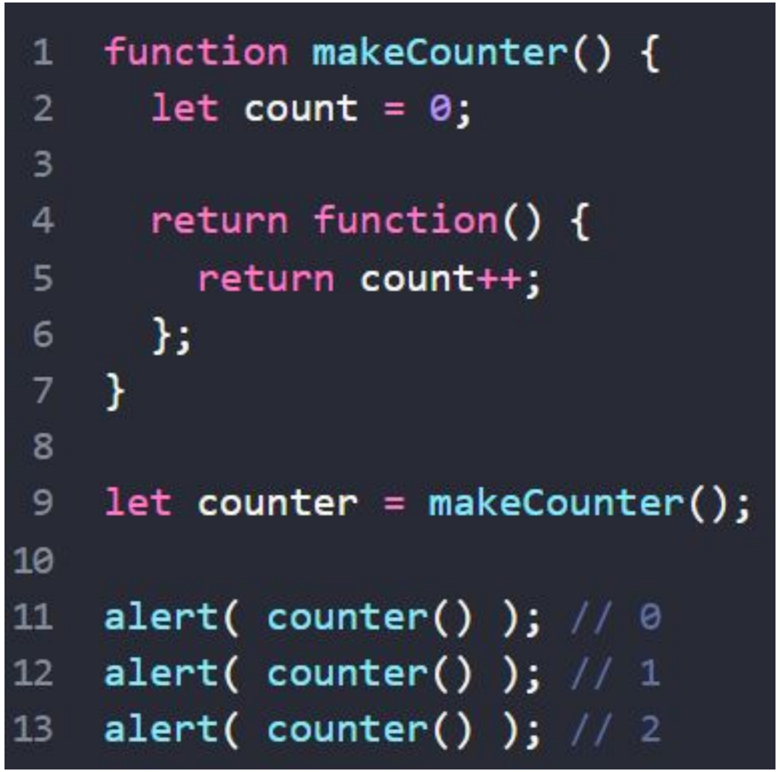
We can use it to organize our code, like this:



```
function sayHiBye(firstName, lastName) {

  // helper nested function to use below
  function getFullName() {
    return firstName + " " + lastName;
  }

  alert( "Hello, " + getFullName() );
  alert( "Bye, " + getFullName() );

}
```

Nested Function

What’s much more interesting, a nested function can be returned: either as a property of a new object or as a result by itself. It can then be used somewhere else. No matter where, it still has access to the same outer variables.



```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();

alert( counter() ); // 0
alert( counter() ); // 1
alert( counter() ); // 2
```

Lexical Environment

In JavaScript, every running function, code block {...}, and the script as a whole have an internal (hidden) associated object known as the Lexical Environment.

The Lexical Environment object consists of two parts:

- Environment Record – an object that stores all local variables as its properties (and some other information like the value of this).
- A reference to the outer lexical environment, the one associated with the outer code.

A “variable” is just a property of the special internal object, Environment Record. “To get or change a variable” means “to get or change a property of that object”.

Lexical Environment

For clarity, the explanation is split into multiple steps.

Step 1: Variables

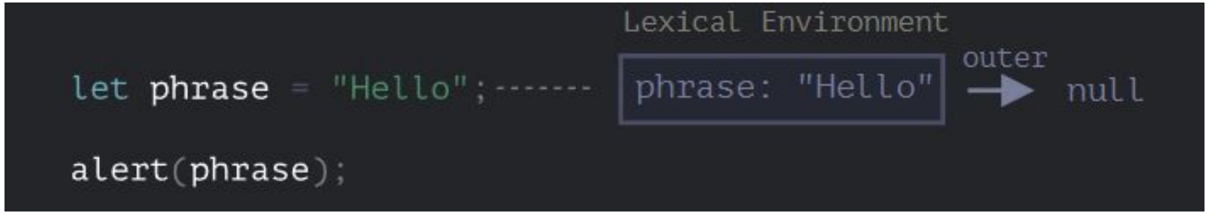
Step 2: Function Declaration

Step 3: Inner and outer lexical environment

Step 4: Returning a function

Step 1 - Variables

In this simple code without functions, there is only one Lexical Environment:



```
let phrase = "Hello";

alert(phrase);
```

This is the so-called global Lexical Environment, associated with the whole script.

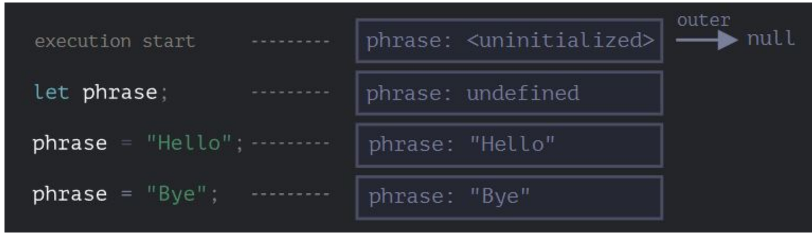
On the picture above, the rectangle means Environment Record (variable store) and the arrow means the outer reference. The global Lexical Environment has no outer reference, that’s why the arrow points to null.

Step 1 - Variables

As the code starts executing and goes on, the Lexical Environment changes

Here's a little bit longer code:.

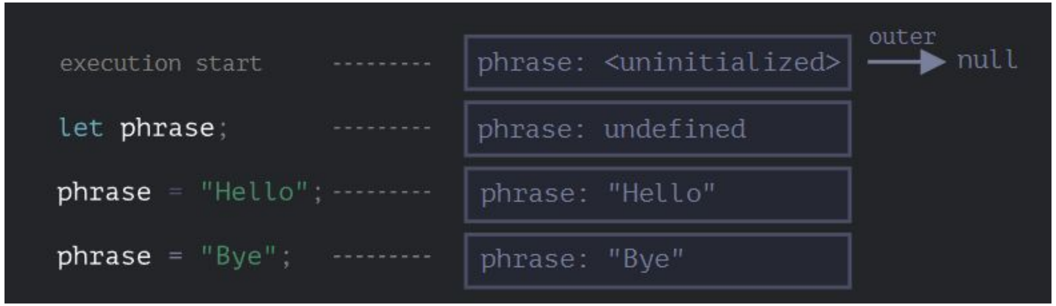
Rectangles on the right-hand side demonstrate how the global Lexical Environment changes during the execution:



```
execution start      ---------  phrase: <uninitialized>  --outer--> null
let phrase;          ---------  phrase: undefined
phrase = "Hello";    ---------  phrase: "Hello"
phrase = "Bye";      ---------  phrase: "Bye"
```

Step 1 - Variables

1. When the script starts, the Lexical Environment is pre-populated with all declared variables.
   - Initially, they are in the “Uninitialized” state. That’s a special internal state, it means that the engine knows about the variable, but it cannot be referenced until it has been declared with let. It’s almost the same as if the variable didn’t exist.
2. Then let phrase definition appears. There’s no assignment yet, so its value is undefined. We can use the variable from this point forward.
3. phrase is assigned a value.
4. phrase changes the value.



```
execution start
let phrase;
phrase = "Hello";
phrase = "Bye";
```

Step 2 - Function Declaration

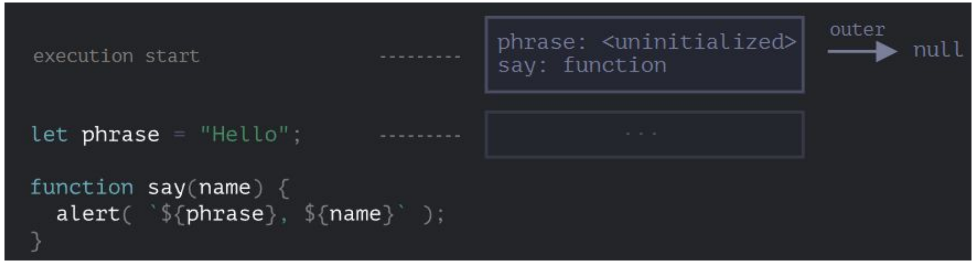
A function is also a value, like a variable.

The difference is that a Function Declaration is instantly fully initialized.

When a Lexical Environment is created, a Function Declaration immediately becomes a ready-to-use function (unlike let, that is unusable till the declaration).

That's why we can use a function, declared as Function Declaration, even before the declaration itself.

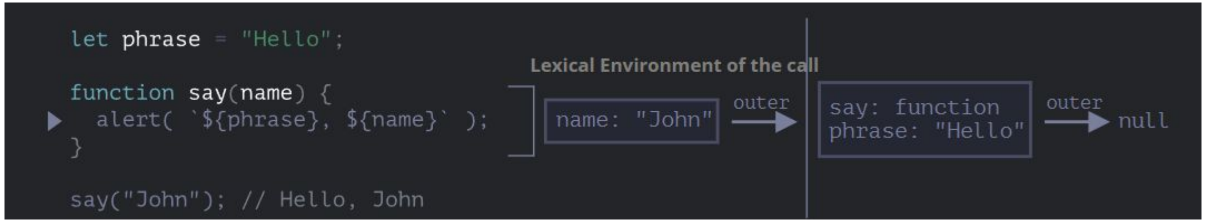
For example, here's the initial state of the global Lexical Environment when we add a function:



```
let phrase = "Hello";

function say(name) {
  alert( `${phrase}, ${name}` );
}
```

Step 3 - Inner and Outer Lexical Environment

When a function runs, at the beginning of the call, a new Lexical Environment is created automatically to store local variables and parameters of the call.



```
let phrase = "Hello";

function say(name) {
  alert( `${phrase}, ${name}` );
}

say("John"); // Hello, John
```

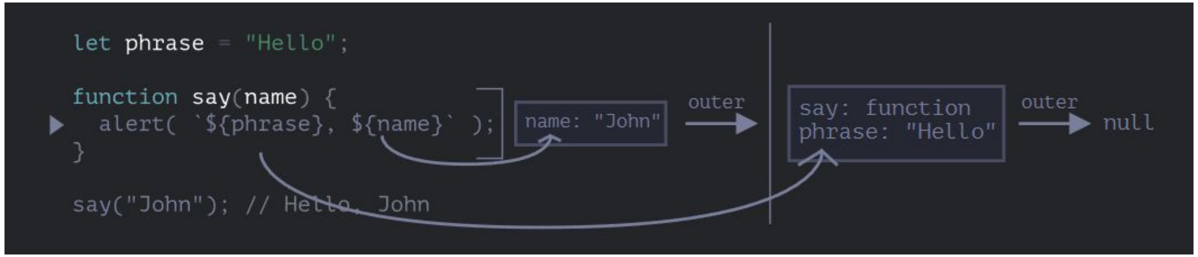
During the function call we have two Lexical Environments: the inner one (for the function call) and the outer one (global):

- The inner Lexical Environment corresponds to the current execution of say. It has a single property: name, the function argument. We called say("John"), so the value of the name is "John".
- The outer Lexical Environment is the global Lexical Environment. It has the phrase variable and the function itself.

Step 3 - Inner and Outer Lexical Environment

The inner Lexical Environment has a reference to the outer one.

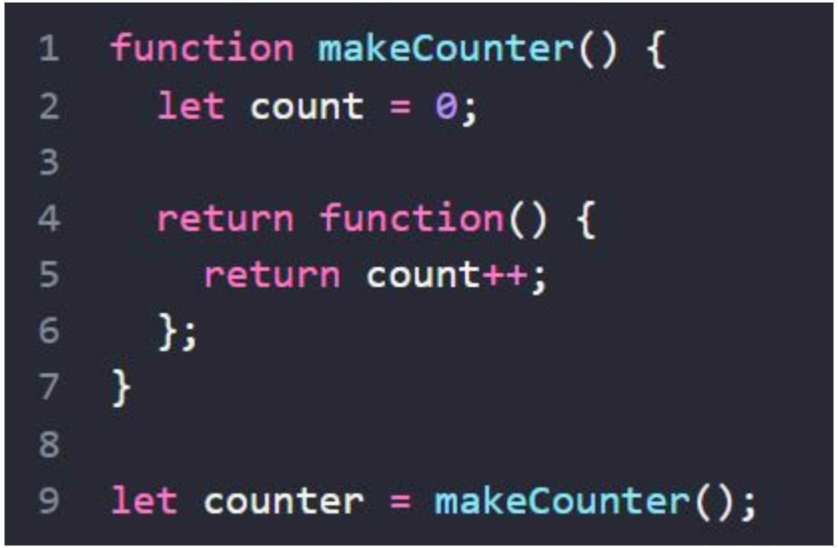
When the code wants to access a variable – the inner Lexical Environment is searched first, then the outer one, then the more outer one and so on until the global one.



```
let phrase = "Hello";

function say(name) {
  alert( `${phrase}, ${name}` );
}

say("John"); // Hello, John
```

Step 4 - Returning a function

Let’s return to the makeCounter example.

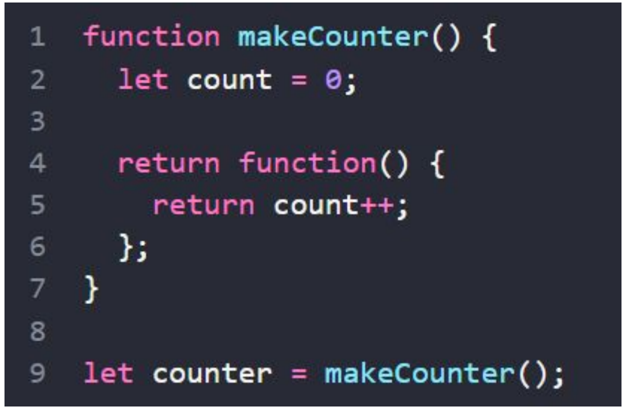


```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();
```

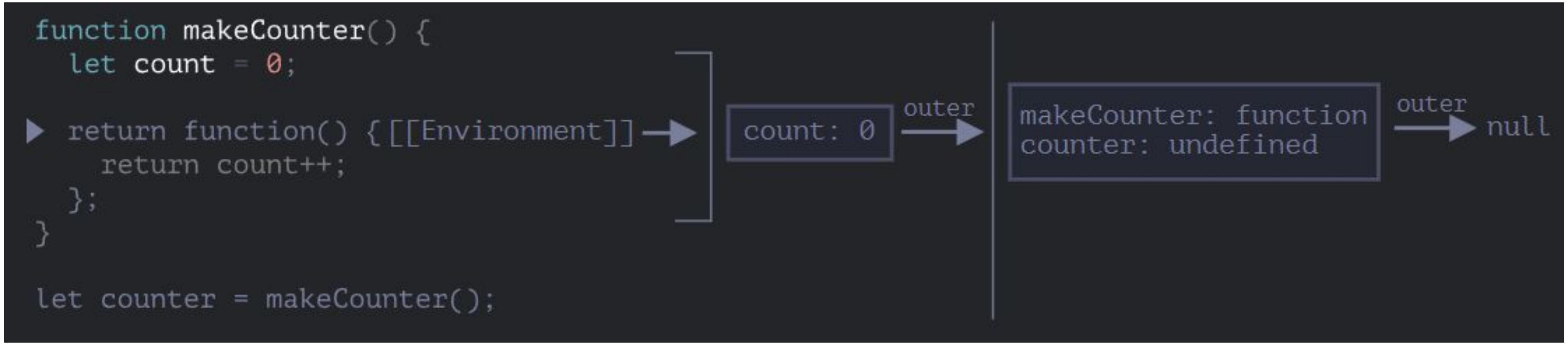
Step 4 - Returning a function

At the beginning of each makeCounter() call, a new Lexical Environment object is created, to store variables for this makeCounter run.

So we have two nested Lexical Environments, just like in the example above:



```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();
```



```
function makeCounter() {
  let count = 0;

▶ return function() { [[Environment]]
    return count++;
  };
}

let counter = makeCounter();
```

Closure

A closure is a function that remembers its outer variables and can access them.

in JavaScript, all functions are naturally closures

That is: they automatically remember where they were created using a hidden [[Environment]] property, and then their code can access outer variables.

When on an interview, a frontend developer gets a question about “what’s a closure?”, a valid answer would be a definition of the closure and an explanation that all functions in JavaScript are closures, and maybe a few more words about technical details: the [[Environment]] property and how Lexical Environments work.

Garbage Collection

Usually, a Lexical Environment is removed from memory with all the variables after the function call finishes. That’s because there are no references to it. As any JavaScript object, it’s only kept in memory while it’s reachable.

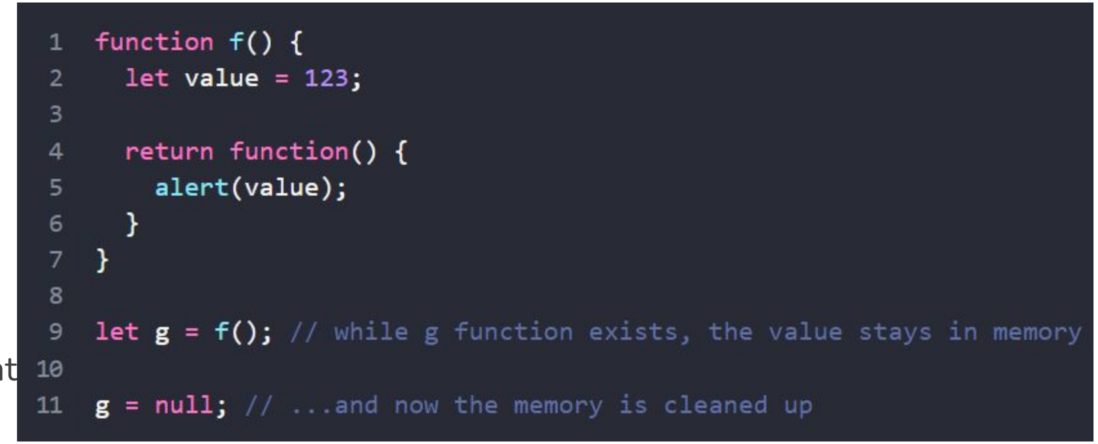
However, if there’s a nested function that is still reachable after the end of a function, then it has [[Environment]] property that references the lexical environment.

In that case the Lexical Environment is still reachable even after the completion of the function, so it stays alive.

Garbage Collection

A Lexical Environment object dies when it becomes unreachable (just like any other object). In other words, it exists only while there's at least one nested function referencing it.

In the code below, after the nested function is removed, its enclosing Lexical Environment (and hence the value) is cleaned from memory:



```
function f() {
  let value = 123;

  return function() {
    alert(value);
  }
}

let g = f(); // while g function exists, the value stays in memory

g = null; // ...and now the memory is cleaned up
```