Miscellaneous function topics

Rest parameters and spread syntax

How do we make a function take in an arbitrary number of arguments? Simple we use the `...` rest operator in the function header.

```
function sum(...args) {
  let sum = 0;
  for (let arg of args) sum += arg;
    return sum;
}
```

When you prefix a parameter with `...` you can pass in an arbitrary number of arguments into the function and it will all be collected into an array that's stored into the parameter `args`.

You can also mix rest parameters with normal parameters like so

```
function showName(firstName, lastName, ...extra) {
  console.log(firstName, lastName);
    console.log(extra);
}


showName("Ricky", "Lu", 30, 40, 50);
```

In this case, the first two parameter will be stored into `firstName` and `lastName` respectively, and any further parameter that you pass in will be stored into `extra` as an array of parameters.

> Keep in mind that the `rest` parameter must be at the end when you use it. You cannot have a function like so
> `function f(arg1, ...rest, arg2)` this will be a syntax error

Spread Syntax

On the other hand, you can also unpack the values from an array or any iterable into a function. For example:

```
// instead of writing
let arr = [3, 5, 1];

console.log(Math.max(arr[0], arr[1], arr[2])); // Too long, and if there are hundreds of values, we are not gonna do this

console.log(Math.max(...arr)); // Much better, this unpacks the arr
```

This is much like the opposite of doing the reverse of rest parameter. We want to spread the values from an array into the parameter of a function.

When `...` is used in a function call it expands the iterable object into the list of arguments.

You can spread multiple iterable into a function

```
let arr1 = [1];
let arr2 = [2, 3, 4, 5, 6];

function foo(a, ...rest) {
	console.log(a, rest);
}

foo(...arr1, ...ar2); // prints out "1 [2, 3, 4, 5, 6]"
```

Merging array

You can also use the spread syntax to merge arrays together, instead of using `arr.concat`

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

let merged = [...arr, ...arr2]; // becomes [3, 5, 1, 8, 9, 15];
```

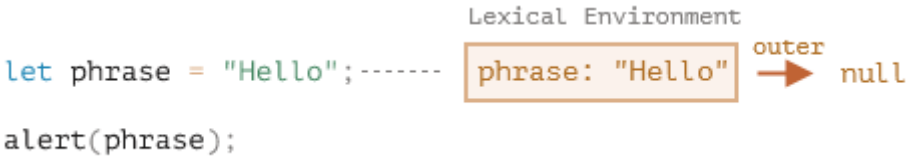
Lexical environment

Okay this is gonna just be a brief summary of what lexical environment is.

Every JavaScript script have something called a lexical environment object that is internal. It consists of: **Environment record** (all of the local variables and methods), and a **reference** to the

outer lexical environment.

When you declare a global variable or global function it is stored in the global lexical environment that is associated with the whole script.



Here in this example above, `phrase` is a global variable and hence its record is stored in the global lexical environment. The global lexical environment does not have reference to a outer lexical environment because it is the most outer one, hence it is just `null`.

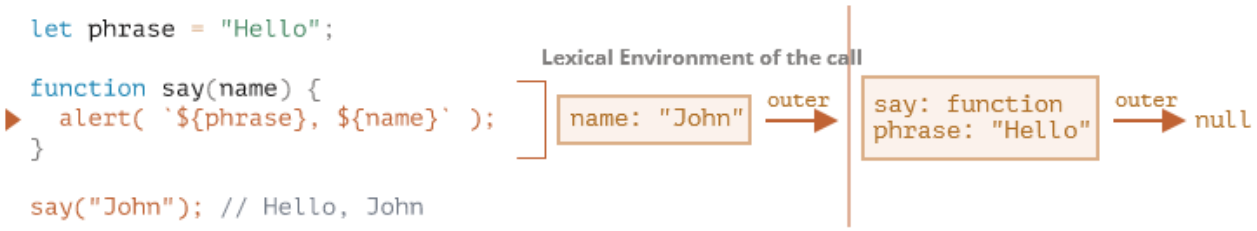
Variable declaration and function declaration

When you declare a variable, it is available in the lexical environment immediately but the value it has is uninitialized from the beginning, and as the script execute to the point where it is initialize, that value is updated.

On the other hand, for function declaration (not function expression or arrow functions), they are available immediately become ready-to-use functions. It doesn't have to wait until the line the function becomes defined to be initialized. This is why we are able to call the function before the function declaration!

Inner and outer lexical environment

When you invoke a function it creates a new lexical environment to store the local variables and parameter of the function call. **Every new function invocation will create a new lexical environment!**



Here in this example, invoking `say` creates a new lexical environment and has the parameter information in it. It has the reference to the outer lexical environment, which in this case is the global lexical environment.

When the function wants to access a variable, the inner lexical environment is searched first, then the outer one, and recursively back up until the global lexical environment.

If the variable is not found anywhere, then it is an error in `strict mode`, without `strict mode` then assignment to a non-existing variable will be automatically added to the global lexical

environment.

Returning a function

If you wrote a function that returns another function say:

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

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

let counter = makeCounter();
counter(); // count becomes 1
counter(); // count becomes 2
```

In this case `makeCounter()` creates a lexical environment that holds the variable `count`, then it returned a function that will increment the outer `count`. How does it do that? When `counter` is invoked later on, it will again create a new lexical environment with nothing in it because it has no variables but rather referring to the outer one. Since it is created in the lexical environment in `makeCounter` the reference that the inner function points to will be `makeCounter`'s lexical environment and it has the `count`. Then when it tries to increment `count` it will be incrementing the `count` under `makecounter`'s lexical environment and it works out!

Closure

Now this is where closure comes in. A closure is a function that is able to remember it's environment context that it was created in. It remembers the outer variable and is able to access them. Some languages don't support closure, and if it doesn't then what we have just talked about isn't possible.

All JavaScript functions are closure, is able to resolve those outer variables that it used, and when those variables goes out of scope it is still able to remember them, have closure per say. The only exception is the `new Function` syntax, it is not closure.

Global object

The global object provides variables and functions that are available anywhere, by default it stores the ones that are built into the language or the runtime environment.

For browsers it is named `window`, for Node.js it is `global`, but it is recently been renamed into `globalThis`

You can access the property of global object directly. In addition, all `var` variables are becomes the property of the global object. Variables without `let` or `var` are implicitly `var` hence they become the property of the global object as well! (Without strict mode that is. With strict mode, it is an error)

Usage of global variable

It is generally discouraged, there should be as few global variables as possible, with access via the global object that is.

The `new Function` syntax

You can create a new function via a string:

```
let func = new Function([arg1, arg2, ...argN], functionBody); // Both the args and functions should be strings
```

For example:

```
let sum = new Function('a', 'b', 'return a + b');
let sum = new Function('a, b', 'return a + b');
// Both are equivalent.


sum(1, 2) // will be 3
```

Now using this way to create function it is not closure. Meaning that it cannot access any outer variables! This is the only exception that functions aren't closure, in all other cases functions in JavaScript are closure.

Named function expression

When you are writing a function expression you don't normally give it a name, but the thing is you can, so writing this is perfectly valid:

```
let sayHi() = function func(who) {
  console.log(`Hi ${who}`);
};


sayHi("John"); // Hi John
```

But what does this achieve? By adding a name to the function expression it did not become a function declaration, it is still a function expression!

You can still call the function as it is using `sayHi`. However, by adding `func` name we are able to let the function calls itself internally, and it is not visible outside of the function.

```
let sayHi = function func(who) {
  if (!who) {
    func("Anonymous");
    }
    else {
    console.log(`Hi ${who}`);
    }
}

sayHi(); // Hello, guest!
```

We use `func` internally instead of `sayHi` is because the value of `sayHi` could be changed down the line, and if it is changed, to say a number, then the reference inside would not be valid anymore.

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest"); // Error: sayHi is not a function
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Error, the nested sayHi call doesn't work any more!
```

Scheduling

setTimeout/setInterval

Both follows the function header:

```
setTimeout/setInterval(func | code, [delay], [arg1], [arg2], ...)
```

- `func`: Refers to the function to execute
- `delay`: The number of milliseconds to wait before the code specified is executed, by default is 0 so execute immediately.

- `arg1, arg2,...`: The arguments for the function that you specified

```
function waveHello() {
  console.log("I am waving hello!")
}

setTimeout(waveHello, 1000); // I am waving hello! After one second
```

The difference between `setTimeout` and `setInterval` is that `setTimeout` will only execute the function once after the specified delay, while `setInterval` will execute that function regularly after every specified `delay`. So if you write `setInterval(waveHello, 1000)` this will wave hello after every second.

clearTimeout/clearInterval

Use these two functions to delete the function that is going to be called after you do `setTimeout/setInterval`. You will have to pass the "timer identifier" that is returned from calling `setTimeout/setinterval`, in order to cancel the timer handler.

Nested setTimeout

A better way of doing interval code execution is via nested `setTimeout`

```
setTimeout(function tick() {
  console.log("Doing work every regularly");
    setTimeout(tick, 2000);
}, 2000);
```

Recalling from named function expression that an function expression can be named and itself can refer to it internally. Now the first function execution will occur after 2 seconds, then it will run the body of the `tick` function, it will do the work and schedule itself to run 2 seconds again later.

Why is this better? It gives us a finer control on when to schedule, instead doing it every 2 seconds, we can control the time of the interval inside the `tick` function based on say CPU-usage, or how much work is given. We can do it every 10 seconds, 20 seconds, or 60 seconds so a variable interval is what this is trying to emulate.

In addition, nested `setTimeout` guarantees the fixed delay. Since `setInterval` the function execution can take up the delay, thus making the interval inaccurate.

Zero delay setTimeout

There is actual a usage for `setTimeout(func, 0)`. This schedules the execution of the function as soon as possible, but the scheduler is invoked only after the current executing script is complete. Hence the function is scheduled to run right after the current script is finished.

```
setTimeout(() => console.log("World"))console.log("hello!");// Prints out hello! World
```

Function binding

If you decide to somehow pass an object's method as a callback into say `setTimeout`, you will lose the `this` keyword in the method

```
let user = {
    firstName: "John",
    sayHi() {
        console.log("Hi I am " + this.firstName);
    }
};

setTimeout(user.sayHi, 1000); // This will print Hi I am undefined
```

The method that was passed into `setTimeout` didn't have a receiver when it was invoked. So again if the method that you invoked doesn't have a receiver, in browser `this` will be binded to `window` object, and for Node.js it will be the `timer` object, but not that relevant.

Solution 1: Wrapper

We can solve this by wrapping the method that we actually want to invoke in another function call like so

```
let user = {
    firstName: "John",
    sayHi() {
        console.log("Hi I am " + this.firstName);
    }
};

setTimeout(function() {
  user.sayHi()
}, 1000); // This will print Hi I am undefined
```

Now because of closure, it is able to resolve `this` to be the appropriate user object and this will be fine.

However, this solution will fail if the `user` object somehow changed before the callback is executed, then it will be invoked on the changed value, not the old one anymore.

Solution 2: bind

To solve the issue that was discussed previously where if the object is somehow changed before the callback is executed, it will be executing callback with the updated object, we can use the `bind` function to fix this.

```
let boundFunc = func.bind(context);
```

The result of calling `bind` on a function is another function that has the same body but with `this=context` fixed. For example:

```
let user = {
  firstName: "John"
};

function func() {
  console.log(this.firstName);
}

let funcUser = func.bind(user);
funcUser(); // John, because this is set to be user.
```

The returned function will have the same spec as the original function the only thing that changed is that `this` is fixed to whatever object that you have provided.

Using `bind` we can solve the problem we have just discussed by fixing the `this` to be that original object, it won't matter if the object changed down the line:

```
let user = {
  firstName: "John",
  sayHi() {
    console.log(`Hello, ${this.firstName}!`);
  }
};

let sayHi = user.sayHi.bind(user); // (*)

// You can run it without an explicit receiver
sayHi(); // Hello, John

setTimeout(sayHi, 1000); // Hello, John
```

```
// Even if user is changed to something else, it will still do Hello, John
user = {};
```

> After you have bind an object, it cannot be changed again! Meaning you cannot do .bind again on the function that was returned.

Partial functions

With the `bind` method you can also fulfill the partial parameter, here is an example

```
function mul(a, b) {
  return a * b;
}

let double = mul.bind(null, 2);

double(3) // 6
double(8) // 16
```

We can partially fill out the function that we are binding with some predetermined parameter, in this case `a=2`, then the user only need to fill out one more parameter `b` in this case and the result will be returned.

You can also make methods that are partial as well if you so to choose, this is done by using the `func.call` method which invokes the method with the option to provide the `this` context. Then you can just return a function that will call the method with the predetermined parameters, and let the user provide additional parameter.

```
function partial(func, ...argsBound) {
  return function(...args) { // Returns a function that is partially filled. Let user put in additional args if needed
    return func.call(this, ...argsBound, ...args);
  }
}

let user = {
  firstName: "John",
  say(time, phrase) {
    alert(`[${time}] ${this.firstName}: ${phrase}!`);
  }
};

// add a partial method with fixed time
```

```
// takes in the function to partially fill, and the args to prefill with
user.sayNow = partial(user.say, new Date().getHours() + ':' + new Date().getMinutes());

// user.sayNow is a prefilled method, it this is binded to the same user object still
// because func.call(this...)
// now you can call the function with any additional method that was needed after prefilled
user.sayNow("Hello");
```

Revision #8
Created 24 December 2022 16:13:21 by Tamarine
Updated 27 July 2023 01:52:09 by Tamarine