4 - Advance Function Utilization

Refer to this link to learn more about recursion and linked lists: (and to do some exercises)
https://javascript.info/recursion

Parameters

Functions can be called with any number of parameters, and it does not matter how it is defined:

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

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

There will be no error for excessive parameters, but only the first two will be counted.
The rest of the parameters can be included in the fucntion definition by using three dots ... followed by the name of the array that will contain them.

```
function sumAll(...args) { // args is the name for the array
  let sum = 0;

  for (let arg of args) sum += arg;

  return sum;
}

alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6

//Here we set the first two parameters and let the rest go into titles
array:

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");
```

Spread syntax

How can we pass more values held in an array to a function?

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

alert( Math.max(arr) ); // NaN
```

This code doesn't work, because Math.max expects a list o numeric arguments and not a single array. We can use the spread syntax to achieve our goal:

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

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

// We can also combine the syntax with other values:

let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25

// Or we can use the spread syntax to merge arrays:
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)

We can also use spread syntax to make copies of Objects or arrays:

let obj = { a: 1, b: 2, c: 3 };

let objCopy = { ...obj }; // spread the object into a list of parameters
                          // then return the result in a new object

// do the objects have the same contents?
alert(JSON.stringify(obj) === JSON.stringify(objCopy)); // true
```

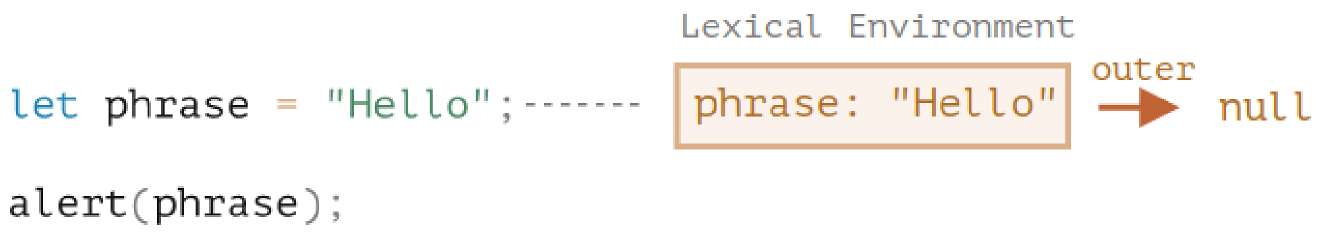
```
// are the objects equal?
alert(obj === objCopy); // false (not same reference)

// modifying our initial object does not modify the copy:
obj.d = 4;
alert(JSON.stringify(obj)); // {"a":1,"b":2,"c":3,"d":4}
alert(JSON.stringify(objCopy)); // {"a":1,"b":2,"c":3}

// it also works to build objects from other objects definitions:

const o1={a:'va',b:'vb'};
const o2={c:'vc', ...o1};
```

Lexical environment

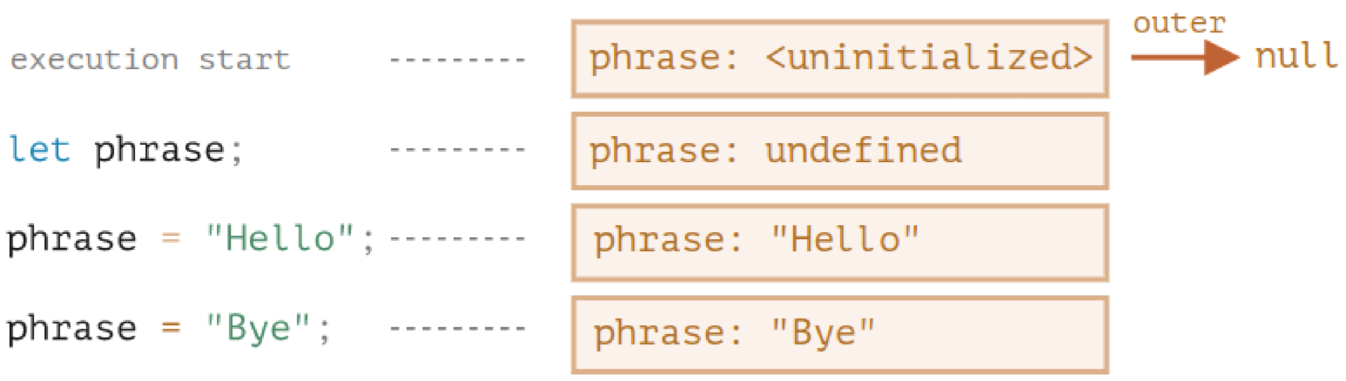
In JS, every running function, code block {} and the script as a whole have an internal associated object known as Lexical Environment. It consist of two parts:

- Environment Record: an object that stores all local variables as its properties (like value of this)
- A reference to the outer lexical environment, the one associarìted 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”.



This is the *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.



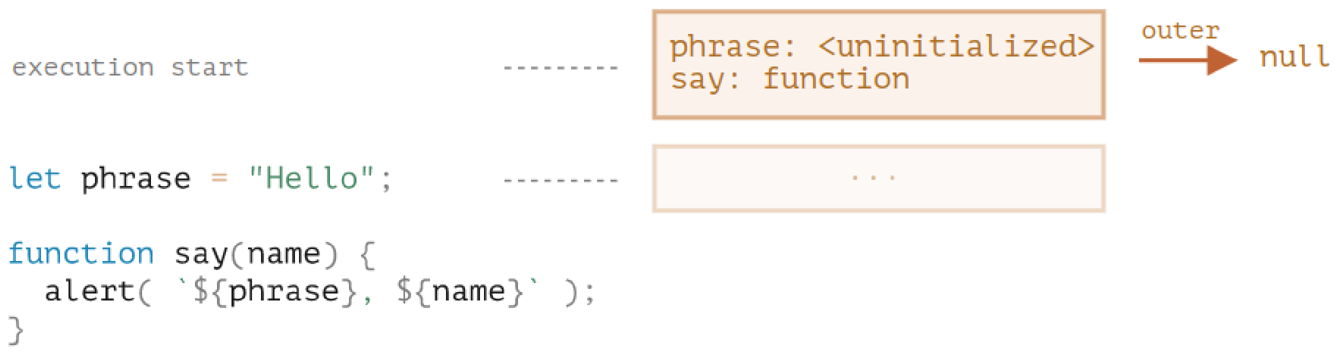
When the script starts, the Lexical Environment is pre-populated with all declared variables.

1. 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.

“Lexical Environment” is a specification object: it only exists “theoretically” in the language specification to describe how things work. We can’t get this object in our code and manipulate it directly.
JavaScript engines also may optimize it, discard variables that are unused to save memory and perform other internal tricks, as long as the visible behavior remains as described.

Functions

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:



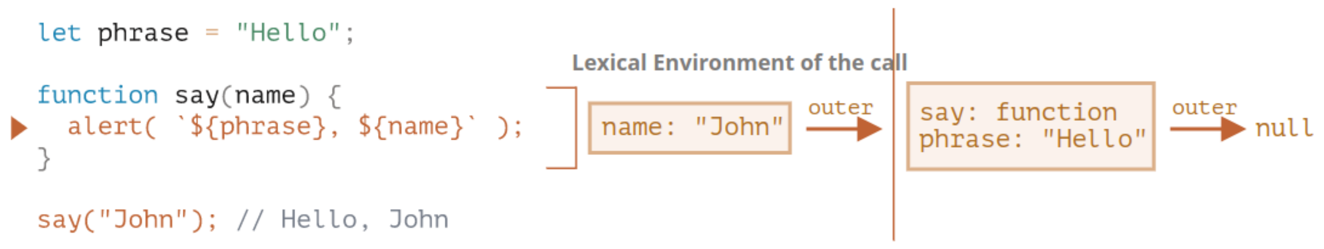
```
let phrase = "Hello";

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

Naturally, this behavior only applies to Function Declarations, not Function Expressions where we assign a function to a variable, such as let say = function(name)...

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.

For instance, for say("John"), it looks like this (the execution is at the line, labelled with an arrow):



```
let phrase = "Hello";

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

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

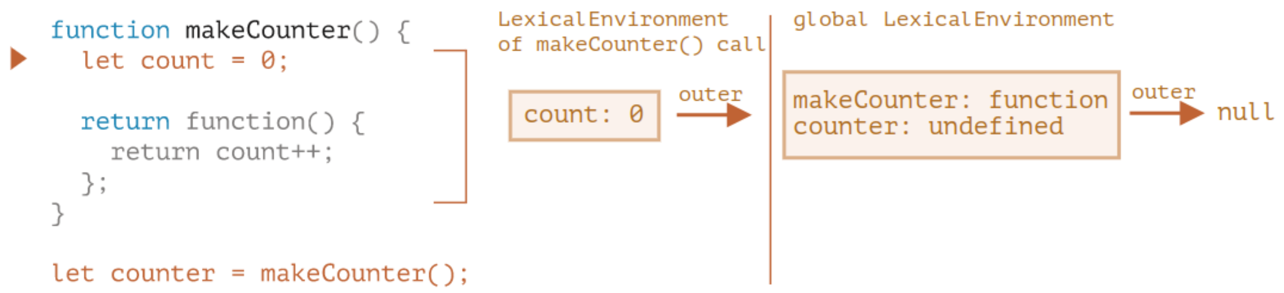
The inner Lexical Environment corresponds to the current execution of say. It has a single property: name, the function argument. We called say("John"), so the value of the name is "John".
The outer Lexical Environment is the global Lexical Environment. It has the phrase variable and the function itself.
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.
If a variable is not found anywhere, that's an error in strict mode (without use strict, an assignment to a non-existing variable creates a new global variable, for compatibility with old code).

In this example the search proceeds as follows:

- For the name variable, the alert inside say finds it immediately in the inner Lexical Environment.
- When it wants to access phrase, then there is no phrase locally, so it follows the reference to the outer Lexical Environment and finds it there.

Example with makeCounter:

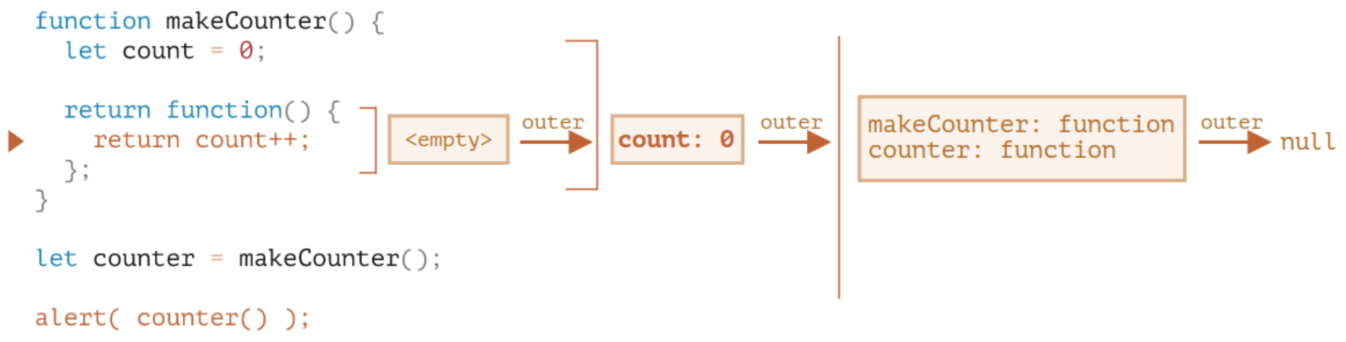
```
function makeCounter() {
  let count = 0;

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

let counter = makeCounter();
```



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

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

let counter = makeCounter();
```

What's different is that, during the execution of makeCounter(), a tiny nested function is created of only one line: return count++. We don't run it yet, only create.

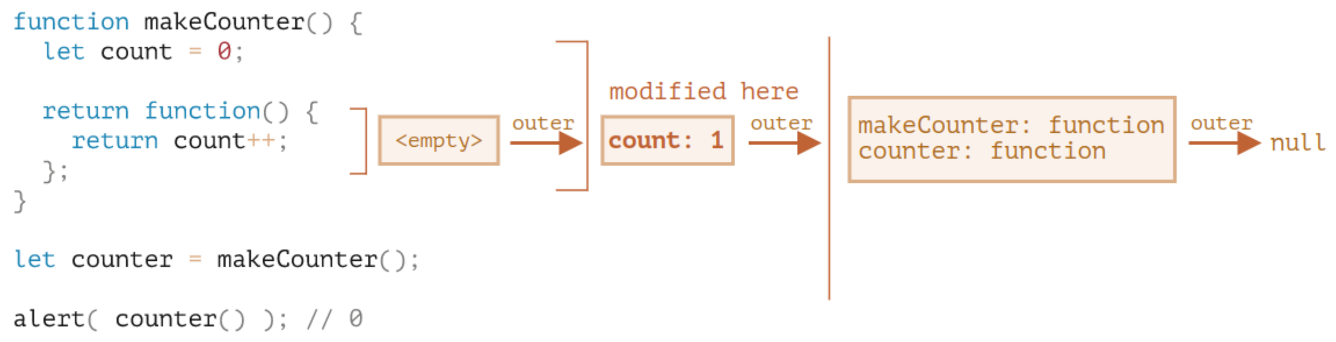
All functions remember the Lexical Environment in which they were made. Technically, there’s no magic here: all functions have the hidden property named [[Environment]], that keeps the reference to the Lexical Environment where the function was created.
So, counter.[[Environment]] has the reference to {count: 0} Lexical Environment. That’s how the function remembers where it was created, no matter where it’s called. The [[Environment]] reference is set once and forever at function creation time.
Later, when counter() is called, a new Lexical Environment is created for the call, and its outer Lexical Environment reference is taken from counter.[[Environment]]:



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

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

let counter = makeCounter();

alert( counter() );
```

Now when the code inside counter() looks for count variable, it first searches its own Lexical Environment (empty, as there are no local variables there), then the Lexical Environment of the outer makeCounter() call, where it finds and changes it.

A variable is updated in the Lexical Environment where it lives.



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

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

let counter = makeCounter();

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

If we call counter() multiple times, the count variable will be increased to 2, 3 and so on, at the same place.

Closure

There is a general programming term “closure”, that developers generally should know.
A closure is a function that remembers its outer variables and can access them. In some languages, that’s not possible, or a function should be written in a special way to make it happen. But as explained above, in JavaScript, all functions are naturally closures (there is only one exception, to be covered in The "new Function" syntax).
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.

For example:

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

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

let g = f(); // g.[[Environment]] stores a reference to the Lexical
Environment
// of the corresponding f() call

// Please note that if f() is called many times, and resulting
// functions are saved, then all corresponding Lexical
// Environment objects will also be retained in memory. In the code below,
all 3 of them:

function f() {
  let value = Math.random();

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

// 3 functions in array, every one of them links to Lexical Environment
// from the corresponding f() run
let arr = [f(), f(), f()];
```

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
```

As we've seen, in theory while a function is alive, all outer variables are also retained.
But in practice, JavaScript engines try to optimize that. They analyze variable usage and if it's obvious from the code that an outer variable is not used – it is removed.
An important side effect in V8 (Chrome, Edge, Opera) is that such variable will become unavailable in debugging.

Global Object

The global object provides functions and variables that are available everywhere. In a browser it is named **window*, but in Node.js it is **Global**. Recently, globalThis was added to the language, as a standardized name for a global object, that should be supported across all environments. It's supported in all major browsers. All properties of the global object are accessible anywhere:

```
alert("Hello");
// is the same as
window.alert("Hello");
```

In the browser, global functions and varibales declared with var become the property of the global object.

```
var gVar = 5;

alert(window.gVar); // 5 (became a property of the global object)
```

That said, using global variables is generally discouraged. There should be as few global variables as possible.

We usually use the global object to test for support of modern language features, like testing if built-in objects like Promise exist.

Function Object

In Javascript, a function is a value, and every value has a type: functions in JS are objects. We can call them and treat them as objects: add and remove properties, pass by reference.

Name property

Function objects contain some useable properties, for example a function name is accessible as the "name" property:

```
function sayHi() {
  alert("Hi");
}

alert(sayHi.name);

// It works even if the function doesn't have a name but it gets immediately
assigned:
let sayHi = function() {
  alert("Hi");
};

alert(sayHi.name); // sayHi (there's a name!)

// And it also works if the assignment is done via default value:
function f(sayHi = function() {}) {
  alert(sayHi.name); // sayHi (works!)
}
f();

// If the function does not provide one, it is figured out from the context.
Object methods have names too:
let user = {

  sayHi() {
    // ...
  },

  sayBye: function() {
    // ...
  }

}
```

```
alert(user.sayHi.name); // sayHi
alert(user.sayBye.name); // sayBye
```

Length property

Return the number of function parameters, for instance:

```
function f1(a) {}
function f2(a, b) {}
function many(a, b, ...more) {}

alert(f1.length); // 1
alert(f2.length); // 2
alert(many.length); // 2 - We can see that the rest is not counted
```

It is often used for introspection, which is the ability to examine properties of an object at runtime. For instance, in the code below the ask function accepts a question to ask and an arbitrary number of handler functions to call.

Once a user provides their answer, the function calls the handlers. We can pass two kinds of handlers:

A zero-argument function, which is only called when the user gives a positive answer.
A function with arguments, which is called in either case and returns an answer.
To call handler the right way, we examine the handler.length property.

The idea is that we have a simple, no-arguments handler syntax for positive cases (most frequent variant), but are able to support universal handlers as well:

```
function ask(question, ...handlers) {
  let isYes = confirm(question);

  for(let handler of handlers) {
    if (handler.length == 0) {
      if (isYes) handler();
    } else {
      handler(isYes);
    }
  }

}

// for positive answer, both handlers are called
// for negative answer, only the second one
ask("Question?", () => alert('You said yes'), result => alert(result));
```

This is a particular case of so-called polymorphism – treating arguments differently depending on their type or, in our case depending on the length. The idea does have a use in JavaScript libraries.

Custom properties

We can also add properties of our own, here we add the counter property to track the total calls count:

```
function sayHi() {
  alert("Hi");

  // let's count how many times we run
  sayHi.counter++;
}
sayHi.counter = 0; // initial value

sayHi(); // Hi
sayHi(); // Hi

alert( `Called ${sayHi.counter} times` ); // Called 2 times
```

A property assigned to a function like sayHi.counter = 0 does not define a local variable counter inside it. In other words, a property counter and a variable let counter are two unrelated things. We can treat a function as an object, store properties in it, but that has no effect on its execution. Variables are not function properties and vice versa. These are just parallel worlds.

Function properties can replace closures:

```
function makeCounter() {
  // instead of:
  // let count = 0

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

  counter.count = 0;

  return counter;
}

let counter = makeCounter();
alert( counter() ); // 0
alert( counter() ); // 1
```

The main difference is that if the value of count lives in an outer variable, then external code is unable to access it. Only nested functions may modify it. And if it’s bound to a function, then such a thing is possible:

```
function makeCounter() {

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

  counter.count = 0;

  return counter;
}

let counter = makeCounter();

counter.count = 10;
alert( counter() ); // 10
```

Named function expressions

Its a term for function expressions that have a name. Let's start from an ordinary function expression:

```
let sayHi = function(who) {
  alert(`Hello, ${who}`);
};
// Let's add a name to it
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
};
// The function is still available as SayHi()
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
};

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

There are two special things about the name func, that are the reason for it:

- Allows the function to reference itself internally.
- It is not visible outside of the function.
  The following function calls itself again with Guest if no who is provided:

```
let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    func("Guest"); // use func to re-call itself
  }
};

sayHi(); // Hello, Guest

// But this won't work:
func(); // Error, func is not defined (not visible outside of the function)

// But why jsut not use SayHi again? Actually we can:
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest");
  }
};
```

But SayH may change in the outer code, if the function gets assigned to another variable instead, the code will give errores.

```
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!
```

That happens because the function takes sayHi from its outer lexical environment. There's no local sayHi, so the outer variable is used. And at the moment of the call that outer sayHi is null. The optional name which we can put into the Function Expression is meant to solve exactly these kinds of problems. Using 'func' works because the name "func" is function-local. It is not taken from outside (and not visible there). The specification guarantees that it will always reference the current function.
The outer code still has its variable sayHi or welcome. And func is an "internal function name", the way for the function to can call itself reliably.

**There's no such thing for Function Declaration
The "internal name" feature described here is only available for Function Expressions, not for Function Declarations. For Function Declarations, there is no syntax for adding an "internal" name.

Sometimes, when we need a reliable internal name, it's the reason to rewrite a Function Declaration to Named Function Expression form.**

Summary

Functions are objects.

Here we covered their properties:

name – the function name. Usually taken from the function definition, but if there's none, JavaScript tries to guess it from the context (e.g. an assignment).
length – the number of arguments in the function definition. Rest parameters are not counted.
If the function is declared as a Function Expression (not in the main code flow), and it carries the name,

then it is called a Named Function Expression. The name can be used inside to reference itself, for recursive calls or such.

Also, functions may carry additional properties. Many well-known JavaScript libraries make great use of this feature.

They create a “main” function and attach many other “helper” functions to it. For instance, the jQuery library creates a function named $. The lodash library creates a function _, and then adds _.clone, _.keyBy and other properties to it (see the docs when you want to learn more about them). Actually, they do it to lessen their pollution of the global space, so that a single library gives only one global variable. That reduces the possibility of naming conflicts.

So, a function can do a useful job by itself and also carry a bunch of other functionality in properties.

https://javascript.info/function-object#tasks

New Function Syntax

There's one more way to create a function, rarely used but there's no alternative sometimes

```
let func = new Function ([arg1, arg2, ...argN], functionBody);
```

The function is created with the arguments arg1,...argN and with a body:

```
let sum = new Function('a', 'b', 'return a + b');

alert( sum(1, 2) ); // 3

// Or even without arguments:

let sayHi = new Function('alert("Hello")');

sayHi(); // Hello
```

The new function allows to turn any string into a function. For example, we can receive a new function from a server and then execute it:

```
let str = ... receive the code from a server dynamically ...

let func = new Function(str);
func();
```

Closure

Usually, a function remembers where it was born in the special property [[Environment]]. It references the Lexical Environment from where it’s created.
But when a function is created using new Function, its [[Environment]] is set to reference not the current

Lexical Environment, but the global one.
So, such function doesn't have access to outer variables, only to the global ones.

```
function getFunc() {
  let value = "test";

  let func = new Function('alert(value)');

  return func;
}

getFunc()(); // error: value is not defined
```

This special feature of new Function looks strange, but appears very useful in practice.
Imagine that we must create a function from a string. The code of that function is not known at the time of writing the script (that's why we don't use regular functions), but will be known in the process of execution. We may receive it from the server or from another source. Our new function needs to interact with the main script.

What if it could access the outer variables?

The problem is that before JavaScript is published to production, it's compressed using a minifier – a special program that shrinks code by removing extra comments, spaces and – what's important, renames local variables into shorter ones.
For instance, if a function has let userName, minifier replaces it with let a (or another letter if this one is occupied), and does it everywhere. That's usually a safe thing to do, because the variable is local, nothing outside the function can access it. And inside the function, minifier replaces every mention of it. Minifiers are smart, they analyze the code structure, so they don't break anything. They're not just a dumb find-and-replace.
So if new Function had access to outer variables, it would be unable to find renamed userName.
If new Function had access to outer variables, it would have problems with minifiers.
Besides, such code would be architecturally bad and prone to errors.
To pass something to a function, created as new Function, we should use its arguments.

Scheduling: setTimeout and setInterval

We may decide to execute a function not right now, but at a certain time later. That’s called “scheduling a call”.

There are two methods for it:

- setTimeout allows us to run a function once after the interval of time.
- setInterval allows us to run a function repeatedly, starting after the interval of time, then repeating continuously at that interval.

These methods are not a part of JavaScript specification. But most environments have the internal scheduler and provide these methods. In particular, they are supported in all browsers and Node.js.

setTimeout

func|code : function or a string of code to execute. Usually, thats a function. For historical reasons, a string of code can be passed but that's not reccomended.

delay: delay before run, in ms, by default 0.

arg1,arg2 Arguments for the function.

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)

function sayHi() {
  alert('Hello');
}
// calling SayHi every second
setTimeout(sayHi, 1000);
// with arguments
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}
setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John

// If first argument is a string, js creates a function from it:

setTimeout("alert('Hello')", 1000);

// but its better to use an arrow function:

setTimeout(() => alert('Hello'), 1000);
```

Pass a function, but don’t run it
Novice developers sometimes make a mistake by adding brackets () after the function:

```
// wrong!
setTimeout(sayHi(), 1000);
```

That doesn’t work, because setTimeout expects a reference to a function. And here sayHi() runs the function, and the result of its execution is passed to setTimeout. In our case the result of sayHi() is undefined (the function returns nothing), so nothing is scheduled.

clearTimeout

We can cancel a setTimeout with this function:

```
let timerId = setTimeout(...);
clearTimeout(timerId);

// here we set a function and then cancel it, so nothing happens:
let timerId = setTimeout(() => alert("never happens"), 1000);
alert(timerId); // timer identifier

clearTimeout(timerId);
alert(timerId); // same identifier (doesn't become null after canceling)
```

setInterval

This method has the same syntax as setTimeout:

```
let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)
```

All arguments have the same meaning. But unlike setTimeout it runs the function not only once, but regularly after the given interval of time.

To stop further calls, we should call clearInterval(timerId).

The following example will show the message every 2 seconds. After 5 seconds, the output is stopped:

```
let timerId = setInterval(() => alert('tick'), 2000);

// after 5 seconds stop
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

Nested setTimeout

There are two ways of running something regularly.

One is setInterval. The other one is a nested setTimeout, like this:

```
/** instead of:
let timerId = setInterval(() => alert('tick'), 2000);
*/

let timerId = setTimeout(function tick() {
  alert('tick');
  timerId = setTimeout(tick, 2000); // (*)
}, 2000);
```

The nested setTimeout is a more flexible method than setInterval. This way the next call may be scheduled differently, depending on the results of the current one.

For instance, we need to write a service that sends a request to the server every 5 seconds asking for data, but in case the server is overloaded, it should increase the interval to 10, 20, 40 seconds…

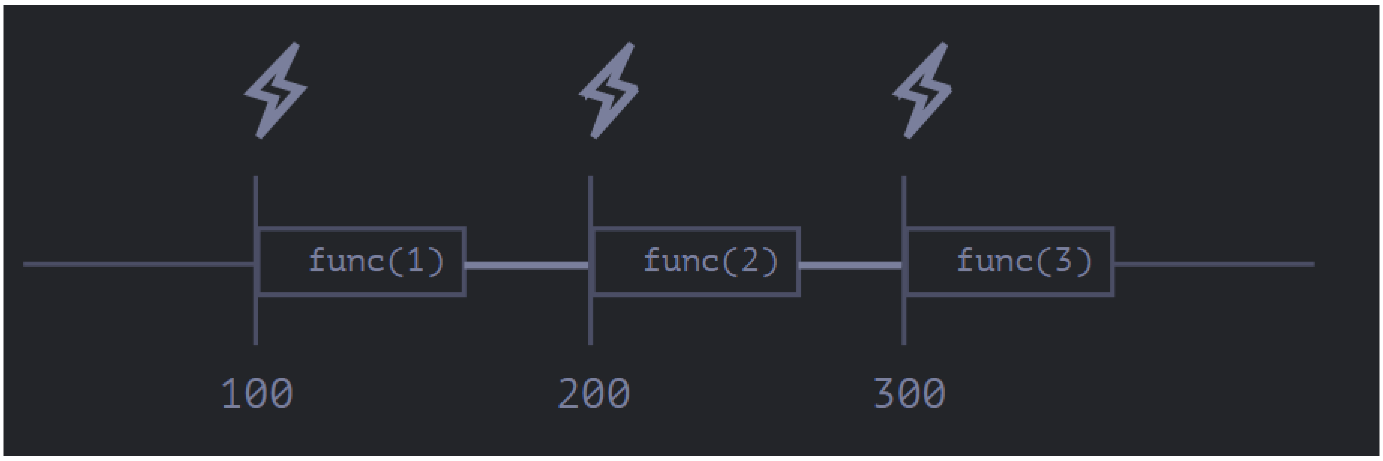
```
Here’s the pseudocode:

let delay = 5000;

let timerId = setTimeout(function request() {
  ...send request...

  if (request failed due to server overload) {
    // increase the interval to the next run
    delay *= 2;
  }

  timerId = setTimeout(request, delay);

}, delay);
```

And if the functions that we’re scheduling are CPU-hungry, then we can measure the time taken by the execution and plan the next call sooner or later.
Nested setTimeout allows to set the delay between the executions more precisely than setInterval.

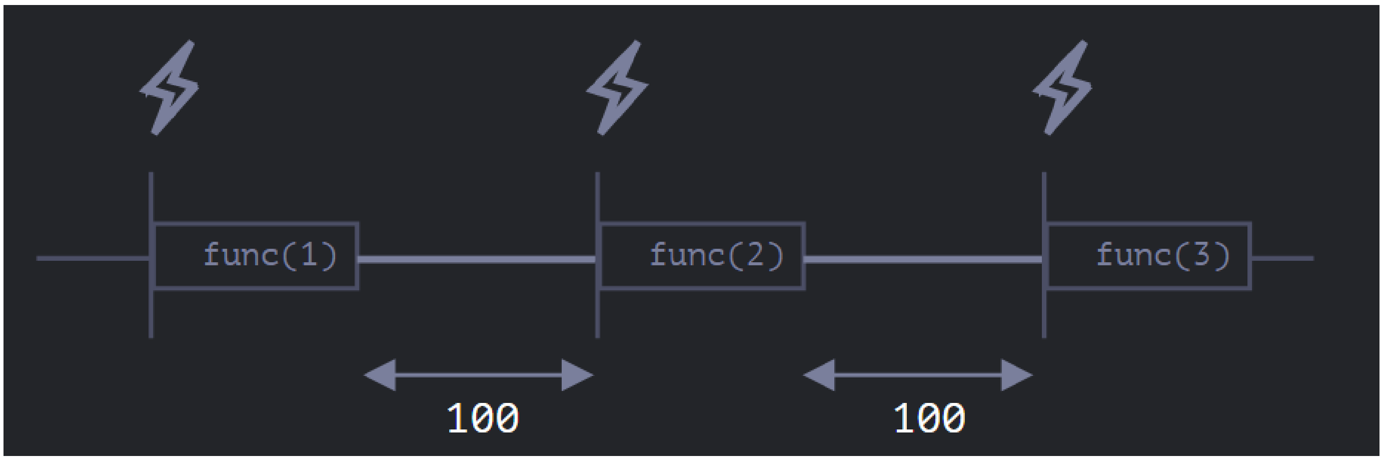
Let's compare nested setTimeouts and setIntervals:

```
// Nested setTimeout
let i = 1;
setTimeout(function run() {
  func(i++);
  setTimeout(run, 100);
}, 100);
```

```
// setInterval
let i = 1;
setInterval(function() {
  func(i++);
}, 100);
```



The real delay between func calls for setInterval is less than in the code!
That’s normal, because the time taken by func's execution “consumes” a part of the interval.
It is possible that func's execution turns out to be longer than we expected and takes more than 100ms.
In this case the engine waits for func to complete, then checks the scheduler and if the time is up, runs it again immediately.
In the edge case, if the function always executes longer than delay ms, then the calls will happen without a pause at all.
And here is the picture for the nested setTimeout:



Nested setTimeouts guarantees the fixed delay.

Garbage collection in setInterval and setTimeout:

When a function is passed in setInterval/setTimeout, an internal reference is created to it and saved in the scheduler. It prevents the function from being garbage collected, even if there are no other references to it. For setInterval the function stays in memory until clearInterval is called.
There’s a side effect. A function references the outer lexical environment, so, while it lives, outer variables live too. They may take much more memory than the function itself. **So when we don’t need the scheduled function anymore, it’s better to cancel it, even if it’s very small.**

Zero delay setTimeout

There's a special use case: setTimeout(func, 0), or just setTimeout(func).
This schedules the execution of func as soon as possible. But the scheduler will invoke it only after the currently executing script is complete.
So the function is scheduled to run "right after" the current script.
For instance, this outputs "Hello", then immediately "World":

```
setTimeout(() => alert("World"));
alert("Hello");
```

The first line "puts the call into calendar after 0ms". But the scheduler will only "check the calendar" after the current script is complete, so "Hello" is first, and "World" – after it.

Zero delay is not zero in a browser

In the browser, there's a limitation of how often nested timers can run. The HTML Living Standard says: "after five nested timers, the interval is forced to be at least 4 milliseconds.".

Let's demonstrate what it means with the example below. The setTimeout call in it re-schedules itself with zero delay. Each call remembers the real time from the previous one in the times array. What do the real delays look like? Let's see:

```
let start = Date.now();
let times = [];

setTimeout(function run() {
  times.push(Date.now() - start); // remember delay from the previous call

  if (start + 100 < Date.now()) alert(times); // show the delays after 100ms
  else setTimeout(run); // else re-schedule
});

// an example of the output:
// 1,1,1,1,9,15,20,24,30,35,40,45,50,55,59,64,70,75,80,85,90,95,100
```

First timers run immediately (just as written in the spec), and then we see 9, 15, 20, 24.... The 4+ ms obligatory delay between invocations comes into play.

The similar thing happens if we use setInterval instead of setTimeout: setInterval(f) runs f few times with zero-delay, and afterwards with 4+ ms delay.

That limitation comes from ancient times and many scripts rely on it, so it exists for historical reasons.

For server-side JavaScript, that limitation does not exist, and there exist other ways to schedule an immediate asynchronous job, like setImmediate for Node.js. So this note is browser-specific.

Summary

- Methods setTimeout(func, delay, ...args) and setInterval(func, delay, ...args) allow us to run the func once/regularly after delay milliseconds.
- To cancel the execution, we should call clearTimeout/clearInterval with the value returned by setTimeout/setInterval.
- Nested setTimeout calls are a more flexible alternative to setInterval, allowing us to set the time between executions more precisely.
- Zero delay scheduling with setTimeout(func, 0) (the same as setTimeout(func)) is used to schedule the call “as soon as possible, but after the current script is complete”.
- The browser limits the minimal delay for five or more nested calls of setTimeout or for setInterval (after 5th call) to 4ms. That’s for historical reasons.

Please note that all scheduling methods do not guarantee the exact delay.
For example, the in-browser timer may slow down for a lot of reasons:

- The CPU is overloaded.
- The browser tab is in the background mode.
- The laptop is on battery saving mode.
  All that may increase the minimal timer resolution (the minimal delay) to 300ms or even 1000ms depending on the browser and OS-level performance settings.

Decorators and forwarding, call/apply

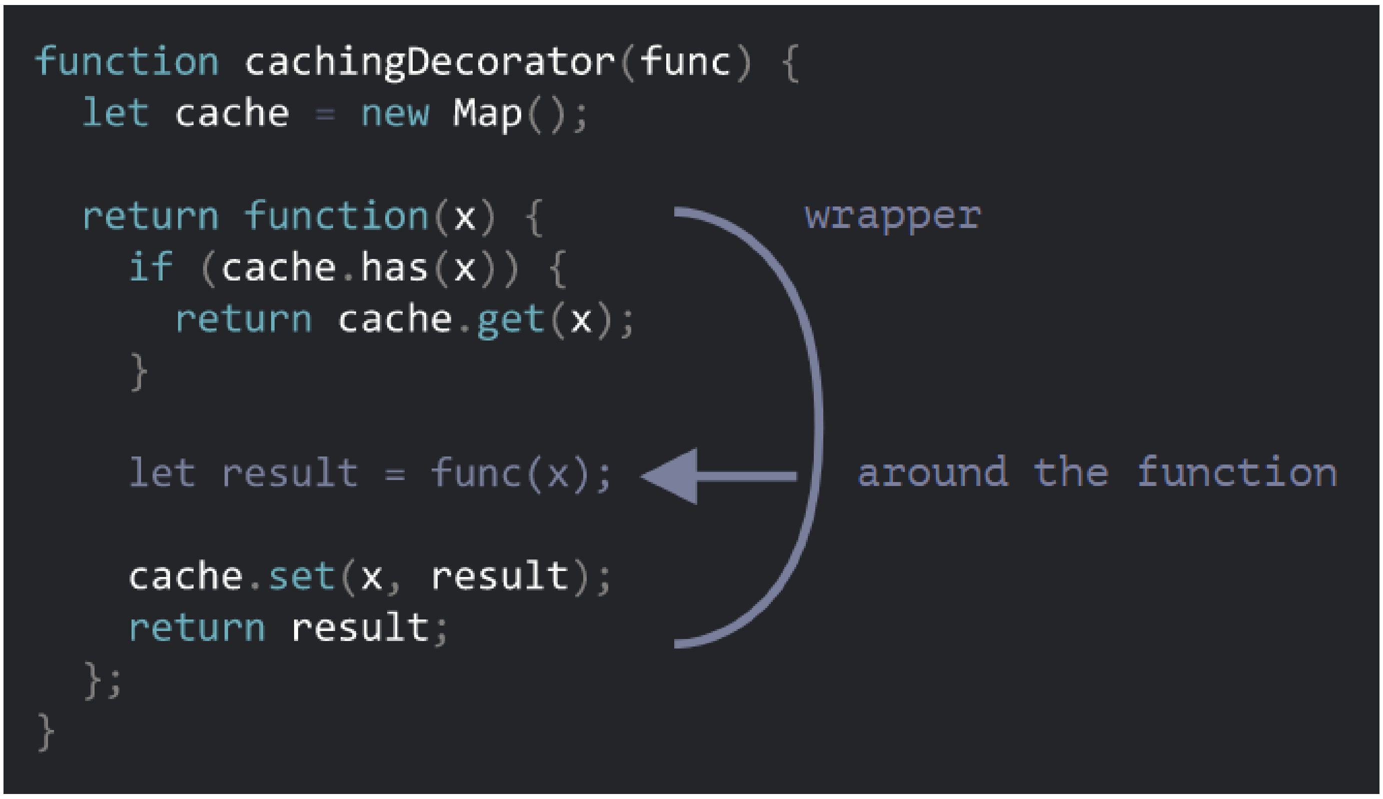
Javawscript gives extreme flexibilty with functions, they can be passed around, used as objects and now we'll see hot to **forward** calls between them and decorate them.

Transparent caching

Let’s say we have a function slow(x) which is CPU-heavy, but its results are stable. In other words, for the same x it always returns the same result.
If the function is called often, we may want to cache (remember) the results to avoid spending extra-time on recalculations.
But instead of adding that functionality into slow() we’ll create a **wrapper function, that adds caching**. As we’ll see, there are many benefits of doing so.
Here’s the code, and explanations follow:

```
function slow(x) {
  // there can be a heavy CPU-intensive job here
  alert(`Called with ${x}`);
  return x;
}

function cachingDecorator(func) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) {    // if there's such key in cache
      return cache.get(x); // read the result from it
    }

    let result = func(x);  // otherwise call func

    cache.set(x, result);  // and cache (remember) the result
    return result;
  };
}

slow = cachingDecorator(slow);

alert( slow(1) ); // slow(1) is cached and the result returned
alert( "Again: " + slow(1) ); // slow(1) result returned from cache

alert( slow(2) ); // slow(2) is cached and the result returned
alert( "Again: " + slow(2) ); // slow(2) result returned from cache
```

In the code above cachingDecorator is a decorator: a special function that takes another function and alters its behavior.
The idea is that we can call cachingDecorator for any function, and it will return the caching wrapper. That's great, because we can have many functions that could use such a feature, and all we need to do is to apply cachingDecorator to them.
By separating caching from the main function code we also keep the main code simpler.
The result of cachingDecorator(func) is a “wrapper”: function(x) that “wraps” the call of func(x) into caching logic:



```
function cachingDecorator(func) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }

    let result = func(x);

    cache.set(x, result);
    return result;
  };
}
```

From an outside code, the wrapped slow function still does the same. It just got a caching aspect added to its behavior.
To summarize, there are several benefits of using a separate cachingDecorator instead of altering the code of slow itself:

- The cachingDecorator is reusable. We can apply it to another function.
- The caching logic is separate, it did not increase the complexity of slow itself (if there was any).
- We can combine multiple decorators if needed (other decorators will follow).

func.call

The caching decorator mentioned above is not suited to work with object methods.
For instance, in the code below worker.slow() stops working after the decoration:

```
// we'll make worker.slow caching
let worker = {
  someMethod() {
    return 1;
  },

  slow(x) {
    // scary CPU-heavy task here
    alert("Called with " + x);
    return x * this.someMethod(); // (*)
  }
};

// same code as before
function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
    let result = func(x); // (**)
    cache.set(x, result);
    return result;
  };
}

alert( worker.slow(1) ); // the original method works

worker.slow = cachingDecorator(worker.slow); // now make it caching

alert( worker.slow(2) ); // Whoops! Error: Cannot read property 'someMethod'
of undefined
```

The error occurs in the line (*) that tries to access this.someMethod and fails. Can you see why?
The reason is that the wrapper calls the original function as func(x) in the line (**). And, when called like that, the function gets this = undefined.
We would observe a similar symptom if we tried to run:

```
let func = worker.slow;
func(2);
```

So, the wrapper passes the call to the original method, but without the context this. Hence the error.
Let’s fix it.
There’s a special built-in function method func.call(context, …args) that allows to call a function explicitly setting this.
The syntax is:

```
func.call(context, arg1, arg2, ...)
```

It runs func providing the first argument as this, and the next as the arguments.
To put it simply, these two calls do almost the same:

```
func.call(context, arg1, arg2, ...)
// and
func(1, 2, 3);
func.call(obj, 1, 2, 3)
```

They both call func with arguments 1, 2 and 3. The only difference is that func.call also sets this to obj.
As an example, in the code below we call sayHi in the context of different objects, sayHi.call(user) runs sayHi providing this=user, and the next line sets this=admin:

```
function sayHi() {
  alert(this.name);
}

let user = { name: "John" };
let admin = { name: "Admin" };

// use call to pass different objects as "this"
sayHi.call( user ); // John
sayHi.call( admin ); // Admin

// And here we use call to call say with the given context and phrase:

function say(phrase) {
  alert(this.name + ': ' + phrase);
}
let user = { name: "John" };
// user becomes this, and "Hello" becomes the first argument
say.call( user, "Hello" ); // John: Hello
```

In our case, we can use call in the wrapper to pass the context to the original function:

```
let worker = {
  someMethod() {
    return 1;
  },

  slow(x) {
    alert("Called with " + x);
    return x * this.someMethod(); // (*)
  }
};
```

```
function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
    let result = func.call(this, x); // "this" is passed correctly now
    cache.set(x, result);
    return result;
  };
}

worker.slow = cachingDecorator(worker.slow); // now make it caching

alert( worker.slow(2) ); // works
alert( worker.slow(2) ); // works, doesn't call the original (cached)
```

Now everything is fine.

To make it all clear, let’s see more deeply how this is passed along:

- After the decoration worker.slow is now the wrapper function (x) { ... }.
- So when worker.slow(2) is executed, the wrapper gets 2 as an argument and this=worker (it’s the object before dot).
- Inside the wrapper, assuming the result is not yet cached, func.call(this, x) passes the current this (=worker) and the current argument (=2) to the original method.

Multi argument caching

Now let’s make cachingDecorator even more universal. Till now it was working only with single-argument functions.
Now how to cache the multi-argument worker.slow method?

```
let worker = {
  slow(min, max) {
    return min + max; // scary CPU-hogger is assumed
  }
};

// should remember same-argument calls
worker.slow = cachingDecorator(worker.slow);
```

Previously, for a single argument x we could just cache.set(x, result) to save the result and cache.get(x) to retrieve it. But now we need to remember the result for a combination of arguments (min,max). The native Map takes single value only as the key.

There are many solutions possible:

- Implement a new (or use a third-party) map-like data structure that is more versatile and allows multi-keys.
- Use nested maps: cache.set(min) will be a Map that stores the pair (max, result). So we can get result as cache.get(min).get(max).
- Join two values into one. In our particular case we can just use a string "min,max" as the Map key. For flexibility, we can allow to provide a hashing function for the decorator, that knows how to make one value from many.
  For many practical applications, the 3rd variant is good enough, so we'll stick to it.

Also we need to pass not just x, but all arguments in func.call. Let's recall that in a function() we can get a pseudo-array of its arguments as arguments, so func.call(this, x) should be replaced with func.call(this, ...arguments).

```
let worker = {
  slow(min, max) {
    alert(`Called with ${min},${max}`);
    return min + max;
  }
};

function cachingDecorator(func, hash) {
  let cache = new Map();
  return function() {
    let key = hash(arguments); // (*)
    if (cache.has(key)) {
      return cache.get(key);
    }

    let result = func.call(this, ...arguments); // (**)

    cache.set(key, result);
    return result;
  };
}

function hash(args) {
  return args[0] + ',' + args[1];
}

worker.slow = cachingDecorator(worker.slow, hash);
```

```
alert( worker.slow(3, 5) ); // works
alert( "Again " + worker.slow(3, 5) ); // same (cached)
```

Now it works with any number of arguments (though the hash function would also need to be adjusted to allow any number of arguments. An interesting way to handle this will be covered below).

There are two changes:

- In the line (*) it calls hash to create a single key from arguments. Here we use a simple “joining” function that turns arguments (3, 5) into the key "3,5". More complex cases may require other hashing functions.
- Then (**) uses func.call(this, ...arguments) to pass both the context and all arguments the wrapper got (not just the first one) to the original function.

func.apply

Instead of func.call(this, ...arguments) we could use func.apply(this, arguments).
The syntax of built-in method func.apply is:

```
func.apply(context, args)

// These calls are equivalent:
func.call(context, ...args);
func.apply(context, args);
```

It runs the func setting this=context and using an array-like object args as the list of arguments.
The only syntax difference between call and apply is that call expects a list of arguments, while apply takes an array-like object with them.
They perform the same call of func with given context and arguments.
There’s only a subtle difference regarding args:

- The spread syntax ... allows to pass iterable args as the list to call.
- The apply accepts only array-like args.

And for objects that are both iterable and array-like, such as a real array, we can use any of them, but apply will probably be faster, because most JavaScript engines internally optimize it better. Passing all arguments along with the context to another function is called call forwarding.

That’s the simplest form of it:

```
let wrapper = function() {
  return func.apply(this, arguments);
};
```

When an external code calls such wrapper, it is indistinguishable from the call of the original function func.

Borrowing a method

Now let’s make one more minor improvement in the hashing function:

```
function hash(args) {
  return args[0] + ',' + args[1];
}

// It would be better if it worked with any number of args, the natural
solution would be to use arr.join method:
function hash(args) {
  return args.join();
}
```

Unfortunately, that won’t work. Because we are calling hash(arguments), and arguments object is both iterable and array-like, but not a real array.

```
// There is an easy way to use array join:
function hash() {
  alert( [].join.call(arguments) ); // 1,2
}

hash(1, 2);
```

The trick is called method borrowing.

We take (borrow) a join method from a regular array ([].join) and use [].join.call to run it in the context of arguments.

Why does it work?

That’s because the internal algorithm of the native method arr.join(glue) is very simple.
Taken from the specification almost “as-is”:

- Let glue be the first argument or, if no arguments, then a comma ",".
- Let result be an empty string.
- Append this[0] to result.
- Append glue and this[1].
- Append glue and this[2].
- …Do so until this.length items are glued.
- Return result.

So, technically it takes this and joins this[0], this[1] …etc together. It's intentionally written in a way that allows any array-like this (not a coincidence, many methods follow this practice). That's why it also works with this=arguments.

Decorators and function properties

It is generally safe to replace a function or a method with a decorated one, except for one little thing. If the original function had properties on it, like func.calledCount or whatever, then the decorated one will not provide them. Because that is a wrapper. So one needs to be careful if one uses them.
E.g. in the example above if slow function had any properties on it, then cachingDecorator(slow) is a wrapper without them.
Some decorators may provide their own properties. E.g. a decorator may count how many times a function was invoked and how much time it took, and expose this information via wrapper properties.
There exists a way to create decorators that keep access to function properties, but this requires using a special Proxy object to wrap a function.

Summary

Decorator is a wrapper around a function that alters its behavior. The main job is still carried out by the function.

Decorators can be seen as "features" or "aspects" that can be added to a function. We can add one or add many. And all this without changing its code!

To implement cachingDecorator, we studied methods:

func.call(context, arg1, arg2…) – calls func with given context and arguments.
func.apply(context, args) – calls func passing context as this and array-like args into a list of arguments.
The generic call forwarding is usually done with apply:

```
let wrapper = function() {
  return original.apply(this, arguments);
};
```

We also saw an example of method borrowing when we take a method from an object and call it in the context of another object. It is quite common to take array methods and apply them to arguments. The alternative is to use rest parameters object that is a real array.

There are many decorators there in the wild. Check how well you got them by solving the tasks of this chapter.

Exercises for this chapter can be found at: https://javascript.info/call-apply-decorators#tasks

Function Binding

When passing object methods as callbacks, like in setTimeout, there's a known problem: "losing **this**".We've already seen examples of losing this. Once a method is passed somewhere separately

from the object – this is lost.
Here’s how it may happen with setTimeout:

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

setTimeout(user.sayHi, 1000); // Hello, undefined!
```

That’s because setTimeout got the function user.sayHi, separately from the object. The last line can be rewritten as:

```
let f = user.sayHi;
setTimeout(f, 1000); // lost user context
```

The method setTimeout in-browser is a little special: it sets this=window for the function call (for Node.js, this becomes the timer object, but doesn’t really matter here). So for this.firstName it tries to get window.firstName, which does not exist. In other similar cases, usually this just becomes undefined.
The task is quite typical – we want to pass an object method somewhere else (here – to the scheduler) where it will be called. How to make sure that it will be called in the right context?

Using a wrapper

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

setTimeout(function() {
  user.sayHi(); // Hello, John!
}, 1000);

// This is the same but shorter
setTimeout(() => user.sayHi(), 1000); // Hello, John!
```

Now it works, because it receives user from the outer lexical environment, and then calls the method normally. Looks fine, but a slight vulnerability appears in our code structure.
What if before setTimeout triggers (there’s one second delay!) user changes value? Then, suddenly, it will call the wrong object!

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

setTimeout(() => user.sayHi(), 1000);

// ...the value of user changes within 1 second
user = {
  sayHi() { alert("Another user in setTimeout!"); }
};

// Another user in setTimeout!
```

Next solution guarantees that this does not happen.

function bind

Functions provide a built-in method that allows to fix this. The basic syntax is:

```
// more complex syntax will come a little later
let boundFunc = func.bind(context);
```

The result of func.bind(context) is a special function-like “exotic object”, that is callable as function and transparently passes the call to func setting this=context.
In other words, calling boundFunc is like func with fixed this.
For instance, here funcUser passes a call to func with this=user:

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

function func() {
  alert(this.firstName);
}

let funcUser = func.bind(user);
funcUser(); // John
```

Here func.bind(user) as a “bound variant” of func, with fixed this=user.
All arguments are passed to the original func “as is”, for instance:

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

function func(phrase) {
  alert(phrase + ', ' + this.firstName);
}

// bind this to user
let funcUser = func.bind(user);

funcUser("Hello"); // Hello, John (argument "Hello" is passed, and
this=user)

// With an object method:

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

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

// can run it without an object
sayHi(); // Hello, John!

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

// even if the value of user changes within 1 second
// sayHi uses the pre-bound value which is reference to the old user object
user = {
  sayHi() { alert("Another user in setTimeout!"); }
};
```

In the line (*) we take the method user.sayHi and bind it to user. The sayHi is a “bound” function, that can be called alone or passed to setTimeout – doesn’t matter, the context will be right.
Here we can see that arguments are passed “as is”, only this is fixed by bind:

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

let say = user.say.bind(user);

say("Hello"); // Hello, John! ("Hello" argument is passed to say)
say("Bye"); // Bye, John! ("Bye" is passed to say)
```

bindAll

If an object has many methods and we plan to actively pass it around, then we could bind them all in a loop:

```
for (let key in user) {
  if (typeof user[key] == 'function') {
    user[key] = user[key].bind(user);
  }
}
```

JavaScript libraries also provide functions for convenient mass binding , e.g. _.bindAll(object, methodNames) in lodash.

Partial functions

Until now we have only been talking about binding this. Let’s take it a step further.
We can bind not only this, but also arguments. That’s rarely done, but sometimes can be handy.
The full syntax of bind:

```
let bound = func.bind(context, [arg1], [arg2], ...);

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

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

alert( double(3) ); // = mul(2, 3) = 6
alert( double(4) ); // = mul(2, 4) = 8
alert( double(5) ); // = mul(2, 5) = 10
```

The call to mul.bind(null, 2) creates a new function double that passes calls to mul, fixing null as the context and 2 as the first argument. Further arguments are passed “as is”.
That’s called partial function application – we create a new function by fixing some parameters of the existing one.
Please note that we actually don’t use this here. But bind requires it, so we must put in something like null. The function triple in the code below triples the value:

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

let triple = mul.bind(null, 3);

alert( triple(3) ); // = mul(3, 3) = 9
alert( triple(4) ); // = mul(3, 4) = 12
alert( triple(5) ); // = mul(3, 5) = 15
```

Why do we usually make a partial function?

The benefit is that we can create an independent function with a readable name (double, triple). We can use it and not provide the first argument every time as it’s fixed with bind.
In other cases, partial application is useful when we have a very generic function and want a less universal variant of it for convenience.
For instance, we have a function send(from, to, text). Then, inside a user object we may want to use a partial variant of it: sendTo(to, text) that sends from the current user.

Partial functions without context

What if we'd like to fix some arguments, but not the context this? For example, for an object method.
The native bind does not allow that. We can't just omit the context and jump to arguments.
Fortunately, a function partial for binding only arguments can be easily implemented.
Like this:

```
function partial(func, ...argsBound) {
  return function(...args) { // (*)
    return func.call(this, ...argsBound, ...args);
  }
}

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

// add a partial method with fixed time
user.sayNow = partial(user.say, new Date().getHours() + ':' + new
Date().getMinutes());

user.sayNow("Hello");
// Something like:
// [10:00] John: Hello!
```

The result of partial(func[, arg1, arg2...]) call is a wrapper (*) that calls func with:

- Same this as it gets (for user.sayNow call it's user)
- Then gives it ...argsBound – arguments from the partial call ("10:00")
- Then gives it ...args – arguments given to the wrapper ("Hello")

So easy to do it with the spread syntax, right?
Also there's a ready _.partial implementation from lodash library.

Summary

Method func.bind(context, ...args) returns a "bound variant" of function func that fixes the context this and first arguments if given.

Usually we apply bind to fix this for an object method, so that we can pass it somewhere. For example, to setTimeout.

When we fix some arguments of an existing function, the resulting (less universal) function is called partially applied or partial.

Partials are convenient when we don't want to repeat the same argument over and over again. Like if we have a send(from, to) function, and from should always be the same for our task, we can get a partial and go on with it.

Arrow functions revisited

Arrow functions are not just a shorthand for writing small stuff, they have some very specific and useful features.
For instance:

- arr.forEach(func) – func is executed by forEach for every array item.
- setTimeout(func) – func is executed by the built-in scheduler.
  and more..

Arrow functions have no this

If this is accessed, it is taken from the outside.
For instance, we can use it to iterate inside an object method:

```
  let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],

  showList() {
    this.students.forEach(
      student => alert(this.title + ': ' + student)
    );
  }
};

group.showList();
```

Here in forEach, the arrow function is used, so this.title in it is exactly the same as in the outer method showList. That is: group.title.
If we used a "regular" function, there would be an error, the error occurs because forEach runs functions with this=undefined by default, so the attempt to access undefined.title is made. That doesn't affect arrow functions, because they just don't have this.

Arrow funcionts can't run with new : Not having this means that they can't be used as constructors as they can't be called with new.

There's a subtle difference between an arrow function => and a regular function called with .bind(this):

- .bind(this) creates a “bound version” of the function.
- The arrow => doesn’t create any binding. The function simply doesn’t have this. The lookup of this is made exactly the same way as a regular variable search: in the outer lexical environment.

Arrows have no arguments

Arrows also have no arguments variable. That's great for decorators, when we need to forward a call with the current this and arguments. That’s great for decorators, when we need to forward a call with the current this and arguments.

For instance, defer(f, ms) gets a function and returns a wrapper around it that delays the call by ms milliseconds:

```
function defer(f, ms) {
  return function() {
    setTimeout(() => f.apply(this, arguments), ms);
  };
}

function sayHi(who) {
  alert('Hello, ' + who);
}

let sayHiDeferred = defer(sayHi, 2000);
sayHiDeferred("John"); // Hello, John after 2 seconds
```

The same without arrow functions would look like:

```
function defer(f, ms) {
  return function(...args) {
    let ctx = this;
    setTimeout(function() {
      return f.apply(ctx, args);
    }, ms);
  };
}
```

Here we had to create additional variables args and ctx so that the function inside setTimeout could take them.

Summary

Arrow functions:

- Do not have this

- Do not have arguments
- Can’t be called with new
- They also don’t have super, but we didn’t study it yet. We will on the chapter Class inheritance
- That’s because they are meant for short pieces of code that do not have their own “context”, but rather work in the current one. And they really shine in that use case.