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:

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).

Using "func.call" for the context

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(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:

1. After the decoration `worker.slow` is now the wrapper `function (x) { ... }`.
2. So when `worker.slow(2)` is executed, the wrapper gets `2` as an argument and `this=worker` (it's the object before dot).
3. 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.

Going multi-argument

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:

1. Implement a new (or use a third-party) map-like data structure that is more versatile and allows multi-keys.
2. 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)`.

3. 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)`.

   Here's a more powerful `cachingDecorator`:

```
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)
```

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.

So these two calls are almost equivalent:

```
func.call(context, ...args);
func.apply(context, args);
```

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];
}
```

As of now, it works only on two arguments. It would be better if it could glue 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.

So calling `join` on it would fail, as we can see below:

```
function hash() {
  alert( arguments.join() ); // Error: arguments.join is not a function
}

hash(1, 2);
```

Still, there's 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":

1. Let `glue` be the first argument or, if no arguments, then a comma `","`.
2. Let `result` be an empty string.
3. Append `this[0]` to `result`.
4. Append `glue` and `this[1]`.
5. Append `glue` and `this[2]`.

6. ...Do so until `this.length` items are glued.
7. 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`.

Bind

Function binding

When passing object methods as callbacks, for instance to `setTimeout`, there's a known problem: "losing `this`".

In this chapter we'll see the ways to fix it.

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!
```

As we can see, the output shows not "John" as `this.firstName`, but `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?

Solution 1: a wrapper

The simplest solution is to use a wrapping function:

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

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

Now it works, because it receives `user` from the outer lexical environment, and then calls the method normally.

The same, but shorter:

```
setTimeout(() => user.sayHi(), 1000); // Hello, John!
```

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!
```

The next solution guarantees that such thing won't happen.

Solution 2: bind

Functions provide a built-in method bind 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)
```

Now let's try 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)
```

Convenience method: `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], ...);
```

It allows to bind context as `this` and starting arguments of the function.

For instance, we have a multiplication function `mul(a, b)`:

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

Let's use `bind` to create a function `double` on its base:

```
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.

Going partial 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.

What is shallow copy and deep copy in JavaScript ?

Shallow Copy: When a reference variable is copied into a new reference variable using the [assignment operator](#), a shallow copy of the referenced object is created. In simple words, a reference variable mainly stores the address of the object it refers to. When a new reference variable is assigned the value of the old reference variable, the address stored in the old reference variable is copied into the new one. This means both the old and new reference variable point to the same object in memory. As a result, if the state of the object changes through any of the reference variables it is reflected for both. Let us take an example to understand it better.

Code Implementation:

```
let employee = {
        eid: "E102",
        ename: "Jack",
        eaddress: "New York",
        salary: 50000
}

console.log("Employee=> ", employee);
let newEmployee = employee; // Shallow copy
console.log("New Employee=> ", newEmployee);

console.log("---------After modification----------");
newEmployee.ename = "Beck";
console.log("Employee=> ", employee);
console.log("New Employee=> ", newEmployee);
```

```
// Name of the employee as well as

// newEmployee is changed.
```

Output:

```
E:\gfg>node emp.js
Employee=>  { eid: 'E102', ename: 'Jack', eaddress: 'New York', salary: 50000 }
New Employee=>  { eid: 'E102', ename: 'Jack', eaddress: 'New York', salary: 50000 }
-------------------------After modification-------------------------
Employee=>  { eid: 'E102', ename: 'Beck', eaddress: 'New York', salary: 50000 }
New Employee=>  { eid: 'E102', ename: 'Beck', eaddress: 'New York', salary: 50000 }
```

Explanation: From the above example, it is seen that when the name of newEmployee is modified, it is also reflected for the old employee object. This can cause data inconsistency. This is known as a shallow copy. The newly created object has the same memory address as the old one. Hence, any change made to either of them changes the attributes for both. To overcome this problem, a deep copy is used. If one of them is removed from memory, the other one ceases to exist. In a way the two objects are interdependent.

Deep Copy: Unlike the shallow copy, deep copy makes a copy of all the members of the old object, allocates a separate memory location for the new object, and then assigns the copied members to the new object. In this way, both the objects are independent of each other and in case of any modification to either one, the other is not affected. Also, if one of the objects is deleted the other still remains in the memory. Now to create a deep copy of an object in JavaScript we use JSON.parse() and JSON.stringify() methods. Let us take an example to understand it better.

Code Implementation:

```
let employee = {

        eid: "E102",

        ename: "Jack",

        eaddress: "New York",

        salary: 50000

}

console.log("=========Deep Copy========");

let newEmployee = JSON.parse(JSON.stringify(employee));

console.log("Employee=> ", employee);

console.log("New Employee=> ", newEmployee);

console.log("---------After modification---------");
```

```
newEmployee.ename = "Beck";

newEmployee.salary = 70000;

console.log("Employee=> ", employee);

console.log("New Employee=> ", newEmployee);
```

```
E:\gfg>node emp.js
==========================Deep Copy==========================
Employee=>  { eid: 'E102', ename: 'Jack', eaddress: 'New York', salary: 50000 }
New Employee=>  { eid: 'E102', ename: 'Jack', eaddress: 'New York', salary: 50000 }
--------------------------After modification------------------------
Employee=>  { eid: 'E102', ename: 'Jack', eaddress: 'New York', salary: 50000 }
New Employee=>  { eid: 'E102', ename: 'Beck', eaddress: 'New York', salary: 70000 }
```

Explanation: Here the new object is created using the JSON.parse() and JSON.stringify() methods of JavaScript. JSON.stringify() takes a JavaScript object as an argument and then transforms it into a JSON string. This JSON string is passed to the JSON.parse() method which then transforms it into a JavaScript object. This method is useful when the object is small and has serializable properties. But if the object is very large and contains certain non-serializable properties then there is a risk of data loss. Especially if an object contains methods then JSON.stringify() will fail as methods are non-serializable. There are better ways to a deep clone of which one is Lodash which allows cloning methods as well.

Lodash To Deep Copy: Lodash is a JavaScript library that provides multiple utility functions and one of the most commonly used functions of the Lodash library is the cloneDeep() method. This method helps in the deep cloning of an object and also clones the non-serializable properties which were a limitation in the JSON.stringify() approach.

Code Implementation:

```
const lodash = require('lodash');

let employee = {

    eid: "E102",

    ename: "Jack",

    eaddress: "New York",

    salary: 50000,

    details: function () {

        return "Employee Name: "

            + this.ename + "-->Salary: "
```

```
                              + this.salary;
          }
    }

    let deepCopy = lodash.cloneDeep(employee);
    console.log("Original Employee Object");
    console.log(employee);
    console.log("Deep Copied Employee Object");
    console.log(deepCopy);
    deepCopy.eid = "E103";
    deepCopy.ename = "Beck";
    deepCopy.details = function () {
          return "Employee ID: " + this.eid
                    + "-->Salary: " + this.salary;
    }
    console.log("----------After Modification----------");
    console.log("Original Employee Object");
    console.log(employee);
    console.log("Deep Copied Employee Object");
    console.log(deepCopy);
    console.log(employee.details());
    console.log(deepCopy.details());
```

```
Original Employee Object
{
  eid: 'E102',
  ename: 'Jack',
  eaddress: 'New York',
  salary: 50000,
  details: [Function: details]
}
Deep Copied Employee Object
{
  eid: 'E102',
  ename: 'Jack',
  eaddress: 'New York',
  salary: 50000,
  details: [Function: details]
}
---------------------------------After Modification----------------------------
Original Employee Object
{
  eid: 'E102',
  ename: 'Jack',
  eaddress: 'New York',
  salary: 50000,
  details: [Function: details]
}
Deep Copied Employee Object
{
  eid: 'E103',
  ename: 'Beck',
  eaddress: 'New York',
  salary: 50000,
  details: [Function]
}
Employee Name: Jack-->Salary: 50000
Employee ID: E103-->Salary: 50000
```

Explanation: Both objects have different properties after the modification. Also, the methods of each object are differently defined and produce different outputs.

Are let and const hoisted?

What is hoisting?

The JavaScript engine before parses the code before executing and during the parsing phase it shifts all the **variable declaration** to the top of the scope. This behavior of the JS engine is called **hoisting**.

Variable Hoisting

Consider the following code snippet –

```
console.log(greeting); // undefined

var greeting = "Hello";
```

We can see that the `greeting` variable can be accessed before its declared. This happens because the JS engine modifies our code snippet into something like this -

```
var greeting;
console.log(greeting); // undefined

var greeting = "Hello";
```

Function Hoisting

The formal function declarations in JavaScript are also hoisted to the top of the scope. For example:

```
greeting(); // Hello

function greeting() {
  console.log("Hello");
}
```

Note: The important distinction between **variable hoisting** and **function hoisting** is that a `var` variable is hoisted and then auto-initialized to `undefined` whereas a function declaration is hoisted and **initialized to its function value**.

Function declaration vs Function expression

Function hoisting only applies to formal `function` declarations and not to `function` expression assignments. Consider:

```
greeting(); // TypeError: greeting is not a function

console.log(greeting); // undefined

var greeting = function greeting() {
  console.log("Hello!");
};
```

Above, we can see that the greeting variable was hoisted but it was not initialized with the function reference. The engine throws us a TypeError: greeting is not a function and not ReferenceError: greeting is not defined. The function expression assignments behave very much like **variable hoisting**.

What about let and const?

So far, I have only talked about var and formal function declarations. What about the let and const. Let's see the following code snippet -

```
console.log(greeting); // cannot access 'greeting' before initialization

let greeting = "Hello";
```

We get a new kind of error, its not a ReferenceError, the engine knows about greeting but doesn't allow us to use it before its initialized. The JS engine doesn't allow us to access the variables declared with let and const before they are declared. This is called **Temporal Dead Zone**.

Let's consider this snippet -

```
let greeting;

console.log(greeting); // undefined

greeting = "Hello";
```

Above, we can see that we are able to access the greeting variable as soon as it was declared.

So, let and const are not hoisted?

After seeing the above two code snippets, I was pretty convinced too that let and const are not hoisted. But they actually are. We can prove this with the help of a few more examples -

```
console.log(typeof iDontExist); // undefined
console.log(typeof greeting); // cannot access 'greeting' before initialization

let greeting = "hello";
```

If the greeting variable was not hoisted, we would expect typeof greeting to be undefined similar to typeof iDontExist. This proves that the JS engine knows about our greeting variable but still doesn't allow us to access it just yet due to **Temporal Dead Zone**.

Let's see another example -

```
let x = 'outer value';
console.log(x); // outer value

 {
  // start TDZ for x
  console.log(x); // cannot access 'x' before initialization
  let x = 'inner value'; // declaration ends TDZ for x
 }
```

Accessing the variable x in the inner scope still causes the TDZ error. If the let x = 'inner value'; was not hoisted then on line 6, it would have logged outer value.

Conclusion

- The var declarations are hoisted and initialized with undefined.
- The formal function declarations are hoisted and initialized with their function reference.
- let and const variables are hoisted too but they cannot be accessed before their declarations. This is called Temporal Dead Zone.
