Cristi Salcescu Follow
Enthusiastic about sharing ideas @Senior Systems Engineer at Systematic
May 14 · 4 min read

You will finally understand what Closure is

Closure is a nested function that has access to the parent variables, even after the parent has executed.

Consider the following code:

```
function init(){
  let state = 0;

  function increment(){
    state += 1;
    return state;
  }

  increment();
}

init();
```

The function `increment()` is a closure.

`increment()` is a nested function. `init()` is its parent. `increment()` accesses the variable `state` from its parent.

Closure becomes important when the nested function survives the invocation of the parent.

Look at these examples:

- Timer callback

```
function initTimer(){
  let state = 0;

  setTimeout(function increment(){
    state += 1;
    return state;
  }, 60000);
}
initTimer();
```

- Event callback

```
function initEvent(){
  let state = 0;

  $("#btn").on("click", function increment(){
    state += 1;
    return state;
  });
}
initEvent();
```

- AJAX callback

```
function initFetch(){
  let state = 0;

  fetch(url).then(function increment(){
    state += 1;
    return state;
  });
}
initFetch();
```

In all these cases `increment()` survives the invocation of the parent. `increment()` is a closure.

Encapsulation

Function with private state

Closure can be created as a function with private state.

Consider the folowing code :

```
let types = ["History", "Travel", "Biographies"];
function createRandomItem(arr){
  return function randomItem(){
    let index = Math.floor(Math.random() * arr.length);
    return arr[index];
  }
}
```

```
let randomType = createRandomItem(types);
randomType();
randomType();
```

`randomItem()` is a nested function that has access to the variable `arr` from its parent. `randomItem()` lives after `createRandomItem()` execution. `randomItem()` is a closure.

`randomType()` points to the same function as `randomItem()` . `randomType()` is a closure. It gives a random book type.

Object with private state

As you've seen, we can create a function with private state. At the same time, we can create many functions sharing the same private state. By doing so, we create an object with private state.

Look at the next code:

```
function BookStore(){
  let books = [];

  function add(book){
    books.push(book);
  }

  function toViewObject(book){
    return Object.freeze(book);
```

```
    }

    function get(){
      books.map(toViewObject).slice(0);
    }

    return Object.freeze({
      add,
      get
    });
  }
```

```
let bookStore = BookStore();
bookStore.add({title : "JS The Good Parts"});
bookStore.add({title : "Functional-Light JS"});
bookStore.add({title : "JS Allongé"});
```

```
console.log(bookStore.get());
```

The `BookStore` function creates an object with private state. The `books` variable is private, it can't be access from the outside. `add` and `get` are two closures sharing the same private state.

`BookStore` is a factory function.

> *For an in-depth comparison between factory function and class take a look at Class vs Factory function: exploring the way forward*

Closure in Functional Programming

Functions taking a function as argument an returning a variation of that function use closure.

Let's look at a simple decorator like `not()` :

```
function not(fn){
  return function decorator(...args){
    return !fn.apply(this, args);
  }
}
```

The `decorator()` function has access to `fn` parameter from its parent.

`decorator()` is a closure.

Closures are highly used in functional programming. For example, all functions from underscore function section create closures.

> *For more decorators take a look at Here are a few function decorators you can write from scratch.*

Closure vs Pure Functions

To make things easier to reason about we can split nested functions in two: closures and pure functions.

- **Pure functions** don't use variables from the outer functions. Besides, they return a value and have no side effects.
- **Closure functions** use variables from the outer functions.

Variables lifetime

The parent variables live as long as the closure function lives.

Let's analyze some situations:

- `setInterval` : variables live the `interval` time
- `setTimout` : variables live forever, till `clearTimeout` is called
- event : variables live forever, till the remove handler is called
- AJAX : variables live till the response gets back from the server and the callback is executed

If a variable is shared by many closures, all closures should be garbage-collected before the variable is garbage-collected.

Conclusion

Closure is a nested function. It has access to variables from the parent functions.

Variables used by the closure function live as long as the closure lives.

Closure make it easy to work with async tasks like timers, events, AJAX calls.

More on Functional Programming in JavaScript :

How point-free composition will make you a better functional programmer

How to make your code better with intention-revealing function names

Here are a few function decorators you can write from scratch

Make your code easier to read with Functional Programming