Function Expressions

We know how to make functions.

```
function fun() {
        //
}
```

To declare a function the first item of the instruction is the function keyword. So whenever the first token of the instruction to declare a function starts with `function` keyword we call it as `function declaration`.

In JS, functions are first class citizen, why ?

- We can return a function from a function
- we can pass a function as an argument to another
- We can also store a function in a variable
- and more..

Let's focus more on storing function in a variable.

```
const myFun = function fun() {
        // ....
}
```

Here we are creating a function but the first valid token is not the `function` keyword, hence we call this type of instruction as `function expression`.

```
const myFun = function fun(x) {
    console.log("calling...", x);
}

// to call the function defined in the variable
myFun(10); // we can just give the variable name and pass the required args in a pair of parenthesis
```

There are more ways to defined function expression:
Example 1:

```
const myFun = function fun(x) { // named function expression
    console.log("calling...", x);
}
```

**Example 2:

```
const myFun = function (x) { // anonymous function expression
    console.log("calling...", x);
}
```

Example 3:

```
const myFun = () => { // arrow function / arrow function expression
    console.log("calling...", x);
}
```

Example 4:

```
(function fun(x) { // IIFE - Immediately invoked function expression
    console.log("calling...", x);
})(10)
```

Named Function Expression

A function expression with a name attached to it is called as named function expression.

```
const myFun = function fun(x) { // named function expression
    console.log("calling...", x);
}
```

Now here the name `fun` is the name of the function expression and this function expression is stored in a variable `myFun`.

Anonymous Function Expression

A function expression without a name attached to it is called as anonymous function expression.

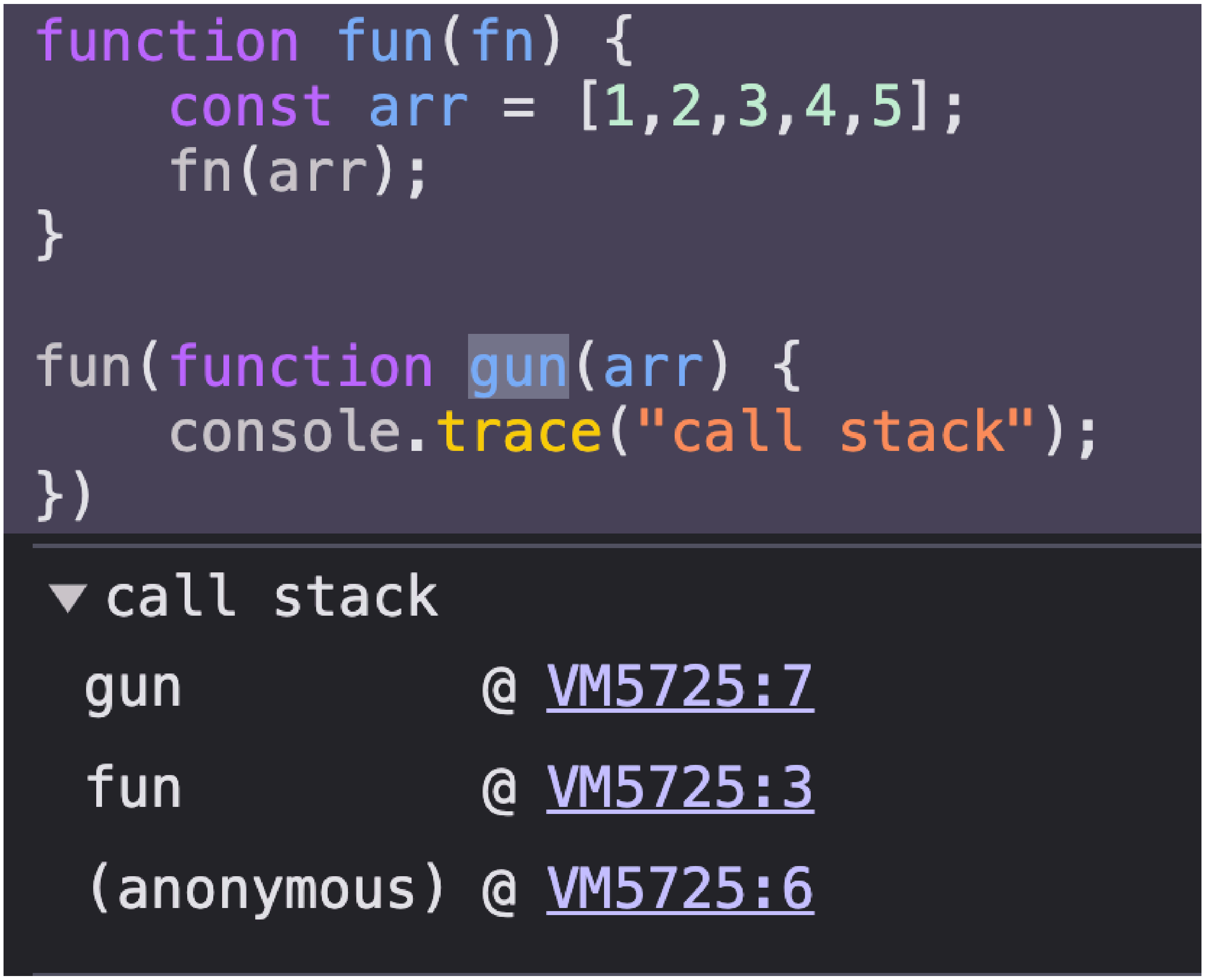
```
const myFun = function (x) { // named function expression
    console.log("calling...", x);
}
```

Now here the function expression has no name and this function expression is stored in a variable called as `myFun`.

Should we use named function expression or anonymous function expression ?

- Named function expression improves readability of the code. Because anonymous function expression don't have name, we have to read their whole logic to understand what they are doing, where as if properly named, then named function expressions are more understandable directly by the name.
- Anonymous functions are hard to debug. Because when you check the call stack of the functions, then you will not function any name for anonymous function expression.

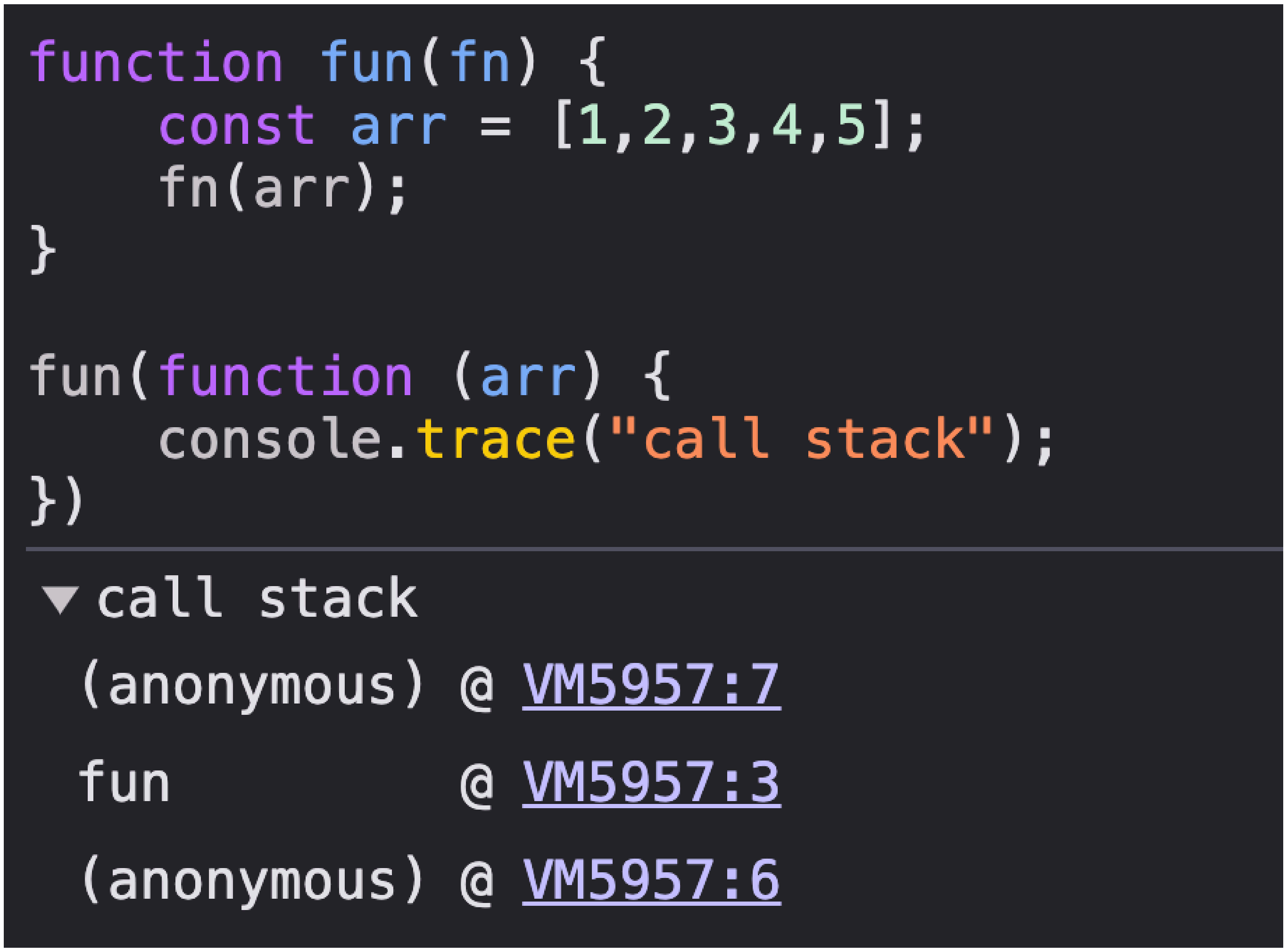
```
function fun(fn) {
        const arr = [1,2,3,4,5];
        fn(arr);
}

fun(function gun(arr) {
        console.trace("call stack");
})
```



```
function fun(fn) {
    const arr = [1,2,3,4,5];
    fn(arr);
}

fun(function gun(arr) {
    console.trace("call stack");
})
```

```
▼ call stack
  gun         @ VM5725:7
  fun         @ VM5725:3
  (anonymous) @ VM5725:6
```

Now in the above code, we can see the stack prints name of each function called.



```
function fun(fn) {
    const arr = [1,2,3,4,5];
    fn(arr);
}

fun(function (arr) {
    console.trace("call stack");
})
```

```
▼ call stack
  (anonymous) @ VM5957:7
  fun         @ VM5957:3
  (anonymous) @ VM5957:6
```

In the above code, the function expression is anonymous hence the call stack trace is not having any name entry, so if we have say 10 back to back function expression calls then it will be having 10 entries without a name, making hard to debug.

- Anonymous functions are hard to use in recursion whereas named function expression can be easily integrated in a recursive environment. To understand this let's take an example with array.map.
    - map is an inbuilt function for arrays in JS. It takes an argument which is expected to be a function. The function which we pass as an argument is automatically internally called by `map` function. This function expression that we pass is expected to have a parameter. Inside this parameter map function automatically passes one by one all the elements of our array. And then whatever is returned collectively by all the function expression calls, the returned values are populated in a new array and returned.

```
const arr = [1,2,3,4,5];
const returnValue = arr.map(function f(element) {
        return element * 2;
})
console.log(returnValue); // [2,4,6,8,10]
```

- Anonymous function expressions are useful when code logic is too simple, or if we are storing the anonymous function expression in a variable and that variable explains most of the logic of the function expression, then also there is no hard in using anonymous function expression.

Note: If you want to implement your own map function then:

```
function customMap(arr, fn) {

    const result = []; // new aresult array

    for(let i = 0; i < arr.length; i++) { // go to one by one to all
elements of the given array
        result.push(fn(arr[i])); // call the fn on the element and store
result in the result array
    }

    return result; // return the result
}
```

Let's say I want to use map function to calculate factorial of every single value in a given array. In this case, if we use a recursive approach then we are bound to use named function expression.

```
arr.map(function factorial(n) {
    if(n == 1) return 1;
    return n * factorial(n-1);
})
```

Here to call the function recursively we need to know the name of the function, hence named function expression become superior in this case.

How will you call recursion on anonymous function expression ?

```
arr.map(function (n) {
    if(n == 1) return 1;
    return n * (n-1); // what to put here ?  this code is wrong
})
```

There is a deprecated function called as `arguments.callee` which can be used to apply recursion.

```
arr.map(function (n) {
    if(n == 1) return 1;
    return n * arguments.callee(n-1); // deprecated function, not
recommended
});
```

Arrow functions

This function expression is generally preferred for concise syntax.

- There is one very fundamental difference in using arrow functions and other function expression i.e. in arrow functions the `this` keyword is resolved lexically whereas in other the `this` keyword is resolved by the call site.
- If arrow function has only one line logic i.e. to return something then we don't need to use return keyword.

```
const square = (x) => x*x;

const cube = function (x) {
        return x**3;
}
```

- In the above piece of code the only think square is expected to do is return `x*x` so using arrow function we don't need to write return keyword
- If the arrow function is only taking a single argument then parenthesis is not mandatory

```
const square = x => x*x;
arr.map(x => x*x); // one mroe example
```

- If there are more than 1 or no arguments then parenthesis is required

Where can we use these function expressions ?

- Function expression can be used to pass a function as an argument to another function.

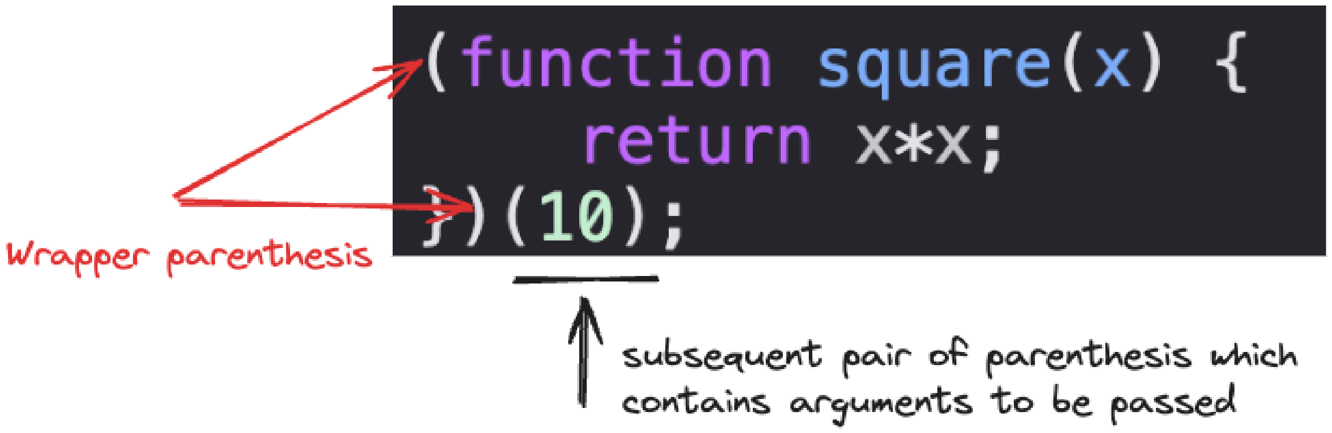
```
function fun(fn) {
        // some logic
        fn();
}

fun(function f() { ... });
```

IIFE (Immediately invoked function expression)

A function expression who is called the moment we define it is called as IIFE.

```
(function square(x) {
        return x*x;
})(10);
```

To define an IIFE, we first wrap a function inside a pair of parenthesis and then immediately call the function by putting a subsequent pair of parenthesis and pass arguments inside it.



```
(function square(x) {
    return x*x;
})(10);
```

Wrapper parenthesis
subsequent pair of parenthesis which contains arguments to be passed

Once we have prepared the IIFE and called it, post that we can never use it again. because IIFE get's wiped off from the memory once the execution is done.

- IIFE can be very useful to avoid name conflict because they don't conflict with those variables who are in outer scopes.
- IIFE can be useful for some temporary logic as well.