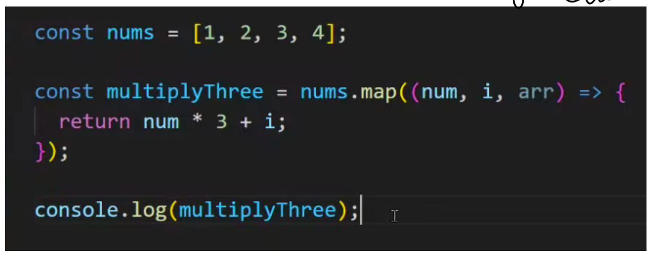
Polyfills for functions

Thursday, 24 August 2023 12:42 PM

→ what is a polyfill?

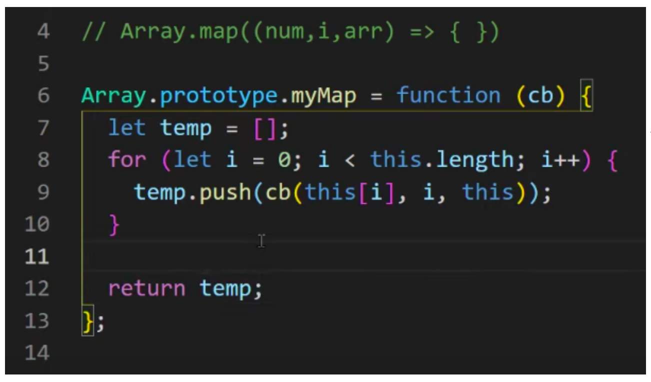
ans. A polyfill is a type of a browser fallback. Eg: what if your browser doesn't have bind() fn? Then you'll have to create your own custom fn for bind(). This is a polyfill.

→ Polyfill for map(): It creates ~~a~~ new array from an existing one by applying a fn to each of the array's elements.



```
const nums = [1, 2, 3, 4];

const multiplyThree = nums.map((num, i, arr) => {
  return num * 3 + i;
});

console.log(multiplyThree);
```

→ We can write the map() fn like this & it can take 3 arguments 'num' (values inside array), 'i' (index) & 'arr' (array which it's pointing to).



```
4   // Array.map((num,i,arr) => { })
5
6   Array.prototype.myMap = function (cb) {
7     let temp = [];
8     for (let i = 0; i < this.length; i++) {
9       temp.push(cb(this[i], i, this));
10    }
11
12    return temp;
13  };
14
```

→ We use Array.prototype to give access of myMap() to all arrays using . (dot) operator.

→ Anonymous fn gets a callback fn as argument (This callback fn is the logic written in line 4's {})

→ Now, we need a new array, since map() returns a new array.

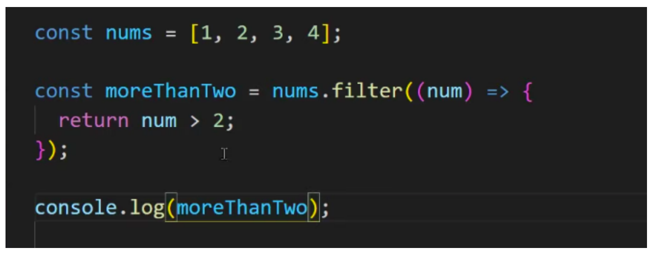
→ this here points to the array myMap() will be attached to.

→ so, this.length is array.length.

→ Inside the callback fn we'll pass (num, i, arr) which is accessed by (this[i], i, this) respectively.

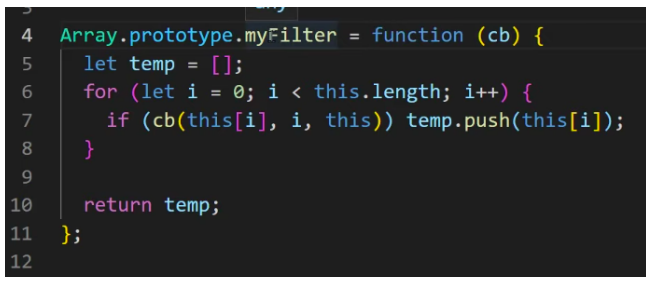
→ Finally we return the array.

→ Polyfill for filter(): It creates a new array by taking each element of the array & applying a condition to each, if the statement is true, then the element gets pushed into the array else the element does not get pushed.



```
const nums = [1, 2, 3, 4];

const moreThanTwo = nums.filter((num) => {
  return num > 2;
});

console.log(moreThanTwo);
```

→ It can also have (num, i, arr).

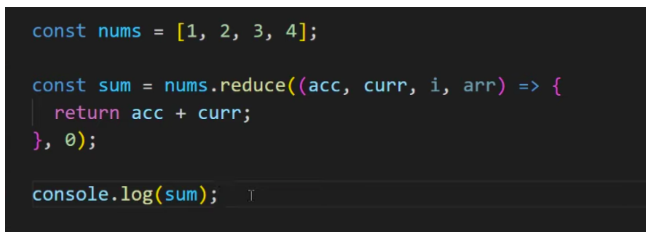


```
Array.prototype.myFilter = function (cb) {
  let temp = [];
  for (let i = 0; i < this.length; i++) {
    if (cb(this[i], i, this)) temp.push(this[i]);
  }

  return temp;
};
```

→ Everything for this polyfill will be same as polyfill for map().

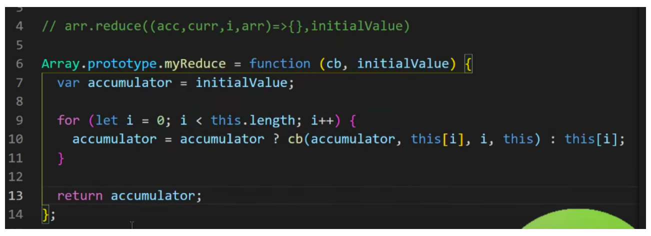
→ the difference would be just we only pass the elements to array if the callback fn returns true.

→ Polyfill for reduce():



```
const nums = [1, 2, 3, 4];

const sum = nums.reduce((acc, curr, i, arr) => {
  return acc + curr;
}, 0);

console.log(sum);
```

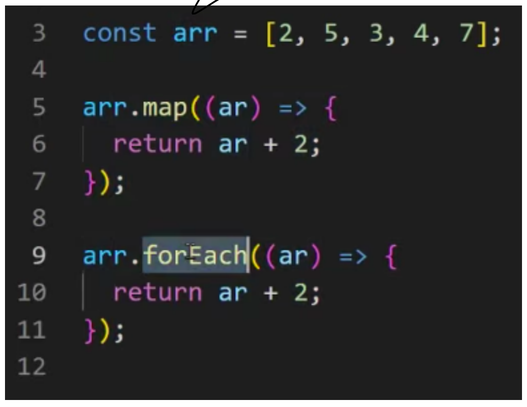
→ It can take the following arguments.
→ If we do not give value to 'acc' then reduce() automatically takes 'acc' as the first value of the nums[] array.



```
// arr.reduce((acc,curr,i,arr)=>{},initialValue)

Array.prototype.myReduce = function (cb, initialValue) {
  var accumulator = initialValue;

  for (let i = 0; i < this.length; i++) {
    accumulator = accumulator ? cb(accumulator, this[i], i, this) : this[i];
  }

  return accumulator;
};
```

→ Here the polyfill takes callback fn & initialValue as arguments.
→ We assign accumulator with initialValue.
→ Now, we run a for loop & add a condition that if initialValue is passed then run the callback fn with (acc, curr, i, arr) i.e (acc, this[i], i, this) respectively otherwise assign accumulator as the first element of array.
→ Since our loop has run 1 time, thus our curr becomes the second element of array.
→ Finally, accumulator is returned.

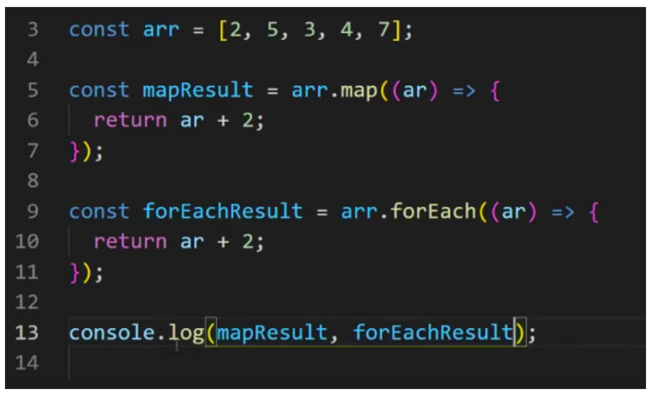
→ Difference between map() and forEach():
ans. These both are fn's used to loop through each of the elements of the array.



```
const arr = [2, 5, 3, 4, 7];

arr.map((ar) => {
  return ar + 2;
});

arr.forEach((ar) => {
  return ar + 2;
});
```

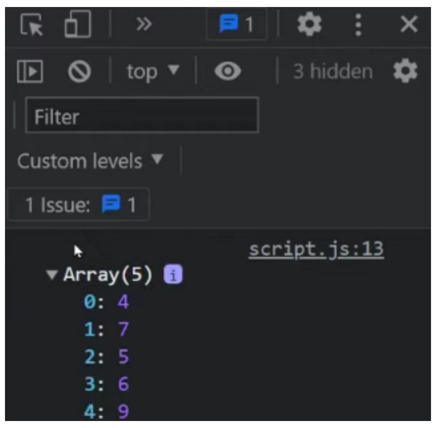
→ Syntax is same for both.
→ Both have (num, i, arr) as their arguments.

Difference1: map() returns a new array whereas forEach() does not return anything & prints undefined if result is stored somewhere.



```
const arr = [2, 5, 3, 4, 7];

const mapResult = arr.map((ar) => {
  return ar + 2;
});

const forEachResult = arr.forEach((ar) => {
  return ar + 2;
});

console.log(mapResult, forEachResult);
```

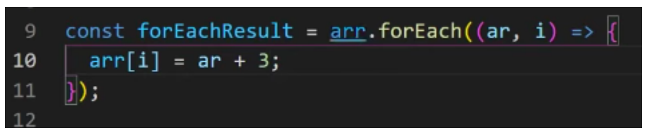
o/p:



```
top ▼   3 hidden
Filter
Custom levels ▼
1 Issue: 1
                     script.js:13
▼Array(5)
   0: 4
   1: 7
   2: 5
   3: 6
   4: 9
```

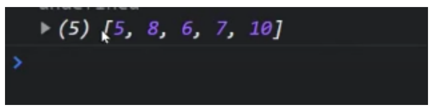
```
length: 5
▸ [[Prototype]]: Array(0)
undefined
>
```

→ map() & forEach() don't modify the original array however they can do so by setting the original elements' in the callback f^n.



```
const forEachResult = arr.forEach((ar, i) => {
  arr[i] = ar + 3;
});
```

→ Thus, this changes the original array. We can do the same with map() also to change the original array.

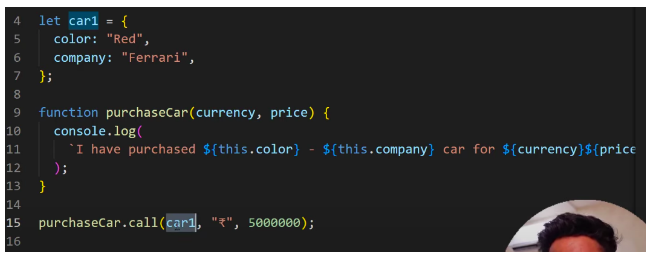


```
▸ (5) [5, 8, 6, 7, 10]
>
```

→ o/p of above

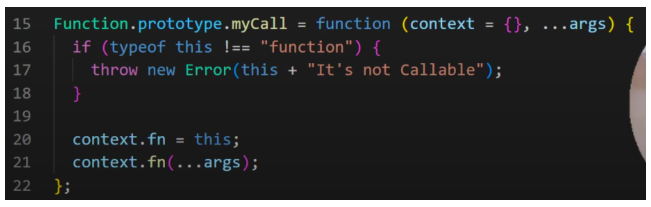
Difference 2: We can chain other methods with map(), but since forEach() doesn't return any array we can't chain methods to it.

→ Poly fill for call ():



```
let car1 = {
  color: "Red",
  company: "Ferrari",
};

function purchaseCar(currency, price) {
  console.log(
    `I have purchased ${this.color} - ${this.company} car for ${currency}${price
  );
}

purchaseCar.call(car1, "₹", 5000000);
```

→ Normal call () f^n takes in a context and other arguments.



```
Function.prototype.myCall = function (context = {}, ...args) {
  if (typeof this !== "function") {
    throw new Error(this + "It's not Callable");
  }

  context.fn = this;
  context.fn(...args);
};
```

→ Polyfill for call()

→ We use ~~Function~~.prototype so that myCall() becomes available to all functions when using dot operator.

→ We pass context as argument in call() f^n so, we accept 'context' as parameter & make its value as "{ }" by default.

→ We can pass many other arguments after context, so we accept them as parameters using rest operator.

→ We then check if the "typeof this" i.e f^n on which myCall() is called is whether a f^n or not, if its not then we throw an error.

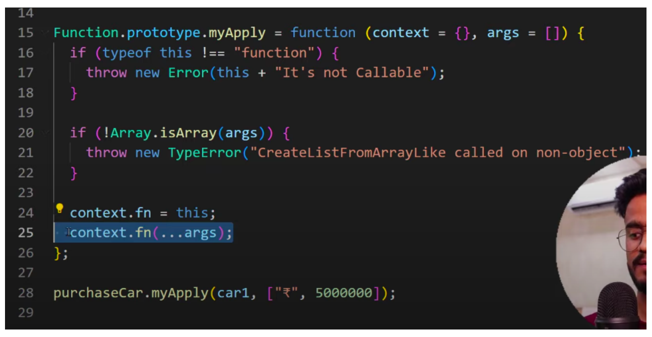
→ Otherwise we create a new key in 'fn' & assign its value as 'this' i.e the f^n on which myCall() is called.

→ We call that f^n using "context.fn()" & pass the "...args" to it which are the arguments this f^n (on which call() is applied) asks for.

Note: When we do "purchaseCar.call(car1)", we explicitly bind purchaseCar() to "car1" which is such that it becomes the method of 'car1', so "this"

to car1 object such that it becomes the method of car1, so this keyword inside "purchaseCar" points to "car1."
~~But~~ since .call() method is being called on "purchaseCar()" so "this" inside .call() method points to "purchaseCar()" because basically we are binding .call() to "purchaseCar()" implicitly (using dot operator).

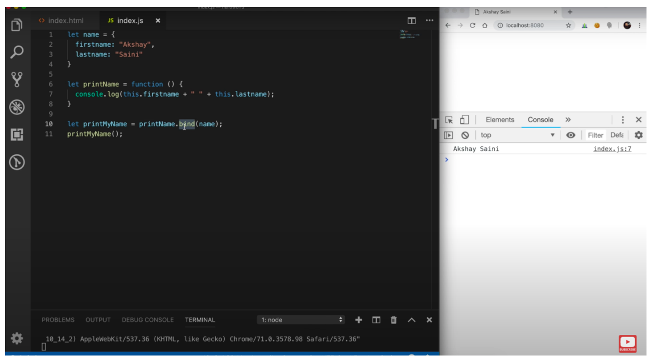
→ Poly fill for apply () :



```
14
15  Function.prototype.myApply = function (context = {}, args = []) {
16    if (typeof this !== "function") {
17      throw new Error(this + "It's not Callable");
18    }
19
20    if (!Array.isArray(args)) {
21      throw new TypeError("CreateListFromArrayLike called on non-object");
22    }
23
24    context.fn = this;
25    context.fn(...args);
26  };
27
28  purchaseCar.myApply(car1, ["₹", 5000000]);
29
```

→ Polyfill for apply () remains same, the only differences are that it accepts an array of arguments so we need to provide a default value of "[ ]" if argument array is not passed.
→ Also we add another check to check whether "args" array passed is an array or not.
→ We pass the arguments using spread operator in line 25 since fⁿ will take separate values.

→ Poly fill for bind () :

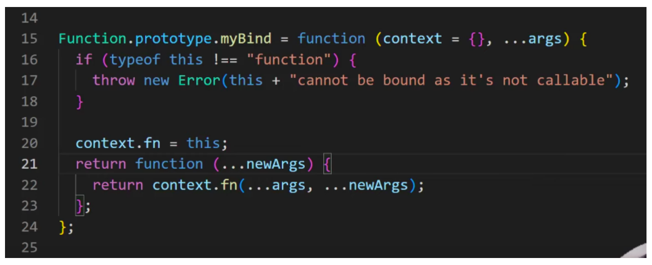


```
let name = {
  firstname: "Akshay",
  lastname: "Saini"
}

let printName = function () {
  console.log(this.firstname + " " + this.lastname);
}

let printMyName = printName.bind(name);
printMyName();
```

```
Akshay Saini                                   index.js:7
```

```
10_14_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/71.0.3578.98 Safari/537.36"
```

→ How we use the bind () fⁿ.
→ bind() fⁿ binds the fⁿ to the object whose reference is passed as an argument to bind() & returns the copy of the fⁿ which can be invoked later.



```
Function.prototype.myBind = function (context = {}, ...args) {
  if (typeof this !== "function") {
    throw new Error(this + "cannot be bound as it's not callable");
  }

  context.fn = this;
  return function (...newArgs) {
    return context.fn(...args, ...newArgs);
  };
};
```

→ So this is same as call() & apply() poly fill. The difference being that here we return a fⁿ & we can pass arguments normally through bind() fⁿ or can send arguments through the reusable fⁿ where we'll store the returned fⁿ. Hence both returned fⁿ & myBind() fⁿ will accept "...args".