Solution

Initialize a new array to store the mapped results. As we loop through the array (via `this`), call the callback on each array element with the following parameters: `element`, `index`, `array`, and `this`. This can be done by either using `Function.prototype.call` or `Function.prototype.apply`.

JavaScript TypeScript

```
/**
 * @template T, U
 * @param { (value: T, index: number, array: Array<T>) => U } callbackFn
 * @param {any} [thisArg]
 * @return {Array<U>}
 */
Array.prototype.myMap = function (callbackFn, thisArg) {
  const len = this.length;
  const array = new Array(len);

  for (let k = 0; k < len; k++) {
    // Ignore index if value is not defined for index (e.g. in sparse arrays).
    if (Object.hasOwn(this, k)) {
      array[k] = callbackFn.call(thisArg, this[k], k, this);
    }
  }

  return array;
};
```

Edge cases

- Passing the `index` and `array` to the map callback.
- Calling the map callback with the correct `this` if `thisArg` is specified.
- Sparse arrays (e.g. `[1, 2, , 4]`). The empty values should be ignored while traversing the array.

Notes

Mutating the array in the map callback is a bad idea and can cause unintended consequences. It is a positive signal to mention that mutation of the array within the callback is possible. The provided solution follows the

TC39 specification for array mutation scenarios:

- The range of elements processed by `map` is set before the first callback is called.
- Elements appended to the array after the call to `map` begins will not be visited by the callback.
- If existing elements of the array are changed, their value as passed to the callback will be the value at the time `map` visits them.
- Elements that are deleted after the call to `map` begins and before being visited are not visited.

The `thisArg` doesn't do anything if the callback is defined as an arrow function as arrow functions don't have their own bindings to `this`.

One-liner solution

You can cheat the autograder by doing this:

```
Array.prototype.myMap = Array.prototype.map;
```

Spec solution

Here's a solution that is based off the `Array.prototype.map` ECMAScript specification.

```
Array.prototype.myMap = function (callbackFn, thisArg) {
  if (
    typeof callbackFn !== 'function' ||
    !callbackFn.call ||
    !callbackFn.apply
  ) {
    throw new TypeError(`${callbackFn} is not a function`);
  }

  const len = this.length;
  const A = new Array(len);
  let k = 0;

  while (k < len) {
    // Ignore index if value is not defined for index (e.g. in sparse arrays).
    const kPresent = Object.hasOwn(this, k);
    if (kPresent) {
      const kValue = this[k];
    const mappedValue = callbackFn.call(thisArg, kValue, k, this);
    A[k] = mappedValue;
  }
  k = k + 1;
 }

 return A;
};
```

Resources

- `Array.prototype.map` | MDN
- `Array.prototype.map` ECMAScript specification