Solution

Initialize a new array to store the filtered 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 be done by either using `Function.prototype.call` or `Function.prototype.apply`.

If the callback evaluates to `true`, push the element into the `results`.

JavaScript TypeScript

```ts
interface Array<T> {
  myFilter(
    callbackFn: (value: T, index: number, array: Array<T>) => boolean,
    thisArg?: any,
  ): Array<T>;
}

Array.prototype.myFilter = function (callbackFn, thisArg) {
  const len = this.length;
  const results = [];

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

  return results;
};
```

Edge cases

- Passing the `index` and `array` to the filter callback.
- Invoking the filter 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 filter 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 `filter` is set before the first callback is called.
- Elements appended to the array after the call to `filter` 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 `filter` visits them.
- Elements that are deleted after the call to `filter` 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.myFilter = Array.prototype.filter;
```

Spec solution

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

```
Array.prototype.myFilter = 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 = [];
```

```
  let k = 0;
  let to = 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 selected = Boolean(callbackFn.call(thisArg, kValue, k, this));
      if (selected === true) {
        A[to] = kValue;
        to += 1;
      }
    }
    k += 1;
  }

  return A;
};
```