Solution

There are a few aspects to this question we need to bear in mind and handle:

1. `Promise`s are meant to be chained, so the function needs to return a `Promise`.
2. If the input array is empty, the returned `Promise` resolves with an empty array.
3. The returned `Promise` contains an array of resolved values in the same order as the input if all of them are fulfilled.
4. The returned `Promise` rejects immediately if any of the input values are rejected or throw an error.
5. The input array can contain non-`Promise`s.

Approach 1: Count unresolved promises using `async`

Since the function needs to return a `Promise`, we can construct a `Promise` at the top level of the function and return it. The bulk of the code will be written within the constructor parameter.

We first check if the input array is empty, and resolve with an empty array if so.

We then need to attempt resolving every item in the input array. This can be achieved using `Array.prototype.forEach` or `Array.prototype.map`. As the returned values will need to preserve the order of the input array, we create a `results` array and slot the value in the right place using its `index` within the input array. To know when all the input array values have been resolved, we keep track of how many unresolved promises there are by initializing a counter of unresolved values and decrementing it whenever a value is resolved. When the counter reaches 0, we can return the `results` array.

One thing to note here is that because the input array can contain non-`Promise` values, if we are not `await`-ing them, we need to wrap each value with `Promise.resolve()` which allows us to use `.then()` on each of them and we don't have to differentiate between `Promise` vs non-`Promise` values and whether they need to be resolved.

Lastly, if any of the values are rejected, we reject the top-level `Promise` immediately without waiting for any other pending promises.

JavaScript TypeScript

```
/**
 * @param {Array} iterable
 * @return {Promise<Array>}
 */
```

```
export default function promiseAll(iterable) {
 return new Promise((resolve, reject) => {
  const results = new Array(iterable.length);
  let unresolved = iterable.length;

  if (unresolved === 0) {
   resolve(results);
   return;
  }

  iterable.forEach(async (item, index) => {
   try {
    const value = await item;
    results[index] = value;
    unresolved -= 1;

    if (unresolved === 0) {
     resolve(results);
    }
   } catch (err) {
    reject(err);
   }
  });
 });
}
```

Approach 2: Count unresolved promises using `Promise.then`

Here's an alternative version which uses `Promise.then()` if you prefer not to use `async` / `await`.

```
/**
 * @param {Array} iterable
 * @return {Promise<Array>}
 */
export default function promiseAll(iterable) {
 return new Promise((resolve, reject) => {
  const results = new Array(iterable.length);
  let unresolved = iterable.length;

  if (unresolved === 0) {
   resolve(results);
```

```
      return;
    }

    iterable.forEach((item, index) => {
      Promise.resolve(item).then(
        (value) => {
          results[index] = value;
          unresolved -= 1;

          if (unresolved === 0) {
            resolve(results);
          }
        },
        (reason) => {
          reject(reason);
        },
      );
    });
  });
}
```

Once one of the `Promise`'s resolving functions (`resolve` or `reject`) is called, the promise is in the "settled" state, and subsequent calls to either function can neither change the fulfillment value or rejection reason nor change the eventual state from "fulfilled" to "rejected" or vice versa.

Edge cases

- Empty input array. An empty array should be returned.
- If the array contains non-`Promise` values, they will still be part of the returned array if all the input values are fulfilled.
- If the outcome is a rejection, the value of the first rejection should be returned.

Techniques

- Knowledge of `Promise`s, how to construct one, how to use them.
- Async programming.

Notes

- The evaluator does not verify that your input array is resolved concurrently rather than sequentially.