guide # typescript

Array Methods

TCSS 460 — Client/Server Programming

Array methods are the primary way you will transform data throughout this course — in Express route handlers, React components, and everywhere in between. When your server fetches a list of movies from an external API, you will filter out irrelevant entries, extract the fields your client needs, and reshape the response before sending it back. When your React front end receives that data, you will use the same methods to render lists and compute derived state. These operations replace the traditional `for` loops you wrote in Java with concise, expressive, and composable function calls.

---

1 Why Array Methods Matter

Every web API deals with collections of data. A database query returns rows of users. An external API responds with an array of movies. A client sends a batch of IDs to look up. In every case, you need to transform that data before passing it along.

In Java (TCSS 142-305), you likely wrote `for` loops to iterate over `ArrayList` elements, building up new lists by hand. Java 8 introduced the Stream API to make this more declarative, and if you used Streams in TCSS 305, the transition to TypeScript array methods will feel natural. If you haven't used Streams, don't worry — TypeScript's approach is simpler because arrays already have these methods built in. No `.stream()` call, no `.collect()` at the end.

Here is a preview of what you will learn in this guide. Consider this Express route handler that fetches movies from an external API and transforms the response for a client:

```
app.get("/api/movies/top-rated", async (req, res) => {
  const response = await
fetch("https://api.themoviedb.org/3/movie/top_rated");
  const data = await response.json();

  const movies = data.results
    .filter((movie: any) => movie.vote_average >= 8.0)
    .map((movie: any) => ({
      id: movie.id,
      title: movie.title,
```

```
      rating: movie.vote_average,
    }))
    .sort((a: any, b: any) => b.rating - a.rating);

  res.json({ movies });
});
```

Three methods — `filter`, `map`, `sort` — replace what would be dozens of lines with a `for` loop, temporary arrays, and manual sorting. By the end of this guide, you will be able to read, write, and chain these methods with confidence.

---

2 forEach

The simplest array method is `forEach`. It calls a function once for each element in the array, in order.

2.1 Basic Usage

```
const genres: string[] = ["Action", "Comedy", "Drama", "Horror"];

genres.forEach((genre) => {
  console.log(`Genre: ${genre}`);
});
```

Output:

```
Genre: Action
Genre: Comedy
Genre: Drama
Genre: Horror
```

The callback function receives each element as its first argument. You can also access the index and the original array:

```
genres.forEach((genre, index) => {
  console.log(`${index + 1}. ${genre}`);
});
```

2.2 Java Comparison

In Java, you have used the enhanced `for` loop and possibly `forEach` with a lambda:

```java
List<String> genres = List.of("Action", "Comedy", "Drama", "Horror");

// Enhanced for loop
for (String genre : genres) {
    System.out.println("Genre: " + genre);
}

// forEach with lambda (Java 8+)
genres.forEach(genre -> System.out.println("Genre: " + genre));
```

In TypeScript, `forEach` works the same way as Java's `forEach` with a lambda. The syntax is nearly identical — arrow functions (`=>`) replace Java's arrow operator (`->`).

2.3 When NOT to Use forEach

`forEach` is for side effects — logging, updating external state, sending responses. It does **not** return a value.

⚠ Prefer `map`, `filter`, or `reduce` Over `forEach`

`forEach` returns `undefined`, cannot be chained, and encourages mutation. In almost every case, there is a better method:

- **Building a new array?** Use `map` (Section 3).
- **Filtering items?** Use `filter` (Section 4).
- **Accumulating a result?** Use `reduce` (Section 7).

If you find yourself pushing into an empty array inside a `forEach`, that is a sign you should be using `map` or `filter`:

```typescript
// BAD — forEach with mutation
const upperGenres: string[] = [];
genres.forEach((genre) =>
upperGenres.push(genre.toUpperCase()));

// GOOD — map returns a new array directly
const upperGenres = genres.map((genre) => genre.toUpperCase());
```

Reserve `forEach` for genuine side effects: logging, sending emails, updating external state — situations where you do not need a return value.

Try It Yourself

1. Create a file called `array-practice.ts`
2. Define an array of five movie titles as strings
3. Use `forEach` to print each title with its index number
4. Try assigning the result of `forEach` to a variable — confirm it is `undefined`
5. Run with `npx ts-node array-practice.ts`

---

3 map — Transform Every Element

`map` is the most important array method you will use in this course. It creates a new array by applying a transformation function to every element of the original array.

3.1 Basic Usage

```
interface User {
  id: number;
  name: string;
  email: string;
  age: number;
}

const users: User[] = [
  { id: 1, name: "Alice", email: "alice@example.com", age: 28 },
  { id: 2, name: "Bob", email: "bob@example.com", age: 34 },
  { id: 3, name: "Carol", email: "carol@example.com", age: 22 },
];

// Extract just the names
const names: string[] = users.map((user) => user.name);
console.log(names); // ["Alice", "Bob", "Carol"]
```

The callback function receives each element and returns a transformed value. The returned array always has the same length as the original.

3.2 Reshaping Objects

In web APIs, `map` is commonly used to reshape data — taking a complex object from an external API and returning only the fields your client needs:

```
interface MovieApiResponse {
  id: number;
  title: string;
  overview: string;
  vote_average: number;
  vote_count: number;
  poster_path: string;
  release_date: string;
  genre_ids: number[];
  adult: boolean;
  original_language: string;
}

interface MovieSummary {
  id: number;
  title: string;
  rating: number;
  posterUrl: string;
}

const apiMovies: MovieApiResponse[] = [
  {
    id: 550, title: "Fight Club", overview: "An insomniac...",
    vote_average: 8.4, vote_count: 26000, poster_path:
"/pB8BM7pdSp6B6Ih7QZ4DrQ3PmJK.jpg",
    release_date: "1999-10-15", genre_ids: [18, 53], adult: false,
    original_language: "en",
  },
  {
    id: 680, title: "Pulp Fiction", overview: "The lives of two mob
hitmen...",
    vote_average: 8.5, vote_count: 25000, poster_path:
"/d5iIlFn5s0ImszYzBPb8JPIfbXD.jpg",
    release_date: "1994-09-10", genre_ids: [80, 53], adult: false,
    original_language: "en",
  },
];

const summaries: MovieSummary[] = apiMovies.map((movie) => ({
  id: movie.id,
  title: movie.title,
  rating: movie.vote_average,
  posterUrl: `https://image.tmdb.org/t/p/w500${movie.poster_path}`,
}));

console.log(summaries);
// [
//   { id: 550, title: "Fight Club", rating: 8.4, posterUrl:
"https://image.tmdb.org/t/p/w500/pB8B..." },
//   { id: 680, title: "Pulp Fiction", rating: 8.5, posterUrl:
"https://image.tmdb.org/t/p/w500/d5iI..." },
// ]
```

Notice the parentheses around the object literal: `(movie) => ({...})`. Without them, TypeScript interprets the curly braces as a function body, not an object. This is a common

syntax trap.

Parentheses Around Returned Object Literals

When returning an object literal from an arrow function, you must wrap it in parentheses:

```
// CORRECT — parentheses tell TS this is an object, not a
function body
const result = items.map((item) => ({ name: item.name }));

// WRONG — TS thinks { name: item.name } is a function body with
a label
const result = items.map((item) => { name: item.name });
```

3.3 Java Comparison

In Java, the equivalent operation uses the Stream API:

```
List<String> names = users.stream()
    .map(user -> user.getName())
    .collect(Collectors.toList());
```

In TypeScript, `map` is a method directly on arrays — no `.stream()` to start and no `.collect()` to finish:

```
const names: string[] = users.map((user) => user.name);
```

This is one of the first things that feels simpler in TypeScript. Arrays are already "streamable." You call the method, you get a new array back.

3.4 Immutability

`map` never modifies the original array. It always returns a new one. This is important in web API code where you may need the original data later (for logging, error recovery, or other transformations).

```
const original = [1, 2, 3];
const doubled = original.map((n) => n * 2);

console.log(original); // [1, 2, 3] — unchanged
console.log(doubled);  // [2, 4, 6] — new array
```

Try It Yourself

1. Define an array of at least four `User` objects (with `id`, `name`, `email`, and `age` fields)
2. Use `map` to create a new array containing only `{ name, email }` for each user
3. Use `map` again to create an array of display strings like `"Alice (age 28)"`
4. Verify that the original array is unchanged after both operations

---

4 filter — Keep What Matches

`filter` creates a new array containing only the elements that pass a test. The callback function returns `true` to keep an element or `false` to discard it.

4.1 Basic Usage

```
interface Movie {
  id: number;
  title: string;
  rating: number;
  releaseYear: number;
  language: string;
}

const movies: Movie[] = [
  { id: 1, title: "Parasite", rating: 8.5, releaseYear: 2019, language: "ko"
},
  { id: 2, title: "The Room", rating: 3.6, releaseYear: 2003, language: "en"
},
  { id: 3, title: "Inception", rating: 8.4, releaseYear: 2010, language: "en"
},
  { id: 4, title: "Spirited Away", rating: 8.6, releaseYear: 2001, language:
"ja" },
  { id: 5, title: "The Godfather", rating: 8.7, releaseYear: 1972, language:
"en" },
];

// Keep only highly rated movies
const topMovies: Movie[] = movies.filter((movie) => movie.rating >= 8.0);

console.log(topMovies.length); // 4
console.log(topMovies.map((m) => m.title));
// ["Parasite", "Inception", "Spirited Away", "The Godfather"]
```

The returned array may be shorter than the original (or the same length, or empty), depending on how many elements pass the test.

4.2 Filtering in an Express Route

Filtering is especially common when you want to let clients narrow down results via query parameters:

```
app.get("/api/movies", (req, res) => {
  const minRating = parseFloat(req.query.minRating as string) || 0;
  const language = req.query.language as string | undefined;

  let results = movies;

  if (minRating > 0) {
    results = results.filter((movie) => movie.rating >= minRating);
  }

  if (language) {
    results = results.filter((movie) => movie.language === language);
  }

  res.json({ count: results.length, movies: results });
});
```

A request to `/api/movies?minRating=8.0&language=en` would return only English-language movies rated 8.0 or higher.

4.3 Java Comparison

In Java with Streams:

```
List<Movie> topMovies = movies.stream()
    .filter(movie -> movie.getRating() >= 8.0)
    .collect(Collectors.toList());
```

In TypeScript:

```
const topMovies: Movie[] = movies.filter((movie) => movie.rating >= 8.0);
```

Again, no `.stream()` and no `.collect()`. The method lives directly on the array.

4.4 Filtering `null` and `undefined`

A practical use of `filter` is removing `null` or `undefined` values from an array. However, TypeScript's type system needs help understanding that the result no longer contains those values:

```ts
const values: (string | null)[] = ["Alice", null, "Bob", null, "Carol"];

// TypeScript still thinks the result type is (string | null)[]
const withoutNulls = values.filter((v) => v !== null);

// Use a type predicate for proper narrowing
const names: string[] = values.filter(
  (v): v is string => v !== null
);
```

The `v is string` syntax is a **type predicate** — it tells TypeScript that if the function returns `true`, the value is guaranteed to be a `string`. You will encounter this pattern when working with API responses that may contain `null` fields.

Try It Yourself

1. Using the `movies` array defined above, filter for movies released after 2000
2. Filter for movies whose title contains the letter "a" (case-insensitive) — hint: use `.toLowerCase().includes()`
3. Combine both filters by chaining: `.filter(...).filter(...)`
4. Verify the original array still has all five movies

---

5 find and findIndex

While `filter` returns all matching elements, `find` returns the **first** element that matches. If no element matches, it returns `undefined`.

5.1 find

```ts
const users: User[] = [
  { id: 1, name: "Alice", email: "alice@example.com", age: 28 },
  { id: 2, name: "Bob", email: "bob@example.com", age: 34 },
  { id: 3, name: "Carol", email: "carol@example.com", age: 22 },
];

const user: User | undefined = users.find((u) => u.id === 2);
console.log(user); // { id: 2, name: "Bob", email: "bob@example.com", age: 34
}

const missing: User | undefined = users.find((u) => u.id === 999);
console.log(missing); // undefined
```

Notice the return type: `User | undefined`. TypeScript knows that `find` might not find anything, so it forces you to handle both cases.

5.2 find in an Express Route

`find` is the natural choice for "get one by ID" endpoints:

```
app.get("/api/users/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const user = users.find((u) => u.id === id);

  if (!user) {
    res.status(404).json({ error: "User not found" });
    return;
  }

  res.json(user);
});
```

5.3 findIndex

`findIndex` works like `find` but returns the **index** of the first matching element, or `-1` if not found:

```
const index: number = users.findIndex((u) => u.name === "Carol");
console.log(index); // 2

const missingIndex: number = users.findIndex((u) => u.name === "Dave");
console.log(missingIndex); // -1
```

This is useful when you need to update or remove an element by position.

5.4 Java Comparison

In Java, finding a single element in a list typically means a `for` loop with a `break`, or using Streams:

```
// Java Stream approach
Optional<User> user = users.stream()
    .filter(u -> u.getId() == 2)
    .findFirst();

if (user.isPresent()) {
    System.out.println(user.get().getName());
} else {
```

```
    System.out.println("Not found");
}
```

Java wraps the result in an `Optional` to handle the not-found case. TypeScript uses `undefined` instead:

```
const user: User | undefined = users.find((u) => u.id === 2);

if (user) {
  console.log(user.name);
} else {
  console.log("Not found");
}
```

Both approaches force you to handle the missing case, but TypeScript's version is more concise. There is no `Optional` wrapper — just the value or `undefined`.

Try It Yourself

1. Use `find` to look up a movie by title from the `movies` array
2. Try finding a movie that does not exist — confirm you get `undefined`
3. Write a short `if/else` that handles both the found and not-found cases
4. Use `findIndex` to find the index of a movie, then use bracket notation to access it: `movies[index]`

---

6 reduce — Accumulate a Result

`reduce` processes every element in an array and accumulates a single result. It is the most powerful array method — and the hardest to read.

6.1 Basic Usage

```
const prices: number[] = [9.99, 24.99, 4.99, 14.99];

const total: number = prices.reduce((sum, price) => sum + price, 0);
console.log(total); // 54.96
```

`reduce` takes two arguments:

1. **A callback** with two parameters: the **accumulator** (running result) and the **current element**

2. **An initial value** for the accumulator (here, `0`)

On each iteration, the callback returns the new accumulator value. After all elements are processed, `reduce` returns the final accumulator.

Here is the step-by-step execution:

Iteration	`sum` (accumulator)	`price` (current)	Returns
1	0	9.99	9.99
2	9.99	24.99	34.98
3	34.98	4.99	39.97
4	39.97	14.99	54.96

6.2 Building Objects with reduce

`reduce` can accumulate any type, not just numbers. A common pattern is building a lookup object (dictionary) from an array:

```
interface Movie {
  id: number;
  title: string;
  rating: number;
}

const movies: Movie[] = [
  { id: 550, title: "Fight Club", rating: 8.4 },
  { id: 680, title: "Pulp Fiction", rating: 8.5 },
  { id: 155, title: "The Dark Knight", rating: 8.5 },
];

// Build a lookup map: id -> movie
const movieById: Record<number, Movie> = movies.reduce(
  (acc, movie) => {
    acc[movie.id] = movie;
    return acc;
  },
  {} as Record<number, Movie>
);

console.log(movieById[680]); // { id: 680, title: "Pulp Fiction", rating: 8.5
}
```

6.3 Counting and Grouping

Another practical use is counting occurrences or grouping elements:

```
const genres: string[] = [
  "Action", "Comedy", "Action", "Drama", "Comedy", "Comedy", "Horror",
];

const counts: Record<string, number> = genres.reduce(
  (acc, genre) => {
    acc[genre] = (acc[genre] || 0) + 1;
    return acc;
  },
  {} as Record<string, number>
);

console.log(counts);
// { Action: 2, Comedy: 3, Drama: 1, Horror: 1 }
```

6.4 Java Comparison

In Java, `reduce` exists on Streams but is rarely used for anything beyond summing:

```
double total = prices.stream()
    .reduce(0.0, (sum, price) -> sum + price);
```

For grouping, Java developers typically use `Collectors.groupingBy`:

```
Map<String, Long> counts = genres.stream()
    .collect(Collectors.groupingBy(g -> g, Collectors.counting()));
```

TypeScript's `reduce` handles both cases with the same method. Whether that is simpler or harder to read depends on the situation.

> **Use reduce Sparingly**
>
> `reduce` is powerful but often hard to read, especially for developers who did not write the original code. Before reaching for `reduce`, ask yourself:
>
> - Could `map` + `filter` accomplish the same thing more clearly?
> - Would a simple `for...of` loop be easier to understand?
>
> In this course, you will mostly use `reduce` for summing numbers and building lookup objects. If your `reduce` callback is more than a few lines long, a `for...of` loop is probably clearer.

Try It Yourself

1. Use `reduce` to find the total `vote_count` across an array of movies
2. Use `reduce` to build a `Record<string, Movie[]>` that groups movies by language
3. Rewrite your grouping logic as a `for...of` loop — compare readability

---

7 some and every

These two methods answer yes-or-no questions about an array. They return a `boolean`.

7.1 some — At Least One?

`some` returns `true` if **at least one** element passes the test:

```
interface User {
  name: string;
  role: "user" | "admin";
  active: boolean;
}

const team: User[] = [
  { name: "Alice", role: "admin", active: true },
  { name: "Bob", role: "user", active: true },
  { name: "Carol", role: "user", active: false },
];

const hasAdmin: boolean = team.some((u) => u.role === "admin");
console.log(hasAdmin); // true

const hasInactive: boolean = team.some((u) => !u.active);
console.log(hasInactive); // true
```

`some` short-circuits — it stops as soon as it finds a match. This is efficient for large arrays.

7.2 every — All of Them?

`every` returns `true` only if **all** elements pass the test:

```
const allActive: boolean = team.every((u) => u.active);
console.log(allActive); // false — Carol is inactive
```

```ts
const allNamed: boolean = team.every((u) => u.name.length > 0);
console.log(allNamed); // true
```

`every` also short-circuits — it stops as soon as it finds an element that fails.

7.3 Validation in Express

`some` and `every` are useful for validation logic in route handlers:

```ts
app.post("/api/team/activate", (req, res) => {
  const { userIds } = req.body as { userIds: number[] };

  // Verify all requested IDs exist
  const allExist = userIds.every(
    (id) => users.find((u) => u.id === id) !== undefined
  );

  if (!allExist) {
    res.status(400).json({ error: "One or more user IDs not found" });
    return;
  }

  // Proceed with activation...
  res.json({ message: `Activated ${userIds.length} users` });
});
```

7.4 Java Comparison

Java Streams have `anyMatch` and `allMatch`, which are directly equivalent:

```java
boolean hasAdmin = team.stream().anyMatch(u -> u.getRole().equals("admin"));
boolean allActive = team.stream().allMatch(u -> u.isActive());
```

In TypeScript:

```ts
const hasAdmin: boolean = team.some((u) => u.role === "admin");
const allActive: boolean = team.every((u) => u.active);
```

The logic is identical. TypeScript uses `some` / `every`; Java uses `anyMatch` / `allMatch`.

Try It Yourself

1. Create an array of movie objects with a `rating` field
2. Use `some` to check if any movie has a perfect 10.0 rating
3. Use `every` to check if all movies are rated above 5.0
4. Try both methods on an empty array — what do `[].some(...)` and `[].every(...)` return?

---

8 Chaining Methods

The real power of array methods emerges when you chain them together. Because `filter`, `map`, and `sort` each return a new array, you can call one method immediately after another.

8.1 A Realistic Chain

Suppose your Express API needs to return a list of top-rated English-language movies, formatted for a client:

```
interface MovieResponse {
  title: string;
  rating: number;
  year: number;
}

const topEnglishMovies: MovieResponse[] = movies
  .filter((movie) => movie.language === "en")
  .filter((movie) => movie.rating >= 8.0)
  .map((movie) => ({
    title: movie.title,
    rating: movie.rating,
    year: movie.releaseYear,
  }))
  .sort((a, b) => b.rating - a.rating);
```

Read this chain top to bottom:

1. **filter** — keep only English-language movies
2. **filter** — keep only those rated 8.0 or higher
3. **map** — reshape each movie into the client-facing format
4. **sort** — order by rating, highest first

Each step receives the output of the previous step. The original `movies` array is never modified.

8.2 Java Comparison — Stream Pipelines

If you used Java Streams in TCSS 305, this pattern will look familiar:

```
List<MovieResponse> topEnglishMovies = movies.stream()
    .filter(m -> m.getLanguage().equals("en"))
    .filter(m -> m.getRating() >= 8.0)
    .map(m -> new MovieResponse(m.getTitle(), m.getRating(),
m.getReleaseYear()))
    .sorted(Comparator.comparingDouble(MovieResponse::getRating).reversed())
    .collect(Collectors.toList());
```

The TypeScript version is structurally identical but without the ceremony of `.stream()`, `Collectors`, and `Comparator` utilities. Both are declarative pipelines that describe *what* you want rather than *how* to loop.

8.3 Keeping Chains Readable

Long chains can become hard to read. Follow these guidelines:

One method per line, aligned by the dot:

```
// GOOD — each step is clearly visible
const result = items
  .filter((item) => item.active)
  .map((item) => item.name)
  .sort();
```

```
// HARD TO READ — everything on one line
const result = items.filter((item) => item.active).map((item) =>
item.name).sort();
```

Break out complex callbacks into named functions:

```
// GOOD — named functions make the intent clear
const isHighlyRated = (movie: Movie): boolean => movie.rating >= 8.0;
const toSummary = (movie: Movie): MovieSummary => ({
  title: movie.title,
  rating: movie.rating,
});

const topMovies = movies
  .filter(isHighlyRated)
  .map(toSummary);
```

Limit chain length. If a chain has more than four or five steps, consider breaking it into named intermediate variables:

```
// Instead of one giant chain, break into logical steps
const englishMovies = movies.filter((m) => m.language === "en");
const highlyRated = englishMovies.filter((m) => m.rating >= 8.0);
const formatted = highlyRated.map(toSummary);
const sorted = formatted.sort((a, b) => b.rating - a.rating);
```

Gen AI & Learning: Array Method Chains

AI coding assistants are very good at writing array method chains. If you describe what you want in natural language — "filter for English movies rated above 8, extract title and rating, sort by rating descending" — an AI assistant will likely produce correct code. The key skill is being able to *read and verify* what the AI generates. Understanding each method in this guide is what lets you confidently review and modify AI-generated chains.

Try It Yourself

1. Start with the `movies` array from Section 4
2. Chain `filter` and `map` to get titles of movies released after 2005
3. Add a `sort` to put them in alphabetical order
4. Try writing the same logic as a single `for...of` loop — notice how much more code it takes

---

9 Sorting

Sorting is a common operation, but it has a significant gotcha in TypeScript that trips up many developers.

9.1 Basic Sorting

The `sort` method sorts an array in place. With no arguments, it converts elements to strings and sorts them alphabetically:

```
const fruits: string[] = ["banana", "apple", "cherry"];
fruits.sort();
console.log(fruits); // ["apple", "banana", "cherry"]
```

For numbers, the default behavior is **wrong**:

```
const nums: number[] = [10, 1, 21, 2];
nums.sort();
console.log(nums); // [1, 10, 2, 21] — sorted as STRINGS, not numbers!
```

You must provide a comparison function for numeric sorting:

```
const nums: number[] = [10, 1, 21, 2];
nums.sort((a, b) => a - b);
console.log(nums); // [1, 2, 10, 21] — correct ascending order
```

The comparison function should return:

- A **negative number** if `a` should come before `b`
- A **positive number** if `a` should come after `b`
- **Zero** if they are equal

9.2 Sorting Objects

When sorting an array of objects, you provide a comparison function that accesses the relevant property:

```
// Sort movies by rating, highest first (descending)
movies.sort((a, b) => b.rating - a.rating);

// Sort movies by title, alphabetically (ascending)
movies.sort((a, b) => a.title.localeCompare(b.title));
```

`localeCompare` handles string comparison correctly across different languages and character sets. Always use it instead of `<` or `>` for strings.

9.3 The Mutation Problem

⚠ sort() Mutates the Original Array

Unlike `map` and `filter`, `sort` modifies the array in place. This can cause subtle bugs:

```
const original = [3, 1, 2];
const sorted = original.sort((a, b) => a - b);

console.log(sorted);   // [1, 2, 3]
console.log(original); // [1, 2, 3] — ALSO sorted! Same
reference!
console.log(sorted === original); // true — they are the same
array
```

If you need a sorted copy without modifying the original, you have two options:

```
// Option 1: Spread into a new array first, then sort
const sorted = [...original].sort((a, b) => a - b);

// Option 2: Use toSorted() (ES2023+, supported in Node 20+)
const sorted = original.toSorted((a, b) => a - b);
```

9.4 toSorted — The Immutable Alternative

`toSorted` works exactly like `sort` but returns a new array, leaving the original unchanged:

```
const original: number[] = [3, 1, 2];
const sorted: number[] = original.toSorted((a, b) => a - b);

console.log(sorted);   // [1, 2, 3]
console.log(original); // [3, 1, 2] — unchanged
```

`toSorted` is available in Node.js 20 and later. Since this course uses a modern Node.js version, you can use it freely. When chaining with other array methods (`filter`, `map`), mutation is less of a concern because those methods already create new arrays — but `toSorted` makes the intent explicit.

9.5 Java Comparison

Java's `Collections.sort()` also mutates in place, so the behavior is similar:

```
// Java — mutates the list
Collections.sort(movies,
Comparator.comparingDouble(Movie::getRating).reversed());
```

```java
// Java — immutable sort via Stream
List<Movie> sorted = movies.stream()
    .sorted(Comparator.comparingDouble(Movie::getRating).reversed())
    .collect(Collectors.toList());
```

In TypeScript:

```typescript
// Mutates — like Collections.sort()
movies.sort((a, b) => b.rating - a.rating);

// Immutable — like Stream.sorted()
const sorted = movies.toSorted((a, b) => b.rating - a.rating);
```

Try It Yourself

1. Create an array of numbers and sort them in descending order
2. Sort the `movies` array by `releaseYear` (oldest first)
3. Use `toSorted` to create a sorted copy — verify the original is unchanged
4. Try sorting strings with `.sort()` (no comparator) and confirm it works correctly for strings

---

10 Summary

Method	Purpose	Returns	Mutates?
`forEach`	Execute a function for each element (side effects)	`undefined`	No
`map`	Transform every element	New array (same length)	No
`filter`	Keep elements that pass a test	New array (same or shorter)	No
`find`	Get the first matching element	Element or `undefined`	No
`findIndex`	Get the index of the first match	Index or `-1`	No

Method	Purpose	Returns	Mutates?
`reduce`	Accumulate elements into a single value	Any type	No
`some`	Check if at least one element passes	`boolean`	No
`every`	Check if all elements pass	`boolean`	No
`sort`	Sort elements in place	Same array (sorted)	**Yes**
`toSorted`	Sort elements into a new array	New array (sorted)	No

Key takeaways:

- **`map` and `filter` are your daily tools.** Most data transformation in Express route handlers uses these two methods, often chained together.
- **`find` is for single lookups.** Use it in "get by ID" endpoints. Always handle the `undefined` case.
- **`reduce` is powerful but use it sparingly.** Summing numbers and building lookup objects are good uses. For complex logic, a `for...of` loop is often clearer.
- **`sort` mutates.** Use `toSorted` or spread (`[...arr].sort(...)`) when you need the original unchanged.
- **Chain methods for readable pipelines.** One method per line, named callbacks for complex logic.

---

11 References

Official Documentation:

- TypeScript Handbook — TypeScript fundamentals and type system
- MDN Web Docs — Array — Complete reference for all array methods
- MDN Web Docs — Array.prototype.map() — Detailed documentation for `map`
- MDN Web Docs — Array.prototype.filter() — Detailed documentation for `filter`
- MDN Web Docs — Array.prototype.reduce() — Detailed documentation for `reduce`

- MDN Web Docs — Array.prototype.toSorted() — Immutable sorting (ES2023)

---

12 Further Reading

External Resources

- Eloquent JavaScript — Higher-Order Functions - Deep dive into functions that operate on other functions, including array methods
- JavaScript.info — Array Methods - Comprehensive tutorial covering all array methods with interactive examples
- Node.js ES2023 Support - Compatibility table showing `toSorted` and other modern array methods in Node.js versions

---

This guide is part of TCSS 460 — Client/Server Programming, School of Engineering and Technology, University of Washington Tacoma.