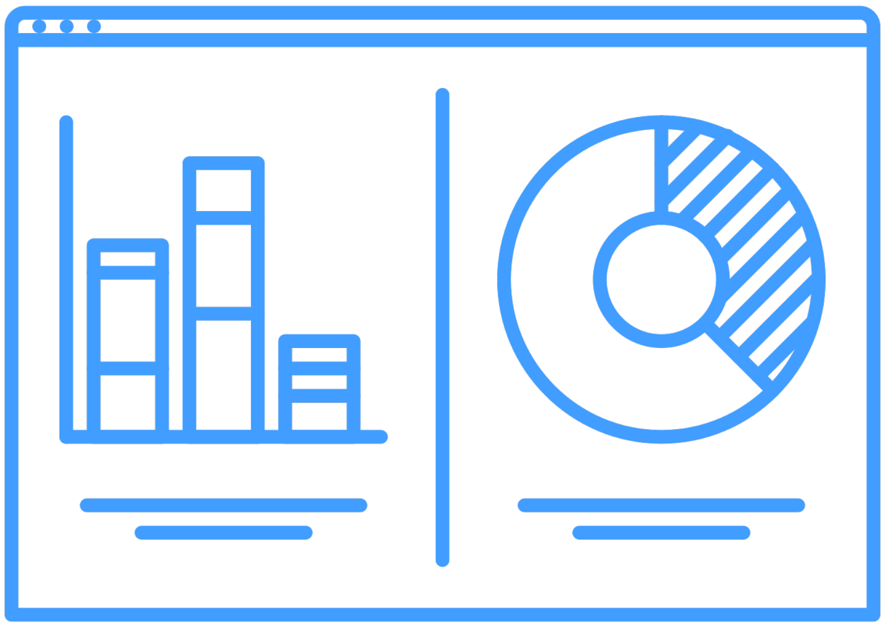
D3.js Data Visualization



readbytes

D3.js Data Visualization

Building interactive, data-driven visuals

readbytes.github.io

2025-07-20

This page is intentionally left blank.

Contents

Chapter 1.
Introduction

1. What Is D3.js and Why Use It?
2. History and Evolution of D3
3. Setting Up Your Development Environment

1 Introduction

1.1 What Is D3.js and Why Use It?

D3.js, short for **Data-Driven Documents**, is a powerful JavaScript library used to create dynamic, interactive data visualizations in web browsers using web standards like **SVG**, **HTML**, and **CSS**. Unlike conventional charting libraries that offer a fixed set of charts (like bar charts, pie charts, and line graphs), D3.js gives you **complete control over how data is visualized and how it interacts with the DOM (Document Object Model)**.

1.1.1 D3.js vs. Charting Libraries

To understand what makes D3.js different, consider the following comparison:

Feature	D3.js	Chart.js / Highcharts
Chart Types Available	Fully customizable, build any chart	Predefined set (e.g., line, bar, pie)
DOM Manipulation	Full access to DOM and styles	Encapsulated rendering (no direct DOM access)
Interactivity	Highly interactive and customizable	Limited to built-in options
Learning Curve	Steep	Moderate to easy
Output Format	HTML/SVG/Canvas	Canvas/SVG (abstracted)

Where Chart.js and Highcharts provide **ready-made components** to build standard charts quickly, D3 is a **low-level toolkit** that gives you the **freedom to build your own visual storytelling from the ground up**.

1.1.2 How D3 Works: Data DOM

At its core, D3 lets you **bind data to DOM elements** and then apply transformations to those elements based on the data. This data-binding mechanism allows you to represent any dataset as shapes, lines, text, or even complex UI components.

Here's a simplified example:

```
d3.selectAll("div")
  .data([10, 20, 30])
  .style("width", d => d + "px")
  .text(d => d);
```

This code selects all `<div>` elements, binds an array of numbers to them, and sets each div's width and text content based on the data. The result? A set of bars directly derived from data — no prebuilt chart object needed.

1.1.3 Why Use D3?

D3 is ideal when:

- You want **full design control** over every visual element.
- Your visualization needs to be **interactive or animated**.
- You're building **custom dashboards** or **data storytelling experiences**.
- You want to work **closely with raw data structures**, like JSON or CSV.
- You need to **integrate visualizations deeply** with your web application's logic and UI.

D3 isn't a tool to just "make charts" — it's a toolkit to **construct new ways to see and explore data**.

1.1.4 When Not to Use D3

Because of its flexibility, D3 has a **steeper learning curve** than plug-and-play chart libraries. If you simply need a quick bar chart or a line graph with minimal configuration, a library like Chart.js or Highcharts might serve you better. However, when your project demands **bespoke visuals, real-time interactivity, or deep integration**, D3 is unmatched.

1.2 History and Evolution of D3

To understand D3.js today, it helps to explore where it came from and why it was created the way it was. D3's design is deeply rooted in academic research and a desire to break away from the limitations of traditional visualization libraries.

1.2.1 Origins: From SIMILE to Protovis

D3.js traces its lineage back to the **SIMILE Project** at the Massachusetts Institute of Technology (MIT), an initiative focused on data exploration and visualization tools for the semantic web. One of the key outputs of SIMILE was **Exhibit**, a lightweight framework for publishing interactive visualizations using only HTML and JavaScript.

From this foundation, researchers **Mike Bostock**, **Jeffrey Heer**, and **Vadim Ogievetsky**—then at the **Stanford Visualization Group**—developed **Protovis** around 2009. Protovis aimed to simplify the creation of rich visualizations using a declarative API. It allowed developers to describe visualizations in terms of "marks" (e.g., bars, dots, lines), which were automatically mapped to data.

Protovis gained early praise for lowering the barrier to visualization, but it had architectural limitations. It was tightly coupled and difficult to extend. These shortcomings led to its eventual replacement.

1.2.2 The Birth of D3.js

In 2011, Mike Bostock and his collaborators introduced **D3.js (Data-Driven Documents)** as the spiritual successor to Protovis. D3 adopted a **modular, low-level approach**, focusing on **binding data to DOM elements** and using standard web technologies—**SVG**, **HTML**, and **CSS**—to render graphics.

The key innovations in D3's design included:

- **Modularity**: Unlike monolithic libraries, D3 was broken into small, reusable pieces.
- **Composability**: Developers could build complex visuals by combining simple building blocks.
- **Declarative Thinking**: Instead of telling the computer *how* to draw something, developers described *what* the final state should look like, and D3 handled the transitions.

This design philosophy empowered users to craft highly customized, data-driven visualizations—something no existing library allowed with such precision.

1.2.3 Version Milestones and Community Growth

Over time, D3 evolved through several major releases. Each brought important enhancements, bug fixes, and refined APIs. Here are some key milestones:

- **v1.x (2011–2014)**: The foundational version that introduced data binding, transitions, scales, and layout tools. D3 gained popularity among developers and academics alike.
- **v3.x (2013–2016)**: Added better support for layouts (e.g., force-directed graphs, treemaps), improved scales, and support for GeoJSON. This version fueled widespread adoption and became the basis for many tutorials and visualizations.
- **v4.x (2016)**: A major rewrite that **modularized the library** into separate packages (e.g., `d3-scale`, `d3-shape`, etc.). This made it easier to import only what you needed, improving performance and tree-shaking in modern bundlers.

- **v5.x (2018)**: Introduced **Promises** for asynchronous data loading, reflecting the shift toward modern JavaScript practices. This version also improved interoperability with modern APIs like `fetch()`.
- **v6.x and v7.x (2020–2021)**: Continued refinements, better support for modern web standards, and improvements to scales, axes, and event handling. The core API remained stable, underscoring D3's maturity.

1.2.4 D3 in the Wild

Over the years, D3 has become the **backbone of countless visualizations** seen in journalism, academic papers, dashboards, and data art. It powers projects by **The New York Times**, **The Guardian**, **NPR**, **NASA**, and many others. Entire ecosystems and wrapper libraries (like Nivo, Vega, and Observable Plot) have emerged around it.

While newer visualization tools offer higher-level abstractions or more convenient APIs, D3 remains the **go-to library for developers who need fine-grained control and expressive power**.

1.2.5 Motivations Behind D3's Design

D3 wasn't built to be easy—it was built to be expressive. Its core principles reflect that:

- **Modularity**: You only use the parts you need. No bloat, no monolith.
- **Composability**: Simple primitives like `selection`, `enter`, `exit`, and `transition` can be combined to build complex interactions.
- **Declarative Mindset**: Instead of imperative "draw" calls, you describe the desired end state, and D3 computes the transitions.

These ideas have influenced not only the world of data visualization, but also broader web development trends—such as the rise of reactive programming and declarative UI frameworks like React.

1.3 Setting Up Your Development Environment

Before you can build stunning visualizations with D3.js, you'll need a solid development environment. This section walks you through the setup using **Visual Studio Code**, a lightweight code editor with powerful features, along with a local development server and `npm` for package management. We'll cover two ways to include D3 in your project: **via a CDN** and **as an npm module**.

By the end of this section, you'll have a working HTML page with D3 loaded and a simple SVG circle drawn to verify everything is in place.

1.3.1 Tools Youll Need

- **Visual Studio Code (VS Code)** – Free editor for writing and debugging code. Download: https://code.visualstudio.com
- **Node.js and npm** – Needed for installing packages and running local tools. Download: https://nodejs.org
- **Live Server Extension** – Lets you preview your web page with automatic refresh.

1.3.2 Step 1: Install VS Code and Extensions

1. **Install VS Code** from the official website.
2. Open VS Code and go to the **Extensions** view (click the square icon in the sidebar or press `Ctrl+Shift+X`).
3. Search for and install the extension called **Live Server** by *Ritwick Dey*.

1.3.3 Step 2: Set Up a Project Folder

1. Create a folder for your project, e.g., `d3-intro`.
2. Open this folder in VS Code:

   ```
   File → Open Folder → Select your project folder
   ```

3. Inside the folder, create two files:
   - `index.html`
   - `script.js`

1.3.4 Step 3: Include D3 via CDN (Quick Start)

For the simplest setup, you can include D3.js using a Content Delivery Network (CDN). This is great for learning and small projects.

`index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Setup Example</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
</head>
<body>
  <svg width="400" height="200"></svg>
  <script src="script.js"></script>
</body>
</html>
```

`script.js`:

```
const svg = d3.select("svg");

svg.append("circle")
  .attr("cx", 100)
  .attr("cy", 100)
  .attr("r", 40)
  .attr("fill", "steelblue");
```

1.3.5 Step 4: Preview with Live Server

1. Right-click `index.html` in the VS Code explorer.
2. Select **"Open with Live Server."**
3. Your default browser will open and you'll see a blue circle on the page.

1.3.6 Step 5: Install with npm (Modular Setup)

For larger projects or build toolchains, you may prefer to manage D3 as a package.

1. Open the terminal in VS Code ('Ctrl+" or View → Terminal).
2. Run the following commands:

```
npm init -y
npm install d3
```

This installs D3 locally in your project under the `node_modules` folder.

1.3.7 Step 6: Use D3 with npm Modern JS (Optional)

To use npm packages with ES Modules and modern bundlers, you'll need a build tool like **Parcel**, **Vite**, or **Webpack**. Here's a quick Parcel-based example:

1. Install Parcel:
```
npm install --save-dev parcel
```

2. Replace your files like so:

`index.html`:
```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 with Parcel</title>
</head>
<body>
  <svg width="400" height="200"></svg>
  <script type="module" src="./script.js"></script>
</body>
</html>
```

`script.js`:
```js
import * as d3 from "d3";

const svg = d3.select("svg");

svg.append("circle")
  .attr("cx", 100)
  .attr("cy", 100)
  .attr("r", 40)
  .attr("fill", "orange");
```

3. Add a `start` script to `package.json`:
```json
"scripts": {
  "start": "parcel index.html"
}
```

4. Run:
```
npm run start
```

Your project will open in the browser with D3 fully functional using npm.

1.3.8 Summary

You've now seen two main ways to set up D3:

- **CDN-based setup**: Quick, easy, and perfect for learning or demos.
- **npm-based setup**: More scalable, modular, and great for real projects.

In both cases, we verified the setup by drawing a simple SVG circle to the page. With your environment ready, you're all set to start building powerful, interactive data visualizations.

Chapter 2.
Understanding the D3 Philosophy

1. Declarative vs Imperative Approaches
2. Selections and Data Binding
3. Enter, Update, Exit Pattern
4. Chaining and Functional Style in D3

2 Understanding the D3 Philosophy

2.1 Declarative vs Imperative Approaches

At the heart of D3.js is a powerful idea: describe *what* you want, and let the library figure out *how* to make it happen. This mindset—called the **declarative programming model**—is a cornerstone of D3's design. To appreciate D3's strengths, it helps to first understand how it contrasts with the more common **imperative style** used in traditional DOM manipulation.

2.1.1 Imperative Programming: Telling the Computer *How*

In an **imperative approach**, you write code that explicitly tells the computer what steps to perform, in what order. You manage low-level details, like creating elements, setting attributes, and inserting them into the DOM.

Example: Creating a red `<div>` using vanilla JavaScript or jQuery-style code

```
const div = document.createElement('div');
div.style.width = '100px';
div.style.height = '100px';
div.style.backgroundColor = 'red';
document.body.appendChild(div);
```

Or with jQuery:

```
$('<div>')
  .css({ width: '100px', height: '100px', backgroundColor: 'red' })
  .appendTo('body');
```

In both examples, you're giving explicit, step-by-step instructions to create and configure the element. This style is clear and direct, but **can quickly become verbose and error-prone**, especially when managing complex visualizations with dynamic data.

2.1.2 Declarative Programming: Describing *What* You Want

In contrast, **declarative programming** focuses on expressing *intent*. Instead of specifying a procedure, you define the desired outcome, and the underlying system figures out how to achieve it.

Example: Creating a red `<div>` with D3

```
d3.select('body')
  .append('div')
  .style('width', '100px')
  .style('height', '100px')
  .style('background-color', 'red');
```

This looks similar on the surface, but conceptually, you're saying:

> "Select the body, append a div to it, and declare what it should look like."

There's no need to manually manage the creation of a `div` object or insert it into the DOM; D3 handles that based on your **declarations**.

2.1.3 A Real-World Comparison: Creating Bars from Data

Let's take a more meaningful example: building a simple bar chart from an array of numbers.

Imperative Style (Vanilla JavaScript)

```javascript
const data = [100, 200, 150];
const container = document.getElementById('chart');

for (let i = 0; i < data.length; i++) {
  const bar = document.createElement('div');
  bar.style.width = data[i] + 'px';
  bar.style.height = '20px';
  bar.style.backgroundColor = 'steelblue';
  bar.style.marginBottom = '5px';
  container.appendChild(bar);
}
```

You manually create each `div`, style it, and append it—controlling every step. The logic is tightly coupled to the data loop.

Declarative Style (D3.js)

```javascript
const data = [100, 200, 150];

d3.select('#chart')
  .selectAll('div')
  .data(data)
  .enter()
  .append('div')
  .style('width', d => d + 'px')
  .style('height', '20px')
  .style('background-color', 'steelblue')
  .style('margin-bottom', '5px');
```

Here, you declare:

> "For each data point, create a `div` and style it according to the data."

D3 abstracts away the loop and DOM bookkeeping. This is the **data-driven paradigm**: data determines structure and appearance. The logic is concise, expressive, and scales gracefully with complexity.

2.1.4 Why D3 Favors Declarative Design

D3's declarative approach has several advantages:

- **Clarity**: Your code mirrors your intent—e.g., "bind this data to these elements and set their visual properties."
- **Efficiency**: D3 can optimize DOM updates under the hood.
- **Flexibility**: You can express interactions, transitions, and layout updates in a consistent, composable way.
- **Scalability**: Managing thousands of elements becomes manageable through abstraction.

This philosophy also aligns with how modern frontend frameworks (like React or Svelte) think about UI: describe what the UI *should* look like, and let the system update the DOM accordingly.

2.1.5 Summary

Imperative	Declarative (D3)
Tells the browser how to do everything step by step	Describes what the final output should be
Manually creates and inserts elements	Binds data to elements and lets D3 create them
Good for simple, static interactions	Better for dynamic, data-driven UIs

D3's declarative style may feel unusual at first, especially if you come from jQuery or imperative JavaScript. But once you embrace it, you'll find it **elegant, expressive, and highly scalable**—perfect for the complex, interactive visualizations D3 is built for.

2.2 Selections and Data Binding

One of the most powerful and distinctive features of D3.js is its **data binding mechanism**—the ability to connect data directly to DOM elements. This data-driven approach forms the foundation of all D3 visualizations.

At the core of this system are **selections**, which allow you to query the DOM and apply transformations based on the data you're working with. Let's explore how this works in practice.

2.2.1 What Is a Selection?

In D3, a **selection** is a reference to one or more elements in the DOM. You use it to **apply changes**, **bind data**, or **append new elements**.

D3 provides two main functions to create selections:

- `d3.select()` – Selects **the first** matching element.
- `d3.selectAll()` – Selects **all** matching elements.

Examples:

```
d3.select("p")         // selects the first <p> element
d3.selectAll("p")      // selects all <p> elements
```

Selections behave like arrays of DOM elements, but they're **enhanced with D3 methods** that let you style, transform, and bind data to those elements.

2.2.2 Binding Data with `.data()`

D3's true power comes when you pair selections with **data** using the `.data()` method. This allows you to map an array of values to a selection of DOM elements, enabling you to create or update elements dynamically based on data.

The basic pattern looks like this:

```
selection.data(data)
```

But that's only the beginning. Let's look at a concrete example.

2.2.3 Example: Binding Data to SVG Circles

Suppose we have a simple dataset of radii, and we want to create a circle for each value. Here's how we can bind the data to SVG elements using D3.

HTML:

```
<svg width="400" height="100"></svg>
```

JavaScript:

```
const data = [10, 20, 30];

const svg = d3.select("svg");

svg.selectAll("circle")
  .data(data)
  .enter()
  .append("circle")
  .attr("cx", (d, i) => 50 + i * 100)  // horizontal position
  .attr("cy", 50)                      // vertical position
  .attr("r", d => d)                   // radius from data
  .attr("fill", "steelblue");
```

Full runnable code:

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Circle Binding Example</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 20px;
    }
    svg {
      border: 1px solid #ccc;
    }
  </style>
</head>
<body>
  <h2>D3: Binding Data to SVG Circles</h2>
  <svg width="400" height="100"></svg>

  <script>
    const data = [10, 20, 30];

    const svg = d3.select("svg");

    svg.selectAll("circle")
      .data(data)
      .enter()
      .append("circle")
      .attr("cx", (d, i) => 50 + i * 100)  // horizontal position
      .attr("cy", 50)                      // vertical position
      .attr("r", d => d)                   // radius from data
      .attr("fill", "steelblue");
  </script>
</body>
</html>
```

2.2.4 Whats Happening Here?

Let's break down each step:

1. `svg.selectAll("circle")`
    - This creates a **selection of all existing `<circle>` elements** (initially none).
2. `.data(data)`

- This **binds the `data` array** to the selection. D3 keeps track of which data item corresponds to which element.

3. `.enter()`
    - Since there are no existing circles yet, all data items are in the **"enter" selection** (more on this in the next section).
4. `.append("circle")`
    - A new `<circle>` is created for each item in the `data` array.
5. `.attr(...)`
    - Each circle is positioned and sized according to the corresponding data value.

The result is three circles, each with a radius based on the values `[10, 20, 30]`.

2.2.5 Summary

Concept	Description
`select()` / `selectAll()`	Query DOM elements (similar to `document.querySelector`)
`.data()`	Bind an array of data to a selection
`.enter()`	Handle new data items with no corresponding DOM elements
`.append()`	Create new elements based on bound data

D3's selection and data-binding model allows you to write code that **responds to data**, not just statically creates elements. This is what makes D3 uniquely powerful for visualizing dynamic or complex datasets.

2.3 Enter, Update, Exit Pattern

At the heart of D3's data-driven approach lies a critical pattern known as the **Enter–Update–Exit** pattern. This is how D3 handles changes in data over time, allowing you to efficiently manage the creation, update, and removal of elements in the DOM.

This pattern is what makes D3 particularly powerful for **dynamic and real-time visualizations**, where data is not static—new points arrive, old ones disappear, and existing ones change.

2.3.1 The Data Join in Three Phases

Whenever you bind data to DOM elements with `.data(data)`, D3 evaluates the relationship between **the data items** and **existing DOM elements**. The result is a **data join**, which breaks into three sub-selections:

Phase	What it means
`enter()`	Items in data **without** a corresponding DOM element → **add new elements**
`update()`	Items in data **with** a matching DOM element → **update attributes**
`exit()`	DOM elements **with no corresponding data** → **remove elements**

2.3.2 Visual Overview

Suppose the DOM has 3 `<circle>` elements and the new dataset has 5 values:

```
DOM Elements:
New Data Items:
```

- The first 3 items are matched → **update**
- The remaining 2 have no elements → **enter**
- No unmatched DOM elements → **exit** not used

Now suppose we later shrink the dataset to just 2 items:

```
DOM Elements:
New Data Items:
```

- First 2 → **update**
- Extra 3 DOM elements → **exit**

2.3.3 Example: Circles That Grow and Shrink

Let's walk through a complete example. We'll start with some circles, then update the data to show how new circles are added and old ones are removed.

HTML:

```html
<svg width="500" height="100"></svg>
```

JavaScript (Initial Render):

```javascript
const svg = d3.select("svg");

let data = [30, 70, 50];

function render(data) {
  const circles = svg.selectAll("circle")
    .data(data);

  // ENTER: Add new circles
  circles.enter()
    .append("circle")
    .attr("cx", (d, i) => i * 100 + 50)
    .attr("cy", 50)
    .attr("r", 0)
    .attr("fill", "steelblue")
    .transition()
    .attr("r", d => d);

  // UPDATE: Modify existing circles
  circles
    .transition()
    .duration(500)
    .attr("r", d => d)
    .attr("cx", (d, i) => i * 100 + 50);

  // EXIT: Remove circles with no data
  circles.exit()
    .transition()
    .attr("r", 0)
    .remove();
}

// Initial render
render(data);
```

JavaScript (Update with New Data):

```
// Update after 2 seconds
setTimeout(() => {
  data = [60, 40, 80, 20];  // One new item added
  render(data);
}, 2000);

// Another update after 4 seconds
setTimeout(() => {
  data = [100, 50];  // Two removed
  render(data);
}, 4000);
```

2.3.4 Step-by-Step Breakdown

`enter()`: Creating New Elements

```
circles.enter()
  .append("circle")
  .attr("r", 0)
  .transition()
  .attr("r", d => d);
```

This handles data items with no corresponding element—new `<circle>` elements are appended and animated into view.

`update()`: Modifying Existing Elements

```
circles
  .transition()
  .attr("r", d => d)
  .attr("cx", (d, i) => i * 100 + 50);
```

This modifies properties of circles that already exist—for example, moving them or changing size based on new data.

`exit()`: Removing Unused Elements

```
circles.exit()
  .transition()
  .attr("r", 0)
  .remove();
```

This removes DOM elements whose data is no longer present, optionally with a graceful transition.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Growing and Shrinking Circles</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 20px;
    }
    svg {
      border: 1px solid #ccc;
      margin-top: 10px;
    }
  </style>
</head>
<body>
  <h2>D3: Circles That Grow and Shrink</h2>
  <svg width="500" height="100"></svg>

  <script>
    const svg = d3.select("svg");

    let data = [30, 70, 50];

    function render(data) {
      const circles = svg.selectAll("circle")
        .data(data, (d, i) => i); // Use index as key to track updates

      // ENTER: Add new circles
      circles.enter()
        .append("circle")
        .attr("cx", (d, i) => i * 100 + 50)
        .attr("cy", 50)
        .attr("r", 0)
        .attr("fill", "steelblue")
        .transition()
        .duration(500)
        .attr("r", d => d);

      // UPDATE: Update existing circles
      circles.transition()
        .duration(500)
        .attr("r", d => d)
        .attr("cx", (d, i) => i * 100 + 50);

      // EXIT: Remove old circles
      circles.exit()
        .transition()
        .duration(500)
        .attr("r", 0)
        .remove();
    }

    // Initial render
    render(data);

    // First update after 2 seconds
    setTimeout(() => {
      data = [60, 40, 80, 20]; // One new item added
      render(data);
    }, 2000);

    // Second update after 4 seconds
    setTimeout(() => {
      data = [100, 50]; // Two items removed
      render(data);
    }, 4000);
  </script>
</body>
</html>
```

2.3.5 Why This Pattern Matters

In static visualizations, you create elements once and forget about them. But in real-world applications—dashboards, live feeds, interactive controls—**data changes constantly**. You need a way to reflect those changes efficiently and cleanly in the DOM.

The enter–update–exit pattern is D3's answer. It provides:

- **Efficiency**: Only the necessary elements are created or removed.
- **Clarity**: Your code structure mirrors how data flows in and out.
- **Animation hooks**: Transitions can be applied separately to each phase.

Without this pattern, you'd have to manually keep track of every DOM element, greatly increasing complexity and risk of bugs.

2.3.6 Summary

Phase	Triggered When…	Common Use
`enter()`	New data items have no DOM elements	Add new visuals
`update()`	Data items match existing elements	Modify properties
`exit()`	DOM elements have no data	Remove visuals

Mastering the Enter–Update–Exit pattern is essential for building responsive, real-time visualizations in D3. In the next section, we'll explore how D3's **chained methods and functional style** let you write expressive, concise code that flows naturally from data to visuals.

2.4 Chaining and Functional Style in D3

D3.js is designed with an expressive, fluid API that encourages **method chaining** and the use of **functions as arguments**. This design allows you to build concise, readable, and highly composable visualization code—turning data into visuals through a smooth pipeline of transformations.

2.4.1 What Is Method Chaining?

Method chaining means calling multiple methods one after another on the same object, where each method returns the object itself (or a related selection), enabling you to write code like this:

```
d3.select("circle")
  .attr("cx", 50)
  .attr("cy", 50)
  .attr("r", 40)
  .style("fill", "orange");
```

Instead of breaking these commands into multiple statements, chaining lets you **express a sequence of changes compactly**. This style improves readability by emphasizing the logical flow of operations.

2.4.2 Functions as Arguments

Many D3 methods accept functions as arguments, allowing you to compute values dynamically based on data or element index.

For example:

```
d3.selectAll("circle")
  .attr("r", d => d.radius)
  .style("fill", (d, i) => i % 2 === 0 ? "steelblue" : "lightgray");
```

Here, the radius and fill color depend on the bound data `d` and the element's index `i`, making the visualization truly **data-driven** and flexible.

2.4.3 Putting It Together: A Simple Interactive Circle

```
d3.select("svg")
  .append("circle")
  .attr("cx", 100)
  .attr("cy", 75)
  .attr("r", 50)
  .style("fill", "teal")
  .on("mouseover", function() {
    d3.select(this).style("fill", "orange");
  })
  .on("mouseout", function() {
    d3.select(this).style("fill", "teal");
  });
```

This snippet shows chaining of:

- `.attr()` — setting SVG attributes
- `.style()` — applying CSS styles
- `.on()` — adding event listeners

The code reads like a **pipeline**: create a circle, style it, then attach interaction handlers—all in a smooth, connected sequence.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Interactive D3 Circle</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 20px;
    }
    svg {
      border: 1px solid #ccc;
    }
  </style>
</head>
<body>
  <h2>D3: Simple Interactive Circle</h2>
  <svg width="300" height="150"></svg>

  <script>
    d3.select("svg")
      .append("circle")
      .attr("cx", 100)
      .attr("cy", 75)
      .attr("r", 50)
      .style("fill", "teal")
      .on("mouseover", function() {
        d3.select(this).style("fill", "orange");
      })
      .on("mouseout", function() {
        d3.select(this).style("fill", "teal");
      });
  </script>
</body>
</html>
```

2.4.4 Benefits of Chaining and Functional Style

- **Readability**: Code flows logically, showing the sequence of transformations in a single, cohesive block.
- **Conciseness**: Fewer variables and less repetition reduce boilerplate.
- **Composability**: Small reusable pieces can be combined naturally, especially with functional arguments.
- **Data-driven flexibility**: Functions allow styling and attributes to depend on dynamic data, unlocking powerful visual mappings.
- **Cleaner updates**: When updating elements, you can chain transitions, attribute changes, and event bindings seamlessly.

2.4.5 Summary

D3's API encourages you to think of your visualization as a **pipeline of transformations**, chaining methods that successively modify elements or bind behaviors. By combining this with functions that compute values based on data, you get a **declarative, elegant style** that scales from simple charts to complex, interactive graphics.

This chaining and functional mindset is key to mastering D3 and writing clean, maintainable visualization code.

Chapter 3.
Drawing with SVG and D3

1. Introduction to SVG in HTML
2. Creating Shapes with D3 (`rect`, `circle`, `path`, `line`, etc.)
3. Grouping and Transforming Elements (`<g>`, `transform`)
4. Styling with CSS vs D3

3 Drawing with SVG and D3

3.1 Introduction to SVG in HTML

When creating visualizations on the web, one of the most powerful tools at your disposal is **SVG**, or **Scalable Vector Graphics**. Unlike raster images (such as PNGs or JPEGs), SVGs are **vector-based**, meaning they describe graphics using geometric shapes, lines, and curves. This makes them infinitely scalable without loss of quality—perfect for crisp, detailed charts and interactive visuals.

3.1.1 What Is SVG?

SVG is an **XML-based markup language** for describing two-dimensional graphics. Because it is text-based and part of the HTML standard, you can embed SVG code directly inside your web pages.

The browser then renders the SVG just like any other DOM element, which means you can style it with CSS and manipulate it with JavaScript—especially with libraries like D3.

3.1.2 Basic SVG Structure and Coordinate Space

An SVG image lives inside an `<svg>` element, which defines its **canvas size** and coordinate system. Inside this canvas, you add shapes like circles, rectangles, lines, and paths.

Example of a simple SVG container:

```
<svg width="400" height="200" style="border: 1px solid #ccc;">
  <!-- Shapes will go here -->
</svg>
```

- `width` and `height` set the size of the SVG viewport in pixels.
- The coordinate system's origin `(0, 0)` is located at the **top-left corner**.
- The x-axis extends right, and the y-axis extends downward.
- Coordinates and sizes are specified in the same unit as the SVG viewport (by default, pixels).

3.1.3 Manually Creating Basic SVG Shapes

Before automating shape creation with D3, let's build intuition by creating some simple shapes using plain SVG markup.

Rectangle (`rect`)

```
<svg width="200" height="100" style="border:1px solid #ccc;">
  <rect x="10" y="10" width="100" height="50" fill="steelblue" />
</svg>
```

- `x` and `y` specify the rectangle's top-left corner position.
- `width` and `height` control the size.
- `fill` sets the interior color.

Circle (`circle`)

```
<svg width="200" height="100" style="border:1px solid #ccc;">
  <circle cx="100" cy="50" r="40" fill="tomato" />
</svg>
```

- `cx` and `cy` define the center of the circle.
- `r` is the radius.
- `fill` sets the color.

Line (`line`)

```
<svg width="200" height="100" style="border:1px solid #ccc;">
  <line x1="10" y1="90" x2="190" y2="10" stroke="green" stroke-width="4" />
</svg>
```

- `x1`, `y1` and `x2`, `y2` define the start and end points.
- `stroke` sets the line color.
- `stroke-width` controls thickness.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Basic SVG Shapes</title>
  <style>
    body {
      font-family: sans-serif;
      margin: 20px;
    }
    h2 {
      margin-top: 40px;
    }
    svg {
      margin-bottom: 20px;
      display: block;
    }
  </style>
</head>
<body>

  <h1>Manually Created SVG Shapes</h1>
  <h2>Rectangle</h2>
  <svg width="200" height="100" style="border:1px solid #ccc;">
    <rect x="10" y="10" width="100" height="50" fill="steelblue" />
  </svg>

  <h2>Circle</h2>
  <svg width="200" height="100" style="border:1px solid #ccc;">
    <circle cx="100" cy="50" r="40" fill="tomato" />
  </svg>

  <h2>Line</h2>
  <svg width="200" height="100" style="border:1px solid #ccc;">
    <line x1="10" y1="90" x2="190" y2="10" stroke="green" stroke-width="4" />
  </svg>

</body>
</html>
```

3.1.4 Understanding the Coordinate Space

All these shapes are positioned relative to the SVG's coordinate space:

- Coordinates start at `(0,0)` in the **top-left corner**.
- Positive x values move right.
- Positive y values move down.
- Coordinates and lengths use the same units (pixels unless otherwise specified).

Because SVG graphics are vector-based, shapes can scale smoothly if you resize the `<svg>` element or use the `viewBox` attribute (explored later).

3.1.5 Summary

SVG is a foundational web technology for creating crisp, scalable graphics through markup embedded directly in HTML. Understanding its coordinate system and basic shapes is essential before automating with D3's powerful data-driven methods.

In this chapter, you'll soon learn how D3 dynamically creates and manipulates SVG elements to build rich, interactive visualizations—starting from these simple building blocks.

3.2 Creating Shapes with D3 (`rect`, `circle`, `path`, `line`, etc.)

One of D3's core strengths is its ability to **dynamically create and manipulate SVG shapes** based on data. Instead of manually writing out each SVG element, you can bind an array of data to SVG elements and let D3 generate the shapes for you — perfect for charts, graphs, and interactive visualizations.

In this section, we'll explore how to use D3 to create basic SVG shapes such as `<rect>`, `<circle>`, and `<path>`, with examples that illustrate a simple bar chart, scatter plot, and line graph.

3.2.1 Creating a Bar Chart with `rect`

A bar chart is a classic visualization where each bar's height corresponds to a data value. D3 makes it easy to bind data to rectangles and position them accordingly.

```html
<svg width="500" height="150"></svg>
```

```javascript
const data = [80, 120, 60, 150, 200];
const svg = d3.select("svg");
const barWidth = 40;
const barSpacing = 10;

svg.selectAll("rect")
  .data(data)
  .enter()
  .append("rect")
  .attr("x", (d, i) => i * (barWidth + barSpacing))
  .attr("y", d => 150 - d)      // SVG origin is top-left; subtract to grow bars upwards
  .attr("width", barWidth)
  .attr("height", d => d)
  .attr("fill", "steelblue");
```

Explanation:

- We bind the `data` array to a selection of `<rect>` elements.
- For each data point, a rectangle is appended.
- `x` positions bars horizontally with spacing.
- `y` and `height` position bars so they grow upwards.
- Bars are styled with `fill`.

Full runnable code:

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Bar Chart</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 20px;
    }
    svg {
      border: 1px solid #ccc;
    }
  </style>
</head>
<body>

  <h2>Bar Chart with D3</h2>
  <svg width="500" height="150"></svg>

  <script>
    const data = [80, 120, 60, 150, 200];
    const svg = d3.select("svg");
    const barWidth = 40;
    const barSpacing = 10;

    svg.selectAll("rect")
      .data(data)
      .enter()
      .append("rect")
      .attr("x", (d, i) => i * (barWidth + barSpacing))
      .attr("y", d => 150 - d) // SVG y starts at top, so we subtract to grow upward
      .attr("width", barWidth)
      .attr("height", d => d)
      .attr("fill", "steelblue");
  </script>

</body>
</html>
```

3.2.2 Creating a Scatter Plot with `circle`

Scatter plots use circles to represent data points positioned by coordinates.

```
<svg width="500" height="150"></svg>
```

```
const data = [
  { x: 30, y: 20 },
  { x: 80, y: 90 },
  { x: 150, y: 50 },
  { x: 220, y: 120 },
];

const svg = d3.select("svg");

svg.selectAll("circle")
  .data(data)
  .enter()
  .append("circle")
  .attr("cx", d => d.x)
  .attr("cy", d => 150 - d.y)  // Invert y to match SVG coordinate system
  .attr("r", 8)
  .attr("fill", "tomato");
```

Explanation:

- Data points are objects with `x` and `y` properties.
- Circles are created and positioned by setting `cx` and `cy`.
- The y-axis is flipped by subtracting `y` from the SVG height to maintain intuitive graph orientation.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Scatter Plot</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 20px;
    }
    svg {
      border: 1px solid #ccc;
    }
  </style>
</head>
<body>

  <h2>Scatter Plot with D3</h2>
  <svg width="500" height="150"></svg>

  <script>
    const data = [
      { x: 30, y: 20 },
      { x: 80, y: 90 },
      { x: 150, y: 50 },
      { x: 220, y: 120 },
    ];

    const svg = d3.select("svg");

    svg.selectAll("circle")
      .data(data)
      .enter()
      .append("circle")
      .attr("cx", d => d.x)
      .attr("cy", d => 150 - d.y)  // Invert y for SVG coordinates
      .attr("r", 8)
      .attr("fill", "tomato");
  </script>

</body>
</html>
```

3.2.3 Creating a Line Graph with `path` and `d3.line()`

Lines are a bit different: instead of one element per data point, a single `<path>` element describes a continuous line connecting the points. D3's `d3.line()` generator makes this easy.

```
<svg width="500" height="150"></svg>
```

```
const data = [
  { x: 0, y: 80 },
  { x: 50, y: 120 },
  { x: 100, y: 60 },
  { x: 150, y: 130 },
  { x: 200, y: 90 },
];

const svg = d3.select("svg");

const lineGenerator = d3.line()
  .x(d => d.x)
  .y(d => 150 - d.y)  // Invert y to keep graph upright
  .curve(d3.curveMonotoneX);  // Optional smooth curve

svg.append("path")
  .datum(data)                 // Bind entire data array to one path element
  .attr("d", lineGenerator)   // Generate the SVG path string
  .attr("fill", "none")
  .attr("stroke", "steelblue")
  .attr("stroke-width", 3);
```

Explanation:

- `d3.line()` is a path generator that converts data points to a path `d` attribute string.
- `.datum(data)` binds the entire dataset to one `<path>`.
- The path draws a smooth line through all points.
- Styling is applied with stroke color and width, and fill is disabled.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Line Graph</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 20px;
    }
    svg {
      border: 1px solid #ccc;
    }
  </style>
</head>
<body>

  <h2>D3 Line Graph</h2>
  <svg width="500" height="150"></svg>

  <script>
    const data = [
      { x: 0, y: 80 },
      { x: 50, y: 120 },
      { x: 100, y: 60 },
      { x: 150, y: 130 },
      { x: 200, y: 90 },
    ];

    const svg = d3.select("svg");

    const lineGenerator = d3.line()
      .x(d => d.x)
      .y(d => 150 - d.y)  // Flip vertically for SVG space
      .curve(d3.curveMonotoneX);  // Smooth line

    svg.append("path")
      .datum(data)
      .attr("d", lineGenerator)
      .attr("fill", "none")
      .attr("stroke", "steelblue")
      .attr("stroke-width", 3);
  </script>

</body>
</html>
```

3.2.4 Summary

Shape	How to Create in D3	Use Case
`<rect>`	`.append("rect").attr(...)`	Bar charts, histograms
`<circle>`	`.append("circle").attr(...)`	Scatter plots, bubble charts
`<path>`	`.append("path").attr("d", d3.line())`	Line charts, area charts

By binding arrays of data to SVG elements, D3 makes creating complex, data-driven shapes intuitive and efficient. You specify *what* to draw and *how* to map your data, while D3 manages the DOM operations behind the scenes.

In the next section, you'll learn how to group SVG elements and apply transformations, making your visuals more organized and dynamic.

3.3 Grouping and Transforming Elements (`<g>`, `transform`)

When building complex visualizations, managing individual SVG shapes one by one quickly becomes cumbersome. This is where the SVG `<g>` element—short for **group**—shines. It lets you **group multiple shapes together** so you can manipulate them as a single unit.

Using `<g>` elements combined with the powerful `transform` attribute lets you organize, position, and modularize your chart components like bars, axes, legends, and labels cleanly and efficiently.

3.3.1 What Is the g Element?

The `<g>` element is a container that groups SVG shapes and other `<g>` elements. It does not render anything on its own but allows you to apply attributes—like styles or transformations—to the entire group at once.

For example:

```
<svg width="400" height="150" style="border:1px solid #ccc;">
  <g id="myGroup" transform="translate(50,20)">
    <rect x="0" y="0" width="50" height="100" fill="steelblue" />
    <circle cx="75" cy="50" r="30" fill="tomato" />
  </g>
</svg>
```

Here, the group `myGroup` shifts both the rectangle and circle by `(50, 20)` pixels.

3.3.2 Why Use g and `transform`?

- **Positioning:** Instead of calculating and applying positions individually for every shape, group them and translate or rotate the whole group.
- **Modularity:** Organize visual elements into logical chunks, such as a **chart body**, **axes**, or **legend**.
- **Maintainability:** Apply styles, events, or transformations once at the group level instead of duplicating them.
- **Nesting:** Groups can contain other groups, enabling hierarchical structure and complex layouts.

3.3.3 Example: Grouping Bars in a Bar Chart and Translating It

Let's build on a simple bar chart example and use `<g>` to group all the bars, then translate the group to add margins or position the chart inside the SVG.

```html
<svg width="500" height="200"></svg>
```

```javascript
const data = [80, 120, 60, 150, 200];
const svg = d3.select("svg");

// Create a group for the bars and move it 50 pixels right and 20 pixels down
const chartGroup = svg.append("g")
  .attr("transform", "translate(50, 20)");

const barWidth = 40;
const barSpacing = 10;

chartGroup.selectAll("rect")
  .data(data)
  .enter()
  .append("rect")
  .attr("x", (d, i) => i * (barWidth + barSpacing))
  .attr("y", d => 150 - d)
  .attr("width", barWidth)
  .attr("height", d => d)
  .attr("fill", "steelblue");
```

Full runnable code:

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Grouped Bar Chart</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    svg {
      border: 1px solid #ccc;
    }
  </style>
</head>
<body>

  <h2>D3 Bar Chart with Group Translation</h2>
  <svg width="500" height="200"></svg>

  <script>
    const data = [80, 120, 60, 150, 200];
    const svg = d3.select("svg");

    // Create a group for the bars and translate it (margin)
    const chartGroup = svg.append("g")
      .attr("transform", "translate(50, 20)");

    const barWidth = 40;
    const barSpacing = 10;

    chartGroup.selectAll("rect")
      .data(data)
      .enter()
      .append("rect")
      .attr("x", (d, i) => i * (barWidth + barSpacing))
      .attr("y", d => 150 - d)  // SVG y=0 is top
      .attr("width", barWidth)
      .attr("height", d => d)
      .attr("fill", "steelblue");
  </script>

</body>
</html>
```

3.3.4 How This Helps

- The whole set of bars is **shifted inside the SVG canvas** by the group's `transform`.
- This approach keeps your code modular; you can add separate groups for:
  - **Axes:** Positioned next to the chart body.
  - **Legends:** Positioned outside the main chart area.
  - **Annotations:** Grouped for easy control.

Grouping and translating avoids manually adjusting every bar's position to account for margins, making your layout simpler and easier to update.

3.3.5 Summary

- The SVG `<g>` element groups multiple shapes or other groups into one logical unit.
- The `transform` attribute (e.g., `translate(x, y)`, `rotate(angle)`) applies transformations to the entire group.
- Using groups lets you modularize your chart into components like chart bodies, axes, and legends.
- This improves maintainability, readability, and flexibility of your visualizations.

Next, we'll explore how to style SVG elements effectively using both CSS and D3's built-in style methods.

3.4 Styling with CSS vs D3

Styling is a crucial part of making your data visualizations not only informative but also visually appealing. In SVG with D3, you have two primary ways to style elements: **using CSS stylesheets** and **applying styles programmatically through D3's API**. Each approach has its strengths and best use cases.

3.4.1 Styling with CSS Classes

Using CSS is a natural, clean way to style your SVG elements. You define styles in an external or internal stylesheet and apply classes to elements, keeping style concerns separate from your JavaScript logic.

Example:

```css
.bar {
  fill: steelblue;
  stroke: black;
  stroke-width: 1px;
  transition: fill 0.3s;
}

.bar:hover {
  fill: orange;
}
```

```javascript
d3.selectAll("rect")
  .data(data)
  .enter()
  .append("rect")
  .attr("class", "bar")
  .attr("x", (d, i) => i * 45)
  .attr("y", d => 150 - d)
  .attr("width", 40)
  .attr("height", d => d);
```

Full runnable code:

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Bar Chart with CSS Styling</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    svg {
      border: 1px solid #ccc;
    }

    .bar {
      fill: steelblue;
      stroke: black;
      stroke-width: 1px;
      transition: fill 0.3s;
    }

    .bar:hover {
      fill: orange;
    }
  </style>
</head>
<body>

  <h2>D3 Bar Chart Styled with CSS Classes</h2>
  <svg width="500" height="200"></svg>

  <script>
    const data = [80, 120, 60, 150, 200];
    const svg = d3.select("svg");

    const barWidth = 40;
    const barSpacing = 5;

    svg.selectAll("rect")
      .data(data)
      .enter()
      .append("rect")
      .attr("class", "bar")
      .attr("x", (d, i) => i * (barWidth + barSpacing))
      .attr("y", d => 150 - d)
      .attr("width", barWidth)
      .attr("height", d => d);
  </script>

</body>
</html>
```

Advantages of CSS classes:

- Keeps styles centralized and reusable.
- Easier to maintain and update across large projects.
- Supports media queries and CSS features like pseudo-classes (`:hover`, `:active`).
- Encourages separation of concerns between style and logic.

3.4.2 Styling Inline with D3s `.style()` and `.attr()`

D3 also lets you set styles and attributes **directly on elements** via methods like `.style()` and `.attr()`. This is useful when you want to compute styles dynamically based on data or interaction state.

Example:

```
d3.selectAll("rect")
  .style("fill", d => d > 100 ? "tomato" : "steelblue")
  .style("stroke", "black")
  .style("stroke-width", "1px");
```

You can also dynamically change styles on events:

```
d3.selectAll("rect")
  .on("mouseover", function() {
    d3.select(this).style("fill", "orange");
  })
  .on("mouseout", function() {
    d3.select(this).style("fill", null); // revert to original
  });
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Inline Styling Example</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    svg {
      border: 1px solid #ccc;
    }
  </style>
</head>
<body>

  <h2>D3 Inline Styling with Events</h2>
  <svg width="500" height="200"></svg>

  <script>
    const data = [80, 120, 60, 150, 200];
    const svg = d3.select("svg");
    const barWidth = 40;
    const barSpacing = 10;

    svg.selectAll("rect")
      .data(data)
      .enter()
      .append("rect")
      .attr("x", (d, i) => i * (barWidth + barSpacing))
      .attr("y", d => 150 - d)
      .attr("width", barWidth)
      .attr("height", d => d)
      .style("fill", d => d > 100 ? "tomato" : "steelblue")
      .style("stroke", "black")
      .style("stroke-width", "1px")
      .on("mouseover", function () {
        d3.select(this).style("fill", "orange");
      })
      .on("mouseout", function (event, d) {
        // Recompute original fill based on data
        d3.select(this).style("fill", d > 100 ? "tomato" : "steelblue");
      });
  </script>

</body>
</html>
```

3.4.3 Using `.classed()` to Toggle CSS Classes

D3's `.classed()` method provides a powerful way to add or remove CSS classes dynamically, combining the best of both worlds:

```
d3.selectAll("rect")
  .on("mouseover", function() {
    d3.select(this).classed("highlighted", true);
  })
  .on("mouseout", function() {
    d3.select(this).classed("highlighted", false);
  });
```

```
.highlighted {
  fill: orange !important;
  stroke-width: 2px;
}
```

This approach allows you to leverage CSS for styling and transitions while controlling when styles apply via JavaScript.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 .classed() Toggle Example</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    svg {
      border: 1px solid #ccc;
    }
    rect {
      fill: steelblue;
      stroke: black;
      stroke-width: 1px;
      transition: fill 0.3s, stroke-width 0.3s;
    }
    .highlighted {
      fill: orange !important;
      stroke-width: 2px;
    }
  </style>
</head>
<body>
  <h2>D3 .classed() Toggle CSS Class on Hover</h2>
  <svg width="500" height="200"></svg>

  <script>
    const data = [80, 120, 60, 150, 200];
    const svg = d3.select("svg");
    const barWidth = 40;
    const barSpacing = 10;

    svg.selectAll("rect")
      .data(data)
      .enter()
      .append("rect")
      .attr("x", (d, i) => i * (barWidth + barSpacing))
      .attr("y", d => 150 - d)
      .attr("width", barWidth)
      .attr("height", d => d)
      .on("mouseover", function() {
        d3.select(this).classed("highlighted", true);
      })
      .on("mouseout", function() {
        d3.select(this).classed("highlighted", false);
      });
  </script>
</body>
</html>
```

3.4.4 Best Practices

When to Use CSS Stylesheets	When to Use D3 Programmatic Styling
Static, reusable styles shared across visuals	Dynamic, data-driven styles based on values
Pseudo-classes and media queries (e.g., hover effects)	Interactive style changes triggered by events
Centralized style management for consistency	Quick overrides or conditional styling
Complex selectors and cascading styles	Fine-grained control on per-element basis

3.4.5 Summary

- **CSS classes** promote separation of concerns, maintainability, and reusability.
- **D3 inline styling** is great for dynamic, data-dependent, or interactive style changes.
- `.classed()` offers a hybrid method to toggle CSS classes dynamically.
- Combining CSS and D3 styles smartly results in clean, scalable visualizations.

By mastering when and how to style with CSS versus D3, you ensure your visuals remain both beautiful and maintainable as your projects grow.

Chapter 4.

Scales and Axes

1. Linear, Ordinal, Band, Time Scales
2. Color Scales and Interpolation
3. Creating and Customizing Axes
4. Handling Scale Updates

4 Scales and Axes

4.1 Linear, Ordinal, Band, Time Scales

In data visualization, **mapping raw data values to visual dimensions**—such as pixel positions, lengths, or colors—is fundamental. D3's powerful **scale functions** perform this mapping by transforming input data from its original domain into a defined range, typically corresponding to screen coordinates or sizes.

Understanding these scales is key to making your visualizations accurate, readable, and adaptable.

4.1.1 What Are Scales?

A **scale** in D3 is a function that takes an input value (from your data) and outputs a corresponding visual value, usually within a specific range on the screen.

Basic form:

```
const scale = d3.scaleSomething()
  .domain([dataMin, dataMax])   // input data values
  .range([pixelMin, pixelMax]); // output visual values (e.g., pixels)
```

For example, if your data values run from 0 to 100, but your chart width is 400 pixels, a scale converts data value `50` to pixel position `200`.

4.1.2 `scaleLinear` Continuous Numeric Data

`scaleLinear` maps continuous numerical input to continuous output, great for things like scatter plots or line charts.

```
const xScale = d3.scaleLinear()
  .domain([0, 100])    // data values from 0 to 100
  .range([0, 400]);    // pixels from 0 to 400
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 scaleLinear Example</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
</head>
<body>
  <svg width="450" height="100" style="border:1px solid #ccc;"></svg>
  <script>
    const svg = d3.select("svg");

    // Define scaleLinear mapping data domain to pixel range
    const xScale = d3.scaleLinear()
      .domain([0, 100])   // input data range
      .range([0, 400]);   // output pixel range

    // Example data points
    const data = [0, 25, 50, 75, 100];

    svg.selectAll("circle")
      .data(data)
      .enter()
      .append("circle")
      .attr("cx", d => xScale(d))
      .attr("cy", 50)
      .attr("r", 8)
      .attr("fill", "steelblue");

    // Add labels to see exact pixel mapping
    svg.selectAll("text")
      .data(data)
      .enter()
      .append("text")
      .attr("x", d => xScale(d))
      .attr("y", 80)
      .attr("text-anchor", "middle")
      .text(d => d);
  </script>
</body>
</html>
```

Example:

Data input	Output (pixels)
0	0
50	200
100	400

Visualizing this as a horizontal axis, a data point with value 50 will be positioned halfway along the 400-pixel wide SVG.

4.1.3 `scaleOrdinal` Discrete Categories

`scaleOrdinal` maps discrete input values (like categories or labels) to output values, often colors or specific positions.

```
const colorScale = d3.scaleOrdinal()
  .domain(["apple", "banana", "cherry"])
  .range(["red", "yellow", "darkred"]);
```

If you pass `"banana"` to `colorScale("banana")`, you get `"yellow"`.

Ordinal scales don't interpolate between values—they pick the exact matching output from the range.

4.1.4 `scaleBand` Categorical Positioning with Bands

`scaleBand` is designed for **categorical data that needs to be mapped to discrete bands or blocks** in a visual, such as bars in a bar chart. It computes evenly spaced bands along a range, allowing you to position and size elements with padding.

```
const xScale = d3.scaleBand()
  .domain(["Q1", "Q2", "Q3", "Q4"])
  .range([0, 400])
  .padding(0.1);
```

- `.domain()` sets the categories.
- `.range()` sets the pixel range.
- `.padding()` adds space between bands.

Example output:

Category	Position (`xScale(category)`)	Bandwidth (`xScale.bandwidth()`)
Q1	0	90
Q2	100	90
Q3	200	90
Q4	300	90

You use `xScale(category)` for the band's start position and `xScale.bandwidth()` for its width.

4.1.5 `scaleTime` Mapping Dates and Times

For datasets with time-based data, `scaleTime` maps date objects or timestamps to continuous output.

```
const timeScale = d3.scaleTime()
  .domain([new Date(2023, 0, 1), new Date(2023, 11, 31)])  // Jan 1 to Dec 31, 2023
  .range([0, 600]);                                        // pixels along an axis
```

Passing a date like `new Date(2023, 5, 15)` returns a pixel position proportionally between 0 and 600, allowing you to position events on a timeline.

4.1.6 Visualizing Scale Transformation

Suppose you want to place circles horizontally based on numeric data `[10, 50, 90]` using a linear scale:

```
const xScale = d3.scaleLinear()
  .domain([0, 100])
  .range([0, 300]);

svg.selectAll("circle")
  .data([10, 50, 90])
  .enter()
  .append("circle")
  .attr("cx", d => xScale(d))  // Transforms data to pixel x-position
  .attr("cy", 50)
  .attr("r", 10)
  .attr("fill", "teal");
```

This results in circles positioned at 30, 150, and 270 pixels respectively.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Scale Transformation Example</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    svg {
      border: 1px solid #ccc;
    }
  </style>
</head>
<body>
  <svg width="400" height="100"></svg>

  <script>
    const svg = d3.select("svg");

    const xScale = d3.scaleLinear()
      .domain([0, 100])
      .range([0, 300]);

    const data = [10, 50, 90];

    svg.selectAll("circle")
      .data(data)
      .enter()
      .append("circle")
      .attr("cx", d => xScale(d))  // Scaled x-position
      .attr("cy", 50)
      .attr("r", 10)
      .attr("fill", "teal");
  </script>
</body>
</html>
```

4.1.7 Summary

Scale Type	Input Domain	Output Range	Use Case
`scaleLinear`	Continuous numeric	Continuous numeric (pixels)	Scatter plots, line charts
`scaleOrdinal`	Discrete categories (strings)	Discrete outputs (colors, positions)	Categorical color mappings
`scaleBand`	Discrete categories (strings)	Discrete pixel bands	Bar chart positions and widths
`scaleTime`	Date/time objects	Continuous numeric (pixels)	Timelines, time series charts

By mastering these scales, you gain precise control over how data translates to visuals — the foundation of any effective D3 visualization.

4.2 Color Scales and Interpolation

Color is a powerful visual encoding in data visualization, helping users quickly grasp patterns, differences, and relationships in data. D3 provides a rich set of **color scales and interpolation functions** to map data values to colors smoothly or categorically, making your visuals both informative and attractive.

4.2.1 D3 Color Scales Overview

D3 offers several types of color scales suited for different data types:

- **Sequential scales (`scaleSequential`)** map continuous numeric input to a smooth gradient of colors.
- **Ordinal scales (`scaleOrdinal`)** assign distinct colors to categorical data.

- A variety of **interpolator functions** (`interpolate*`) define how colors transition along a gradient.
- Built-in **color schemes** provide ready-to-use palettes for common needs.

4.2.2 Sequential Color Scales and Interpolators

`d3.scaleSequential` maps a continuous domain (e.g., temperature, intensity) to colors by interpolating between two or more colors.

```
const sequentialScale = d3.scaleSequential()
  .domain([0, 100])                 // Input data range
  .interpolator(d3.interpolateViridis);  // Color gradient (blue to yellow-green)
```

Passing a value to `sequentialScale` returns a color string:

```
sequentialScale(0);    // "#440154" (dark purple)
sequentialScale(50);   // "#21908d" (teal)
sequentialScale(100);  // "#fde725" (yellow)
```

You can visualize this gradient as a color bar or use it to color shapes dynamically.

4.2.3 Color Interpolation Functions

D3 provides many built-in interpolators such as:

- `d3.interpolateViridis` (perceptually uniform)
- `d3.interpolateInferno`
- `d3.interpolateMagma`
- `d3.interpolatePlasma`
- `d3.interpolateRainbow`
- `d3.interpolateCool`
- `d3.interpolateWarm`

Each generates smooth transitions between colors optimized for different uses and aesthetics.

4.2.4 Ordinal Color Scales for Categories

For categorical data (e.g., types, groups), use `d3.scaleOrdinal()` to assign distinct colors from a fixed palette.

```
const categoryScale = d3.scaleOrdinal()
  .domain(["Apple", "Banana", "Cherry", "Date"])
  .range(d3.schemeSet2);  // A predefined color scheme of distinct colors
```

Calling `categoryScale("Banana")` might return a bright yellow, while `"Cherry"` maps to red.

4.2.5 Example: Building a Basic Heatmap Using Sequential Scale

Imagine you have a grid of values from 0 to 100, and you want to color each cell according to its intensity.

```
const data = [10, 30, 55, 80, 95];
const colorScale = d3.scaleSequential()
  .domain([0, 100])
  .interpolator(d3.interpolatePlasma);

const svg = d3.select("svg");

svg.selectAll("rect")
  .data(data)
  .enter()
  .append("rect")
  .attr("x", (d, i) => i * 30)
  .attr("y", 0)
  .attr("width", 25)
  .attr("height", 25)
  .attr("fill", d => colorScale(d));
```

Each rectangle's fill color corresponds to its data value's position along the plasma color gradient.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Sequential Heatmap Example</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
</head>
<body>
  <svg width="200" height="50" style="border:1px solid #ccc;"></svg>

  <script>
    const data = [10, 30, 55, 80, 95];

    const colorScale = d3.scaleSequential()
      .domain([0, 100])
      .interpolator(d3.interpolatePlasma);

    const svg = d3.select("svg");

    svg.selectAll("rect")
      .data(data)
      .enter()
      .append("rect")
      .attr("x", (d, i) => i * 30)
      .attr("y", 0)
      .attr("width", 25)
      .attr("height", 25)
      .attr("fill", d => colorScale(d));
  </script>
</body>
</html>
```

4.2.6 Example: Creating a Categorical Legend with `scaleOrdinal`

```
const categories = ["Low", "Medium", "High"];
const colorScale = d3.scaleOrdinal()
  .domain(categories)
  .range(["#2ca02c", "#ff7f0e", "#d62728"]); // green, orange, red

const svg = d3.select("svg");

svg.selectAll("rect")
  .data(categories)
  .enter()
  .append("rect")
  .attr("x", (d, i) => i * 60)
  .attr("y", 0)
  .attr("width", 50)
  .attr("height", 20)
  .attr("fill", d => colorScale(d));

svg.selectAll("text")
  .data(categories)
  .enter()
  .append("text")
  .attr("x", (d, i) => i * 60 + 25)
  .attr("y", 40)
  .attr("text-anchor", "middle")
  .text(d => d);
```

This creates a simple legend mapping each category to a color block and label.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Categorical Legend Example</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    text {
      font-family: sans-serif;
      font-size: 14px;
    }
  </style>
```

```html
</head>
<body>
  <svg width="200" height="60" style="border:1px solid #ccc;"></svg>

  <script>
    const categories = ["Low", "Medium", "High"];
    const colorScale = d3.scaleOrdinal()
      .domain(categories)
      .range(["#2ca02c", "#ff7f0e", "#d62728"]); // green, orange, red

    const svg = d3.select("svg");

    svg.selectAll("rect")
      .data(categories)
      .enter()
      .append("rect")
      .attr("x", (d, i) => i * 60)
      .attr("y", 0)
      .attr("width", 50)
      .attr("height", 20)
      .attr("fill", d => colorScale(d));

    svg.selectAll("text")
      .data(categories)
      .enter()
      .append("text")
      .attr("x", (d, i) => i * 60 + 25)
      .attr("y", 40)
      .attr("text-anchor", "middle")
      .text(d => d);
  </script>
</body>
</html>
```

4.2.7 Summary

- Use **`scaleSequential`** with an **interpolator** for smooth color gradients mapped from continuous numeric data.
- Use **`scaleOrdinal`** for assigning distinct colors to categorical data.
- D3 provides many **interpolator functions** and **color schemes** that are perceptually optimized.
- Combining color scales with your data-driven shapes lets you build heatmaps, legends, and color-encoded charts that enhance insight and visual appeal.

Mastering color scales unlocks a vital dimension of effective, expressive data visualizations.

4.3 Creating and Customizing Axes

Axes are essential components of many visualizations, providing context that helps users understand the data's scale and meaning. D3 offers convenient axis generators like `d3.axisBottom()` and `d3.axisLeft()` to create and customize axes easily.

In this section, we'll explore how to build X and Y axes, adjust tick formatting and density, and add gridlines—all with a practical scatter plot example.

4.3.1 Basic Axes with D3

D3 provides axis generator functions that create axes based on scales:

- **`d3.axisBottom(scale)`**: Creates a horizontal axis positioned below the chart (typically the X axis).
- **`d3.axisLeft(scale)`**: Creates a vertical axis positioned on the left side (typically the Y axis).

You supply a scale (like `scaleLinear` or `scaleTime`), and the axis generator handles creating ticks, labels, and lines.

4.3.2 Building a Scatter Plot with Axes: Full Example

```
<svg width="500" height="400"></svg>
```

```
const svg = d3.select("svg");
const margin = { top: 20, right: 20, bottom: 50, left: 60 };
const width = +svg.attr("width") - margin.left - margin.right;
const height = +svg.attr("height") - margin.top - margin.bottom;

// Sample data: points with x (percentage) and y (value)
const data = [
  { x: 0.1, y: 30 },
  { x: 0.4, y: 80 },
  { x: 0.6, y: 45 },
  { x: 0.9, y: 120 },
];

// Create group for chart body
const chart = svg.append("g")
  .attr("transform", `translate(${margin.left},${margin.top})`);

// Define scales
const xScale = d3.scaleLinear()
  .domain([0, 1])              // Input: 0% to 100%
  .range([0, width]);

const yScale = d3.scaleLinear()
```

```
  .domain([0, 130])            // Input: data max + some padding
  .range([height, 0]);         // Note: SVG y=0 is top

// Create axes
const xAxis = d3.axisBottom(xScale)
  .ticks(5)
  .tickFormat(d3.format(".0%"));  // Format as percentage

const yAxis = d3.axisLeft(yScale)
  .ticks(6);

// Append X axis
chart.append("g")
  .attr("transform", `translate(0, ${height})`)  // Move to bottom
  .call(xAxis)
  .call(g => g.append("text")                     // X axis label
    .attr("x", width / 2)
    .attr("y", 40)
    .attr("fill", "black")
    .attr("text-anchor", "middle")
    .text("Percentage"));

// Append Y axis
chart.append("g")
  .call(yAxis)
  .call(g => g.append("text")                     // Y axis label
    .attr("x", -height / 2)
    .attr("y", -45)
    .attr("transform", "rotate(-90)")
    .attr("fill", "black")
    .attr("text-anchor", "middle")
    .text("Value"));

// Plot data points
chart.selectAll("circle")
  .data(data)
  .enter()
  .append("circle")
  .attr("cx", d => xScale(d.x))
  .attr("cy", d => yScale(d.y))
  .attr("r", 7)
  .attr("fill", "steelblue");
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Scatter Plot with Axes</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
    }
    .axis-label {
      font-size: 14px;
      font-weight: bold;
    }
  </style>
</head>
<body>
  <svg width="500" height="400" style="border: 1px solid #ccc;"></svg>

  <script>
    const svg = d3.select("svg");
    const margin = { top: 20, right: 20, bottom: 50, left: 60 };
    const width = +svg.attr("width") - margin.left - margin.right;
    const height = +svg.attr("height") - margin.top - margin.bottom;

    // Sample data: points with x (percentage) and y (value)
    const data = [
      { x: 0.1, y: 30 },
      { x: 0.4, y: 80 },
      { x: 0.6, y: 45 },
      { x: 0.9, y: 120 },
    ];

    // Create group for chart body
    const chart = svg.append("g")
      .attr("transform", `translate(${margin.left},${margin.top})`);

    // Define scales
    const xScale = d3.scaleLinear()
      .domain([0, 1])              // Input: 0% to 100%
      .range([0, width]);

    const yScale = d3.scaleLinear()
      .domain([0, 130])            // Input: data max + some padding
      .range([height, 0]);         // Note: SVG y=0 is top

    // Create axes
    const xAxis = d3.axisBottom(xScale)
      .ticks(5)
      .tickFormat(d3.format(".0%"));  // Format as percentage

    const yAxis = d3.axisLeft(yScale)
      .ticks(6);

    // Append X axis
    chart.append("g")
      .attr("transform", `translate(0, ${height})`)  // Move to bottom
      .call(xAxis)
      .call(g => g.append("text")                     // X axis label
        .attr("class", "axis-label")
        .attr("x", width / 2)
        .attr("y", 40)
        .attr("fill", "black")
        .attr("text-anchor", "middle")
        .text("Percentage"));

    // Append Y axis
    chart.append("g")
      .call(yAxis)
      .call(g => g.append("text")                     // Y axis label
        .attr("class", "axis-label")
        .attr("x", -height / 2)
        .attr("y", -45)
        .attr("transform", "rotate(-90)")
        .attr("fill", "black")
        .attr("text-anchor", "middle")
        .text("Value"));

    // Plot data points
    chart.selectAll("circle")
      .data(data)
      .enter()
      .append("circle")
      .attr("cx", d => xScale(d.x))
      .attr("cy", d => yScale(d.y))
      .attr("r", 7)
      .attr("fill", "steelblue");
  </script>
</body>
</html>
```

4.3.3 Customizing Tick Density and Formatting

- `.ticks(count)` suggests the number of ticks (D3 may adjust for readability).
- `.tickFormat(formatFunction)` customizes how tick labels appear.

For example, the X axis above formats numbers as percentages (`0.1 → 10%`).

You can create your own formatter or use built-in ones like:

- `d3.format(".2f")` — fixed decimals
- `d3.format(",")` — comma separators for thousands
- `d3.timeFormat("%b %d")` — for dates (used with `scaleTime`)

4.3.4 Adding Gridlines for Better Readability

Gridlines help users visually align data points with axis ticks.

```
// Add horizontal gridlines (Y axis)
chart.append("g")
  .attr("class", "grid")
  .call(d3.axisLeft(yScale)
    .ticks(6)
    .tickSize(-width)        // Extend ticks across chart width
    .tickFormat("")          // No labels for gridlines
  );
```

Apply CSS to style gridlines:

```
.grid line {
  stroke: lightgray;
  stroke-opacity: 0.7;
  shape-rendering: crispEdges;
}

.grid path {
  stroke-width: 0;
}
```

4.3.5 Summary

- Use **`d3.axisBottom()`** and **`d3.axisLeft()`** with scales to create X and Y axes.
- Customize ticks with `.ticks()` for density and `.tickFormat()` for labels (percentages, dates, decimals).
- Add axis labels by appending text elements inside axis groups.
- Enhance readability with gridlines using axis tick lines extended across the chart.
- Axes bring context to your visualizations, making data easier to interpret.

With this knowledge, you can create polished charts and tailor axes to fit any dataset or design style.

4.4 Handling Scale Updates

In real-world applications, data often changes over time—whether from live feeds, user input, or animations. To keep your visualization accurate and meaningful, your **scales and axes must update dynamically** to reflect new data.

This section shows how to smoothly update scale domains, redraw axes, and animate transitions for a responsive, polished user experience.

4.4.1 Why Update Scales and Axes?

When data values change (e.g., new data points arrive or existing data shifts), your scales' input domain may need adjustment to accommodate new min/max values. If the scale domain remains static, visual elements may clip or misrepresent the data.

Updating scales and axes ensures:

- The chart always fits the latest data.
- Axes reflect new ranges and labels.

- Transitions smooth the changes to reduce visual disruption.

4.4.2 Example: Dynamic Scatter Plot with Updating Scales and Axes

Let's simulate a simple dynamic dataset that updates every second, and update the scatter plot accordingly.

```
<svg width="600" height="300"></svg>
```

```
const svg = d3.select("svg");
const margin = { top: 20, right: 20, bottom: 50, left: 60 };
const width = +svg.attr("width") - margin.left - margin.right;
const height = +svg.attr("height") - margin.top - margin.bottom;

const chart = svg.append("g")
  .attr("transform", `translate(${margin.left},${margin.top})`);

let data = generateRandomData(20);

// Initial scales
let xScale = d3.scaleLinear()
  .domain(d3.extent(data, d => d.x))
  .range([0, width]);

let yScale = d3.scaleLinear()
  .domain(d3.extent(data, d => d.y))
  .range([height, 0]);

// Initial axes
let xAxis = chart.append("g")
  .attr("transform", `translate(0,${height})`)
  .call(d3.axisBottom(xScale));

let yAxis = chart.append("g")
  .call(d3.axisLeft(yScale));

// Plot circles
let circles = chart.selectAll("circle")
  .data(data)
  .enter()
  .append("circle")
  .attr("cx", d => xScale(d.x))
  .attr("cy", d => yScale(d.y))
  .attr("r", 6)
  .attr("fill", "teal");

// Function to update the chart with new data
function updateChart(newData) {
  // Update scales domain based on new data extents
  xScale.domain(d3.extent(newData, d => d.x));
  yScale.domain(d3.extent(newData, d => d.y));

  // Update axes with transition
  xAxis.transition()
    .duration(750)
    .call(d3.axisBottom(xScale));

  yAxis.transition()
    .duration(750)
    .call(d3.axisLeft(yScale));

  // Bind new data to circles
  circles = circles.data(newData);

  // Handle exiting points
  circles.exit()
    .transition()
    .duration(500)
    .attr("r", 0)
    .remove();

  // Handle updating existing points
  circles.transition()
    .duration(750)
    .attr("cx", d => xScale(d.x))
    .attr("cy", d => yScale(d.y));

  // Handle entering new points
  circles.enter()
    .append("circle")
    .attr("cx", d => xScale(d.x))
    .attr("cy", d => yScale(d.y))
    .attr("r", 0)
    .attr("fill", "teal")
    .transition()
    .duration(750)
    .attr("r", 6);
}

// Simulate data updates every 2 seconds
setInterval(() => {
  data = generateRandomData(20);
  updateChart(data);
}, 2000);

// Helper: generate random data points within [0,100]
function generateRandomData(count) {
  return Array.from({ length: count }, () => ({
    x: Math.random() * 100,
    y: Math.random() * 100
  }));
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Dynamic Scatter Plot with Updating Axes</title>
<script src="https://d3js.org/d3.v7.min.js"></script>
<style>
```

```
  body {
    font-family: sans-serif;
  }
  svg {
    border: 1px solid #ccc;
  }
</style>
</head>
<body>

<svg width="600" height="300"></svg>

<script>
  const svg = d3.select("svg");
  const margin = { top: 20, right: 20, bottom: 50, left: 60 };
  const width = +svg.attr("width") - margin.left - margin.right;
  const height = +svg.attr("height") - margin.top - margin.bottom;

  const chart = svg.append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

  let data = generateRandomData(20);

  // Initial scales
  let xScale = d3.scaleLinear()
    .domain(d3.extent(data, d => d.x))
    .range([0, width]);

  let yScale = d3.scaleLinear()
    .domain(d3.extent(data, d => d.y))
    .range([height, 0]);

  // Initial axes
  let xAxis = chart.append("g")
    .attr("transform", `translate(0,${height})`)
    .call(d3.axisBottom(xScale));

  let yAxis = chart.append("g")
    .call(d3.axisLeft(yScale));

  // Plot circles
  let circles = chart.selectAll("circle")
    .data(data)
    .enter()
    .append("circle")
    .attr("cx", d => xScale(d.x))
    .attr("cy", d => yScale(d.y))
    .attr("r", 6)
    .attr("fill", "teal");

  // Function to update the chart with new data
  function updateChart(newData) {
    // Update scales domain based on new data extents
    xScale.domain(d3.extent(newData, d => d.x));
    yScale.domain(d3.extent(newData, d => d.y));

    // Update axes with transition
    xAxis.transition()
```

```
      .duration(750)
      .call(d3.axisBottom(xScale));

    yAxis.transition()
      .duration(750)
      .call(d3.axisLeft(yScale));

    // Bind new data to circles
    circles = chart.selectAll("circle").data(newData);

    // Handle exiting points
    circles.exit()
      .transition()
      .duration(500)
      .attr("r", 0)
      .remove();

    // Handle updating existing points
    circles.transition()
      .duration(750)
      .attr("cx", d => xScale(d.x))
      .attr("cy", d => yScale(d.y));

    // Handle entering new points
    circles.enter()
      .append("circle")
      .attr("cx", d => xScale(d.x))
      .attr("cy", d => yScale(d.y))
      .attr("r", 0)
      .attr("fill", "teal")
      .transition()
      .duration(750)
      .attr("r", 6);
  }

  // Simulate data updates every 2 seconds
  setInterval(() => {
    data = generateRandomData(20);
    updateChart(data);
  }, 2000);

  // Helper: generate random data points within [0,100]
  function generateRandomData(count) {
    return Array.from({ length: count }, () => ({
      x: Math.random() * 100,
      y: Math.random() * 100
    }));
  }
</script>

</body>
</html>
```

4.4.3 Key Takeaways

- Use `d3.extent(data, accessor)` to dynamically find min and max values for scale domains.
- Update scales' domains with `.domain(newDomain)` before redrawing axes.
- Use `.transition()` on axes and elements to animate updates smoothly.
- Manage **enter**, **update**, and **exit** selections properly to add, move, or remove elements as data changes.
- Keep margins and chart structure intact while updating content inside the chart group.

4.4.4 Responsive & Smooth Visuals

Transitions not only make updates visually appealing but also help users track data changes, preventing confusion. Combining scale updates with smooth animations elevates your chart's professionalism and usability.

4.4.5 Summary

- Dynamic data requires **scale domain updates** to accommodate new values.
- Axes are redrawn with updated scales, often with animated transitions.
- Data join's enter-update-exit pattern manages changing elements.
- Responsive updates improve clarity and maintain engagement in live or interactive charts.

With these techniques, your D3 visualizations will gracefully handle real-time data changes, providing fluid and accurate insights.

Chapter 5.
Working with Data

1. Loading External Data (CSV, JSON, TSV)
2. Parsing and Transforming Data
3. Nesting and Grouping with `d3.group`, `d3.rollup`
4. Data-Driven Thinking

5 Working with Data

5.1 Loading External Data (CSV, JSON, TSV)

Real-world data rarely lives inside your code—it's often stored externally in files like CSV, JSON, or TSV. D3 simplifies loading these files asynchronously with built-in functions, allowing you to fetch, parse, and visualize data smoothly.

5.1.1 Loading Data Asynchronously with D3

D3 provides convenient functions to load common file formats:

- **`d3.csv(url[, row])`** — Loads CSV (comma-separated values) files.
- **`d3.tsv(url[, row])`** — Loads TSV (tab-separated values) files.
- **`d3.json(url[, row])`** — Loads JSON files.

These functions return **Promises** that resolve to the parsed data, enabling you to handle the data asynchronously using `.then()` or `async/await`.

5.1.2 How It Works

```
d3.csv("data.csv")
  .then(data => {
    // `data` is an array of objects representing rows
    console.log(data);
  })
  .catch(error => {
    console.error("Error loading the CSV file:", error);
  });
```

- The **Promise** resolves with the parsed data.
- You can optionally provide a **row conversion function** as the second argument to preprocess each row.
- Errors such as network failures or parsing issues are caught in `.catch()`.

5.1.3 Accessing Nested Properties

For JSON data with nested structures, you can access nested properties directly or map them during loading:

```
d3.json("data.json").then(data => {
  data.forEach(d => {
    console.log(d.user.name);  // Access nested property
  });
});
```

You can also use the row conversion function with JSON by mapping properties or flattening data if needed.

5.1.4 Example: Loading and Visualizing a CSV File

Suppose you have a CSV file named `fruits.csv` with contents:

```
name,color,sweetness
Apple,red,7
Banana,yellow,9
Cherry,red,8
Date,brown,6
```

Here's how to load and create a simple visualization showing each fruit's sweetness:

```
<svg width="400" height="150"></svg>
```

```
const svg = d3.select("svg");

d3.csv("fruits.csv", d => ({
  name: d.name,
  color: d.color,
  sweetness: +d.sweetness  // Convert sweetness from string to number
})).then(data => {
  const xScale = d3.scaleBand()
    .domain(data.map(d => d.name))
    .range([0, 400])
    .padding(0.3);

  const yScale = d3.scaleLinear()
    .domain([0, 10])  // sweetness scale from 0 to 10
    .range([150, 0]);

  // Draw bars representing sweetness
  svg.selectAll("rect")
    .data(data)
    .enter()
    .append("rect")
    .attr("x", d => xScale(d.name))
    .attr("y", d => yScale(d.sweetness))
    .attr("width", xScale.bandwidth())
    .attr("height", d => 150 - yScale(d.sweetness))
    .attr("fill", d => d.color);

}).catch(error => {
  console.error("Failed to load CSV data:", error);
});
```

5.1.5 Summary

- Use **d3.csv()**, **d3.tsv()**, and **d3.json()** to load data asynchronously, returning Promises.
- Optionally provide a **row conversion function** to parse and transform data types.
- Handle errors gracefully with `.catch()` to debug issues like file not found or malformed data.
- Access nested properties in JSON directly or via mapping.
- Combining loading with visualization code lets you build dynamic, data-driven charts seamlessly.

Mastering external data loading is the first step to unlocking D3's true power in real-world projects.

5.2 Parsing and Transforming Data

Raw data often isn't ready for visualization straight out of the file. Before you bind data to visual elements, you typically need to **parse, clean, and transform** it. This preparation step converts strings into numbers, parses dates, and reshapes the data into a structure tailored to your visualization's needs.

5.2.1 Common Data Preparation Tasks

Converting Strings to Numbers

CSV and JSON data frequently represent numeric values as strings. D3's loader functions allow you to **coerce these strings into numbers** during loading:

```
d3.csv("data.csv", d => ({
  name: d.name,
  value: +d.value   // The unary plus converts string to number
}));
```

Alternatively, use `parseFloat()` or `Number()` explicitly.

Parsing Dates with `d3.timeParse`

Dates in data files often appear as strings in various formats (e.g., `"2023-07-13"` or `"07/13/2023"`). Use D3's `d3.timeParse` to convert these strings into JavaScript `Date` objects:

```
const parseDate = d3.timeParse("%Y-%m-%d");

d3.csv("data.csv", d => ({
  date: parseDate(d.date),  // Convert "2023-07-13" → Date object
  value: +d.value
}));
```

The format string uses directives similar to `strftime` (e.g., `%Y` = 4-digit year, `%m` = zero-padded month, `%d` = zero-padded day).

Reshaping Arrays: `map`, `filter`, and More

Once data is loaded, you often need to **transform the dataset shape** or **filter out unwanted entries**.

- Use `.map()` to convert or restructure each data item.
- Use `.filter()` to exclude data points that don't meet certain criteria.

Example: Filter out entries with missing values, then compute a new property:

```
const cleanedData = data
  .filter(d => d.value !== null && !isNaN(d.value))
  .map(d => ({
    ...d,
    adjustedValue: d.value * 1.1   // Scale value by 10%
  }));
```

5.2.2 Example: Preparing a Dataset for Visualization

Suppose you load this CSV:

```
date,value
2023-07-01,100
2023-07-02,120
2023-07-03,
2023-07-04,90
```

You want to parse dates, convert values to numbers, and remove rows with missing data.

```
const parseDate = d3.timeParse("%Y-%m-%d");

d3.csv("data.csv", d => ({
  date: parseDate(d.date),
  value: d.value === "" ? null : +d.value
})).then(data => {
  const filteredData = data.filter(d => d.value !== null);

  console.log(filteredData);
  // Now ready for visualization
});
```

5.2.3 Summary

- Convert numeric strings to numbers early for accurate scales and calculations.
- Parse date strings into `Date` objects with `d3.timeParse` for time-based visualizations.
- Use array methods like `.map()` and `.filter()` to reshape and clean datasets.
- Data transformation bridges the gap between raw input and effective visual encoding.

By preparing your data carefully, you ensure smoother, more accurate, and meaningful visualizations downstream.

5.3 Nesting and Grouping with `d3.group`, `d3.rollup`

Grouping and aggregating data are common operations that help summarize and organize datasets by categories or keys. D3 provides powerful, flexible functions—`d3.group`, `d3.rollup`, and `d3.flatGroup`—to perform these tasks efficiently.

5.3.1 Understanding Grouping and Aggregation in D3

- **Grouping** means collecting data elements into subsets based on one or more keys (like category, date, region).
- **Aggregation** means computing summary statistics (such as sums, averages, counts) for each group.

D3's grouping functions simplify these processes, producing nested or flat structures suitable for visualization or further processing.

5.3.2 Key Functions

Function	Description	Output Structure
`d3.group(data, ...keys)`	Groups data by key(s), returns a nested `Map` of arrays	`Map` of key → array of original data objects
`d3.rollup(data, reduceFn, ...keys)`	Groups and aggregates data by key(s), returns a nested `Map` of reduced values	`Map` of key → aggregated value (e.g., sum, mean)
`d3.flatGroup(data, ...keys)`	Like `d3.group`, but returns a flat array of `[key(s), array]` pairs instead of nested Map	Array of `[key(s), array]`

5.3.3 `d3.group`: Group Data into Nested Maps

`d3.group` groups data by specified keys, collecting matching items into arrays:

```
const data = [
  { category: "Fruit", item: "Apple", sales: 100 },
  { category: "Fruit", item: "Banana", sales: 80 },
  { category: "Vegetable", item: "Carrot", sales: 50 },
  { category: "Fruit", item: "Cherry", sales: 120 },
  { category: "Vegetable", item: "Broccoli", sales: 70 },
];

const grouped = d3.group(data, d => d.category);

console.log(grouped);
```

Output: A `Map` where keys are categories (`"Fruit"`, `"Vegetable"`) and values are arrays of data objects in those categories.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>d3.group Example</title>
<script src="https://d3js.org/d3.v7.min.js"></script>
</head>
<body>
<script>
  const data = [
    { category: "Fruit", item: "Apple", sales: 100 },
    { category: "Fruit", item: "Banana", sales: 80 },
    { category: "Vegetable", item: "Carrot", sales: 50 },
    { category: "Fruit", item: "Cherry", sales: 120 },
    { category: "Vegetable", item: "Broccoli", sales: 70 },
  ];

  const grouped = d3.group(data, d => d.category);

  console.log("Grouped data:", grouped);

  // Optional: log each group with its items
  for (const [key, values] of grouped) {
    console.log(`Category: ${key}`);
    console.table(values);
  }
</script>
</body>
</html>
```

5.3.4 `d3.rollup`: Group and Aggregate Data

`d3.rollup` goes a step further by applying a **reducer function** to each group to compute aggregated values:

```
const salesByCategory = d3.rollup(
  data,
  v => d3.sum(v, d => d.sales), // sum sales in each group
  d => d.category
);

console.log(salesByCategory);
```

Output: A `Map` of category → total sales:

- `"Fruit"` → 300 (100 + 80 + 120)
- `"Vegetable"` → 120 (50 + 70)

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>d3.rollup Example</title>
<script src="https://d3js.org/d3.v7.min.js"></script>
</head>
<body>
<script>
  const data = [
    { category: "Fruit", item: "Apple", sales: 100 },
    { category: "Fruit", item: "Banana", sales: 80 },
    { category: "Vegetable", item: "Carrot", sales: 50 },
    { category: "Fruit", item: "Cherry", sales: 120 },
    { category: "Vegetable", item: "Broccoli", sales: 70 },
  ];

  // Use d3.rollup to sum sales by category
  const salesByCategory = d3.rollup(
    data,
    v => d3.sum(v, d => d.sales),
    d => d.category
  );

  console.log("Sales by category (Map):", salesByCategory);

  // Print out results clearly
  for (const [category, totalSales] of salesByCategory) {
    console.log(`${category}: ${totalSales}`);
  }
</script>
</body>
</html>
```

5.3.5 `d3.flatGroup`: Flat Grouping for Simpler Iteration

`d3.flatGroup` produces an array of `[key, groupArray]` pairs, rather than a nested Map. This can be useful when you want a simple iterable structure:

```
const flatGrouped = d3.flatGroup(data, d => d.category);

console.log(flatGrouped);
// [
//   ["Fruit", [...]],
//   ["Vegetable", [...]]
// ]
```

5.3.6 Practical Example: Group Sales by Category and Sum Values

```
const data = [
  { category: "Fruit", item: "Apple", sales: 100 },
  { category: "Fruit", item: "Banana", sales: 80 },
  { category: "Vegetable", item: "Carrot", sales: 50 },
  { category: "Fruit", item: "Cherry", sales: 120 },
  { category: "Vegetable", item: "Broccoli", sales: 70 },
];

// Group and sum sales per category
const salesByCategory = d3.rollup(
  data,
  group => d3.sum(group, d => d.sales),
  d => d.category
);

// Iterate and log
for (const [category, totalSales] of salesByCategory) {
  console.log(`${category}: ${totalSales}`);
}

// Output:
// Fruit: 300
// Vegetable: 120
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Group Sales by Category</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
</head>
<body>
<script>
  const data = [
    { category: "Fruit", item: "Apple", sales: 100 },
    { category: "Fruit", item: "Banana", sales: 80 },
    { category: "Vegetable", item: "Carrot", sales: 50 },
    { category: "Fruit", item: "Cherry", sales: 120 },
    { category: "Vegetable", item: "Broccoli", sales: 70 },
  ];

  // Group and sum sales per category
  const salesByCategory = d3.rollup(
    data,
    group => d3.sum(group, d => d.sales),
    d => d.category
  );

  // Iterate and log
  for (const [category, totalSales] of salesByCategory) {
    console.log(`${category}: ${totalSales}`);
  }
</script>
</body>
</html>
```

5.3.7 Summary

- **`d3.group`**: Groups data into nested Maps without aggregation.
- **`d3.rollup`**: Groups data and computes aggregate values per group.
- **`d3.flatGroup`**: Like `group`, but returns a flat array of `[key, values]` pairs.
- Use grouping and aggregation to prepare data summaries essential for grouped bar charts, stacked charts, or hierarchical visualizations.

Mastering these functions lets you transform raw data into structured insights, unlocking more complex and meaningful visualizations.

5.4 Data-Driven Thinking

One of D3's most powerful ideas—and what sets it apart—is the concept of **data-driven thinking**. Instead of manually manipulating individual DOM elements, you let your data *drive* what appears on the screen. Your visualization becomes a direct reflection of your dataset, automatically adjusting as your data changes.

5.4.1 What Is Data-Driven Thinking?

Think of your data as the *single source of truth.* The DOM elements in your chart or graph are simply a visual manifestation of that data. When the data changes, the visualization updates itself accordingly.

This mindset encourages you to:

- Focus on your data first.
- Define how each data item maps to visual elements.
- Let D3's enter-update-exit pattern handle the DOM updates automatically.

5.4.2 Enter-Update-Exit: The Heart of Data-Driven Visualization

The **enter-update-exit pattern** is a fundamental way to link data with the DOM:

- **Enter**: Create new elements for new data points.
- **Update**: Modify existing elements when data changes.
- **Exit**: Remove elements when data points are deleted.

This pattern makes your visualization *reactive*—whenever your data changes, the chart updates itself without manual DOM juggling.

5.4.3 Example: Updating a Bar Chart with New Data

```
// Initial data
let data = [30, 80, 45, 60];

// Function to render bars based on data
function render(data) {
  const bars = d3.select("svg")
    .selectAll("rect")
    .data(data);

  // ENTER: Create new bars for new data
  bars.enter()
    .append("rect")
    .attr("x", (d, i) => i * 30)
    .attr("y", d => 100 - d)
    .attr("width", 25)
    .attr("height", d => d)
    .attr("fill", "steelblue")

  // UPDATE: Update existing bars to new heights
  bars
    .attr("y", d => 100 - d)
    .attr("height", d => d);

  // EXIT: Remove bars that no longer have data
  bars.exit().remove();
}

// Initial render
```

```
render(data);

// Later: data changes
setTimeout(() => {
  data = [20, 90, 50];  // Changed data length and values
  render(data);         // Visualization updates automatically
}, 2000);
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Bar Chart Update Example</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    svg {
      border: 1px solid #ccc;
    }
  </style>
</head>
<body>

<svg width="200" height="120"></svg>

<script>
  // Initial data
  let data = [30, 80, 45, 60];

  // Function to render bars based on data
  function render(data) {
    const bars = d3.select("svg")
      .selectAll("rect")
      .data(data);

    // ENTER: Create new bars for new data
    bars.enter()
      .append("rect")
      .attr("x", (d, i) => i * 30)
      .attr("y", d => 100 - d)
      .attr("width", 25)
      .attr("height", d => d)
      .attr("fill", "steelblue");

    // UPDATE: Update existing bars to new heights
    bars
      .attr("y", d => 100 - d)
      .attr("height", d => d);

    // EXIT: Remove bars that no longer have data
    bars.exit().remove();
  }

  // Initial render
  render(data);
  // Later: data changes
  setTimeout(() => {
    data = [20, 90, 50];  // Changed data length and values
    render(data);         // Visualization updates automatically
  }, 2000);
</script>

</body>
</html>
```

5.4.4 Why This Matters

- **Declarative**: You declare what data corresponds to, rather than how to manipulate elements.
- **Maintainable**: Your code becomes easier to reason about and less error-prone.
- **Dynamic**: Real-time data changes or user interactions flow naturally into visual updates.
- **Expressive**: The structure of your data directly informs the structure of your visualization.

5.4.5 Summary

- Treat your dataset as the *driver* of your visualization's state.
- Use the enter-update-exit pattern to reflect data changes smoothly in the DOM.
- Focus on the *what* (data) instead of the *how* (DOM manipulation).
- Data-driven thinking leads to cleaner, more powerful, and adaptable visualizations.

Embracing this mindset transforms how you build visuals—your charts become living, breathing representations of your data.

Chapter 6.
Building Core Chart Types

1. Bar Charts (Vertical and Horizontal)
2. Line Charts and Area Charts
3. Scatter Plots
4. Pie and Donut Charts
5. Histogram and Boxplots

6 Building Core Chart Types

6.1 Bar Charts (Vertical and Horizontal)

Bar charts are one of the most fundamental and widely used visualization types. They help compare quantities across categories using rectangular bars, making data easy to interpret at a glance. In this section, we'll build a **vertical bar chart** from scratch with D3, then extend it to a **horizontal bar chart**. We'll also add responsive resizing and tooltips for interactivity.

6.1.1 Building a Basic Vertical Bar Chart

Step 1: Setup SVG and Data

```
<svg width="600" height="400"></svg>
<div id="tooltip" style="position:absolute; padding:6px; background:#333; color:#fff; border-radius:4px
```

```
const svg = d3.select("svg");
const tooltip = d3.select("#tooltip");

const margin = { top: 30, right: 20, bottom: 50, left: 60 };
const width = +svg.attr("width") - margin.left - margin.right;
const height = +svg.attr("height") - margin.top - margin.bottom;

const data = [
  { category: "Apples", value: 30 },
  { category: "Bananas", value: 80 },
  { category: "Cherries", value: 45 },
  { category: "Dates", value: 60 },
  { category: "Elderberries", value: 20 }
];

const chart = svg.append("g")
  .attr("transform", `translate(${margin.left},${margin.top})`);
```

Step 2: Define Scales

```
const xScale = d3.scaleBand()
  .domain(data.map(d => d.category))
  .range([0, width])
  .padding(0.2);

const yScale = d3.scaleLinear()
  .domain([0, d3.max(data, d => d.value)])
  .range([height, 0]);
```

- `scaleBand` maps categories to discrete bands along X.
- `scaleLinear` maps values to pixel heights along Y (inverted since SVG y=0 is top).

Step 3: Draw Axes

```
const xAxis = d3.axisBottom(xScale);
const yAxis = d3.axisLeft(yScale);

chart.append("g")
  .attr("transform", `translate(0,${height})`)
  .call(xAxis);

chart.append("g")
  .call(yAxis);
```

Step 4: Draw Bars with Interactivity

```
chart.selectAll("rect")
  .data(data)
  .enter()
  .append("rect")
  .attr("x", d => xScale(d.category))
  .attr("y", d => yScale(d.value))
  .attr("width", xScale.bandwidth())
  .attr("height", d => height - yScale(d.value))
  .attr("fill", "steelblue")
  .on("mousemove", (event, d) => {
    tooltip.style("opacity", 1)
      .html(`<strong>${d.category}</strong>: ${d.value}`)
      .style("left", (event.pageX + 10) + "px")
      .style("top", (event.pageY - 28) + "px");
  })
  .on("mouseout", () => {
    tooltip.style("opacity", 0);
  });
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Basic Vertical Bar Chart with Tooltip</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    svg {
      border: 1px solid #ccc;
      font-family: sans-serif;
    }
    #tooltip {
      position: absolute;
      padding: 6px 10px;
      background: rgba(51, 51, 51, 0.9);
      color: white;
      border-radius: 4px;
      pointer-events: none;
      opacity: 0;
      transition: opacity 0.2s;
      font-size: 12px;
    }
  </style>
</head>
<body>

<svg width="600" height="400"></svg>
<div id="tooltip"></div>

<script>
  const svg = d3.select("svg");
  const tooltip = d3.select("#tooltip");

  const margin = { top: 30, right: 20, bottom: 50, left: 60 };
  const width = +svg.attr("width") - margin.left - margin.right;
  const height = +svg.attr("height") - margin.top - margin.bottom;

  const data = [
    { category: "Apples", value: 30 },
    { category: "Bananas", value: 80 },
    { category: "Cherries", value: 45 },
    { category: "Dates", value: 60 },
    { category: "Elderberries", value: 20 }
  ];

  const chart = svg.append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

  const xScale = d3.scaleBand()
    .domain(data.map(d => d.category))
    .range([0, width])
    .padding(0.2);

  const yScale = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.value)])
    .range([height, 0]);

  const xAxis = d3.axisBottom(xScale);
  const yAxis = d3.axisLeft(yScale);

  chart.append("g")
    .attr("transform", `translate(0,${height})`)
    .call(xAxis);

  chart.append("g")
    .call(yAxis);

  chart.selectAll("rect")
    .data(data)
    .enter()
    .append("rect")
    .attr("x", d => xScale(d.category))
    .attr("y", d => yScale(d.value))
    .attr("width", xScale.bandwidth())
    .attr("height", d => height - yScale(d.value))
    .attr("fill", "steelblue")
    .on("mousemove", (event, d) => {
      tooltip.style("opacity", 1)
        .html(`<strong>${d.category}</strong>: ${d.value}`)
        .style("left", (event.pageX + 10) + "px")
        .style("top", (event.pageY - 28) + "px");
    })
    .on("mouseout", () => {
      tooltip.style("opacity", 0);
    });
</script>

</body>
</html>
```

6.1.2 Creating a Horizontal Bar Chart

Switching to horizontal bars requires swapping the roles of scales and axes:

- Use `scaleBand` on the **Y-axis** for categories.
- Use `scaleLinear` on the **X-axis** for values.
- Rotate axes accordingly.

```
// New scales for horizontal chart
const yScaleH = d3.scaleBand()
  .domain(data.map(d => d.category))
  .range([0, height])
  .padding(0.2);

const xScaleH = d3.scaleLinear()
  .domain([0, d3.max(data, d => d.value)])
  .range([0, width]);

// Clear previous chart group
chart.selectAll("*").remove();

// Draw horizontal bars
chart.selectAll("rect")
  .data(data)
  .enter()
  .append("rect")
  .attr("y", d => yScaleH(d.category))
  .attr("x", 0)
  .attr("height", yScaleH.bandwidth())
  .attr("width", d => xScaleH(d.value))
  .attr("fill", "tomato")
  .on("mousemove", (event, d) => {
    tooltip.style("opacity", 1)
      .html(`<strong>${d.category}</strong>: ${d.value}`)
      .style("left", (event.pageX + 10) + "px")
      .style("top", (event.pageY - 28) + "px");
  })
  .on("mouseout", () => {
    tooltip.style("opacity", 0);
  });

// Draw axes
chart.append("g")
```

```
  .call(d3.axisLeft(yScaleH));

chart.append("g")
  .attr("transform", `translate(0,${height})`)
  .call(d3.axisBottom(xScaleH));
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Horizontal Bar Chart with D3</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    svg {
      border: 1px solid #ccc;
      font-family: sans-serif;
    }
    #tooltip {
      position: absolute;
      padding: 6px 10px;
      background: rgba(51, 51, 51, 0.9);
      color: white;
      border-radius: 4px;
      pointer-events: none;
      opacity: 0;
      transition: opacity 0.2s;
      font-size: 12px;
    }
  </style>
</head>
<body>

<svg width="600" height="400"></svg>
<div id="tooltip"></div>

<script>
  const svg = d3.select("svg");
  const tooltip = d3.select("#tooltip");

  const margin = { top: 30, right: 20, bottom: 50, left: 100 };
  const width = +svg.attr("width") - margin.left - margin.right;
  const height = +svg.attr("height") - margin.top - margin.bottom;

  const data = [
    { category: "Apples", value: 30 },
    { category: "Bananas", value: 80 },
    { category: "Cherries", value: 45 },
    { category: "Dates", value: 60 },
    { category: "Elderberries", value: 20 }
  ];

  const chart = svg.append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

  // Scales for horizontal chart
  const yScaleH = d3.scaleBand()
    .domain(data.map(d => d.category))
    .range([0, height])
    .padding(0.2);

  const xScaleH = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.value)])
    .range([0, width]);

  // Draw horizontal bars
  chart.selectAll("rect")
    .data(data)
    .enter()
    .append("rect")
    .attr("y", d => yScaleH(d.category))
    .attr("x", 0)
    .attr("height", yScaleH.bandwidth())
    .attr("width", d => xScaleH(d.value))
    .attr("fill", "tomato")
    .on("mousemove", (event, d) => {
      tooltip.style("opacity", 1)
        .html(`<strong>${d.category}</strong>: ${d.value}`)
        .style("left", (event.pageX + 10) + "px")
        .style("top", (event.pageY - 28) + "px");
    })
    .on("mouseout", () => {
      tooltip.style("opacity", 0);
    });

  // Draw axes
  chart.append("g")
    .call(d3.axisLeft(yScaleH));

  chart.append("g")
    .attr("transform", `translate(0,${height})`)
    .call(d3.axisBottom(xScaleH));
</script>

</body>
</html>
```

6.1.3 Adding Responsiveness

Make your chart resize gracefully by recalculating width and height on window resize and updating scales, axes, and bars accordingly:

```
function renderChart() {
  const svgWidth = parseInt(svg.style("width"));
  const svgHeight = parseInt(svg.style("height"));
  const w = svgWidth - margin.left - margin.right;
  const h = svgHeight - margin.top - margin.bottom;

  xScale.range([0, w]);
  yScale.range([h, 0]);
  chart.attr("transform", `translate(${margin.left},${margin.top})`);

  // Update axes and bars with new scales...
  // (Similar to above code but updating attributes and calling axes)
}

window.addEventListener("resize", renderChart);
renderChart();
```

For simplicity, use CSS to set SVG width to 100% and control height via JS or CSS.

6.1.4 Summary

- Build vertical bar charts using `rect` elements with `scaleBand` for categories and `scaleLinear` for values.
- Switch to horizontal bars by swapping scales and axes orientation.
- Add **tooltips** using mouse events to enhance interactivity.
- Support **responsiveness** by updating scales and re-rendering on resize.
- This foundational chart type forms the basis for many complex visualizations.

With these building blocks, you're ready to create clear, engaging bar charts tailored to your data's story.

6.2 Line Charts and Area Charts

Line and area charts are ideal for showing trends and changes over continuous data, such as time series. D3 provides powerful generator functions — `d3.line()` and `d3.area()` — to create smooth, data-driven paths easily. In this section, we'll explore how to use these generators to build both line and area charts.

6.2.1 Using the `d3.line()` Generator

The `d3.line()` function creates a path string (`d` attribute) for SVG `<path>` elements based on your data points.

Basic Usage

1. Define scales for your x and y dimensions (e.g., time on the x-axis, value on the y-axis).
2. Create a line generator specifying how to extract x and y coordinates from data.
3. Append an SVG `<path>` and set its `"d"` attribute to the output of the line generator.

Example: Time-Series Line Chart

```
const data = [
  { date: new Date(2023, 0, 1), value: 30 },
  { date: new Date(2023, 0, 2), value: 80 },
  { date: new Date(2023, 0, 3), value: 45 },
  { date: new Date(2023, 0, 4), value: 60 },
  { date: new Date(2023, 0, 5), value: 20 }
];

const margin = { top: 30, right: 30, bottom: 30, left: 50 };
const width = 600 - margin.left - margin.right;
const height = 300 - margin.top - margin.bottom;

const svg = d3.select("svg")
  .attr("width", width + margin.left + margin.right)
  .attr("height", height + margin.top + margin.bottom);

const chart = svg.append("g")
  .attr("transform", `translate(${margin.left},${margin.top})`);

const xScale = d3.scaleTime()
  .domain(d3.extent(data, d => d.date))
  .range([0, width]);

const yScale = d3.scaleLinear()
  .domain([0, d3.max(data, d => d.value)])
  .range([height, 0]);

const lineGenerator = d3.line()
  .x(d => xScale(d.date))
  .y(d => yScale(d.value))
  .curve(d3.curveMonotoneX);  // smooths the line

chart.append("path")
  .datum(data)
  .attr("fill", "none")
  .attr("stroke", "steelblue")
  .attr("stroke-width", 2)
  .attr("d", lineGenerator);

// Axes (optional)
chart.append("g")
  .attr("transform", `translate(0,${height})`)
  .call(d3.axisBottom(xScale));
chart.append("g")
  .call(d3.axisLeft(yScale));
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Time-Series Line Chart</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    svg {
      border: 1px solid #ccc;
      font-family: sans-serif;
    }
  </style>
</head>
<body>

<svg></svg>

<script>
  const data = [
    { date: new Date(2023, 0, 1), value: 30 },
    { date: new Date(2023, 0, 2), value: 80 },
    { date: new Date(2023, 0, 3), value: 45 },
    { date: new Date(2023, 0, 4), value: 60 },
    { date: new Date(2023, 0, 5), value: 20 }
  ];

  const margin = { top: 30, right: 30, bottom: 30, left: 50 };
  const width = 600 - margin.left - margin.right;
  const height = 300 - margin.top - margin.bottom;

  const svg = d3.select("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom);

  const chart = svg.append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

  const xScale = d3.scaleTime()
    .domain(d3.extent(data, d => d.date))
    .range([0, width]);

  const yScale = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.value)])
    .range([height, 0]);

  const lineGenerator = d3.line()
    .x(d => xScale(d.date))
    .y(d => yScale(d.value))
    .curve(d3.curveMonotoneX);  // smooth curve

  chart.append("path")
    .datum(data)
    .attr("fill", "none")
    .attr("stroke", "steelblue")
    .attr("stroke-width", 2)
    .attr("d", lineGenerator);

  // Axes
  chart.append("g")
    .attr("transform", `translate(0,${height})`)
    .call(d3.axisBottom(xScale).ticks(5).tickFormat(d3.timeFormat("%b %d")));

  chart.append("g")
    .call(d3.axisLeft(yScale));
</script>
```

```
</body>
</html>
```

6.2.2 Extending to Area Charts with `d3.area()`

The `d3.area()` generator builds on the line generator by filling the area between two y-values (`y0` and `y1`) across the x-axis.

- `y0` defines the baseline (often the chart's bottom or zero line).
- `y1` defines the top line, typically the data value.

Example: Basic Area Chart

```
const areaGenerator = d3.area()
  .x(d => xScale(d.date))
  .y0(height)           // baseline at bottom of chart
  .y1(d => yScale(d.value))
  .curve(d3.curveMonotoneX);

chart.append("path")
  .datum(data)
  .attr("fill", "lightsteelblue")
  .attr("stroke", "steelblue")
  .attr("stroke-width", 1.5)
  .attr("d", areaGenerator);
```

This fills the area between the x-axis baseline (`y0 = height`) and the line defined by your data (`y1 = yScale(d.value)`).

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Basic Area Chart</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    svg {
      border: 1px solid #ccc;
      font-family: sans-serif;
    }
  </style>
</head>
<body>

<svg></svg>

<script>
  const data = [
    { date: new Date(2023, 0, 1), value: 30 },
    { date: new Date(2023, 0, 2), value: 80 },
    { date: new Date(2023, 0, 3), value: 45 },
    { date: new Date(2023, 0, 4), value: 60 },
    { date: new Date(2023, 0, 5), value: 20 }
  ];

  const margin = { top: 30, right: 30, bottom: 30, left: 50 };
  const width = 600 - margin.left - margin.right;
  const height = 300 - margin.top - margin.bottom;

  const svg = d3.select("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom);

  const chart = svg.append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

  const xScale = d3.scaleTime()
    .domain(d3.extent(data, d => d.date))
    .range([0, width]);

  const yScale = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.value)])
    .range([height, 0]);

  // Area generator
  const areaGenerator = d3.area()
    .x(d => xScale(d.date))
    .y0(height) // bottom of the chart
    .y1(d => yScale(d.value))
    .curve(d3.curveMonotoneX);

  // Draw area
  chart.append("path")
    .datum(data)
    .attr("fill", "lightsteelblue")
    .attr("stroke", "steelblue")
    .attr("stroke-width", 1.5)
    .attr("d", areaGenerator);

  // Axes
  chart.append("g")
    .attr("transform", `translate(0,${height})`)
    .call(d3.axisBottom(xScale).ticks(5).tickFormat(d3.timeFormat("%b %d")));

  chart.append("g")
    .call(d3.axisLeft(yScale));
</script>

</body>
</html>
```

6.2.3 Key Takeaways

- **`d3.line()`** converts data points into a smooth SVG path representing a line.
- **`d3.area()`** creates a filled area between two y-values across the x-axis.
- Both accept `x` and `y` accessors and support curve interpolation for smooth visuals.
- Use `y0` and `y1` in `d3.area()` to define the vertical bounds of the area.
- Line and area charts are ideal for showing trends, cumulative totals, or variability over continuous domains like time.

With these generators, creating elegant line and area charts in D3 becomes straightforward and customizable to your data story.

6.3 Scatter Plots

Scatter plots are a fundamental visualization to explore relationships between two continuous variables. By plotting individual data points on an x-y coordinate system, you can visually detect correlations, clusters, or outliers. In this section, we'll build a scatter plot and see how to enhance it by encoding extra data dimensions using color and size.

6.3.1 Building a Basic Scatter Plot

Given a dataset with two quantitative variables (e.g., height and weight), each point is positioned on the chart according to its x and y values.

Step 1: Sample Data

```
const data = [
  { height: 170, weight: 65, gender: "Male" },
  { height: 160, weight: 55, gender: "Female" },
  { height: 180, weight: 75, gender: "Male" },
  { height: 155, weight: 50, gender: "Female" },
  { height: 165, weight: 60, gender: "Female" },
  { height: 175, weight: 70, gender: "Male" }
];
```

Step 2: Setup SVG and Scales

```
const margin = { top: 30, right: 30, bottom: 50, left: 50 };
const width = 600 - margin.left - margin.right;
const height = 400 - margin.top - margin.bottom;

const svg = d3.select("svg")
  .attr("width", width + margin.left + margin.right)
  .attr("height", height + margin.top + margin.bottom);
```

```
const chart = svg.append("g")
  .attr("transform", `translate(${margin.left},${margin.top})`);

const xScale = d3.scaleLinear()
  .domain([140, d3.max(data, d => d.height) + 10])  // padding domain
  .range([0, width]);

const yScale = d3.scaleLinear()
  .domain([40, d3.max(data, d => d.weight) + 10])
  .range([height, 0]);
```

Step 3: Define Color and Size Scales

We can use **color** to encode gender and **size** to encode weight or another variable:

```
const colorScale = d3.scaleOrdinal()
  .domain(["Male", "Female"])
  .range(["steelblue", "tomato"]);

const sizeScale = d3.scaleLinear()
  .domain(d3.extent(data, d => d.weight))
  .range([5, 15]);
```

Step 4: Draw Axes

```
chart.append("g")
  .attr("transform", `translate(0,${height})`)
  .call(d3.axisBottom(xScale).ticks(6));

chart.append("g")
  .call(d3.axisLeft(yScale));
```

Step 5: Plot the Points

```
chart.selectAll("circle")
  .data(data)
  .enter()
  .append("circle")
  .attr("cx", d => xScale(d.height))
  .attr("cy", d => yScale(d.weight))
  .attr("r", d => sizeScale(d.weight))
  .attr("fill", d => colorScale(d.gender))
  .attr("opacity", 0.7);
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Basic Scatter Plot</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
    }
    svg {
      border: 1px solid #ccc;
    }
  </style>
</head>
<body>

<svg></svg>

<script>
  // Step 1: Sample Data
  const data = [
    { height: 170, weight: 65, gender: "Male" },
    { height: 160, weight: 55, gender: "Female" },
    { height: 180, weight: 75, gender: "Male" },
    { height: 155, weight: 50, gender: "Female" },
    { height: 165, weight: 60, gender: "Female" },
    { height: 175, weight: 70, gender: "Male" }
  ];

  // Step 2: Setup SVG and Scales
  const margin = { top: 30, right: 30, bottom: 50, left: 50 };
  const width = 600 - margin.left - margin.right;
  const height = 400 - margin.top - margin.bottom;

  const svg = d3.select("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom);

  const chart = svg.append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

  const xScale = d3.scaleLinear()
    .domain([140, d3.max(data, d => d.height) + 10])
    .range([0, width]);

  const yScale = d3.scaleLinear()
    .domain([40, d3.max(data, d => d.weight) + 10])
    .range([height, 0]);

  // Step 3: Define Color and Size Scales
  const colorScale = d3.scaleOrdinal()
    .domain(["Male", "Female"])
    .range(["steelblue", "tomato"]);

  const sizeScale = d3.scaleLinear()
    .domain(d3.extent(data, d => d.weight))
    .range([5, 15]);

  // Step 4: Draw Axes
  chart.append("g")
    .attr("transform", `translate(0,${height})`)
    .call(d3.axisBottom(xScale).ticks(6))
    .append("text")
    .attr("x", width / 2)
    .attr("y", 40)
    .attr("fill", "black")
    .attr("text-anchor", "middle")
    .text("Height (cm)");

  chart.append("g")
    .call(d3.axisLeft(yScale))
    .append("text")
    .attr("transform", "rotate(-90)")
    .attr("x", -height / 2)
    .attr("y", -40)
    .attr("fill", "black")
    .attr("text-anchor", "middle")
    .text("Weight (kg)");

  // Step 5: Plot the Points
  chart.selectAll("circle")
    .data(data)
    .enter()
    .append("circle")
    .attr("cx", d => xScale(d.height))
    .attr("cy", d => yScale(d.weight))
    .attr("r", d => sizeScale(d.weight))
    .attr("fill", d => colorScale(d.gender))
    .attr("opacity", 0.7);
</script>

</body>
</html>
```

6.3.2 Enhancing Interpretation

- **Color encoding** lets viewers quickly identify categories (e.g., gender).
- **Size encoding** adds a third dimension, highlighting differences in magnitude (e.g., heavier individuals have bigger circles).
- You can further enhance with **tooltips**, **labels**, or **legends** to explain encodings.

6.3.3 What You See

- Each circle corresponds to a person's height and weight.
- Blue circles represent males, red circles females.
- Larger circles indicate greater weight.
- The scatter pattern reveals any visible relationship between height and weight and differences between genders.

6.3.4 Summary

- Scatter plots map two quantitative variables onto x and y axes using points.
- Additional variables can be encoded with color, size, shape, or opacity.
- D3's flexible scales and data binding make building scatter plots straightforward.
- This chart is an excellent starting point for exploring correlations and groupings.

Scatter plots provide a rich canvas for multidimensional data visualization, letting you tell stories that go beyond simple x-y comparisons.

6.4 Pie and Donut Charts

Pie charts and donut charts are popular ways to visualize proportions of categories within a whole. Using D3's `d3.pie()` and `d3.arc()` functions, you can generate slices (arcs) that represent each category's share, with flexibility to customize appearance and behavior. This section guides you through creating both pie and donut charts, complete with labels and simple animation.

Example input data:

```
const data = [
  { category: "Apples", value: 30 },
  { category: "Bananas", value: 70 },
  { category: "Cherries", value: 45 },
  { category: "Dates", value: 65 }
];
```

6.4.1 The `d3.pie()` Function: Computing Slice Angles

The `d3.pie()` function processes your categorical data and returns an array of objects describing each slice's start and end angles based on the values. It abstracts away the math needed to convert raw values into angular slices.

6.4.2 The `d3.arc()` Function: Generating SVG Path Strings

`d3.arc()` generates the `d` attribute for SVG `<path>` elements representing arcs or slices. You specify inner and outer radii and angles, and it creates the corresponding arc shape.

- For a **pie chart**, inner radius = 0 (a solid circle slice).
- For a **donut chart**, inner radius > 0 (creates a hollow center).

6.4.3 Building a Basic Pie Chart

```
const width = 400;
const height = 400;
const radius = Math.min(width, height) / 2;

const svg = d3.select("svg")
  .attr("width", width)
  .attr("height", height)
  .append("g")
  .attr("transform", `translate(${width / 2},${height / 2})`);

const pie = d3.pie()
  .value(d => d.value);

const arcs = pie(data);

const arcGenerator = d3.arc()
  .innerRadius(0)  // Pie chart: no hole
  .outerRadius(radius - 10);

const color = d3.scaleOrdinal()
  .domain(data.map(d => d.category))
  .range(d3.schemeCategory10);

const path = svg.selectAll("path")
  .data(arcs)
  .enter()
  .append("path")
  .attr("fill", d => color(d.data.category))
  .attr("d", arcGenerator)
  .each(function(d) { this._current = d; }); // store initial state for animation
```

6.4.4 Adding Labels

Place category labels at the centroid of each slice:

```
svg.selectAll("text")
  .data(arcs)
  .enter()
  .append("text")
  .attr("transform", d => `translate(${arcGenerator.centroid(d)})`)
  .attr("text-anchor", "middle")
  .attr("alignment-baseline", "middle")
  .style("font-size", "12px")
  .text(d => d.data.category);
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Basic Pie Chart with D3</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
    }
    svg {
      display: block;
      margin: auto;
    }
  </style>
</head>
<body>

<svg></svg>

<script>
  // Input data
  const data = [
    { category: "Apples", value: 30 },
    { category: "Bananas", value: 70 },
    { category: "Cherries", value: 45 },
    { category: "Dates", value: 65 }
  ];

  // Dimensions
  const width = 400;
  const height = 400;
  const radius = Math.min(width, height) / 2;

  // SVG setup
  const svg = d3.select("svg")
    .attr("width", width)
    .attr("height", height)
    .append("g")
    .attr("transform", `translate(${width / 2},${height / 2})`);

  // Pie layout
  const pie = d3.pie()
    .value(d => d.value);

  const arcs = pie(data);

  // Arc generator
  const arcGenerator = d3.arc()
    .innerRadius(0)  // For pie chart, not donut
    .outerRadius(radius - 10);

  // Color scale
  const color = d3.scaleOrdinal()
    .domain(data.map(d => d.category))
    .range(d3.schemeCategory10);

  // Draw slices
  svg.selectAll("path")
    .data(arcs)
    .enter()
    .append("path")
    .attr("fill", d => color(d.data.category))
    .attr("d", arcGenerator)
    .attr("stroke", "white")
    .attr("stroke-width", 2);

  // Add labels
  svg.selectAll("text")
    .data(arcs)
    .enter()
    .append("text")
    .attr("transform", d => `translate(${arcGenerator.centroid(d)})`)
    .attr("text-anchor", "middle")
    .attr("alignment-baseline", "middle")
    .style("font-size", "12px")
    .text(d => d.data.category);
</script>

</body>
</html>
```

6.4.5 Turning the Pie into a Donut

Simply set a positive **innerRadius** in the arc generator:

```
const arcGeneratorDonut = d3.arc()
  .innerRadius(radius / 2)  // creates the hole
  .outerRadius(radius - 10);

svg.selectAll("path")
  .data(arcs)
  .join("path")
  .attr("fill", d => color(d.data.category))
  .attr("d", arcGeneratorDonut);
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Donut Chart with D3</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
    }
    svg {
      display: block;
      margin: auto;
    }
  </style>
</head>
<body>
```

```
<svg></svg>

<script>
  const data = [
    { category: "Apples", value: 30 },
    { category: "Bananas", value: 70 },
    { category: "Cherries", value: 45 },
    { category: "Dates", value: 65 }
  ];

  const width = 400;
  const height = 400;
  const radius = Math.min(width, height) / 2;

  const svg = d3.select("svg")
    .attr("width", width)
    .attr("height", height)
    .append("g")
    .attr("transform", `translate(${width / 2},${height / 2})`);

  const pie = d3.pie()
    .value(d => d.value);

  const arcs = pie(data);

  const arcGeneratorDonut = d3.arc()
    .innerRadius(radius / 2)     // Creates the hole
    .outerRadius(radius - 10);   // Outer edge of donut

  const color = d3.scaleOrdinal()
    .domain(data.map(d => d.category))
    .range(d3.schemeCategory10);

  svg.selectAll("path")
    .data(arcs)
    .enter()
    .append("path")
    .attr("fill", d => color(d.data.category))
    .attr("d", arcGeneratorDonut)
    .attr("stroke", "white")
    .attr("stroke-width", 2);

  svg.selectAll("text")
    .data(arcs)
    .enter()
    .append("text")
    .attr("transform", d => `translate(${arcGeneratorDonut.centroid(d)})`)
    .attr("text-anchor", "middle")
    .attr("alignment-baseline", "middle")
    .style("font-size", "12px")
    .text(d => d.data.category);
</script>

</body>
</html>
```

6.4.6 Adding Basic Animation on Render

Animate the pie slices on initial render using `d3.interpolate`:

```
path.transition()
  .duration(1000)
  .attrTween("d", function(d) {
    const interpolate = d3.interpolate(
      { startAngle: 0, endAngle: 0 }, d
    );
    return t => arcGenerator(interpolate(t));
  });
```

This smoothly "grows" each slice from 0 degrees to its final angle.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Animated Pie Chart</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
    }
    svg {
      display: block;
      margin: auto;
    }
  </style>
</head>
<body>

<svg></svg>

<script>
  const data = [
    { category: "Apples", value: 30 },
    { category: "Bananas", value: 70 },
    { category: "Cherries", value: 45 },
    { category: "Dates", value: 65 }
  ];

  const width = 400;
  const height = 400;
  const radius = Math.min(width, height) / 2;

  const svg = d3.select("svg")
    .attr("width", width)
    .attr("height", height)
    .append("g")
    .attr("transform", `translate(${width / 2},${height / 2})`);

  const pie = d3.pie()
    .value(d => d.value);
  const arcs = pie(data);

  const arcGenerator = d3.arc()
    .innerRadius(0)
    .outerRadius(radius - 10);

  const color = d3.scaleOrdinal()
    .domain(data.map(d => d.category))
    .range(d3.schemeCategory10);

  svg.selectAll("path")
    .data(arcs)
    .enter()
    .append("path")
    .attr("fill", d => color(d.data.category))
    .transition()
    .duration(1000)
    .attrTween("d", function(d) {
      const i = d3.interpolate(
        { startAngle: 0, endAngle: 0 },
        d
      );
      return t => arcGenerator(i(t));
    });

  svg.selectAll("text")
    .data(arcs)
    .enter()
    .append("text")
    .attr("transform", d => `translate(${arcGenerator.centroid(d)})`)
    .attr("text-anchor", "middle")
    .attr("alignment-baseline", "middle")
    .style("font-size", "12px")
    .text(d => d.data.category);
</script>

</body>
</html>
```

6.4.7 Summary

- Use `d3.pie()` to calculate slice angles from categorical data values.
- Use `d3.arc()` to create SVG paths for pie or donut slices.
- Adjust `innerRadius` in `d3.arc()` to switch between pie (solid) and donut (hollow) charts.
- Add labels using arc centroids for clarity.
- Animate slice appearance with transitions for engaging visuals.

Pie and donut charts are great for presenting part-to-whole relationships in an intuitive and visually appealing way—D3 gives you the control to make them interactive and polished.

6.5 Histogram and Boxplots

Histograms and boxplots are essential tools for understanding the distribution and spread of your data. Histograms group data into buckets to show frequency counts, while boxplots summarize data with key statistics, highlighting central tendency and variability. In this section, we'll learn how to build both using D3.

6.5.1 Histograms: Grouping Data into Buckets

A **histogram** visualizes the distribution of a numerical dataset by dividing the range into intervals (bins) and showing how many data points fall into each bin.

Using `d3.bin()` to Create Histogram Bins

D3's `d3.bin()` function automatically groups raw data into bins.

```
const data = [4, 8, 15, 16, 23, 42, 9, 12, 25, 18, 20, 22];
```

Example Dataset

```
const binGenerator = d3.bin()
  .domain([0, d3.max(data)])  // set range of data
  .thresholds(5);             // number of bins

const bins = binGenerator(data);
```

Step 1: Create the Bins Each bin is an array of values falling into that range and has properties like `x0` (start) and `x1` (end) of the bin.

Step 2: Visualize Histogram Bars

```
const width = 500;
const height = 300;
const margin = { top: 20, right: 30, bottom: 30, left: 40 };

const svg = d3.select("svg")
  .attr("width", width)
  .attr("height", height);

const xScale = d3.scaleLinear()
  .domain([0, d3.max(data)])
  .range([margin.left, width - margin.right]);

const yScale = d3.scaleLinear()
  .domain([0, d3.max(bins, d => d.length)])
  .range([height - margin.bottom, margin.top]);

svg.selectAll("rect")
```

```
  .data(bins)
  .enter()
  .append("rect")
  .attr("x", d => xScale(d.x0) + 1)
  .attr("y", d => yScale(d.length))
  .attr("width", d => xScale(d.x1) - xScale(d.x0) - 1)
  .attr("height", d => yScale(0) - yScale(d.length))
  .attr("fill", "steelblue");
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Histogram with d3.bin()</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
    }
    svg {
      display: block;
      margin: auto;
    }
  </style>
</head>
<body>

<svg></svg>

<script>
  const data = [4, 8, 15, 16, 23, 42, 9, 12, 25, 18, 20, 22];

  // Step 1: Generate bins
  const binGenerator = d3.bin()
    .domain([0, d3.max(data)])
    .thresholds(5);

  const bins = binGenerator(data);

  // Step 2: Setup SVG dimensions
  const width = 500;
  const height = 300;
  const margin = { top: 20, right: 30, bottom: 30, left: 40 };

  const svg = d3.select("svg")
    .attr("width", width)
    .attr("height", height);

  // Scales
  const xScale = d3.scaleLinear()
    .domain([0, d3.max(data)])
    .range([margin.left, width - margin.right]);

  const yScale = d3.scaleLinear()
    .domain([0, d3.max(bins, d => d.length)])
    .range([height - margin.bottom, margin.top]);

  // Bars
  svg.selectAll("rect")
    .data(bins)
    .enter()
    .append("rect")
    .attr("x", d => xScale(d.x0) + 1)
    .attr("y", d => yScale(d.length))
    .attr("width", d => xScale(d.x1) - xScale(d.x0) - 1)
    .attr("height", d => yScale(0) - yScale(d.length))
    .attr("fill", "steelblue");

  // X Axis
  svg.append("g")
    .attr("transform", `translate(0,${height - margin.bottom})`)
    .call(d3.axisBottom(xScale));

  // Y Axis
  svg.append("g")
    .attr("transform", `translate(${margin.left},0)`)
    .call(d3.axisLeft(yScale));
</script>

</body>
</html>
```

6.5.2 Boxplots: Summarizing Data with Five-Number Summary

A **boxplot** (or box-and-whisker plot) visually represents the minimum, first quartile (Q1), median, third quartile (Q3), and maximum of a dataset.

Step 1: Compute the Five-Number Summary

```
const sorted = data.slice().sort(d3.ascending);

const min = d3.min(sorted);
const max = d3.max(sorted);
const q1 = d3.quantile(sorted, 0.25);
const median = d3.quantile(sorted, 0.5);
const q3 = d3.quantile(sorted, 0.75);
```

Step 2: Draw the Boxplot Components

- **Box:** from Q1 to Q3
- **Median line:** inside the box
- **Whiskers:** lines from min to Q1 and Q3 to max
- **Optional:** individual outlier points

```
const svgWidth = 200;
const svgHeight = 100;
const margin = { top: 20, right: 30, bottom: 30, left: 40 };
const xScaleBox = d3.scaleLinear()
  .domain([min, max])
  .range([margin.left, svgWidth - margin.right]);

const svgBox = d3.select("#boxplot")
  .attr("width", svgWidth)
  .attr("height", svgHeight);

// Whiskers
svgBox.append("line")  // min to Q1
  .attr("x1", xScaleBox(min))
  .attr("x2", xScaleBox(q1))
  .attr("y1", svgHeight / 2)
  .attr("y2", svgHeight / 2)
  .attr("stroke", "black");

svgBox.append("line")  // Q3 to max
  .attr("x1", xScaleBox(q3))
  .attr("x2", xScaleBox(max))
  .attr("y1", svgHeight / 2)
  .attr("y2", svgHeight / 2)
  .attr("stroke", "black");

// Box from Q1 to Q3
svgBox.append("rect")
  .attr("x", xScaleBox(q1))
  .attr("y", svgHeight / 4)
  .attr("width", xScaleBox(q3) - xScaleBox(q1))
  .attr("height", svgHeight / 2)
  .attr("fill", "lightsteelblue")
  .attr("stroke", "black");

// Median line
svgBox.append("line")
  .attr("x1", xScaleBox(median))
  .attr("x2", xScaleBox(median))
  .attr("y1", svgHeight / 4)
  .attr("y2", svgHeight * 3 / 4)
  .attr("stroke", "black");
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Boxplot Example</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      text-align: center;
    }
    svg {
      margin-top: 30px;
    }
  </style>
```

```
</head>
<body>

<h3>D3 Boxplot</h3>
<svg id="boxplot"></svg>

<script>
  const data = [7, 9, 4, 3, 15, 18, 12, 8, 5, 10, 13, 21];

  const sorted = data.slice().sort(d3.ascending);
  const min = d3.min(sorted);
  const max = d3.max(sorted);
  const q1 = d3.quantile(sorted, 0.25);
  const median = d3.quantile(sorted, 0.5);
  const q3 = d3.quantile(sorted, 0.75);

  const svgWidth = 300;
  const svgHeight = 100;
  const margin = { top: 20, right: 20, bottom: 30, left: 20 };

  const xScale = d3.scaleLinear()
    .domain([min, max])
    .range([margin.left, svgWidth - margin.right]);

  const svg = d3.select("#boxplot")
    .attr("width", svgWidth)
    .attr("height", svgHeight);

  const midY = svgHeight / 2;
  const boxHeight = 30;

  // Whisker: min to Q1
  svg.append("line")
    .attr("x1", xScale(min))
    .attr("x2", xScale(q1))
    .attr("y1", midY)
    .attr("y2", midY)
    .attr("stroke", "black");

  // Whisker: Q3 to max
  svg.append("line")
    .attr("x1", xScale(q3))
    .attr("x2", xScale(max))
    .attr("y1", midY)
    .attr("y2", midY)
    .attr("stroke", "black");

  // Box from Q1 to Q3
  svg.append("rect")
    .attr("x", xScale(q1))
    .attr("y", midY - boxHeight / 2)
    .attr("width", xScale(q3) - xScale(q1))
    .attr("height", boxHeight)
    .attr("fill", "lightsteelblue")
    .attr("stroke", "black");

  // Median line
  svg.append("line")
```

```
    .attr("x1", xScale(median))
    .attr("x2", xScale(median))
    .attr("y1", midY - boxHeight / 2)
    .attr("y2", midY + boxHeight / 2)
    .attr("stroke", "black");

  // Min line
  svg.append("line")
    .attr("x1", xScale(min))
    .attr("x2", xScale(min))
    .attr("y1", midY - 10)
    .attr("y2", midY + 10)
    .attr("stroke", "black");

  // Max line
  svg.append("line")
    .attr("x1", xScale(max))
    .attr("x2", xScale(max))
    .attr("y1", midY - 10)
    .attr("y2", midY + 10)
    .attr("stroke", "black");
</script>

</body>
</html>
```

6.5.3 Summary

- **Histograms** group numeric data into bins to visualize frequency distributions; use `d3.bin()` to generate bins.
- **Boxplots** summarize data with minimum, Q1, median, Q3, and maximum, revealing central tendency and spread.
- D3's scales and SVG shapes let you build these statistical visuals programmatically.
- Combining histograms and boxplots offers a deeper understanding of your dataset's distribution and variability.

With histograms and boxplots, you gain powerful tools to explore data patterns and communicate insights effectively.

Chapter 7.

Interactivity and Tooltips

1. Mouse Events in D3 (`mouseover`, `click`, `mousemove`)
2. Creating Custom Tooltips
3. Hover Effects and Dynamic Highlighting
4. Filtering and Brushing

7 Interactivity and Tooltips

7.1 Mouse Events in D3 (`mouseover`, `click`, `mousemove`)

Interactivity is key to making your visualizations engaging and informative. D3 offers straightforward ways to listen and respond to user interactions by attaching event listeners directly to data-bound DOM elements. This section explains how to use common mouse events like `mouseover`, `click`, and `mousemove` to enhance your charts with interactive behaviors.

7.1.1 Handling DOM Events in D3

D3 uses the familiar `.on()` method to attach event listeners to selections. The syntax looks like:

```
selection.on(eventType, eventHandlerFunction);
```

Where:

- `eventType` is the name of the event like `"mouseover"`, `"click"`, or `"mousemove"`.
- `eventHandlerFunction` is a callback executed when the event fires.

Because D3 selections are often bound to data, your handler functions receive the bound data (`d`) and index (`i`) as arguments, allowing you to create data-driven interactivity.

7.1.2 Example: Highlighting a Bar on `mouseover`

Suppose you have a simple bar chart with rectangles representing data. You can change the bar color when the mouse hovers over it.

```
svg.selectAll("rect")
  .data(data)
  .enter()
  .append("rect")
  .attr("x", d => xScale(d.category))
  .attr("y", d => yScale(d.value))
  .attr("width", xScale.bandwidth())
  .attr("height", d => height - yScale(d.value))
  .attr("fill", "steelblue")
  .on("mouseover", function(event, d) {
    d3.select(this)
      .attr("fill", "orange");  // highlight on hover
  })
  .on("mouseout", function(event, d) {
    d3.select(this)
      .attr("fill", "steelblue");  // revert color when mouse leaves
  });
```

Notes:

- The first argument `event` provides the native DOM event, useful for advanced interactions.
- `this` refers to the hovered DOM element, so you can select and modify it directly.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Bar Hover Highlight</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      text-align: center;
    }
    svg {
      margin-top: 30px;
    }
  </style>
</head>
<body>

<h3>Bar Chart with Hover Highlight</h3>
<svg id="barChart"></svg>

<script>
  const data = [
    { category: "A", value: 30 },
    { category: "B", value: 80 },
    { category: "C", value: 45 },
    { category: "D", value: 60 },
    { category: "E", value: 20 }
  ];

  const width = 500;
  const height = 300;
  const margin = { top: 30, right: 20, bottom: 40, left: 40 };

  const svg = d3.select("#barChart")
    .attr("width", width)
    .attr("height", height);

  const xScale = d3.scaleBand()
    .domain(data.map(d => d.category))
    .range([margin.left, width - margin.right])
    .padding(0.1);

  const yScale = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.value)])
    .range([height - margin.bottom, margin.top]);

  // Axes
  svg.append("g")
```

```
    .attr("transform", `translate(0, ${height - margin.bottom})`)
    .call(d3.axisBottom(xScale));

  svg.append("g")
    .attr("transform", `translate(${margin.left}, 0)`)
    .call(d3.axisLeft(yScale));

  // Bars with hover effect
  svg.selectAll("rect")
    .data(data)
    .enter()
    .append("rect")
    .attr("x", d => xScale(d.category))
    .attr("y", d => yScale(d.value))
    .attr("width", xScale.bandwidth())
    .attr("height", d => height - margin.bottom - yScale(d.value))
    .attr("fill", "steelblue")
    .on("mouseover", function(event, d) {
      d3.select(this).attr("fill", "orange");
    })
    .on("mouseout", function(event, d) {
      d3.select(this).attr("fill", "steelblue");
    });
</script>

</body>
</html>
```

7.1.3 Handling `click` Events for Selection or Interaction

You can capture clicks on elements to trigger actions like filtering or showing details.

```
svg.selectAll("circle")
  .data(data)
  .enter()
  .append("circle")
  .attr("cx", d => xScale(d.x))
  .attr("cy", d => yScale(d.y))
  .attr("r", 5)
  .attr("fill", "steelblue")
  .on("click", function(event, d) {
    alert(`Clicked point: (${d.x}, ${d.y})`);
  });
```

This popup provides immediate feedback tied to the clicked data point.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Circle Click Interaction</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    svg {
      border: 1px solid #ccc;
      display: block;
      margin: 20px auto;
      background: #fafafa;
    }
  </style>
</head>
<body>

<svg width="500" height="300"></svg>

<script>
  const data = [
    { x: 30, y: 50 },
    { x: 80, y: 90 },
    { x: 130, y: 40 },
    { x: 180, y: 120 },
    { x: 230, y: 70 }
  ];

  const svg = d3.select("svg");
  const width = +svg.attr("width");
  const height = +svg.attr("height");

  // Scales assuming data x,y are within [0, 250]
  const xScale = d3.scaleLinear()
    .domain([0, 250])
    .range([40, width - 40]);

  const yScale = d3.scaleLinear()
    .domain([0, 150])
    .range([height - 40, 40]);

  // Draw circles
  svg.selectAll("circle")
    .data(data)
    .enter()
    .append("circle")
    .attr("cx", d => xScale(d.x))
    .attr("cy", d => yScale(d.y))
    .attr("r", 8)
    .attr("fill", "steelblue")
    .style("cursor", "pointer")
    .on("click", (event, d) => {
      alert(`Clicked point: (${d.x}, ${d.y})`);
    });
</script>

</body>
</html>
```

7.1.4 Using `mousemove` for Dynamic Interaction

`mousemove` events let you track cursor movement over elements, enabling advanced interactions like tooltips or crosshairs.

Example:

```
svg.on("mousemove", function(event) {
  const [mouseX, mouseY] = d3.pointer(event);
  console.log(`Mouse at: ${mouseX}, ${mouseY}`);
  // You can update tooltip position or highlight nearby points here
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>D3 Mousemove Interaction</title>
<script src="https://d3js.org/d3.v7.min.js"></script>
<style>
  svg {
    border: 1px solid #ccc;
    display: block;
    margin: 20px auto;
    background: #f9f9f9;
  }
  #coords {
    text-align: center;
    font-family: sans-serif;
    margin-top: 10px;
    font-size: 16px;
  }
</style>
</head>
<body>

<svg width="500" height="300"></svg>
<div id="coords">Move your mouse over the SVG</div>

<script>
  const svg = d3.select("svg");
  const coordsDiv = d3.select("#coords");

  svg.on("mousemove", function(event) {
    // d3.pointer(event) gives [x, y] relative to the SVG
    const [mouseX, mouseY] = d3.pointer(event);
    coordsDiv.text(`Mouse at: ${mouseX.toFixed(1)}, ${mouseY.toFixed(1)}`);
  });

  svg.on("mouseleave", () => {
    coordsDiv.text("Move your mouse over the SVG");
  });
</script>

</body>
```

```
</html>
```

7.1.5 Summary

- Use `.on("event", handler)` to attach event listeners in D3, with access to the event, data, and element.
- `mouseover` and `mouseout` are great for hover effects such as color changes or highlighting.
- `click` enables user interaction for selection, filtering, or detailed data inspection.
- `mousemove` tracks cursor position for dynamic effects like tooltips or live updates.

Mastering mouse events unlocks the ability to build rich, interactive visualizations that respond intuitively to user input.

7.2 Creating Custom Tooltips

Tooltips are invaluable for providing users with additional context and detailed information about specific data points without cluttering the main visualization. In D3, creating custom tooltips using HTML `<div>` elements is flexible and allows rich styling and dynamic content. This section walks you through building a responsive, interactive tooltip that follows the mouse cursor.

7.2.1 Step 1: Create the Tooltip `div`

Start by adding a hidden `<div>` in your HTML that will serve as the tooltip container:

```
<div id="tooltip" style="
  position: absolute;
  pointer-events: none;  /* allows mouse events to pass through */
  background-color: rgba(0, 0, 0, 0.7);
  color: white;
  padding: 6px 10px;
  border-radius: 4px;
  font-size: 12px;
  opacity: 0;          /* initially invisible */
  transition: opacity 0.3s ease;
"></div>
```

7.2.2 Step 2: Select Tooltip in Your Script

In your JavaScript, select this tooltip for later manipulation:

```
const tooltip = d3.select("#tooltip");
```

7.2.3 Step 3: Show and Position Tooltip on Mouse Events

Attach mouse event handlers to your data-bound elements. For example, when hovering over circles in a scatter plot:

```
svg.selectAll("circle")
  .data(data)
  .enter()
  .append("circle")
  .attr("cx", d => xScale(d.x))
  .attr("cy", d => yScale(d.y))
  .attr("r", 6)
  .attr("fill", "steelblue")
  .on("mouseover", (event, d) => {
    tooltip.style("opacity", 1)
      .html(`
        <strong>${d.category}</strong><br/>
        X: ${d.x}<br/>
        Y: ${d.y}
      `);
  })
  .on("mousemove", (event) => {
    tooltip.style("left", (event.pageX + 10) + "px")
           .style("top", (event.pageY + 10) + "px");
  })
  .on("mouseout", () => {
    tooltip.style("opacity", 0);
  });
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Tooltip Example</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
    }

    svg {
      background-color: #f4f4f4;
      border: 1px solid #ccc;
      display: block;
      margin: 30px auto;
    }
    #tooltip {
      position: absolute;
      pointer-events: none;
      background-color: rgba(0, 0, 0, 0.7);
      color: white;
      padding: 6px 10px;
      border-radius: 4px;
      font-size: 12px;
      opacity: 0;
      transition: opacity 0.3s ease;
    }
  </style>
</head>
<body>

<svg width="500" height="300"></svg>
<div id="tooltip"></div>

<script>
  const data = [
    { category: "A", x: 30, y: 40 },
    { category: "B", x: 120, y: 90 },
    { category: "C", x: 220, y: 150 },
    { category: "D", x: 320, y: 60 },
    { category: "E", x: 420, y: 200 }
  ];

  const svg = d3.select("svg");
  const tooltip = d3.select("#tooltip");

  const xScale = d3.scaleLinear().domain([0, 500]).range([0, 500]);
  const yScale = d3.scaleLinear().domain([0, 300]).range([300, 0]);

  svg.selectAll("circle")
    .data(data)
    .enter()
    .append("circle")
    .attr("cx", d => xScale(d.x))
    .attr("cy", d => yScale(d.y))
    .attr("r", 6)
    .attr("fill", "steelblue")
    .on("mouseover", (event, d) => {
      tooltip.style("opacity", 1)
        .html(`
          <strong>${d.category}</strong><br/>
          X: ${d.x}<br/>
          Y: ${d.y}
        `);
    })
    .on("mousemove", (event) => {
      tooltip
        .style("left", (event.pageX + 10) + "px")
        .style("top", (event.pageY + 10) + "px");
    })
    .on("mouseout", () => {
      tooltip.style("opacity", 0);
    });
</script>
```

```
</body>
</html>
```

7.2.4 Explanation of Key Parts

- **mouseover**: Shows the tooltip by setting opacity to 1 and injects dynamic HTML content based on the hovered data point.
- **mousemove**: Updates the tooltip's position dynamically to follow the cursor with a small offset to avoid covering the pointer. Uses `event.pageX` and `event.pageY` for absolute positioning.
- **mouseout**: Hides the tooltip smoothly by fading opacity back to 0.

7.2.5 Handling Tooltip Positioning and Responsiveness

- Positioning with `pageX`/`pageY` ensures the tooltip aligns with the mouse anywhere on the page, even if the SVG is scrolled.
- Adding an offset (e.g., 10px) prevents the tooltip from covering the mouse pointer.
- You can add logic to keep the tooltip inside viewport bounds to avoid clipping (optional for advanced handling).
- Styling via CSS lets you customize colors, fonts, padding, and shadows for clarity and aesthetics.

7.2.6 Optional: Smooth Transitions

You can use D3 transitions for fading tooltips in and out:

```
tooltip.transition()
  .duration(200)
  .style("opacity", 1);

tooltip.transition()
  .duration(200)
  .style("opacity", 0);
```

Replace the direct `style("opacity", …)` calls in the handlers to make showing and hiding smoother.

7.2.7 Summary

- Use an absolutely positioned HTML `<div>` outside the SVG for flexible, styled tooltips.
- Dynamically inject content based on the hovered data point for rich information display.
- Update tooltip position on every `mousemove` event to follow the cursor.
- Use opacity and CSS transitions for smooth showing and hiding effects.
- Consider viewport boundaries for better user experience (advanced).

Custom tooltips dramatically improve usability and insight in your D3 visualizations by revealing detailed data precisely when and where users need it.

7.3 Hover Effects and Dynamic Highlighting

Hover effects and dynamic highlighting help users focus on specific data points while subtly de-emphasizing unrelated elements. In D3, these interactive visual cues enhance comprehension and make your charts feel responsive and polished. This section shows how to apply hover styles using `.style()` or CSS class toggling, plus how to coordinate highlighting across multiple elements.

7.3.1 Changing Appearance on Hover

You can easily change an element's style during hover by attaching `mouseover` and `mouseout` events and using `.style()` or `.classed()` to toggle CSS classes.

Example: Simple color change with inline style

```
svg.selectAll("circle")
  .data(data)
  .enter()
  .append("circle")
  .attr("cx", d => xScale(d.x))
  .attr("cy", d => yScale(d.y))
  .attr("r", 6)
  .attr("fill", "steelblue")
  .on("mouseover", function() {
    d3.select(this).style("fill", "orange");
  })
  .on("mouseout", function() {
    d3.select(this).style("fill", "steelblue");
  });
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
  <meta charset="UTF-8">
  <title>D3 Hover Style Change Example</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
    }

    svg {
      background-color: #f4f4f4;
      border: 1px solid #ccc;
      display: block;
      margin: 30px auto;
    }
  </style>
</head>
<body>

<svg width="500" height="300"></svg>

<script>
  const data = [
    { x: 50, y: 50 },
    { x: 150, y: 80 },
    { x: 250, y: 120 },
    { x: 350, y: 60 },
    { x: 450, y: 200 }
  ];

  const svg = d3.select("svg");

  const xScale = d3.scaleLinear().domain([0, 500]).range([0, 500]);
  const yScale = d3.scaleLinear().domain([0, 300]).range([300, 0]);

  svg.selectAll("circle")
    .data(data)
    .enter()
    .append("circle")
    .attr("cx", d => xScale(d.x))
    .attr("cy", d => yScale(d.y))
    .attr("r", 6)
    .attr("fill", "steelblue")
    .on("mouseover", function () {
      d3.select(this).style("fill", "orange");
    })
    .on("mouseout", function () {
      d3.select(this).style("fill", "steelblue");
    });
</script>

</body>
</html>
```

7.3.2 Using CSS Classes for Hover States

For cleaner code and better maintainability, toggle CSS classes that define hover styles:

```
.hovered {
  fill: orange;
  stroke: darkorange;
  stroke-width: 2px;
}
.dimmed {
  opacity: 0.3;
}
```

Then in JavaScript:

```
svg.selectAll("circle")
  .on("mouseover", function() {
    d3.select(this).classed("hovered", true);
  })
  .on("mouseout", function() {
    d3.select(this).classed("hovered", false);
  });
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Hover with CSS Classes</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    svg {
      background: #f0f0f0;
      border: 1px solid #ccc;
      display: block;
      margin: 30px auto;
    }

    .hovered {
      fill: orange;
      stroke: darkorange;
      stroke-width: 2px;
    }

    .dimmed {
      opacity: 0.3;
    }
  </style>
</head>
<body>

<svg width="500" height="300"></svg>

<script>
  const data = [
    { x: 50, y: 50 },
    { x: 150, y: 80 },
    { x: 250, y: 120 },
    { x: 350, y: 60 },
    { x: 450, y: 200 }
  ];

  const svg = d3.select("svg");

  const xScale = d3.scaleLinear().domain([0, 500]).range([0, 500]);
  const yScale = d3.scaleLinear().domain([0, 300]).range([300, 0]);

  svg.selectAll("circle")
    .data(data)
    .enter()
    .append("circle")
    .attr("cx", d => xScale(d.x))
    .attr("cy", d => yScale(d.y))
    .attr("r", 8)
    .attr("fill", "steelblue")
    .on("mouseover", function(event, d) {
      // Dim all circles
      svg.selectAll("circle").classed("dimmed", true);
      // Highlight hovered one
      d3.select(this)
        .classed("dimmed", false)
        .classed("hovered", true);
    })
    .on("mouseout", function(event, d) {
      svg.selectAll("circle")
        .classed("dimmed", false)
        .classed("hovered", false);
    });
</script>

</body>
</html>
```

7.3.3 Coordinated Highlighting: Dim Others on Hover

To help users focus, dim all elements except the hovered one:

```
svg.selectAll("circle")
  .on("mouseover", function() {
    svg.selectAll("circle")
      .classed("dimmed", true);
    d3.select(this)
      .classed("dimmed", false)
      .classed("hovered", true);
  })
  .on("mouseout", function() {
    svg.selectAll("circle")
      .classed("dimmed", false)
      .classed("hovered", false);
  });
```

This effect highlights the hovered circle by making others semi-transparent.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Coordinated Highlighting</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    svg {
      background: #fafafa;
      border: 1px solid #ccc;
      display: block;
      margin: 40px auto;
    }

    .hovered {
      fill: orange;
      stroke: darkorange;
      stroke-width: 2px;
    }

    .dimmed {
      opacity: 0.3;
    }
  </style>
</head>
<body>

<svg width="500" height="300"></svg>

<script>
  const data = [
    { x: 50, y: 50 },
    { x: 150, y: 100 },
    { x: 250, y: 180 },
    { x: 350, y: 60 },
    { x: 450, y: 220 }
  ];

  const svg = d3.select("svg");

  const xScale = d3.scaleLinear().domain([0, 500]).range([0, 500]);
  const yScale = d3.scaleLinear().domain([0, 300]).range([300, 0]);

  svg.selectAll("circle")
    .data(data)
    .enter()
    .append("circle")
    .attr("cx", d => xScale(d.x))
    .attr("cy", d => yScale(d.y))
    .attr("r", 8)
    .attr("fill", "steelblue")
    .on("mouseover", function(event, d) {
      svg.selectAll("circle")
        .classed("dimmed", true);
      d3.select(this)
        .classed("dimmed", false)
        .classed("hovered", true);
    })
    .on("mouseout", function() {
      svg.selectAll("circle")
        .classed("dimmed", false)
        .classed("hovered", false);
    });
</script>

</body>
</html>
```

7.3.4 Cross-Chart Highlighting Example

If you have multiple charts or elements, you can highlight related data points across them by coordinating selections:

```
svg1.selectAll("rect")
  .on("mouseover", function(event, d) {
    svg2.selectAll("circle")
      .classed("dimmed", true);

    svg2.selectAll("circle")
      .filter(cd => cd.category === d.category)
      .classed("dimmed", false)
      .classed("hovered", true);
  })
  .on("mouseout", function() {
    svg2.selectAll("circle")
      .classed("dimmed", false)
      .classed("hovered", false);
  });
```

This way, hovering a bar in one chart highlights matching circles in another.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Cross-Chart Highlighting</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
    }
    svg {
      background: #f9f9f9;
      margin: 20px;
      border: 1px solid #ccc;
    }
    .hovered {
      fill: orange;
      stroke: darkorange;
      stroke-width: 2px;
    }
    .dimmed {
      opacity: 0.3;
    }
  </style>
</head>
<body>

<svg id="chart1" width="300" height="200"></svg>
<svg id="chart2" width="300" height="200"></svg>

<script>
  const data = [
    { category: "A", value: 40 },
    { category: "B", value: 70 },
    { category: "C", value: 50 }
  ];

  const svg1 = d3.select("#chart1");
  const svg2 = d3.select("#chart2");

  const xScale1 = d3.scaleBand()
    .domain(data.map(d => d.category))
    .range([40, 260])
    .padding(0.2);

  const yScale1 = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.value)])
    .range([160, 20]);

  const xScale2 = d3.scalePoint()
    .domain(data.map(d => d.category))
    .range([60, 240]);

  const yScale2 = d3.scaleLinear()
    .domain([0, 100])
    .range([160, 20]);

  // Chart 1: Bar Chart
  svg1.selectAll("rect")
    .data(data)
    .enter()
    .append("rect")
    .attr("x", d => xScale1(d.category))
    .attr("y", d => yScale1(d.value))
    .attr("width", xScale1.bandwidth())
    .attr("height", d => 160 - yScale1(d.value))
    .attr("fill", "steelblue")
    .on("mouseover", function(event, d) {
      svg2.selectAll("circle")
        .classed("dimmed", true);

      svg2.selectAll("circle")
```

```
        .filter(cd => cd.category === d.category)
        .classed("dimmed", false)
        .classed("hovered", true);
    })
    .on("mouseout", function() {
      svg2.selectAll("circle")
        .classed("dimmed", false)
        .classed("hovered", false);
    });

  // Chart 2: Circles
  svg2.selectAll("circle")
    .data(data)
    .enter()
    .append("circle")
    .attr("cx", d => xScale2(d.category))
    .attr("cy", d => yScale2(d.value))
    .attr("r", 10)
    .attr("fill", "steelblue");
</script>

</body>
</html>
```

7.3.5 Summary

- Use `.style()` or `.classed()` in `mouseover` and `mouseout` handlers to change element appearance dynamically.
- CSS classes help keep styles organized and reusable for hover and dimming effects.
- Coordinated highlighting—dimming non-focused elements—improves focus and readability.
- Cross-chart coordination allows linked interactions between multiple visualizations, enhancing storytelling.

Mastering hover and highlighting effects will make your visualizations more interactive and intuitive, guiding users' attention exactly where you want it.

7.4 Filtering and Brushing

Interactivity often involves letting users explore data by filtering or selecting subsets dynamically. Two powerful techniques in D3 for this are **filtering** through UI controls and **brushing**, a direct manipulation method for selecting ranges on the visualization. This section introduces both approaches and demonstrates how brushing can update connected charts seamlessly.

7.4.1 Filtering with UI Controls

Filtering allows users to narrow down data displayed by interacting with dropdown menus, sliders, or checkboxes.

Example: Filtering with a Dropdown

Imagine you have a scatter plot showing data points for different categories, and you want to filter by category.

HTML dropdown:

```html
<select id="categoryFilter">
  <option value="all">All</option>
  <option value="A">Category A</option>
  <option value="B">Category B</option>
  <option value="C">Category C</option>
</select>
```

JavaScript filtering logic:

```javascript
const allData = [...];  // your full dataset

d3.select("#categoryFilter").on("change", function(event) {
  const selected = event.target.value;
  const filteredData = selected === "all"
    ? allData
    : allData.filter(d => d.category === selected);

  updateScatterPlot(filteredData);
});

function updateScatterPlot(data) {
  const circles = svg.selectAll("circle")
    .data(data, d => d.id);  // key function for consistent binding

  circles.exit().remove();

  circles.enter()
    .append("circle")
    .merge(circles)
    .attr("cx", d => xScale(d.x))
    .attr("cy", d => yScale(d.y))
    .attr("r", 6)
    .attr("fill", "steelblue");
}
```

The dropdown triggers filtering, and the visualization updates accordingly.

Full runnable code:

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Dropdown Filtering Example</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 20px;
    }
    select {
      margin-bottom: 10px;
      padding: 4px;
    }
    svg {
      border: 1px solid #ccc;
      background: #f9f9f9;
    }
  </style>
</head>
<body>

<h3>Filter by Category:</h3>
<select id="categoryFilter">
  <option value="all">All</option>
  <option value="A">Category A</option>
  <option value="B">Category B</option>
  <option value="C">Category C</option>
</select>

<svg id="scatter" width="500" height="300"></svg>

<script>
  const allData = [
    { id: 1, x: 30, y: 40, category: "A" },
    { id: 2, x: 80, y: 90, category: "B" },
    { id: 3, x: 130, y: 70, category: "C" },
    { id: 4, x: 180, y: 40, category: "A" },
    { id: 5, x: 230, y: 60, category: "B" },
    { id: 6, x: 280, y: 90, category: "C" },
    { id: 7, x: 330, y: 50, category: "A" },
    { id: 8, x: 380, y: 80, category: "B" },
    { id: 9, x: 430, y: 40, category: "C" }
  ];

  const svg = d3.select("#scatter");
  const width = +svg.attr("width");
  const height = +svg.attr("height");

  const xScale = d3.scaleLinear()
    .domain([0, 500])
    .range([40, width - 20]);

  const yScale = d3.scaleLinear()
    .domain([0, 100])
    .range([height - 30, 20]);

  function updateScatterPlot(data) {
    const circles = svg.selectAll("circle")
      .data(data, d => d.id);

    circles.exit().remove();
    circles.enter()
      .append("circle")
      .merge(circles)
      .attr("cx", d => xScale(d.x))
      .attr("cy", d => yScale(d.y))
      .attr("r", 8)
      .attr("fill", "steelblue");
  }

  d3.select("#categoryFilter").on("change", function(event) {
    const selected = event.target.value;
    const filteredData = selected === "all"
      ? allData
      : allData.filter(d => d.category === selected);
    updateScatterPlot(filteredData);
  });

  // Initial render
  updateScatterPlot(allData);
</script>

</body>
</html>
```

7.4.2 Brushing: Selecting Data Ranges Directly

Brushing lets users click and drag to select a rectangular region on the chart, typically for zooming, filtering, or highlighting data points within the selected area.

D3 provides a built-in brush behavior via `d3.brush()`.

Step 1: Create a Brush

```
const brush = d3.brush()
  .extent([[margin.left, margin.top], [width - margin.right, height - margin.bottom]])
  .on("brush end", brushed);

svg.append("g")
  .attr("class", "brush")
  .call(brush);
```

Step 2: Handle the Brush Event

The `brushed` function receives the selection box `[ [x0, y0], [x1, y1] ]` in pixel coordinates.

```
function brushed(event) {
  if (!event.selection) return;  // Ignore empty selections

  const [[x0, y0], [x1, y1]] = event.selection;

  // Convert pixel space back to data space
  const xDomain = [xScale.invert(x0), xScale.invert(x1)];
  const yDomain = [yScale.invert(y1), yScale.invert(y0)];  // y is inverted in SVG

  // Filter data points within brush extent
  const brushedData = allData.filter(d =>
    d.x >= xDomain[0] && d.x <= xDomain[1] &&
    d.y >= yDomain[0] && d.y <= yDomain[1]
  );

  updateScatterPlot(brushedData);
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Brushing Example</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 20px;
    }
    svg {
      border: 1px solid #ccc;
      background: #fafafa;
    }
    circle {
      fill: steelblue;
    }
    circle.selected {
      fill: orange;
      stroke: darkorange;
      stroke-width: 2px;
    }
  </style>
</head>
<body>

<h3>Scatter Plot with Brushing</h3>
<svg id="scatter" width="600" height="400"></svg>

<script>
  const svg = d3.select("#scatter");
  const width = +svg.attr("width");
  const height = +svg.attr("height");
  const margin = { top: 20, right: 20, bottom: 30, left: 40 };

  const allData = d3.range(50).map((d, i) => ({
    id: i,
    x: Math.random() * 100,
    y: Math.random() * 100
  }));

  const xScale = d3.scaleLinear()
    .domain([0, 100])
    .range([margin.left, width - margin.right]);

  const yScale = d3.scaleLinear()
    .domain([0, 100])
    .range([height - margin.bottom, margin.top]);

  // Render axes
  svg.append("g")
    .attr("transform", `translate(0, ${height - margin.bottom})`)
    .call(d3.axisBottom(xScale));

  svg.append("g")
    .attr("transform", `translate(${margin.left}, 0)`)
    .call(d3.axisLeft(yScale));

  // Add points
  const points = svg.append("g")
    .attr("class", "points")
    .selectAll("circle")
    .data(allData)
    .enter()
    .append("circle")
    .attr("cx", d => xScale(d.x))
    .attr("cy", d => yScale(d.y))
    .attr("r", 6);

  // Brush
  const brush = d3.brush()
    .extent([[margin.left, margin.top], [width - margin.right, height - margin.bottom]])
    .on("brush end", brushed);

  svg.append("g")
    .attr("class", "brush")
    .call(brush);

  function brushed(event) {
    if (!event.selection) {
      points.classed("selected", false);
      return;
    }

    const [[x0, y0], [x1, y1]] = event.selection;

    points.classed("selected", d => {
      const x = xScale(d.x);
      const y = yScale(d.y);
      return x >= x0 && x <= x1 && y >= y0 && y <= y1;
    });
  }
</script>

</body>
</html>
```

7.4.3 Linking Brushing to Other Visualizations

You can extend brushing to update other connected charts. For example, brushing on a scatter plot could filter a bar chart to show counts only for brushed points.

```
function updateBarChart(filteredData) {
  // Update bar chart based on filteredData
}

function brushed(event) {
  // ... same as above

  updateScatterPlot(brushedData);
  updateBarChart(brushedData);
}
```

This linked interactivity empowers multi-view exploration, allowing users to drill down into data details seamlessly.

7.4.4 Summary

- **Filtering** with UI controls (dropdowns, sliders) lets users select subsets of data programmatically.
- **Brushing** provides a direct, visual way to select data ranges by dragging on the chart.
- D3's `d3.brush()` handles selection areas and emits events with coordinates you can map back to data domains.
- Linking brushing selections to other charts fosters coordinated views and deeper data insights.

Together, filtering and brushing turn static visuals into dynamic exploration tools, greatly enhancing user engagement and understanding.

Chapter 8.

Transitions and Animation

8 Transitions and Animation

8.1 D3 Transitions Explained

Transitions are a cornerstone of creating smooth, engaging visualizations with D3. They enable gradual changes of visual attributes over time, helping users visually track changes in the data or interface. Instead of abrupt jumps, transitions animate properties like position, size, color, or opacity, making your charts feel more natural and polished.

8.1.1 What Are Transitions in D3?

A **transition** in D3 is essentially a timed interpolation between starting and ending states of DOM elements. When you call `.transition()` on a selection, D3 schedules updates to element attributes or styles, smoothly animating from their current value to a new target value.

This capability is built directly into D3's API and integrates seamlessly with selections and data binding.

8.1.2 Basic Usage: `.transition()`, `.duration()`, and `.ease()`

Here's the typical workflow:

```
selection.transition()        // start a transition on the selection
  .duration(1000)             // length of the animation in milliseconds
  .ease(d3.easeCubic)         // easing function to control speed curve
  .attr("attribute", value)  // animate an attribute to a new value
  .style("property", value); // animate a CSS style property
```

- `.duration()` sets how long the transition lasts (e.g., 1000ms = 1 second).
- `.ease()` defines the pacing (speed over time) — common options include linear, cubic, bounce, and elastic.
- You can chain multiple attribute or style changes to animate several properties simultaneously.

8.1.3 Example: Animating a Bar Chart Update

Suppose you update the data in a vertical bar chart, changing the height of bars to reflect new values. Without transitions, bars would jump instantly to their new height:

```
svg.selectAll("rect")
  .data(newData)
  .join("rect")
  .attr("x", d => xScale(d.category))
  .attr("width", xScale.bandwidth())
  .attr("y", d => yScale(d.value))
  .attr("height", d => height - yScale(d.value));
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Bar Chart Transition</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
    }
    button {
      margin: 10px;
      padding: 6px 12px;
    }
  </style>
</head>
<body>
  <button id="update">Update Data</button>
  <svg width="500" height="300"></svg>

  <script>
    const svg = d3.select("svg");
    const width = +svg.attr("width");
    const height = +svg.attr("height");
    const margin = { top: 20, right: 20, bottom: 30, left: 40 };

    const data = [
      { category: "A", value: 30 },
      { category: "B", value: 80 },
      { category: "C", value: 45 },
      { category: "D", value: 60 },
      { category: "E", value: 20 }
    ];

    const xScale = d3.scaleBand()
      .domain(data.map(d => d.category))
      .range([margin.left, width - margin.right])
      .padding(0.2);

    const yScale = d3.scaleLinear()
      .domain([0, 100])
      .range([height - margin.bottom, margin.top]);

    svg.append("g")
      .attr("transform", `translate(0,${height - margin.bottom})`)
      .call(d3.axisBottom(xScale));
    svg.append("g")
      .attr("transform", `translate(${margin.left},0)`)
      .call(d3.axisLeft(yScale));

    // Initial draw
    svg.selectAll("rect")
      .data(data, d => d.category)
      .join("rect")
      .attr("x", d => xScale(d.category))
      .attr("width", xScale.bandwidth())
      .attr("y", d => yScale(d.value))
      .attr("height", d => height - margin.bottom - yScale(d.value))
      .attr("fill", "steelblue");

    // Update data on button click
    d3.select("#update").on("click", () => {
      const newData = data.map(d => ({
        ...d,
        value: Math.floor(Math.random() * 100)
      }));

      svg.selectAll("rect")
        .data(newData, d => d.category)
        .transition()
        .duration(800)
        .attr("y", d => yScale(d.value))
        .attr("height", d => height - margin.bottom - yScale(d.value));
    });
  </script>
</body>
</html>
```

To animate the height changes smoothly, add a transition:

```
svg.selectAll("rect")
  .data(newData)
  .join("rect")
  .attr("x", d => xScale(d.category))
  .attr("width", xScale.bandwidth())
  .transition()
  .duration(800)
  .ease(d3.easeCubicInOut)
  .attr("y", d => yScale(d.value))
  .attr("height", d => height - yScale(d.value));
```

When new data arrives, bars grow or shrink fluidly, helping users visually follow the update.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Smooth Bar Transitions</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
    }
    button {
      margin: 10px;
      padding: 6px 12px;
    }
  </style>
</head>
<body>
  <button id="update">Update Data</button>
  <svg width="500" height="300"></svg>

  <script>
    const svg = d3.select("svg");
    const width = +svg.attr("width");
    const height = +svg.attr("height");
    const margin = { top: 20, right: 20, bottom: 30, left: 40 };

    const data = [
      { category: "A", value: 30 },
      { category: "B", value: 80 },
      { category: "C", value: 45 },
      { category: "D", value: 60 },
      { category: "E", value: 20 }
    ];

    const xScale = d3.scaleBand()
      .domain(data.map(d => d.category))
      .range([margin.left, width - margin.right])
      .padding(0.2);

    const yScale = d3.scaleLinear()
      .domain([0, 100])
      .range([height - margin.bottom, margin.top]);

    svg.append("g")
      .attr("transform", `translate(0,${height - margin.bottom})`)
      .call(d3.axisBottom(xScale));

    svg.append("g")
      .attr("transform", `translate(${margin.left},0)`)
      .call(d3.axisLeft(yScale));

    // Initial bars
    svg.selectAll("rect")
      .data(data, d => d.category)
      .join("rect")
      .attr("x", d => xScale(d.category))
      .attr("width", xScale.bandwidth())
      .attr("y", d => yScale(d.value))
      .attr("height", d => height - margin.bottom - yScale(d.value))
      .attr("fill", "steelblue");

    d3.select("#update").on("click", () => {
      const newData = data.map(d => ({
        ...d,
        value: Math.floor(Math.random() * 100)
      }));
      svg.selectAll("rect")
        .data(newData, d => d.category)
        .join("rect")
        .attr("x", d => xScale(d.category))
        .attr("width", xScale.bandwidth())
        .transition()
        .duration(800)
        .ease(d3.easeCubicInOut)
        .attr("y", d => yScale(d.value))
        .attr("height", d => height - margin.bottom - yScale(d.value));
    });
  </script>
</body>
</html>
```

8.1.4 Animating Positions and Colors

Transitions aren't limited to size or position—you can animate any numeric attribute or style, such as color:

```
svg.selectAll("circle")
  .data(data)
  .join("circle")
  .attr("cx", d => xScale(d.x))
  .attr("cy", d => yScale(d.y))
  .attr("r", 5)
  .attr("fill", "steelblue")
  .transition()
  .duration(1500)
  .ease(d3.easeBounce)
  .attr("cx", d => xScale(d.x + 10))  // move circles 10 units right
  .attr("fill", "orange");             // change color
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>D3 Animate Circle Position & Color</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
    }
  </style>
</head>
<body>
  <svg width="600" height="200"></svg>

  <script>
    const svg = d3.select("svg");
    const width = +svg.attr("width");
    const height = +svg.attr("height");
    const margin = { top: 20, right: 20, bottom: 30, left: 40 };

    // Sample data with x and y values
    const data = [
      { x: 10, y: 50 },
      { x: 50, y: 80 },
      { x: 90, y: 40 },
      { x: 130, y: 70 },
      { x: 170, y: 60 }
    ];

    // Scales for positioning circles
    const xScale = d3.scaleLinear()
      .domain([0, 200])
      .range([margin.left, width - margin.right]);

    const yScale = d3.scaleLinear()
      .domain([0, 100])
      .range([height - margin.bottom, margin.top]);

    // Initial circles
    svg.selectAll("circle")
      .data(data)
      .join("circle")
      .attr("cx", d => xScale(d.x))
      .attr("cy", d => yScale(d.y))
      .attr("r", 10)
      .attr("fill", "steelblue")
      .transition()
      .duration(1500)
      .ease(d3.easeBounce)
      .attr("cx", d => xScale(d.x + 10))  // move circles 10 units to the right
      .attr("fill", "orange");
  </script>
</body>
</html>
```

8.1.5 Summary

- D3 transitions interpolate attributes and styles over time, making updates smooth and visually appealing.
- Use `.transition()` to start animating a selection, with `.duration()` to set animation length and `.ease()` to control pacing.
- Transitions can animate positions, sizes, colors, opacity, and more.
- Smooth animations help users better understand data changes and improve overall user experience.

Mastering transitions is essential for building professional, fluid data visualizations with D3.

8.2 Animating Chart Elements

Animating chart elements brings your visualizations to life, making changes easier to understand and more engaging. In this section, we'll explore how to animate properties such as height, position, and color in response to updated data. We'll also highlight the power of chaining transitions to create smooth, coordinated animations.

8.2.1 Animating Bar Height and Position on Data Update

Imagine you have a vertical bar chart, and the data values change dynamically. Instead of bars instantly jumping to new heights, you can animate their growth or shrinkage smoothly.

```
// Assume svg, xScale, yScale, and height are already defined
function updateBars(data) {
  const bars = svg.selectAll("rect")
    .data(data, d => d.category);

  bars.enter()
    .append("rect")
    .attr("x", d => xScale(d.category))
    .attr("width", xScale.bandwidth())
    .attr("y", height)           // start from bottom
    .attr("height", 0)           // height zero for entering bars
    .attr("fill", "steelblue")
    .merge(bars)                 // merge enter + update
    .transition()
    .duration(800)
    .ease(d3.easeCubicInOut)
    .attr("y", d => yScale(d.value))
    .attr("height", d => height - yScale(d.value));

  bars.exit()
    .transition()
    .duration(500)
    .attr("y", height)
    .attr("height", 0)
    .remove();
}
```

- Entering bars start with zero height at the bottom and grow up to their new value.
- Updating bars smoothly transition to new heights and positions.
- Exiting bars shrink down and disappear.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Animated Bar Chart Update</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
```

```
    body {
      font-family: sans-serif;
    }
    svg {
      border: 1px solid #ccc;
    }
  </style>
</head>
<body>

<button id="updateBtn">Update Data</button>
<svg width="600" height="400"></svg>

<script>
  const svg = d3.select("svg");
  const margin = { top: 30, right: 20, bottom: 50, left: 60 };
  const width = +svg.attr("width") - margin.left - margin.right;
  const height = +svg.attr("height") - margin.top - margin.bottom;

  const chart = svg.append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

  // Initial data
  let data = [
    { category: "A", value: 30 },
    { category: "B", value: 80 },
    { category: "C", value: 45 },
    { category: "D", value: 60 }
  ];

  // Scales
  const xScale = d3.scaleBand()
    .domain(data.map(d => d.category))
    .range([0, width])
    .padding(0.2);

  const yScale = d3.scaleLinear()
    .domain([0, 100])  // fixed max for demo consistency
    .range([height, 0]);

  // Axes
  chart.append("g")
    .attr("transform", `translate(0,${height})`)
    .call(d3.axisBottom(xScale));

  chart.append("g")
    .call(d3.axisLeft(yScale));

  // Function to update bars with animation
  function updateBars(data) {
    const bars = chart.selectAll("rect")
      .data(data, d => d.category);

    // ENTER
    bars.enter()
      .append("rect")
      .attr("x", d => xScale(d.category))
      .attr("width", xScale.bandwidth())
      .attr("y", height)      // start at bottom
      .attr("height", 0)      // initial height 0
      .attr("fill", "steelblue")
      // merge enter + update
      .merge(bars)
      .transition()
      .duration(800)
      .ease(d3.easeCubicInOut)
      .attr("x", d => xScale(d.category))
      .attr("width", xScale.bandwidth())
      .attr("y", d => yScale(d.value))
      .attr("height", d => height - yScale(d.value));

    // EXIT
    bars.exit()
      .transition()
      .duration(500)
      .attr("y", height)
      .attr("height", 0)
      .remove();
  }

  // Initial render
  updateBars(data);

  // Update data on button click with random values and possible new category
  d3.select("#updateBtn").on("click", () => {
    // Randomly change values and add/remove category
    const categories = ["A", "B", "C", "D", "E"];
    // Random length data array (3 to 5 categories)
    const count = Math.floor(Math.random() * 3) + 3;
    const newData = categories.slice(0, count).map(cat => ({
      category: cat,
      value: Math.floor(Math.random() * 100)
    }));

    // Update xScale domain for new categories
    xScale.domain(newData.map(d => d.category));

    // Update x-axis
    chart.select("g")
      .call(d3.axisBottom(xScale));

    // Update bars with new data
    updateBars(newData);
  });
</script>

</body>
</html>
```

8.2.2 Animating Color Changes

You can animate color changes alongside size or position changes to emphasize updates:

```
svg.selectAll("rect")
  .data(data, d => d.category)
  .transition()
  .duration(800)
  .attr("fill", d => d.value > threshold ? "orange" : "steelblue");
```

Colors gradually shift based on the data condition, adding another layer of visual feedback.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Animating Bar Height and Color</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body { font-family: sans-serif; }
    svg { border: 1px solid #ccc; }
  </style>
</head>
<body>

<button id="updateBtn">Update Data</button>
<svg width="600" height="400"></svg>

<script>
  const svg = d3.select("svg");
  const margin = { top: 30, right: 20, bottom: 50, left: 60 };
  const width = +svg.attr("width") - margin.left - margin.right;
  const height = +svg.attr("height") - margin.top - margin.bottom;
  const threshold = 50;

  const chart = svg.append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

  // Initial data
  let data = [
    { category: "A", value: 30 },
    { category: "B", value: 80 },
    { category: "C", value: 45 },
    { category: "D", value: 60 }
  ];

  const xScale = d3.scaleBand()
    .domain(data.map(d => d.category))
    .range([0, width])
    .padding(0.2);

  const yScale = d3.scaleLinear()
    .domain([0, 100])
    .range([height, 0]);

  // Axes
  chart.append("g")
    .attr("transform", `translate(0,${height})`)
    .call(d3.axisBottom(xScale));
  chart.append("g")
    .call(d3.axisLeft(yScale));

  // Update bars function with color animation
  function updateBars(data) {
    const bars = chart.selectAll("rect")
      .data(data, d => d.category);

    // ENTER
    bars.enter()
      .append("rect")
      .attr("x", d => xScale(d.category))
      .attr("width", xScale.bandwidth())
      .attr("y", height)
      .attr("height", 0)
      .attr("fill", "steelblue")
      // merge enter + update
      .merge(bars)
      .transition()
      .duration(800)
      .ease(d3.easeCubicInOut)
      .attr("x", d => xScale(d.category))
      .attr("width", xScale.bandwidth())
      .attr("y", d => yScale(d.value))
      .attr("height", d => height - yScale(d.value))
      .attr("fill", d => d.value > threshold ? "orange" : "steelblue");

    // EXIT
    bars.exit()
      .transition()
      .duration(500)
      .attr("y", height)
      .attr("height", 0)
      .remove();
  }

  // Initial render
  updateBars(data);

  // Update data and re-render on button click
  d3.select("#updateBtn").on("click", () => {
    const categories = ["A", "B", "C", "D", "E"];
    const count = Math.floor(Math.random() * 3) + 3;
    const newData = categories.slice(0, count).map(cat => ({
      category: cat,
      value: Math.floor(Math.random() * 100)
    }));

    xScale.domain(newData.map(d => d.category));
    chart.select("g")
      .call(d3.axisBottom(xScale));

    updateBars(newData);
  });
</script>

</body>
</html>
```

8.2.3 Chaining Transitions for Coordinated Effects

Transitions can be chained to animate multiple properties sequentially or simultaneously for complex effects:

```
svg.selectAll("rect")
  .data(data)
  .transition()
  .duration(600)
  .attr("y", d => yScale(d.value))
  .attr("height", d => height - yScale(d.value))
  .transition()
  .duration(400)
  .attr("fill", "purple");
```

- First, bars smoothly resize.
- Then, color changes to purple, completing the animation in two phases.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Chained Transitions Example</title>
<script src="https://d3js.org/d3.v7.min.js"></script>
<style>
  svg { border: 1px solid #ccc; }
  body { font-family: sans-serif; }
</style>
</head>
<body>

<button id="updateBtn">Update Data</button>
<svg width="600" height="400"></svg>

<script>
  const svg = d3.select("svg");
  const margin = { top: 30, right: 20, bottom: 50, left: 60 };
  const width = +svg.attr("width") - margin.left - margin.right;
  const height = +svg.attr("height") - margin.top - margin.bottom;

  const chart = svg.append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

  // Initial data
  let data = [
    { category: "A", value: 40 },
    { category: "B", value: 80 },
    { category: "C", value: 55 },
    { category: "D", value: 60 }
```

```
];

const xScale = d3.scaleBand()
  .domain(data.map(d => d.category))
  .range([0, width])
  .padding(0.2);

const yScale = d3.scaleLinear()
  .domain([0, 100])
  .range([height, 0]);

// Axes
const xAxis = chart.append("g")
  .attr("transform", `translate(0,${height})`)
  .call(d3.axisBottom(xScale));

const yAxis = chart.append("g")
  .call(d3.axisLeft(yScale));

// Initial bars
chart.selectAll("rect")
  .data(data, d => d.category)
  .enter()
  .append("rect")
  .attr("x", d => xScale(d.category))
  .attr("width", xScale.bandwidth())
  .attr("y", height)
  .attr("height", 0)
  .attr("fill", "steelblue")
  .transition()
  .duration(600)
  .attr("y", d => yScale(d.value))
  .attr("height", d => height - yScale(d.value))
  .transition()
  .duration(400)
  .attr("fill", "purple");

// Update bars with chained transitions
function updateBars(data) {
  // Update xScale domain if categories changed
  xScale.domain(data.map(d => d.category));
  xAxis.transition().duration(500).call(d3.axisBottom(xScale));

  const bars = chart.selectAll("rect")
    .data(data, d => d.category);

  // EXIT
  bars.exit()
    .transition()
    .duration(400)
    .attr("y", height)
    .attr("height", 0)
    .remove();

  // UPDATE + ENTER
  bars.enter()
    .append("rect")
    .attr("x", d => xScale(d.category))
```

```
      .attr("width", xScale.bandwidth())
      .attr("y", height)
      .attr("height", 0)
      .attr("fill", "steelblue")
      .merge(bars)
      .transition()
      .duration(600)
      .attr("x", d => xScale(d.category))
      .attr("width", xScale.bandwidth())
      .attr("y", d => yScale(d.value))
      .attr("height", d => height - yScale(d.value))
      .transition()
      .duration(400)
      .attr("fill", "purple");
  }

  // Button to update data and trigger transitions
  d3.select("#updateBtn").on("click", () => {
    const categories = ["A", "B", "C", "D", "E"];
    const count = Math.floor(Math.random() * 3) + 3;  // 3 to 5 bars
    const newData = categories.slice(0, count).map(cat => ({
      category: cat,
      value: Math.floor(Math.random() * 100)
    }));

    updateBars(newData);
  });
</script>

</body>
</html>
```

8.2.4 Summary

- Animate height and position changes to visualize data updates smoothly.
- Use transitions on color to highlight or differentiate data dynamically.
- Chain transitions to build more elaborate, staged animations.
- Coordinated animations improve user comprehension and engagement.

By animating chart elements thoughtfully, you transform static visuals into compelling stories that flow naturally with the data.

8.3 Smooth Enter/Exit Animations

A key strength of D3 is its **enter-update-exit** pattern, which manages how elements are added, updated, or removed from the DOM when data changes. Smoothly animating these transitions enhances the user experience by visually guiding the viewer through data updates

— bars growing into view or gracefully shrinking away feel intuitive and polished.

8.3.1 The Enter-Update-Exit Pattern Recap

- **Enter**: Handles new data points by creating new elements.
- **Update**: Updates existing elements to reflect changed data.
- **Exit**: Removes elements for data points no longer present.

Animating these phases with transitions makes the changes visually continuous rather than abrupt.

8.3.2 Example: Animating Bars with Enter and Exit

Suppose you filter a dataset, which causes some bars to disappear and new ones to appear. Here's how to smoothly animate those changes:

```
function updateChart(filteredData) {
  const bars = svg.selectAll("rect")
    .data(filteredData, d => d.category);  // key function for consistency

  // ENTER: New bars start with zero height and grow up
  bars.enter()
    .append("rect")
    .attr("x", d => xScale(d.category))
    .attr("width", xScale.bandwidth())
    .attr("y", height)
    .attr("height", 0)
    .attr("fill", "teal")
    .transition()
    .duration(800)
    .attr("y", d => yScale(d.value))
    .attr("height", d => height - yScale(d.value));

  // UPDATE: Existing bars transition to new height/position
  bars.transition()
    .duration(800)
    .attr("x", d => xScale(d.category))
    .attr("y", d => yScale(d.value))
    .attr("height", d => height - yScale(d.value))
    .attr("width", xScale.bandwidth());

  // EXIT: Bars for removed data shrink and fade out before removal
  bars.exit()
    .transition()
    .duration(500)
    .attr("y", height)
    .attr("height", 0)
    .style("fill-opacity", 0)
    .remove();
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Animated Enter-Update-Exit Bar Chart</title>
<script src="https://d3js.org/d3.v7.min.js"></script>
<style>
  svg { border: 1px solid #ccc; }
  body { font-family: sans-serif; }
  button { margin: 10px; }
</style>
</head>
<body>

<button id="filterBtn">Toggle Filter</button>
<svg width="600" height="400"></svg>

<script>
  const svg = d3.select("svg");
  const margin = { top: 30, right: 20, bottom: 50, left: 60 };
  const width = +svg.attr("width") - margin.left - margin.right;
  const height = +svg.attr("height") - margin.top - margin.bottom;

  const chart = svg.append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

  const allData = [
    { category: "A", value: 40 },
    { category: "B", value: 80 },
    { category: "C", value: 55 },
    { category: "D", value: 60 },
    { category: "E", value: 20 }
  ];

  const xScale = d3.scaleBand()
    .domain(allData.map(d => d.category))
    .range([0, width])
    .padding(0.2);

  const yScale = d3.scaleLinear()
    .domain([0, d3.max(allData, d => d.value) * 1.1])
    .range([height, 0]);

  // Axes
  chart.append("g")
    .attr("transform", `translate(0,${height})`)
    .call(d3.axisBottom(xScale));

  chart.append("g")
    .call(d3.axisLeft(yScale));

  // Initial render with all data
  updateChart(allData);
  // Toggle filter on button click
  let filtered = false;
  d3.select("#filterBtn").on("click", () => {
    filtered = !filtered;
    const dataToUse = filtered
      ? allData.filter(d => d.value >= 50)  // Filter values >= 50
      : allData;
    updateChart(dataToUse);
  });

  function updateChart(filteredData) {
    const bars = chart.selectAll("rect")
      .data(filteredData, d => d.category);

    // ENTER
    bars.enter()
      .append("rect")
      .attr("x", d => xScale(d.category))
      .attr("width", xScale.bandwidth())
      .attr("y", height)
      .attr("height", 0)
      .attr("fill", "teal")
      .transition()
      .duration(800)
      .attr("y", d => yScale(d.value))
      .attr("height", d => height - yScale(d.value));

    // UPDATE
    bars.transition()
      .duration(800)
      .attr("x", d => xScale(d.category))
      .attr("y", d => yScale(d.value))
      .attr("height", d => height - yScale(d.value))
      .attr("width", xScale.bandwidth());

    // EXIT
    bars.exit()
      .transition()
      .duration(500)
      .attr("y", height)
      .attr("height", 0)
      .style("fill-opacity", 0)
      .remove();
  }
</script>

</body>
</html>
```

8.3.3 Whats Happening?

- **Entering bars** start invisible at the bottom (`y = height`, `height = 0`) and grow to their full height with a smooth animation.

- **Updating bars** smoothly move and resize to their new positions and sizes.
- **Exiting bars** shrink back down to zero height and fade out, then get removed from the DOM.

This approach creates a fluid, natural transition whenever the dataset changes, reducing visual confusion and enhancing storytelling.

8.3.4 Summary

- Use the **enter-update-exit** pattern combined with `.transition()` to animate data-driven changes.
- Animate entering elements from zero size or opacity to full size for smooth addition.
- Animate exiting elements shrinking and fading out before removal to avoid abrupt disappearance.
- Smooth transitions help users intuitively follow data changes and maintain context.

Mastering smooth enter and exit animations is crucial for building professional, polished D3 visualizations that respond gracefully to dynamic data.

8.4 Custom Timing and Easing Functions

Smooth transitions become even more compelling when you control **how** they progress over time. This is where **easing functions** come in — they define the pacing or acceleration curve of your animations, shaping their "feel" and making motions appear natural, dynamic, or playful. In this section, we'll explore common easing functions in D3 and when to use them.

8.4.1 What Are Easing Functions?

An easing function controls the interpolation rate during a transition, determining whether the animation moves at a constant speed, accelerates, decelerates, or bounces. D3 provides many easing presets in its `d3.ease*` namespace.

You apply an easing function to a transition with `.ease()`:

```
selection.transition()
  .duration(1000)
  .ease(d3.easeCubic)    // apply easing function
  .attr("attribute", newValue);
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>D3 Easing Functions Demo</title>
<script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
<style>
  svg {
    border: 1px solid #ccc;
  }
  text {
    font-family: sans-serif;
    font-size: 12px;
    text-anchor: middle;
  }
</style>
</head>
<body>

<svg width="700" height="200"></svg>

<script>
const svg = d3.select("svg");
const width = +svg.attr("width");
const height = +svg.attr("height");

const easingFunctions = [
  { name: "easeLinear", fn: d3.easeLinear },
  { name: "easeCubic", fn: d3.easeCubic },
  { name: "easeBounce", fn: d3.easeBounce },
  { name: "easeElastic", fn: d3.easeElastic },
  { name: "easeBack", fn: d3.easeBack }
];

const margin = 50;
const spacing = (width - 2 * margin) / easingFunctions.length;
const circleRadius = 15;

easingFunctions.forEach((easing, i) => {
  const x = margin + i * spacing + spacing / 2;

  // Draw the label
  svg.append("text")
    .attr("x", x)
    .attr("y", margin / 2)
    .text(easing.name);

  // Draw initial circle at top
  const circle = svg.append("circle")
    .attr("cx", x)
    .attr("cy", margin)
    .attr("r", circleRadius)
    .attr("fill", "steelblue");

  // Animate downward with easing
  circle.transition()
    .duration(2000)
    .ease(easing.fn)
    .attr("cy", height - margin);
});
</script>

</body>
</html>
```

8.4.2 Common Easing Functions and Their Effects

Easing Function	Description	Best Use Cases
`easeLinear`	Constant speed from start to finish	Simple or mechanical animations
`easeCubic`	Smooth acceleration and deceleration	Natural, organic movement
`easeBounce`	Ends with a bouncing effect	Playful or attention-grabbing effects
`easeElastic`	Oscillates with overshoot and rebound	Fun, springy motions
`easeBack`	Slight overshoot past the target before settling	Emphasizes motion arrival

8.4.3 Visualizing Easing Effects

Imagine animating a bar growing to its new height over 1 second.

```
svg.selectAll("rect")
  .data(data)
  .transition()
  .duration(1000)
  .ease(d3.easeLinear)   // steady, mechanical growth
  .attr("height", d => height - yScale(d.value));
```

This will produce a steady, uniform growth—good for data updates where a neutral feel is desired.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>D3 Easing on Bar Growth</title>
<script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
<style>
  svg {
    border: 1px solid #ccc;
  }
  text {
    font-family: sans-serif;
    font-size: 12px;
    text-anchor: middle;
  }
</style>
</head>
<body>

<svg width="700" height="200"></svg>

<script>
const svg = d3.select("svg");
const width = +svg.attr("width");
const height = +svg.attr("height");

const margin = { top: 40, right: 20, bottom: 40, left: 60 };
const chartHeight = height - margin.top - margin.bottom;
const chartWidth = width - margin.left - margin.right;

const easingFunctions = [
  { name: "easeLinear", fn: d3.easeLinear },
  { name: "easeCubic", fn: d3.easeCubic },
  { name: "easeBounce", fn: d3.easeBounce },
  { name: "easeElastic", fn: d3.easeElastic },
  { name: "easeBack", fn: d3.easeBack }
];

const barWidth = 40;
const spacing = (chartWidth - easingFunctions.length * barWidth) / (easingFunctions.length + 1);

const maxValue = 100;
const yScale = d3.scaleLinear()
  .domain([0, maxValue])
  .range([chartHeight, 0]);

// Add group for chart content
const chart = svg.append("g")
  .attr("transform", `translate(${margin.left},${margin.top})`);

// Draw baseline
chart.append("line")
  .attr("x1", 0)
  .attr("x2", chartWidth)
  .attr("y1", chartHeight)
  .attr("y2", chartHeight)
  .attr("stroke", "#000");

// Add labels and bars
easingFunctions.forEach((easing, i) => {
  const x = spacing + i * (barWidth + spacing);

  // Label
  chart.append("text")
    .attr("x", x + barWidth / 2)
    .attr("y", -10)
    .text(easing.name)
    .attr("text-anchor", "middle");

  // Initial bar (height 0)
  const bar = chart.append("rect")
    .attr("x", x)
    .attr("y", chartHeight)
    .attr("width", barWidth)
    .attr("height", 0)
    .attr("fill", "steelblue");

  // Animate bar growing to height representing value 70
  bar.transition()
    .duration(1000)
    .ease(easing.fn)
    .attr("y", yScale(70))
    .attr("height", chartHeight - yScale(70));
});
</script>

</body>
</html>
```

8.4.4 Smooth and Natural with `easeCubic`

```
.transition()
.duration(1000)
.ease(d3.easeCubic)
.attr("height", d => height - yScale(d.value));
```

The bar starts slowly, speeds up, then slows as it reaches its final height — this feels more natural and pleasing.

8.4.5 Adding Playfulness with `easeBounce`

```
.transition()
.duration(1000)
.ease(d3.easeBounce)
.attr("height", d => height - yScale(d.value));
```

The bar “bounces” slightly at the end, adding energy and a fun effect, great for informal or playful visuals.

8.4.6 When to Use Different Easing Functions

- Use **`easeLinear`** for straightforward, no-nonsense animations, such as real-time updates where clarity and immediacy are key.
- Use **`easeCubic`** (or other smooth easing) for polished, professional transitions that feel fluid and natural.
- Use **`easeBounce`** or **`easeElastic`** sparingly to highlight special events or create playful interactions, but avoid overusing them in serious or dense data contexts.
- For complex UI motion (menus, tooltips), easing functions like **`easeBack`** add a touch of polish with overshoot.

8.4.7 Summary

- Easing functions control the acceleration and pacing of animations, shaping their emotional impact.
- D3 includes a rich set of easing options like `easeLinear`, `easeCubic`, `easeBounce`, and more.
- Choose easing thoughtfully based on the message and tone of your visualization.
- Combine duration and easing for fully customized, compelling motion design.

Using easing well transforms transitions from simple property changes into meaningful, expressive animations that engage your audience.

Chapter 9.

Responsive and Adaptive Design

9 Responsive and Adaptive Design

9.1 Making Charts Fit Any Screen

In today's multi-device world, data visualizations need to look great and function correctly on everything from large desktop monitors to mobile phones. Responsive design ensures your charts scale and adapt to various screen sizes and containers, enhancing accessibility and usability for all users.

9.1.1 Why Responsiveness Matters

Users expect websites and dashboards to work on any screen. A fixed-size D3 chart may overflow on small devices or appear tiny on high-resolution screens. Making your visualizations responsive helps with:

- **Accessibility**: Viewable on phones, tablets, and desktops.
- **Layout integration**: Charts fit neatly inside flexible grids or resizable panels.
- **Maintainability**: Reduces need for separate visualizations per device type.

9.1.2 Step 1: Use Dynamic Dimensions

Instead of hardcoding chart width and height, use the size of the container to determine the chart's dimensions.

```html
<div id="chart-container" style="width: 100%; height: 400px;"></div>
```

```javascript
const container = d3.select("#chart-container");
const width = parseInt(container.style("width"));
const height = parseInt(container.style("height"));

// Create responsive SVG
const svg = container.append("svg")
  .attr("width", width)
  .attr("height", height);
```

This ensures the chart fills the available space and can adapt if the container resizes.

9.1.3 Step 2: Listen for Resize Events

You can respond to window resizes by redrawing the chart:

```javascript
window.addEventListener("resize", () => {
  // Clear and re-render
  d3.select("#chart-container svg").remove();
  drawChart();  // call your chart drawing function again
});
```

This pattern ensures that the chart always fits its container, even when the window changes size.

9.1.4 Step 3: Combine with ViewBox for SVG Scaling

Using the `viewBox` and `preserveAspectRatio` attributes allows the SVG to scale proportionally:

```
const svg = container.append("svg")
  .attr("viewBox", `0 0 ${width} ${height}`)
  .attr("preserveAspectRatio", "xMidYMid meet")
  .style("width", "100%")
  .style("height", "auto");
```

This method allows the SVG to scale responsively while maintaining its aspect ratio.

9.1.5 Example: Responsive Bar Chart in a Resizable Container

```
<div id="bar-chart" style="width: 100%; height: 300px;"></div>
```

```
function drawBarChart(data) {
  const container = d3.select("#bar-chart");
  const width = parseInt(container.style("width"));
  const height = parseInt(container.style("height"));

  const svg = container.append("svg")
    .attr("viewBox", `0 0 ${width} ${height}`)
    .attr("preserveAspectRatio", "xMidYMid meet")
    .style("width", "100%")
    .style("height", "auto");

  const x = d3.scaleBand()
    .domain(data.map(d => d.label))
    .range([0, width])
    .padding(0.1);

  const y = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.value)])
    .range([height, 0]);

  svg.selectAll("rect")
    .data(data)
    .join("rect")
    .attr("x", d => x(d.label))
    .attr("y", d => y(d.value))
    .attr("width", x.bandwidth())
    .attr("height", d => height - y(d.value))
    .attr("fill", "steelblue");
}

// Sample data
const sampleData = [
  { label: "A", value: 30 },
  { label: "B", value: 80 },
  { label: "C", value: 45 },
  { label: "D", value: 60 },
];

drawBarChart(sampleData);

// Redraw on resize
window.addEventListener("resize", () => {
  d3.select("#bar-chart svg").remove();
  drawBarChart(sampleData);
});
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Responsive Bar Chart</title>
<script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
<style>
  #bar-chart {
    width: 80%;
    max-width: 700px;
    height: 300px;
    margin: 20px auto;
    border: 1px solid #ccc;
  }
</style>
</head>
<body>

<div id="bar-chart"></div>

<script>
function drawBarChart(data) {
  const container = d3.select("#bar-chart");
  const width = parseInt(container.style("width"));
  const height = parseInt(container.style("height"));

  const svg = container.append("svg")
    .attr("viewBox", `0 0 ${width} ${height}`)
    .attr("preserveAspectRatio", "xMidYMid meet")
    .style("width", "100%")
    .style("height", "auto");

  const margin = {top: 20, right: 20, bottom: 40, left: 40};
  const innerWidth = width - margin.left - margin.right;
  const innerHeight = height - margin.top - margin.bottom;

  const x = d3.scaleBand()
    .domain(data.map(d => d.label))
    .range([0, innerWidth])
    .padding(0.1);

  const y = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.value)])
    .nice()
    .range([innerHeight, 0]);

  const chart = svg.append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

  // Axes
  chart.append("g")
    .attr("transform", `translate(0,${innerHeight})`)
    .call(d3.axisBottom(x));

  chart.append("g")
    .call(d3.axisLeft(y));

  // Bars
  chart.selectAll("rect")
    .data(data)
    .join("rect")
    .attr("x", d => x(d.label))
    .attr("y", d => y(d.value))
    .attr("width", x.bandwidth())
    .attr("height", d => innerHeight - y(d.value))
    .attr("fill", "steelblue");
}

// Sample data
const sampleData = [
  { label: "A", value: 30 },
  { label: "B", value: 80 },
  { label: "C", value: 45 },
  { label: "D", value: 60 },
];

drawBarChart(sampleData);

// Redraw on resize (debounced for performance)
let resizeTimeout;
window.addEventListener("resize", () => {
  clearTimeout(resizeTimeout);
  resizeTimeout = setTimeout(() => {
    d3.select("#bar-chart svg").remove();
    drawBarChart(sampleData);
  }, 200);
});
</script>

</body>
</html>
```

9.1.6 Summary

- Responsive charts improve user experience on all screen sizes.
- Use dynamic container-based sizing instead of fixed pixel values.
- Leverage SVG's `viewBox` and `preserveAspectRatio` for scalable visuals.
- Add resize listeners to redraw charts when the window changes.

Responsive design is not just a bonus—it's a necessity for modern, accessible data visualizations.

9.2 Recalculating Scales on Resize

Making your D3 charts responsive involves more than just resizing the SVG — you must also **recalculate the scales** and **redraw chart elements** to reflect the new layout. This ensures that axes, bars, circles, and lines continue to align with their data after a screen resize or layout change.

9.2.1 Why Recalculate Scales?

D3 scales map data values to pixel positions. When the chart's dimensions change (e.g., from 800px to 400px wide), those mappings need to be updated. Otherwise, the chart elements may appear misaligned, distorted, or cut off.

9.2.2 Detecting Resize Events

You can listen for changes to the window or container size using JavaScript's built-in `resize` event:

```
window.addEventListener("resize", () => {
  // Logic to resize/redraw the chart
});
```

When a resize is detected, clear the old chart and redraw it with new dimensions and recomputed scales.

9.2.3 Example: Resizable Bar Chart

Let's walk through a full example that:

1. Draws a bar chart based on the container width.
2. Listens for resize events.
3. Recalculates scales and redraws everything accordingly.

```html
<div id="responsive-bar" style="width: 100%; height: 300px;"></div>
```

```javascript
const data = [
  { label: "A", value: 40 },
  { label: "B", value: 80 },
  { label: "C", value: 55 },
  { label: "D", value: 70 },
];

function drawBarChart() {
  const container = d3.select("#responsive-bar");
  const width = parseInt(container.style("width"));
  const height = parseInt(container.style("height"));
  const margin = { top: 20, right: 20, bottom: 30, left: 40 };
  const innerWidth = width - margin.left - margin.right;
  const innerHeight = height - margin.top - margin.bottom;

  // Clear previous content
  container.selectAll("svg").remove();

  const svg = container.append("svg")
    .attr("width", width)
    .attr("height", height);

  const chart = svg.append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

  const x = d3.scaleBand()
    .domain(data.map(d => d.label))
    .range([0, innerWidth])
    .padding(0.1);

  const y = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.value)])
    .range([innerHeight, 0]);

  // Draw axes
  chart.append("g")
    .attr("transform", `translate(0,${innerHeight})`)
    .call(d3.axisBottom(x));

  chart.append("g")
    .call(d3.axisLeft(y));

  // Draw bars
  chart.selectAll("rect")
    .data(data)
    .join("rect")
    .attr("x", d => x(d.label))
    .attr("y", d => y(d.value))
    .attr("width", x.bandwidth())
    .attr("height", d => innerHeight - y(d.value))
    .attr("fill", "steelblue");
}
```

```
// Initial render
drawBarChart();

// Redraw chart on window resize
window.addEventListener("resize", drawBarChart);
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Resizable Bar Chart Example</title>
<script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
<style>
  /* Optional: center container and give it a border */
  #responsive-bar {
    width: 80%;
    max-width: 700px;
    height: 300px;
    margin: 30px auto;
    border: 1px solid #ccc;
  }
</style>
</head>
<body>

<div id="responsive-bar"></div>

<script>
const data = [
  { label: "A", value: 40 },
  { label: "B", value: 80 },
  { label: "C", value: 55 },
  { label: "D", value: 70 },
];

function drawBarChart() {
  const container = d3.select("#responsive-bar");
  const width = parseInt(container.style("width"));
  const height = parseInt(container.style("height"));
  const margin = { top: 20, right: 20, bottom: 30, left: 40 };
  const innerWidth = width - margin.left - margin.right;
  const innerHeight = height - margin.top - margin.bottom;

  // Clear previous SVG if any
  container.selectAll("svg").remove();

  const svg = container.append("svg")
    .attr("width", width)
    .attr("height", height);

  const chart = svg.append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

  const x = d3.scaleBand()
    .domain(data.map(d => d.label))
    .range([0, innerWidth])
    .padding(0.1);

  const y = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.value)])
    .range([innerHeight, 0]);

  // Draw axes
  chart.append("g")
    .attr("transform", `translate(0,${innerHeight})`)
    .call(d3.axisBottom(x));

  chart.append("g")
    .call(d3.axisLeft(y));

  // Draw bars
  chart.selectAll("rect")
    .data(data)
    .join("rect")
    .attr("x", d => x(d.label))
    .attr("y", d => y(d.value))
    .attr("width", x.bandwidth())
    .attr("height", d => innerHeight - y(d.value))
    .attr("fill", "steelblue");
}

// Initial render
drawBarChart();

// Redraw chart on window resize (debounced for performance)
let resizeTimeout;
window.addEventListener("resize", () => {
  clearTimeout(resizeTimeout);
  resizeTimeout = setTimeout(drawBarChart, 200);
});
</script>

</body>
</html>
```

9.2.4 Whats Happening?

- On each resize, the chart's width is recalculated from the container's current size.
- All scales and axes are recomputed using the new width/height.
- The previous SVG is cleared to prevent duplication.
- The chart is redrawn with up-to-date dimensions and positioning.

9.2.5 Summary

- Use `window.addEventListener('resize')` to detect layout changes.
- Recompute scale domains and ranges to match new dimensions.
- Redraw axes, shapes, and labels using updated scales.
- Clearing and redrawing ensures consistent, pixel-perfect layouts.

Recalculating scales on resize ensures your visualizations remain **accurate, responsive, and adaptive**, regardless of the screen or container they're displayed in.

9.3 Using `viewBox` and `preserveAspectRatio`

SVG (Scalable Vector Graphics) is inherently resolution-independent, making it ideal for creating responsive data visualizations. Two attributes — `viewBox` and `preserveAspectRatio` — are key to making SVG charts scale cleanly across devices and container sizes.

9.3.1 What Is `viewBox`?

The `viewBox` attribute defines the coordinate system for all drawing inside an SVG. It takes four values:

```
<svg viewBox="minX minY width height">
```

- `minX` and `minY` — the top-left corner of the viewable area.
- `width` and `height` — the dimensions of the internal drawing area.

Think of it as a "window" through which you look at your chart, regardless of the actual rendered size.

9.3.2 What Is `preserveAspectRatio`?

The `preserveAspectRatio` attribute controls how the SVG scales to fit its container:

```
<svg preserveAspectRatio="xMidYMid meet">
```

- `xMidYMid` — centers the content horizontally and vertically.
- `meet` — scales the graphic uniformly to fit inside the container without distortion.

You can tweak the values for different behaviors (e.g., `xMinYMin slice` to crop instead of fit).

9.3.3 Benefits of Using `viewBox` and `preserveAspectRatio`

- **Scales with screen size**: No need to recalculate dimensions manually.
- **Resolution-independent**: Looks sharp on retina and high-DPI displays.
- **Works with CSS sizing**: You can use `%`, `em`, or `vw` units for styling.

9.3.4 Example: Resizable Bar Chart Using `viewBox`

This example shows how to use `viewBox` and `preserveAspectRatio` to create a chart that automatically resizes without losing quality.

```
<div id="chart-container" style="width: 100%; max-width: 600px;"></div>
```

```
const data = [
  { label: "A", value: 40 },
  { label: "B", value: 60 },
  { label: "C", value: 80 },
  { label: "D", value: 50 },
];

function drawChart() {
  const width = 600;
  const height = 300;
  const margin = { top: 20, right: 20, bottom: 30, left: 40 };
  const innerWidth = width - margin.left - margin.right;
  const innerHeight = height - margin.top - margin.bottom;

  const svg = d3.select("#chart-container")
    .append("svg")
    .attr("viewBox", `0 0 ${width} ${height}`)
    .attr("preserveAspectRatio", "xMidYMid meet")
    .style("width", "100%")
    .style("height", "auto");

  const g = svg.append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

  const x = d3.scaleBand()
    .domain(data.map(d => d.label))
    .range([0, innerWidth])
    .padding(0.1);

  const y = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.value)])
    .nice()
    .range([innerHeight, 0]);

  g.append("g")
    .attr("transform", `translate(0,${innerHeight})`)
    .call(d3.axisBottom(x));

  g.append("g")
    .call(d3.axisLeft(y));
  g.selectAll("rect")
    .data(data)
    .join("rect")
    .attr("x", d => x(d.label))
    .attr("y", d => y(d.value))
    .attr("width", x.bandwidth())
    .attr("height", d => innerHeight - y(d.value))
    .attr("fill", "steelblue");
}

drawChart();
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>Resizable Bar Chart with viewBox</title>
<script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
<style>
  #chart-container {
    width: 100%;
    max-width: 600px;
    margin: 30px auto;
    border: 1px solid #ccc;
  }
</style>
</head>
<body>

<div id="chart-container"></div>

<script>
const data = [
  { label: "A", value: 40 },
  { label: "B", value: 60 },
  { label: "C", value: 80 },
  { label: "D", value: 50 },
];

function drawChart() {
  const width = 600;
  const height = 300;
  const margin = { top: 20, right: 20, bottom: 30, left: 40 };
  const innerWidth = width - margin.left - margin.right;
  const innerHeight = height - margin.top - margin.bottom;

  // Clear previous svg if any
  d3.select("#chart-container").selectAll("svg").remove();

  const svg = d3.select("#chart-container")
    .append("svg")
    .attr("viewBox", `0 0 ${width} ${height}`)
    .attr("preserveAspectRatio", "xMidYMid meet")
    .style("width", "100%")
    .style("height", "auto");
  const g = svg.append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

  const x = d3.scaleBand()
    .domain(data.map(d => d.label))
    .range([0, innerWidth])
    .padding(0.1);

  const y = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.value)])
    .nice()
    .range([innerHeight, 0]);

  g.append("g")
    .attr("transform", `translate(0,${innerHeight})`)
    .call(d3.axisBottom(x));

  g.append("g")
    .call(d3.axisLeft(y));

  g.selectAll("rect")
    .data(data)
    .join("rect")
    .attr("x", d => x(d.label))
    .attr("y", d => y(d.value))
    .attr("width", x.bandwidth())
    .attr("height", d => innerHeight - y(d.value))
    .attr("fill", "steelblue");
}

drawChart();
</script>

</body>
</html>
```

9.3.5 How It Works

- **`viewBox`** ensures that the drawing space scales to any container size.
- **`preserveAspectRatio="xMidYMid meet"`** keeps the chart centered and proportionally scaled.
- **CSS styles** (`width: 100%; height: auto`) make the SVG stretch with the container.

No need to listen for resize events or recalculate sizes — the browser handles scaling automatically.

9.3.6 Summary

- Use `viewBox` to define scalable SVG coordinate systems.
- Use `preserveAspectRatio` to control how the chart scales and aligns.
- Combine these with responsive CSS (`width: 100%`) for adaptive, pixel-perfect visualizations.
- This technique works especially well for static or lightly interactive charts.

Mastering `viewBox` and `preserveAspectRatio` is essential for building modern, responsive D3 visualizations that look great everywhere.

9.4 Mobile-First Visualization Tips

Designing data visualizations for mobile devices brings unique challenges. Small screens, touch interaction, limited bandwidth, and diverse device capabilities demand a different approach from desktop-first design. A mobile-first mindset ensures that your D3 charts are usable, legible, and performant on the smallest screens — and scale gracefully to larger ones.

9.4.1 Why Mobile-First?

Mobile-first design begins with optimizing for the smallest screens and then enhancing the experience for larger ones. This leads to cleaner, more focused visualizations that work universally.

Mobile-first D3 charts should:

- Emphasize **clarity over complexity**.
- Prioritize **touch interactions**.
- Minimize **screen clutter**.

9.4.2 Design Principles for Mobile D3 Charts

Simplify the Visual Layout

- **Remove unnecessary chart elements** such as gridlines or excessive ticks.
- Use **flat, bold color palettes** for faster recognition.
- Avoid complex multi-axis layouts unless absolutely necessary.

YES *Example*: Instead of multiple grouped bar charts, use a simple horizontal bar chart that scrolls vertically.

Enlarge Touch Targets

- Make sure interactive elements like bars, dots, or buttons are at least **40×40 pixels**.
- Add invisible padding using `pointer-events: all` on larger hit areas.

```
svg.selectAll("circle")
  .attr("r", 6)
  .style("pointer-events", "visible")
  .on("touchstart", d => showTooltip(d));
```

YES *Tip*: Invisible `rect` overlays can improve usability without altering appearance.

Adapt Labels and Legends

- **Rotate or hide axis labels** if they don't fit.
- **Abbreviate or truncate long text** on mobile.
- Use **legend toggles** or collapsible panels to save space.

```
if (window.innerWidth < 400) {
  xAxis.tickFormat(d => d.slice(0, 3));  // Shorten month names, e.g., "Jan"
}
```

YES *Example*: Move a horizontal legend to a vertical sidebar on narrow viewports.

Streamline Tooltips

- Use a **single tooltip `<div>`** that floats and follows touch/mouse.
- Avoid tooltip content that's too long or wide.
- Use responsive styles with readable fonts and padding.

```
tooltip.style("left", `${event.pageX + 10}px`)
       .style("top", `${event.pageY + 10}px`)
       .html(`<strong>${d.label}</strong><br>${d.value}`);
```

YES *Mobile tip*: Trigger tooltip on `touchstart` instead of `mouseover`.

Minimize Animation and Complexity

- Avoid long or complex transitions on mobile.
- Limit the number of simultaneously animated elements.
- Use simpler layouts: favor bar, donut, or line charts over scatter plots with thousands of points.

YES *Example*: On small screens, reduce animation duration and disable chart zooming to improve performance.

9.4.3 Responsive Design Pattern for D3 Mobile Charts

1. **Detect screen width** with JavaScript or CSS.
2. **Adjust chart type or behavior** conditionally.
3. **Use `viewBox`, flexible scales, and `resize` listeners** to adapt layout.

```
const isMobile = window.innerWidth < 500;
const margin = isMobile
  ? { top: 10, right: 10, bottom: 40, left: 30 }
  : { top: 20, right: 20, bottom: 60, left: 50 };
```

9.4.4 Summary: Mobile D3 Design Checklist

Feature	Mobile-First Best Practice
Layout	Reduce complexity, prefer simple charts
Touch targets	40×40px, add invisible overlays if needed
Labels & Legends	Abbreviate, truncate, or reposition
Tooltips	Use compact, responsive HTML tooltips
Interaction	Simplify gestures, avoid zoom/pan where unnecessary
Performance	Reduce animation complexity and count

By following these tips, you'll create D3 visualizations that feel native on mobile devices — fast, intuitive, and accessible — without sacrificing clarity or impact.

Chapter 10.

Advanced Visualizations

1. Force-Directed Graphs
2. Choropleth and Geo Maps
3. Radial and Polar Charts
4. Hierarchical Data (TreeMaps, Sunburst, Dendrograms)

10 Advanced Visualizations

10.1 Force-Directed Graphs

Force-directed graphs are a powerful way to visualize **networks** — collections of nodes (entities) and links (relationships). Unlike grid-based charts, these layouts use simulated physical forces to position elements dynamically, helping reveal underlying structure and clustering patterns in data.

10.1.1 What Is a Force-Directed Graph?

A **force-directed graph** uses a physics-based simulation to arrange nodes and links automatically. D3's `d3.forceSimulation` module handles this by applying **forces** like attraction, repulsion, gravity, and collision between nodes.

Each node "moves" according to these forces until the system stabilizes — resulting in an organic, readable layout.

10.1.2 Core Forces in D3

D3's simulation supports modular forces you can combine to influence layout:

Force Name	Purpose
`d3.forceLink()`	Keeps linked nodes at a fixed distance.
`d3.forceCenter()`	Pulls the whole graph toward the center of view.
`d3.forceManyBody()`	Attracts or repels nodes (default is repulsion).
`d3.forceCollide()`	Prevents nodes from overlapping.

You combine these forces with `d3.forceSimulation()`.

10.1.3 Example: Simple Social Network

Let's create a basic network of people connected as friends.

Sample Data

```
const nodes = [
  { id: "Alice" },
  { id: "Bob" },
  { id: "Carol" },
  { id: "Dave" },
  { id: "Eve" },
];

const links = [
  { source: "Alice", target: "Bob" },
  { source: "Alice", target: "Carol" },
  { source: "Bob", target: "Dave" },
  { source: "Carol", target: "Eve" },
];
```

Force-Directed Graph Code

```
<svg width="600" height="400"></svg>
```

```
const svg = d3.select("svg");
const width = +svg.attr("width");
const height = +svg.attr("height");

// Create simulation
const simulation = d3.forceSimulation(nodes)
  .force("link", d3.forceLink(links).id(d => d.id).distance(100))
  .force("charge", d3.forceManyBody().strength(-200))
  .force("center", d3.forceCenter(width / 2, height / 2))
  .force("collision", d3.forceCollide(40)); // Prevent overlap

// Draw lines (links)
const link = svg.append("g")
  .attr("stroke", "#aaa")
  .selectAll("line")
  .data(links)
  .join("line");

// Draw circles (nodes)
const node = svg.append("g")
  .attr("stroke", "#fff")
  .attr("stroke-width", 1.5)
  .selectAll("circle")
  .data(nodes)
  .join("circle")
  .attr("r", 20)
  .attr("fill", "steelblue")
  .call(drag(simulation));

// Add labels
const label = svg.append("g")
  .attr("font-size", 12)
  .attr("text-anchor", "middle")
  .selectAll("text")
  .data(nodes)
  .join("text")
  .text(d => d.id)
  .attr("dy", 4);

// Update positions each tick
```

```
simulation.on("tick", () => {
  link
    .attr("x1", d => d.source.x)
    .attr("y1", d => d.source.y)
    .attr("x2", d => d.target.x)
    .attr("y2", d => d.target.y);

  node
    .attr("cx", d => d.x)
    .attr("cy", d => d.y);

  label
    .attr("x", d => d.x)
    .attr("y", d => d.y);
});

// Enable drag interaction
function drag(simulation) {
  return d3.drag()
    .on("start", (event, d) => {
      if (!event.active) simulation.alphaTarget(0.3).restart();
      d.fx = d.x;
      d.fy = d.y;
    })
    .on("drag", (event, d) => {
      d.fx = event.x;
      d.fy = event.y;
    })
    .on("end", (event, d) => {
      if (!event.active) simulation.alphaTarget(0);
      d.fx = null;
      d.fy = null;
    });
}
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Simple Social Network</title>
  <script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
  <style>
    body {
      font-family: sans-serif;
    }
    svg {
      border: 1px solid #ccc;
    }
  </style>
</head>
<body>

<svg width="600" height="400"></svg>

<script>
```

```
// Sample data
const nodes = [
  { id: "Alice" },
  { id: "Bob" },
  { id: "Carol" },
  { id: "Dave" },
  { id: "Eve" },
];

const links = [
  { source: "Alice", target: "Bob" },
  { source: "Alice", target: "Carol" },
  { source: "Bob", target: "Dave" },
  { source: "Carol", target: "Eve" },
];

// Set up SVG and simulation
const svg = d3.select("svg");
const width = +svg.attr("width");
const height = +svg.attr("height");

const simulation = d3.forceSimulation(nodes)
  .force("link", d3.forceLink(links).id(d => d.id).distance(100))
  .force("charge", d3.forceManyBody().strength(-200))
  .force("center", d3.forceCenter(width / 2, height / 2))
  .force("collision", d3.forceCollide(40));

// Draw links
const link = svg.append("g")
  .attr("stroke", "#aaa")
  .selectAll("line")
  .data(links)
  .join("line");

// Draw nodes
const node = svg.append("g")
  .attr("stroke", "#fff")
  .attr("stroke-width", 1.5)
  .selectAll("circle")
  .data(nodes)
  .join("circle")
  .attr("r", 20)
  .attr("fill", "steelblue")
  .call(drag(simulation));

// Draw labels
const label = svg.append("g")
  .attr("font-size", 12)
  .attr("text-anchor", "middle")
  .selectAll("text")
  .data(nodes)
  .join("text")
  .text(d => d.id)
  .attr("dy", 4);

// Tick update
simulation.on("tick", () => {
  link
```

```
    .attr("x1", d => d.source.x)
    .attr("y1", d => d.source.y)
    .attr("x2", d => d.target.x)
    .attr("y2", d => d.target.y);

  node
    .attr("cx", d => d.x)
    .attr("cy", d => d.y);

  label
    .attr("x", d => d.x)
    .attr("y", d => d.y);
});

// Drag behavior
function drag(simulation) {
  return d3.drag()
    .on("start", (event, d) => {
      if (!event.active) simulation.alphaTarget(0.3).restart();
      d.fx = d.x;
      d.fy = d.y;
    })
    .on("drag", (event, d) => {
      d.fx = event.x;
      d.fy = event.y;
    })
    .on("end", (event, d) => {
      if (!event.active) simulation.alphaTarget(0);
      d.fx = null;
      d.fy = null;
    });
}
</script>

</body>
</html>
```

10.1.4 Whats Happening

- **Nodes** represent people.
- **Links** represent friendships.
- The simulation uses:
  - `forceLink` to connect friends.
  - `forceCharge` to push nodes apart.
  - `forceCenter` to keep everything centered.
  - `forceCollide` to avoid overlap.
- Users can **drag nodes** to explore the network.

10.1.5 Additional Tips

- You can encode **node size** by importance (e.g., number of connections).
- Use **color** to group nodes by category or cluster.
- **Zoom and pan** can be added for large networks.
- For large graphs, consider **WebGL rendering** or **progressive layout updates**.

10.1.6 Summary

Force-directed graphs offer an intuitive way to visualize relationships and clusters. With `d3.forceSimulation` and a few basic forces, you can create dynamic, interactive network diagrams that help uncover structure in connected data.

Feature	Description
`d3.forceLink`	Keeps connected nodes at specified distance
`d3.forceManyBody`	Repels or attracts nodes
`d3.forceCenter`	Keeps layout centered
`d3.forceCollide`	Prevents overlapping nodes

Force-directed layouts are ideal for visualizing social networks, citation graphs, internet topology, and more.

10.2 Choropleth and Geo Maps

Geographic visualizations bring spatial data to life, allowing users to immediately recognize patterns tied to regions, countries, or coordinates. In D3, geo maps — especially **choropleths** (maps where regions are shaded based on data values) — are built using a combination of geospatial data formats, projections, and D3's path rendering capabilities.

10.2.1 Geographic Data: GeoJSON and TopoJSON

D3 works natively with **GeoJSON**, a standard format for encoding geographic features. Each feature (like a country or state) is represented with a geometry (shape) and associated properties.

Example GeoJSON snippet:

```
{
  "type": "Feature",
  "properties": { "name": "France" },
  "geometry": {
    "type": "Polygon",
    "coordinates": [[[2.0, 48.9], [3.0, 48.8], ...]]
  }
}
```

TopoJSON is a compressed alternative to GeoJSON that reduces file size by encoding shared boundaries. D3 can convert TopoJSON into GeoJSON using `topojson.feature()` (you'll need the topojson-client package).

10.2.2 Projections and Paths

Since the Earth is spherical, we need a **projection** to flatten coordinates onto a 2D screen. D3 supports many projections — the most common is `d3.geoMercator()`.

```
const projection = d3.geoMercator()
  .scale(120)
  .translate([width / 2, height / 2]);

const path = d3.geoPath().projection(projection);
```

The `d3.geoPath()` function converts geographic shapes into SVG `d` attributes using the selected projection.

10.2.3 Example: Choropleth Map of World Population

Let's build a world map shaded by population using GeoJSON and D3 scales.

Required Data

1. **World GeoJSON** (e.g., from https://geojson-maps.ash.ms)
2. **Population dataset** with country codes and values:

```
const populationById = {
  USA: 331000000,
  CHN: 1444000000,
  IND: 1393000000,
  FRA: 67000000,
  ...
};
```

Basic Setup

```
<svg width="960" height="500"></svg>
```

```
const svg = d3.select("svg");
const width = +svg.attr("width");
const height = +svg.attr("height");

const projection = d3.geoMercator()
  .scale(150)
  .translate([width / 2, height / 1.5]);

const path = d3.geoPath().projection(projection);

// Define color scale
const color = d3.scaleSequential()
  .domain([0, 1500000000]) // up to 1.5 billion
  .interpolator(d3.interpolateBlues);

// Load GeoJSON
d3.json("world.geojson").then(geoData => {
  svg.selectAll("path")
    .data(geoData.features)
    .join("path")
    .attr("d", path)
    .attr("fill", d => {
      const id = d.properties.ISO_A3; // use ISO 3-letter country code
      const pop = populationById[id];
      return pop ? color(pop) : "#ccc";
    })
    .attr("stroke", "#333")
    .attr("stroke-width", 0.5)
    .on("mouseover", (event, d) => {
      const country = d.properties.ADMIN;
      const id = d.properties.ISO_A3;
      const pop = populationById[id] || "N/A";
      tooltip.html(`<strong>${country}</strong><br>Population: ${pop}`)
             .style("left", event.pageX + "px")
             .style("top", event.pageY + "px")
             .style("opacity", 1);
    })
    .on("mouseout", () => {
      tooltip.style("opacity", 0);
    });
});
```

10.2.4 Tip: Add a Legend

Use `d3.scaleSequential().ticks()` and an SVG gradient to create a legend showing the color range and associated population values.

10.2.5 Using TopoJSON?

If your map is in TopoJSON format, you can convert it like this (with topojson-client installed):

```
import { feature } from "topojson-client";

d3.json("world.topojson").then(topology => {
  const geoData = feature(topology, topology.objects.countries);
  // Proceed as before...
});
```

10.2.6 Summary

Feature	Purpose
`GeoJSON` / `TopoJSON`	Provide regional shapes
`d3.geoProjection()`	Transforms latitude/longitude to screen
`d3.geoPath()`	Converts geometry into SVG paths
`d3.scaleSequential()`	Maps data values to colors

Choropleth maps offer a compelling way to compare metrics across regions — from population and GDP to environmental or health indicators. With D3's projection and path tools, it's easy to bind your data to the map and make it interactive and informative.

10.3 Radial and Polar Charts

D3 allows you to go beyond standard Cartesian layouts by tapping into **radial** and **polar** coordinate systems. These formats enable unique and expressive data visualizations — especially when representing cyclical data, periodic trends, or multidimensional comparisons.

10.3.1 Understanding Radial and Polar Coordinates

- **Polar coordinates** use an **angle ( )** and **radius (r)** to position points.
- **Radial charts** use the polar system to lay out data in a circular or arc-based structure.

This opens the door to creative chart types such as:

- **Radial bar charts** (circular bar plots)
- **Radar charts** (spider plots)

- **Clock-style time visualizations**
- **Wind rose diagrams**

10.3.2 When to Use Radial and Polar Charts

Use these formats when:

- The data is **cyclical**, like time-of-day or months of the year.
- You want to **compare categories** around a circle.
- A circular layout improves **visual space utilization** or storytelling.

10.3.3 Building a Radial Bar Chart with D3

Let's walk through a radial bar chart that visualizes, for example, website visits by hour of the day.

Sample Data (Visits per Hour)

```
const data = [
  { hour: 0, visits: 120 },
  { hour: 1, visits: 80 },
  ...
  { hour: 23, visits: 200 }
];
```

10.3.4 Step 1: Set Up Dimensions and Scales

```
const width = 600;
const height = 600;
const innerRadius = 100;
const outerRadius = Math.min(width, height) / 2 - 40;

const svg = d3.select("body")
  .append("svg")
  .attr("width", width)
  .attr("height", height)
  .append("g")
  .attr("transform", `translate(${width / 2},${height / 2})`);
```

10.3.5 Step 2: Define Scales

- `xScale` maps **hours (0–23)** to angles around the circle.
- `yScale` maps **visits** to bar **length (radius)**.

```
const xScale = d3.scaleBand()
  .domain(data.map(d => d.hour))
  .range([0, 2 * Math.PI])
  .align(0);

const yScale = d3.scaleLinear()
  .domain([0, d3.max(data, d => d.visits)])
  .range([innerRadius, outerRadius]);
```

10.3.6 Step 3: Draw Bars Using Arcs

Use `d3.arc()` to construct each bar between `innerRadius` and a value-dependent outer radius.

```
const arc = d3.arc()
  .innerRadius(innerRadius)
  .outerRadius(d => yScale(d.visits))
  .startAngle(d => xScale(d.hour))
  .endAngle(d => xScale(d.hour) + xScale.bandwidth())
  .padAngle(0.01)
  .padRadius(innerRadius);

svg.append("g")
  .selectAll("path")
  .data(data)
  .join("path")
  .attr("fill", "steelblue")
  .attr("d", arc);
```

10.3.7 Step 4: Add Hour Labels

```
svg.append("g")
  .selectAll("text")
  .data(data)
  .join("text")
  .attr("text-anchor", "middle")
  .attr("x", d => Math.sin(xScale(d.hour) + xScale.bandwidth() / 2) * (innerRadius - 10))
  .attr("y", d => -Math.cos(xScale(d.hour) + xScale.bandwidth() / 2) * (innerRadius - 10))
  .text(d => d.hour);
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="UTF-8">
  <title>Radial Bar Chart - Visits Per Hour</title>
  <script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
  <style>
    body {
      font-family: sans-serif;
      background: #f5f5f5;
      margin: 0;
      padding: 20px;
      display: flex;
      justify-content: center;
      align-items: center;
    }
    svg {
      background: white;
      box-shadow: 0 0 5px #ccc;
      border-radius: 8px;
    }
  </style>
</head>
<body>

<script>
const data = Array.from({ length: 24 }, (_, i) => ({
  hour: i,
  visits: Math.floor(50 + 150 * Math.abs(Math.sin(i / 24 * Math.PI * 2)))
}));

const width = 600;
const height = 600;
const innerRadius = 100;
const outerRadius = Math.min(width, height) / 2 - 40;

const svg = d3.select("body")
  .append("svg")
  .attr("width", width)
  .attr("height", height)
  .append("g")
  .attr("transform", `translate(${width / 2},${height / 2})`);

const xScale = d3.scaleBand()
  .domain(data.map(d => d.hour))
  .range([0, 2 * Math.PI])
  .align(0);

const yScale = d3.scaleLinear()
  .domain([0, d3.max(data, d => d.visits)])
  .range([innerRadius, outerRadius]);

const arc = d3.arc()
  .innerRadius(innerRadius)
  .outerRadius(d => yScale(d.visits))
  .startAngle(d => xScale(d.hour))
  .endAngle(d => xScale(d.hour) + xScale.bandwidth())
  .padAngle(0.01)
  .padRadius(innerRadius);
```

```
svg.append("g")
  .selectAll("path")
  .data(data)
  .join("path")
  .attr("fill", "steelblue")
  .attr("d", arc);

// Add hour labels
svg.append("g")
  .selectAll("text")
  .data(data)
  .join("text")
  .attr("text-anchor", "middle")
  .attr("font-size", 12)
  .attr("x", d => Math.sin(xScale(d.hour) + xScale.bandwidth() / 2) * (innerRadius - 20))
  .attr("y", d => -Math.cos(xScale(d.hour) + xScale.bandwidth() / 2) * (innerRadius - 20))
  .text(d => `${d.hour}`);
</script>

</body>
</html>
```

10.3.8 Polar Angle vs. Radius Mapping

In polar charts:

- The **angle ( )** encodes categories (e.g., time, months, dimensions).
- The **radius (r)** encodes quantitative values (e.g., counts, percentages).

This radial bar chart uses angle to differentiate hours and radius to represent visit counts.

10.3.9 Enhancements

- Animate bars growing from `innerRadius` to final radius.
- Color-code bars by value intensity using `d3.scaleSequential`.
- Convert to donut-style radar charts using `lineRadial` and `radar-like` polygon shapes.

10.3.10 Summary

Feature	Description
`d3.scaleBand()`	Maps categorical data to angles
`d3.scaleLinear()`	Maps values to radius

Feature	Description
`d3.arc()`	Generates radial segments
Polar layout	Uses angle + radius instead of x/y Cartesian system

Radial and polar charts can turn common datasets into compelling visual stories. D3's expressive flexibility allows you to customize every element — from smooth angles to animated arcs — to create data-driven circular charts that are both engaging and informative.

10.4 Hierarchical Data (TreeMaps, Sunburst, Dendrograms)

Visualizing **hierarchical data**—data that has nested levels, like file systems, organizational charts, or taxonomies—is a powerful use case in D3. With the help of `d3.hierarchy()` and layout generators like `d3.treemap`, `d3.partition`, and `d3.cluster`, you can represent complex relationships in intuitive formats such as **treemaps**, **sunbursts**, and **dendrograms**.

10.4.1 What Is Hierarchical Data?

Hierarchical data consists of nested structures, often expressed as parent-child relationships. Here's a simple example in JSON:

```
const data = {
  name: "root",
  children: [
    { name: "A", value: 100 },
    {
      name: "B",
      children: [
        { name: "B1", value: 30 },
        { name: "B2", value: 70 }
      ]
    }
  ]
};
```

D3 uses the `d3.hierarchy()` function to convert this structure into a format compatible with its layout algorithms.

10.4.2 Step 1: Build a `d3.hierarchy` Object

```
const root = d3.hierarchy(data)
  .sum(d => d.value)       // Aggregate leaf values
  .sort((a, b) => b.value - a.value); // Optional: sort by value
```

This structure supports traversal and layout calculations.

10.4.3 Treemap Example

A **treemap** divides a rectangle into nested sub-rectangles, sized by value. It's excellent for comparing proportions at multiple levels.

10.4.4 Step-by-Step:

```
const width = 600;
const height = 400;

const treemap = d3.treemap()
  .size([width, height])
  .padding(1);

treemap(root); // Compute layout

const svg = d3.select("body").append("svg")
  .attr("width", width)
  .attr("height", height);

svg.selectAll("rect")
  .data(root.leaves())
  .join("rect")
  .attr("x", d => d.x0)
  .attr("y", d => d.y0)
  .attr("width", d => d.x1 - d.x0)
  .attr("height", d => d.y1 - d.y0)
  .attr("fill", "steelblue");

svg.selectAll("text")
  .data(root.leaves())
  .join("text")
  .attr("x", d => d.x0 + 4)
  .attr("y", d => d.y0 + 14)
  .text(d => d.data.name)
  .attr("fill", "white")
  .attr("font-size", "10px");
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Treemap Example</title>
  <script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
  <style>
    body {
      font-family: sans-serif;
      background: #f0f0f0;
      display: flex;
      justify-content: center;
      padding: 20px;
    }
    svg {
      background: white;
      box-shadow: 0 0 5px #ccc;
      border-radius: 8px;
    }
  </style>
</head>
<body>

<script>
// Sample hierarchical data
const data = {
  name: "root",
  children: [
    { name: "A", value: 100 },
    { name: "B", value: 300 },
    { name: "C", value: 200 },
    {
      name: "Group D", children: [
        { name: "D1", value: 80 },
        { name: "D2", value: 120 }
      ]
    }
  ]
};

const width = 600;
const height = 400;

const root = d3.hierarchy(data)
  .sum(d => d.value)
  .sort((a, b) => b.value - a.value);

d3.treemap()
  .size([width, height])
  .padding(1)(root);

const svg = d3.select("body").append("svg")
  .attr("width", width)
  .attr("height", height);

const color = d3.scaleOrdinal(d3.schemeCategory10);

svg.selectAll("rect")
  .data(root.leaves())
  .join("rect")
  .attr("x", d => d.x0)
  .attr("y", d => d.y0)
  .attr("width", d => d.x1 - d.x0)
  .attr("height", d => d.y1 - d.y0)
  .attr("fill", d => color(d.parent.data.name));

svg.selectAll("text")
  .data(root.leaves())
  .join("text")
  .attr("x", d => d.x0 + 4)
  .attr("y", d => d.y0 + 14)
  .text(d => d.data.name)
  .attr("fill", "white")
  .attr("font-size", "10px")
  .attr("pointer-events", "none");
</script>

</body>
</html>
```

10.4.5 Sunburst Example

A **sunburst chart** is a radial version of a partition layout. It's ideal for displaying hierarchical depth while maintaining relative value proportions.

10.4.6 Step-by-Step:

```
const radius = 250;

const partition = d3.partition()
  .size([2 * Math.PI, radius]);

partition(root); // Compute layout

const arc = d3.arc()
  .startAngle(d => d.x0)
  .endAngle(d => d.x1)
  .innerRadius(d => d.y0)
  .outerRadius(d => d.y1);

const svg = d3.select("body").append("svg")
  .attr("width", radius * 2)
  .attr("height", radius * 2)
  .append("g")
  .attr("transform", `translate(${radius},${radius})`);

svg.selectAll("path")
  .data(root.descendants())
  .join("path")
  .attr("d", arc)
  .attr("fill", d => d.depth === 0 ? "#fff" : d3.schemeCategory10[d.depth % 10])
  .attr("stroke", "#fff");
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Sunburst Example</title>
  <script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
  <style>
    body {
      font-family: sans-serif;
      background: #f0f0f0;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
    }
    svg {
      background: white;
      box-shadow: 0 0 5px #ccc;
      border-radius: 8px;
    }
  </style>
</head>
<body>

<script>
// Sample hierarchical data
const data = {
  name: "root",
  children: [
    {
      name: "Group A",
      children: [
        { name: "A1", value: 100 },
        { name: "A2", value: 300 }
      ]
    },
    {
      name: "Group B",
      children: [
        { name: "B1", value: 200 },
        { name: "B2", value: 150 }
      ]
    }
  ]
};

const radius = 250;
const width = radius * 2;
const height = radius * 2;

// Convert to hierarchy
const root = d3.hierarchy(data)
  .sum(d => d.value)
  .sort((a, b) => b.value - a.value);

// Compute partition layout
d3.partition().size([2 * Math.PI, radius])(root);

// Define arc generator
const arc = d3.arc()
  .startAngle(d => d.x0)
  .endAngle(d => d.x1)
  .innerRadius(d => d.y0)
  .outerRadius(d => d.y1);

// Create SVG and group
const svg = d3.select("body").append("svg")
  .attr("width", width)
  .attr("height", height)
  .append("g")
  .attr("transform", `translate(${radius},${radius})`);

// Draw arcs
svg.selectAll("path")
  .data(root.descendants())
  .join("path")
  .attr("d", arc)
  .attr("fill", d => d.depth === 0 ? "#fff" : d3.schemeCategory10[d.depth % 10])
  .attr("stroke", "#fff");

// Optional: Add text labels
svg.selectAll("text")
  .data(root.descendants().filter(d => d.depth > 0))
  .join("text")
  .attr("transform", d => {
    const angle = ((d.x0 + d.x1) / 2) * 180 / Math.PI;
    const radius = (d.y0 + d.y1) / 2;
    return `rotate(${angle - 90}) translate(${radius},0) rotate(${angle < 180 ? 0 : 180})`;
  })
  .attr("dy", "0.35em")
  .attr("text-anchor", d => ((d.x0 + d.x1) / 2) < Math.PI ? "start" : "end")
  .text(d => d.data.name)
  .style("font-size", "10px");
</script>

</body>
</html>
```

10.4.7 Dendrogram (Bonus)

A **dendrogram** is a tree diagram, often used in biology or clustering. Use `d3.cluster()` or `d3.tree()` with `d3.hierarchy()` and draw links and nodes with SVG lines and circles.

```
const treeLayout = d3.cluster().size([height, width]);

treeLayout(root);
```

```
// Draw links
svg.selectAll("path")
  .data(root.links())
  .join("path")
  .attr("d", d3.linkHorizontal()
    .x(d => d.y)
    .y(d => d.x));
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Dendrogram Example</title>
  <script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
  <style>
    body {
      font-family: sans-serif;
      background: #f9f9f9;
      display: flex;
      justify-content: center;
      align-items: center;
      height: 100vh;
    }
    svg {
      background: white;
      box-shadow: 0 0 5px #ccc;
    }
  </style>
</head>
<body>

<script>
const data = {
  name: "Root",
  children: [
    {
      name: "Branch A",
      children: [
        { name: "Leaf A1" },
        { name: "Leaf A2" }
      ]
    },
    {
      name: "Branch B",
      children: [
        {
          name: "Sub-branch B1",
          children: [
            { name: "Leaf B1a" },
            { name: "Leaf B1b" }
          ]
        },
        { name: "Leaf B2" }
      ]
    }
  ]
};

const width = 600;
const height = 400;

const root = d3.hierarchy(data);

// Set up layout
const treeLayout = d3.cluster().size([height, width - 160]);
treeLayout(root);

// Create SVG
const svg = d3.select("body").append("svg")
  .attr("width", width)
  .attr("height", height)
  .append("g")
  .attr("transform", "translate(80,0)");

// Draw links
svg.selectAll("path")
  .data(root.links())
  .join("path")
  .attr("fill", "none")
  .attr("stroke", "#555")
  .attr("stroke-width", 1.5)
  .attr("d", d3.linkHorizontal()
    .x(d => d.y)
    .y(d => d.x));

// Draw nodes
svg.selectAll("circle")
  .data(root.descendants())
  .join("circle")
  .attr("cx", d => d.y)
  .attr("cy", d => d.x)
  .attr("r", 4)
  .attr("fill", d => d.children ? "#555" : "#999");

// Add labels
svg.selectAll("text")
  .data(root.descendants())
  .join("text")
  .attr("x", d => d.y + 6)
  .attr("y", d => d.x)
  .attr("dy", "0.32em")
  .text(d => d.data.name)
  .style("font-size", "12px");
</script>

</body>
</html>
```

10.4.8 Summary

Visualization	Layout Used	Best For
Treemap	`d3.treemap()`	Space-efficient comparisons
Sunburst	`d3.partition()`	Radial breakdown of categories
Dendrogram	`d3.cluster()`	Tree structures and hierarchies

D3's hierarchical layouts make it easy to explore nested data structures visually. Whether you're compressing a file system into a treemap, revealing category hierarchies in a sunburst, or mapping relationships in a dendrogram, D3 provides flexible tools to bring structure and insight to complex data.

Chapter 11.

Modularizing and Reusing Code

11 Modularizing and Reusing Code

11.1 Encapsulating Charts in Functions

As your D3 visualizations become more complex, structuring your code for **reusability** and **modularity** becomes critical. Instead of rewriting visualization logic every time, you can encapsulate chart behavior in reusable functions—enabling easier maintenance, testing, and integration.

11.1.1 Why Encapsulate D3 Charts?

Encapsulating a chart in a function allows you to:

- Reuse charts across pages or projects
- Separate concerns (data, layout, DOM binding)
- Improve testability and readability
- Make charts configurable (e.g., data source, size, colors)

11.1.2 Basic Chart Wrapper Structure

A reusable chart function takes a config object with parameters like:

```
function barChart({ data, width, height, selector }) {
  // chart implementation here
}
```

You can then call it as:

```
barChart({
  data: myData,
  width: 500,
  height: 300,
  selector: "#chart-container"
});
```

11.1.3 Example: Reusable Bar Chart Function

```
function barChart({ data, width = 600, height = 400, selector }) {
  const margin = { top: 20, right: 20, bottom: 40, left: 40 };
  const innerWidth = width - margin.left - margin.right;
  const innerHeight = height - margin.top - margin.bottom;

  const x = d3.scaleBand()
```

```
    .domain(data.map(d => d.name))
    .range([0, innerWidth])
    .padding(0.1);

  const y = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.value)])
    .nice()
    .range([innerHeight, 0]);

  d3.select(selector).select("svg").remove(); // Clear previous chart if any

  const svg = d3.select(selector)
    .append("svg")
    .attr("width", width)
    .attr("height", height);

  const g = svg.append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

  g.append("g")
    .call(d3.axisLeft(y));

  g.append("g")
    .attr("transform", `translate(0,${innerHeight})`)
    .call(d3.axisBottom(x));

  g.selectAll("rect")
    .data(data)
    .join("rect")
    .attr("x", d => x(d.name))
    .attr("y", d => y(d.value))
    .attr("width", x.bandwidth())
    .attr("height", d => innerHeight - y(d.value))
    .attr("fill", "steelblue");
}
```

11.1.4 Using the Function

```
const sampleData = [
  { name: "A", value: 30 },
  { name: "B", value: 80 },
  { name: "C", value: 45 }
];

barChart({
  data: sampleData,
  selector: "#chart1",
  width: 500,
  height: 300
});
```

Make sure you have a container element like this in your HTML:

```
<div id="chart1"></div>
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Reusable D3 Bar Chart</title>
  <script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 20px;
    }
    #chart1 {
      margin-top: 20px;
    }
  </style>
</head>
<body>

<h2>Bar Chart Example</h2>
<div id="chart1"></div>

<script>
function barChart({ data, width = 600, height = 400, selector }) {
  const margin = { top: 20, right: 20, bottom: 40, left: 40 };
  const innerWidth = width - margin.left - margin.right;
  const innerHeight = height - margin.top - margin.bottom;

  const x = d3.scaleBand()
    .domain(data.map(d => d.name))
    .range([0, innerWidth])
    .padding(0.1);

  const y = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.value)])
    .nice()
    .range([innerHeight, 0]);

  d3.select(selector).select("svg").remove(); // Clear previous chart if any

  const svg = d3.select(selector)
    .append("svg")
    .attr("width", width)
    .attr("height", height);

  const g = svg.append("g")
    .attr("transform", `translate(${margin.left},${margin.top})`);

  g.append("g")
    .call(d3.axisLeft(y));

  g.append("g")
    .attr("transform", `translate(0,${innerHeight})`)
    .call(d3.axisBottom(x));
  g.selectAll("rect")
    .data(data)
    .join("rect")
    .attr("x", d => x(d.name))
    .attr("y", d => y(d.value))
    .attr("width", x.bandwidth())
    .attr("height", d => innerHeight - y(d.value))
    .attr("fill", "steelblue");
}

// Example usage
const sampleData = [
  { name: "A", value: 30 },
  { name: "B", value: 80 },
  { name: "C", value: 45 }
];

barChart({
  data: sampleData,
  selector: "#chart1",
  width: 500,
  height: 300
});
</script>

</body>
</html>
```

11.1.5 Optional Enhancements

- Add parameters for `color`, `title`, `margin`, `xLabel`, etc.
- Emit events (e.g., on bar click) for interactivity.
- Return a handle to update or destroy the chart.

11.1.6 Summary

Encapsulating D3 charts in functions is a vital practice for scalable, maintainable data visualizations. By passing configuration objects, you allow your charts to be flexible and portable, making them easy to adapt across use cases. In later sections, you'll build on this pattern to integrate charts into reusable components, frameworks, and dashboards.

11.2 Reusable D3 Chart Patterns

To build scalable, maintainable D3 visualizations, it's essential to structure your code in a way that promotes **reuse** and **encapsulation**. One of the most influential techniques for this is **Mike Bostock's reusable chart pattern**—a flexible and composable design that turns a D3 chart into a configurable function.

11.2.1 What Is the Reusable Chart Pattern?

Instead of writing charts as imperative one-off scripts, this pattern wraps your D3 logic inside a **closure** that returns a function. This function, when called with a selection, draws or updates the chart.

The structure:

```
function myChart() {
  // internal state (e.g., width, height, data)

  function chart(selection) {
    // D3 enter-update-exit logic here
  }

  // Getter/setter methods for configuration
  chart.width = function(_) {
    return arguments.length ? (width = _, chart) : width;
  };

  return chart;
}
```

By using closures, your chart maintains private state while exposing a flexible public interface.

11.2.2 Example: Reusable Line Chart

Let's build a reusable line chart that supports different datasets and dimensions.

Step 1: Define the Line Chart Module

```
function lineChart() {
  let width = 500;
  let height = 300;
  let margin = { top: 20, right: 20, bottom: 30, left: 40 };
  let data = [];

  function chart(selection) {
    selection.each(function () {
      const innerWidth = width - margin.left - margin.right;
      const innerHeight = height - margin.top - margin.bottom;
      const x = d3.scaleTime()
        .domain(d3.extent(data, d => d.date))
        .range([0, innerWidth]);

      const y = d3.scaleLinear()
        .domain([0, d3.max(data, d => d.value)])
        .nice()
        .range([innerHeight, 0]);

      const line = d3.line()
        .x(d => x(d.date))
        .y(d => y(d.value));

      const svg = d3.select(this).selectAll("svg")
        .data([data])
        .join("svg")
        .attr("width", width)
        .attr("height", height);

      svg.selectAll("*").remove(); // Clear for re-render

      const g = svg.append("g")
        .attr("transform", `translate(${margin.left},${margin.top})`);

      g.append("g")
        .call(d3.axisLeft(y));

      g.append("g")
        .attr("transform", `translate(0,${innerHeight})`)
        .call(d3.axisBottom(x));

      g.append("path")
        .datum(data)
        .attr("fill", "none")
        .attr("stroke", "steelblue")
        .attr("stroke-width", 2)
        .attr("d", line);
    });
  }

  // Configuration API
  chart.data = function (_) {
    return arguments.length ? (data = _, chart) : data;
  };

  chart.width = function (_) {
    return arguments.length ? (width = _, chart) : width;
  };

  chart.height = function (_) {
    return arguments.length ? (height = _, chart) : height;
  };

  chart.margin = function (_) {
    return arguments.length ? (margin = _, chart) : margin;
  };

  return chart;
```

```
}
```

11.2.3 Step 2: Use the Chart with Different Data

```
const sampleData = [
  { date: new Date(2024, 0, 1), value: 30 },
  { date: new Date(2024, 1, 1), value: 50 },
  { date: new Date(2024, 2, 1), value: 45 },
  { date: new Date(2024, 3, 1), value: 60 }
];

const chart = lineChart()
  .width(600)
  .height(350)
  .data(sampleData);

d3.select("#linechart").call(chart);
```

Make sure the HTML includes a container:

```
<div id="linechart"></div>
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Reusable Line Chart with D3</title>
  <script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
  <style>
    body {
      font-family: sans-serif;
      padding: 20px;
    }
    #linechart {
      margin-top: 20px;
    }
  </style>
</head>
<body>

<h2>Reusable Line Chart Example</h2>
<div id="linechart"></div>

<script>
function lineChart() {
  let width = 500;
  let height = 300;
  let margin = { top: 20, right: 20, bottom: 30, left: 40 };
  let data = [];

  function chart(selection) {
    selection.each(function () {
      const innerWidth = width - margin.left - margin.right;
      const innerHeight = height - margin.top - margin.bottom;

      const x = d3.scaleTime()
        .domain(d3.extent(data, d => d.date))
        .range([0, innerWidth]);

      const y = d3.scaleLinear()
        .domain([0, d3.max(data, d => d.value)])
        .nice()
        .range([innerHeight, 0]);

      const line = d3.line()
        .x(d => x(d.date))
        .y(d => y(d.value));

      const svg = d3.select(this).selectAll("svg")
        .data([data])
        .join("svg")
        .attr("width", width)
        .attr("height", height);

      svg.selectAll("*").remove(); // Clear for re-render

      const g = svg.append("g")
        .attr("transform", `translate(${margin.left},${margin.top})`);

      g.append("g")
        .call(d3.axisLeft(y));

      g.append("g")
        .attr("transform", `translate(0,${innerHeight})`)
        .call(d3.axisBottom(x));

      g.append("path")
        .datum(data)
        .attr("fill", "none")
        .attr("stroke", "steelblue")
        .attr("stroke-width", 2)
        .attr("d", line);
    });
  }

  chart.data = function (_) {
    return arguments.length ? (data = _, chart) : data;
  };

  chart.width = function (_) {
    return arguments.length ? (width = _, chart) : width;
  };

  chart.height = function (_) {
    return arguments.length ? (height = _, chart) : height;
  };

  chart.margin = function (_) {
    return arguments.length ? (margin = _, chart) : margin;
  };

  return chart;
}

// Sample data
const sampleData = [
  { date: new Date(2024, 0, 1), value: 30 },
  { date: new Date(2024, 1, 1), value: 50 },
  { date: new Date(2024, 2, 1), value: 45 },
  { date: new Date(2024, 3, 1), value: 60 }
];

// Create and draw the chart
const chart = lineChart()
  .width(600)
  .height(350)
  .data(sampleData);

d3.select("#linechart").call(chart);
</script>

</body>
</html>
```

11.2.4 Reusability in Practice

This pattern allows you to:

- Chain configuration: `chart.width(800).height(400).data(myData)`
- Render the same chart in multiple places with different settings
- Encapsulate transitions, tooltips, or event handling inside the chart logic
- Compose charts in dashboards or embed them in frameworks

11.2.5 Summary

Mike Bostock's reusable chart pattern is a foundation for building modular, maintainable D3 components. By designing your chart as a function with a configurable API, you unlock reusability, flexibility, and testability. In the next sections, you'll learn how to use these patterns in React, Vue, and other frameworks to build powerful, integrated visual applications.

Chapter 12.

Debugging and Performance Optimization

12 Debugging and Performance Optimization

12.1 Understanding D3 Error Messages

When working with D3.js, encountering runtime errors is common—especially during the data binding and DOM manipulation phases. Understanding what these errors mean and how to diagnose them is essential for smooth development and debugging.

12.1.1 Common D3 Runtime Errors and Their Causes

`Cannot read property x of undefined`

This error usually occurs when you try to access a property on a data object that doesn't exist or hasn't been properly bound. For example:

```
selection
  .attr("cx", d => d.x)
```

If some elements in the selection don't have bound data (`d` is undefined), `d.x` causes this error.

Cause:

- Data array length doesn't match the number of selected elements.
- Incorrect use of `.data()` or missing `.enter()` to create elements.
- Asynchronous data loading not handled properly, causing empty or undefined data.

Selection Mismatches or Empty Selections

Sometimes you might write:

```
d3.selectAll(".bar").data(data);
```

but find no changes or errors downstream. This often means:

- The selection `.bar` matches no DOM elements (empty selection).
- Data is bound to zero elements, so no updates happen.
- Using `.select()` vs `.selectAll()` incorrectly (select picks one, selectAll picks many).

12.1.2 Diagnosing and Fixing Errors

Use `console.log()` Strategically

Logging selections and data at key points helps identify issues:

```
const bars = d3.selectAll(".bar").data(data);
console.log("Bars selection:", bars);
console.log("Bound data:", bars.data());
```

If you see an empty selection or unexpected data, verify:

- The data array is correctly loaded and structured.
- The DOM elements exist before binding data.
- The selection string matches existing elements.

Inspect Elements with Browser Dev Tools

Open your browser's Developer Tools (usually F12 or right-click → Inspect):

- Use the **Elements** tab to check if the expected SVG or HTML elements are present.
- Verify classes and IDs you're targeting with D3 selections.
- Check the Console for error stack traces pointing to problematic lines.

12.1.3 Example Walkthrough: Fixing a Common Binding Error

Suppose this code throws an error:

```
const circles = d3.select("svg")
  .selectAll("circle")
  .data([10, 20, 30]);

circles.attr("r", d => d);  // Error: Cannot read property 'r' of undefined
```

Issue: `circles` is an **update** selection without elements because no `<circle>` elements exist yet.

Fix: Use the enter selection to create missing elements:

```
const circles = d3.select("svg")
  .selectAll("circle")
  .data([10, 20, 30]);

circles.enter()
  .append("circle")
  .attr("cx", (d,i) => (i + 1) * 50)
  .attr("cy", 50)
  .attr("r", d => d);
```

12.1.4 Tips for Preventing Errors

- Always use `.enter()` to create new elements for new data.
- Check data length vs existing DOM elements before binding.
- Use clear, consistent selectors.
- Wrap code that depends on data in asynchronous callbacks or promises if loading data

externally.
- Use defensive coding, e.g., `d => d ? d.x : 0`, to avoid null errors temporarily.

By systematically logging data and selections, inspecting the DOM, and understanding D3's data join lifecycle, you can quickly identify the root cause of errors and fix them effectively. Debugging is an essential skill to master for building robust D3 visualizations.

12.2 Profiling and Reducing Re-renders

As your D3 visualizations grow more complex, performance can become a concern—especially with large datasets or frequent updates. Identifying bottlenecks and reducing unnecessary work helps keep your visuals smooth and responsive.

12.2.1 Identifying Performance Bottlenecks with Chrome DevTools

The Chrome DevTools **Performance** tab is a powerful ally in diagnosing where your visualization might be slowing down:

1. **Open DevTools** (F12 or right-click → Inspect).
2. Navigate to the **Performance** tab.
3. Click the **Record** button (circle icon), then interact with your visualization (e.g., update data, resize).
4. Stop recording after the interaction.

The profiling timeline will show:

- **JavaScript execution time:** Time spent running your D3 code.
- **Style recalculations and layout:** Costly DOM updates and reflows.
- **Paint:** How long the browser spends rendering changes.

Look for long frames or spikes—these indicate expensive operations.

12.2.2 Key Causes of Slowdowns in D3 Visualizations

- **Excessive or repeated data joins:** Binding data and modifying DOM elements repeatedly for the entire dataset when only a subset changed.
- **Unnecessary DOM manipulations:** Updating attributes or styles that haven't changed.
- **Overly frequent redraws or transitions:** Animations or interactions that trigger full re-renders too often.

12.2.3 Practical Techniques to Improve Performance

Minimize Data Joins

- **Bind data only when necessary.** Avoid re-binding on every frame or event if data hasn't changed.
- **Use the enter-update-exit pattern efficiently:** Update only affected elements rather than all.

```
// Example: update only new or changed data
const update = svg.selectAll("rect").data(data);
update.enter().append("rect") /* enter new */
  .merge(update)             /* update existing */
  .attr("height", d => yScale(d));
update.exit().remove();      /* remove old */
```

Conditional Updates

Before setting attributes or styles, check if the value changed to avoid redundant DOM writes:

```
selection.each(function(d, i) {
  const currentHeight = +d3.select(this).attr("height");
  const newHeight = yScale(d);
  if (currentHeight !== newHeight) {
    d3.select(this).attr("height", newHeight);
  }
});
```

Avoid Unnecessary DOM Manipulations

- Group multiple attribute or style changes in a single chain to reduce browser layout thrashing.
- Cache selections rather than querying DOM repeatedly.
- Use `requestAnimationFrame` to batch updates for smoother rendering.

Debounce or Throttle Expensive Operations

For events like window resizing or mouse movements, use debouncing or throttling to limit update frequency.

```
let resizeTimeout;
window.addEventListener("resize", () => {
  clearTimeout(resizeTimeout);
  resizeTimeout = setTimeout(() => {
    redrawChart();
  }, 200);
});
```

12.2.4 Summary

Profiling your D3 code helps reveal the "hot spots" slowing down your visualization. By minimizing data joins, applying conditional updates, and avoiding unnecessary DOM writes, you ensure your charts remain performant and responsive—even as complexity grows.

In the next section, we'll explore the differences between the Virtual DOM and Real DOM and how they relate to D3's direct DOM manipulations.

12.3 Virtual DOM vs Real DOM in D3 Context

When working with modern web applications, understanding the difference between **Virtual DOM** and **Real DOM** is crucial—especially when integrating D3.js with other frameworks or optimizing performance.

12.3.1 What is the Real DOM?

The **Real DOM** is the browser's actual Document Object Model—an in-memory tree representing the HTML elements on the page. When you use D3's API to:

- select elements (`d3.select()`),
- append nodes (`.append()`),
- or set attributes and styles (`.attr()`, `.style()`),

you are directly modifying this Real DOM.

These changes are immediately reflected in the page, and the browser must re-compute layouts, repaint, and potentially trigger expensive reflows.

12.3.2 What is the Virtual DOM?

Frameworks like **React**, **Vue**, and **Svelte** use a **Virtual DOM**:

- It is an abstraction: a lightweight, in-memory representation of the UI.
- When state changes, the framework **diffs** the new Virtual DOM tree against the previous one.
- Only the minimal necessary changes are applied to the Real DOM.

This approach reduces costly DOM operations by batching updates and avoiding unnecessary mutations.

12.3.3 D3s Imperative Real DOM Manipulation: Benefits and Challenges

D3 embraces **imperative** programming:

- You directly control exactly *which* elements are created, updated, or removed.
- This fine-grained control makes D3 extremely powerful for custom visualizations and animations.
- It lets you implement complex enter-update-exit patterns and fine-tune performance.

However, challenges arise:

- Frequent or large-scale DOM manipulations can cause performance bottlenecks.
- Without an intermediate abstraction like the Virtual DOM, you are responsible for minimizing and batching DOM updates.
- Coordinating complex UI state alongside D3's direct DOM changes can be tricky in large applications.

12.3.4 Managing Large-Scale Redraws Effectively with D3

Here are some strategies to keep performance high when manipulating the Real DOM with D3:

- **Isolate DOM updates:** Perform all necessary calculations (scales, data joins) first, then batch all DOM updates to minimize layout thrashing.
- **Use the enter-update-exit pattern:** Update only changed elements instead of redrawing the entire chart.
- **Avoid repeated selections:** Cache selections when possible.
- **Throttle or debounce updates:** For frequent events like resizing or streaming data, limit redraw frequency.
- **Integrate D3 with Virtual DOM frameworks cautiously:**
  - Let React or Vue manage DOM elements and state.
  - Use D3 primarily for scales, math, and path generation.
  - Delegate direct DOM manipulations to D3 only when necessary, keeping React's Virtual DOM in charge of rendering.

12.3.5 Summary

While React's Virtual DOM abstracts and optimizes DOM updates by diffing, D3 gives you direct, imperative access to the Real DOM. This provides unmatched control and flexibility but requires careful management to maintain performance.

Understanding these paradigms helps you choose the right approach and balance for your project, especially when building large, complex, or interactive data visualizations.

Next, we will explore practical tips and tricks for working with **large datasets** efficiently in D3.

12.4 Large Datasets: Tips and Tricks

Visualizing large datasets—think tens or hundreds of thousands of points—poses unique challenges. Rendering every data point directly in SVG can quickly overwhelm the browser, leading to sluggish interactions or crashes. To maintain performance while delivering meaningful insights, you need smart strategies.

12.4.1 Use Canvas-Based Rendering for High Volume

SVG excels at rich, interactive graphics with moderate numbers of elements but struggles with tens of thousands of DOM nodes. The HTML5 `<canvas>` element offers a pixel-based drawing surface that can render thousands or millions of points efficiently because it doesn't maintain individual DOM elements for each shape.

Example: Drawing 100,000 points in a scatter plot using canvas:

```
const canvas = d3.select("#chart").append("canvas")
  .attr("width", width)
  .attr("height", height)
  .node();

const context = canvas.getContext("2d");

data.forEach(d => {
  context.beginPath();
  context.arc(xScale(d.x), yScale(d.y), 2, 0, 2 * Math.PI);
  context.fillStyle = "steelblue";
  context.fill();
});
```

D3 scales and math can still be used; only rendering happens via canvas APIs.

12.4.2 Downsampling and Data Simplification

Rendering every data point isn't always necessary. You can reduce dataset size while preserving overall patterns by:

- **Random sampling:** Select a subset of points.
- **Aggregation:** Compute averages or bins to represent clusters.
- **Filtering:** Display only relevant data slices (e.g., by zoom level or user selection).

Example of downsampling with JavaScript's `.filter()` or a library like simple-statistics.

12.4.3 Clustering and Aggregation Techniques

Clustering groups nearby data points into clusters, reducing visual clutter and computation:

- Use clustering algorithms (e.g., k-means, DBSCAN).
- Display clusters as single aggregated points or shapes with size proportional to cluster size.
- Offer zoom or drill-down interactions to reveal details.

This is common in large scatterplots or geographic visualizations.

12.4.4 Pagination and Progressive Rendering

For very large datasets, consider:

- **Pagination:** Show data in manageable chunks with navigation controls.
- **Progressive rendering:** Use `requestIdleCallback` or `setTimeout` to render data in batches, preventing UI blocking.

Example using `requestIdleCallback` to chunk rendering:

```
let index = 0;
function drawBatch(deadline) {
  while (index < data.length && deadline.timeRemaining() > 0) {
    drawPoint(data[index]);
    index++;
  }
  if (index < data.length) {
    requestIdleCallback(drawBatch);
  }
}
requestIdleCallback(drawBatch);
```

This keeps the page responsive by yielding control back to the browser between batches.

12.4.5 Summary

Handling large datasets with D3 requires blending creative rendering techniques with smart data management:

- Use **canvas** when the number of elements is huge.
- Apply **downsampling** or **clustering** to reduce visual noise.
- Implement **pagination** or **progressive rendering** to maintain responsiveness.

These strategies empower you to create scalable, performant visualizations that remain interactive and insightful—even at massive data scales.

With these advanced tips, you are well equipped to tackle demanding datasets while maintaining smooth, dynamic visualizations.

Chapter 13.

Real-World Projects

13 Real-World Projects

13.1 Live Data Dashboard

Chapter 13: Real-World Projects Section 1: Live Data Dashboard

Building a live data dashboard is one of the most exciting applications of D3.js. It brings together many concepts: dynamic data updates, smooth transitions, and coordinating multiple charts in a clean layout. In this section, we'll walk through creating a dashboard that updates in real-time using periodic polling, though WebSocket integration follows a similar pattern.

13.1.1 Choosing the Dataset: Cryptocurrency Prices

For this example, we'll use a simulated stream of cryptocurrency price data (e.g., Bitcoin prices). This dataset updates frequently, making it perfect to demonstrate live updates.

13.1.2 Step 1: Dashboard Layout Composition

A typical dashboard consists of multiple charts arranged logically. Let's plan for:

- A **line chart** showing price over time.
- A **bar chart** showing trading volume by time intervals.
- A **summary panel** displaying current price and percentage change.

Use CSS Grid or Flexbox to create a responsive layout. For instance:

```html
<div id="dashboard" style="display: grid; grid-template-columns: 2fr 1fr; gap: 20px;">
  <div id="price-chart"></div>
  <div id="volume-chart"></div>
  <div id="summary-panel"></div>
</div>
```

13.1.3 Step 2: Initial Chart Setup

Create your charts inside their respective containers, setting up SVG elements, scales, and axes as usual. For example, the price line chart will have:

- Time scale on the X-axis (`d3.scaleTime()`)
- Linear scale on the Y-axis for price (`d3.scaleLinear()`)

The volume bar chart uses a similar time scale but with a linear Y-axis for volume.

13.1.4 Step 3: Fetching Live Data Periodically

To simulate live data updates, use `setInterval` or `d3.interval` to fetch new data every few seconds.

```
d3.interval(() => {
  fetchNewData().then(newDataPoint => {
    updateDashboard(newDataPoint);
  });
}, 5000); // Every 5 seconds
```

`fetchNewData()` can be a call to a real API or a mock function generating random data.

13.1.5 Step 4: Updating Charts with New Data Points

In `updateDashboard(newDataPoint)`, append the new data point to your dataset and update scales and axes accordingly.

Example for the line chart update:

```
// Add new point to data array
data.push(newDataPoint);

// Keep only last N points for performance
if (data.length > maxPoints) data.shift();

// Update scales domains
xScale.domain(d3.extent(data, d => d.time));
yScale.domain([0, d3.max(data, d => d.price)]);

// Update axes
svg.select(".x-axis")
  .transition().duration(750)
  .call(d3.axisBottom(xScale));

svg.select(".y-axis")
  .transition().duration(750)
  .call(d3.axisLeft(yScale));

// Update line path
svg.select(".line-path")
  .datum(data)
  .transition().duration(750)
  .attr("d", lineGenerator);
```

For the bar chart, update similarly by rebinding data and redrawing bars with smooth transitions.

13.1.6 Step 5: Best Practices for Real-Time Visuals

- **Limit data size:** Keep a sliding window of recent data (e.g., last 100 points) to prevent slowdowns.
- **Smooth transitions:** Use `.transition()` to animate updates without abrupt jumps.
- **Debounce updates:** Avoid flooding the browser with too many renders; batch or throttle updates if needed.
- **Clear user feedback:** Show loading indicators or status messages if fetching live data involves network latency.
- **Use efficient data joins:** Properly apply enter-update-exit patterns to minimize unnecessary DOM changes.

13.1.7 Bonus: Adding a Summary Panel

Update a simple text panel showing the latest price and percent change:

```
d3.select("#summary-panel").html(`
  <h2>BTC/USD</h2>
  <p>Price: $${newDataPoint.price.toFixed(2)}</p>
  <p>Change: ${calculatePercentChange(data).toFixed(2)}%</p>
`);
```

13.1.8 Wrapping Up

By combining D3's powerful update patterns with a clean dashboard layout and timely data fetching, you create live, interactive visualizations that keep users informed in real time. This pattern generalizes well to many data sources—from weather stations to IoT sensors—making it a foundational skill for modern data visualization.

Ready to take this further? Try integrating a WebSocket feed for instant data pushes or combine brushing and filtering across charts to build a truly interactive analytics dashboard.

13.2 Interactive Time Series Explorer

Time series data is one of the most common and insightful data types to visualize. An interactive time series explorer lets users zoom in, pan across dates, brush to select specific ranges, and gain rich contextual insights through tooltips and annotations. In this section, we'll build such a visualization step-by-step, using a COVID-19 daily case counts dataset as our example.

13.2.1 Step 1: Preparing the Data and SVG Setup

Assume you have a dataset of daily case counts with fields like:

```
[
  { date: "2020-01-22", cases: 555 },
  { date: "2020-01-23", cases: 654 },
  ...
]
```

Parse dates with `d3.timeParse`:

```
const parseDate = d3.timeParse("%Y-%m-%d");
data.forEach(d => d.date = parseDate(d.date));
```

Set up your SVG container with margins for axes:

```
const margin = { top: 20, right: 40, bottom: 110, left: 50 },
      width = 960 - margin.left - margin.right,
      height = 500 - margin.top - margin.bottom,
      heightOverview = 100;  // For brushing context

const svg = d3.select("#chart")
  .append("svg")
  .attr("width", width + margin.left + margin.right)
  .attr("height", height + margin.top + margin.bottom + heightOverview)
  .append("g")
  .attr("transform", `translate(${margin.left},${margin.top})`);
```

13.2.2 Step 2: Define Scales and Axes

Create x and y scales for the main chart and a smaller overview chart (brush area):

```
const x = d3.scaleTime().range([0, width]);
const y = d3.scaleLinear().range([height, 0]);

const xOverview = d3.scaleTime().range([0, width]);
const yOverview = d3.scaleLinear().range([heightOverview, 0]);
```

Set domains from the data:

```
x.domain(d3.extent(data, d => d.date));
y.domain([0, d3.max(data, d => d.cases)]);
xOverview.domain(x.domain());
yOverview.domain(y.domain());
```

Add axes groups:

```
svg.append("g").attr("class", "x axis").attr("transform", `translate(0,${height})`);
svg.append("g").attr("class", "y axis");
```

13.2.3 Step 3: Draw Line and Area Charts

Create a line generator for the main chart and a simplified area for the brush overview:

```
const line = d3.line()
  .x(d => x(d.date))
  .y(d => y(d.cases));

const areaOverview = d3.area()
  .x(d => xOverview(d.date))
  .y0(heightOverview)
  .y1(d => yOverview(d.cases));
```

Append the main line and the overview area:

```
svg.append("path")
  .datum(data)
  .attr("class", "line")
  .attr("d", line);

const overview = svg.append("g")
  .attr("transform", `translate(0,${height + margin.bottom / 2})`);

overview.append("path")
  .datum(data)
  .attr("class", "area-overview")
  .attr("d", areaOverview);
```

13.2.4 Step 4: Add Brushing for Range Selection

Define a brush on the overview to select a time range:

```
const brush = d3.brushX()
  .extent([[0, 0], [width, heightOverview]])
  .on("brush end", brushed);

overview.append("g")
  .attr("class", "brush")
  .call(brush)
  .call(brush.move, x.range());  // Initialize full range selected
```

The `brushed` function filters the main chart's displayed range:

```
function brushed(event) {
  if (event.selection) {
    const [x0, x1] = event.selection.map(xOverview.invert);
    x.domain([x0, x1]);
    svg.select(".line")
      .transition()
      .duration(500)
      .attr("d", line);
    svg.select(".x.axis")
      .transition()
      .duration(500)
      .call(d3.axisBottom(x));
  }
}
```

13.2.5 Step 5: Implement Zooming and Panning

Add zoom behavior to allow users to zoom or pan the main chart directly:

```
const zoom = d3.zoom()
  .scaleExtent([1, 10])
  .translateExtent([[0, 0], [width, height]])
  .extent([[0, 0], [width, height]])
  .on("zoom", zoomed);

svg.append("rect")
  .attr("class", "zoom")
  .attr("width", width)
  .attr("height", height)
  .style("fill", "none")
  .style("pointer-events", "all")
  .call(zoom);

function zoomed(event) {
  const t = event.transform;
  const newX = t.rescaleX(xOverview);
  x.domain(newX.domain());

  svg.select(".line")
    .attr("d", line);
  svg.select(".x.axis")
    .call(d3.axisBottom(x));

  overview.select(".brush")
    .call(brush.move, x.range().map(t.invertX, t));
}
```

13.2.6 Step 6: Add Tooltip Overlays

Create a tooltip `<div>` hidden by default:

```
<div id="tooltip" style="position:absolute; pointer-events:none; opacity:0; background:#fff; border:1px
```

Add circles at data points and show tooltip on hover:

```
svg.selectAll(".dot")
  .data(data)
  .enter().append("circle")
  .attr("class", "dot")
  .attr("cx", d => x(d.date))
  .attr("cy", d => y(d.cases))
  .attr("r", 4)
  .on("mouseover", (event, d) => {
    d3.select("#tooltip")
      .style("opacity", 1)
      .html(`<strong>Date:</strong> ${d3.timeFormat("%b %d, %Y")(d.date)}<br/><strong>Cases:</strong> $
      .style("left", (event.pageX + 10) + "px")
      .style("top", (event.pageY - 28) + "px");
  })
  .on("mouseout", () => {
    d3.select("#tooltip").style("opacity", 0);
  });
```

13.2.7 Step 7: Add Annotation Markers

Annotations can highlight important dates or events. Create small vertical lines and labels:

```
const annotations = [
  { date: parseDate("2020-03-11"), label: "WHO declares pandemic" },
  { date: parseDate("2020-12-14"), label: "First vaccine rollout" }
];

const annotationGroup = svg.append("g").attr("class", "annotations");

annotationGroup.selectAll("line")
  .data(annotations)
  .enter()
  .append("line")
  .attr("x1", d => x(d.date))
  .attr("y1", 0)
  .attr("x2", d => x(d.date))
  .attr("y2", height)
  .attr("stroke", "red")
  .attr("stroke-dasharray", "4 2");

annotationGroup.selectAll("text")
  .data(annotations)
  .enter()
  .append("text")
  .attr("x", d => x(d.date) + 5)
  .attr("y", 15)
  .text(d => d.label)
  .style("fill", "red")
  .style("font-size", "10px");
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Line Chart with Brush, Zoom, Tooltip, Annotations</title>
  <script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
  <style>
    body {
      font-family: sans-serif;
      margin: 30px;
    }
    .line {
      fill: none;
      stroke: steelblue;
      stroke-width: 2px;
    }
    .area-overview {
      fill: lightsteelblue;
    }
    .dot {
      fill: steelblue;
      stroke: white;
      stroke-width: 1.5px;
    }
    .brush .selection {
      fill: #777;
      fill-opacity: 0.3;
      stroke: #fff;
      shape-rendering: crispEdges;
    }
    .zoom {
      cursor: move;
    }
    .annotations line {
      stroke: red;
      stroke-dasharray: 4 2;
    }
    .annotations text {
      fill: red;
      font-size: 10px;
    }
    #tooltip {
      position: absolute;
      pointer-events: none;
      opacity: 0;
      background: #fff;
      border: 1px solid #ccc;
      padding: 5px;
      font-size: 12px;
    }
  </style>
</head>
<body>

<h2>D3 Interactive Line Chart with Brushing, Zooming, and Tooltips</h2>
<div id="chart"></div>
<div id="tooltip"></div>

<script>
const rawData = [
  { date: "2020-01-22", cases: 555 },
  { date: "2020-01-23", cases: 654 },
  { date: "2020-01-24", cases: 941 },
  { date: "2020-01-25", cases: 1434 },
  { date: "2020-01-26", cases: 2118 },
  { date: "2020-01-27", cases: 2927 },
  { date: "2020-01-28", cases: 5578 },
  { date: "2020-03-11", cases: 118319 },  // WHO pandemic declaration
  { date: "2020-12-14", cases: 72373294 } // Vaccine rollout
];

const parseDate = d3.timeParse("%Y-%m-%d");
const data = rawData.map(d => ({ date: parseDate(d.date), cases: d.cases }));

const margin = { top: 20, right: 40, bottom: 110, left: 50 },
      width = 960 - margin.left - margin.right,
      height = 500 - margin.top - margin.bottom,
      heightOverview = 100;

const svg = d3.select("#chart")
  .append("svg")
  .attr("width", width + margin.left + margin.right)
  .attr("height", height + margin.top + margin.bottom + heightOverview)
  .append("g")
  .attr("transform", `translate(${margin.left},${margin.top})`);

const x = d3.scaleTime().range([0, width]);
const y = d3.scaleLinear().range([height, 0]);
const xOverview = d3.scaleTime().range([0, width]);
const yOverview = d3.scaleLinear().range([heightOverview, 0]);

x.domain(d3.extent(data, d => d.date));
y.domain([0, d3.max(data, d => d.cases)]);
xOverview.domain(x.domain());
yOverview.domain(y.domain());

svg.append("g")
  .attr("class", "x axis")
  .attr("transform", `translate(0,${height})`)
  .call(d3.axisBottom(x));

svg.append("g")
  .attr("class", "y axis")
  .call(d3.axisLeft(y));

const line = d3.line()
  .x(d => x(d.date))
  .y(d => y(d.cases));

const areaOverview = d3.area()
  .x(d => xOverview(d.date))
  .y0(heightOverview)
  .y1(d => yOverview(d.cases));

svg.append("path")
  .datum(data)
  .attr("class", "line")
  .attr("d", line);

const overview = svg.append("g")
  .attr("transform", `translate(0,${height + margin.bottom / 2})`);

overview.append("path")
  .datum(data)
  .attr("class", "area-overview")
  .attr("d", areaOverview);

const brush = d3.brushX()
  .extent([[0, 0], [width, heightOverview]])
  .on("brush end", brushed);

overview.append("g")
  .attr("class", "brush")
  .call(brush)
  .call(brush.move, x.range());

function brushed(event) {
  if (event.selection) {
    const [x0, x1] = event.selection.map(xOverview.invert);
    x.domain([x0, x1]);
    svg.select(".line")
      .transition()
      .duration(500)
      .attr("d", line);
    svg.select(".x.axis")
      .transition()
      .duration(500)
      .call(d3.axisBottom(x));

    svg.selectAll(".dot")
      .attr("cx", d => x(d.date))
      .attr("cy", d => y(d.cases));
  }
}

const zoom = d3.zoom()
  .scaleExtent([1, 10])
  .translateExtent([[0, 0], [width, height]])
  .extent([[0, 0], [width, height]])
  .on("zoom", zoomed);

svg.append("rect")
  .attr("class", "zoom")
  .attr("width", width)
  .attr("height", height)
  .style("fill", "none")
  .style("pointer-events", "all")
  .call(zoom);

function zoomed(event) {
  const t = event.transform;
  const newX = t.rescaleX(xOverview);
  x.domain(newX.domain());

  svg.select(".line").attr("d", line);
  svg.select(".x.axis").call(d3.axisBottom(x));
  overview.select(".brush").call(brush.move, x.range().map(t.invertX, t));

  svg.selectAll(".dot")
    .attr("cx", d => x(d.date))
    .attr("cy", d => y(d.cases));
}
```

```
svg.selectAll(".dot")
  .data(data)
  .enter().append("circle")
  .attr("class", "dot")
  .attr("cx", d => x(d.date))
  .attr("cy", d => y(d.cases))
  .attr("r", 4)
  .on("mouseover", (event, d) => {
    d3.select("#tooltip")
      .style("opacity", 1)
      .html(`<strong>Date:</strong> ${d3.timeFormat("%b %d, %Y")(d.date)}<br/><strong>Cases:</strong> $
      .style("left", (event.pageX + 10) + "px")
      .style("top", (event.pageY - 28) + "px");
  })
  .on("mouseout", () => {
    d3.select("#tooltip").style("opacity", 0);
  });

const annotations = [
  { date: parseDate("2020-03-11"), label: "WHO declares pandemic" },
  { date: parseDate("2020-12-14"), label: "First vaccine rollout" }
];

const annotationGroup = svg.append("g").attr("class", "annotations");

annotationGroup.selectAll("line")
  .data(annotations)
  .enter()
  .append("line")
  .attr("x1", d => x(d.date))
  .attr("y1", 0)
  .attr("x2", d => x(d.date))
  .attr("y2", height)
  .attr("stroke", "red")
  .attr("stroke-dasharray", "4 2");

annotationGroup.selectAll("text")
  .data(annotations)
  .enter()
  .append("text")
  .attr("x", d => x(d.date) + 5)
  .attr("y", 15)
  .text(d => d.label)
  .style("fill", "red")
  .style("font-size", "10px");
</script>

</body>
</html>
```

13.2.8 Conclusion

With brushing, zooming, panning, tooltips, and annotations, your interactive time series explorer becomes a powerful tool for data analysis and storytelling. Users can dive deep into specific time ranges, get contextual info, and explore trends fluidly.

This pattern applies widely, whether exploring financial stock prices, climate data, or epidemic curves — giving your users dynamic control over the story the data tells.

13.3 Visualizing a Social Network

Visualizing social networks helps reveal hidden patterns of connection—friend groups, influence hubs, or communication flows. In this section, we'll build a dynamic force-directed graph using simulated social network data. We'll cover node and link rendering, force simulation with D3, and interactivity such as dragging and hovering.

13.3.1 Step 1: Simulated Data Structure

A basic social network graph has **nodes** (people) and **links** (relationships).

```
const nodes = [
  { id: 'Alice', group: 1 },
  { id: 'Bob', group: 1 },
  { id: 'Carol', group: 2 },
  { id: 'David', group: 2 },
  { id: 'Eve', group: 3 }
];

const links = [
  { source: 'Alice', target: 'Bob' },
  { source: 'Alice', target: 'Carol' },
  { source: 'Bob', target: 'David' },
  { source: 'Carol', target: 'David' },
  { source: 'Eve', target: 'Alice' }
];
```

Here, each `group` value represents a community cluster.

13.3.2 Step 2: Set Up the SVG Container

```
const width = 800;
const height = 600;

const svg = d3.select("#network")
```

```
  .append("svg")
  .attr("width", width)
  .attr("height", height);
```

13.3.3 Step 3: Initialize the Force Simulation

D3 provides physics-based simulation through `d3.forceSimulation()`.

```
const simulation = d3.forceSimulation(nodes)
  .force("link", d3.forceLink(links).id(d => d.id).distance(100))
  .force("charge", d3.forceManyBody().strength(-200)) // repulsion
  .force("center", d3.forceCenter(width / 2, height / 2))
  .force("collision", d3.forceCollide().radius(30));
```

13.3.4 Step 4: Draw Links and Nodes

```
const link = svg.append("g")
  .attr("stroke", "#aaa")
  .selectAll("line")
  .data(links)
  .enter().append("line")
  .attr("stroke-width", 2);

const node = svg.append("g")
  .attr("stroke", "#fff")
  .attr("stroke-width", 1.5)
  .selectAll("circle")
  .data(nodes)
  .enter().append("circle")
  .attr("r", 15)
  .attr("fill", d => color(d.group))
  .call(drag(simulation));
```

Use a categorical color scale to distinguish groups:

```
const color = d3.scaleOrdinal(d3.schemeCategory10);
```

Add labels:

```
const labels = svg.append("g")
  .selectAll("text")
  .data(nodes)
  .enter().append("text")
  .text(d => d.id)
  .attr("font-size", 12)
  .attr("text-anchor", "middle")
  .attr("dy", -20);
```

13.3.5 Step 5: Enable Drag Interactivity

```
function drag(simulation) {
  return d3.drag()
    .on("start", (event, d) => {
      if (!event.active) simulation.alphaTarget(0.3).restart();
      d.fx = d.x;
      d.fy = d.y;
    })
    .on("drag", (event, d) => {
      d.fx = event.x;
      d.fy = event.y;
    })
    .on("end", (event, d) => {
      if (!event.active) simulation.alphaTarget(0);
      d.fx = null;
      d.fy = null;
    });
}
```

13.3.6 Step 6: Update Positions on Each Simulation Tick

```
simulation.on("tick", () => {
  link
    .attr("x1", d => d.source.x)
    .attr("y1", d => d.source.y)
    .attr("x2", d => d.target.x)
    .attr("y2", d => d.target.y);

  node
    .attr("cx", d => d.x)
    .attr("cy", d => d.y);

  labels
    .attr("x", d => d.x)
    .attr("y", d => d.y);
});
```

13.3.7 Step 7: Add Hover Interactions

```
node.on("mouseover", function (event, d) {
    d3.select(this).attr("stroke", "#000").attr("stroke-width", 3);
    tooltip
      .style("opacity", 1)
      .html(`<strong>${d.id}</strong><br/>Group: ${d.group}`)
      .style("left", (event.pageX + 10) + "px")
      .style("top", (event.pageY - 10) + "px");
  })
  .on("mouseout", function () {
    d3.select(this).attr("stroke", "#fff").attr("stroke-width", 1.5);
    tooltip.style("opacity", 0);
  });
```

And create the tooltip div:

```
<div id="tooltip" style="position:absolute; opacity:0; background:#fff; padding:5px; border:1px solid #c
```

13.3.8 Step 8: Add Optional Clustering Force (Community Gravity)

To visually separate communities, use `d3.forceX` or `d3.forceY`:

```
const groupCenters = {
  1: width / 4,
  2: width / 2,
  3: 3 * width / 4
};

simulation.force("cluster", d3.forceX(d => groupCenters[d.group]).strength(0.1));
```

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>D3 Force-Directed Graph</title>
  <script src="https://d3js.org/d3.v7.min.js"></script>
  <style>
    body {
      font-family: sans-serif;
      margin: 0;
      padding: 0;
    }
    svg {
      border: 1px solid #ccc;
      background-color: #f9f9f9;
    }
    #tooltip {
      position: absolute;
      opacity: 0;
      background: #fff;
      padding: 5px;
      border: 1px solid #ccc;
      pointer-events: none;
      font-size: 12px;
    }
  </style>
</head>
<body>

<div id="network"></div>
<div id="tooltip"></div>
```

```
<script>
  const nodes = [
    { id: 'Alice', group: 1 },
    { id: 'Bob', group: 1 },
    { id: 'Carol', group: 2 },
    { id: 'David', group: 2 },
    { id: 'Eve', group: 3 }
  ];

  const links = [
    { source: 'Alice', target: 'Bob' },
    { source: 'Alice', target: 'Carol' },
    { source: 'Bob', target: 'David' },
    { source: 'Carol', target: 'David' },
    { source: 'Eve', target: 'Alice' }
  ];

  const width = 800;
  const height = 600;

  const svg = d3.select("#network")
    .append("svg")
    .attr("width", width)
    .attr("height", height);

  const color = d3.scaleOrdinal(d3.schemeCategory10);

  const simulation = d3.forceSimulation(nodes)
    .force("link", d3.forceLink(links).id(d => d.id).distance(100))
    .force("charge", d3.forceManyBody().strength(-200))
    .force("center", d3.forceCenter(width / 2, height / 2))
    .force("collision", d3.forceCollide().radius(30));

  // Optional community clustering
  const groupCenters = {
    1: width / 4,
    2: width / 2,
    3: 3 * width / 4
  };

  simulation.force("cluster", d3.forceX(d => groupCenters[d.group]).strength(0.1));

  const link = svg.append("g")
    .attr("stroke", "#aaa")
    .selectAll("line")
    .data(links)
    .enter().append("line")
    .attr("stroke-width", 2);

  const node = svg.append("g")
    .attr("stroke", "#fff")
    .attr("stroke-width", 1.5)
    .selectAll("circle")
    .data(nodes)
    .enter().append("circle")
    .attr("r", 15)
    .attr("fill", d => color(d.group))
    .call(drag(simulation));
  const labels = svg.append("g")
    .selectAll("text")
    .data(nodes)
    .enter().append("text")
    .text(d => d.id)
    .attr("font-size", 12)
    .attr("text-anchor", "middle")
    .attr("dy", -20);

  function drag(simulation) {
    return d3.drag()
      .on("start", (event, d) => {
        if (!event.active) simulation.alphaTarget(0.3).restart();
        d.fx = d.x;
        d.fy = d.y;
      })
      .on("drag", (event, d) => {
        d.fx = event.x;
        d.fy = event.y;
      })
      .on("end", (event, d) => {
        if (!event.active) simulation.alphaTarget(0);
        d.fx = null;
        d.fy = null;
      });
  }

  const tooltip = d3.select("#tooltip");

  node.on("mouseover", function (event, d) {
      d3.select(this).attr("stroke", "#000").attr("stroke-width", 3);
      tooltip
        .style("opacity", 1)
        .html(`<strong>${d.id}</strong><br/>Group: ${d.group}`)
        .style("left", (event.pageX + 10) + "px")
        .style("top", (event.pageY - 10) + "px");
    })
    .on("mouseout", function () {
      d3.select(this).attr("stroke", "#fff").attr("stroke-width", 1.5);
      tooltip.style("opacity", 0);
    });

  simulation.on("tick", () => {
    link
      .attr("x1", d => d.source.x)
      .attr("y1", d => d.source.y)
      .attr("x2", d => d.target.x)
      .attr("y2", d => d.target.y);

    node
      .attr("cx", d => d.x)
      .attr("cy", d => d.y);

    labels
      .attr("x", d => d.x)
      .attr("y", d => d.y);
  });
</script>
```

```
</body>
</html>
```

13.3.9 Conclusion

A force-directed network visualization allows users to explore complex relationships naturally. By leveraging D3's force simulation, interactivity, and modular design, we created a layout where users can:

- See clusters and community structures.
- Drag nodes to reposition them.
- Hover to inspect node data.

This structure is flexible for social networks, dependency graphs, communication structures, and more.

13.4 Customizable Chart Toolkit for End-Users

Interactive and customizable charting tools empower non-developers to explore data visually. In this section, we'll walk through building a chart toolkit where end-users can upload their own datasets (e.g., CSV), choose chart types from a dropdown, and configure visual settings such as color schemes and axis ranges using UI controls like sliders and toggles.

This pattern is the foundation for data journalism tools, embedded analytics platforms, and custom dashboards.

13.4.1 Goals of a Chart Toolkit

- **User uploads data (CSV)**
- **Select chart type**: bar, line, scatter
- **Choose color scheme**
- **Configure axes with sliders or inputs**
- **Render responsive D3 charts based on selections**

13.4.2 Step 1: HTML Structure and UI Controls

```
<div id="toolkit">
  <input type="file" id="upload" accept=".csv" />
  <label for="chartType">Chart Type:</label>
  <select id="chartType">
    <option value="bar">Bar</option>
    <option value="line">Line</option>
    <option value="scatter">Scatter</option>
  </select>

  <label for="colorScheme">Color Scheme:</label>
  <select id="colorScheme">
    <option value="schemeCategory10">Category10</option>
    <option value="schemeAccent">Accent</option>
    <option value="schemeDark2">Dark2</option>
  </select>

  <label for="xRange">X Axis Range:</label>
  <input type="range" id="xRange" min="0" max="100" step="1" value="100" />

  <div id="chart"></div>
</div>
```

13.4.3 Step 2: Handle File Upload and Parse CSV

```
document.getElementById('upload').addEventListener('change', function (event) {
  const file = event.target.files[0];
  if (!file) return;

  const reader = new FileReader();
  reader.onload = function (e) {
    const text = e.target.result;
    const data = d3.csvParse(text);
    updateChart(data);
  };
  reader.readAsText(file);
});
```

13.4.4 Step 3: Modular Chart Renderer

Create a reusable `renderChart(data, options)` function that reads current settings and draws the appropriate chart.

```
function updateChart(data) {
  const chartType = document.getElementById('chartType').value;
  const colorScheme = d3[document.getElementById('colorScheme').value];
  const xMax = +document.getElementById('xRange').value;
  // Clean previous chart
  d3.select("#chart").selectAll("*").remove();

  if (chartType === "bar") {
    renderBarChart(data, colorScheme, xMax);
  } else if (chartType === "line") {
    renderLineChart(data, colorScheme, xMax);
  } else if (chartType === "scatter") {
    renderScatterPlot(data, colorScheme, xMax);
  }
}
```

Each chart type would be a standalone function taking standardized parameters.

13.4.5 Step 4: Example Bar Chart Renderer

```
function renderBarChart(data, colorScheme, xMax) {
  const width = 600, height = 400, margin = { top: 20, right: 20, bottom: 40, left: 40 };

  const svg = d3.select("#chart").append("svg")
    .attr("viewBox", [0, 0, width, height]);

  const x = d3.scaleBand()
    .domain(data.map(d => d.label))
    .range([margin.left, width - margin.right])
    .padding(0.2);

  const y = d3.scaleLinear()
    .domain([0, Math.min(d3.max(data, d => +d.value), xMax)])
    .nice()
    .range([height - margin.bottom, margin.top]);

  const color = d3.scaleOrdinal(colorScheme).domain(data.map(d => d.label));

  svg.append("g")
    .selectAll("rect")
    .data(data)
    .join("rect")
    .attr("x", d => x(d.label))
    .attr("y", d => y(+d.value))
    .attr("height", d => y(0) - y(+d.value))
    .attr("width", x.bandwidth())
    .attr("fill", d => color(d.label));

  svg.append("g")
    .attr("transform", `translate(0,${height - margin.bottom})`)
    .call(d3.axisBottom(x));

  svg.append("g")
    .attr("transform", `translate(${margin.left},0)`)
    .call(d3.axisLeft(y));
}
```

Note: All chart renderers (`renderLineChart`, `renderScatterPlot`, etc.) follow the same signature for consistency.

13.4.6 Step 5: Hook Up Event Listeners for Live Updates

```
document.querySelectorAll("#chartType, #colorScheme, #xRange")
  .forEach(input => input.addEventListener("change", () => updateChart(currentData)));
```

Where `currentData` is the most recently loaded dataset.

Full runnable code:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8" />
<title>D3 Chart Toolkit</title>
<script src="https://cdn.jsdelivr.net/npm/d3@7"></script>
<style>
  body {
    font-family: sans-serif;
    margin: 20px;
  }
  #toolkit label {
    margin-left: 10px;
    margin-right: 5px;
  }
  #chart svg {
    width: 100%;
    max-width: 700px;
    height: 400px;
    border: 1px solid #ccc;
    background: #f9f9f9;
  }
</style>
</head>
<body>

<div id="toolkit">
  <input type="file" id="upload" accept=".csv" />

  <label for="chartType">Chart Type:</label>
  <select id="chartType">
    <option value="bar">Bar</option>
    <option value="line">Line</option>
    <option value="scatter">Scatter</option>
  </select>

  <label for="colorScheme">Color Scheme:</label>
  <select id="colorScheme">
    <option value="schemeCategory10">Category10</option>
    <option value="schemeAccent">Accent</option>
    <option value="schemeDark2">Dark2</option>
  </select>

  <label for="xRange">X Axis Max:</label>
  <input type="range" id="xRange" min="0" max="100" step="1" value="100" />
  <span id="xRangeVal">100</span>
</div>

<div id="chart"></div>

<script>
  let currentData = null;

  // Listen for CSV upload
  document.getElementById('upload').addEventListener('change', function(event) {
    const file = event.target.files[0];
    if (!file) return;

    const reader = new FileReader();
    reader.onload = function(e) {
      const text = e.target.result;
      let data = d3.csvParse(text);

      // Expect CSV columns: label, value, x, y (some may not be used depending on chart)
      // Normalize data for bar & line: label & value; scatter: x & y
      data = data.map(d => {
        return {
          label: d.label || d.Label || d.name || "N/A",
          value: +d.value || +d.Value || 0,
          x: +d.x || +d.X || 0,
          y: +d.y || +d.Y || 0
        };
      });

      currentData = data;
      updateChart(currentData);
    };
    reader.readAsText(file);
  });

  // Update chart when controls change
  document.querySelectorAll("#chartType, #colorScheme, #xRange").forEach(el => {
    el.addEventListener("input", () => {
      if (currentData) updateChart(currentData);
    });
  });

  // Display slider value dynamically
  document.getElementById('xRange').addEventListener('input', function() {
    document.getElementById('xRangeVal').textContent = this.value;
  });

  function updateChart(data) {
    const chartType = document.getElementById('chartType').value;
    const colorSchemeName = document.getElementById('colorScheme').value;
    const colorScheme = d3[colorSchemeName];
    const xMax = +document.getElementById('xRange').value;

    d3.select("#chart").selectAll("*").remove();
    if (chartType === "bar") {
      renderBarChart(data, colorScheme, xMax);
    } else if (chartType === "line") {
      renderLineChart(data, colorScheme, xMax);
    } else if (chartType === "scatter") {
      renderScatterPlot(data, colorScheme, xMax);
    }
  }

  // --- Bar Chart ---
  function renderBarChart(data, colorScheme, xMax) {
    const width = 700, height = 400, margin = { top: 30, right: 30, bottom: 50, left: 60 };

    const svg = d3.select("#chart").append("svg")
      .attr("viewBox", [0, 0, width, height]);

    const x = d3.scaleBand()
      .domain(data.map(d => d.label))
      .range([margin.left, width - margin.right])
      .padding(0.2);

    const y = d3.scaleLinear()
      .domain([0, Math.min(d3.max(data, d => d.value), xMax)])
      .nice()
      .range([height - margin.bottom, margin.top]);

    const color = d3.scaleOrdinal(colorScheme).domain(data.map(d => d.label));

    svg.append("g")
      .attr("transform", `translate(0,${height - margin.bottom})`)
      .call(d3.axisBottom(x))
      .selectAll("text")
      .attr("transform", "rotate(-40)")
      .style("text-anchor", "end");

    svg.append("g")
      .attr("transform", `translate(${margin.left},0)`)
      .call(d3.axisLeft(y));

    svg.selectAll("rect")
      .data(data)
      .join("rect")
      .attr("x", d => x(d.label))
      .attr("y", d => y(d.value))
      .attr("height", d => y(0) - y(d.value))
      .attr("width", x.bandwidth())
      .attr("fill", d => color(d.label));
  }

  // --- Line Chart ---
  function renderLineChart(data, colorScheme, xMax) {
    const width = 700, height = 400, margin = { top: 30, right: 30, bottom: 50, left: 60 };

    // x axis: label is ordinal, so convert to index for line
    const x = d3.scalePoint()
      .domain(data.map(d => d.label))
      .range([margin.left, width - margin.right]);
    const y = d3.scaleLinear()
      .domain([0, Math.min(d3.max(data, d => d.value), xMax)])
      .nice()
      .range([height - margin.bottom, margin.top]);

    const color = d3.scaleOrdinal(colorScheme).domain(data.map(d => d.label));

    const line = d3.line()
      .x(d => x(d.label))
      .y(d => y(d.value))
      .curve(d3.curveMonotoneX);

    const svg = d3.select("#chart").append("svg")
      .attr("viewBox", [0, 0, width, height]);

    svg.append("g")
      .attr("transform", `translate(0,${height - margin.bottom})`)
      .call(d3.axisBottom(x))
      .selectAll("text")
      .attr("transform", "rotate(-40)")
      .style("text-anchor", "end");

    svg.append("g")
      .attr("transform", `translate(${margin.left},0)`)
      .call(d3.axisLeft(y));

    svg.append("path")
      .datum(data)
      .attr("fill", "none")
      .attr("stroke", "steelblue")
      .attr("stroke-width", 2)
      .attr("d", line);

    // Circles on points
    svg.selectAll("circle")
      .data(data)
      .join("circle")
      .attr("cx", d => x(d.label))
      .attr("cy", d => y(d.value))
      .attr("r", 4)
      .attr("fill", d => color(d.label));
  }

  // --- Scatter Plot ---
  function renderScatterPlot(data, colorScheme, xMax) {
    const width = 700, height = 400, margin = { top: 30, right: 30, bottom: 50, left: 60 };

    // Use data.x and data.y for scatter plot
    // Filter out points with no x or y (NaN)
    const filteredData = data.filter(d => !isNaN(d.x) && !isNaN(d.y));

    if (filteredData.length === 0) {
      d3.select("#chart").append("div").text("Data missing numeric x,y columns for scatter plot.");
      return;
    }

    const x = d3.scaleLinear()
      .domain([0, Math.min(d3.max(filteredData, d => d.x), xMax)])
```

```
      .nice()
      .range([margin.left, width - margin.right]);

    const y = d3.scaleLinear()
      .domain([0, d3.max(filteredData, d => d.y)])
      .nice()
      .range([height - margin.bottom, margin.top]);

    const color = d3.scaleOrdinal(colorScheme).domain(filteredData.map(d => d.label));

    const svg = d3.select("#chart").append("svg")
      .attr("viewBox", [0, 0, width, height]);

    svg.append("g")
      .attr("transform", `translate(0,${height - margin.bottom})`)
      .call(d3.axisBottom(x));

    svg.append("g")
      .attr("transform", `translate(${margin.left},0)`)
      .call(d3.axisLeft(y));

    svg.selectAll("circle")
      .data(filteredData)
      .join("circle")
      .attr("cx", d => x(d.x))
      .attr("cy", d => y(d.y))
      .attr("r", 5)
      .attr("fill", d => color(d.label))
      .attr("opacity", 0.7);
  }
</script>

<p>Example CSV format for bar/line charts (label,value):</p>
<pre>
label,value
A,30
B,80
C,45
D,60
E,20
</pre>

<p>Example CSV format for scatter plot (label,x,y):</p>
<pre>
label,x,y
A,5,20
B,15,35
C,25,40
D,35,30
E,45,50
</pre>

</body>
</html>
```

13.4.7 Design Considerations

Modularization

Structure your project with reusable modules:

- `renderBarChart()`
- `renderLineChart()`
- `parseUploadedData()`
- `createUIControls()`

This makes testing and extension easy (e.g., adding a donut chart).

Accessibility and UX

- Use labels and legends
- Increase font sizes and spacing for mobile
- Ensure dropdowns and sliders are keyboard-accessible

Responsiveness

Use SVG's `viewBox` and resize logic (from Chapter 9) to make charts adapt to container size.

13.4.8 Summary

By decoupling the UI logic from rendering code and exposing a clean, modular API, we created a toolkit where:

- Users control the chart without writing code
- D3 renders visualizations dynamically based on input
- Multiple chart types are supported using a unified interface

This approach is scalable for internal dashboards, no-code data tools, or embeddable chart widgets.

Next Challenge: Add chart exporting (`svg` to `png`), dataset filters, and theme switching for a fully production-grade toolkit.