Contents

Chapter 1

Chapter 1: HTML, CSS, JavaScript Fundamentals and Advanced Concepts

1.1 HTML Fundamentals and Advanced Concepts

1.1.1 Document Structure and Semantic HTML

Semantic HTML is not merely about choosing appropriate tags but understanding how document structure impacts accessibility tree construction, SEO indexing, and assistive technology navigation.

1.1.1.1 Document Outline Algorithm

The HTML5 outline algorithm was designed to create document structure from sectioning elements. While not fully implemented in browsers, understanding it reveals how to create meaningful hierarchies.

```
<body>
  <header>
    <h1>Site Title</h1>
    <nav>
      <h2>Navigation</h2> <!-- Creates subsection -->
      <ul>...</ul>
    </nav>
  </header>

  <main>
    <article>
      <h1>Article Title</h1> <!-- New sectioning root -->
      <section>
        <h2>Section within article</h2>
        <p>Content...</p>
```

```
      </section>
    </article>

    <aside>
      <h2>Related Content</h2> <!-- Separate section -->
    </aside>
  </main>

  <footer>
    <h2>Footer</h2>
  </footer>
</body>
```

Each sectioning element (`<article>`, `<section>`, `<nav>`, `<aside>`) creates a new section in the outline. Heading levels restart within each section. However, in practice, use only one `<h1>` per page and maintain sequential heading hierarchy for maximum compatibility with assistive technologies.

1.1.2 Forms and Advanced Accessibility

Form accessibility extends beyond label associations. Understanding ARIA roles, states, and properties enables creating complex custom form controls.

1.1.2.1 Native vs Custom Controls

Native form elements have built-in accessibility: - Keyboard navigation (Tab, Arrow keys, Space, Enter) - Focus management and visual indicators - Screen reader announcements - Form validation and error reporting - Mobile input optimizations

When building custom controls, replicate all native behavior:

```
<!-- Custom checkbox implementation -->
<div role="checkbox"
     aria-checked="false"
     aria-labelledby="label-id"
     tabindex="0"
     class="custom-checkbox">
  <span id="label-id">Accept terms</span>
</div>
```

```
const checkbox = document.querySelector('[role="checkbox"]');

checkbox.addEventListener('click', toggleCheckbox);
checkbox.addEventListener('keydown', (e) => {
  if (e.key === ' ' || e.key === 'Enter') {
    e.preventDefault();
    toggleCheckbox();
  }
});
```

```
function toggleCheckbox() {
  const checked = checkbox.getAttribute('aria-checked') === 'true';
  checkbox.setAttribute('aria-checked', !checked);
  // Update visual state
  checkbox.classList.toggle('checked', !checked);
}
```

This requires managing: - Role announcement - Checked state - Keyboard interaction - Focus management - Visual styling

Always prefer native elements when possible.

Form Validation Patterns

HTML5 validation provides declarative constraints:

```
<form novalidate> <!-- Disable native UI, handle validation manually -->
  <label for="email">Email</label>
  <input
    type="email"
    id="email"
    required
    pattern="[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,}$"
    aria-describedby="email-error email-hint"
  >
  <span id="email-hint">We'll never share your email</span>
  <span id="email-error" role="alert" aria-live="assertive"></span>

  <button type="submit">Submit</button>
</form>
```

```
const form = document.querySelector('form');
const emailInput = document.getElementById('email');
const errorSpan = document.getElementById('email-error');

emailInput.addEventListener('blur', validateEmail);
form.addEventListener('submit', handleSubmit);

function validateEmail() {
  if (emailInput.validity.valueMissing) {
    showError('Email is required');
    emailInput.setAttribute('aria-invalid', 'true');
  } else if (emailInput.validity.typeMismatch) {
    showError('Please enter a valid email address');
    emailInput.setAttribute('aria-invalid', 'true');
  } else if (emailInput.validity.patternMismatch) {
    showError('Email format is incorrect');
    emailInput.setAttribute('aria-invalid', 'true');
  } else {
```

```
    clearError();
    emailInput.setAttribute('aria-invalid', 'false');
  }
}

function showError(message) {
  errorSpan.textContent = message;
  // aria-live="assertive" announces error immediately
}

function clearError() {
  errorSpan.textContent = '';
}

function handleSubmit(e) {
  e.preventDefault();

  // Validate all fields
  const isValid = form.checkValidity();

  if (!isValid) {
    // Find first invalid field
    const firstInvalid = form.querySelector(':invalid');
    firstInvalid.focus();
    return;
  }

  // Submit form
  submitForm(new FormData(form));
}
```

Constraint Validation API provides programmatic validation access: - `validity.valueMissing` - required field empty - `validity.typeMismatch` - incorrect type (email, url, number) - `validity.patternMismatch` - doesn't match pattern - `validity.tooLong/tooShort` - length constraints - `validity.rangeOverflow/rangeUnderflow` - number range - `validity.stepMismatch` - doesn't match step - `validity.customError` - custom validation via setCustomValidity()

ARIA Live Regions

Live regions announce dynamic content changes to screen readers without moving focus:

```
<div aria-live="polite" aria-atomic="true" role="status">
  Items in cart: <span id="cart-count">0</span>
</div>

<div aria-live="assertive" role="alert">
  <!-- Critical errors appear here -->
```

```
</div>

<div aria-live="off">
  <!-- Updates not announced -->
</div>
```

- `aria-live="polite"` - announces when user is idle
- `aria-live="assertive"` - announces immediately, interrupting other speech
- `aria-atomic="true"` - announces entire region, not just changed part
- `role="status"` - implies aria-live="polite"
- `role="alert"` - implies aria-live="assertive"

Web Components Deep Dive

Web Components provide framework-agnostic component encapsulation through three technologies: Custom Elements, Shadow DOM, and HTML Templates.

Custom Elements Lifecycle

```
class UserCard extends HTMLElement {
  constructor() {
    super();
    // Element created, but not attached to DOM
    // Don't access attributes or children here
    this._shadowRoot = this.attachShadow({ mode: 'open' });
  }

  connectedCallback() {
    // Element added to DOM
    // Safe to access attributes and render
    this.render();
    this.attachEventListeners();
  }

  disconnectedCallback() {
    // Element removed from DOM
    // Clean up: remove event listeners, cancel timers
    this.cleanup();
  }

  adoptedCallback() {
    // Element moved to new document
    // Rare, occurs with document.adoptNode()
  }

  attributeChangedCallback(name, oldValue, newValue) {
    // Observed attribute changed
    if (oldValue !== newValue) {
      this.render();
```

```
  }
}

// Specify which attributes trigger attributeChangedCallback
static get observedAttributes() {
  return ['name', 'avatar', 'email'];
}

render() {
  const name = this.getAttribute('name') || 'Anonymous';
  const avatar = this.getAttribute('avatar') || 'default.jpg';
  const email = this.getAttribute('email') || '';

  this._shadowRoot.innerHTML = `
    <style>
      :host {
        display: block;
        border: 1px solid #ddd;
        border-radius: 8px;
        padding: 16px;
      }

      :host([hidden]) {
        display: none;
      }

      .card {
        display: flex;
        gap: 16px;
      }

      img {
        width: 64px;
        height: 64px;
        border-radius: 50%;
      }
    </style>

    <div class="card">
      <img src="${avatar}" alt="${name}">
      <div>
        <h3>${name}</h3>
        <p>${email}</p>
        <slot></slot> <!-- Content projection -->
      </div>
    </div>
  `;
```

```
  }

  attachEventListeners() {
    this._shadowRoot.querySelector('.card').addEventListener('click', () => {
      this.dispatchEvent(new CustomEvent('user-selected', {
        detail: { name: this.getAttribute('name') },
        bubbles: true,
        composed: true // Event crosses shadow boundary
      }));
    });
  }

  cleanup() {
    // Remove listeners if stored as references
  }
}

// Register custom element
customElements.define('user-card', UserCard);

// Usage
/*
<user-card name="John Doe" email="john@example.com" avatar="john.jpg">
  <button>View Profile</button>
</user-card>
*/
```

Shadow DOM Encapsulation

Shadow DOM creates isolated DOM trees with style and markup encapsulation:

```
class StyledButton extends HTMLElement {
  constructor() {
    super();
    const shadow = this.attachShadow({ mode: 'open' });

    shadow.innerHTML = `
      <style>
        /* Styles scoped to shadow DOM */
        button {
          background: blue;
          color: white;
          padding: 10px 20px;
          border: none;
          cursor: pointer;
        }

        /* ::slotted() styles content projected into slots */
```

```
        ::slotted(svg) {
          width: 16px;
          height: 16px;
          margin-right: 8px;
        }

        /* :host styles the custom element itself */
        :host {
          display: inline-block;
        }

        /* :host() applies when element matches selector */
        :host(.primary) button {
          background: green;
        }

        /* :host-context() applies when ancestor matches */
        :host-context(.dark-theme) button {
          background: #333;
        }
      </style>

      <button>
        <slot name="icon"></slot>
        <slot>Default Text</slot>
      </button>
    `;
  }
}

customElements.define('styled-button', StyledButton);

// Usage with named slots
/*
<styled-button class="primary">
  <svg slot="icon">...</svg>
  Click Me
</styled-button>
*/
```

Shadow DOM boundaries: - Styles don't leak in or out (except inherited properties like color, font) - DOM queries don't cross boundary - Events cross boundary if `composed: true` - `<slot>` projects light DOM content into shadow DOM

CSS Variables pierce shadow boundary:

```
shadow.innerHTML = `
  <style>
```

```
    button {
      background: var(--button-bg, blue);
      color: var(--button-color, white);
    }
  </style>
  <button><slot></slot></button>
`;

// External styles can customize via CSS variables
/*
<style>
  styled-button {
    --button-bg: red;
    --button-color: yellow;
  }
</style>
*/
```

Template Element

Templates hold inert markup for cloning:

```
<template id="product-card">
  <style>
    .card {
      border: 1px solid #ddd;
      padding: 16px;
    }
  </style>

  <div class="card">
    <h3 class="title"></h3>
    <p class="description"></p>
    <span class="price"></span>
  </div>
</template>
```

```
function createProductCard(product) {
  const template = document.getElementById('product-card');
  const clone = template.content.cloneNode(true);

  // Modify clone
  clone.querySelector('.title').textContent = product.name;
  clone.querySelector('.description').textContent = product.description;
  clone.querySelector('.price').textContent = `${product.price}`;

  return clone;
}
```

```
// Usage
const container = document.getElementById('products');
products.forEach(product => {
  container.appendChild(createProductCard(product));
});
```

Templates are more performant than creating elements via `createElement()` or setting `innerHTML` for repeated structures.

CSS Fundamentals and Advanced Concepts

The CSS Cascade in Detail

Understanding cascade order prevents specificity wars and allows predictable styling.

Cascade Sorting Algorithm

When multiple declarations apply to an element, the cascade determines which wins through this ordered criteria:

1. Origin and Importance
    - User agent !important
    - User !important
    - Author !important
    - Author normal
    - User normal
    - User agent normal
2. Context (shadow DOM)
    - Styles in inner shadow tree
    - Styles in outer shadow tree
    - Styles in light DOM
3. Layers
    - Unlayered styles
    - Layered styles (in declaration order)
4. Specificity
    - (inline, id, class, element)
5. Order of Appearance
    - Later declarations win

Cascade Layers Example:

```
/* Define layer order first */
@layer reset, base, components, utilities;

/* Reset layer - lowest priority */
@layer reset {
  * {
    margin: 0;
    padding: 0;
  }
```

```
  h1 {
    font-size: 2em; /* Will be overridden by later layers */
  }
}

/* Base layer */
@layer base {
  h1 {
    font-size: 2.5em;
    color: #333;
  }
}

/* Component layer */
@layer components {
  .card h1 {
    font-size: 1.5em; /* More specific, but still lower priority */
  }
}

/* Utilities layer - highest priority among layers */
@layer utilities {
  .text-2xl {
    font-size: 3em !important; /* !important needed to beat unlayered */
  }
}

/* Unlayered - beats all layers regardless of specificity */
h1 {
  color: blue; /* Wins over layered declarations */
}
```

Layers provide specificity inversion - later layers beat earlier layers regardless of selector specificity within them. This enables utility classes without !important.

Specificity Calculation Deep Dive

Specificity is a four-part tuple: (inline, id, class, element)

```
/* (0, 0, 0, 1) */
h1 { }

/* (0, 0, 1, 1) */
h1.title { }

/* (0, 1, 0, 1) */
#header h1 { }
```

```css
/* (0, 1, 2, 2) */
#header nav.primary a:hover { }

/* (0, 0, 1, 0) - attribute selectors same as classes */
[type="checkbox"] { }

/* (0, 0, 1, 0) - pseudo-classes same as classes */
:hover { }

/* (0, 0, 0, 1) - pseudo-elements same as elements */
::before { }

/* (0, 0, 2, 2) - :is() uses most specific argument */
:is(#header, .main) h1 { }
/* Equivalent to #header h1 specificity: (0, 1, 0, 2) */

/* (0, 0, 0, 2) - :where() has zero specificity */
:where(#header, .main) h1 { }
/* Equivalent to h1 specificity: (0, 0, 0, 1) */

/* (0, 0, 0, 0) - :where() is perfect for resets */
:where(h1, h2, h3) {
  margin: 0; /* Easy to override */
}

/* (0, 0, 1, 0) - :not() uses argument specificity */
:not(#header) { }
/* Has id specificity even though negating it */

/* (0, 0, 0, 0) - universal selector */
* { }

/* Combinators don't add specificity */
div > p + span { } /* (0, 0, 0, 3) */
```

Inheritance and the Initial/Inherit Keywords

Not all properties inherit by default:

Inherited properties: - Text: color, font-family, font-size, font-weight, line-height, text-align - Lists: list-style - Cursor: cursor - Visibility: visibility

Non-inherited properties: - Box model: margin, padding, border, width, height - Positioning: position, top, left - Background: background-* - Display: display

Control inheritance explicitly:

```css
.parent {
  color: blue;
  border: 1px solid black;
```

```
}

.child {
  color: inherit; /* Explicitly inherit blue from parent */
  border: inherit; /* Force border inheritance (normally doesn't inherit) */

  margin: initial; /* Reset to spec default (0 for margin) */
  display: unset; /* inherit if inheritable, otherwise initial */
  all: revert; /* Reset all properties to user agent stylesheet */
}
```

Box Model Mastery

Content Box vs Border Box

```
/* Default box-sizing: content-box */
.content-box {
  width: 200px;
  padding: 20px;
  border: 5px solid;
  /* Total width = 200 + (20 * 2) + (5 * 2) = 250px */
}

/* Alternative: border-box */
.border-box {
  box-sizing: border-box;
  width: 200px;
  padding: 20px;
  border: 5px solid;
  /* Total width = 200px (includes padding and border) */
  /* Content width = 200 - 40 - 10 = 150px */
}

/* Global border-box (recommended) */
*, *::before, *::after {
  box-sizing: border-box;
}
```

Margin Collapsing

Adjacent vertical margins collapse to the larger margin:

```
<div class="box1"></div>
<div class="box2"></div>
```

```
.box1 {
  margin-bottom: 30px;
}

.box2 {
```

```
  margin-top: 20px;
}

/* Spacing between boxes = 30px, not 50px */
```

Margin collapsing occurs when: - Adjacent siblings (vertical margins only) - Parent and first/last child (if no padding/border/content separating) - Empty blocks (top and bottom margins collapse)

Prevent margin collapsing: - Add padding or border to parent - Use `overflow: auto` on parent (creates BFC) - Use flexbox or grid (items don't collapse) - Use `display: flow-root` (creates BFC without side effects)

```
.prevent-collapse {
  display: flow-root; /* Modern solution */
}

/* Or older methods */
.prevent-collapse-old {
  overflow: auto; /* Creates BFC but adds scrollbars if content overflows */
  padding-top: 1px; /* Separates parent/child margins but affects layout */
}
```

Block Formatting Context (BFC)

BFCs isolate layout. Elements inside BFC don't affect outside elements.

BFC created by: - `overflow` other than visible - `display: flow-root` - `float: left/right` - `position: absolute/fixed` - `display: inline-block` - Flex/grid items - `contain: layout/content/strict`

BFC behaviors: - Contains internal floats (no clearfix needed) - Excludes external floats - Margins don't collapse with outside elements

```
<div class="container">
  <div class="float">Float</div>
  <p>Text wraps around float</p>
</div>
```

```
.float {
  float: left;
  width: 100px;
  height: 100px;
  background: blue;
}

.container {
  /* Without BFC: container height collapses, ignoring float */

  /* With BFC: container expands to contain float */
  display: flow-root;
```

```
}

p {
  /* Create BFC to prevent text wrapping around float */
  display: flow-root;
  /* Now p starts below float instead of wrapping */
}
```

Layout Systems Deep Dive

Flexbox Internals

Flexbox lays out items along main axis with distribution algorithms.

```
.container {
  display: flex;
  flex-direction: row; /* main axis: left to right */
  flex-wrap: wrap; /* allow wrapping */
  justify-content: space-between; /* main axis distribution */
  align-items: center; /* cross axis alignment */
  align-content: flex-start; /* multi-line cross axis */
  gap: 16px; /* modern spacing (replaces margins) */
}

.item {
  /* flex: <grow> <shrink> <basis> */
  flex: 1 1 200px;

  /* Equivalent to: */
  flex-grow: 1;    /* take extra space proportionally */
  flex-shrink: 1;  /* shrink when constrained proportionally */
  flex-basis: 200px; /* initial size before growing/shrinking */

  /* Item-specific cross-axis alignment */
  align-self: flex-end;
}
```

Flex Size Calculation Algorithm:

1. Determine flex basis (initial size)
   - If flex-basis is auto, use content size or width/height
   - Otherwise use flex-basis value
2. Calculate free space
   - Free space = container size - sum of flex bases - gaps
3. Distribute free space
   - If positive (items can grow): distribute according to flex-grow
   - If negative (items must shrink): distribute according to flex-shrink weighted by base size

```css
/* Example with 600px container */
.container {
  width: 600px;
  display: flex;
  gap: 20px;
}

.item-1 {
  flex: 1 1 100px; /* basis: 100px, grow: 1 */
}

.item-2 {
  flex: 2 1 200px; /* basis: 200px, grow: 2 */
}

/* Total basis = 100 + 200 + 20 (gap) = 320px */
/* Free space = 600 - 320 = 280px */
/* Grow ratio = 1:2 */
/* item-1 gets: 100 + (280 * 1/3) = 193.33px */
/* item-2 gets: 200 + (280 * 2/3) = 386.67px */
```

Common Flexbox Patterns:

```css
/* Perfect centering */
.center {
  display: flex;
  justify-content: center;
  align-items: center;
}

/* Sticky footer */
.page {
  display: flex;
  flex-direction: column;
  min-height: 100vh;
}

.content {
  flex: 1; /* Grows to push footer down */
}

/* Equal height columns */
.columns {
  display: flex;
  /* Items stretch by default (align-items: stretch) */
}

/* Space between items, except at edges */
```

```css
.spaced {
  display: flex;
  gap: 16px; /* Modern */
}

/* Old way without gap */
.spaced-old {
  display: flex;
  margin: -8px; /* Negative margin to offset */
}

.spaced-old > * {
  margin: 8px; /* Creates 16px gaps between items */
}

/* Holy grail layout */
.holy-grail {
  display: flex;
  flex-direction: column;
  min-height: 100vh;
}

.holy-grail-body {
  display: flex;
  flex: 1;
}

.holy-grail-nav {
  flex: 0 0 200px;
  order: -1; /* Move before content in source */
}

.holy-grail-content {
  flex: 1;
}

.holy-grail-ads {
  flex: 0 0 200px;
}
```

Grid Layout Mastery

Grid creates two-dimensional layouts with explicit rows and columns.

```css
.grid-container {
  display: grid;

  /* Define columns */
```

```
  grid-template-columns: 200px 1fr 2fr;
  /* Fixed | Flexible (1 part) | Flexible (2 parts) */

  /* Define rows */
  grid-template-rows: auto 1fr auto;
  /* Content height | Flexible | Content height */

  /* Gap between cells */
  gap: 20px 40px; /* row-gap column-gap */

  /* Named grid areas */
  grid-template-areas:
    "header header header"
    "sidebar content content"
    "footer footer footer";
}

.header {
  grid-area: header;
}

.sidebar {
  grid-area: sidebar;
}

.content {
  grid-area: content;
}

.footer {
  grid-area: footer;
}

/* Positioning items by line numbers */
.item {
  grid-column: 1 / 3; /* Start line 1, end line 3 (spans 2 columns) */
  grid-row: 2 / 4; /* Start line 2, end line 4 (spans 2 rows) */

  /* Or use span */
  grid-column: 1 / span 2;
  grid-row: 2 / span 2;

  /* Shorthand */
  grid-area: 2 / 1 / 4 / 3; /* row-start / col-start / row-end / col-end */
}
```

Advanced Grid Techniques:

```css
/* Responsive columns without media queries */
.auto-grid {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));
  gap: 16px;
}

/* auto-fit: collapses empty tracks */
/* auto-fill: keeps empty tracks */

/* Example with 800px container and 250px minimum:
   auto-fit: Creates 3 columns (250px + 16px gap fits 3 times)
   Each column gets: (800 - 32px gaps) / 3 = ~256px

   With 1 item:
   auto-fit: 1 column taking full width
   auto-fill: Still 3 columns, 2 empty */

/* Dense packing */
.masonry-like {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));
  grid-auto-flow: dense;
  /* Fills gaps with later items */
}

/* Asymmetric layouts */
.magazine {
  display: grid;
  grid-template-columns: repeat(12, 1fr);
  gap: 20px;
}

.feature {
  grid-column: 1 / 9; /* Spans 8 columns */
}

.sidebar-item {
  grid-column: 9 / 13; /* Spans 4 columns */
}

/* Overlapping items */
.overlap {
  display: grid;
  grid-template-columns: 1fr 1fr;
}
```

```css
.background {
  grid-column: 1 / 2;
  grid-row: 1;
  z-index: 0;
}

.foreground {
  grid-column: 1 / 3; /* Overlaps background */
  grid-row: 1;
  z-index: 1;
}

/* Subgrid (align nested grid with parent) */
.parent-grid {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
  gap: 20px;
}

.nested-grid {
  grid-column: span 2;
  display: grid;
  grid-template-columns: subgrid; /* Inherits parent's column tracks */
  /* Now child items align with parent grid */
}
```

Grid vs Flexbox Decision Matrix:

Use Grid when: - Two-dimensional layout needed - Overlapping items - Gap-based spacing - Named areas improve readability - Precise control over rows and columns

Use Flexbox when: - One-dimensional layout - Content-driven sizing - Simple alignment - Source order flexibility with `order` - Better browser support needed (older browsers)

Logical Properties for Internationalization

Logical properties adapt to writing direction (LTR/RTL) automatically:

```css
.element {
  /* Physical properties (not RTL-aware) */
  margin-left: 20px;
  padding-right: 10px;
  border-left: 1px solid black;
  left: 0;

  /* Logical properties (RTL-aware) */
  margin-inline-start: 20px; /* left in LTR, right in RTL */
  padding-inline-end: 10px; /* right in LTR, left in RTL */
  border-inline-start: 1px solid black; /* left in LTR, right in RTL */
```

```
  inset-inline-start: 0; /* left in LTR, right in RTL */

  /* Block axis (vertical in horizontal writing mode) */
  margin-block-start: 10px; /* top in horizontal mode */
  margin-block-end: 10px; /* bottom in horizontal mode */

  /* Shorthands */
  margin-inline: 20px 10px; /* start and end */
  padding-block: 10px; /* same start and end */

  /* Size properties */
  inline-size: 200px; /* width in horizontal mode */
  block-size: 100px; /* height in horizontal mode */
  max-inline-size: 600px; /* max-width */
  min-block-size: 400px; /* min-height */
}

/* Apply RTL */
html[dir="rtl"] .element {
  /* Logical properties automatically flip */
  /* No need for separate RTL rules */
}
```

Mapping physical to logical:

Horizontal writing mode (English): - inline = horizontal - block = vertical - start = left (LTR) or right (RTL) - end = right (LTR) or left (RTL)

Vertical writing mode (Japanese vertical text): - inline = vertical - block = horizontal - start/end adapt accordingly

Modern CSS Features

Container Queries - Component-Level Responsiveness

Container queries enable responsive components based on container size, not viewport:

```
.card-container {
  container-type: inline-size; /* Make element a container */
  container-name: card; /* Optional name */
}

.card {
  display: grid;
  grid-template-columns: 1fr;
}

/* Query container width */
@container (min-width: 400px) {
```

```
  .card {
    grid-template-columns: 200px 1fr;
  }
}

@container (min-width: 600px) {
  .card {
    grid-template-columns: 250px 1fr 200px;
  }
}

/* Named container queries */
@container card (min-width: 500px) {
  .card-title {
    font-size: 2em;
  }
}

/* Container query units */
.card-title {
  font-size: 5cqw; /* 5% of container width */
}

/* Units available:
   cqw: 1% of container width
   cqh: 1% of container height
   cqi: 1% of container inline size
   cqb: 1% of container block size
   cqmin: smaller of cqi or cqb
   cqmax: larger of cqi or cqb */
```

Container types: - `inline-size`: Query inline dimension (width in horizontal writing) - `size`: Query both dimensions (element doesn't affect parent size) - `normal`: Not a query container

Subgrid - Nested Grid Alignment

Subgrid allows nested grids to participate in parent grid:

```
.parent {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
  gap: 20px;
}

.child {
  grid-column: span 2;
  display: grid;
  grid-template-columns: subgrid; /* Inherit parent columns */
```

```
  grid-template-rows: subgrid; /* Inherit parent rows */
}

/* Use case: card with header/content aligned across cards */
.card-grid {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(300px, 1fr));
  grid-auto-rows: auto auto 1fr auto; /* header, meta, content, footer */
  gap: 20px;
}

.card {
  display: grid;
  grid-template-rows: subgrid;
  grid-row: span 4; /* Span all 4 rows */
}

.card-header {
  grid-row: 1;
}

.card-meta {
  grid-row: 2;
}

.card-content {
  grid-row: 3;
}

.card-footer {
  grid-row: 4;
}

/* All card headers align, all footers align, content area flexible */
```

CSS Custom Properties (Variables) Deep Dive

Custom properties cascade and inherit, enabling powerful patterns:

```
:root {
  /* Global theme variables */
  --color-primary: #007bff;
  --color-secondary: #6c757d;
  --spacing-unit: 8px;

  /* Computed values */
  --spacing-2: calc(var(--spacing-unit) * 2);
  --spacing-3: calc(var(--spacing-unit) * 3);
```

```
}

.dark-theme {
  /* Override for dark theme */
  --color-primary: #0d6efd;
  --color-secondary: #adb5bd;
}

.component {
  /* Local variables with fallbacks */
  --component-bg: var(--color-primary, blue);
  --component-padding: var(--spacing-2, 16px);

  background: var(--component-bg);
  padding: var(--component-padding);
}

/* JavaScript manipulation */
/*
const root = document.documentElement;
root.style.setProperty('--color-primary', '#ff0000');

const value = getComputedStyle(root).getPropertyValue('--color-primary');
*/

/* Custom properties in calc() */
.responsive-text {
  --min-font-size: 16;
  --max-font-size: 24;
  --min-viewport: 320;
  --max-viewport: 1200;

  font-size: calc(
    var(--min-font-size) * 1px +
    (var(--max-font-size) - var(--min-font-size)) *
    ((100vw - var(--min-viewport) * 1px) / (var(--max-viewport) - var(--min-viewport)))
  );

  /* Simpler with clamp */
  font-size: clamp(
    calc(var(--min-font-size) * 1px),
    1rem + 1vw,
    calc(var(--max-font-size) * 1px)
  );
}

/* Invalid at computed value time */
```

```css
.invalid {
  --color: 20px; /* Not a color, but valid custom property */
  background: var(--color); /* Invalid, falls back to initial (transparent) */
  background: var(--color, red); /* Falls back to red */
}

/* Empty value */
.empty {
  --gap: ; /* Empty, but valid */
  margin-top: var(--gap, 20px); /* Uses 20px */
}

/* Space-separated values */
.space-sep {
  --shadow-color: 0 0 0 rgba(0,0,0,0.3);
  box-shadow: 2px 2px 4px var(--shadow-color);
}
```

Advanced Use Cases:

```css
/* Responsive design system */
:root {
  --container-width: 90vw;
  --max-container-width: 1200px;
}

@media (min-width: 768px) {
  :root {
    --container-width: 85vw;
  }
}

@media (min-width: 1024px) {
  :root {
    --container-width: 80vw;
  }
}

.container {
  width: min(var(--container-width), var(--max-container-width));
}

/* Component variants */
.button {
  --button-bg: var(--color-primary);
  --button-color: white;
  --button-padding: var(--spacing-2);
```

```
  background: var(--button-bg);
  color: var(--button-color);
  padding: var(--button-padding);
}

.button--secondary {
  --button-bg: var(--color-secondary);
}

.button--large {
  --button-padding: var(--spacing-3);
}

/* Contextual overrides */
.card {
  --card-padding: var(--spacing-3);
}

.sidebar .card {
  --card-padding: var(--spacing-2);
}

/* All cards in sidebar automatically use smaller padding */
```

Math Functions: clamp(), min(), max()

```
/* Fluid typography without media queries */
.fluid-text {
  /* clamp(minimum, preferred, maximum) */
  font-size: clamp(1rem, 2vw + 1rem, 3rem);
  /* Never smaller than 1rem, never larger than 3rem, scales with viewport */
}

/* Responsive containers */
.container {
  /* Take 90% of viewport, but never exceed 1200px */
  width: min(90vw, 1200px);

  /* Take larger of 50% viewport or 300px */
  min-width: max(50vw, 300px);
}

/* Grid with minimum column size */
.grid {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(max(200px, 30%), 1fr));
  /* Columns at least 200px or 30% of container, whichever is larger */
}
```

```css
/* Fluid spacing */
.section {
  padding: clamp(1rem, 5vw, 5rem);
  /* Padding grows with viewport between 1rem and 5rem */
}
```

Aspect Ratio

```css
.video-container {
  aspect-ratio: 16 / 9;
  width: 100%;
  /* Height automatically calculated */
}

.square {
  aspect-ratio: 1;
  width: 100px;
  /* Height becomes 100px */
}

/* Replaces padding-bottom hack */
/* Old way: */
.old-aspect-ratio {
  width: 100%;
  padding-bottom: 56.25%; /* 9/16 * 100% */
  position: relative;
}

.old-aspect-ratio > * {
  position: absolute;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
}
```

JavaScript Fundamentals and Advanced Concepts

Execution Context and Scope Chain Deep Dive

Understanding how JavaScript engines manage execution contexts is fundamental to mastering the language.

Execution Context Components

Every execution context has three components:

1. Variable Environment
   - Environment record (stores var declarations and function declarations)
   - Outer environment reference (for scope chain)

2. Lexical Environment
    - Environment record (stores let/const declarations)
    - Outer environment reference
    - this binding
3. this Binding
    - Determined at call time

Execution Context Lifecycle:

```
// Creation Phase
function example(a, b) {
  var x = 10;
  let y = 20;
  const z = 30;

  function inner() {
    console.log(x, y, z);
  }
}

// When example() called:
// 1. Creation Phase:
//    - Arguments object created {a, b}
//    - Function declarations hoisted: inner
//    - var declarations hoisted: x = undefined
//    - let/const in TDZ: y, z
//    - this binding set
//    - Outer environment reference set

// 2. Execution Phase:
//    - Assignments executed
//    - let/const initialized
//    - Code runs line by line
```

Visualizing scope chain:

```
const global = 'global';

function outer() {
  const outerVar = 'outer';

  function middle() {
    const middleVar = 'middle';

    function inner() {
      const innerVar = 'inner';
      console.log(innerVar, middleVar, outerVar, global);
      // Scope chain: inner -> middle -> outer -> global
    }
```

```
    return inner;
  }

  return middle;
}

const middleFn = outer();
const innerFn = middleFn();
innerFn();

/* Scope chain resolution:
   1. innerVar: found in inner's environment
   2. middleVar: not in inner, check outer reference -> found in middle
   3. outerVar: not in inner or middle, check outer -> found in outer
   4. global: traverse to global environment
*/
```

Closures - Memory Model and Performance

Closures retain entire scope chain, which has memory implications:

```
function createHeavyClosures() {
  const hugeArray = new Array(1000000).fill('data');

  return {
    // This closure retains hugeArray even though it doesn't use it
    getLength: function() {
      return 'Some string'; // Doesn't use hugeArray
    },
    // This closure also retains hugeArray
    logFirst: function() {
      console.log(hugeArray[0]); // Uses hugeArray
    }
  };
}

// Both closures keep hugeArray in memory
// Even though getLength doesn't use it

// Better: Nullify references when done
function createHeavyClosuresBetter() {
  let hugeArray = new Array(1000000).fill('data');
  const firstElement = hugeArray[0];

  const result = {
    getLength: function() {
      return 'Some string';
    },
```

```
    logFirst: function() {
      console.log(firstElement); // Only closes over firstElement
    },
    cleanup: function() {
      hugeArray = null; // Allow garbage collection
    }
  };

  hugeArray = null; // If not needed after setup
  return result;
}
```

Classic Closure Pitfall - Loop:

```
// Problem: All closures reference same variable
for (var i = 0; i < 5; i++) {
  setTimeout(function() {
    console.log(i); // Prints 5 five times
  }, i * 1000);
}

// Why: var is function-scoped, all closures see same i
// When setTimeout callbacks run, i === 5

// Solution 1: IIFE creates new scope per iteration
for (var i = 0; i < 5; i++) {
  (function(capturedI) {
    setTimeout(function() {
      console.log(capturedI); // Prints 0, 1, 2, 3, 4
    }, capturedI * 1000);
  })(i);
}

// Solution 2: let creates new binding per iteration
for (let i = 0; i < 5; i++) {
  setTimeout(function() {
    console.log(i); // Prints 0, 1, 2, 3, 4
  }, i * 1000);
}

// Behind the scenes, roughly equivalent to:
{
  let i = 0;
  {
    let $$i = i; // New binding
    setTimeout(function() { console.log($$i); }, 0);
  }
  i++;
```

```
  {
    let $$i = i; // New binding
    setTimeout(function() { console.log($$i); }, 1000);
  }
  // ... etc
}
```

Practical Closure Patterns:

```
// Module Pattern (private variables)
const counterModule = (function() {
  let count = 0; // Private variable

  return {
    increment() {
      count++;
      return count;
    },
    decrement() {
      count--;
      return count;
    },
    getCount() {
      return count;
    }
  };
})();

counterModule.increment(); // 1
counterModule.increment(); // 2
console.log(counterModule.count); // undefined (private)

// Factory Function
function createCalculator(initialValue) {
  let value = initialValue;

  return {
    add(n) {
      value += n;
      return this;
    },
    subtract(n) {
      value -= n;
      return this;
    },
    multiply(n) {
      value *= n;
      return this;
```

```
    },
    getValue() {
      return value;
    }
  };
}

const calc = createCalculator(10);
calc.add(5).multiply(2).subtract(10); // Method chaining
console.log(calc.getValue()); // 20

// Partial Application
function multiply(a, b) {
  return a * b;
}

function partial(fn, ...fixedArgs) {
  return function(...remainingArgs) {
    return fn(...fixedArgs, ...remainingArgs);
  };
}

const double = partial(multiply, 2);
const triple = partial(multiply, 3);

console.log(double(5)); // 10
console.log(triple(5)); // 15

// Memoization
function memoize(fn) {
  const cache = new Map();

  return function(...args) {
    const key = JSON.stringify(args);

    if (cache.has(key)) {
      console.log('Cache hit');
      return cache.get(key);
    }

    console.log('Computing...');
    const result = fn(...args);
    cache.set(key, result);
    return result;
  };
}
```

```
function expensiveCalculation(n) {
  let sum = 0;
  for (let i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}

const memoized = memoize(expensiveCalculation);
memoized(1000000); // Computing... (slow)
memoized(1000000); // Cache hit (instant)

// Event Handler with State
function createClickHandler() {
  let clickCount = 0;

  return function(event) {
    clickCount++;
    console.log(`Clicked ${clickCount} times`);

    if (clickCount >= 5) {
      event.target.removeEventListener('click', this);
      console.log('Handler removed');
    }
  };
}

const button = document.querySelector('button');
button.addEventListener('click', createClickHandler());
```

Hoisting Mechanics

Hoisting moves declarations to top of scope during compilation:

```
console.log(x); // undefined (not ReferenceError)
var x = 5;

// Interpreted as:
var x;
console.log(x); // undefined
x = 5;

// Functions fully hoisted
sayHello(); // Works fine
function sayHello() {
  console.log('Hello');
}
```

```
// Function expressions not hoisted
sayGoodbye(); // TypeError: sayGoodbye is not a function
var sayGoodbye = function() {
  console.log('Goodbye');
};

// Interpreted as:
var sayGoodbye;
sayGoodbye(); // undefined is not a function
sayGoodbye = function() { console.log('Goodbye'); };

// let/const in Temporal Dead Zone
console.log(a); // ReferenceError: Cannot access 'a' before initialization
let a = 10;

console.log(b); // ReferenceError: Cannot access 'b' before initialization
const b = 20;

// TDZ spans from block start to declaration
{
  // TDZ starts
  console.log(x); // ReferenceError
  let x = 5; // TDZ ends
}

// Function parameters in TDZ
function example(a = b, b = 2) {
  // Error: Cannot access 'b' before initialization
  // a's default value evaluated before b declared
}

example(undefined, 2); // Error

// Correct order
function example2(b = 2, a = b) {
  return a + b;
}

example2(); // 4 (b=2, a=2)
```

Hoisting with Classes:

```
const instance = new MyClass(); // ReferenceError
class MyClass {}

// Classes not hoisted like function declarations
// Behave like let/const (TDZ)
```

this Binding - Complete Rules

`this` binding determined by call-site (where function called):

```
// 1. Default Binding (function call)
function standalone() {
  console.log(this);
}

standalone(); // undefined (strict mode) or window (non-strict)

// 2. Implicit Binding (method call)
const obj = {
  name: 'Object',
  greet() {
    console.log(this.name);
  }
};

obj.greet(); // 'Object' (this = obj)

// Lost implicit binding
const greet = obj.greet;
greet(); // undefined (this = window/undefined)

// 3. Explicit Binding (call/apply/bind)
function sayName(greeting) {
  console.log(`${greeting}, ${this.name}`);
}

const person = { name: 'John' };

sayName.call(person, 'Hello'); // Hello, John
sayName.apply(person, ['Hello']); // Hello, John

const boundSayName = sayName.bind(person);
boundSayName('Hello'); // Hello, John (always uses person as this)

// 4. new Binding (constructor call)
function Person(name) {
  this.name = name;
  // Implicit: return this;
}

const john = new Person('John');
console.log(john.name); // John (this = new object)

// 5. Arrow Function Binding (lexical this)
```

```
const obj2 = {
  name: 'Object2',
  regular: function() {
    console.log(this.name);

    const arrow = () => {
      console.log(this.name); // Inherits this from regular()
    };

    arrow();
  }
};

obj2.regular(); // Object2, Object2

// Arrow functions ignore call/apply/bind for this
const obj3 = { name: 'Object3' };
const arrowFn = () => console.log(this);
arrowFn.call(obj3); // Still uses lexical this, not obj3

// Binding Precedence
// new > explicit (bind) > implicit > default

function test() {
  console.log(this.value);
}

const obj4 = { value: 1 };
const obj5 = { value: 2 };

test.call(obj4); // 1 (explicit)
obj5.test = test;
obj5.test(); // 2 (implicit)
obj5.test.call(obj4); // 1 (explicit > implicit)

const boundTest = test.bind(obj4);
boundTest.call(obj5); // 1 (bind > call)
obj5.boundTest = boundTest;
obj5.boundTest(); // 1 (bind > implicit)

const instance = new boundTest(); // { value: undefined } (new > bind)
```

Real-world this pitfalls and solutions:

```
class Component {
  constructor() {
    this.state = { count: 0 };
```

```
    // Problem: Lost binding
    document.querySelector('#btn1').addEventListener('click', this.handleClick);
    // When clicked, this = button element, not component

    // Solution 1: Arrow function
    document.querySelector('#btn2').addEventListener('click',
      (e) => this.handleClick(e)
    );

    // Solution 2: Bind in constructor
    this.handleClick = this.handleClick.bind(this);
    document.querySelector('#btn3').addEventListener('click', this.handleClick);
  }

  handleClick() {
    console.log(this.state.count);
    this.state.count++;
  }

  // Solution 3: Class field with arrow function
  handleClickArrow = () => {
    console.log(this.state.count);
    this.state.count++;
  }
}

// React patterns
class ReactComponent {
  // Old way: bind in constructor
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // this correctly bound
  }

  // Modern way: class field
  handleClickModern = () => {
    // this automatically bound
  }

  render() {
    return (
      <div>
        <button onClick={this.handleClick}>Old Way</button>
```

```
        <button onClick={this.handleClickModern}>Modern Way</button>
        <button onClick={() => this.handleClick()}>Arrow Wrapper</button>
      </div>
    );
  }
}
```

Prototypes and Inheritance

Every object has internal `[[Prototype]]` link (accessed via `__proto__` or `Object.getPrototypeOf()`):

```
// Creating objects with specific prototype
const animal = {
  type: 'Animal',
  speak() {
    console.log(`${this.name} makes a sound`);
  }
};

// Method 1: Object.create()
const dog = Object.create(animal);
dog.name = 'Rex';
dog.speak(); // Rex makes a sound

console.log(Object.getPrototypeOf(dog) === animal); // true

// Method 2: Constructor functions
function Animal(name) {
  this.name = name;
}

Animal.prototype.speak = function() {
  console.log(`${this.name} makes a sound`);
};

const cat = new Animal('Whiskers');
cat.speak(); // Whiskers makes a sound

console.log(cat.constructor === Animal); // true
console.log(Object.getPrototypeOf(cat) === Animal.prototype); // true

// Method 3: ES6 Classes (syntactic sugar)
class AnimalClass {
  constructor(name) {
    this.name = name;
  }
```

```
  speak() {
    console.log(`${this.name} makes a sound`);
  }

  static info() {
    console.log('This is an Animal class');
  }
}

const bird = new AnimalClass('Tweety');
bird.speak(); // Tweety makes a sound
AnimalClass.info(); // This is an Animal class

// Inheritance
class Dog extends AnimalClass {
  constructor(name, breed) {
    super(name); // Call parent constructor
    this.breed = breed;
  }

  speak() {
    console.log(`${this.name} barks`);
  }

  getInfo() {
    return `${this.name} is a ${this.breed}`;
  }
}

const rex = new Dog('Rex', 'German Shepherd');
rex.speak(); // Rex barks
console.log(rex.getInfo()); // Rex is a German Shepherd

// Prototype chain
console.log(rex instanceof Dog); // true
console.log(rex instanceof AnimalClass); // true
console.log(rex instanceof Object); // true

// Chain: rex -> Dog.prototype -> AnimalClass.prototype -> Object.prototype -> null
```

Prototype chain resolution:

```
const obj = {
  a: 1
};

// Lookup chain for obj.toString():
// 1. Check obj itself - not found
```

```
// 2. Check Object.prototype - found

obj.toString(); // [object Object]

// Shadowing
obj.toString = function() {
  return 'Custom toString';
};

obj.toString(); // Custom toString (shadows Object.prototype.toString)

// Check if property exists on object itself (not prototype)
console.log(obj.hasOwnProperty('a')); // true
console.log(obj.hasOwnProperty('toString')); // true (after shadowing)

// Property enumeration
for (let key in obj) {
  if (obj.hasOwnProperty(key)) {
    console.log(key); // Only own properties
  }
}

// Modern way
Object.keys(obj); // Own enumerable properties
Object.getOwnPropertyNames(obj); // All own properties (including non-enumerable)
Object.getOwnPropertySymbols(obj); // Own symbol properties
```

Implementing inheritance manually:

```
function Shape(x, y) {
  this.x = x;
  this.y = y;
}

Shape.prototype.move = function(dx, dy) {
  this.x += dx;
  this.y += dy;
};

function Circle(x, y, radius) {
  // Call parent constructor
  Shape.call(this, x, y);
  this.radius = radius;
}

// Set up prototype chain
Circle.prototype = Object.create(Shape.prototype);
Circle.prototype.constructor = Circle;
```

```
Circle.prototype.area = function() {
  return Math.PI * this.radius ** 2;
};

const circle = new Circle(0, 0, 5);
circle.move(10, 10);
console.log(circle.x, circle.y); // 10, 10
console.log(circle.area()); // 78.53981633974483
console.log(circle instanceof Circle); // true
console.log(circle instanceof Shape); // true
```

Type Coercion Deep Dive

JavaScript performs implicit type conversion in many scenarios:

```
// String coercion
console.log('5' + 3); // '53' (number -> string)
console.log('5' + true); // '5true' (boolean -> string)
console.log('5' + null); // '5null' (null -> 'null')
console.log('5' + undefined); // '5undefined'

// Numeric coercion
console.log('5' - 3); // 2 (string -> number)
console.log('5' * '2'); // 10
console.log('5' / '2'); // 2.5
console.log('5' % '2'); // 1
console.log('abc' - 3); // NaN

console.log(+'5'); // 5 (unary plus converts to number)
console.log(+'abc'); // NaN
console.log(+true); // 1
console.log(+false); // 0
console.log(+null); // 0
console.log(+undefined); // NaN

// Boolean coercion
if ('hello') { } // true (non-empty string)
if ('') { } // false (empty string)
if (0) { } // false
if (1) { } // true
if (null) { } // false
if(undefined) { } // false
if (NaN) { } // false
if ([]) { } // true (empty array is object)
if ({}) { } // true (empty object)

// Explicit boolean conversion
console.log(Boolean('hello')); // true
```

```
console.log(!!'hello'); // true (double NOT)

// Falsy values (only 7)
// false, 0, -0, 0n, '', null, undefined, NaN

// Equality coercion (==)
console.log(5 == '5'); // true (string coerced to number)
console.log(null == undefined); // true (special case)
console.log(false == 0); // true
console.log(true == 1); // true
console.log([] == false); // true ([] -> '' -> 0)
console.log([1] == 1); // true ([1] -> '1' -> 1)

// Strict equality (===) - no coercion
console.log(5 === '5'); // false
console.log(null === undefined); // false

// Object to primitive conversion
const obj = {
  valueOf() {
    return 42;
  },
  toString() {
    return 'Object';
  }
};

console.log(obj + 1); // 43 (valueOf() called)
console.log(String(obj)); // 'Object' (toString() called)
console.log(`${obj}`); // 'Object' (toString() for string context)

// Symbol.toPrimitive for full control
const obj2 = {
  [Symbol.toPrimitive](hint) {
    if (hint === 'number') {
      return 42;
    }
    if (hint === 'string') {
      return 'Object2';
    }
    return null; // hint === 'default'
  }
};

console.log(obj2 + 1); // 1 (default hint -> null + 1)
console.log(+obj2); // 42 (number hint)
console.log(`${obj2}`); // 'Object2' (string hint)
```

```
// Tricky cases
console.log([] + []); // '' (both to empty string)
console.log([] + {}); // '[object Object]'
console.log({} + []); // '[object Object]' or 0 (depends on context)
console.log({} + {}); // '[object Object][object Object]' or NaN
console.log(true + true); // 2
console.log(true + false); // 1
console.log(![] + []); // 'false'
```

Best practices: - Always use === and !== - Use explicit conversion: `Number()`, `String()`, `Boolean()` - For number checks, use `Number.isNaN()` not global `isNaN()` - For integer checks, use `Number.isInteger()`

ES6+ Features Deep Dive

Destructuring Assignment:

```
// Array destructuring
const [a, b, c] = [1, 2, 3];
console.log(a, b, c); // 1 2 3

// Skip elements
const [first, , third] = [1, 2, 3];
console.log(first, third); // 1 3

// Rest operator
const [head, ...tail] = [1, 2, 3, 4, 5];
console.log(head); // 1
console.log(tail); // [2, 3, 4, 5]

// Default values
const [x = 10, y = 20] = [1];
console.log(x, y); // 1 20

// Nested destructuring
const [a2, [b2, c2]] = [1, [2, 3]];
console.log(a2, b2, c2); // 1 2 3

// Object destructuring
const { name, age } = { name: 'John', age: 30, city: 'NYC' };
console.log(name, age); // John 30

// Rename variables
const { name: personName, age: personAge } = { name: 'John', age: 30 };
console.log(personName, personAge); // John 30

// Default values
const { name2 = 'Anonymous', age2 = 0 } = { name2: 'John' };
```

```
console.log(name2, age2); // John 0

// Rest operator
const { firstName, ...rest } = { firstName: 'John', lastName: 'Doe', age: 30 };
console.log(firstName); // John
console.log(rest); // { lastName: 'Doe', age: 30 }

// Nested destructuring
const { address: { street, city } } = {
  name: 'John',
  address: { street: '123 Main', city: 'NYC' }
};
console.log(street, city); // 123 Main NYC

// Function parameters
function greet({ name, age = 0 }) {
  console.log(`Hello ${name}, age ${age}`);
}

greet({ name: 'John' }); // Hello John, age 0
greet({ name: 'Jane', age: 25 }); // Hello Jane, age 25

// Swap variables
let x1 = 1, y1 = 2;
[x1, y1] = [y1, x1];
console.log(x1, y1); // 2 1

// Computed property names
const key = 'dynamicKey';
const { [key]: value } = { dynamicKey: 'value' };
console.log(value); // 'value'
```

Spread and Rest Operators:

```
// Array spread
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combined = [...arr1, ...arr2];
console.log(combined); // [1, 2, 3, 4, 5, 6]

// Cloning array (shallow)
const original = [1, 2, 3];
const clone = [...original];
clone[0] = 999;
console.log(original); // [1, 2, 3] (unchanged)

// Object spread
const obj1 = { a: 1, b: 2 };
```

```
const obj2 = { c: 3, d: 4 };
const merged = { ...obj1, ...obj2 };
console.log(merged); // { a: 1, b: 2, c: 3, d: 4 }

// Overriding properties
const base = { a: 1, b: 2 };
const extended = { ...base, b: 999, c: 3 };
console.log(extended); // { a: 1, b: 999, c: 3 }

// Shallow clone
const user = { name: 'John', address: { city: 'NYC' } };
const userClone = { ...user };
userClone.address.city = 'LA';
console.log(user.address.city); // 'LA' (address is shared reference)

// Function rest parameters
function sum(...numbers) {
  return numbers.reduce((acc, n) => acc + n, 0);
}

console.log(sum(1, 2, 3, 4)); // 10

// Must be last parameter
function example(first, second, ...rest) {
  console.log(first, second, rest);
}

example(1, 2, 3, 4, 5); // 1 2 [3, 4, 5]

// Spread in function calls
const numbers = [1, 2, 3];
console.log(Math.max(...numbers)); // 3

// Instead of
console.log(Math.max.apply(null, numbers)); // 3
```

Template Literals:

```
// String interpolation
const name = 'John';
const age = 30;
console.log(`Hello, I'm ${name} and I'm ${age} years old`);

// Multi-line strings
const multiline = `
  This is a
  multi-line
  string
```

```
`;

// Expression evaluation
console.log(`2 + 2 = ${2 + 2}`); // 2 + 2 = 4

// Nested templates
const nested = `Outer ${`Inner ${5}`}`;
console.log(nested); // Outer Inner 5

// Tagged templates (advanced)
function highlight(strings, ...values) {
  return strings.reduce((result, str, i) => {
    return result + str + (values[i] ? `<strong>${values[i]}</strong>` : '');
  }, '');
}

const name2 = 'John';
const age2 = 30;
const tagged = highlight`Hello, I'm ${name2} and I'm ${age2} years old`;
console.log(tagged);
// Hello, I'm <strong>John</strong> and I'm <strong>30</strong> years old

// Practical: SQL queries
function sql(strings, ...values) {
  // Build parameterized query
  let query = strings[0];
  values.forEach((value, i) => {
    query += '?' + strings[i + 1];
  });
  return { query, values };
}

const userId = 123;
const userName = "O'Connor";
const result = sql`SELECT * FROM users WHERE id = ${userId} AND name = ${userName}`;
console.log(result);
// { query: "SELECT * FROM users WHERE id = ? AND name = ?", values: [123, "O'Connor"]

// Raw strings
function raw(strings, ...values) {
  console.log(strings.raw); // Access raw strings (with escape sequences)
}

raw`Line 1\nLine 2`;
// strings.raw[0] === "Line 1\\nLine 2" (literal backslash-n)
// strings[0] === "Line 1\nLine 2" (interpreted newline)
```

Modules (ES6):

```
// math.js
export const PI = 3.14159;

export function add(a, b) {
  return a + b;
}

export function subtract(a, b) {
  return a - b;
}

// Default export
export default function multiply(a, b) {
  return a * b;
}

// Or at the end
function divide(a, b) {
  return a / b;
}

export { divide };

// main.js
import multiply, { add, subtract, PI } from './math.js';

console.log(add(2, 3)); // 5
console.log(multiply(2, 3)); // 6
console.log(PI); // 3.14159

// Rename imports
import { add as addition } from './math.js';
console.log(addition(2, 3)); // 5

// Import all
import * as Math from './math.js';
console.log(Math.add(2, 3)); // 5
console.log(Math.default(2, 3)); // 6 (default export)

// Re-export (for aggregating modules)
// index.js
export { add, subtract } from './math.js';
export { default as multiply } from './math.js';
export * from './geometry.js'; // Re-export all named exports
```

Module features: - Modules are singletons (imported once, shared) - Modules have

their own scope (not global) - `this` in modules is `undefined` (not `window`) - Imports are hoisted - Imports are read-only live bindings

```
// counter.js
export let count = 0;

export function increment() {
  count++;
}

// main.js
import { count, increment } from './counter.js';

console.log(count); // 0
increment();
console.log(count); // 1 (live binding, sees updated value)

count = 5; // Error: Assignment to constant variable (read-only)
```

Dynamic imports:

```
// Code splitting
button.addEventListener('click', async () => {
  const module = await import('./heavy-feature.js');
  module.initialize();
});

// Conditional loading
if (condition) {
  import('./module-a.js').then(moduleA => {
    moduleA.run();
  });
} else {
  import('./module-b.js').then(moduleB => {
    moduleB.run();
  });
}

// Top-level await (in modules)
const response = await fetch('/api/data');
const data = await response.json();
export default data;
```

This completes an in-depth look at JavaScript fundamentals. The document continues with similarly detailed coverage of all remaining topics.

Chapter 2

Chapter 2: Frontend Libraries and Frameworks (UI, State Management, Build Tools)

Modern frontend frameworks emerged to solve fundamental problems: keeping UI in sync with state, composing complex UIs from reusable components, managing application state, and optimizing bundle sizes. Each framework makes different tradeoffs regarding developer experience, performance, and flexibility.

2.1 React - Component Architecture and Virtual DOM

React revolutionized frontend development by treating UI as a function of state and introducing the Virtual DOM diffing algorithm.

2.1.1 Virtual DOM Reconciliation Deep Dive

React's Virtual DOM is a lightweight JavaScript representation of the actual DOM. When state changes, React creates a new Virtual DOM tree and uses a diffing algorithm to compute minimal DOM updates.

```
// Simplified Virtual DOM representation
const vdom = {
  type: 'div',
  props: {
    className: 'container',
    children: [
      {
        type: 'h1',
        props: {
          children: 'Hello World'
        }
      },
```

```
      {
        type: 'p',
        props: {
          children: 'Welcome to React'
        }
      }
    ]
  }
};

// React.createElement generates VDOM
const element = React.createElement(
  'div',
  { className: 'container' },
  React.createElement('h1', null, 'Hello World'),
  React.createElement('p', null, 'Welcome to React')
);

// JSX transpiles to createElement calls
/*
<div className="container">
  <h1>Hello World</h1>
  <p>Welcome to React</p>
</div>
*/
```

Reconciliation Algorithm:

React's diffing algorithm makes assumptions to achieve O(n) complexity (instead of $O(n^3)$ for general tree diff):

1. Elements of different types produce different trees
2. Developer can hint at stable child elements with keys

```
// Diffing different element types
// Old: <div><Counter /></div>
// New: <span><Counter /></span>
// React destroys div and Counter, builds new span and Counter from scratch

// Diffing same element types
// Old: <div className="old" />
// New: <div className="new" />
// React keeps same DOM node, updates className attribute

// List reconciliation with keys
// Without keys
<ul>
  <li>Item 1</li>
  <li>Item 2</li>
```

```
</ul>

// Add Item 0 at beginning
<ul>
  <li>Item 0</li>  {/* React thinks Item 1 changed to Item 0 */}
  <li>Item 1</li>  {/* React thinks Item 2 changed to Item 1 */}
  <li>Item 2</li>  {/* React creates new li */}
</ul>

// With keys
<ul>
  <li key="item-1">Item 1</li>
  <li key="item-2">Item 2</li>
</ul>

// Add Item 0 at beginning
<ul>
  <li key="item-0">Item 0</li>  {/* React creates new li */}
  <li key="item-1">Item 1</li>  {/* React keeps existing li */}
  <li key="item-2">Item 2</li>  {/* React keeps existing li */}
</ul>
```

Keys must be: - Stable (don't change between renders) - Unique among siblings - Avoid using array indices as keys when order can change

React Fiber Architecture

Fiber is React's reconciliation engine rewrite (React 16+), enabling:

1. Incremental rendering - split work into chunks
2. Pause/abort/resume work
3. Assign priority to different update types
4. Concurrent features

```
// Simplified Fiber node structure
const fiber = {
  type: 'div',              // Component type
  key: null,                // Key for reconciliation
  stateNode: domElement,    // Actual DOM node

  // Relationships
  return: parentFiber,      // Parent fiber
  child: firstChildFiber,   // First child
  sibling: nextSiblingFiber, // Next sibling

  // Work
  pendingProps: {},         // New props
  memoizedProps: {},        // Previous props
  memoizedState: {},        // Previous state
```

```
  updateQueue: [],          // State updates queue

  // Effects
  effectTag: 'UPDATE',      // Type of work (PLACEMENT, UPDATE, DELETION)
  nextEffect: nextFiber,    // Next fiber with effects

  // Scheduler
  lanes: 0,                 // Priority lanes
  alternate: oldFiber       // Previous fiber (double buffering)
};

// Work loop
function workLoop(deadline) {
  while (nextUnitOfWork && deadline.timeRemaining() > 1) {
    nextUnitOfWork = performUnitOfWork(nextUnitOfWork);
  }

  if (nextUnitOfWork) {
    // More work to do, schedule next chunk
    requestIdleCallback(workLoop);
  } else {
    // All work done, commit changes to DOM
    commitRoot();
  }
}

requestIdleCallback(workLoop);
```

Priority Levels:

- Immediate (user input, click)
- User-blocking (hover, scroll)
- Normal (data fetch, network response)
- Low (analytics)
- Idle (off-screen content)

React Hooks Deep Dive

Hooks enable functional components to use state and lifecycle features.

```
// useState implementation (simplified)
let state = [];
let setters = [];
let cursor = 0;

function useState(initialValue) {
  const cursorCopy = cursor; // Capture current cursor

  if (state[cursor] === undefined) {
```

```
    state[cursor] = initialValue;
  }

  const setter = (newValue) => {
    state[cursorCopy] = newValue;
    render(); // Trigger re-render
  };

  setters[cursor] = setter;

  const value = state[cursor];
  const setter = setters[cursor];

  cursor++;
  return [value, setter];
}

function render() {
  cursor = 0; // Reset cursor before each render
  // Re-run component function
}

// Why hooks must be called in same order
function Component() {
  // First render: cursor 0
  const [name, setName] = useState('John');

  // First render: cursor 1
  const [age, setAge] = useState(30);

  // If conditionally called, cursor mismatch on re-render
  if (condition) {
    const [email, setEmail] = useState(''); // DON'T DO THIS
  }

  // cursor 2 or 3 depending on condition - breaks state mapping!
  const [city, setCity] = useState('NYC');
}
```

useEffect Implementation:

```
let effects = [];
let effectCursor = 0;

function useEffect(callback, deps) {
  const hasNoDeps = !deps;
  const prevDeps = effects[effectCursor]?.deps;
```

```
  const hasChangedDeps = prevDeps
    ? deps.some((dep, i) => dep !== prevDeps[i])
    : true;

  if (hasNoDeps || hasChangedDeps) {
    effects[effectCursor] = { callback, deps };
  }

  effectCursor++;
}

function commitEffects() {
  effectCursor = 0;
  effects.forEach(effect => {
    const cleanup = effect.callback();
    effect.cleanup = cleanup; // Store cleanup function
  });
}

function runCleanups() {
  effects.forEach(effect => {
    if (effect.cleanup) {
      effect.cleanup();
    }
  });
}

// Lifecycle
// 1. Render phase: useEffect called, effects recorded
// 2. Commit phase: DOM updated
// 3. commitEffects: effect callbacks run
// 4. Next render: runCleanups for changed effects
```

useEffect vs useLayoutEffect:

```
// useEffect: Runs AFTER paint
useEffect(() => {
  // Measure DOM after browser paint
  const rect = element.getBoundingClientRect();
}, []);

// useLayoutEffect: Runs BEFORE paint (blocks rendering)
useLayoutEffect(() => {
  // Measure DOM before browser paint
  // Synchronous, blocks visual updates
  const rect = element.getBoundingClientRect();
  // Update state based on measurement
  setPosition(rect.top);
```

```
}, []);

// Use useLayoutEffect when:
// - Reading layout (offsetHeight, getBoundingClientRect)
// - Updating state based on layout (prevents flicker)

// Use useEffect for:
// - Data fetching
// - Subscriptions
// - Timers
// - Logging
```

useMemo and useCallback:

```
// useMemo: memoize computed values
const memoizedValue = useMemo(() => {
  return expensiveCalculation(a, b);
}, [a, b]); // Recalculate only when a or b change

// useCallback: memoize functions
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]); // Create new function only when a or b change

// Equivalent to:
const memoizedCallback = useMemo(() => {
  return () => doSomething(a, b);
}, [a, b]);

// Real-world example: preventing child re-renders
const Parent = () => {
  const [count, setCount] = useState(0);
  const [text, setText] = useState('');

  // Without useCallback: new function every render
  // Child re-renders even when count doesn't change
  const handleClick = () => {
    console.log(count);
  };

  // With useCallback: same function reference when count unchanged
  const handleClickMemo = useCallback(() => {
    console.log(count);
  }, [count]);

  return (
    <>
      <input value={text} onChange={e => setText(e.target.value)} />
```

```
      <Child onClick={handleClickMemo} />
    </>
  );
};

// Child uses React.memo to skip re-renders when props unchanged
const Child = React.memo(({ onClick }) => {
  console.log('Child rendered');
  return <button onClick={onClick}>Click</button>;
});
```

Custom Hooks Pattern:

```
// useFetch hook
function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    let cancelled = false;

    setLoading(true);

    fetch(url)
      .then(res => res.json())
      .then(data => {
        if (!cancelled) {
          setData(data);
          setLoading(false);
        }
      })
      .catch(error => {
        if (!cancelled) {
          setError(error);
          setLoading(false);
        }
      });

    return () => {
      cancelled = true; // Cleanup: ignore response if unmounted
    };
  }, [url]);

  return { data, loading, error };
}

// Usage
```

```
function UserProfile({ userId }) {
  const { data, loading, error } = useFetch(`/api/users/${userId}`);

  if (loading) return <Spinner />;
  if (error) return <Error message={error.message} />;
  return <Profile user={data} />;
}

// useLocalStorage hook
function useLocalStorage(key, initialValue) {
  const [storedValue, setStoredValue] = useState(() => {
    try {
      const item = window.localStorage.getItem(key);
      return item ? JSON.parse(item) : initialValue;
    } catch (error) {
      console.error(error);
      return initialValue;
    }
  });

  const setValue = (value) => {
    try {
      const valueToStore = value instanceof Function
        ? value(storedValue)
        : value;

      setStoredValue(valueToStore);
      window.localStorage.setItem(key, JSON.stringify(valueToStore));
    } catch (error) {
      console.error(error);
    }
  };

  return [storedValue, setValue];
}

// Usage
function Settings() {
  const [theme, setTheme] = useLocalStorage('theme', 'light');

  return (
    <button onClick={() => setTheme(theme === 'light' ? 'dark' : 'light')}>
      Toggle Theme (current: {theme})
    </button>
  );
}
```

```javascript
// useDebounce hook
function useDebounce(value, delay) {
  const [debouncedValue, setDebouncedValue] = useState(value);

  useEffect(() => {
    const handler = setTimeout(() => {
      setDebouncedValue(value);
    }, delay);

    return () => {
      clearTimeout(handler);
    };
  }, [value, delay]);

  return debouncedValue;
}

// Usage
function SearchBar() {
  const [searchTerm, setSearchTerm] = useState('');
  const debouncedSearchTerm = useDebounce(searchTerm, 500);

  useEffect(() => {
    if (debouncedSearchTerm) {
      // API call only after user stops typing for 500ms
      fetch(`/api/search?q=${debouncedSearchTerm}`)
        .then(res => res.json())
        .then(setResults);
    }
  }, [debouncedSearchTerm]);

  return (
    <input
      value={searchTerm}
      onChange={e => setSearchTerm(e.target.value)}
    />
  );
}
```

React Context API

Context provides a way to pass data through component tree without prop drilling.

```javascript
// Create context
const ThemeContext = React.createContext();

// Provider component
function App() {
```

```
  const [theme, setTheme] = useState('light');

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

// Consumer (hooks)
function ThemedButton() {
  const { theme, setTheme } = useContext(ThemeContext);

  return (
    <button
      style={{ background: theme === 'light' ? '#fff' : '#000' }}
      onClick={() => setTheme(theme === 'light' ? 'dark' : 'light')}
    >
      Toggle Theme
    </button>
  );
}

// Context with custom hook pattern
const UserContext = React.createContext();

function UserProvider({ children }) {
  const [user, setUser] = useState(null);

  const login = async (credentials) => {
    const user = await api.login(credentials);
    setUser(user);
  };

  const logout = () => {
    setUser(null);
  };

  return (
    <UserContext.Provider value={{ user, login, logout }}>
      {children}
    </UserContext.Provider>
  );
}

// Custom hook encapsulates context
function useUser() {
```

```
  const context = useContext(UserContext);

  if (!context) {
    throw new Error('useUser must be used within UserProvider');
  }

  return context;
}

// Usage
function Profile() {
  const { user, logout } = useUser();

  return (
    <div>
      <h1>{user.name}</h1>
      <button onClick={logout}>Logout</button>
    </div>
  );
}
```

Context Performance Optimization:

```
// Problem: Context value changes trigger all consumers to re-render
function App() {
  const [count, setCount] = useState(0);
  const [user, setUser] = useState({ name: 'John' });

  // New object every render, even if count unchanged
  return (
    <UserContext.Provider value={{ user, setUser, count, setCount }}>
      <Child />
    </UserContext.Provider>
  );
}

// Solution 1: useMemo
function App() {
  const [count, setCount] = useState(0);
  const [user, setUser] = useState({ name: 'John' });

  const value = useMemo(
    () => ({ user, setUser, count, setCount }),
    [user, count] // Only create new object when these change
  );

  return (
    <UserContext.Provider value={value}>
```

```
      <Child />
    </UserContext.Provider>
  );
}

// Solution 2: Split contexts
const UserContext = React.createContext();
const CountContext = React.createContext();

function App() {
  const [count, setCount] = useState(0);
  const [user, setUser] = useState({ name: 'John' });

  const userValue = useMemo(() => ({ user, setUser }), [user]);
  const countValue = useMemo(() => ({ count, setCount }), [count]);

  return (
    <UserContext.Provider value={userValue}>
      <CountContext.Provider value={countValue}>
        <Child />
      </CountContext.Provider>
    </UserContext.Provider>
  );
}

// Components only re-render when their context changes
function UserDisplay() {
  const { user } = useContext(UserContext);
  // Only re-renders when user changes, not count
  return <div>{user.name}</div>;
}

function Counter() {
  const { count, setCount } = useContext(CountContext);
  // Only re-renders when count changes, not user
  return <button onClick={() => setCount(c => c + 1)}>{count}</button>;
}
```

React Server Components (RSC)

RSC enable components to render on server, streaming HTML with client components embedded.

```
// Server Component (default)
// - Runs only on server
// - Can access backend resources directly
// - Cannot use hooks or event handlers
// - Zero bundle size
```

```
async function BlogPost({ id }) {
  // Direct database access on server
  const post = await db.posts.find(id);
  const comments = await db.comments.where({ postId: id });

  return (
    <article>
      <h1>{post.title}</h1>
      <p>{post.content}</p>

      {/* Client Component for interactivity */}
      <LikeButton postId={id} initialLikes={post.likes} />

      <CommentList comments={comments} />
    </article>
  );
}

// Client Component (marked with 'use client')
'use client';

function LikeButton({ postId, initialLikes }) {
  const [likes, setLikes] = useState(initialLikes);
  const [liked, setLiked] = useState(false);

  const handleLike = async () => {
    if (!liked) {
      setLikes(l => l + 1);
      setLiked(true);
      await fetch(`/api/posts/${postId}/like`, { method: 'POST' });
    }
  };

  return (
    <button onClick={handleLike}>
      {liked ? 'Liked' : 'Like'} ({likes})
    </button>
  );
}

// Rules:
// 1. Server Components can import Client Components
// 2. Client Components CANNOT import Server Components
// 3. But can pass Server Components as children/props

// Correct: Server Component as children
'use client';
```

```
function ClientLayout({ children }) {
  return (
    <div className="layout">
      {children} {/* children can be Server Component */}
    </div>
  );
}

// Usage
function Page() {
  return (
    <ClientLayout>
      <ServerComponent /> {/* Works! */}
    </ClientLayout>
  );
}
```

Benefits of RSC:

1. Zero bundle size - server components don't ship to client
2. Direct backend access - no API needed
3. Automatic code splitting - client components auto-split
4. Improved initial load - server sends rendered HTML
5. Streaming - progressive rendering as data arrives

```
// Streaming with Suspense
async function SlowComponent() {
  const data = await slowApiCall(); // Takes 3 seconds
  return <div>{data}</div>;
}

function Page() {
  return (
    <div>
      <h1>My Page</h1>

      {/* Shows fallback immediately, streams SlowComponent when ready */}
      <Suspense fallback={<Spinner />}>
        <SlowComponent />
      </Suspense>

      {/* Rest of page renders immediately */}
      <Footer />
    </div>
  );
}
```

Vue 3 - Composition API and Reactivity System

Vue 3 rebuilt its reactivity system using Proxies for better performance and type inference.

Reactivity Implementation

```
// Simplified Vue 3 reactivity
const targetMap = new WeakMap();
let activeEffect = null;

// track: record dependency
function track(target, key) {
  if (!activeEffect) return;

  let depsMap = targetMap.get(target);
  if (!depsMap) {
    targetMap.set(target, (depsMap = new Map()));
  }

  let dep = depsMap.get(key);
  if (!dep) {
    depsMap.set(key, (dep = new Set()));
  }

  dep.add(activeEffect);
}

// trigger: run effects depending on property
function trigger(target, key) {
  const depsMap = targetMap.get(target);
  if (!depsMap) return;

  const dep = depsMap.get(key);
  if (dep) {
    dep.forEach(effect => effect());
  }
}

// reactive: create proxy that tracks access
function reactive(target) {
  return new Proxy(target, {
    get(target, key, receiver) {
      const result = Reflect.get(target, key, receiver);
      track(target, key); // Track access
      return result;
    },

    set(target, key, value, receiver) {
      const result = Reflect.set(target, key, value, receiver);
```

```
      trigger(target, key); // Trigger effects
      return result;
    }
  });
}

// effect: run function and track dependencies
function effect(fn) {
  activeEffect = fn;
  fn(); // Run and track dependencies
  activeEffect = null;
}

// Example usage
const state = reactive({
  count: 0,
  message: 'Hello'
});

// This effect depends on count
effect(() => {
  console.log(`Count is: ${state.count}`);
});
// Logs: "Count is: 0"

state.count++; // Triggers effect
// Logs: "Count is: 1"

state.message = 'Hi'; // Doesn't trigger effect (not accessed in effect)
```

ref vs reactive:

```
import { ref, reactive, computed, watch } from 'vue';

// ref: wraps primitive values
const count = ref(0);
console.log(count.value); // Access via .value
count.value++; // Update via .value

// In template, auto-unwrapped
// <div>{{ count }}</div>  // No .value needed

// reactive: for objects
const state = reactive({
  count: 0,
  nested: {
    value: 1
  }
```

```
});

console.log(state.count); // Direct access
state.count++; // Direct update

// Computed: derived state
const doubled = computed(() => count.value * 2);
console.log(doubled.value); // 2 (if count is 1)

// Computed with setter
const fullName = computed({
  get() {
    return `${firstName.value} ${lastName.value}`;
  },
  set(newValue) {
    [firstName.value, lastName.value] = newValue.split(' ');
  }
});

// watch: side effects
watch(count, (newValue, oldValue) => {
  console.log(`Count changed from ${oldValue} to ${newValue}`);
});

// watch multiple sources
watch([count, state], ([newCount, newState], [oldCount, oldState]) => {
  console.log('Either count or state changed');
});

// watchEffect: auto-track dependencies
watchEffect(() => {
  console.log(`Count is ${count.value}`);
  // Automatically re-runs when count changes
});
```

Composition API Patterns:

```
// Component with Composition API
<template>
  <div>
    <input v-model="searchQuery" placeholder="Search..." />
    <ul>
      <li v-for="user in filteredUsers" :key="user.id">
        {{ user.name }}
      </li>
    </ul>
  </div>
</template>
```

```
<script setup>
import { ref, computed, onMounted } from 'vue';

// Reactive state
const searchQuery = ref('');
const users = ref([]);

// Computed property
const filteredUsers = computed(() => {
  return users.value.filter(user =>
    user.name.toLowerCase().includes(searchQuery.value.toLowerCase())
  );
});

// Lifecycle hook
onMounted(async () => {
  const response = await fetch('/api/users');
  users.value = await response.json();
});
</script>

// Composable (reusable logic)
// composables/useFetch.js
export function useFetch(url) {
  const data = ref(null);
  const error = ref(null);
  const loading = ref(false);

  async function fetchData() {
    loading.value = true;
    error.value = null;

    try {
      const response = await fetch(url.value || url);
      data.value = await response.json();
    } catch (e) {
      error.value = e;
    } finally {
      loading.value = false;
    }
  }

  // Auto-fetch on mount
  onMounted(fetchData);

  // Re-fetch when URL changes
  if (isRef(url)) {
```

```
    watch(url, fetchData);
  }

  return { data, error, loading, refetch: fetchData };
}

// Usage in component
<script setup>
import { ref } from 'vue';
import { useFetch } from './composables/useFetch';

const userId = ref(1);
const { data: user, loading, error } = useFetch(
  computed(() => `/api/users/${userId.value}`)
);
</script>

<template>
  <div v-if="loading">Loading...</div>
  <div v-else-if="error">Error: {{ error.message }}</div>
  <div v-else>{{ user.name }}</div>
</template>
```

Vue Compiler Optimizations:

Vue compiler analyzes templates and generates optimized render functions.

```
// Template
<template>
  <div>
    <p>Static text</p>
    <p>{{ dynamicText }}</p>
    <button @click="handleClick">{{ buttonText }}</button>
  </div>
</template>

// Compiled output (simplified)
function render(_ctx) {
  return (_openBlock(), _createBlock("div", null, [
    _hoisted_1, // Static node hoisted outside render function
    _createVNode("p", null, _toDisplayString(_ctx.dynamicText), 1 /* TEXT */),
    _createVNode("button", { onClick: _ctx.handleClick },
      _toDisplayString(_ctx.buttonText),
      9 /* TEXT, PROPS */,
      ["onClick"]
    )
  ]))
}
```

```
// Optimization flags:
// TEXT: only text children (skip full diff)
// CLASS: only class binding (skip other attrs)
// STYLE: only style binding
// PROPS: dynamic props (track which ones)
// FULL_PROPS: all props dynamic
// HYDRATE_EVENTS: events need hydration
// STABLE_FRAGMENT: fragment with stable children
// KEYED_FRAGMENT: fragment with keyed children
// UNKEYED_FRAGMENT: fragment without keys
// NEED_PATCH: needs patching
// DYNAMIC_SLOTS: component with dynamic slots
// DEV_ROOT_FRAGMENT: dev only
```

Block Tree optimization:

```
// Traditional VNode tree requires full traversal
// Block tree only tracks dynamic nodes

<template>
  <div>
    <header>
      <h1>{{ title }}</h1>  {/* Dynamic */}
      <nav>...</nav>        {/* Static */}
    </header>
    <main>
      <p>Static content</p>
      <p>{{ content }}</p>  {/* Dynamic */}
    </main>
  </div>
</template>

// Compiled with Block tree
const _hoisted_1 = _createVNode("nav", ...)  // Hoisted static node

function render(_ctx) {
  return (_openBlock(), _createBlock("div", null, [
    _createVNode("header", null, [
      _createVNode("h1", null, _ctx.title),  // Tracked
      _hoisted_1
    ]),
    _createVNode("main", null, [
      _hoisted_2,  // Static <p>
      _createVNode("p", null, _ctx.content)  // Tracked
    ])
  ], 64 /* STABLE_FRAGMENT */))
}
```

```
// Only h1 and content p are tracked for updates
// Static nodes completely skipped during diff
```

Angular - Dependency Injection and RxJS

Angular's architecture centers on dependency injection and reactive programming.

Dependency Injection Deep Dive:

```
// Injectable service
@Injectable({
  providedIn: 'root'  // Singleton, available app-wide
})
export class UserService {
  private users: User[] = [];

  getUsers(): Observable<User[]> {
    return this.http.get<User[]>('/api/users');
  }

  getUserById(id: number): Observable<User> {
    return this.http.get<User>(`/api/users/${id}`);
  }
}

// Component with DI
@Component({
  selector: 'app-user-list',
  template: `
    <ul>
      <li *ngFor="let user of users$ | async">
        {{ user.name }}
      </li>
    </ul>
  `
})
export class UserListComponent implements OnInit {
  users$: Observable<User[]>;

  // Dependency injected via constructor
  constructor(private userService: UserService) {}

  ngOnInit() {
    this.users$ = this.userService.getUsers();
  }
}

// Hierarchical injectors
```

```
@Injectable({
  providedIn: 'root'  // App-level singleton
})

@Injectable({
  providedIn: 'platform'  // Platform-level (rare)
})

@Injectable({
  providedIn: 'any'  // New instance per lazy-loaded module
})

// Component-level provider
@Component({
  providers: [SpecialService]  // New instance for this component tree
})

// Module-level provider
@NgModule({
  providers: [ModuleService]  // Shared within module
})
```

Injection Tokens for non-class dependencies:

```
// Configuration object
export interface AppConfig {
  apiUrl: string;
  timeout: number;
}

// Create injection token
export const APP_CONFIG = new InjectionToken<AppConfig>('app.config');

// Provide value
@NgModule({
  providers: [
    {
      provide: APP_CONFIG,
      useValue: {
        apiUrl: 'https://api.example.com',
        timeout: 5000
      }
    }
  ]
})

// Inject
@Injectable()
```

```
export class ApiService {
  constructor(@Inject(APP_CONFIG) private config: AppConfig) {
    console.log(this.config.apiUrl);
  }
}

// Factory provider
export function createLogger(config: AppConfig): Logger {
  return config.environment === 'prod'
    ? new ProductionLogger()
    : new DevLogger();
}

@NgModule({
  providers: [
    {
      provide: Logger,
      useFactory: createLogger,
      deps: [APP_CONFIG]  // Dependencies for factory
    }
  ]
})
```

RxJS Integration:

```
// HTTP with RxJS
@Injectable()
export class DataService {
  constructor(private http: HttpClient) {}

  // Basic request
  getData(): Observable<Data> {
    return this.http.get<Data>('/api/data');
  }

  // Transform response
  getUsers(): Observable<User[]> {
    return this.http.get<ApiResponse>('/api/users').pipe(
      map(response => response.data),
      map(users => users.map(u => ({
        ...u,
        fullName: `${u.firstName} ${u.lastName}`
      })))
    );
  }

  // Error handling
  getUserWithRetry(id: number): Observable<User> {
```

```
    return this.http.get<User>(`/api/users/${id}`).pipe(
      retry(3),
      catchError(error => {
        console.error('Failed to fetch user', error);
        return of(null);  // Return default value
      })
    );
  }

  // Combine multiple requests
  getUserWithPosts(id: number): Observable<UserWithPosts> {
    return forkJoin({
      user: this.http.get<User>(`/api/users/${id}`),
      posts: this.http.get<Post[]>(`/api/users/${id}/posts`)
    });
  }

  // Sequential dependent requests
  getUserAndComments(id: number): Observable<UserWithComments> {
    return this.http.get<User>(`/api/users/${id}`).pipe(
      switchMap(user =>
        this.http.get<Comment[]>(`/api/users/${id}/comments`).pipe(
          map(comments => ({ user, comments }))
        )
      )
    );
  }
}

// Component with subscriptions
@Component({
  selector: 'app-user-profile',
  template: `
    <div *ngIf="user$ | async as user">
      <h1>{{ user.name }}</h1>
      <p>{{ user.email }}</p>
    </div>
  `
})
export class UserProfileComponent implements OnInit, OnDestroy {
  user$: Observable<User>;
  private destroy$ = new Subject<void>();

  constructor(
    private route: ActivatedRoute,
    private userService: UserService
  ) {}
```

```
  ngOnInit() {
    // React to route parameter changes
    this.user$ = this.route.params.pipe(
      map(params => params['id']),
      switchMap(id => this.userService.getUserById(id)),
      takeUntil(this.destroy$)  // Auto-unsubscribe
    );
  }

  ngOnDestroy() {
    this.destroy$.next();
    this.destroy$.complete();
  }
}
```

RxJS Operators Deep Dive:

```
// Transformation operators
// map: transform each value
of(1, 2, 3).pipe(
  map(x => x * 2)
).subscribe(console.log);  // 2, 4, 6

// switchMap: cancel previous, switch to new observable
searchInput$.pipe(
  debounceTime(300),
  switchMap(query => this.search(query))
  // Cancels previous search if new query arrives
)

// mergeMap: don't cancel, merge all
clicks$.pipe(
  mergeMap(() => this.http.get('/api/data'))
  // All requests complete, results merged
)

// concatMap: queue, maintain order
actions$.pipe(
  concatMap(action => this.process(action))
  // Processes sequentially, waits for each to complete
)

// exhaustMap: ignore new while active
saveButton$.pipe(
  exhaustMap(() => this.save())
  // Ignores clicks while save in progress
)
```

```
// Filtering operators
// filter: conditionally pass values
numbers$.pipe(
  filter(x => x % 2 === 0)
)

// distinct: unique values
of(1, 2, 1, 3, 2, 4).pipe(
  distinct()
).subscribe(console.log);  // 1, 2, 3, 4

// distinctUntilChanged: only when changed
of(1, 1, 2, 2, 3).pipe(
  distinctUntilChanged()
).subscribe(console.log);  // 1, 2, 3

// Combination operators
// combineLatest: emit when any source emits
combineLatest([user$, settings$, preferences$]).pipe(
  map(([user, settings, prefs]) => ({
    user,
    settings,
    prefs
  }))
)

// forkJoin: wait for all to complete
forkJoin({
  users: this.getUsers(),
  posts: this.getPosts(),
  comments: this.getComments()
}).subscribe(({ users, posts, comments }) => {
  // All completed
});

// merge: combine multiple observables
merge(clicks$, keypress$, scroll$).subscribe(event => {
  // Any event triggers
});

// Rate limiting operators
// debounceTime: wait for quiet period
searchInput$.pipe(
  debounceTime(300)  // Wait 300ms after last input
)

// throttleTime: limit frequency
```

```
scroll$.pipe(
  throttleTime(100)  // At most once per 100ms
)

// auditTime: emit last value from time window
clicks$.pipe(
  auditTime(1000)  // Last click of each second
)

// Error handling
// catchError: handle and recover
this.http.get('/api/data').pipe(
  catchError(error => {
    console.error(error);
    return of(defaultValue);  // Continue with default
  })
)

// retry: retry on error
this.http.get('/api/data').pipe(
  retry(3),  // Retry up to 3 times
  catchError(error => throwError(error))
)

// retryWhen: custom retry logic
this.http.get('/api/data').pipe(
  retryWhen(errors =>
    errors.pipe(
      delay(1000),  // Wait 1s between retries
      take(3),      // Max 3 retries
      concat(throwError('Max retries exceeded'))
    )
  )
)
```

Change Detection Strategies:

```
// Default change detection - checks entire tree
@Component({
  changeDetection: ChangeDetectionStrategy.Default
})
export class DefaultComponent {
  // Checked on:
  // - Any event in the component tree
  // - HTTP response
  // - Timer (setTimeout, setInterval)
  // - Any async operation
}
```

```typescript
// OnPush - only checks when:
// 1. Input reference changes
// 2. Event in component
// 3. Manual markForCheck()
@Component({
  selector: 'app-efficient',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <div>{{ user.name }}</div>
    <button (click)="update()">Update</button>
  `
})
export class EfficientComponent {
  @Input() user: User;  // Only checks when reference changes

  constructor(private cdr: ChangeDetectorRef) {}

  // Component event triggers check
  update() {
    // Modify user
  }

  // Async data needs manual trigger
  ngOnInit() {
    this.dataService.getData().subscribe(data => {
      this.data = data;
      this.cdr.markForCheck();  // Tell Angular to check
    });
  }
}

// Best practices for OnPush:
// 1. Use immutable data
@Input() user: User;

updateUser() {
  // DON'T: mutation doesn't trigger change detection
  this.user.name = 'New Name';

  // DO: new reference triggers change detection
  this.user = { ...this.user, name: 'New Name' };
}

// 2. Use async pipe (handles markForCheck automatically)
user$: Observable<User>;

ngOnInit() {
```

```
  this.user$ = this.userService.getUser();
}

// Template
<div>{{ (user$ | async)?.name }}</div>

// 3. Detach for manual control
constructor(private cdr: ChangeDetectorRef) {
  this.cdr.detach();  // Disable automatic change detection
}

updateData() {
  this.data = newData;
  this.cdr.detectChanges();  // Manually trigger check
}
```

Redux and State Management Patterns

Redux implements unidirectional data flow with pure functions.

Redux Core Concepts:

```
// Store: single source of truth
const initialState = {
  users: [],
  loading: false,
  error: null
};

// Reducer: pure function (state, action) => newState
function usersReducer(state = initialState, action) {
  switch (action.type) {
    case 'FETCH_USERS_REQUEST':
      return {
        ...state,
        loading: true,
        error: null
      };

    case 'FETCH_USERS_SUCCESS':
      return {
        ...state,
        loading: false,
        users: action.payload
      };

    case 'FETCH_USERS_FAILURE':
      return {
        ...state,
```

```
        loading: false,
        error: action.payload
      };

    case 'ADD_USER':
      return {
        ...state,
        users: [...state.users, action.payload]
      };

    case 'UPDATE_USER':
      return {
        ...state,
        users: state.users.map(user =>
          user.id === action.payload.id
            ? { ...user, ...action.payload }
            : user
        )
      };

    case 'DELETE_USER':
      return {
        ...state,
        users: state.users.filter(user => user.id !== action.payload)
      };

    default:
      return state;
  }
}

// Action creators
function fetchUsersRequest() {
  return { type: 'FETCH_USERS_REQUEST' };
}

function fetchUsersSuccess(users) {
  return { type: 'FETCH_USERS_SUCCESS', payload: users };
}

function fetchUsersFailure(error) {
  return { type: 'FETCH_USERS_FAILURE', payload: error };
}

// Async action with redux-thunk middleware
function fetchUsers() {
  return async (dispatch) => {
```

```
    dispatch(fetchUsersRequest());

    try {
      const response = await fetch('/api/users');
      const users = await response.json();
      dispatch(fetchUsersSuccess(users));
    } catch (error) {
      dispatch(fetchUsersFailure(error.message));
    }
  };
}

// Create store
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';

const store = createStore(
  usersReducer,
  applyMiddleware(thunk)
);

// Dispatch actions
store.dispatch(fetchUsers());

// Subscribe to changes
store.subscribe(() => {
  console.log('State:', store.getState());
});
```

Redux Toolkit (modern approach):

```
import { createSlice, createAsyncThunk, configureStore } from '@reduxjs/toolkit';

// Async thunk
export const fetchUsers = createAsyncThunk(
  'users/fetchUsers',
  async (_, { rejectWithValue }) => {
    try {
      const response = await fetch('/api/users');
      return await response.json();
    } catch (error) {
      return rejectWithValue(error.message);
    }
  }
);

// Slice (combines actions, reducers)
const usersSlice = createSlice({
```

```
  name: 'users',
  initialState: {
    items: [],
    loading: false,
    error: null
  },
  reducers: {
    // Synchronous actions
    addUser: (state, action) => {
      // Immer allows "mutations"
      state.items.push(action.payload);
    },
    updateUser: (state, action) => {
      const user = state.items.find(u => u.id === action.payload.id);
      if (user) {
        Object.assign(user, action.payload);
      }
    },
    deleteUser: (state, action) => {
      state.items = state.items.filter(u => u.id !== action.payload);
    }
  },
  extraReducers: (builder) => {
    // Async thunk states
    builder
      .addCase(fetchUsers.pending, (state) => {
        state.loading = true;
        state.error = null;
      })
      .addCase(fetchUsers.fulfilled, (state, action) => {
        state.loading = false;
        state.items = action.payload;
      })
      .addCase(fetchUsers.rejected, (state, action) => {
        state.loading = false;
        state.error = action.payload;
      });
  }
});

export const { addUser, updateUser, deleteUser } = usersSlice.actions;
export default usersSlice.reducer;

// Configure store
const store = configureStore({
  reducer: {
    users: usersSlice.reducer,
```

```
    posts: postsSlice.reducer
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(logger)
});

// React integration
import { Provider, useSelector, useDispatch } from 'react-redux';

function App() {
  return (
    <Provider store={store}>
      <UserList />
    </Provider>
  );
}

function UserList() {
  const dispatch = useDispatch();
  const { items: users, loading, error } = useSelector(state => state.users);

  useEffect(() => {
    dispatch(fetchUsers());
  }, [dispatch]);

  if (loading) return <Spinner />;
  if (error) return <Error message={error} />;

  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>
          {user.name}
          <button onClick={() => dispatch(deleteUser(user.id))}>
            Delete
          </button>
        </li>
      ))}
    </ul>
  );
}
```

Selectors and Memoization:

```
import { createSelector } from '@reduxjs/toolkit';

// Basic selector
const selectUsers = state => state.users.items;
```

```
const selectFilter = state => state.filters.search;

// Memoized selector
const selectFilteredUsers = createSelector(
  [selectUsers, selectFilter],
  (users, filter) => {
    console.log('Computing filtered users');  // Only logs when inputs change
    return users.filter(user =>
      user.name.toLowerCase().includes(filter.toLowerCase())
    );
  }
);

// Usage
function UserList() {
  // Only re-computes when users or filter changes
  const filteredUsers = useSelector(selectFilteredUsers);

  return (
    <ul>
      {filteredUsers.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}

// Selector with parameters
const makeSelectUserById = () => createSelector(
  [selectUsers, (state, userId) => userId],
  (users, userId) => users.find(user => user.id === userId)
);

// Usage with parameters
function UserProfile({ userId }) {
  const selectUserById = useMemo(makeSelectUserById, []);
  const user = useSelector(state => selectUserById(state, userId));

  return <div>{user?.name}</div>;
}
```

Redux Middleware:

```
// Middleware signature: store => next => action => result

// Logger middleware
const logger = store => next => action => {
  console.log('Dispatching:', action);
```

```
  const result = next(action);
  console.log('Next state:', store.getState());
  return result;
};

// Analytics middleware
const analytics = store => next => action => {
  // Track user actions
  if (action.type.startsWith('user/')) {
    trackEvent(action.type, action.payload);
  }
  return next(action);
};

// API middleware
const api = store => next => action => {
  if (!action.meta || !action.meta.api) {
    return next(action);
  }

  const { url, method, data } = action.meta.api;
  const { type } = action;

  next({ type: `${type}_REQUEST` });

  return fetch(url, { method, body: JSON.stringify(data) })
    .then(res => res.json())
    .then(data => next({ type: `${type}_SUCCESS`, payload: data }))
    .catch(error => next({ type: `${type}_FAILURE`, payload: error }));
};

// Apply middleware
const store = configureStore({
  reducer: rootReducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(logger, analytics, api)
});
```

Redux DevTools Integration:

```
import { configureStore } from '@reduxjs/toolkit';

const store = configureStore({
  reducer: rootReducer,
  devTools: process.env.NODE_ENV !== 'production'
});

// DevTools features:
```

```
// - Action history (time travel debugging)
// - State snapshots
// - Action replay
// - Performance monitoring
// - Export/import state
```

Chapter 3

Chapter 3: Design Patterns (Functional, Reactive, Declarative)

3.1 Functional Programming Patterns

3.1.1 Pure Functions and Immutability

```
// Impure: modifies input, depends on external state
let count = 0;

function impureIncrement(arr) {
  count++;  // Side effect: modifies external state
  arr.push(count);  // Side effect: modifies input
  return arr;
}

// Pure: no side effects, same input => same output
function pureIncrement(arr, count) {
  return [...arr, count + 1];  // Returns new array
}

// Immutable operations
const arr = [1, 2, 3];

// Array operations
const newArr = [
  ...arr,  // Spread: copy array
  4        // Add element
];

const withoutFirst = arr.slice(1);  // Remove first
```

```javascript
const withoutLast = arr.slice(0, -1);  // Remove last
const replaced = arr.map((x, i) => i === 1 ? 99 : x);  // Replace element

// Object operations
const obj = { a: 1, b: 2 };

const updated = {
  ...obj,
  b: 99,  // Override property
  c: 3    // Add property
};

const { b, ...rest } = obj;  // Remove property (rest = { a: 1 })

// Nested immutable update
const state = {
  user: {
    profile: {
      name: 'John',
      age: 30
    },
    settings: {
      theme: 'dark'
    }
  }
};

// Update nested property immutably
const newState = {
  ...state,
  user: {
    ...state.user,
    profile: {
      ...state.user.profile,
      age: 31
    }
  }
};

// Immer library simplifies nested updates
import produce from 'immer';

const newState2 = produce(state, draft => {
  draft.user.profile.age = 31;  // "Mutate" draft safely
});
```

Higher-Order Functions:

```javascript
// Function that returns function
function multiplier(factor) {
  return function(number) {
    return number * factor;
  };
}

const double = multiplier(2);
const triple = multiplier(3);

console.log(double(5));  // 10
console.log(triple(5));  // 15

// Function that takes function as argument
function withLogging(fn) {
  return function(...args) {
    console.log(`Calling with args:`, args);
    const result = fn(...args);
    console.log(`Result:`, result);
    return result;
  };
}

const add = (a, b) => a + b;
const loggedAdd = withLogging(add);

loggedAdd(2, 3);
// Logs: Calling with args: [2, 3]
// Logs: Result: 5

// Practical: Array transformation pipeline
const users = [
  { name: 'John', age: 30, active: true },
  { name: 'Jane', age: 25, active: false },
  { name: 'Bob', age: 35, active: true }
];

const activeUserNames = users
  .filter(user => user.active)
  .map(user => user.name)
  .sort();

// Custom higher-order functions
function filter(predicate) {
  return function(array) {
    return array.filter(predicate);
  };
```

```
}

function map(mapper) {
  return function(array) {
    return array.map(mapper);
  };
}

const filterActive = filter(user => user.active);
const mapToNames = map(user => user.name);

const result = mapToNames(filterActive(users));
```

Currying and Partial Application:

```
// Currying: transform f(a, b, c) => f(a)(b)(c)
function curry(fn) {
  return function curried(...args) {
    if (args.length >= fn.length) {
      return fn.apply(this, args);
    } else {
      return function(...nextArgs) {
        return curried.apply(this, args.concat(nextArgs));
      };
    }
  };
}

// Usage
function add(a, b, c) {
  return a + b + c;
}

const curriedAdd = curry(add);

console.log(curriedAdd(1)(2)(3));  // 6
console.log(curriedAdd(1, 2)(3));  // 6
console.log(curriedAdd(1)(2, 3));  // 6

// Partial application
function partial(fn, ...fixedArgs) {
  return function(...remainingArgs) {
    return fn(...fixedArgs, ...remainingArgs);
  };
}

function greet(greeting, name) {
  return `${greeting}, ${name}!`;
```

```
}

const sayHello = partial(greet, 'Hello');
const sayGoodbye = partial(greet, 'Goodbye');

console.log(sayHello('John'));  // Hello, John!
console.log(sayGoodbye('Jane'));  // Goodbye, Jane!

// Real-world: Event handling
function handleEvent(eventType, selector, handler, event) {
  if (event.target.matches(selector)) {
    handler(event);
  }
}

const handleClick = partial(handleEvent, 'click');
const handleButtonClick = partial(handleClick, 'button');

document.addEventListener('click', handleButtonClick((e) => {
  console.log('Button clicked:', e.target);
}));
```

Function Composition:

```
// compose: right-to-left
function compose(...fns) {
  return function(value) {
    return fns.reduceRight((acc, fn) => fn(acc), value);
  };
}

// pipe: left-to-right (more intuitive)
function pipe(...fns) {
  return function(value) {
    return fns.reduce((acc, fn) => fn(acc), value);
  };
}

// Example functions
const addOne = x => x + 1;
const double = x => x * 2;
const square = x => x * x;

const composedFn = compose(square, double, addOne);
console.log(composedFn(3));  // square(double(addOne(3))) = square(double(4)) = square(

const pipedFn = pipe(addOne, double, square);
console.log(pipedFn(3));  // square(double(addOne(3))) = square(double(4)) = square(8)
```

```javascript
// Real-world: Data transformation pipeline
const users = [/*...*/];

const processUsers = pipe(
  users => users.filter(u => u.active),
  users => users.map(u => ({ ...u, fullName: `${u.first} ${u.last}` })),
  users => users.sort((a, b) => a.fullName.localeCompare(b.fullName)),
  users => users.slice(0, 10)
);

const topActiveUsers = processUsers(users);

// Async composition
const composeAsync = (...fns) => {
  return async function(value) {
    let result = value;
    for (const fn of fns.reverse()) {
      result = await fn(result);
    }
    return result;
  };
};

const pipeAsync = (...fns) => {
  return async function(value) {
    let result = value;
    for (const fn of fns) {
      result = await fn(result);
    }
    return result;
  };
};

// Usage
const fetchUser = id => fetch(`/api/users/${id}`).then(r => r.json());
const fetchPosts = user => fetch(`/api/users/${user.id}/posts`).then(r => r.json()).the
const formatUser = user => ({ ...user, displayName: user.name.toUpperCase() });

const getUserWithPosts = pipeAsync(
  fetchUser,
  fetchPosts,
  formatUser
);

getUserWithPosts(123).then(console.log);
```

Functors and Monads:

```javascript
// Functor: type with map method
// Array is a functor
[1, 2, 3].map(x => x * 2);  // [2, 4, 6]

// Custom functor: Box
class Box {
  constructor(value) {
    this._value = value;
  }

  map(fn) {
    return new Box(fn(this._value));
  }

  valueOf() {
    return this._value;
  }
}

const result = new Box(5)
  .map(x => x * 2)
  .map(x => x + 1)
  .valueOf();  // 11

// Maybe monad: handles null/undefined
class Maybe {
  constructor(value) {
    this._value = value;
  }

  static of(value) {
    return new Maybe(value);
  }

  isNothing() {
    return this._value === null || this._value === undefined;
  }

  map(fn) {
    return this.isNothing() ? this : Maybe.of(fn(this._value));
  }

  flatMap(fn) {
    return this.isNothing() ? this : fn(this._value);
  }

  getOrElse(defaultValue) {
```

```
    return this.isNothing() ? defaultValue : this._value;
  }
}

// Usage
function getUserById(id) {
  const user = users.find(u => u.id === id);
  return Maybe.of(user);
}

const userName = getUserById(123)
  .map(user => user.profile)
  .map(profile => profile.name)
  .map(name => name.toUpperCase())
  .getOrElse('Unknown');

// No null checks needed! Handles missing values gracefully

// Either monad: handles errors
class Either {
  constructor(value) {
    this._value = value;
  }

  static left(value) {
    const either = new Either(value);
    either._isLeft = true;
    return either;
  }

  static right(value) {
    const either = new Either(value);
    either._isLeft = false;
    return either;
  }

  isLeft() {
    return this._isLeft;
  }

  isRight() {
    return !this._isLeft;
  }

  map(fn) {
    return this.isLeft() ? this : Either.right(fn(this._value));
  }
```

```
  flatMap(fn) {
    return this.isLeft() ? this : fn(this._value);
  }

  fold(leftFn, rightFn) {
    return this.isLeft() ? leftFn(this._value) : rightFn(this._value);
  }
}

// Usage
function parseJSON(str) {
  try {
    return Either.right(JSON.parse(str));
  } catch (error) {
    return Either.left(error.message);
  }
}

function validateUser(user) {
  if (!user.name) {
    return Either.left('Name is required');
  }
  if (!user.email) {
    return Either.left('Email is required');
  }
  return Either.right(user);
}

const result = parseJSON('{"name":"John","email":"john@example.com"}')
  .flatMap(validateUser)
  .map(user => ({ ...user, validated: true }))
  .fold(
    error => ({ error }),
    user => ({ user })
  );

// Promise is a monad
Promise.resolve(5)
  .then(x => x * 2)  // map
  .then(x => Promise.resolve(x + 1))  // flatMap
  .then(console.log);  // 11
```

Reactive Programming Patterns

Observable Streams:

```
// Basic Observable implementation
class Observable {
```

```
  constructor(subscribe) {
    this._subscribe = subscribe;
  }

  subscribe(observer) {
    return this._subscribe(observer);
  }

  static create(subscribe) {
    return new Observable(subscribe);
  }

  map(fn) {
    return Observable.create(observer => {
      return this.subscribe({
        next: value => observer.next(fn(value)),
        error: err => observer.error(err),
        complete: () => observer.complete()
      });
    });
  }

  filter(predicate) {
    return Observable.create(observer => {
      return this.subscribe({
        next: value => {
          if (predicate(value)) {
            observer.next(value);
          }
        },
        error: err => observer.error(err),
        complete: () => observer.complete()
      });
    });
  }
}

// Create observable
const numbers$ = Observable.create(observer => {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();

  // Return cleanup function
  return () => console.log('Unsubscribed');
});
```

```
// Subscribe
const subscription = numbers$
  .map(x => x * 2)
  .filter(x => x > 2)
  .subscribe({
    next: value => console.log(value),  // 4, 6
    error: err => console.error(err),
    complete: () => console.log('Done')
  });

// Unsubscribe
subscription();  // Logs: Unsubscribed

// RxJS operators
import { fromEvent, interval } from 'rxjs';
import { map, filter, debounceTime, distinctUntilChanged, switchMap } from 'rxjs/operat

// Example: Search input
const searchInput = document.getElementById('search');

const search$ = fromEvent(searchInput, 'input').pipe(
  map(event => event.target.value),
  debounceTime(300),  // Wait 300ms after last input
  distinctUntilChanged(),  // Only if value changed
  filter(term => term.length > 2),  // At least 3 characters
  switchMap(term => fetch(`/api/search?q=${term}`).then(r => r.json()))
);

search$.subscribe(results => {
  displayResults(results);
});

// Example: Auto-save
const form$ = fromEvent(formElement, 'input').pipe(
  debounceTime(1000),  // Wait 1s after last edit
  map(() => getFormData()),
  distinctUntilChanged((a, b) => JSON.stringify(a) === JSON.stringify(b)),
  switchMap(data => saveData(data))
);

form$.subscribe(
  () => showSaveIndicator('Saved'),
  error => showSaveIndicator('Error: ' + error)
);
```

Subject Types:

```
import { Subject, BehaviorSubject, ReplaySubject, AsyncSubject } from 'rxjs';

// Subject: no initial value, no replay
const subject = new Subject();

subject.subscribe(v => console.log('A:', v));
subject.next(1);  // A: 1
subject.next(2);  // A: 2

subject.subscribe(v => console.log('B:', v));  // Doesn't receive 1 or 2
subject.next(3);  // A: 3, B: 3

// BehaviorSubject: has initial value, replays last value
const behavior = new BehaviorSubject(0);

behavior.subscribe(v => console.log('A:', v));  // A: 0 (immediate)
behavior.next(1);  // A: 1
behavior.next(2);  // A: 2

behavior.subscribe(v => console.log('B:', v));  // B: 2 (gets last value)
behavior.next(3);  // A: 3, B: 3

console.log(behavior.getValue());  // 3 (synchronous access)

// ReplaySubject: replays N last values
const replay = new ReplaySubject(2);  // Remember last 2 values

replay.next(1);
replay.next(2);
replay.next(3);

replay.subscribe(v => console.log('A:', v));  // A: 2, A: 3 (last 2)
replay.next(4);  // A: 4

// AsyncSubject: emits only last value on complete
const async = new AsyncSubject();

async.subscribe(v => console.log('A:', v));

async.next(1);
async.next(2);
async.next(3);
// Nothing logged yet

async.complete();  // A: 3 (only last value, only on complete)
```

Backpressure Strategies:

```
import { interval } from 'rxjs';
import {
  buffer,
  bufferTime,
  throttle,
  debounce,
  sample,
  audit
} from 'rxjs/operators';

// Problem: Fast producer, slow consumer
const fast$ = interval(10);  // Emits every 10ms

fast$.subscribe(v => {
  // Slow processing (100ms)
  processSlowly(v);  // Can't keep up!
});

// Solution 1: Buffer
fast$.pipe(
  bufferTime(1000)  // Collect values for 1 second
).subscribe(values => {
  processBatch(values);  // Process batch once per second
});

// Solution 2: Throttle (emit, then ignore for duration)
fast$.pipe(
  throttleTime(1000)  // Emit first, ignore rest for 1s
).subscribe(v => {
  process(v);  // Processes at most once per second
});

// Solution 3: Debounce (wait for quiet period)
const userInput$ = fromEvent(input, 'input');

userInput$.pipe(
  debounceTime(500)  // Wait 500ms after last input
).subscribe(v => {
  search(v);  // Only search after user stops typing
});

// Solution 4: Sample (emit latest at intervals)
fast$.pipe(
  sampleTime(1000)  // Emit latest value every 1s
).subscribe(v => {
  update(v);  // Uses most recent value
});
```

```
// Solution 5: Audit (emit last value from time window)
fast$.pipe(
  auditTime(1000)  // Emit last value after 1s window
).subscribe(v => {
  process(v);
});
```

Declarative Programming Patterns

State Machines:

```
// Finite State Machine for form submission
class FormStateMachine {
  constructor() {
    this.state = 'idle';
    this.data = null;
    this.error = null;
  }

  transition(action, payload) {
    const transitions = {
      idle: {
        SUBMIT: () => {
          this.state = 'submitting';
          this.data = payload;
          this.error = null;
        }
      },
      submitting: {
        SUCCESS: () => {
          this.state = 'success';
          this.data = payload;
        },
        FAILURE: () => {
          this.state = 'error';
          this.error = payload;
        }
      },
      success: {
        RESET: () => {
          this.state = 'idle';
          this.data = null;
          this.error = null;
        }
      },
      error: {
        RETRY: () => {
          this.state = 'submitting';
```

```
          this.error = null;
        },
        RESET: () => {
          this.state = 'idle';
          this.data = null;
          this.error = null;
        }
      }
    };

    const stateTransitions = transitions[this.state];
    const transition = stateTransitions?.[action];

    if (transition) {
      transition();
    } else {
      console.warn(`Invalid transition: ${action} from ${this.state}`);
    }
  }

  canSubmit() {
    return this.state === 'idle';
  }

  canRetry() {
    return this.state === 'error';
  }
}

// Usage
const form = new FormStateMachine();

form.transition('SUBMIT', formData);  // idle -> submitting

// After async operation
if (success) {
  form.transition('SUCCESS', response);  // submitting -> success
} else {
  form.transition('FAILURE', error);  // submitting -> error
}

// XState library for complex state machines
import { createMachine, interpret } from 'xstate';

const trafficLightMachine = createMachine({
  id: 'trafficLight',
  initial: 'green',
```

```
  states: {
    green: {
      after: {
        3000: 'yellow'  // After 3s, transition to yellow
      }
    },
    yellow: {
      after: {
        1000: 'red'
      }
    },
    red: {
      after: {
        4000: 'green'
      }
    }
  }
});

const service = interpret(trafficLightMachine)
  .onTransition(state => {
    console.log('Current state:', state.value);
  })
  .start();

// Manual transitions
service.send('NEXT');

// Stop
service.stop();
```

Reactive Declarations:

```
// Svelte's reactive declarations
/*
<script>
  let count = 0;

  // Reactive statement: automatically re-runs when count changes
  $: doubled = count * 2;

  // Reactive block
  $: {
    console.log(`Count is ${count}`);
    if (count > 10) {
      alert('Count is over 10!');
    }
  }
```

```
  // Reactive array filtering
  $: evenNumbers = numbers.filter(n => n % 2 === 0);
</script>

<button on:click={() => count++}>
  {count} (doubled: {doubled})
</button>
*/

// Vue computed properties
/*
<script setup>
import { ref, computed } from 'vue';

const firstName = ref('John');
const lastName = ref('Doe');

// Automatically updates when dependencies change
const fullName = computed(() => {
  return `${firstName.value} ${lastName.value}`;
});
</script>

<template>
  <div>{{ fullName }}</div>
</template>
*/

// MobX reactive state
/*
import { observable, computed, autorun, action } from 'mobx';

class TodoStore {
  @observable todos = [];

  @computed get completedCount() {
    return this.todos.filter(t => t.completed).length;
  }

  @action addTodo(title) {
    this.todos.push({
      id: Date.now(),
      title,
      completed: false
    });
  }
```

```
  @action toggleTodo(id) {
    const todo = this.todos.find(t => t.id === id);
    if (todo) {
      todo.completed = !todo.completed;
    }
  }
}

const store = new TodoStore();

// Automatically re-runs when dependencies change
autorun(() => {
  console.log(`Completed: ${store.completedCount}/${store.todos.length}`);
});

store.addTodo('Learn MobX');  // Logs: Completed: 0/1
store.toggleTodo(store.todos[0].id);  // Logs: Completed: 1/1
*/
```

Component Design Patterns:

```
// Container/Presentational Pattern
// Container: handles logic and state
function UserListContainer() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetchUsers()
      .then(setUsers)
      .finally(() => setLoading(false));
  }, []);

  const handleDelete = (id) => {
    deleteUser(id).then(() => {
      setUsers(users.filter(u => u.id !== id));
    });
  };

  return (
    <UserListPresentation
      users={users}
      loading={loading}
      onDelete={handleDelete}
    />
  );
}
```

```
// Presentational: pure rendering
function UserListPresentation({ users, loading, onDelete }) {
  if (loading) return <Spinner />;

  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>
          {user.name}
          <button onClick={() => onDelete(user.id)}>Delete</button>
        </li>
      ))}
    </ul>
  );
}

// Compound Components Pattern
const Tab = ({ children, active }) => (
  active ? <div>{children}</div> : null
);

const Tabs = ({ children }) => {
  const [activeIndex, setActiveIndex] = useState(0);

  return (
    <div>
      <div className="tab-buttons">
        {React.Children.map(children, (child, index) => (
          <button onClick={() => setActiveIndex(index)}>
            {child.props.label}
          </button>
        ))}
      </div>

      {React.Children.map(children, (child, index) =>
        React.cloneElement(child, { active: index === activeIndex })
      )}
    </div>
  );
};

// Usage
function App() {
  return (
    <Tabs>
      <Tab label="Tab 1">Content 1</Tab>
      <Tab label="Tab 2">Content 2</Tab>
```

```
      <Tab label="Tab 3">Content 3</Tab>
    </Tabs>
  );
}

// Render Props Pattern
class Mouse extends React.Component {
  state = { x: 0, y: 0 };

  handleMouseMove = (event) => {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  };

  render() {
    return (
      <div onMouseMove={this.handleMouseMove}>
        {this.props.render(this.state)}
      </div>
    );
  }
}

// Usage
<Mouse render={({ x, y }) => (
  <div>Mouse position: {x}, {y}</div>
)} />

// Higher-Order Component Pattern
function withAuth(Component) {
  return function AuthenticatedComponent(props) {
    const { user, loading } = useAuth();

    if (loading) return <Spinner />;
    if (!user) return <Redirect to="/login" />;

    return <Component {...props} user={user} />;
  };
}

// Usage
const ProtectedPage = withAuth(DashboardPage);

// Custom Hook Pattern
function useLocalStorage(key, initialValue) {
```

```
  const [storedValue, setStoredValue] = useState(() => {
    try {
      const item = window.localStorage.getItem(key);
      return item ? JSON.parse(item) : initialValue;
    } catch (error) {
      return initialValue;
    }
  });

  const setValue = (value) => {
    try {
      setStoredValue(value);
      window.localStorage.setItem(key, JSON.stringify(value));
    } catch (error) {
      console.error(error);
    }
  };

  return [storedValue, setValue];
}

// Usage
function Settings() {
  const [theme, setTheme] = useLocalStorage('theme', 'light');

  return (
    <button onClick={() => setTheme(theme === 'light' ? 'dark' : 'light')}>
      Toggle Theme
    </button>
  );
}
```

Chapter 4

Chapter 4: Advanced JavaScript - Async Programming, Generators, and Promises

Advanced asynchronous patterns are essential for modern JavaScript applications. Understanding how async mechanisms work internally enables writing performant, maintainable code.

4.1 Asynchronous Programming Deep Dive

4.1.1 Event Loop Architecture

The JavaScript event loop coordinates execution of code, collection of events, and processing of sub-tasks.

```
// Event loop phases
// 1. Timers (setTimeout, setInterval callbacks)
// 2. Pending callbacks (I/O callbacks deferred to next iteration)
// 3. Idle, prepare (internal use)
// 4. Poll (retrieve new I/O events, execute callbacks)
// 5. Check (setImmediate callbacks)
// 6. Close callbacks (socket.on('close'))

// Microtasks vs Macrotasks
console.log('1: Synchronous');

setTimeout(() => {
  console.log('2: setTimeout (macrotask)');
}, 0);

Promise.resolve().then(() => {
  console.log('3: Promise (microtask)');
});
```

```
console.log('4: Synchronous');

// Output: 1, 4, 3, 2
// Microtasks execute before next macrotask

// Detailed event loop
console.log('Script start');

setTimeout(() => console.log('setTimeout 1'), 0);

Promise.resolve()
  .then(() => console.log('Promise 1'))
  .then(() => console.log('Promise 2'));

setTimeout(() => console.log('setTimeout 2'), 0);

queueMicrotask(() => console.log('queueMicrotask'));

console.log('Script end');

// Output:
// Script start
// Script end
// Promise 1
// queueMicrotask
// Promise 2
// setTimeout 1
// setTimeout 2
```

Implementing Promises from Scratch:

```
const PENDING = 'PENDING';
const FULFILLED = 'FULFILLED';
const REJECTED = 'REJECTED';

class CustomPromise {
  constructor(executor) {
    this.state = PENDING;
    this.value = undefined;
    this.reason = undefined;
    this.onFulfilledCallbacks = [];
    this.onRejectedCallbacks = [];

    const resolve = (value) => {
      // Only transition from PENDING
      if (this.state !== PENDING) return;
```

```
    // Handle Promise resolution (thenable unwrapping)
    if (value instanceof CustomPromise) {
      value.then(resolve, reject);
      return;
    }

    this.state = FULFILLED;
    this.value = value;

    // Execute all registered callbacks asynchronously
    queueMicrotask(() => {
      this.onFulfilledCallbacks.forEach(callback => callback(this.value));
    });
  };

  const reject = (reason) => {
    if (this.state !== PENDING) return;

    this.state = REJECTED;
    this.reason = reason;

    queueMicrotask(() => {
      this.onRejectedCallbacks.forEach(callback => callback(this.reason));
    });
  };

  // Execute executor immediately
  try {
    executor(resolve, reject);
  } catch (error) {
    reject(error);
  }
}

then(onFulfilled, onRejected) {
  // Ensure callbacks are functions
  onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : value => value;
  onRejected = typeof onRejected === 'function' ? onRejected : reason => { throw reas

  // Return new Promise for chaining
  const promise2 = new CustomPromise((resolve, reject) => {
    if (this.state === FULFILLED) {
      queueMicrotask(() => {
        try {
          const x = onFulfilled(this.value);
          resolvePromise(promise2, x, resolve, reject);
        } catch (error) {
```

```
          reject(error);
        }
      });
    } else if (this.state === REJECTED) {
      queueMicrotask(() => {
        try {
          const x = onRejected(this.reason);
          resolvePromise(promise2, x, resolve, reject);
        } catch (error) {
          reject(error);
        }
      });
    } else {
      // PENDING: register callbacks
      this.onFulfilledCallbacks.push((value) => {
        try {
          const x = onFulfilled(value);
          resolvePromise(promise2, x, resolve, reject);
        } catch (error) {
          reject(error);
        }
      });

      this.onRejectedCallbacks.push((reason) => {
        try {
          const x = onRejected(reason);
          resolvePromise(promise2, x, resolve, reject);
        } catch (error) {
          reject(error);
        }
      });
    }
  });

  return promise2;
}

catch(onRejected) {
  return this.then(null, onRejected);
}

finally(callback) {
  return this.then(
    value => CustomPromise.resolve(callback()).then(() => value),
    reason => CustomPromise.resolve(callback()).then(() => { throw reason; })
  );
}
```

```
  static resolve(value) {
    if (value instanceof CustomPromise) {
      return value;
    }

    return new CustomPromise(resolve => resolve(value));
  }

  static reject(reason) {
    return new CustomPromise((_, reject) => reject(reason));
  }

  static all(promises) {
    return new CustomPromise((resolve, reject) => {
      if (!Array.isArray(promises)) {
        reject(new TypeError('Argument must be an array'));
        return;
      }

      if (promises.length === 0) {
        resolve([]);
        return;
      }

      const results = [];
      let completed = 0;

      promises.forEach((promise, index) => {
        CustomPromise.resolve(promise).then(
          value => {
            results[index] = value;
            completed++;

            if (completed === promises.length) {
              resolve(results);
            }
          },
          reject
        );
      });
    });
  }

  static race(promises) {
    return new CustomPromise((resolve, reject) => {
      if (!Array.isArray(promises)) {
        reject(new TypeError('Argument must be an array'));
```

```
      return;
    }

    promises.forEach(promise => {
      CustomPromise.resolve(promise).then(resolve, reject);
    });
  });
}

static allSettled(promises) {
  return new CustomPromise((resolve) => {
    if (!Array.isArray(promises)) {
      resolve([]);
      return;
    }

    if (promises.length === 0) {
      resolve([]);
      return;
    }

    const results = [];
    let completed = 0;

    promises.forEach((promise, index) => {
      CustomPromise.resolve(promise).then(
        value => {
          results[index] = { status: 'fulfilled', value };
          completed++;
          if (completed === promises.length) resolve(results);
        },
        reason => {
          results[index] = { status: 'rejected', reason };
          completed++;
          if (completed === promises.length) resolve(results);
        }
      );
    });
  });
}

static any(promises) {
  return new CustomPromise((resolve, reject) => {
    if (!Array.isArray(promises)) {
      reject(new TypeError('Argument must be an array'));
      return;
    }
```

```
      if (promises.length === 0) {
        reject(new AggregateError([], 'All promises were rejected'));
        return;
      }

      const errors = [];
      let rejected = 0;

      promises.forEach((promise, index) => {
        CustomPromise.resolve(promise).then(
          resolve,
          error => {
            errors[index] = error;
            rejected++;

            if (rejected === promises.length) {
              reject(new AggregateError(errors, 'All promises were rejected'));
            }
          }
        );
      });
    });
  }
}

// Promise resolution procedure
function resolvePromise(promise2, x, resolve, reject) {
  // Prevent circular reference
  if (promise2 === x) {
    reject(new TypeError('Chaining cycle detected'));
    return;
  }

  // If x is a Promise
  if (x instanceof CustomPromise) {
    x.then(resolve, reject);
    return;
  }

  // If x is an object or function (thenable)
  if (x !== null && (typeof x === 'object' || typeof x === 'function')) {
    let called = false;

    try {
      const then = x.then;

      if (typeof then === 'function') {
```

```
        then.call(
          x,
          (y) => {
            if (called) return;
            called = true;
            resolvePromise(promise2, y, resolve, reject);
          },
          (r) => {
            if (called) return;
            called = true;
            reject(r);
          }
        );
      } else {
        resolve(x);
      }
    } catch (error) {
      if (called) return;
      called = true;
      reject(error);
    }
  } else {
    // x is a primitive value
    resolve(x);
  }
}

// Testing
const p1 = new CustomPromise((resolve) => {
  setTimeout(() => resolve('Result 1'), 1000);
});

p1
  .then(result => {
    console.log(result);
    return 'Result 2';
  })
  .then(result => {
    console.log(result);
  });

// Chaining
CustomPromise.resolve(5)
  .then(x => x * 2)
  .then(x => x + 3)
  .then(console.log);  // 13
```

```
// Error handling
CustomPromise.reject('Error!')
  .catch(error => {
    console.log('Caught:', error);
    return 'Recovered';
  })
  .then(console.log);  // Recovered
```

Async/Await Implementation:

Async/await is syntactic sugar over Promises, implemented using generators internally:

```
// Before async/await
function fetchUserData(userId) {
  return fetch(`/api/users/${userId}`)
    .then(response => response.json())
    .then(user => {
      return fetch(`/api/users/${userId}/posts`);
    })
    .then(response => response.json())
    .then(posts => {
      return { user, posts };
    })
    .catch(error => {
      console.error('Error:', error);
      throw error;
    });
}

// With async/await
async function fetchUserData(userId) {
  try {
    const userResponse = await fetch(`/api/users/${userId}`);
    const user = await userResponse.json();

    const postsResponse = await fetch(`/api/users/${userId}/posts`);
    const posts = await postsResponse.json();

    return { user, posts };
  } catch (error) {
    console.error('Error:', error);
    throw error;
  }
}

// Async function always returns Promise
async function example() {
```

```
  return 42;
}

example().then(console.log);  // 42

// Equivalent to:
function example() {
  return Promise.resolve(42);
}

// Error handling
async function withError() {
  throw new Error('Oops');
}

// Equivalent to:
function withError() {
  return Promise.reject(new Error('Oops'));
}

// Parallel execution
async function sequential() {
  const user = await fetchUser();     // Wait
  const posts = await fetchPosts();   // Then wait
  return { user, posts };
}

async function parallel() {
  const [user, posts] = await Promise.all([
    fetchUser(),
    fetchPosts()
  ]);
  return { user, posts };
}

// Performance comparison
console.time('sequential');
await sequential();  // ~2000ms (1000ms + 1000ms)
console.timeEnd('sequential');

console.time('parallel');
await parallel();  // ~1000ms (max of both)
console.timeEnd('parallel');
```

Generators In-Depth:

Generators are functions that can pause and resume, enabling powerful async patterns:

```javascript
// Basic generator
function* numberGenerator() {
  console.log('Starting');
  yield 1;
  console.log('After first yield');
  yield 2;
  console.log('After second yield');
  yield 3;
  console.log('Done');
  return 4;
}

const gen = numberGenerator();

console.log(gen.next());  // { value: 1, done: false }
// Logs: Starting

console.log(gen.next());  // { value: 2, done: false }
// Logs: After first yield

console.log(gen.next());  // { value: 3, done: false }
// Logs: After second yield

console.log(gen.next());  // { value: 4, done: true }
// Logs: Done

console.log(gen.next());  // { value: undefined, done: true }

// Two-way communication
function* dataExchange() {
  console.log('Started');
  const x = yield 'First value';
  console.log('Received:', x);
  const y = yield 'Second value';
  console.log('Received:', y);
  return 'Done';
}

const gen2 = dataExchange();

console.log(gen2.next());           // { value: 'First value', done: false }
// Logs: Started

console.log(gen2.next('Input 1')); // { value: 'Second value', done: false }
// Logs: Received: Input 1

console.log(gen2.next('Input 2')); // { value: 'Done', done: true }
```

```
// Logs: Received: Input 2

// Error handling in generators
function* errorGenerator() {
  try {
    yield 1;
    yield 2;
    yield 3;
  } catch (error) {
    console.log('Caught:', error);
    yield 'Error handled';
  }
}

const gen3 = errorGenerator();

console.log(gen3.next());              // { value: 1, done: false }
console.log(gen3.next());              // { value: 2, done: false }
console.log(gen3.throw('Error!'));     // { value: 'Error handled', done: false }
// Logs: Caught: Error!

// return() method
function* controlledGenerator() {
  try {
    yield 1;
    yield 2;
    yield 3;
  } finally {
    console.log('Cleanup');
  }
}

const gen4 = controlledGenerator();

console.log(gen4.next());              // { value: 1, done: false }
console.log(gen4.return('Terminated')); // { value: 'Terminated', done: true }
// Logs: Cleanup

// Implementing async/await with generators
function run(generatorFunction) {
  const generator = generatorFunction();

  function handle(result) {
    if (result.done) return Promise.resolve(result.value);

    return Promise.resolve(result.value)
      .then(value => handle(generator.next(value)))
```

```
      .catch(error => handle(generator.throw(error)));
  }

  try {
    return handle(generator.next());
  } catch (error) {
    return Promise.reject(error);
  }
}

// Usage
run(function* () {
  try {
    const user = yield fetch('/api/user').then(r => r.json());
    console.log('User:', user);

    const posts = yield fetch(`/api/users/${user.id}/posts`).then(r => r.json());
    console.log('Posts:', posts);

    return { user, posts };
  } catch (error) {
    console.error('Error:', error);
  }
});

// Async generators (ES2018)
async function* asyncNumberGenerator() {
  yield await Promise.resolve(1);
  yield await Promise.resolve(2);
  yield await Promise.resolve(3);
}

// Consume async generator
(async () => {
  for await (const num of asyncNumberGenerator()) {
    console.log(num);
  }
})();

// Real-world: Pagination with async generator
async function* fetchPages(baseUrl) {
  let page = 1;
  let hasMore = true;

  while (hasMore) {
    const response = await fetch(`${baseUrl}?page=${page}`);
    const data = await response.json();
```

```
    yield data.items;

    hasMore = data.hasMore;
    page++;
  }
}

// Usage
(async () => {
  for await (const items of fetchPages('/api/data')) {
    console.log('Page items:', items);
    processItems(items);
  }
})();

// Lazy evaluation with generators
function* fibonacci() {
  let [a, b] = [0, 1];

  while (true) {
    yield a;
    [a, b] = [b, a + b];
  }
}

// Take first N values
function take(n, iterable) {
  const result = [];
  const iterator = iterable[Symbol.iterator]();

  for (let i = 0; i < n; i++) {
    const { value, done } = iterator.next();
    if (done) break;
    result.push(value);
  }

  return result;
}

console.log(take(10, fibonacci()));
// [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

// Generator composition
function* gen1() {
  yield 1;
  yield 2;
}
```

```
function* gen2() {
  yield 3;
  yield 4;
}

function* combined() {
  yield* gen1();  // Delegate to gen1
  yield* gen2();  // Delegate to gen2
  yield 5;
}

console.log([...combined()]);  // [1, 2, 3, 4, 5]
```

Advanced Async Patterns:

```
// Retry with exponential backoff
async function retryWithBackoff(fn, maxRetries = 3, delay = 1000) {
  for (let i = 0; i < maxRetries; i++) {
    try {
      return await fn();
    } catch (error) {
      if (i === maxRetries - 1) throw error;

      const waitTime = delay * Math.pow(2, i);
      console.log(`Retry ${i + 1} after ${waitTime}ms`);
      await new Promise(resolve => setTimeout(resolve, waitTime));
    }
  }
}

// Usage
await retryWithBackoff(() => fetch('/api/data').then(r => r.json()));

// Timeout wrapper
function withTimeout(promise, ms) {
  return Promise.race([
    promise,
    new Promise((_, reject) =>
      setTimeout(() => reject(new Error('Timeout')), ms)
    )
  ]);
}

// Usage
try {
  const data = await withTimeout(fetch('/api/slow-endpoint'), 5000);
} catch (error) {
```

```
  console.error('Request timed out');
}

// Parallel limit (run N promises at a time)
async function parallelLimit(tasks, limit) {
  const results = [];
  const executing = [];

  for (const [index, task] of tasks.entries()) {
    const promise = Promise.resolve().then(() => task()).then(result => {
      results[index] = result;
    });

    results.push(promise);

    if (limit <= tasks.length) {
      const execute = promise.then(() => {
        executing.splice(executing.indexOf(execute), 1);
      });
      executing.push(execute);

      if (executing.length >= limit) {
        await Promise.race(executing);
      }
    }
  }

  await Promise.all(results);
  return results;
}

// Usage
const tasks = Array.from({ length: 10 }, (_, i) =>
  () => fetch(`/api/item/${i}`).then(r => r.json())
);

await parallelLimit(tasks, 3);  // Run 3 at a time

// Queue with concurrency control
class AsyncQueue {
  constructor(concurrency = 1) {
    this.concurrency = concurrency;
    this.running = 0;
    this.queue = [];
  }

  async push(task) {
```

```
    while (this.running >= this.concurrency) {
      await new Promise(resolve => this.queue.push(resolve));
    }

    this.running++;

    try {
      return await task();
    } finally {
      this.running--;

      const resolve = this.queue.shift();
      if (resolve) resolve();
    }
  }
}

// Usage
const queue = new AsyncQueue(2);  // Max 2 concurrent

for (let i = 0; i < 10; i++) {
  queue.push(() => fetch(`/api/item/${i}`)).then(console.log);
}

// Debounce async function
function debounceAsync(fn, delay) {
  let timeoutId;
  let latestResolve;
  let latestReject;

  return function(...args) {
    return new Promise((resolve, reject) => {
      latestResolve = resolve;
      latestReject = reject;

      clearTimeout(timeoutId);

      timeoutId = setTimeout(async () => {
        try {
          const result = await fn(...args);
          latestResolve(result);
        } catch (error) {
          latestReject(error);
        }
      }, delay);
    });
  };
```

```
}

// Usage
const debouncedSearch = debounceAsync(async (query) => {
  const response = await fetch(`/api/search?q=${query}`);
  return await response.json();
}, 300);

// User types: debounced search called only after 300ms pause
searchInput.addEventListener('input', async (e) => {
  const results = await debouncedSearch(e.target.value);
  displayResults(results);
});

// Cancel previous requests
class CancellableRequest {
  constructor() {
    this.currentController = null;
  }

  async fetch(url, options = {}) {
    // Cancel previous request
    if (this.currentController) {
      this.currentController.abort();
    }

    // Create new controller
    this.currentController = new AbortController();

    try {
      const response = await fetch(url, {
        ...options,
        signal: this.currentController.signal
      });

      return await response.json();
    } catch (error) {
      if (error.name === 'AbortError') {
        console.log('Request cancelled');
        return null;
      }
      throw error;
    } finally {
      this.currentController = null;
    }
  }
}
```

```
// Usage
const request = new CancellableRequest();

searchInput.addEventListener('input', async (e) => {
  const results = await request.fetch(`/api/search?q=${e.target.value}`);
  if (results) displayResults(results);
});
```

Chapter 5

Chapter 5: Browser Rendering and Performance Optimization

Understanding the browser's rendering pipeline is crucial for building performant web applications.

5.1 Critical Rendering Path In Detail

The sequence from HTML to pixels involves multiple steps, each with optimization opportunities.

```
// 1. DOM Construction
// HTML → Parse → DOM Tree
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="styles.css">
  </head>
  <body>
    <div id="app">
      <h1>Title</h1>
      <p>Content</p>
    </div>
    <script src="app.js"></script>
  </body>
</html>

// Parser creates DOM nodes incrementally
// Blocks on:
// - Synchronous <script> tags
// - Stylesheets (because scripts might query styles)

// 2. CSSOM Construction
```

```
// CSS → Parse → CSSOM Tree

/* styles.css */
body {
  font-size: 16px;
  color: #333;
}

#app {
  max-width: 1200px;
  margin: 0 auto;
}

h1 {
  font-size: 2em;
  font-weight: bold;
}

// CSSOM blocks rendering until complete
// CSS is render-blocking

// 3. Render Tree
// DOM + CSSOM → Render Tree (only visible nodes)

// Render tree excludes:
// - display: none elements
// - <script>, <meta>, <link>
// - <head> contents

// 4. Layout (Reflow)
// Calculate positions and dimensions

// Triggers layout:
// - Add/remove/update DOM elements
// - Change: width, height, padding, margin, border, position, display
// - Change: font-size, font-family
// - Window resize
// - Reading: offsetWidth, offsetHeight, clientWidth, getBoundingClientRect

// Layout thrashing example (BAD)
elements.forEach(el => {
  const width = el.offsetWidth;  // Read (forces layout)
  el.style.width = (width + 10) + 'px';  // Write
});
// Forces layout N times

// Optimized (GOOD)
```

```
const widths = elements.map(el => el.offsetWidth);  // Read all
elements.forEach((el, i) => {
  el.style.width = (widths[i] + 10) + 'px';  // Write all
});
// Forces layout once

// 5. Paint
// Fill pixels for visible elements

// Paint triggers:
// - Color, background, shadow, border-radius
// - Visibility, outline

// 6. Composite
// Combine layers

// Composite-only properties (fastest):
// - transform
// - opacity
// - filter

// Animation performance comparison
// Bad: triggers layout + paint
.animated {
  animation: slide 1s;
}

@keyframes slide {
  from { left: 0; }
  to { left: 100px; }
}

// Good: triggers only composite
.animated {
  animation: slide 1s;
}

@keyframes slide {
  from { transform: translateX(0); }
  to { transform: translateX(100px); }
}
```

Performance Optimization Techniques:

```
// 1. Batch DOM updates
// Bad
for (let i = 0; i < 1000; i++) {
  const div = document.createElement('div');
```

```
  div.textContent = `Item ${i}`;
  container.appendChild(div);  // Reflow on each append
}

// Good: DocumentFragment
const fragment = document.createDocumentFragment();
for (let i = 0; i < 1000; i++) {
  const div = document.createElement('div');
  div.textContent = `Item ${i}`;
  fragment.appendChild(div);  // No reflow
}
container.appendChild(fragment);  // Single reflow

// 2. Use CSS classes instead of inline styles
// Bad
element.style.width = '100px';
element.style.height = '100px';
element.style.backgroundColor = 'red';
// Three style changes, three potential reflows

// Good
element.classList.add('box');  // Single class change
/* .box { width: 100px; height: 100px; background: red; } */

// 3. Avoid layout thrashing
// Bad
function resizeAllDiv() {
  divs.forEach(div => {
    const width = div.offsetWidth;  // READ (forces layout)
    div.style.width = width * 1.5 + 'px';  // WRITE
  });
}

// Good
function resizeAllDivOptimized() {
  // Batch reads
  const widths = divs.map(div => div.offsetWidth);

  // Batch writes
  divs.forEach((div, i) => {
    div.style.width = widths[i] * 1.5 + 'px';
  });
}

// Or use requestAnimationFrame
function resizeWithRAF() {
  requestAnimationFrame(() => {
```

```
    const widths = divs.map(div => div.offsetWidth);

    requestAnimationFrame(() => {
      divs.forEach((div, i) => {
        div.style.width = widths[i] * 1.5 + 'px';
      });
    });
  });
}

// 4. Use will-change for animations
.animated {
  /* Hint browser to create layer in advance */
  will-change: transform, opacity;
}

.animated:hover {
  transform: scale(1.1);
  opacity: 0.8;
}

// Remove will-change after animation
element.addEventListener('mouseenter', () => {
  element.style.willChange = 'transform';
});

element.addEventListener('mouseleave', () => {
  element.style.willChange = 'auto';
});

// 5. Use CSS containment
.container {
  /* Layout changes inside don't affect outside */
  contain: layout;

  /* Style changes inside don't affect outside */
  contain: style;

  /* Paint operations clipped to padding box */
  contain: paint;

  /* Element dimensions don't depend on children */
  contain: size;

  /* All containment */
  contain: strict;  /* layout + style + paint + size */
  contain: content; /* layout + style + paint */
```

```
}

// 6. content-visibility for off-screen content
.list-item {
  /* Skip rendering off-screen items */
  content-visibility: auto;

  /* Reserve space to avoid layout shifts */
  contain-intrinsic-size: 0 500px;
}

// 7. Debounce expensive operations
function debounce(fn, delay) {
  let timeoutId;

  return function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => fn.apply(this, args), delay);
  };
}

// Debounce scroll handler
const handleScroll = debounce(() => {
  console.log('Scroll position:', window.scrollY);
  updateUI();
}, 100);

window.addEventListener('scroll', handleScroll);

// Throttle: ensure maximum frequency
function throttle(fn, limit) {
  let inThrottle;

  return function(...args) {
    if (!inThrottle) {
      fn.apply(this, args);
      inThrottle = true;
      setTimeout(() => inThrottle = false, limit);
    }
  };
}

const handleResize = throttle(() => {
  console.log('Window size:', window.innerWidth, window.innerHeight);
  recalculateLayout();
}, 200);
```

```
window.addEventListener('resize', handleResize);

// 8. Passive event listeners
// Allows scroll without waiting for handler
window.addEventListener('scroll', handleScroll, { passive: true });

// Can't preventDefault with passive
window.addEventListener('touchstart', (e) => {
  // e.preventDefault(); // Error with passive: true
}, { passive: true });

// 9. Virtual scrolling for long lists
class VirtualList {
  constructor(container, items, itemHeight) {
    this.container = container;
    this.items = items;
    this.itemHeight = itemHeight;

    this.visibleCount = Math.ceil(container.clientHeight / itemHeight) + 1;
    this.startIndex = 0;

    this.totalHeight = items.length * itemHeight;
    this.container.style.height = this.totalHeight + 'px';
    this.container.style.position = 'relative';

    this.render();

    container.addEventListener('scroll', () => this.onScroll(), { passive: true });
  }

  onScroll() {
    const scrollTop = this.container.scrollTop;
    this.startIndex = Math.floor(scrollTop / this.itemHeight);
    this.render();
  }

  render() {
    const endIndex = Math.min(
      this.startIndex + this.visibleCount,
      this.items.length
    );

    const fragment = document.createDocumentFragment();

    for (let i = this.startIndex; i < endIndex; i++) {
      const item = document.createElement('div');
      item.className = 'list-item';
```

```
      item.style.position = 'absolute';
      item.style.top = (i * this.itemHeight) + 'px';
      item.style.height = this.itemHeight + 'px';
      item.textContent = this.items[i];

      fragment.appendChild(item);
    }

    this.container.innerHTML = '';
    this.container.appendChild(fragment);
  }
}

// Usage
const items = Array.from({ length: 10000 }, (_, i) => `Item ${i}`);
new VirtualList(document.getElementById('list'), items, 50);

// 10. Intersection Observer for lazy loading
const imageObserver = new IntersectionObserver((entries) => {
  entries.forEach(entry => {
    if (entry.isIntersecting) {
      const img = entry.target;
      img.src = img.dataset.src;
      imageObserver.unobserve(img);
    }
  });
}, {
  rootMargin: '50px'  // Load slightly before visible
});

document.querySelectorAll('img[data-src]').forEach(img => {
  imageObserver.observe(img);
});
```

Web Workers for Performance:

```
// main.js
const worker = new Worker('worker.js');

// Send data to worker
worker.postMessage({ data: largeDataset, operation: 'process' });

// Receive result from worker
worker.addEventListener('message', (event) => {
  console.log('Result:', event.data);
  updateUI(event.data);
});
```

```javascript
// Error handling
worker.addEventListener('error', (error) => {
  console.error('Worker error:', error);
});

// Terminate worker
worker.terminate();

// worker.js
self.addEventListener('message', (event) => {
  const { data, operation } = event.data;

  if (operation === 'process') {
    // CPU-intensive work off main thread
    const result = processData(data);

    // Send result back
    self.postMessage(result);
  }
});

function processData(data) {
  // Heavy computation
  return data.map(item => {
    // Complex calculations
    return transformItem(item);
  });
}

// Shared Worker (shared across tabs/windows)
// shared-worker.js
const connections = [];

self.addEventListener('connect', (event) => {
  const port = event.ports[0];
  connections.push(port);

  port.addEventListener('message', (event) => {
    // Broadcast to all connections
    connections.forEach(p => {
      p.postMessage(event.data);
    });
  });

  port.start();
});
```

```
// main.js
const worker = new SharedWorker('shared-worker.js');

worker.port.addEventListener('message', (event) => {
  console.log('Received:', event.data);
});

worker.port.start();
worker.port.postMessage('Hello');
```

Memory Management:

```
// Memory leaks
// 1. Forgotten timers
const interval = setInterval(() => {
  updateData();
}, 1000);

// Fix: clear when done
clearInterval(interval);

// 2. Forgotten event listeners
const handler = () => console.log('clicked');
element.addEventListener('click', handler);

// Element removed but listener still referenced
element.remove();  // Memory leak!

// Fix: remove listener
element.removeEventListener('click', handler);
element.remove();

// Or use AbortController
const controller = new AbortController();

element.addEventListener('click', handler, {
  signal: controller.signal
});

// Later: removes all listeners with this signal
controller.abort();
element.remove();

// 3. Closures retaining large objects
function createProcessor() {
  const hugeData = new Array(1000000).fill('data');

  return {
```

```
    process: () => {
      // Closure retains hugeData even if unused
      return 'processed';
    }
  };
}

// Fix: null reference when done
function createProcessorFixed() {
  let hugeData = new Array(1000000).fill('data');
  const firstItem = hugeData[0];

  const processor = {
    process: () => {
      return firstItem;  // Only retains firstItem
    },
    cleanup: () => {
      hugeData = null;  // Allow GC
    }
  };

  hugeData = null;  // If not needed after creation
  return processor;
}

// 4. Detached DOM nodes
let detached = document.createElement('div');
detached.innerHTML = '<div>'.repeat(10000);

// detached kept in memory even though not in document
// Fix: null when done
detached = null;

// Chrome DevTools Memory Profiling
// 1. Take heap snapshot
// 2. Perform action
// 3. Take another snapshot
// 4. Compare snapshots
// 5. Look for objects that weren't garbage collected

// Measure memory usage
if (performance.memory) {
  console.log('Used JS Heap:', performance.memory.usedJSHeapSize);
  console.log('Total JS Heap:', performance.memory.totalJSHeapSize);
  console.log('Heap Limit:', performance.memory.jsHeapSizeLimit);
}
```

```
// WeakMap/WeakSet for metadata
// Garbage collected when key object is no longer referenced
const metadata = new WeakMap();

function attachMetadata(element, data) {
  metadata.set(element, data);
}

// When element is removed and no longer referenced,
// metadata is automatically garbage collected

// Object pool for frequently created objects
class ObjectPool {
  constructor(createFn, resetFn, initialSize = 10) {
    this.createFn = createFn;
    this.resetFn = resetFn;
    this.pool = [];

    for (let i = 0; i < initialSize; i++) {
      this.pool.push(this.createFn());
    }
  }

  acquire() {
    if (this.pool.length > 0) {
      return this.pool.pop();
    }
    return this.createFn();
  }

  release(obj) {
    this.resetFn(obj);
    this.pool.push(obj);
  }
}

// Usage
const particlePool = new ObjectPool(
  () => ({ x: 0, y: 0, vx: 0, vy: 0 }),
  (particle) => {
    particle.x = 0;
    particle.y = 0;
    particle.vx = 0;
    particle.vy = 0;
  },
  100
);
```

```
// Reuse particles instead of creating new ones
const particle = particlePool.acquire();
// Use particle...
particlePool.release(particle);
```

Chapter 6

Chapter 6: Networking and Browser APIs

Modern web applications rely heavily on browser APIs for data fetching, storage, and real-time communication.

6.1 Fetch API Deep Dive

The Fetch API provides a modern interface for making HTTP requests.

```
// Basic fetch
fetch('/api/users')
  .then(response => {
    // Response object contains metadata
    console.log('Status:', response.status);
    console.log('Headers:', response.headers);
    console.log('OK:', response.ok);  // true if status 200-299

    return response.json();
  })
  .then(data => {
    console.log('Data:', data);
  })
  .catch(error => {
    console.error('Error:', error);
  });

// Fetch only rejects on network errors, not HTTP errors
fetch('/api/not-found')
  .then(response => {
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
```

```
    return response.json();
  })
  .catch(error => {
    console.error('Failed:', error);
  });

// Request configuration
const options = {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer token123'
  },
  body: JSON.stringify({
    name: 'John',
    email: 'john@example.com'
  }),
  credentials: 'include',  // Send cookies
  mode: 'cors',  // cors, no-cors, same-origin
  cache: 'no-cache',  // default, no-store, reload, no-cache, force-cache
  redirect: 'follow'  // follow, error, manual
};

fetch('/api/users', options)
  .then(response => response.json())
  .then(data => console.log(data));

// Abort controller for cancellation
const controller = new AbortController();
const signal = controller.signal;

fetch('/api/data', { signal })
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => {
    if (error.name === 'AbortError') {
      console.log('Request was aborted');
    }
  });

// Cancel after 5 seconds
setTimeout(() => controller.abort(), 5000);

// Response types
// text()
fetch('/api/text')
  .then(response => response.text())
```

```
  .then(text => console.log(text));

// json()
fetch('/api/json')
  .then(response => response.json())
  .then(data => console.log(data));

// blob() for files
fetch('/images/photo.jpg')
  .then(response => response.blob())
  .then(blob => {
    const url = URL.createObjectURL(blob);
    img.src = url;
  });

// arrayBuffer() for binary data
fetch('/api/binary')
  .then(response => response.arrayBuffer())
  .then(buffer => {
    const view = new DataView(buffer);
    // Process binary data
  });

// formData()
fetch('/api/upload')
  .then(response => response.formData())
  .then(formData => console.log(formData));

// Streaming response
fetch('/api/large-file')
  .then(response => {
    const reader = response.body.getReader();

    return new ReadableStream({
      start(controller) {
        function push() {
          reader.read().then(({ done, value }) => {
            if (done) {
              controller.close();
              return;
            }

            controller.enqueue(value);
            push();
          });
        }
```

```
        push();
      }
    });
  })
  .then(stream => new Response(stream))
  .then(response => response.blob())
  .then(blob => console.log('Downloaded:', blob.size));

// Upload with progress
async function uploadWithProgress(file) {
  const formData = new FormData();
  formData.append('file', file);

  const xhr = new XMLHttpRequest();

  return new Promise((resolve, reject) => {
    xhr.upload.addEventListener('progress', (e) => {
      if (e.lengthComputable) {
        const percentComplete = (e.loaded / e.total) * 100;
        console.log(`Upload progress: ${percentComplete}%`);
      }
    });

    xhr.addEventListener('load', () => {
      if (xhr.status >= 200 && xhr.status < 300) {
        resolve(JSON.parse(xhr.responseText));
      } else {
        reject(new Error(`HTTP ${xhr.status}`));
      }
    });

    xhr.addEventListener('error', () => reject(new Error('Network error')));

    xhr.open('POST', '/api/upload');
    xhr.send(formData);
  });
}

// Wrapper with retry logic
async function fetchWithRetry(url, options = {}, retries = 3, delay = 1000) {
  for (let i = 0; i < retries; i++) {
    try {
      const response = await fetch(url, options);

      if (!response.ok) {
        throw new Error(`HTTP ${response.status}`);
      }
```

```
      return await response.json();
    } catch (error) {
      if (i === retries - 1) throw error;

      console.log(`Retry ${i + 1}/${retries} after ${delay}ms`);
      await new Promise(resolve => setTimeout(resolve, delay));
      delay *= 2;  // Exponential backoff
    }
  }
}
```

6.2 Advanced Error Handling and Edge Cases

Real-world applications must handle numerous edge cases: network failures, timeouts, malformed responses, race conditions, and more.

6.2.1 Comprehensive Error Handling

```
// Custom error types for different scenarios
class NetworkError extends Error {
  constructor(message, statusCode, response) {
    super(message);
    this.name = 'NetworkError';
    this.statusCode = statusCode;
    this.response = response;
  }
}

class TimeoutError extends Error {
  constructor(message, timeout) {
    super(message);
    this.name = 'TimeoutError';
    this.timeout = timeout;
  }
}

class AbortError extends Error {
  constructor(message) {
    super(message);
    this.name = 'AbortError';
  }
}

class ParseError extends Error {
  constructor(message, rawData) {
```

```
    super(message);
    this.name = 'ParseError';
    this.rawData = rawData;
  }
}

// Robust fetch wrapper with all error cases
async function robustFetch(url, options = {}) {
  const {
    timeout = 10000,
    retries = 3,
    retryDelay = 1000,
    // Default retryable status codes (transient errors)
    // 408: Request Timeout
    // 429: Too Many Requests (rate limit)
    // 500: Internal Server Error
    // 502: Bad Gateway
    // 503: Service Unavailable
    // 504: Gateway Timeout
    retryOn = [408, 429, 500, 502, 503, 504],
    onRetry = null,
    signal,
    ...fetchOptions
  } = options;

  // Create combined abort controller for timeout and cancellation
  const controller = new AbortController();
  const combinedSignal = signal ? combineSignals([signal, controller.signal]) : control

  let lastError;

  for (let attempt = 0; attempt < retries; attempt++) {
    try {
      // Set up timeout
      const timeoutId = setTimeout(() => controller.abort(), timeout);

      const response = await fetch(url, {
        ...fetchOptions,
        signal: combinedSignal
      });

      clearTimeout(timeoutId);

      // Handle HTTP errors
      if (!response.ok) {
        // Check if we should retry this status code
        if (retryOn.includes(response.status) && attempt < retries - 1) {
```

```
        const delay = retryDelay * Math.pow(2, attempt); // Exponential backoff

        if (onRetry) {
          onRetry(attempt + 1, delay, response.status);
        }

        await sleep(delay);
        continue;
      }

      // Clone response to read body for error
      const errorBody = await response.clone().text().catch(() => '');

      throw new NetworkError(
        `HTTP ${response.status}: ${response.statusText}`,
        response.status,
        errorBody
      );
    }

    // Parse response based on content type
    const contentType = response.headers.get('content-type');
    let data;

    try {
      if (contentType?.includes('application/json')) {
        data = await response.json();
      } else if (contentType?.includes('text/')) {
        data = await response.text();
      } else {
        data = await response.blob();
      }
    } catch (parseError) {
      throw new ParseError(
        'Failed to parse response',
        await response.text().catch(() => '')
      );
    }

    return {
      data,
      response,
      ok: true,
      status: response.status
    };

  } catch (error) {
```

```
      lastError = error;

      // Handle abort errors
      if (error.name === 'AbortError') {
        if (signal?.aborted) {
          throw new AbortError('Request was cancelled by user');
        }
        throw new TimeoutError(`Request timeout after ${timeout}ms`, timeout);
      }

      // Network errors (no internet, CORS, etc.)
      if (error instanceof TypeError && error.message.includes('fetch')) {
        if (attempt < retries - 1) {
          const delay = retryDelay * Math.pow(2, attempt);

          if (onRetry) {
            onRetry(attempt + 1, delay, 'Network Error');
          }

          await sleep(delay);
          continue;
        }

        throw new NetworkError('Network request failed. Check your connection.', 0, nul
      }

      // Don't retry for non-retryable errors
      if (error instanceof NetworkError || error instanceof ParseError) {
        throw error;
      }

      // Unknown error - retry if attempts remain
      if (attempt < retries - 1) {
        const delay = retryDelay * Math.pow(2, attempt);
        await sleep(delay);
        continue;
      }

      throw error;
    }
  }

  throw lastError;
}

// Helper to combine multiple AbortSignals
function combineSignals(signals) {
```

```
  const controller = new AbortController();

  for (const signal of signals) {
    if (signal.aborted) {
      controller.abort();
      break;
    }
    signal.addEventListener('abort', () => controller.abort());
  }

  return controller.signal;
}

function sleep(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

// Usage with comprehensive error handling
async function fetchUserData(userId) {
  try {
    // Example 1: Use default retryable status codes
    const result = await robustFetch(`/api/users/${userId}`, {
      timeout: 5000,
      retries: 3,
      retryDelay: 1000,
      // retryOn uses default: [408, 429, 500, 502, 503, 504]
      onRetry: (attempt, delay, reason) => {
        console.log(`Retry attempt ${attempt} after ${delay}ms (reason: ${reason})`);
      }
    });

    return result.data;

  } catch (error) {
    if (error instanceof TimeoutError) {
      console.error('Request timed out:', error.timeout);
      // Show timeout message to user
      showNotification('Request is taking longer than expected. Please try again.');
    } else if (error instanceof NetworkError) {
      console.error('Network error:', error.statusCode, error.message);

      if (error.statusCode === 401) {
        // Handle unauthorized - redirect to login
        redirectToLogin();
      } else if (error.statusCode === 403) {
        // Handle forbidden - show permission error
        showNotification('You do not have permission to access this resource.');
```

```
      } else if (error.statusCode === 404) {
        // Handle not found
        showNotification('User not found.');
      } else if (error.statusCode >= 500) {
        // Handle server errors
        showNotification('Server error. Please try again later.');
      }
    } else if (error instanceof AbortError) {
      console.log('Request was cancelled');
    } else if (error instanceof ParseError) {
      console.error('Failed to parse response:', error.rawData);
      showNotification('Received invalid data from server.');
    } else {
      console.error('Unexpected error:', error);
      showNotification('An unexpected error occurred.');
    }

    throw error;
  }
}

// Example 2: Customize retryable status codes
async function fetchCriticalData() {
  try {
    // Only retry on server errors, not rate limits
    const result = await robustFetch('/api/critical-data', {
      retries: 5,
      retryOn: [500, 502, 503, 504], // Exclude 408 and 429
      retryDelay: 2000
    });
    return result.data;
  } catch (error) {
    console.error('Failed after retries:', error);
    throw error;
  }
}

// Example 3: Don't retry at all (empty array)
async function fetchNonRetryable() {
  try {
    const result = await robustFetch('/api/no-retry', {
      retryOn: [], // Never retry on HTTP errors
      retries: 0
    });
    return result.data;
  } catch (error) {
    console.error('Failed without retry:', error);
```

```
    throw error;
  }
}

// Example 4: Retry on custom status codes
async function fetchWithCustomRetry() {
  try {
    // Also retry on 404 (maybe resource not ready yet)
    const result = await robustFetch('/api/pending-resource', {
      retryOn: [404, 408, 429, 500, 502, 503, 504],
      retries: 10,
      retryDelay: 500,
      onRetry: (attempt, delay, reason) => {
        if (reason === 404) {
          console.log(`Resource not ready yet, retry ${attempt}...`);
        }
      }
    });
    return result.data;
  } catch (error) {
    console.error('Resource never became available:', error);
    throw error;
  }
}

// Example 5: Only retry on rate limits
async function fetchWithRateLimitRetry() {
  try {
    const result = await robustFetch('/api/rate-limited', {
      retryOn: [429], // Only retry rate limits
      retries: 5,
      retryDelay: 5000, // Wait longer for rate limits
      onRetry: (attempt, delay, reason) => {
        console.log(`Rate limited. Waiting ${delay}ms before retry ${attempt}...`);
      }
    });
    return result.data;
  } catch (error) {
    console.error('Still rate limited after retries:', error);
    throw error;
  }
}
```

6.2.2 Request Cancellation and Cleanup

```
// Advanced cancellation patterns

// 1. Cancelling on component unmount (React example)
function UserProfile({ userId }) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    const controller = new AbortController();

    async function loadUser() {
      try {
        const result = await robustFetch(`/api/users/${userId}`, {
          signal: controller.signal
        });
        setUser(result.data);
      } catch (error) {
        if (error instanceof AbortError) {
          console.log('Request cancelled - component unmounted');
        } else {
          console.error('Failed to load user:', error);
        }
      }
    }

    loadUser();

    // Cleanup: abort request if component unmounts
    return () => controller.abort();
  }, [userId]);

  return user ? <div>{user.name}</div> : <div>Loading...</div>;
}

// 2. Cancelling previous request when new one starts
class SearchComponent {
  constructor() {
    this.currentRequest = null;
  }

  async search(query) {
    // Cancel previous request
    if (this.currentRequest) {
      this.currentRequest.abort();
    }

    // Create new controller for this request
    const controller = new AbortController();
```

```
    this.currentRequest = controller;

    try {
      const result = await robustFetch(`/api/search?q=${query}`, {
        signal: controller.signal
      });

      this.displayResults(result.data);
    } catch (error) {
      if (!(error instanceof AbortError)) {
        console.error('Search failed:', error);
      }
    } finally {
      // Clear if this was the current request
      if (this.currentRequest === controller) {
        this.currentRequest = null;
      }
    }
  }
}

// 3. Timeout-based cancellation
async function fetchWithTimeout(url, timeout = 5000) {
  const controller = new AbortController();

  const timeoutId = setTimeout(() => {
    controller.abort();
  }, timeout);

  try {
    const response = await fetch(url, { signal: controller.signal });
    clearTimeout(timeoutId);
    return await response.json();
  } catch (error) {
    clearTimeout(timeoutId);

    if (error.name === 'AbortError') {
      throw new TimeoutError(`Request timeout after ${timeout}ms`, timeout);
    }
    throw error;
  }
}

// 4. Manual cancellation with token
class CancellableRequest {
  constructor() {
    this.controller = new AbortController();
```

```
  }

  async fetch(url, options = {}) {
    return fetch(url, {
      ...options,
      signal: this.controller.signal
    });
  }

  cancel(reason = 'User cancelled') {
    this.controller.abort();
    console.log('Request cancelled:', reason);
  }

  get isCancelled() {
    return this.controller.signal.aborted;
  }
}

// Usage
const request = new CancellableRequest();

// Start request
request.fetch('/api/large-data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => {
    if (error.name === 'AbortError') {
      console.log('Request was cancelled');
    }
  });

// Cancel from button click
button.addEventListener('click', () => {
  request.cancel('User clicked cancel');
});
```

6.2.3 Parallel Requests and Concurrency Control

```
// 1. Simple parallel requests
async function loadDashboard() {
  try {
    const [users, posts, comments] = await Promise.all([
      robustFetch('/api/users'),
      robustFetch('/api/posts'),
      robustFetch('/api/comments')
```

```
    ]);

    return {
      users: users.data,
      posts: posts.data,
      comments: comments.data
    };
  } catch (error) {
    console.error('Failed to load dashboard:', error);
    throw error;
  }
}

// 2. Parallel requests with partial failure handling
async function loadDashboardWithFallbacks() {
  const results = await Promise.allSettled([
    robustFetch('/api/users'),
    robustFetch('/api/posts'),
    robustFetch('/api/comments')
  ]);

  const dashboard = {
    users: [],
    posts: [],
    comments: [],
    errors: []
  };

  results.forEach((result, index) => {
    const key = ['users', 'posts', 'comments'][index];

    if (result.status === 'fulfilled') {
      dashboard[key] = result.value.data;
    } else {
      console.error(`Failed to load ${key}:`, result.reason);
      dashboard.errors.push({ key, error: result.reason });
    }
  });

  return dashboard;
}

// 3. Controlled concurrency - limit parallel requests
class RequestQueue {
  constructor(maxConcurrent = 3) {
    this.maxConcurrent = maxConcurrent;
    this.running = 0;
```

```
    this.queue = [];
  }

  async add(fetchFn) {
    // Wait if at max concurrency
    while (this.running >= this.maxConcurrent) {
      await new Promise(resolve => this.queue.push(resolve));
    }

    this.running++;

    try {
      const result = await fetchFn();
      return result;
    } finally {
      this.running--;

      // Process next in queue
      const resolve = this.queue.shift();
      if (resolve) resolve();
    }
  }
}

// Usage: limit to 3 concurrent requests
const queue = new RequestQueue(3);

async function loadManyUsers(userIds) {
  const promises = userIds.map(id =>
    queue.add(() => robustFetch(`/api/users/${id}`))
  );

  return Promise.all(promises);
}

// 4. Request deduplication - avoid duplicate in-flight requests
class RequestDeduplicator {
  constructor() {
    this.cache = new Map();
  }

  async fetch(url, options = {}) {
    // Create cache key from url and options
    const key = JSON.stringify({ url, options });

    // Return existing promise if request is in flight
    if (this.cache.has(key)) {
```

```
      console.log('Returning cached promise for:', url);
      return this.cache.get(key);
    }

    // Create new request
    const promise = robustFetch(url, options)
      .finally(() => {
        // Remove from cache when complete
        this.cache.delete(key);
      });

    this.cache.set(key, promise);
    return promise;
  }

  clear() {
    this.cache.clear();
  }
}

// Usage: multiple components requesting same data
const deduplicator = new RequestDeduplicator();

// Both calls will use the same underlying request
const data1 = await deduplicator.fetch('/api/user/123');
const data2 = await deduplicator.fetch('/api/user/123'); // Returns same promise

// 5. Race condition handling - use latest response
class LatestRequestTracker {
  constructor() {
    this.requestId = 0;
  }

  async fetch(url, options = {}) {
    const currentId = ++this.requestId;

    const result = await robustFetch(url, options);

    // Check if this is still the latest request
    if (currentId !== this.requestId) {
      throw new Error('Stale request - newer request in progress');
    }

    return result;
  }
}
```

```
// Usage: only use response from latest request
const tracker = new LatestRequestTracker();

input.addEventListener('input', async (e) => {
  try {
    const result = await tracker.fetch(`/api/search?q=${e.target.value}`);
    displayResults(result.data);
  } catch (error) {
    if (error.message.includes('Stale request')) {
      console.log('Ignoring stale response');
    } else {
      console.error('Search failed:', error);
    }
  }
});
```

6.2.4 Offline Handling and Request Queuing

```
// Comprehensive offline support

class OfflineRequestQueue {
  constructor() {
    this.queue = [];
    this.isOnline = navigator.onLine;
    this.processing = false;

    // Listen for online/offline events
    window.addEventListener('online', () => this.handleOnline());
    window.addEventListener('offline', () => this.handleOffline());

    // Load persisted queue from IndexedDB
    this.loadQueue();
  }

  async loadQueue() {
    try {
      const db = await this.openDB();
      const transaction = db.transaction(['requests'], 'readonly');
      const store = transaction.objectStore('requests');
      const requests = await this.getAllFromStore(store);
      this.queue = requests;
    } catch (error) {
      console.error('Failed to load request queue:', error);
    }
  }
```

```
  async persistQueue() {
    try {
      const db = await this.openDB();
      const transaction = db.transaction(['requests'], 'readwrite');
      const store = transaction.objectStore('requests');

      // Clear existing
      await store.clear();

      // Add all queued requests
      for (const request of this.queue) {
        await store.add(request);
      }
    } catch (error) {
      console.error('Failed to persist queue:', error);
    }
  }

  async openDB() {
    return new Promise((resolve, reject) => {
      const request = indexedDB.open('OfflineQueue', 1);

      request.onupgradeneeded = (e) => {
        const db = e.target.result;
        if (!db.objectStoreNames.contains('requests')) {
          db.createObjectStore('requests', { keyPath: 'id', autoIncrement: true });
        }
      };

      request.onsuccess = () => resolve(request.result);
      request.onerror = () => reject(request.error);
    });
  }

  getAllFromStore(store) {
    return new Promise((resolve, reject) => {
      const request = store.getAll();
      request.onsuccess = () => resolve(request.result);
      request.onerror = () => reject(request.error);
    });
  }

  handleOnline() {
    console.log('Connection restored');
    this.isOnline = true;
    this.processQueue();
  }
```

```
  handleOffline() {
    console.log('Connection lost');
    this.isOnline = false;
  }

  async enqueue(url, options = {}) {
    const request = {
      id: Date.now() + Math.random(),
      url,
      options,
      timestamp: Date.now(),
      attempts: 0
    };

    this.queue.push(request);
    await this.persistQueue();

    // Try to process if online
    if (this.isOnline) {
      this.processQueue();
    }

    return request.id;
  }

  async processQueue() {
    if (this.processing || !this.isOnline || this.queue.length === 0) {
      return;
    }

    this.processing = true;

    while (this.queue.length > 0 && this.isOnline) {
      const request = this.queue[0];

      try {
        const result = await robustFetch(request.url, {
          ...request.options,
          timeout: 30000,
          retries: 2
        });

        console.log('Successfully processed offline request:', request.url);

        // Remove from queue
        this.queue.shift();
        await this.persistQueue();
```

```
        // Notify success
        this.dispatchEvent('request-success', { request, result });

      } catch (error) {
        console.error('Failed to process offline request:', error);

        request.attempts++;

        // Remove if max attempts reached
        if (request.attempts >= 3) {
          console.log('Max attempts reached, removing request');
          this.queue.shift();
          await this.persistQueue();

          this.dispatchEvent('request-failed', { request, error });
        } else {
          // Move to back of queue
          this.queue.shift();
          this.queue.push(request);
          await this.persistQueue();
        }

        // Wait before next attempt
        await sleep(1000);
      }
    }

    this.processing = false;
  }

  dispatchEvent(name, detail) {
    window.dispatchEvent(new CustomEvent(`offline-queue:${name}`, { detail }));
  }

  getQueueSize() {
    return this.queue.length;
  }

  clearQueue() {
    this.queue = [];
    this.persistQueue();
  }
}

// Usage
const offlineQueue = new OfflineRequestQueue();
```

```
// Listen for queue events
window.addEventListener('offline-queue:request-success', (e) => {
  console.log('Request processed:', e.detail);
  showNotification('Pending changes synced');
});

window.addEventListener('offline-queue:request-failed', (e) => {
  console.log('Request failed permanently:', e.detail);
  showNotification('Failed to sync some changes');
});

// Make request (queued if offline)
async function saveUserData(userData) {
  if (!navigator.onLine) {
    console.log('Offline - queueing request');
    await offlineQueue.enqueue('/api/users', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(userData)
    });

    showNotification('Saved locally. Will sync when online.');
    return { queued: true };
  }

  try {
    const result = await robustFetch('/api/users', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(userData)
    });

    return result.data;
  } catch (error) {
    // Queue if network error
    if (error instanceof NetworkError && error.statusCode === 0) {
      await offlineQueue.enqueue('/api/users', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(userData)
      });

      showNotification('Saved locally. Will sync when online.');
      return { queued: true };
    }

    throw error;
```

```
  }
}

// Optimistic updates with offline support
async function updateUserProfile(userId, updates) {
  // Optimistically update UI
  const previousData = getUserFromCache(userId);
  updateUserInCache(userId, { ...previousData, ...updates });
  renderUser(userId);

  try {
    const result = await robustFetch(`/api/users/${userId}`, {
      method: 'PATCH',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(updates)
    });

    // Update with server response
    updateUserInCache(userId, result.data);
    renderUser(userId);

  } catch (error) {
    if (!navigator.onLine || (error instanceof NetworkError && error.statusCode === 0)) {
      // Queue for later
      await offlineQueue.enqueue(`/api/users/${userId}`, {
        method: 'PATCH',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(updates)
      });

      showNotification('Changes saved locally. Will sync when online.');
    } else {
      // Revert optimistic update
      updateUserInCache(userId, previousData);
      renderUser(userId);

      showNotification('Failed to save changes');
      throw error;
    }
  }
}
```

6.2.5 Request and Response Interceptors

```
// HTTP client with interceptor support (similar to Axios)
```

```
class HttpClient {
  constructor(baseURL = '', defaultOptions = {}) {
    this.baseURL = baseURL;
    this.defaultOptions = defaultOptions;
    this.requestInterceptors = [];
    this.responseInterceptors = [];
  }

  // Add request interceptor
  addRequestInterceptor(onFulfilled, onRejected) {
    this.requestInterceptors.push({ onFulfilled, onRejected });

    // Return ID for removal
    return this.requestInterceptors.length - 1;
  }

  // Add response interceptor
  addResponseInterceptor(onFulfilled, onRejected) {
    this.responseInterceptors.push({ onFulfilled, onRejected });
    return this.responseInterceptors.length - 1;
  }

  // Remove interceptor by ID
  removeRequestInterceptor(id) {
    this.requestInterceptors[id] = null;
  }

  removeResponseInterceptor(id) {
    this.responseInterceptors[id] = null;
  }

  async request(url, options = {}) {
    // Build full URL
    const fullURL = url.startsWith('http') ? url : `${this.baseURL}${url}`;

    // Merge options
    let config = {
      ...this.defaultOptions,
      ...options,
      url: fullURL
    };

    try {
      // Run request interceptors
      for (const interceptor of this.requestInterceptors) {
        if (interceptor && interceptor.onFulfilled) {
          config = await interceptor.onFulfilled(config);
```

```
      }
    }

    // Make request
    const response = await fetch(config.url, config);

    let result = {
      data: null,
      status: response.status,
      statusText: response.statusText,
      headers: response.headers,
      config,
      response
    };

    // Parse response
    const contentType = response.headers.get('content-type');
    if (contentType?.includes('application/json')) {
      result.data = await response.json();
    } else if (contentType?.includes('text/')) {
      result.data = await response.text();
    } else {
      result.data = await response.blob();
    }

    // Run response interceptors
    for (const interceptor of this.responseInterceptors) {
      if (interceptor && interceptor.onFulfilled) {
        result = await interceptor.onFulfilled(result);
      }
    }

    // Check for HTTP errors after interceptors
    if (!response.ok) {
      throw new NetworkError(
        `HTTP ${response.status}: ${response.statusText}`,
        response.status,
        result.data
      );
    }

    return result;

  } catch (error) {
    // Run error interceptors
    let finalError = error;
```

```
    for (const interceptor of this.requestInterceptors) {
      if (interceptor && interceptor.onRejected) {
        try {
          finalError = await interceptor.onRejected(finalError);
        } catch (err) {
          finalError = err;
        }
      }
    }

    for (const interceptor of this.responseInterceptors) {
      if (interceptor && interceptor.onRejected) {
        try {
          finalError = await interceptor.onRejected(finalError);
        } catch (err) {
          finalError = err;
        }
      }
    }

    throw finalError;
  }
}

get(url, options) {
  return this.request(url, { ...options, method: 'GET' });
}

post(url, data, options) {
  return this.request(url, {
    ...options,
    method: 'POST',
    body: JSON.stringify(data),
    headers: {
      'Content-Type': 'application/json',
      ...options?.headers
    }
  });
}

put(url, data, options) {
  return this.request(url, {
    ...options,
    method: 'PUT',
    body: JSON.stringify(data),
    headers: {
      'Content-Type': 'application/json',
```

```
        ...options?.headers
      }
    });
  }

  delete(url, options) {
    return this.request(url, { ...options, method: 'DELETE' });
  }
}

// Usage: Create client with interceptors

const api = new HttpClient('https://api.example.com', {
  headers: {
    'Content-Type': 'application/json'
  }
});

// 1. Authentication interceptor - add token to requests
api.addRequestInterceptor(
  (config) => {
    const token = localStorage.getItem('authToken');
    if (token) {
      config.headers = {
        ...config.headers,
        'Authorization': `Bearer ${token}`
      };
    }
    return config;
  },
  (error) => {
    console.error('Request interceptor error:', error);
    return Promise.reject(error);
  }
);

// 2. Logging interceptor
api.addRequestInterceptor(
  (config) => {
    console.log('Request:', config.method, config.url);
    config.metadata = { startTime: Date.now() };
    return config;
  }
);

api.addResponseInterceptor(
  (response) => {
```

```
    const duration = Date.now() - response.config.metadata.startTime;
    console.log(`Response: ${response.status} (${duration}ms)`);
    return response;
  }
);

// 3. Token refresh interceptor
let isRefreshing = false;
let failedQueue = [];

const processQueue = (error, token = null) => {
  failedQueue.forEach(prom => {
    if (error) {
      prom.reject(error);
    } else {
      prom.resolve(token);
    }
  });

  failedQueue = [];
};

api.addResponseInterceptor(
  (response) => response,
  async (error) => {
    const originalRequest = error.config;

    // If 401 and not already retrying
    if (error.statusCode === 401 && !originalRequest._retry) {
      if (isRefreshing) {
        // Queue this request
        return new Promise((resolve, reject) => {
          failedQueue.push({ resolve, reject });
        }).then(token => {
          originalRequest.headers['Authorization'] = `Bearer ${token}`;
          return api.request(originalRequest.url, originalRequest);
        });
      }

      originalRequest._retry = true;
      isRefreshing = true;

      try {
        // Refresh token
        const refreshToken = localStorage.getItem('refreshToken');
        const response = await fetch('/api/auth/refresh', {
          method: 'POST',
```

```
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({ refreshToken })
        });

        const data = await response.json();
        const newToken = data.token;

        localStorage.setItem('authToken', newToken);

        // Update original request
        originalRequest.headers['Authorization'] = `Bearer ${newToken}`;

        // Process queued requests
        processQueue(null, newToken);

        // Retry original request
        return api.request(originalRequest.url, originalRequest);

      } catch (refreshError) {
        processQueue(refreshError, null);

        // Redirect to login
        localStorage.removeItem('authToken');
        localStorage.removeItem('refreshToken');
        window.location.href = '/login';

        return Promise.reject(refreshError);
      } finally {
        isRefreshing = false;
      }
    }

    return Promise.reject(error);
  }
);

// 4. Error transformation interceptor
api.addResponseInterceptor(
  (response) => response,
  (error) => {
    // Transform error to user-friendly message
    const userMessage = {
      400: 'Invalid request. Please check your input.',
      401: 'Please log in to continue.',
      403: 'You do not have permission to access this resource.',
      404: 'The requested resource was not found.',
      429: 'Too many requests. Please try again later.',
```

```
      500: 'Server error. Please try again later.',
      503: 'Service temporarily unavailable. Please try again later.'
    };

    error.userMessage = userMessage[error.statusCode] || 'An unexpected error occurred.

    return Promise.reject(error);
  }
);

// 5. Retry interceptor for specific errors
api.addResponseInterceptor(
  (response) => response,
  async (error) => {
    const config = error.config;

    // Retry on network errors or 503
    const shouldRetry = !config._retryCount && (
      error.statusCode === 503 ||
      error.statusCode === 0
    );

    if (shouldRetry) {
      config._retryCount = (config._retryCount || 0) + 1;

      if (config._retryCount <= 3) {
        const delay = Math.pow(2, config._retryCount) * 1000;
        console.log(`Retrying request (attempt ${config._retryCount}) after ${delay}ms`

        await sleep(delay);
        return api.request(config.url, config);
      }
    }

    return Promise.reject(error);
  }
);

// Use the API client
async function getUser(userId) {
  try {
    const response = await api.get(`/users/${userId}`);
    return response.data;
  } catch (error) {
    console.error('Failed to get user:', error.userMessage);
    throw error;
  }
```

```
}
```

6.2.6 Response Caching Strategies

```
// In-memory cache with TTL and LRU eviction

class ResponseCache {
  constructor(options = {}) {
    this.maxSize = options.maxSize || 100;
    this.defaultTTL = options.defaultTTL || 300000; // 5 minutes
    this.cache = new Map();
  }

  generateKey(url, options = {}) {
    // Create cache key from URL and relevant options
    const key = {
      url,
      method: options.method || 'GET',
      body: options.body
    };
    return JSON.stringify(key);
  }

  get(url, options) {
    const key = this.generateKey(url, options);
    const cached = this.cache.get(key);

    if (!cached) return null;

    // Check if expired
    if (Date.now() > cached.expiresAt) {
      this.cache.delete(key);
      return null;
    }

    // Move to end (LRU)
    this.cache.delete(key);
    this.cache.set(key, cached);

    return cached.data;
  }

  set(url, options, data, ttl = this.defaultTTL) {
    const key = this.generateKey(url, options);

    // Enforce size limit (LRU eviction)
```

```
    if (this.cache.size >= this.maxSize) {
      // Remove oldest (first) entry
      const firstKey = this.cache.keys().next().value;
      this.cache.delete(firstKey);
    }

    this.cache.set(key, {
      data,
      cachedAt: Date.now(),
      expiresAt: Date.now() + ttl
    });
  }

  invalidate(url, options) {
    const key = this.generateKey(url, options);
    this.cache.delete(key);
  }

  invalidatePattern(pattern) {
    // Invalidate all keys matching pattern
    for (const key of this.cache.keys()) {
      if (key.includes(pattern)) {
        this.cache.delete(key);
      }
    }
  }

  clear() {
    this.cache.clear();
  }

  getStats() {
    return {
      size: this.cache.size,
      maxSize: this.maxSize,
      entries: Array.from(this.cache.keys())
    };
  }
}

// HTTP client with caching
class CachedHttpClient extends HttpClient {
  constructor(baseURL, options = {}) {
    super(baseURL, options);
    this.cache = new ResponseCache({
      maxSize: options.cacheSize || 100,
      defaultTTL: options.cacheTTL || 300000
```

```
    });
  }

  async request(url, options = {}) {
    const cacheOptions = options.cache || {};

    // Check if should use cache (GET requests only by default)
    const useCache = cacheOptions.enabled !== false &&
                     (!options.method || options.method === 'GET');

    if (useCache) {
      // Try cache first
      const cached = this.cache.get(url, options);
      if (cached) {
        console.log('Cache hit:', url);
        return {
          data: cached,
          status: 200,
          statusText: 'OK (Cached)',
          cached: true
        };
      }
    }

    // Make request
    const response = await super.request(url, options);

    // Cache successful GET responses
    if (useCache && response.status === 200) {
      const ttl = cacheOptions.ttl || this.cache.defaultTTL;
      this.cache.set(url, options, response.data, ttl);
    }

    // Invalidate cache on mutations
    if (options.method && ['POST', 'PUT', 'PATCH', 'DELETE'].includes(options.method))
      this.cache.invalidatePattern(url.split('/').slice(0, -1).join('/'));
    }

    return response;
  }
}

// Usage with different caching strategies

const cachedApi = new CachedHttpClient('https://api.example.com', {
  cacheSize: 200,
  cacheTTL: 600000 // 10 minutes default
```

```
});

// 1. Cache with default TTL
const users = await cachedApi.get('/users');

// 2. Cache with custom TTL
const posts = await cachedApi.get('/posts', {
  cache: { ttl: 60000 } // 1 minute
});

// 3. Disable cache for specific request
const freshData = await cachedApi.get('/users', {
  cache: { enabled: false }
});

// 4. Manual cache invalidation
cachedApi.cache.invalidate('/users');
cachedApi.cache.invalidatePattern('/users/');

// Stale-while-revalidate pattern
class StaleWhileRevalidateCache extends ResponseCache {
  async fetch(url, options, fetchFn) {
    const key = this.generateKey(url, options);
    const cached = this.cache.get(key);

    if (cached) {
      // Return stale data immediately
      const staleData = cached.data;

      // Revalidate in background
      this.revalidate(url, options, fetchFn, key);

      return {
        data: staleData,
        stale: true
      };
    }

    // No cache, fetch fresh
    const data = await fetchFn();
    this.set(url, options, data);
    return { data, stale: false };
  }

  async revalidate(url, options, fetchFn, key) {
    try {
      const fresh = await fetchFn();
```

```
      this.cache.set(key, {
        data: fresh,
        cachedAt: Date.now(),
        expiresAt: Date.now() + this.defaultTTL
      });

      // Notify listeners that data was updated
      this.notifyUpdate(key, fresh);
    } catch (error) {
      console.error('Revalidation failed:', error);
    }
  }

  onUpdate(callback) {
    this.updateCallbacks = this.updateCallbacks || [];
    this.updateCallbacks.push(callback);
  }

  notifyUpdate(key, data) {
    if (this.updateCallbacks) {
      this.updateCallbacks.forEach(cb => cb(key, data));
    }
  }
}
```

6.2.7 Rate Limiting and Throttling

```
// Rate limiter for API calls

class RateLimiter {
  constructor(maxRequests, windowMs) {
    this.maxRequests = maxRequests;
    this.windowMs = windowMs;
    this.requests = [];
  }

  async acquire() {
    const now = Date.now();

    // Remove old requests outside window
    this.requests = this.requests.filter(time => now - time < this.windowMs);

    // Check if at limit
    if (this.requests.length >= this.maxRequests) {
      // Calculate wait time
      const oldestRequest = this.requests[0];
```

```
      const waitTime = this.windowMs - (now - oldestRequest);

      console.log(`Rate limit reached. Waiting ${waitTime}ms`);
      await sleep(waitTime);

      return this.acquire(); // Try again
    }

    // Record this request
    this.requests.push(now);
  }
}

// Token bucket algorithm
class TokenBucket {
  constructor(capacity, refillRate) {
    this.capacity = capacity;
    this.tokens = capacity;
    this.refillRate = refillRate; // tokens per second
    this.lastRefill = Date.now();
  }

  async acquire(tokens = 1) {
    this.refill();

    while (this.tokens < tokens) {
      const waitTime = ((tokens - this.tokens) / this.refillRate) * 1000;
      console.log(`Not enough tokens. Waiting ${waitTime}ms`);
      await sleep(waitTime);
      this.refill();
    }

    this.tokens -= tokens;
  }

  refill() {
    const now = Date.now();
    const timePassed = (now - this.lastRefill) / 1000;
    const tokensToAdd = timePassed * this.refillRate;

    this.tokens = Math.min(this.capacity, this.tokens + tokensToAdd);
    this.lastRefill = now;
  }
}

// HTTP client with rate limiting
class RateLimitedHttpClient extends HttpClient {
```

```
  constructor(baseURL, options = {}) {
    super(baseURL, options);
    this.rateLimiter = new TokenBucket(
      options.rateLimit?.capacity || 10,
      options.rateLimit?.refillRate || 1
    );
  }

  async request(url, options = {}) {
    // Acquire token before making request
    await this.rateLimiter.acquire();

    return super.request(url, options);
  }
}

// Usage
const limitedApi = new RateLimitedHttpClient('https://api.example.com', {
  rateLimit: {
    capacity: 10,  // 10 requests
    refillRate: 1  // 1 per second (10 requests per 10 seconds)
  }
});
```

6.2.8 Connection Pooling and Keep-Alive

```
// Reuse connections for better performance

// Enable keep-alive for fetch (modern browsers do this automatically)
const keepAliveAgent = {
  keepalive: true,
  keepAliveMsecs: 30000, // 30 seconds
  maxSockets: 50,
  maxFreeSockets: 10
};

// For Node.js
/*
const http = require('http');
const https = require('https');

const httpAgent = new http.Agent({
  keepAlive: true,
  maxSockets: 50
});
```

```
const httpsAgent = new https.Agent({
  keepAlive: true,
  maxSockets: 50
});

fetch(url, {
  agent: url.startsWith('https:') ? httpsAgent : httpAgent
});
*/

// Connection state monitoring
class ConnectionMonitor {
  constructor() {
    this.quality = 'good';
    this.rtt = 0; // Round trip time
    this.downlink = 0; // Mbps
    this.effectiveType = '4g';

    if ('connection' in navigator) {
      this.updateConnectionInfo();
      navigator.connection.addEventListener('change', () => {
        this.updateConnectionInfo();
      });
    }
  }

  updateConnectionInfo() {
    const conn = navigator.connection;
    this.rtt = conn.rtt || 0;
    this.downlink = conn.downlink || 0;
    this.effectiveType = conn.effectiveType || '4g';

    // Determine quality
    if (this.effectiveType === '4g' && this.rtt < 100) {
      this.quality = 'good';
    } else if (this.effectiveType === '4g' || this.effectiveType === '3g') {
      this.quality = 'moderate';
    } else {
      this.quality = 'poor';
    }

    console.log('Connection quality:', this.quality, {
      rtt: this.rtt,
      downlink: this.downlink,
      effectiveType: this.effectiveType
    });
  }
```

```
  shouldReduceQuality() {
    return this.quality === 'poor' || this.quality === 'moderate';
  }

  getOptimalTimeout() {
    return {
      good: 5000,
      moderate: 10000,
      poor: 15000
    }[this.quality];
  }
}

// Adaptive request strategy based on connection
class AdaptiveHttpClient extends HttpClient {
  constructor(baseURL, options = {}) {
    super(baseURL, options);
    this.connectionMonitor = new ConnectionMonitor();
  }

  async request(url, options = {}) {
    // Adjust timeout based on connection quality
    const adaptiveOptions = {
      ...options,
      timeout: options.timeout || this.connectionMonitor.getOptimalTimeout()
    };

    // Reduce payload quality if poor connection
    if (this.connectionMonitor.shouldReduceQuality()) {
      adaptiveOptions.headers = {
        ...adaptiveOptions.headers,
        'Accept': 'application/json; q=low',
        'X-Quality': 'reduced'
      };
    }

    return super.request(url, adaptiveOptions);
  }
}
```

6.3 CORS Deep Dive

```
// Simple requests (no preflight)
// - Methods: GET, HEAD, POST
// - Safe headers only
```

```
// - Content-Type: application/x-www-form-urlencoded, multipart/form-data, text/plain

fetch('https://api.example.com/data', {
  method: 'GET'
});
// No preflight, server responds with:
// Access-Control-Allow-Origin: *
// or
// Access-Control-Allow-Origin: https://mysite.com

// Preflighted requests
// - Other methods (PUT, DELETE, PATCH)
// - Custom headers
// - Content-Type: application/json

fetch('https://api.example.com/data', {
  method: 'PUT',
  headers: {
    'Content-Type': 'application/json',
    'X-Custom-Header': 'value'
  },
  body: JSON.stringify({ data: 'value' })
});

// Browser sends preflight OPTIONS request:
// OPTIONS /data
// Origin: https://mysite.com
// Access-Control-Request-Method: PUT
// Access-Control-Request-Headers: content-type, x-custom-header

// Server responds:
// Access-Control-Allow-Origin: https://mysite.com
// Access-Control-Allow-Methods: PUT, POST, DELETE
// Access-Control-Allow-Headers: content-type, x-custom-header
// Access-Control-Max-Age: 86400  // Cache preflight for 24 hours

// Credentials (cookies, auth headers)
fetch('https://api.example.com/data', {
  credentials: 'include'  // Send cookies
});

// Server must respond with:
// Access-Control-Allow-Origin: https://mysite.com  // Can't be *
// Access-Control-Allow-Credentials: true

// Server-side CORS setup (Express)
/*
```

```
app.use((req, res, next) => {
  res.header('Access-Control-Allow-Origin', req.headers.origin);
  res.header('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, OPTIONS');
  res.header('Access-Control-Allow-Headers', 'Content-Type, Authorization');
  res.header('Access-Control-Allow-Credentials', 'true');
  res.header('Access-Control-Max-Age', '86400');

  if (req.method === 'OPTIONS') {
    return res.sendStatus(204);
  }

  next();
});
*/
```

6.4 WebSockets for Real-Time Communication

```
// Create WebSocket connection
const ws = new WebSocket('wss://example.com/socket');

// Connection opened
ws.addEventListener('open', (event) => {
  console.log('Connected to WebSocket');
  ws.send('Hello Server!');
});

// Listen for messages
ws.addEventListener('message', (event) => {
  console.log('Message from server:', event.data);

  // Parse JSON if applicable
  try {
    const data = JSON.parse(event.data);
    handleMessage(data);
  } catch (error) {
    console.log('Raw message:', event.data);
  }
});

// Connection closed
ws.addEventListener('close', (event) => {
  console.log('WebSocket closed:', event.code, event.reason);

  // Reconnect logic
  setTimeout(() => {
    console.log('Reconnecting...');
```

```
    createWebSocket();
  }, 1000);
});

// Connection error
ws.addEventListener('error', (error) => {
  console.error('WebSocket error:', error);
});

// Send different data types
// String
ws.send('Hello');

// JSON
ws.send(JSON.stringify({ type: 'message', content: 'Hello' }));

// Binary data
const buffer = new ArrayBuffer(8);
ws.send(buffer);

const blob = new Blob(['Hello'], { type: 'text/plain' });
ws.send(blob);

// Close connection
ws.close(1000, 'Normal closure');

// WebSocket wrapper with reconnection
class ReconnectingWebSocket {
  constructor(url, options = {}) {
    this.url = url;
    this.reconnectDelay = options.reconnectDelay || 1000;
    this.maxReconnectDelay = options.maxReconnectDelay || 30000;
    this.reconnectAttempts = 0;
    this.listeners = new Map();

    this.connect();
  }

  connect() {
    this.ws = new WebSocket(this.url);

    this.ws.addEventListener('open', (event) => {
      console.log('WebSocket connected');
      this.reconnectAttempts = 0;
      this.reconnectDelay = 1000;
      this.emit('open', event);
    });
```

```
    this.ws.addEventListener('message', (event) => {
      this.emit('message', event);
    });

    this.ws.addEventListener('close', (event) => {
      console.log('WebSocket closed');
      this.emit('close', event);
      this.reconnect();
    });

    this.ws.addEventListener('error', (error) => {
      console.error('WebSocket error:', error);
      this.emit('error', error);
    });
  }

  reconnect() {
    this.reconnectAttempts++;

    const delay = Math.min(
      this.reconnectDelay * Math.pow(2, this.reconnectAttempts),
      this.maxReconnectDelay
    );

    console.log(`Reconnecting in ${delay}ms (attempt ${this.reconnectAttempts})`);

    setTimeout(() => {
      console.log('Attempting to reconnect...');
      this.connect();
    }, delay);
  }

  send(data) {
    if (this.ws.readyState === WebSocket.OPEN) {
      this.ws.send(data);
    } else {
      console.warn('WebSocket not open, queuing message');
      // Could implement message queue here
    }
  }

  on(event, callback) {
    if (!this.listeners.has(event)) {
      this.listeners.set(event, []);
    }
    this.listeners.get(event).push(callback);
  }
```

```
  emit(event, data) {
    if (this.listeners.has(event)) {
      this.listeners.get(event).forEach(callback => callback(data));
    }
  }

  close() {
    this.reconnectAttempts = Infinity;  // Prevent reconnection
    this.ws.close();
  }
}

// Usage
const socket = new ReconnectingWebSocket('wss://example.com/socket');

socket.on('open', () => {
  console.log('Connected!');
});

socket.on('message', (event) => {
  console.log('Received:', event.data);
});

socket.send('Hello Server!');
```

6.5 IndexedDB for Client-Side Storage

```
// Open database
const request = indexedDB.open('MyDatabase', 1);

// Database version upgrade (schema changes)
request.onupgradeneeded = (event) => {
  const db = event.target.result;

  // Create object store
  const objectStore = db.createObjectStore('users', {
    keyPath: 'id',
    autoIncrement: true
  });

  // Create indexes
  objectStore.createIndex('email', 'email', { unique: true });
  objectStore.createIndex('name', 'name', { unique: false });
  objectStore.createIndex('age', 'age', { unique: false });
};
```

```
// Success
request.onsuccess = (event) => {
  const db = event.target.result;
  console.log('Database opened successfully');

  // Add data
  addUser(db, { name: 'John', email: 'john@example.com', age: 30 });
};

// Error
request.onerror = (event) => {
  console.error('Database error:', event.target.error);
};

// Add data
function addUser(db, user) {
  const transaction = db.transaction(['users'], 'readwrite');
  const objectStore = transaction.objectStore('users');
  const request = objectStore.add(user);

  request.onsuccess = () => {
    console.log('User added:', user);
  };

  request.onerror = () => {
    console.error('Error adding user:', request.error);
  };
}

// Get data by key
function getUserById(db, id) {
  return new Promise((resolve, reject) => {
    const transaction = db.transaction(['users'], 'readonly');
    const objectStore = transaction.objectStore('users');
    const request = objectStore.get(id);

    request.onsuccess = () => {
      resolve(request.result);
    };

    request.onerror = () => {
      reject(request.error);
    };
  });
}

// Get data by index
```

```
function getUserByEmail(db, email) {
  return new Promise((resolve, reject) => {
    const transaction = db.transaction(['users'], 'readonly');
    const objectStore = transaction.objectStore('users');
    const index = objectStore.index('email');
    const request = index.get(email);

    request.onsuccess = () => {
      resolve(request.result);
    };

    request.onerror = () => {
      reject(request.error);
    };
  });
}

// Get all data
function getAllUsers(db) {
  return new Promise((resolve, reject) => {
    const transaction = db.transaction(['users'], 'readonly');
    const objectStore = transaction.objectStore('users');
    const request = objectStore.getAll();

    request.onsuccess = () => {
      resolve(request.result);
    };

    request.onerror = () => {
      reject(request.error);
    };
  });
}

// Cursor for iteration
function iterateUsers(db) {
  const transaction = db.transaction(['users'], 'readonly');
  const objectStore = transaction.objectStore('users');
  const request = objectStore.openCursor();

  request.onsuccess = (event) => {
    const cursor = event.target.result;

    if (cursor) {
      console.log('User:', cursor.value);
      cursor.continue();  // Move to next
    } else {
```

```
      console.log('No more users');
    }
  };
}

// Update data
function updateUser(db, user) {
  return new Promise((resolve, reject) => {
    const transaction = db.transaction(['users'], 'readwrite');
    const objectStore = transaction.objectStore('users');
    const request = objectStore.put(user);  // put updates or adds

    request.onsuccess = () => {
      resolve(request.result);
    };

    request.onerror = () => {
      reject(request.error);
    };
  });
}

// Delete data
function deleteUser(db, id) {
  return new Promise((resolve, reject) => {
    const transaction = db.transaction(['users'], 'readwrite');
    const objectStore = transaction.objectStore('users');
    const request = objectStore.delete(id);

    request.onsuccess = () => {
      resolve();
    };

    request.onerror = () => {
      reject(request.error);
    };
  });
}

// Range queries
function getUsersInAgeRange(db, minAge, maxAge) {
  return new Promise((resolve, reject) => {
    const transaction = db.transaction(['users'], 'readonly');
    const objectStore = transaction.objectStore('users');
    const index = objectStore.index('age');

    const range = IDBKeyRange.bound(minAge, maxAge);
```

```
    const request = index.getAll(range);

    request.onsuccess = () => {
      resolve(request.result);
    };

    request.onerror = () => {
      reject(request.error);
    };
  });
}

// IDBKeyRange options:
// IDBKeyRange.only(value)              // Exact match
// IDBKeyRange.lowerBound(value, open)  // >= value (or > if open=true)
// IDBKeyRange.upperBound(value, open)  // <= value (or < if open=true)
// IDBKeyRange.bound(lower, upper, lowerOpen, upperOpen)

// IndexedDB wrapper class
class IndexedDBWrapper {
  constructor(dbName, version = 1) {
    this.dbName = dbName;
    this.version = version;
    this.db = null;
  }

  async open(onUpgrade) {
    return new Promise((resolve, reject) => {
      const request = indexedDB.open(this.dbName, this.version);

      request.onupgradeneeded = (event) => {
        this.db = event.target.result;
        if (onUpgrade) {
          onUpgrade(this.db, event);
        }
      };

      request.onsuccess = (event) => {
        this.db = event.target.result;
        resolve(this.db);
      };

      request.onerror = () => {
        reject(request.error);
      };
    });
  }
```

```
  async add(storeName, data) {
    const transaction = this.db.transaction([storeName], 'readwrite');
    const store = transaction.objectStore(storeName);

    return new Promise((resolve, reject) => {
      const request = store.add(data);
      request.onsuccess = () => resolve(request.result);
      request.onerror = () => reject(request.error);
    });
  }

  async get(storeName, key) {
    const transaction = this.db.transaction([storeName], 'readonly');
    const store = transaction.objectStore(storeName);

    return new Promise((resolve, reject) => {
      const request = store.get(key);
      request.onsuccess = () => resolve(request.result);
      request.onerror = () => reject(request.error);
    });
  }

  async getAll(storeName) {
    const transaction = this.db.transaction([storeName], 'readonly');
    const store = transaction.objectStore(storeName);

    return new Promise((resolve, reject) => {
      const request = store.getAll();
      request.onsuccess = () => resolve(request.result);
      request.onerror = () => reject(request.error);
    });
  }

  async update(storeName, data) {
    const transaction = this.db.transaction([storeName], 'readwrite');
    const store = transaction.objectStore(storeName);

    return new Promise((resolve, reject) => {
      const request = store.put(data);
      request.onsuccess = () => resolve(request.result);
      request.onerror = () => reject(request.error);
    });
  }

  async delete(storeName, key) {
    const transaction = this.db.transaction([storeName], 'readwrite');
    const store = transaction.objectStore(storeName);
```

```
    return new Promise((resolve, reject) => {
      const request = store.delete(key);
      request.onsuccess = () => resolve();
      request.onerror = () => reject(request.error);
    });
  }
}

// Usage
const db = new IndexedDBWrapper('MyApp', 1);

await db.open((database, event) => {
  // Schema upgrade
  const store = database.createObjectStore('todos', {
    keyPath: 'id',
    autoIncrement: true
  });
  store.createIndex('completed', 'completed', { unique: false });
});

await db.add('todos', { text: 'Learn IndexedDB', completed: false });
const todos = await db.getAll('todos');
console.log(todos);
```

6.6 Service Workers and Caching

```
// Register service worker
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js')
    .then(registration => {
      console.log('Service Worker registered:', registration);

      // Check for updates
      registration.update();
    })
    .catch(error => {
      console.error('Service Worker registration failed:', error);
    });
}

// sw.js - Service Worker file
const CACHE_NAME = 'my-app-v1';
const urlsToCache = [
  '/',
  '/styles.css',
```

```
  '/script.js',
  '/images/logo.png'
];

// Install event - cache assets
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(cache => {
        console.log('Opened cache');
        return cache.addAll(urlsToCache);
      })
  );

  // Force activation
  self.skipWaiting();
});

// Activate event - clean old caches
self.addEventListener('activate', (event) => {
  event.waitUntil(
    caches.keys()
      .then(cacheNames => {
        return Promise.all(
          cacheNames.map(cacheName => {
            if (cacheName !== CACHE_NAME) {
              console.log('Deleting old cache:', cacheName);
              return caches.delete(cacheName);
            }
          })
        );
      })
  );

  // Take control immediately
  self.clients.claim();
});

// Fetch event - serve from cache or network
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request)
      .then(response => {
        // Cache hit - return response
        if (response) {
          return response;
        }
```

```
        // Clone request for fetch
        const fetchRequest = event.request.clone();

        return fetch(fetchRequest)
          .then(response => {
            // Check valid response
            if (!response || response.status !== 200 || response.type !== 'basic') {
              return response;
            }

            // Clone response for cache
            const responseToCache = response.clone();

            caches.open(CACHE_NAME)
              .then(cache => {
                cache.put(event.request, responseToCache);
              });

            return response;
          });
      })
  );
});

// Caching strategies

// 1. Cache First (good for static assets)
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request)
      .then(cached => cached || fetch(event.request))
  );
});

// 2. Network First (good for API calls)
self.addEventListener('fetch', (event) => {
  event.respondWith(
    fetch(event.request)
      .then(response => {
        const clone = response.clone();
        caches.open(CACHE_NAME)
          .then(cache => cache.put(event.request, clone));
        return response;
      })
      .catch(() => caches.match(event.request))
  );
});
```

```
// 3. Stale While Revalidate
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.open(CACHE_NAME)
      .then(cache => {
        return cache.match(event.request)
          .then(cached => {
            const fetched = fetch(event.request)
              .then(response => {
                cache.put(event.request, response.clone());
                return response;
              });

            return cached || fetched;
          });
      })
  );
});

// 4. Cache Only
self.addEventListener('fetch', (event) => {
  event.respondWith(caches.match(event.request));
});

// 5. Network Only
self.addEventListener('fetch', (event) => {
  event.respondWith(fetch(event.request));
});

// Background Sync
self.addEventListener('sync', (event) => {
  if (event.tag === 'sync-messages') {
    event.waitUntil(syncMessages());
  }
});

async function syncMessages() {
  // Get pending messages from IndexedDB
  const messages = await getPendingMessages();

  for (const message of messages) {
    try {
      await fetch('/api/messages', {
        method: 'POST',
        body: JSON.stringify(message)
      });
```

```
      await markMessageSent(message.id);
    } catch (error) {
      console.error('Failed to sync message:', error);
    }
  }
}

// Register background sync from main thread
navigator.serviceWorker.ready.then(registration => {
  registration.sync.register('sync-messages');
});

// Push Notifications
self.addEventListener('push', (event) => {
  const data = event.data.json();

  const options = {
    body: data.body,
    icon: '/images/icon.png',
    badge: '/images/badge.png',
    data: { url: data.url }
  };

  event.waitUntil(
    self.registration.showNotification(data.title, options)
  );
});

self.addEventListener('notificationclick', (event) => {
  event.notification.close();

  event.waitUntil(
    clients.openWindow(event.notification.data.url)
  );
});
```

Chapter 7

Chapter 7: Microfrontends - Architecture and Patterns

Microfrontends extend the microservices concept to the frontend, allowing teams to work independently on different parts of a web application.

7.1 Core Concepts

Microfrontends solve several problems:

1. Team Autonomy: Different teams can own different features end-to-end
2. Independent Deployment: Deploy parts of the app independently
3. Technology Agnostic: Different parts can use different frameworks
4. Incremental Upgrades: Modernize legacy apps piece by piece
5. Smaller Codebases: Easier to understand and maintain

Trade-offs:

1. Increased Complexity: More moving parts to coordinate
2. Performance Overhead: Multiple frameworks, duplicate code
3. Consistency Challenges: UI/UX consistency across teams
4. Shared State: Communication between microfrontends is complex
5. Tooling: Build pipelines become more complex

7.2 Implementation Patterns

7.2.1 Pattern 1: Build-Time Integration (Not True Microfrontends)

```
// Package.json
{
  "dependencies": {
    "team-a-feature": "^1.0.0",
    "team-b-feature": "^2.0.0"
  }
```

```
}

// App.js
import TeamAFeature from 'team-a-feature';
import TeamBFeature from 'team-b-feature';

function App() {
  return (
    <div>
      <TeamAFeature />
      <TeamBFeature />
    </div>
  );
}

// Pros:
// - Simple to implement
// - No runtime overhead
// - Type safety

// Cons:
// - Not independent deployment
// - Tight coupling
// - Must redeploy everything for changes
```

7.2.2 Pattern 2: Run-Time Integration via IFrames

```
<!DOCTYPE html>
<html>
<head>
  <title>Container App</title>
</head>
<body>
  <nav id="nav-container"></nav>

  <iframe
    id="feature-a"
    src="https://team-a.example.com/feature"
    style="width: 100%; height: 600px; border: none;"
  ></iframe>

  <iframe
    id="feature-b"
    src="https://team-b.example.com/feature"
    style="width: 100%; height: 600px; border: none;"
  ></iframe>
```

```
  <script>
    // Communication via postMessage
    const featureA = document.getElementById('feature-a');

    featureA.contentWindow.postMessage({
      type: 'USER_LOGGED_IN',
      user: { id: 123, name: 'John' }
    }, 'https://team-a.example.com');

    window.addEventListener('message', (event) => {
      if (event.origin !== 'https://team-a.example.com') return;

      console.log('Message from Feature A:', event.data);
    });
  </script>
</body>
</html>
```

Pros: - Complete isolation - True independent deployment - No conflicts (CSS, JS globals)

Cons: - Performance overhead - Difficult routing - Poor UX (separate contexts) - Complex communication

7.2.3 Pattern 3: Run-Time Integration via JavaScript (Module Federation)

Webpack Module Federation (most popular approach):

```
// Container app - webpack.config.js
const ModuleFederationPlugin = require('webpack/lib/container/ModuleFederationPlugin');

module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: 'container',
      remotes: {
        teamA: 'teamA@https://team-a.example.com/remoteEntry.js',
        teamB: 'teamB@https://team-b.example.com/remoteEntry.js'
      },
      shared: {
        react: { singleton: true, requiredVersion: '^18.0.0' },
        'react-dom': { singleton: true, requiredVersion: '^18.0.0' }
      }
    })
  ]
};
```

```
// Container app - App.js
import React, { lazy, Suspense } from 'react';

const TeamAFeature = lazy(() => import('teamA/Feature'));
const TeamBFeature = lazy(() => import('teamB/Feature'));

function App() {
  return (
    <div>
      <nav>Global Navigation</nav>

      <Suspense fallback={<div>Loading Team A...</div>}>
        <TeamAFeature />
      </Suspense>

      <Suspense fallback={<div>Loading Team B...</div>}>
        <TeamBFeature />
      </Suspense>
    </div>
  );
}

export default App;

// Team A - webpack.config.js
module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: 'teamA',
      filename: 'remoteEntry.js',
      exposes: {
        './Feature': './src/Feature'
      },
      shared: {
        react: { singleton: true },
        'react-dom': { singleton: true }
      }
    })
  ]
};

// Team A - src/Feature.js
import React from 'react';

export default function Feature() {
  return <div>Team A Feature</div>;
}
```

7.2.4 Pattern 4: Run-Time Integration via Web Components

```
// Team A - feature.js
class TeamAFeature extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
  }

  connectedCallback() {
    this.render();
  }

  static get observedAttributes() {
    return ['user'];
  }

  attributeChangedCallback(name, oldValue, newValue) {
    if (name === 'user') {
      this.render();
    }
  }

  render() {
    const user = JSON.parse(this.getAttribute('user') || '{}');

    this.shadowRoot.innerHTML = `
      <style>
        .container {
          padding: 20px;
          background: #f0f0f0;
        }
      </style>
      <div class="container">
        <h2>Team A Feature</h2>
        <p>User: ${user.name || 'Guest'}</p>
      </div>
    `;
  }

  // Public API for communication
  updateUser(user) {
    this.setAttribute('user', JSON.stringify(user));
  }
}

customElements.define('team-a-feature', TeamAFeature);
```

```
// Container app
<!DOCTYPE html>
<html>
<head>
  <script src="https://team-a.example.com/feature.js"></script>
  <script src="https://team-b.example.com/feature.js"></script>
</head>
<body>
  <team-a-feature id="feature-a" user='{"name":"John"}'></team-a-feature>
  <team-b-feature id="feature-b"></team-b-feature>

  <script>
    // Update feature
    const featureA = document.getElementById('feature-a');
    featureA.updateUser({ name: 'Jane' });

    // Listen to events
    featureA.addEventListener('user-clicked', (event) => {
      console.log('User clicked:', event.detail);
    });
  </script>
</body>
</html>
```

Pros: - Framework agnostic - Strong encapsulation (Shadow DOM) - Browser native

Cons: - Limited browser support for older features - Complex state management - SSR challenges

7.3 Communication Strategies

7.3.1 Custom Events

```
// Microfrontend A - emit event
const event = new CustomEvent('user-selected', {
  detail: { userId: 123, name: 'John' },
  bubbles: true,
  composed: true  // Cross shadow DOM boundary
});
this.dispatchEvent(event);

// Container - listen to event
document.addEventListener('user-selected', (event) => {
  console.log('User selected:', event.detail);

  // Notify other microfrontends
  const featureB = document.querySelector('feature-b');
```

```
  featureB.updateUser(event.detail);
});
```

7.3.2 Event Bus

```
// event-bus.js
class EventBus {
  constructor() {
    this.events = {};
  }

  on(event, callback) {
    if (!this.events[event]) {
      this.events[event] = [];
    }
    this.events[event].push(callback);
  }

  off(event, callback) {
    if (!this.events[event]) return;

    this.events[event] = this.events[event].filter(cb => cb !== callback);
  }

  emit(event, data) {
    if (!this.events[event]) return;

    this.events[event].forEach(callback => callback(data));
  }
}

export const eventBus = new EventBus();

// Microfrontend A
import { eventBus } from './event-bus';

eventBus.emit('user-logged-in', { userId: 123 });

// Microfrontend B
import { eventBus } from './event-bus';

eventBus.on('user-logged-in', (user) => {
  console.log('User logged in:', user);
  this.setState({ user });
});
```

7.3.3 Shared State

```
// shared-state.js
class SharedState {
  constructor() {
    this.state = {};
    this.subscribers = {};
  }

  subscribe(key, callback) {
    if (!this.subscribers[key]) {
      this.subscribers[key] = [];
    }
    this.subscribers[key].push(callback);

    // Return unsubscribe function
    return () => {
      this.subscribers[key] = this.subscribers[key].filter(cb => cb !== callback);
    };
  }

  setState(key, value) {
    this.state[key] = value;

    if (this.subscribers[key]) {
      this.subscribers[key].forEach(callback => callback(value));
    }
  }

  getState(key) {
    return this.state[key];
  }
}

export const sharedState = new SharedState();

// Microfrontend A - write state
import { sharedState } from './shared-state';

sharedState.setState('user', { id: 123, name: 'John' });

// Microfrontend B - read state
import { sharedState } from './shared-state';

const unsubscribe = sharedState.subscribe('user', (user) => {
  console.log('User changed:', user);
  this.updateUser(user);
```

```
});

// Clean up
unsubscribe();
```

7.4 Routing in Microfrontends

```
// Container app router
class MicroFrontendRouter {
  constructor() {
    this.routes = new Map();
    this.currentMicroFrontend = null;

    window.addEventListener('popstate', () => {
      this.handleRouteChange();
    });
  }

  register(path, loadMicroFrontend) {
    this.routes.set(path, loadMicroFrontend);
  }

  navigate(path) {
    window.history.pushState({}, '', path);
    this.handleRouteChange();
  }

  handleRouteChange() {
    const path = window.location.pathname;

    // Find matching route
    for (const [routePath, loadMicroFrontend] of this.routes) {
      if (path.startsWith(routePath)) {
        this.loadMicroFrontend(loadMicroFrontend, path);
        return;
      }
    }

    // 404
    this.render404();
  }

  async loadMicroFrontend(loadMicroFrontend, path) {
    // Unmount current
    if (this.currentMicroFrontend && this.currentMicroFrontend.unmount) {
      this.currentMicroFrontend.unmount();
```

```
    }

    // Load and mount new
    this.currentMicroFrontend = await loadMicroFrontend();
    this.currentMicroFrontend.mount(document.getElementById('app'), path);
  }

  render404() {
    document.getElementById('app').innerHTML = '<h1>404 Not Found</h1>';
  }
}

// Usage
const router = new MicroFrontendRouter();

router.register('/products', () => import('./microfrontends/products'));
router.register('/cart', () => import('./microfrontends/cart'));
router.register('/checkout', () => import('./microfrontends/checkout'));

router.handleRouteChange();

// Microfrontend lifecycle
export default {
  async mount(container, path) {
    // Initialize microfrontend
    const root = ReactDOM.createRoot(container);
    root.render(<App initialPath={path} />);
    this.root = root;
  },

  unmount() {
    // Clean up
    if (this.root) {
      this.root.unmount();
    }
  }
};
```

7.5 Single-SPA Framework

Single-SPA is a popular framework for building microfrontends:

```
// Container app - index.js
import { registerApplication, start } from 'single-spa';

// Register microfrontends
registerApplication({
```

```
  name: '@org/navbar',
  app: () => System.import('@org/navbar'),
  activeWhen: () => true  // Always active
});

registerApplication({
  name: '@org/products',
  app: () => System.import('@org/products'),
  activeWhen: '/products'
});

registerApplication({
  name: '@org/cart',
  app: () => System.import('@org/cart'),
  activeWhen: '/cart'
});

start();

// Microfrontend - products/src/root.component.js
import React from 'react';
import ReactDOM from 'react-dom';
import singleSpaReact from 'single-spa-react';
import App from './App';

const lifecycles = singleSpaReact({
  React,
  ReactDOM,
  rootComponent: App
});

export const { bootstrap, mount, unmount } = lifecycles;

// Microfrontend - products/webpack.config.js
module.exports = {
  output: {
    filename: 'products.js',
    libraryTarget: 'system'
  },
  externals: ['react', 'react-dom'],
  plugins: [
    new ModuleFederationPlugin({
      name: 'products',
      filename: 'remoteEntry.js',
      exposes: {
        './Products': './src/root.component'
      },
```

```
      shared: {
        react: { singleton: true },
        'react-dom': { singleton: true }
      }
    })
  ]
};
```

7.6 Shared Dependencies and Version Management

```
// Container webpack.config.js
new ModuleFederationPlugin({
  name: 'container',
  remotes: {
    teamA: 'teamA@https://team-a.example.com/remoteEntry.js'
  },
  shared: {
    // Singleton - only one version loaded
    react: {
      singleton: true,
      requiredVersion: '^18.0.0',
      strictVersion: false  // Allow compatible versions
    },

    // Eager - load immediately, don't code split
    lodash: {
      eager: true,
      requiredVersion: '^4.0.0'
    },

    // Share scope - group dependencies
    '@company/ui-library': {
      singleton: true,
      shareScope: 'company'
    }
  }
});

// Version conflict resolution
// If teamA needs React 18.1 and teamB needs React 18.2:
// - With singleton: true, only one version loads
// - With strictVersion: false, higher version loads
// - With strictVersion: true, throws error

// Fallback behavior
shared: {
```

```
  react: {
    singleton: true,
    requiredVersion: '^18.0.0',
    import: 'react',  // Default import
    shareKey: 'react',  // Share under this key
    shareScope: 'default',
    packageName: 'react',
    version: '18.2.0',  // Current version
    fallback: false  // Throw error if not available
  }
}
```

7.7 CSS Isolation Strategies

7.7.1 CSS Modules

```
// feature.module.css
.container {
  padding: 20px;
}

.title {
  color: blue;
}

// Component.js
import styles from './feature.module.css';

function Feature() {
  return (
    <div className={styles.container}>
      <h1 className={styles.title}>Feature</h1>
    </div>
  );
}
```

7.7.2 CSS-in-JS

```
import styled from 'styled-components';

const Container = styled.div`
  padding: 20px;
`;

const Title = styled.h1`
```

```
  color: blue;
`;

function Feature() {
  return (
    <Container>
      <Title>Feature</Title>
    </Container>
  );
}
```

7.7.3 Shadow DOM

```
class Feature extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
  }

  connectedCallback() {
    this.shadowRoot.innerHTML = `
      <style>
        .container { padding: 20px; }
        .title { color: blue; }
      </style>
      <div class="container">
        <h1 class="title">Feature</h1>
      </div>
    `;
  }
}
```

7.7.4 BEM or Namespacing

```
/* team-a-feature.css */
.team-a-feature__container {
  padding: 20px;
}

.team-a-feature__title {
  color: blue;
}
```

7.8 Performance Optimization

```
// Lazy loading microfrontends
const ProductsMicroFrontend = lazy(() => import('./microfrontends/products'));

// Preload microfrontends on hover
function NavLink({ to, children }) {
  const handleMouseEnter = () => {
    import('./microfrontends/products');  // Preload
  };

  return (
    <a href={to} onMouseEnter={handleMouseEnter}>
      {children}
    </a>
  );
}

// Cache microfrontends
const cache = new Map();

async function loadMicroFrontend(name, url) {
  if (cache.has(name)) {
    return cache.get(name);
  }

  const module = await import(url);
  cache.set(name, module);
  return module;
}

// Share common dependencies
new ModuleFederationPlugin({
  shared: {
    react: { singleton: true, eager: true },
    'react-dom': { singleton: true, eager: true },
    '@company/ui-library': { singleton: true, eager: true }
  }
});
```

Chapter 8

Chapter 8: Advanced DOM Manipulation

The Document Object Model (DOM) is the browser's representation of HTML. Understanding DOM manipulation is crucial for performance and interactivity.

8.1 DOM Tree Structure

```
Document
  └─ html (Element)
      ├─ head (Element)
      │   └─ title (Element)
      │       └─ "Page Title" (Text)
      └─ body (Element)
          ├─ div (Element)
          │   ├─ class="container" (Attribute)
          │   └─ p (Element)
          │       └─ "Hello" (Text)
          └─ "  " (Text - whitespace)
```

8.2 Node Types

```javascript
// Node types
console.log(Node.ELEMENT_NODE);                // 1
console.log(Node.TEXT_NODE);                   // 3
console.log(Node.COMMENT_NODE);                // 8
console.log(Node.DOCUMENT_NODE);               // 9
console.log(Node.DOCUMENT_FRAGMENT_NODE);      // 11

// Check node type
const element = document.querySelector('div');
console.log(element.nodeType);  // 1
```

```
console.log(element.nodeName);  // DIV

const textNode = element.firstChild;
console.log(textNode.nodeType);  // 3
console.log(textNode.nodeName);  // #text
```

8.3 Selecting Elements

```
// By ID (fastest)
const element = document.getElementById('myId');

// By class name (returns HTMLCollection - live)
const elements = document.getElementsByClassName('myClass');

// By tag name (returns HTMLCollection - live)
const divs = document.getElementsByTagName('div');

// Query selector (returns first match)
const element = document.querySelector('.myClass');
const element = document.querySelector('#myId');
const element = document.querySelector('div.myClass[data-id="123"]');

// Query selector all (returns NodeList - static)
const elements = document.querySelectorAll('.myClass');

// Difference: HTMLCollection vs NodeList
const htmlCollection = document.getElementsByClassName('item');
// HTMLCollection is live - updates automatically
document.body.innerHTML += '<div class="item"></div>';
console.log(htmlCollection.length);  // Increased!

const nodeList = document.querySelectorAll('.item');
// NodeList from querySelectorAll is static
document.body.innerHTML += '<div class="item"></div>';
console.log(nodeList.length);  // Same!

// NodeList from childNodes is live
const liveNodeList = document.body.childNodes;
document.body.appendChild(document.createElement('div'));
console.log(liveNodeList.length);  // Increased!

// Convert to array
const array = Array.from(elements);
const array = [...elements];
```

8.4 Creating and Modifying Elements

```
// Create element
const div = document.createElement('div');
div.id = 'myDiv';
div.className = 'container';
div.textContent = 'Hello World';

// Set attributes
div.setAttribute('data-id', '123');
div.setAttribute('aria-label', 'My Div');

// Get attributes
const id = div.getAttribute('data-id');
const hasAttr = div.hasAttribute('data-id');

// Remove attributes
div.removeAttribute('data-id');

// Dataset API (for data-* attributes)
div.dataset.id = '123';  // Sets data-id
div.dataset.userId = '456';  // Sets data-user-id (camelCase to kebab-case)
console.log(div.dataset.id);  // '123'

// Create text node
const textNode = document.createTextNode('Hello');
div.appendChild(textNode);

// Create comment
const comment = document.createComment('This is a comment');
document.body.appendChild(comment);

// Clone element
const clone = div.cloneNode(false);  // Shallow clone (no children)
const deepClone = div.cloneNode(true);  // Deep clone (with children)
```

8.5 Inserting Elements

```
const parent = document.querySelector('.parent');
const newElement = document.createElement('div');

// Append to end
parent.appendChild(newElement);
parent.append(newElement);  // Can also append text
parent.append('Text', newElement, 'More text');
```

```javascript
// Prepend to beginning
parent.prepend(newElement);

// Insert before
const referenceNode = document.querySelector('.reference');
parent.insertBefore(newElement, referenceNode);

// Insert adjacent
referenceNode.insertAdjacentElement('beforebegin', newElement);  // Before element
referenceNode.insertAdjacentElement('afterbegin', newElement);   // First child
referenceNode.insertAdjacentElement('beforeend', newElement);    // Last child
referenceNode.insertAdjacentElement('afterend', newElement);     // After element

// Insert HTML
element.insertAdjacentHTML('beforeend', '<div>Hello</div>');

// Replace
const oldElement = document.querySelector('.old');
oldElement.replaceWith(newElement);

// Remove
element.remove();
parent.removeChild(element);
```

8.6 Traversing the DOM

```javascript
const element = document.querySelector('.myClass');

// Parent
const parent = element.parentElement;
const parentNode = element.parentNode;  // Same as parentElement for elements

// Children
const children = element.children;  // HTMLCollection (only elements)
const childNodes = element.childNodes;  // NodeList (all nodes including text)
const firstChild = element.firstElementChild;  // First element child
const lastChild = element.lastElementChild;  // Last element child

// Siblings
const nextSibling = element.nextElementSibling;
const previousSibling = element.previousElementSibling;

// Closest ancestor matching selector
const ancestor = element.closest('.ancestor');

// Matches selector
```

```
const matches = element.matches('.myClass');  // true/false

// Find all ancestors
function getAncestors(element) {
  const ancestors = [];
  let current = element.parentElement;

  while (current) {
    ancestors.push(current);
    current = current.parentElement;
  }

  return ancestors;
}
```

8.7 Class Manipulation

```
const element = document.querySelector('.myClass');

// classList API
element.classList.add('newClass');
element.classList.add('class1', 'class2', 'class3');
element.classList.remove('oldClass');
element.classList.toggle('active');  // Add if not present, remove if present
element.classList.toggle('active', true);  // Force add
element.classList.toggle('active', false);  // Force remove
element.classList.contains('myClass');  // true/false
element.classList.replace('oldClass', 'newClass');

// className (space-separated string)
element.className = 'class1 class2 class3';
console.log(element.className);  // 'class1 class2 class3'
```

8.8 Style Manipulation

```
const element = document.querySelector('.myClass');

// Inline styles
element.style.color = 'red';
element.style.backgroundColor = 'blue';  // camelCase
element.style.fontSize = '20px';

// CSS text
element.style.cssText = 'color: red; background-color: blue; font-size: 20px;';
```

```
// Get computed style (read-only)
const computed = window.getComputedStyle(element);
console.log(computed.color);  // 'rgb(255, 0, 0)'
console.log(computed.fontSize);  // '20px'
console.log(computed.getPropertyValue('font-size'));  // '20px'

// CSS custom properties (CSS variables)
document.documentElement.style.setProperty('--main-color', 'blue');
const mainColor = getComputedStyle(document.documentElement)
  .getPropertyValue('--main-color');
```

8.9 DocumentFragment for Performance

```
// BAD: Multiple reflows
for (let i = 0; i < 1000; i++) {
  const div = document.createElement('div');
  div.textContent = `Item ${i}`;
  document.body.appendChild(div);  // Reflow on each append!
}

// GOOD: Single reflow
const fragment = document.createDocumentFragment();

for (let i = 0; i < 1000; i++) {
  const div = document.createElement('div');
  div.textContent = `Item ${i}`;
  fragment.appendChild(div);  // Append to fragment (not in DOM)
}

document.body.appendChild(fragment);  // Single reflow!

// Alternative: Build HTML string
const html = Array.from({ length: 1000 }, (_, i) =>
  `<div>Item ${i}</div>`
).join('');
document.body.innerHTML = html;

// Alternative: Use template
const template = document.createElement('template');
template.innerHTML = '<div class="item"></div>';

const fragment = document.createDocumentFragment();
for (let i = 0; i < 1000; i++) {
  const clone = template.content.cloneNode(true);
  clone.querySelector('.item').textContent = `Item ${i}`;
```

```
  fragment.appendChild(clone);
}

document.body.appendChild(fragment);
```

8.10 Event Delegation

```
// BAD: Add listener to each element
document.querySelectorAll('.item').forEach(item => {
  item.addEventListener('click', handleClick);
});
// Problems: Memory overhead, doesn't work for dynamically added elements

// GOOD: Event delegation
document.querySelector('.container').addEventListener('click', (event) => {
  // Check if clicked element matches
  if (event.target.matches('.item')) {
    handleClick(event);
  }

  // Or use closest for nested elements
  const item = event.target.closest('.item');
  if (item) {
    handleClick(event, item);
  }
});

// Event delegation helper
function delegate(element, selector, eventType, handler) {
  element.addEventListener(eventType, (event) => {
    const target = event.target.closest(selector);
    if (target && element.contains(target)) {
      handler.call(target, event);
    }
  });
}

// Usage
delegate(document.body, '.item', 'click', function(event) {
  console.log('Clicked:', this);  // this is the .item element
});
```

8.11 Custom Events

```
// Create custom event
const event = new CustomEvent('user-selected', {
  detail: { userId: 123, name: 'John' },
  bubbles: true,
  cancelable: true,
  composed: true  // Cross shadow DOM boundary
});

// Dispatch event
element.dispatchEvent(event);

// Listen to custom event
element.addEventListener('user-selected', (event) => {
  console.log('User selected:', event.detail);

  // Prevent default
  event.preventDefault();

  // Stop propagation
  event.stopPropagation();
  event.stopImmediatePropagation();  // Stop all listeners
});

// Old-style custom event (IE support)
const event = document.createEvent('CustomEvent');
event.initCustomEvent('user-selected', true, true, { userId: 123 });
element.dispatchEvent(event);
```

8.12 MutationObserver

```
// Watch for DOM changes
const observer = new MutationObserver((mutations) => {
  mutations.forEach((mutation) => {
    console.log('Type:', mutation.type);

    if (mutation.type === 'childList') {
      console.log('Added nodes:', mutation.addedNodes);
      console.log('Removed nodes:', mutation.removedNodes);
    }

    if (mutation.type === 'attributes') {
      console.log('Attribute changed:', mutation.attributeName);
      console.log('Old value:', mutation.oldValue);
    }
```

```
    if (mutation.type === 'characterData') {
      console.log('Text changed:', mutation.target.textContent);
    }
  });
});

// Start observing
observer.observe(document.body, {
  childList: true,  // Watch for child additions/removals
  attributes: true,  // Watch for attribute changes
  characterData: true,  // Watch for text changes
  subtree: true,  // Watch entire subtree
  attributeOldValue: true,  // Record old attribute values
  characterDataOldValue: true  // Record old text values
});

// Stop observing
observer.disconnect();

// Get pending mutations
const mutations = observer.takeRecords();

// Use case: Auto-initialize components
const componentObserver = new MutationObserver((mutations) => {
  mutations.forEach((mutation) => {
    mutation.addedNodes.forEach((node) => {
      if (node.nodeType === Node.ELEMENT_NODE) {
        if (node.matches('[data-component]')) {
          initializeComponent(node);
        }

        // Check descendants
        node.querySelectorAll('[data-component]').forEach(initializeComponent);
      }
    });
  });
});

componentObserver.observe(document.body, { childList: true, subtree: true });

function initializeComponent(element) {
  const componentName = element.dataset.component;
  console.log('Initializing component:', componentName);
  // Initialize component...
}
```

8.13 IntersectionObserver

```
// Watch for element visibility
const observer = new IntersectionObserver((entries) => {
  entries.forEach((entry) => {
    if (entry.isIntersecting) {
      console.log('Element is visible:', entry.target);

      // Lazy load image
      if (entry.target.tagName === 'IMG') {
        entry.target.src = entry.target.dataset.src;
        observer.unobserve(entry.target);
      }
    } else {
      console.log('Element is not visible:', entry.target);
    }
  });
}, {
  root: null,  // viewport
  rootMargin: '0px',  // Margin around root
  threshold: 0.5  // 50% visible
});

// Observe elements
document.querySelectorAll('img[data-src]').forEach((img) => {
  observer.observe(img);
});

// Multiple thresholds
const observer = new IntersectionObserver((entries) => {
  entries.forEach((entry) => {
    console.log('Intersection ratio:', entry.intersectionRatio);
  });
}, {
  threshold: [0, 0.25, 0.5, 0.75, 1]
});

// Infinite scroll
const sentinelObserver = new IntersectionObserver((entries) => {
  entries.forEach((entry) => {
    if (entry.isIntersecting) {
      loadMoreItems();
    }
  });
});

const sentinel = document.querySelector('.sentinel');
```

```
sentinelObserver.observe(sentinel);
```

8.14 ResizeObserver

```
// Watch for element size changes
const observer = new ResizeObserver((entries) => {
  entries.forEach((entry) => {
    console.log('Element:', entry.target);
    console.log('Content rect:', entry.contentRect);
    console.log('Border box size:', entry.borderBoxSize);
    console.log('Content box size:', entry.contentBoxSize);

    const width = entry.contentRect.width;
    const height = entry.contentRect.height;

    // Adjust layout based on size
    if (width < 600) {
      entry.target.classList.add('mobile');
    } else {
      entry.target.classList.remove('mobile');
    }
  });
});

// Observe element
const element = document.querySelector('.container');
observer.observe(element);

// Stop observing
observer.unobserve(element);
observer.disconnect();
```

8.15 Virtual DOM Implementation (Simplified)

```
// Virtual DOM node
function h(tag, props, ...children) {
  return {
    tag,
    props: props || {},
    children: children.flat()
  };
}

// Render virtual DOM to real DOM
```

```
function createElement(vnode) {
  if (typeof vnode === 'string') {
    return document.createTextNode(vnode);
  }

  const element = document.createElement(vnode.tag);

  // Set props
  Object.entries(vnode.props).forEach(([key, value]) => {
    if (key.startsWith('on')) {
      const eventName = key.substring(2).toLowerCase();
      element.addEventListener(eventName, value);
    } else if (key === 'className') {
      element.className = value;
    } else {
      element.setAttribute(key, value);
    }
  });

  // Append children
  vnode.children.forEach((child) => {
    element.appendChild(createElement(child));
  });

  return element;
}

// Diff and patch
function updateElement(parent, newVNode, oldVNode, index = 0) {
  if (!oldVNode) {
    parent.appendChild(createElement(newVNode));
  } else if (!newVNode) {
    parent.removeChild(parent.childNodes[index]);
  } else if (changed(newVNode, oldVNode)) {
    parent.replaceChild(
      createElement(newVNode),
      parent.childNodes[index]
    );
  } else if (newVNode.tag) {
    const newLength = newVNode.children.length;
    const oldLength = oldVNode.children.length;

    for (let i = 0; i < newLength || i < oldLength; i++) {
      updateElement(
        parent.childNodes[index],
        newVNode.children[i],
        oldVNode.children[i],
```

```
        i
      );
    }
  }
}

function changed(node1, node2) {
  return typeof node1 !== typeof node2 ||
         typeof node1 === 'string' && node1 !== node2 ||
         node1.tag !== node2.tag;
}

// Usage
const vdom1 = h('div', { className: 'container' },
  h('h1', {}, 'Hello'),
  h('p', {}, 'World')
);

const vdom2 = h('div', { className: 'container' },
  h('h1', {}, 'Hello'),
  h('p', {}, 'Universe')
);

const root = document.getElementById('app');
root.appendChild(createElement(vdom1));

// Update
updateElement(root, vdom2, vdom1, 0);
```

Chapter 9

Chapter 9: CSS In-Depth - Advanced Concepts

CSS has evolved significantly with modern features that enable sophisticated layouts and designs.

9.1 CSS Specificity Deep Dive

```
/* Specificity calculation: (inline, IDs, classes/attributes/pseudo-classes, elements/p

/* (0, 0, 0, 1) - Specificity: 1 */
p {
  color: black;
}

/* (0, 0, 1, 0) - Specificity: 10 */
.my-class {
  color: blue;
}

/* (0, 0, 1, 1) - Specificity: 11 */
p.my-class {
  color: red;
}

/* (0, 0, 2, 1) - Specificity: 21 */
p.my-class.another-class {
  color: green;
}

/* (0, 1, 0, 0) - Specificity: 100 */
#my-id {
```

```
  color: yellow;
}

/* (0, 1, 1, 1) - Specificity: 111 */
#my-id p.my-class {
  color: purple;
}

/* (1, 0, 0, 0) - Specificity: 1000 */
<p style="color: orange;">Inline style</p>

/* !important overrides everything (but use sparingly) */
p {
  color: pink !important;
}

/* Universal selector has no specificity */
* {
  box-sizing: border-box;  /* (0, 0, 0, 0) */
}

/* Combinators don't add specificity */
div > p {
  /* (0, 0, 0, 2) - just the elements */
}

div + p {
  /* (0, 0, 0, 2) */
}

div ~ p {
  /* (0, 0, 0, 2) */
}

/* :not() doesn't add specificity, but its argument does */
:not(p) {
  /* (0, 0, 0, 1) - specificity of p */
}

/* :is() and :where() */
:is(h1, h2, h3) {
  /* Takes specificity of most specific argument: (0, 0, 0, 1) */
}

:where(h1, h2, h3) {
  /* Always has 0 specificity: (0, 0, 0, 0) */
}
```

9.2 The Cascade

```
/* Cascade order (highest to lowest priority): */
/* 1. Importance and origin */
/*    - User agent !important */
/*    - User !important */
/*    - Author !important */
/*    - Author styles */
/*    - User styles */
/*    - User agent styles */

/* 2. Specificity (see above) */

/* 3. Order of appearance (last wins) */

/* Example */
p { color: red; }
p { color: blue; }  /* This wins (same specificity, appears later) */

/* Layers (CSS @layer) - newest addition */
@layer base, components, utilities;

@layer base {
  p { color: red; }
}

@layer components {
  p { color: blue; }  /* This wins over base layer */
}

@layer utilities {
  p { color: green; }  /* This wins over components layer */
}

/* Unlayered styles win over layered styles */
p { color: purple; }  /* This wins over all layers */
```

9.3 Box Model Deep Dive

```
/* Standard box model */
.box {
  width: 200px;
  height: 100px;
  padding: 20px;
  border: 5px solid black;
  margin: 10px;
```

```css
  /* Total width: 200 + 20*2 + 5*2 = 250px */
  /* Total height: 100 + 20*2 + 5*2 = 150px */
  /* Space occupied: 250 + 10*2 = 270px wide */
}

/* Border-box model (usually preferred) */
.box {
  box-sizing: border-box;
  width: 200px;  /* Includes padding and border */
  height: 100px;
  padding: 20px;
  border: 5px solid black;

  /* Content width: 200 - 20*2 - 5*2 = 150px */
  /* Content height: 100 - 20*2 - 5*2 = 50px */
}

/* Apply to all elements */
*, *::before, *::after {
  box-sizing: border-box;
}

/* Margin collapsing */
.box1 {
  margin-bottom: 20px;
}

.box2 {
  margin-top: 30px;
  /* Gap between boxes is 30px (not 50px) - larger margin wins */
}

/* Prevent margin collapsing */
.parent {
  /* Add border, padding, or overflow */
  overflow: auto;  /* Creates new BFC */
}

/* Negative margins */
.box {
  margin-left: -20px;  /* Pulls element left */
  margin-top: -10px;  /* Pulls element up */
}
```

9.4 Block Formatting Context (BFC)

```
/* BFC is created by: */
/* - root element (<html>) */
/* - floats (float !== none) */
/* - absolutely positioned elements (position: absolute/fixed) */
/* - inline-blocks (display: inline-block) */
/* - table cells (display: table-cell) */
/* - overflow !== visible */
/* - display: flow-root (explicit BFC) */
/* - flex/grid items */
/* - elements with contain: layout/content/paint */

/* BFC contains floats */
.container {
  overflow: auto;  /* Creates BFC */
}

.container .float {
  float: left;
  /* Contained within .container */
}

/* BFC prevents margin collapsing */
.bfc {
  display: flow-root;  /* Explicit BFC */
}

.bfc .child {
  margin-top: 20px;  /* Doesn't collapse with .bfc margin */
}

/* BFC prevents overlap with floats */
.float {
  float: left;
  width: 200px;
}

.content {
  overflow: auto;  /* Creates BFC, doesn't overlap float */
}
```

9.5 Positioning Deep Dive

```
/* Static (default) */
.static {
```

```css
  position: static;
  /* Not affected by top, right, bottom, left */
}

/* Relative */
.relative {
  position: relative;
  top: 10px;  /* Offset from normal position */
  left: 20px;
  /* Original space still occupied */
  /* Creates positioning context for absolute children */
}

/* Absolute */
.absolute {
  position: absolute;
  top: 0;  /* Relative to nearest positioned ancestor */
  right: 0;
  /* Removed from document flow */
  /* Width shrinks to content (unless specified) */
}

/* Fixed */
.fixed {
  position: fixed;
  bottom: 20px;  /* Relative to viewport */
  right: 20px;
  /* Removed from document flow */
  /* Stays in place when scrolling */
}

/* Sticky */
.sticky {
  position: sticky;
  top: 0;  /* Threshold for sticking */
  /* Hybrid: relative until threshold, then fixed */
  /* Parent must have height > sticky element */
}

/* Stacking context */
.parent {
  position: relative;
  z-index: 0;  /* Creates stacking context */
}

.child1 {
  position: absolute;
```

```
  z-index: 100;  /* Only compared within parent */
}

.child2 {
  position: absolute;
  z-index: 200;  /* Higher than child1 */
}

/* Center with absolute positioning */
.center {
  position: absolute;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%);
}

/* Alternative without transform */
.center-alt {
  position: absolute;
  inset: 0;  /* Shorthand for top, right, bottom, left: 0 */
  margin: auto;
  width: 200px;
  height: 100px;
}
```

9.6 Flexbox Deep Dive

```
.container {
  display: flex;

  /* Main axis direction */
  flex-direction: row;  /* row | row-reverse | column | column-reverse */

  /* Wrapping */
  flex-wrap: nowrap;  /* nowrap | wrap | wrap-reverse */

  /* Shorthand */
  flex-flow: row wrap;  /* flex-direction flex-wrap */

  /* Main axis alignment */
  justify-content: flex-start;  /* flex-start | flex-end | center | space-between | spa

  /* Cross axis alignment */
  align-items: stretch;  /* stretch | flex-start | flex-end | center | baseline */
```

```
  /* Multi-line cross axis alignment */
  align-content: flex-start;  /* Same values as justify-content */

  /* Gap */
  gap: 20px;  /* gap between items */
  row-gap: 20px;
  column-gap: 10px;
}

.item {
  /* Growth factor */
  flex-grow: 0;  /* How much to grow relative to siblings */

  /* Shrink factor */
  flex-shrink: 1;  /* How much to shrink relative to siblings */

  /* Base size */
  flex-basis: auto;  /* auto | 200px | 50% */

  /* Shorthand */
  flex: 0 1 auto;  /* flex-grow flex-shrink flex-basis */
  flex: 1;  /* flex: 1 1 0 (common for equal sizing) */
  flex: auto;  /* flex: 1 1 auto */
  flex: none;  /* flex: 0 0 auto */

  /* Individual alignment */
  align-self: auto;  /* auto | flex-start | flex-end | center | baseline | stretch */

  /* Order */
  order: 0;  /* Change visual order (doesn't affect tab order) */
}

/* Common patterns */

/* Equal width columns */
.item {
  flex: 1;
}

/* Fixed sidebar, flexible main */
.sidebar {
  flex: 0 0 200px;
}

.main {
  flex: 1;
}
```

```css
/* Center everything */
.container {
  display: flex;
  justify-content: center;
  align-items: center;
  min-height: 100vh;
}

/* Holy grail layout */
.container {
  display: flex;
  flex-direction: column;
  min-height: 100vh;
}

.header, .footer {
  flex: 0 0 auto;
}

.main {
  display: flex;
  flex: 1;
}

.sidebar {
  flex: 0 0 200px;
}

.content {
  flex: 1;
}
```

9.7 Grid Deep Dive

```css
.container {
  display: grid;

  /* Define columns */
  grid-template-columns: 200px 1fr 2fr;  /* Fixed, 1 fraction, 2 fractions */
  grid-template-columns: repeat(3, 1fr);  /* 3 equal columns */
  grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));  /* Responsive */
  grid-template-columns: repeat(auto-fill, minmax(200px, 1fr));

  /* Define rows */
  grid-template-rows: 100px auto 50px;
```

```
  grid-template-rows: repeat(3, 100px);

  /* Gaps */
  gap: 20px;
  row-gap: 20px;
  column-gap: 10px;

  /* Alignment (container) */
  justify-items: start;  /* start | end | center | stretch */
  align-items: start;
  place-items: center;  /* align-items justify-items */

  /* Alignment (grid) */
  justify-content: start;  /* start | end | center | stretch | space-between | space-ar
  align-content: start;
  place-content: center;  /* align-content justify-content */

  /* Auto rows/columns */
  grid-auto-rows: 100px;  /* Size for implicit rows */
  grid-auto-columns: 200px;

  /* Flow direction */
  grid-auto-flow: row;  /* row | column | row dense | column dense */

  /* Named grid areas */
  grid-template-areas:
    "header header header"
    "sidebar main main"
    "footer footer footer";
}

.item {
  /* Column placement */
  grid-column-start: 1;
  grid-column-end: 3;
  grid-column: 1 / 3;  /* Shorthand */
  grid-column: 1 / span 2;  /* Span 2 columns */
  grid-column: 1 / -1;  /* Span to end */

  /* Row placement */
  grid-row-start: 1;
  grid-row-end: 3;
  grid-row: 1 / 3;

  /* Shorthand */
  grid-area: 1 / 1 / 3 / 3;  /* row-start / column-start / row-end / column-end */
```

```css
  /* Named areas */
  grid-area: header;

  /* Individual alignment */
  justify-self: start;  /* start | end | center | stretch */
  align-self: start;
  place-self: center;  /* align-self justify-self */
}

/* Common patterns */

/* Responsive grid */
.grid {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));
  gap: 20px;
}

/* 12-column grid system */
.grid {
  display: grid;
  grid-template-columns: repeat(12, 1fr);
  gap: 20px;
}

.col-6 {
  grid-column: span 6;
}

.col-4 {
  grid-column: span 4;
}

/* Asymmetric layout */
.grid {
  display: grid;
  grid-template-columns: 2fr 1fr;
  grid-template-rows: auto 1fr auto;
  grid-template-areas:
    "header header"
    "main sidebar"
    "footer footer";
  gap: 20px;
  min-height: 100vh;
}

.header { grid-area: header; }
```

```css
.main { grid-area: main; }
.sidebar { grid-area: sidebar; }
.footer { grid-area: footer; }

/* Subgrid */
.grid {
  display: grid;
  grid-template-columns: repeat(4, 1fr);
}

.item {
  display: grid;
  grid-column: span 2;
  grid-template-columns: subgrid;  /* Inherit parent's columns */
}
```

9.8 Modern CSS Features

```css
/* Container Queries */
.card {
  container-type: inline-size;  /* Makes element a container */
  container-name: card;
}

@container card (min-width: 400px) {
  .card-title {
    font-size: 2rem;
  }
}

/* Logical Properties */
.box {
  /* Instead of margin-left/right */
  margin-inline-start: 20px;
  margin-inline-end: 20px;
  margin-inline: 20px;  /* Shorthand */

  /* Instead of margin-top/bottom */
  margin-block-start: 10px;
  margin-block-end: 10px;
  margin-block: 10px;

  /* Works for padding, border, etc. */
  padding-inline: 20px;
  padding-block: 10px;
  border-inline: 1px solid black;
```

```css
  /* Positioning */
  inset-inline-start: 0;  /* left in LTR, right in RTL */
  inset-inline-end: 0;
  inset-block-start: 0;  /* top */
  inset-block-end: 0;  /* bottom */
  inset: 0;  /* Shorthand for all */
}

/* CSS Custom Properties (Variables) */
:root {
  --primary-color: #007bff;
  --secondary-color: #6c757d;
  --spacing: 20px;
  --font-size-base: 16px;
}

.button {
  background-color: var(--primary-color);
  padding: var(--spacing);
  font-size: var(--font-size-base);

  /* Fallback */
  color: var(--text-color, black);
}

/* Modify variables */
.dark-theme {
  --primary-color: #0056b3;
  --secondary-color: #545b62;
}

/* calc() with variables */
.box {
  width: calc(100% - var(--spacing) * 2);
  padding: calc(var(--spacing) / 2);
}

/* clamp() - responsive sizing */
.title {
  font-size: clamp(1.5rem, 5vw, 3rem);
  /* min, preferred, max */
}

.container {
  width: clamp(300px, 90%, 1200px);
}
```

```css
/* min() / max() */
.box {
  width: min(90%, 1200px);  /* Smaller of the two */
  height: max(200px, 50vh);  /* Larger of the two */
}

/* aspect-ratio */
.video {
  aspect-ratio: 16 / 9;
  width: 100%;
  /* Height automatically calculated */
}

.square {
  aspect-ratio: 1;
  width: 200px;
}

/* Gap (works with flex and grid) */
.flex {
  display: flex;
  gap: 20px;
}

.grid {
  display: grid;
  gap: 20px 10px;  /* row column */
}

/* :is() and :where() */
:is(h1, h2, h3, h4, h5, h6) {
  margin-block: 1em;
}

:where(article, section, aside) > p {
  line-height: 1.5;
}

/* :has() - parent selector */
.card:has(img) {
  display: grid;
  grid-template-columns: 200px 1fr;
}

/* Select card that has a .featured class in any descendant */
.card:has(.featured) {
  border: 2px solid gold;
```

```css
}

/* Select label that has a required input */
label:has(+ input:required) {
  font-weight: bold;
}

/* @supports - feature queries */
@supports (display: grid) {
  .container {
    display: grid;
  }
}

@supports not (display: grid) {
  .container {
    display: flex;
  }
}

@supports (display: grid) and (gap: 20px) {
  .container {
    display: grid;
    gap: 20px;
  }
}

/* @media with range syntax */
@media (width >= 768px) {
  .container {
    max-width: 1200px;
  }
}

@media (400px <= width <= 1000px) {
  .container {
    padding: 20px;
  }
}

/* Scroll snap */
.scroll-container {
  scroll-snap-type: x mandatory;  /* x | y | both; mandatory | proximity */
  overflow-x: scroll;
  display: flex;
}
```

```css
.scroll-item {
  scroll-snap-align: start;  /* start | end | center */
  scroll-snap-stop: always;  /* always | normal */
  flex: 0 0 100%;
}

/* Smooth scrolling */
html {
  scroll-behavior: smooth;
}

/* overscroll-behavior */
.modal {
  overscroll-behavior: contain;  /* Prevent scroll chaining */
}

/* content-visibility */
.section {
  content-visibility: auto;  /* Defer rendering offscreen content */
  contain-intrinsic-size: 0 500px;  /* Estimated size */
}
```

Chapter 10

Chapter 10: Frontend Interview Questions - Advanced

This section covers common intermediate to advanced frontend interview questions.

10.1 JavaScript Interview Questions

10.1.1 Question: Explain the event loop. What are microtasks vs macrotasks?

Answer:

```
// The event loop processes tasks in this order:
// 1. Execute synchronous code
// 2. Process all microtasks
// 3. Render (if needed)
// 4. Process one macrotask
// 5. Repeat from step 2

// Macrotasks (Task Queue):
// - setTimeout
// - setInterval
// - setImmediate (Node.js)
// - I/O operations
// - UI rendering

// Microtasks (Microtask Queue):
// - Promise callbacks (.then, .catch, .finally)
// - queueMicrotask()
// - MutationObserver
// - process.nextTick (Node.js) - even higher priority

console.log('1');  // Synchronous
```

```javascript
setTimeout(() => {
  console.log('2');  // Macrotask
}, 0);

Promise.resolve().then(() => {
  console.log('3');  // Microtask
});

console.log('4');  // Synchronous

// Output: 1, 4, 3, 2

// Complex example
console.log('Start');

setTimeout(() => {
  console.log('Timeout 1');
  Promise.resolve().then(() => console.log('Promise 1'));
}, 0);

Promise.resolve().then(() => {
  console.log('Promise 2');
  setTimeout(() => console.log('Timeout 2'), 0);
});

console.log('End');

// Output:
// Start
// End
// Promise 2
// Timeout 1
// Promise 1
// Timeout 2

// Why?
// 1. Sync: Start, End
// 2. Microtasks: Promise 2 (schedules Timeout 2)
// 3. Macrotask: Timeout 1 (schedules Promise 1)
// 4. Microtasks: Promise 1
// 5. Macrotask: Timeout 2
```

10.1.2 Question: What are closures and where are they used?

Answer:

```javascript
// Closure: Function that has access to outer function's variables

// Example 1: Private variables
function createCounter() {
  let count = 0;  // Private variable

  return {
    increment() {
      count++;
      return count;
    },
    decrement() {
      count--;
      return count;
    },
    getCount() {
      return count;
    }
  };
}

const counter = createCounter();
console.log(counter.increment());  // 1
console.log(counter.increment());  // 2
console.log(counter.getCount());   // 2
// console.log(counter.count);  // undefined - private!

// Example 2: Event handlers
function setupButton() {
  let clickCount = 0;

  document.getElementById('btn').addEventListener('click', function() {
    clickCount++;
    console.log(`Clicked ${clickCount} times`);
  });
}

// Example 3: Function factories
function createMultiplier(multiplier) {
  return function(value) {
    return value * multiplier;
  };
}

const double = createMultiplier(2);
const triple = createMultiplier(3);
```

```
console.log(double(5));  // 10
console.log(triple(5));  // 15

// Example 4: Partial application
function partial(fn, ...fixedArgs) {
  return function(...remainingArgs) {
    return fn(...fixedArgs, ...remainingArgs);
  };
}

function add(a, b, c) {
  return a + b + c;
}

const add5 = partial(add, 5);
console.log(add5(3, 2));  // 10

// Common closure gotcha
for (var i = 0; i < 3; i++) {
  setTimeout(function() {
    console.log(i);  // Prints 3, 3, 3
  }, 100);
}

// Fix 1: Use let (block scope)
for (let i = 0; i < 3; i++) {
  setTimeout(function() {
    console.log(i);  // Prints 0, 1, 2
  }, 100);
}

// Fix 2: Use IIFE
for (var i = 0; i < 3; i++) {
  (function(j) {
    setTimeout(function() {
      console.log(j);  // Prints 0, 1, 2
    }, 100);
  })(i);
}
```

10.1.3 Advanced Closure Utility Functions

Closures are the foundation for many powerful utility functions. This section provides exhaustive examples of closure-based utilities used in production applications.

10.1.3.1 A. Execution Control Utilities

These functions control when or how often another function executes by maintaining state through closures.

```
/**
 * Creates a function that invokes fn only once.
 * Subsequent calls return the cached result.
 *
 * Use cases:
 * - Initialization functions
 * - Expensive one-time computations
 * - Event handlers that should fire once
 */
function once(fn) {
  let called = false;
  let result;

  return function(...args) {
    if (!called) {
      called = true;
      result = fn.apply(this, args);
    }
    return result;
  };
}

// Example 1: Expensive initialization
const initializeApp = once(() => {
  console.log('Initializing application...');
  // Load configurations, connect to services, etc.
  return {
    config: { apiUrl: 'https://api.example.com' },
    initialized: true
  };
});

const app1 = initializeApp();  // Logs: "Initializing application..."
const app2 = initializeApp();  // No log, returns cached result
console.log(app1 === app2);    // true - same object

// Example 2: One-time event handler
const button = document.getElementById('subscribe-btn');
const handleSubscribe = once(async function() {
  console.log('Subscribing user...');
  await fetch('/api/subscribe', { method: 'POST' });
  this.textContent = 'Subscribed!';
```

```
  this.disabled = true;
});

button.addEventListener('click', handleSubscribe);
// Even if clicked multiple times, subscription happens only once

// Example 3: Singleton pattern
const createDatabase = once(() => {
  console.log('Connecting to database...');
  return {
    connection: 'db-connection-instance',
    query: (sql) => console.log('Executing:', sql)
  };
});

const db1 = createDatabase();  // Creates connection
const db2 = createDatabase();  // Returns same instance
console.log(db1 === db2);      // true
```

10.1.3.1.1 1. once(fn) - Execute Function Only Once

```

/**
 * Creates a function that invokes fn at most n times.
 * After n calls, returns the result of the last invocation.
 *
 * Use cases:
 * - Trial features (allow n free uses)
 * - Rate limiting user actions
 * - Demo functionality
 */
function before(n, fn) {
  let count = 0;
  let lastResult;

  return function(...args) {
    if (count < n) {
      count++;
      lastResult = fn.apply(this, args);
    }
    return lastResult;
  };
}

// Example 1: Free trial with limited uses
const freeSearch = before(3, (query) => {
  console.log(`Searching for: ${query}`);
```

```
  return `Results for "${query}"`;
});

console.log(freeSearch('javascript'));  // Works: "Results for javascript"
console.log(freeSearch('closures'));    // Works: "Results for closures"
console.log(freeSearch('patterns'));    // Works: "Results for patterns"
console.log(freeSearch('advanced'));    // Returns last result, doesn't search

// Example 2: Limited hints in a game
const getHint = before(5, (level) => {
  console.log(`Providing hint for level ${level}`);
  return `Hint: Look at the ${level} pattern`;
});

for (let i = 1; i <= 7; i++) {
  const hint = getHint(i);
  console.log(`Level ${i}:`, hint);
  // Only first 5 calls actually generate hints
}

// Example 3: Preview feature
const previewFeature = before(10, (data) => {
  console.log('Using premium feature with:', data);
  return { success: true, data };
});

// Users can try premium feature 10 times before paywall
```

10.1.3.1.2 2. before(n, fn) - Limit Executions

```
/**
 * Creates a function that invokes fn only after being called n times.
 *
 * Use cases:
 * - Wait for multiple async operations
 * - Batch processing triggers
 * - Warming up before execution
 */
function after(n, fn) {
  let count = 0;

  return function(...args) {
    count++;
    if (count >= n) {
      return fn.apply(this, args);
    }
```

```
  };
}

// Example 1: Wait for multiple resources to load
const resources = ['config', 'user', 'settings'];
const allResourcesLoaded = after(resources.length, () => {
  console.log('All resources loaded! Starting app...');
  startApplication();
});

// Simulating async resource loading
resources.forEach((resource, index) => {
  setTimeout(() => {
    console.log(`Loaded: ${resource}`);
    allResourcesLoaded();
  }, (index + 1) * 100);
});

// Example 2: Batch confirmation
const confirmDeletion = after(3, () => {
  console.log('Triple-confirmed! Deleting...');
  deleteAccount();
});

deleteButton.addEventListener('click', confirmDeletion);
// User must click 3 times to actually delete

// Example 3: Warm-up period
let warmupRuns = 0;
const optimizedFunction = after(5, (data) => {
  console.log('Fully optimized! Processing:', data);
  // JIT compilation should be complete after 5 runs
  return heavyComputation(data);
});

// First 4 calls warm up the function
for (let i = 0; i < 10; i++) {
  optimizedFunction({ value: i });
}
```

10.1.3.1.3 3. after(n, fn) - Execute After n Calls

```
/**
 * Creates a throttled function that only invokes fn at most once per wait ms.
 * Leading edge execution by default.
 *
```

```
 * Use cases:
 * - Scroll handlers
 * - Window resize handlers
 * - API call limiting
 */
function throttle(fn, wait, options = {}) {
  let timeout;
  let lastRan = 0;
  const { leading = true, trailing = true } = options;

  return function(...args) {
    const context = this;
    const now = Date.now();

    // First call or enough time has passed
    if (!lastRan && !leading) {
      lastRan = now;
    }

    const remaining = wait - (now - lastRan);

    if (remaining <= 0 || remaining > wait) {
      if (timeout) {
        clearTimeout(timeout);
        timeout = null;
      }

      lastRan = now;
      fn.apply(context, args);
    } else if (!timeout && trailing) {
      timeout = setTimeout(() => {
        lastRan = leading ? Date.now() : 0;
        timeout = null;
        fn.apply(context, args);
      }, remaining);
    }
  };
}

// Example 1: Throttle scroll handler
let scrollCount = 0;
const handleScroll = throttle(() => {
  scrollCount++;
  console.log(`Scroll handler called: ${scrollCount} times`);
  console.log('Scroll position:', window.scrollY);

  // Update UI based on scroll position
```

```
  updateScrollProgress();
}, 200);

window.addEventListener('scroll', handleScroll);
// Even with rapid scrolling, handler runs at most once per 200ms

// Example 2: Throttle API calls
const searchAPI = throttle(async (query) => {
  console.log('Calling API with:', query);
  const response = await fetch(`/api/search?q=${query}`);
  const results = await response.json();
  displayResults(results);
}, 1000);

searchInput.addEventListener('input', (e) => {
  searchAPI(e.target.value);
});
// API called at most once per second, even with fast typing

// Example 3: Throttle expensive calculations
const updateChart = throttle((data) => {
  console.log('Redrawing chart...');
  // Expensive chart rendering
  chart.update(data);
}, 500);

// Real-time data updates
dataStream.on('data', updateChart);
// Chart updates at most twice per second, preventing UI lag

// Example 4: Throttle with trailing edge only
const saveProgress = throttle((gameState) => {
  console.log('Saving game progress...');
  localStorage.setItem('gameState', JSON.stringify(gameState));
}, 5000, { leading: false, trailing: true });

// Auto-save every 5 seconds max
setInterval(() => {
  saveProgress(getCurrentGameState());
}, 100);
```

10.1.3.1.4 4. throttle(fn, wait) - Rate Limit Execution

```
/**
 * Creates a debounced function that delays invoking fn until after
 * wait ms have elapsed since the last call.
```

```
 *
 * Use cases:
 * - Search input (wait for user to finish typing)
 * - Form validation
 * - Window resize handlers
 */
function debounce(fn, wait, options = {}) {
  let timeout;
  const { leading = false, trailing = true, maxWait } = options;
  let lastCallTime;
  let lastInvokeTime = 0;

  function invokeFunc(time, args, context) {
    lastInvokeTime = time;
    return fn.apply(context, args);
  }

  function shouldInvoke(time) {
    const timeSinceLastCall = time - (lastCallTime || 0);
    const timeSinceLastInvoke = time - lastInvokeTime;

    return (
      !lastCallTime ||
      timeSinceLastCall >= wait ||
      timeSinceLastCall < 0 ||
      (maxWait !== undefined && timeSinceLastInvoke >= maxWait)
    );
  }

  return function(...args) {
    const context = this;
    const time = Date.now();
    const isInvoking = shouldInvoke(time);

    lastCallTime = time;

    if (isInvoking && leading && !timeout) {
      invokeFunc(time, args, context);
    }

    if (timeout) {
      clearTimeout(timeout);
    }

    timeout = setTimeout(() => {
      const time = Date.now();
      timeout = null;
```

```
      if (trailing) {
        invokeFunc(time, args, context);
      }

      lastCallTime = undefined;
    }, wait);
  };
}

// Example 1: Search input debouncing
let apiCallCount = 0;
const debouncedSearch = debounce(async (query) => {
  apiCallCount++;
  console.log(`API call #${apiCallCount}: searching for "${query}"`);

  const response = await fetch(`/api/search?q=${query}`);
  const results = await response.json();
  displayResults(results);
}, 300);

searchInput.addEventListener('input', (e) => {
  debouncedSearch(e.target.value);
});
// API only called 300ms after user stops typing

// Example 2: Form validation
const validateEmail = debounce(async (email) => {
  console.log('Validating email:', email);

  const response = await fetch('/api/validate-email', {
    method: 'POST',
    body: JSON.stringify({ email })
  });

  const { valid, message } = await response.json();

  if (valid) {
    emailInput.classList.add('valid');
    emailInput.classList.remove('invalid');
  } else {
    emailInput.classList.add('invalid');
    emailInput.classList.remove('valid');
    showError(message);
  }
}, 500);

emailInput.addEventListener('input', (e) => {
```

```
  validateEmail(e.target.value);
});

// Example 3: Window resize handler
const handleResize = debounce(() => {
  console.log('Window resized to:', window.innerWidth, 'x', window.innerHeight);

  // Recalculate layout
  recalculateLayout();

  // Update responsive components
  updateResponsiveComponents();
}, 250);

window.addEventListener('resize', handleResize);
// Layout calculations only happen after resize is complete

// Example 4: Auto-save with debounce
const autoSave = debounce((content) => {
  console.log('Auto-saving document...');

  fetch('/api/documents/save', {
    method: 'POST',
    body: JSON.stringify({ content })
  })
  .then(() => {
    console.log('Document saved!');
    showSaveIndicator();
  });
}, 2000);

editor.addEventListener('input', (e) => {
  autoSave(e.target.value);
});
// Saves 2 seconds after user stops typing

// Example 5: Debounce with maxWait (guaranteed execution)
const criticalSave = debounce((data) => {
  console.log('Saving critical data...');
  saveToDB(data);
}, 1000, { maxWait: 5000 });

// Will execute at most every 5 seconds, even if called continuously
setInterval(() => {
  criticalSave(getCurrentData());
}, 100);
```

10.1.3.1.5 5. debounce(fn, wait) - Delay Until Idle

```
/**
 * Delays execution of fn by ms milliseconds.
 * Returns a function that can be called with arguments later.
 *
 * Use cases:
 * - Tooltip delays
 * - Intentional UX delays
 * - Animation timing
 */
function delay(fn, ms) {
  return function(...args) {
    const context = this;
    return new Promise((resolve) => {
      setTimeout(() => {
        resolve(fn.apply(context, args));
      }, ms);
    });
  };
}

// Example 1: Tooltip delay
const showTooltip = delay((element, message) => {
  console.log('Showing tooltip:', message);
  const tooltip = document.createElement('div');
  tooltip.className = 'tooltip';
  tooltip.textContent = message;
  element.appendChild(tooltip);
  return tooltip;
}, 500);

helpIcon.addEventListener('mouseenter', function() {
  showTooltip(this, 'Click for more information');
});

helpIcon.addEventListener('mouseleave', function() {
  // Only show tooltip if mouse stays for 500ms
  const tooltip = this.querySelector('.tooltip');
  if (tooltip) tooltip.remove();
});

// Example 2: Delayed notification
const showNotification = delay((message, type = 'info') => {
  console.log(`[${type.toUpperCase()}] ${message}`);
```

```
  const notification = document.createElement('div');
  notification.className = `notification ${type}`;
  notification.textContent = message;
  document.body.appendChild(notification);

  return notification;
}, 1000);

async function processOrder(order) {
  console.log('Processing order...');
  // Process order

  await showNotification('Order processed successfully!', 'success');
}

// Example 3: Delayed redirect
const delayedRedirect = delay((url) => {
  console.log('Redirecting to:', url);
  window.location.href = url;
}, 3000);

loginButton.addEventListener('click', async () => {
  const success = await login();
  if (success) {
    showMessage('Login successful! Redirecting...');
    await delayedRedirect('/dashboard');
  }
});

// Example 4: Staggered animations
const animateIn = delay((element, duration) => {
  element.style.transition = `all ${duration}ms`;
  element.style.opacity = '1';
  element.style.transform = 'translateY(0)';
}, 100);

const items = document.querySelectorAll('.list-item');
items.forEach((item, index) => {
  // Stagger animations by 100ms each
  setTimeout(() => animateIn(item, 300), index * 100);
});
```

10.1.3.1.6 6. delay(fn, ms) - Defer Execution

```
/**
 * Queues function execution to the next event loop tick.
```

```
 * Useful for ensuring DOM updates or letting other operations complete.
 *
 * Use cases:
 * - DOM manipulation after render
 * - Breaking up heavy computations
 * - Microtask scheduling
 */
function defer(fn) {
  return function(...args) {
    const context = this;
    return new Promise((resolve) => {
      setTimeout(() => {
        resolve(fn.apply(context, args));
      }, 0);
    });
  };
}

// Example 1: Defer DOM updates
const updateDOM = defer(() => {
  console.log('Updating DOM after render...');

  // Measure layout
  const rect = element.getBoundingClientRect();
  console.log('Element position:', rect);

  // Update based on measurements
  element.style.width = `${rect.width * 1.5}px`;
});

// DOM modifications
element.textContent = 'New content';
element.classList.add('expanded');

// Update happens after browser paints
updateDOM();

// Example 2: Break up heavy computation
function processLargeDataset(data) {
  const deferredProcess = defer((chunk) => {
    console.log('Processing chunk of size:', chunk.length);
    // Heavy processing
    return chunk.map(item => expensiveOperation(item));
  });

  const chunkSize = 1000;
  const promises = [];
```

```
  for (let i = 0; i < data.length; i += chunkSize) {
    const chunk = data.slice(i, i + chunkSize);
    promises.push(deferredProcess(chunk));
  }

  return Promise.all(promises).then(results => results.flat());
}

// Example 3: Priority queue simulation
const lowPriorityTask = defer(() => {
  console.log('Low priority task executed');
  return 'low-priority-result';
});

const highPriorityTask = () => {
  console.log('High priority task executed');
  return 'high-priority-result';
};

// High priority executes first
highPriorityTask();
lowPriorityTask();  // Executes in next tick
```

10.1.3.1.7 7. defer(fn) - Next Tick Execution

```
/**
 * Limits function to a maximum number of calls per time window.
 * Different from throttle - tracks all calls in window.
 *
 * Use cases:
 * - API rate limiting
 * - User action restrictions
 * - Resource usage control
 */
function rateLimit(fn, limit, windowMs) {
  const calls = [];

  return function(...args) {
    const now = Date.now();

    // Remove calls outside current window
    while (calls.length > 0 && calls[0] <= now - windowMs) {
      calls.shift();
    }

    // Check if under limit
```

```
    if (calls.length < limit) {
      calls.push(now);
      return fn.apply(this, args);
    } else {
      const oldestCall = calls[0];
      const waitTime = windowMs - (now - oldestCall);
      throw new Error(`Rate limit exceeded. Try again in ${waitTime}ms`);
    }
  };
}

// Example 1: API rate limiting
const callAPI = rateLimit(async (endpoint, data) => {
  console.log(`Calling API: ${endpoint}`);
  const response = await fetch(endpoint, {
    method: 'POST',
    body: JSON.stringify(data)
  });
  return response.json();
}, 10, 60000);  // 10 calls per minute

// Usage
try {
  for (let i = 0; i < 15; i++) {
    await callAPI('/api/data', { index: i });
  }
} catch (error) {
  console.error(error.message);  // Rate limit exceeded after 10 calls
}

// Example 2: User action limiting
const sendMessage = rateLimit((message) => {
  console.log('Sending message:', message);
  socket.emit('message', message);
  return true;
}, 5, 10000);  // 5 messages per 10 seconds

sendButton.addEventListener('click', () => {
  try {
    const message = messageInput.value;
    sendMessage(message);
    messageInput.value = '';
  } catch (error) {
    showNotification(error.message, 'warning');
  }
});
```

```javascript
// Example 3: Download limiting
const downloadFile = rateLimit(async (fileId) => {
  console.log('Downloading file:', fileId);
  const response = await fetch(`/api/files/${fileId}/download`);
  const blob = await response.blob();
  saveFile(blob, `file-${fileId}.pdf`);
}, 3, 3600000);  // 3 downloads per hour

// Example 4: Adaptive rate limiting
function createAdaptiveRateLimit(fn, baseLimit, windowMs) {
  let currentLimit = baseLimit;
  const limiter = rateLimit(fn, currentLimit, windowMs);

  return {
    call: function(...args) {
      return limiter.apply(this, args);
    },
    increaseLimit: (amount) => {
      currentLimit += amount;
      console.log('Rate limit increased to:', currentLimit);
    },
    decreaseLimit: (amount) => {
      currentLimit = Math.max(1, currentLimit - amount);
      console.log('Rate limit decreased to:', currentLimit);
    }
  };
}
```

10.1.3.1.8 8. rateLimit(fn, limit, window) - Time Window Limiting

```javascript
/**
 * Continuously invokes fn at fixed intervals until stopped.
 * Returns a controller to start/stop polling.
 *
 * Use cases:
 * - Status checking
 * - Real-time updates fallback
 * - Health monitoring
 */
function poll(fn, interval) {
  let timerId = null;
  let isPolling = false;

  return {
    start: function() {
      if (isPolling) return;
```

```
      isPolling = true;
      console.log('Starting poll...');

      const execute = async () => {
        try {
          await fn();
        } catch (error) {
          console.error('Poll error:', error);
        }

        if (isPolling) {
          timerId = setTimeout(execute, interval);
        }
      };

      execute();  // Execute immediately
    },

    stop: function() {
      isPolling = false;
      if (timerId) {
        clearTimeout(timerId);
        timerId = null;
      }
      console.log('Polling stopped');
    },

    isActive: function() {
      return isPolling;
    }
  };
}

// Example 1: Poll job status
const checkJobStatus = poll(async () => {
  console.log('Checking job status...');
  const response = await fetch('/api/job/status');
  const { status, progress } = await response.json();

  updateProgressBar(progress);

  if (status === 'completed') {
    console.log('Job completed!');
    jobPoller.stop();
    showCompletionMessage();
  }
}, 2000);  // Check every 2 seconds
```

```javascript
// Start polling when job is submitted
submitButton.addEventListener('click', async () => {
  await submitJob();
  checkJobStatus.start();
});

// Example 2: Server health check
const healthCheck = poll(async () => {
  try {
    const response = await fetch('/api/health', { timeout: 5000 });
    const { status } = await response.json();

    if (status === 'healthy') {
      serverIndicator.className = 'status-healthy';
    } else {
      serverIndicator.className = 'status-degraded';
    }
  } catch (error) {
    serverIndicator.className = 'status-down';
    console.error('Server unreachable');
  }
}, 30000);  // Check every 30 seconds

healthCheck.start();

// Example 3: Conditional polling
const pollUntilCondition = (fn, interval, condition) => {
  const poller = poll(async () => {
    const result = await fn();
    if (condition(result)) {
      poller.stop();
    }
  }, interval);

  return poller;
};

// Usage: Poll until specific condition met
const waitForData = pollUntilCondition(
  async () => {
    const response = await fetch('/api/data');
    return response.json();
  },
  1000,
  (data) => data.ready === true
);
```

```
waitForData.start();

// Example 4: Polling with exponential backoff
function pollWithBackoff(fn, initialInterval, maxInterval) {
  let currentInterval = initialInterval;
  let timerId = null;
  let isPolling = false;

  return {
    start: function() {
      isPolling = true;

      const execute = async () => {
        try {
          const result = await fn();

          // Success - reset interval
          if (result.success) {
            currentInterval = initialInterval;
          } else {
            // Failure - increase interval
            currentInterval = Math.min(currentInterval * 2, maxInterval);
          }
        } catch (error) {
          currentInterval = Math.min(currentInterval * 2, maxInterval);
        }

        if (isPolling) {
          console.log(`Next poll in ${currentInterval}ms`);
          timerId = setTimeout(execute, currentInterval);
        }
      };

      execute();
    },

    stop: function() {
      isPolling = false;
      if (timerId) clearTimeout(timerId);
    }
  };
}
```

10.1.3.1.9 9. poll(fn, interval) - Continuous Polling

```javascript
/**
 * Retries a failing function up to times attempts.
 * Supports exponential backoff and custom retry conditions.
 *
 * Use cases:
 * - Network request retries
 * - Flaky operation handling
 * - Resilient API calls
 */
function retry(fn, times, options = {}) {
  const {
    delay = 1000,
    exponential = true,
    onRetry = null,
    shouldRetry = () => true
  } = options;

  return async function(...args) {
    let lastError;

    for (let attempt = 0; attempt < times; attempt++) {
      try {
        return await fn.apply(this, args);
      } catch (error) {
        lastError = error;

        // Check if should retry
        if (!shouldRetry(error, attempt)) {
          throw error;
        }

        // Last attempt - don't delay
        if (attempt === times - 1) {
          throw error;
        }

        // Calculate delay
        const waitTime = exponential
          ? delay * Math.pow(2, attempt)
          : delay;

        console.log(`Attempt ${attempt + 1} failed. Retrying in ${waitTime}ms...`);

        if (onRetry) {
          onRetry(error, attempt + 1, waitTime);
        }
```

```
        // Wait before retry
        await new Promise(resolve => setTimeout(resolve, waitTime));
      }
    }

    throw lastError;
  };
}

// Example 1: Retry API call
const fetchWithRetry = retry(
  async (url) => {
    console.log('Fetching:', url);
    const response = await fetch(url);

    if (!response.ok) {
      throw new Error(`HTTP ${response.status}`);
    }

    return response.json();
  },
  3,
  {
    delay: 1000,
    exponential: true,
    onRetry: (error, attempt, delay) => {
      console.log(`Retry attempt ${attempt} after ${delay}ms`);
      showNotification(`Retrying... (${attempt}/3)`);
    }
  }
);

// Usage
try {
  const data = await fetchWithRetry('/api/data');
  console.log('Success:', data);
} catch (error) {
  console.error('Failed after 3 retries:', error);
  showError('Unable to load data. Please try again later.');
}

// Example 2: Retry with conditional logic
const fetchCriticalData = retry(
  async () => {
    const response = await fetch('/api/critical');
    return response.json();
  },
```

```
  5,
  {
    delay: 2000,
    shouldRetry: (error, attempt) => {
      // Retry on network errors and 5xx, but not 4xx
      if (error.message.includes('HTTP 4')) {
        return false;  // Client error - don't retry
      }
      return true;  // Network or server error - retry
    }
  }
);

// Example 3: Retry database connection
const connectWithRetry = retry(
  async (config) => {
    console.log('Attempting database connection...');
    const connection = await database.connect(config);

    // Test connection
    await connection.query('SELECT 1');

    console.log('Database connected successfully!');
    return connection;
  },
  10,
  {
    delay: 5000,
    exponential: true,
    onRetry: (error, attempt) => {
      console.error(`Connection failed (attempt ${attempt}):`, error.message);
    }
  }
);

// Example 4: Retry with circuit breaker
function retryWithCircuitBreaker(fn, times, options = {}) {
  let failures = 0;
  let circuitOpen = false;
  let resetTimer = null;

  const maxFailures = options.maxFailures || 5;
  const resetTimeout = options.resetTimeout || 60000;

  return async function(...args) {
    // Circuit breaker is open
    if (circuitOpen) {
```

```
      throw new Error('Circuit breaker is open. Service temporarily unavailable.');
    }

    try {
      const result = await retry(fn, times, options).apply(this, args);

      // Success - reset failure count
      failures = 0;
      if (resetTimer) {
        clearTimeout(resetTimer);
        resetTimer = null;
      }

      return result;
    } catch (error) {
      failures++;

      // Open circuit if too many failures
      if (failures >= maxFailures) {
        circuitOpen = true;
        console.log('Circuit breaker opened due to repeated failures');

        // Attempt to close circuit after timeout
        resetTimer = setTimeout(() => {
          console.log('Attempting to close circuit breaker...');
          circuitOpen = false;
          failures = 0;
        }, resetTimeout);
      }

      throw error;
    }
  };
}

// Usage
const resilientFetch = retryWithCircuitBreaker(
  async (url) => {
    const response = await fetch(url);
    if (!response.ok) throw new Error(`HTTP ${response.status}`);
    return response.json();
  },
  3,
  {
    delay: 1000,
    maxFailures: 5,
    resetTimeout: 60000
```

```
  }
);
```

10.1.3.1.10 10. retry(fn, times) - Retry Failed Operations

```
/**
 * Queues multiple calls and executes them together in batches.
 * Useful for optimizing bulk operations.
 *
 * Use cases:
 * - Bulk API requests
 * - Database batch inserts
 * - Analytics event batching
 */
function batch(fn, limit) {
  let queue = [];
  let timer = null;

  return function(item) {
    return new Promise((resolve, reject) => {
      queue.push({ item, resolve, reject });

      // Clear existing timer
      if (timer) {
        clearTimeout(timer);
      }

      // Process immediately if limit reached
      if (queue.length >= limit) {
        processQueue();
      } else {
        // Otherwise wait a bit for more items
        timer = setTimeout(processQueue, 100);
      }
    });

    function processQueue() {
      const batch = queue.splice(0, limit);
      const items = batch.map(b => b.item);

      console.log(`Processing batch of ${items.length} items`);

      fn(items)
        .then(results => {
          batch.forEach((b, index) => {
            b.resolve(results[index]);
```

```
        });
      })
      .catch(error => {
        batch.forEach(b => b.reject(error));
      });
    }
  };
}

// Example 1: Batch API requests
const batchedFetch = batch(async (userIds) => {
  console.log('Fetching users:', userIds);
  const response = await fetch('/api/users/batch', {
    method: 'POST',
    body: JSON.stringify({ ids: userIds })
  });
  return response.json();
}, 10);

// Individual calls are automatically batched
async function loadUserProfile(userId) {
  const user = await batchedFetch(userId);
  displayUser(user);
}

// These 15 calls result in only 2 API calls (10 + 5)
for (let i = 1; i <= 15; i++) {
  loadUserProfile(i);
}

// Example 2: Analytics event batching
const trackEvent = batch(async (events) => {
  console.log(`Sending ${events.length} analytics events`);
  await fetch('/api/analytics', {
    method: 'POST',
    body: JSON.stringify({ events })
  });
  return events.map(() => ({ success: true }));
}, 20);

// Individual tracking calls
document.querySelectorAll('button').forEach(button => {
  button.addEventListener('click', () => {
    trackEvent({
      type: 'click',
      target: button.id,
      timestamp: Date.now()
```

```
    });
  });
});

// Example 3: Database batch insert
const batchInsert = batch(async (records) => {
  console.log(`Inserting ${records.length} records`);
  const query = `INSERT INTO logs (message, timestamp) VALUES ${
    records.map(() => '(?, ?)').join(', ')
  }`;
  const params = records.flatMap(r => [r.message, r.timestamp]);
  await db.execute(query, params);
  return records.map(() => ({ inserted: true }));
}, 100);

// Usage
function logMessage(message) {
  return batchInsert({
    message,
    timestamp: Date.now()
  });
}

// Example 4: Image loading batching
const batchLoadImages = batch(async (imageUrls) => {
  console.log(`Loading ${imageUrls.length} images`);

  const promises = imageUrls.map(url =>
    new Promise((resolve, reject) => {
      const img = new Image();
      img.onload = () => resolve({ url, width: img.width, height: img.height });
      img.onerror = () => reject(new Error(`Failed to load ${url}`));
      img.src = url;
    })
  );

  return Promise.all(promises);
}, 5);

// Usage in image gallery
images.forEach(url => {
  batchLoadImages(url).then(imageData => {
    displayImage(imageData);
  });
});
```

10.1.3.1.11 11. batch(fn, limit) - Batch Multiple Calls

10.1.3.2 B. Caching & Optimization Utilities

These functions store results or intermediate data between calls to improve performance through memoization and caching strategies.

```
/**
 * Creates a memoized function that caches results based on input arguments.
 *
 * Use cases:
 * - Expensive computations
 * - Recursive functions (Fibonacci, factorial)
 * - API call caching
 */
function memoize(fn, resolver) {
  const cache = new Map();

  return function(...args) {
    const key = resolver ? resolver(...args) : JSON.stringify(args);

    if (cache.has(key)) {
      console.log('Cache hit for:', key);
      return cache.get(key);
    }

    console.log('Cache miss for:', key);
    const result = fn.apply(this, args);
    cache.set(key, result);
    return result;
  };
}

// Example 1: Memoize expensive calculation
const fibonacci = memoize((n) => {
  console.log(`Calculating fibonacci(${n})`);
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
});

console.log(fibonacci(10));  // Calculates many values
console.log(fibonacci(10));  // Returns cached result immediately
console.log(fibonacci(11));  // Only calculates fibonacci(11), rest cached

// Example 2: Memoize with custom resolver
const getUserData = memoize(
  async (userId, includeOrders) => {
    console.log(`Fetching user ${userId} with orders: ${includeOrders}`);
    const response = await fetch(`/api/users/${userId}?orders=${includeOrders}`);
```

```
    return response.json();
  },
  (userId, includeOrders) => `${userId}-${includeOrders}`  // Custom cache key
);

await getUserData(123, true);   // API call
await getUserData(123, true);   // Cached
await getUserData(123, false);  // Different key, new API call

// Example 3: Memoize DOM queries
const memoizedQuery = memoize(
  (selector) => {
    console.log('Querying DOM:', selector);
    return document.querySelectorAll(selector);
  },
  (selector) => selector
);

const buttons1 = memoizedQuery('.btn');  // Queries DOM
const buttons2 = memoizedQuery('.btn');  // Returns cached NodeList

// Example 4: Memoize with LRU cache
function memoizeWithLRU(fn, maxSize = 100) {
  const cache = new Map();

  return function(...args) {
    const key = JSON.stringify(args);

    if (cache.has(key)) {
      // Move to end (most recently used)
      const value = cache.get(key);
      cache.delete(key);
      cache.set(key, value);
      return value;
    }

    const result = fn.apply(this, args);

    // Add new entry
    cache.set(key, result);

    // Remove oldest if over limit
    if (cache.size > maxSize) {
      const firstKey = cache.keys().next().value;
      cache.delete(firstKey);
    }
```

```
    return result;
  };
}

// Usage
const expensiveOperation = memoizeWithLRU((x, y) => {
  console.log(`Computing ${x} * ${y}`);
  return x * y;
}, 50);  // Keep 50 most recent results

// Example 5: Memoize with TTL
function memoizeWithTTL(fn, ttlMs = 60000) {
  const cache = new Map();

  return function(...args) {
    const key = JSON.stringify(args);
    const now = Date.now();

    if (cache.has(key)) {
      const { value, expiry } = cache.get(key);
      if (now < expiry) {
        console.log('Cache hit (not expired)');
        return value;
      }
      console.log('Cache expired, refreshing...');
      cache.delete(key);
    }

    const result = fn.apply(this, args);
    cache.set(key, {
      value: result,
      expiry: now + ttlMs
    });

    return result;
  };
}

// Usage: Cache API responses for 5 minutes
const fetchWithCache = memoizeWithTTL(async (endpoint) => {
  console.log('Fetching:', endpoint);
  const response = await fetch(endpoint);
  return response.json();
}, 300000);
```

10.1.3.2.1 1. memoize(fn, resolver?) - Cache Function Results

```javascript
/**
 * Async version of once - executes only once and caches the promise.
 * Important: Caches the promise itself, not just the result.
 *
 * Use cases:
 * - One-time API initialization
 * - Singleton resource creation
 * - Configuration loading
 */
function onceAsync(fn) {
  let promise = null;
  let called = false;

  return function(...args) {
    if (!called) {
      called = true;
      promise = Promise.resolve(fn.apply(this, args));
    }
    return promise;
  };
}

// Example 1: Initialize API client once
const initializeAPI = onceAsync(async () => {
  console.log('Initializing API client...');

  // Expensive async operation
  const config = await fetch('/api/config').then(r => r.json());
  const auth = await authenticate();

  return {
    config,
    auth,
    client: createAPIClient(config, auth)
  };
});

// Multiple calls return the same promise
async function makeRequest1() {
  const api = await initializeAPI();  // Initializes
  return api.client.get('/data');
}

async function makeRequest2() {
  const api = await initializeAPI();  // Returns cached promise
  return api.client.get('/users');
```

```
}

// Example 2: Load configuration once
const loadConfig = onceAsync(async () => {
  console.log('Loading configuration...');

  const [appConfig, userPrefs, features] = await Promise.all([
    fetch('/api/config/app').then(r => r.json()),
    fetch('/api/config/user').then(r => r.json()),
    fetch('/api/features').then(r => r.json())
  ]);

  return { appConfig, userPrefs, features };
});

// Example 3: Database connection pool
const getDatabasePool = onceAsync(async () => {
  console.log('Creating database connection pool...');

  const pool = await createPool({
    host: 'localhost',
    database: 'myapp',
    max: 20,
    idleTimeoutMillis: 30000
  });

  // Test connection
  const client = await pool.connect();
  await client.query('SELECT NOW()');
  client.release();

  console.log('Database pool ready!');
  return pool;
});

// Example 4: Handle initialization errors
function onceAsyncWithErrorRetry(fn) {
  let promise = null;
  let called = false;
  let succeeded = false;

  return function(...args) {
    // If never called or previous attempt failed, try again
    if (!called || !succeeded) {
      called = true;
      promise = Promise.resolve(fn.apply(this, args))
        .then(result => {
```

```
          succeeded = true;
          return result;
        })
        .catch(error => {
          // Allow retry on next call
          called = false;
          throw error;
        });
    }
    return promise;
  };
}

// Usage
const connectWithRetry = onceAsyncWithErrorRetry(async () => {
  console.log('Attempting connection...');
  const response = await fetch('/api/connect');
  if (!response.ok) throw new Error('Connection failed');
  return response.json();
});

// If first call fails, second call will retry
try {
  await connectWithRetry();
} catch (error) {
  console.error('Failed, will retry...');
  await connectWithRetry();  // Retries instead of returning cached failure
}
```

10.1.3.2.2 2. onceAsync(fn) - Cache Async Function Result

```
/**
 * Similar to memoize but with custom key function and cache strategies.
 *
 * Use cases:
 * - Complex caching logic
 * - Multi-level caching
 * - Cache invalidation patterns
 */
function cache(fn, keyFn, options = {}) {
  const cacheStore = new Map();
  const { maxSize = Infinity, ttl = Infinity, onEvict = null } = options;

  return {
    execute: function(...args) {
      const key = keyFn(...args);
```

```
    const now = Date.now();

    // Check if cached and valid
    if (cacheStore.has(key)) {
      const entry = cacheStore.get(key);

      if (now - entry.timestamp < ttl) {
        console.log('Cache hit:', key);
        entry.hits++;
        entry.lastAccess = now;
        return entry.value;
      } else {
        console.log('Cache expired:', key);
        if (onEvict) onEvict(key, entry.value, 'expired');
        cacheStore.delete(key);
      }
    }

    // Compute and cache
    console.log('Cache miss:', key);
    const value = fn.apply(this, args);

    // Evict oldest if over size limit
    if (cacheStore.size >= maxSize) {
      const oldestKey = this.findLRU();
      const oldEntry = cacheStore.get(oldestKey);
      if (onEvict) onEvict(oldestKey, oldEntry.value, 'size-limit');
      cacheStore.delete(oldestKey);
    }

    cacheStore.set(key, {
      value,
      timestamp: now,
      lastAccess: now,
      hits: 0
    });

    return value;
  },

  findLRU: function() {
    let oldestKey = null;
    let oldestTime = Infinity;

    for (const [key, entry] of cacheStore.entries()) {
      if (entry.lastAccess < oldestTime) {
        oldestTime = entry.lastAccess;
```

```
          oldestKey = key;
        }
      }

      return oldestKey;
    },

    clear: function() {
      cacheStore.clear();
    },

    delete: function(key) {
      return cacheStore.delete(key);
    },

    size: function() {
      return cacheStore.size;
    },

    stats: function() {
      const entries = Array.from(cacheStore.entries());
      return {
        size: entries.length,
        totalHits: entries.reduce((sum, [, entry]) => sum + entry.hits, 0),
        entries: entries.map(([key, entry]) => ({
          key,
          hits: entry.hits,
          age: Date.now() - entry.timestamp
        }))
      };
    }
  };
}

// Example 1: User profile cache with TTL
const userCache = cache(
  (userId) => {
    console.log(`Fetching user ${userId} from database...`);
    return database.query('SELECT * FROM users WHERE id = ?', [userId]);
  },
  (userId) => `user:${userId}`,
  {
    maxSize: 100,
    ttl: 60000,  // 1 minute
    onEvict: (key, value, reason) => {
      console.log(`Evicted ${key} (reason: ${reason})`);
    }
```

```
  }
);

const user1 = userCache.execute(123);  // Database query
const user2 = userCache.execute(123);  // Cached
setTimeout(() => {
  const user3 = userCache.execute(123);  // Re-queries after TTL
}, 61000);

console.log('Cache stats:', userCache.stats());

// Example 2: Multi-parameter caching
const searchCache = cache(
  (query, filters, page) => {
    console.log(`Searching: ${query}, filters: ${JSON.stringify(filters)}, page: ${page
    return performSearch(query, filters, page);
  },
  (query, filters, page) => `search:${query}:${JSON.stringify(filters)}:${page}`,
  { maxSize: 50, ttl: 300000 }
);

// Example 3: Hierarchical cache invalidation
function createHierarchicalCache(fn, keyFn) {
  const cacheInstance = cache(fn, keyFn);
  const dependencies = new Map();  // key -> Set of dependent keys

  return {
    execute: cacheInstance.execute,

    addDependency: function(parentKey, childKey) {
      if (!dependencies.has(parentKey)) {
        dependencies.set(parentKey, new Set());
      }
      dependencies.get(parentKey).add(childKey);
    },

    invalidate: function(key) {
      // Invalidate key
      cacheInstance.delete(key);

      // Recursively invalidate dependents
      if (dependencies.has(key)) {
        for (const childKey of dependencies.get(key)) {
          this.invalidate(childKey);
        }
        dependencies.delete(key);
      }
```

```
    },

    stats: cacheInstance.stats
  };
}
```

10.1.3.2.3 3. cache(fn, keyFn) - Custom Cache Strategy

```
/**
 * Executes function only once when first accessed, stores result.
 * Different from once - returns a value, not a function.
 *
 * Use cases:
 * - Lazy initialization
 * - Deferred expensive computations
 * - Circular dependency resolution
 */
function lazy(fn) {
  let computed = false;
  let value;

  return function() {
    if (!computed) {
      console.log('Computing lazy value...');
      value = fn();
      computed = true;
    }
    return value;
  };
}

// Example 1: Lazy module loading
const heavyModule = lazy(() => {
  console.log('Loading heavy module...');
  // Expensive module initialization
  return {
    process: (data) => {
      console.log('Processing with heavy module:', data);
      return data.map(x => x * 2);
    },
    config: { version: '1.0.0' }
  };
});

// Module not loaded until first use
console.log('App started');
```

```
// ... later when needed ...
const module = heavyModule();  // Loads now
module.process([1, 2, 3]);

const module2 = heavyModule();  // Returns same instance
console.log(module === module2);  // true

// Example 2: Lazy configuration
const appConfig = lazy(() => {
  console.log('Loading application configuration...');
  return {
    apiUrl: process.env.API_URL || 'https://api.example.com',
    timeout: parseInt(process.env.TIMEOUT) || 5000,
    features: {
      darkMode: true,
      analytics: true
    }
  };
});

// Config not loaded until accessed
function makeAPICall(endpoint) {
  const config = appConfig();  // Loaded on first API call
  return fetch(`${config.apiUrl}${endpoint}`, {
    timeout: config.timeout
  });
}

// Example 3: Lazy expensive calculation
const primeNumbers = lazy(() => {
  console.log('Calculating first 10000 primes...');
  const primes = [];
  let num = 2;

  while (primes.length < 10000) {
    if (isPrime(num)) primes.push(num);
    num++;
  }

  return primes;
});

// Only calculated when needed
function findNthPrime(n) {
  const primes = primeNumbers();  // Expensive calculation happens here
  return primes[n - 1];
}
```

```javascript
// Example 4: Lazy with dependencies
function lazyWithDeps(...deps) {
  return (fn) => {
    let computed = false;
    let value;

    return function() {
      if (!computed) {
        // Resolve all dependencies first
        const resolvedDeps = deps.map(dep =>
          typeof dep === 'function' ? dep() : dep
        );
        value = fn(...resolvedDeps);
        computed = true;
      }
      return value;
    };
  };
}

// Usage
const database = lazy(() => createDatabase());
const userService = lazyWithDeps(database)((db) => {
  console.log('Creating user service...');
  return createUserService(db());
});
const authService = lazyWithDeps(database, userService)((db, users) => {
  console.log('Creating auth service...');
  return createAuthService(db(), users());
});
```

10.1.3.2.4 4. lazy(fn) - Lazy Evaluation

```javascript

/**
 * Initializes a resource once and returns the same instance.
 * Thread-safe singleton pattern with proper error handling.
 *
 * Use cases:
 * - Singleton services
 * - Resource pools
 * - Global state management
 */
function initOnce(factory, options = {}) {
  let instance = null;
  let initializing = false;
  let initPromise = null;
```

```
  let error = null;
  const { onError = null, allowRetry = true } = options;

  return {
    getInstance: async function() {
      // Return existing instance
      if (instance) {
        return instance;
      }

      // Return error if failed and retry not allowed
      if (error && !allowRetry) {
        throw error;
      }

      // Wait if currently initializing
      if (initializing) {
        return initPromise;
      }

      // Initialize
      initializing = true;
      initPromise = (async () => {
        try {
          console.log('Initializing singleton...');
          instance = await factory();
          error = null;
          console.log('Singleton initialized successfully');
          return instance;
        } catch (err) {
          error = err;
          if (onError) onError(err);
          throw err;
        } finally {
          initializing = false;
          initPromise = null;
        }
      })();

      return initPromise;
    },

    isInitialized: function() {
      return instance !== null;
    },

    reset: function() {
```

```
      instance = null;
      error = null;
      initializing = false;
      initPromise = null;
    },

    getError: function() {
      return error;
    }
  };
}

// Example 1: Database connection singleton
const dbConnection = initOnce(
  async () => {
    console.log('Connecting to database...');
    const conn = await database.connect({
      host: 'localhost',
      database: 'myapp'
    });

    // Test connection
    await conn.query('SELECT 1');

    return conn;
  },
  {
    allowRetry: true,
    onError: (error) => {
      console.error('Database connection failed:', error);
      notifyAdmin('Database connection failed');
    }
  }
);

// Usage across application
async function getUsers() {
  const db = await dbConnection.getInstance();  // Initializes on first call
  return db.query('SELECT * FROM users');
}

async function getProducts() {
  const db = await dbConnection.getInstance();  // Returns same instance
  return db.query('SELECT * FROM products');
}

// Example 2: Logger singleton
```

```
const logger = initOnce(async () => {
  console.log('Initializing logger...');

  const config = await loadLoggerConfig();
  const transport = await createTransport(config);

  return {
    log: (level, message, meta) => {
      transport.write({ level, message, meta, timestamp: Date.now() });
    },
    info: (message, meta) => this.log('info', message, meta),
    error: (message, meta) => this.log('error', message, meta),
    warn: (message, meta) => this.log('warn', message, meta)
  };
});

// Example 3: Feature flags singleton
const featureFlags = initOnce(async () => {
  console.log('Loading feature flags...');

  const response = await fetch('/api/feature-flags');
  const flags = await response.json();

  return {
    isEnabled: (flagName) => flags[flagName] === true,
    get: (flagName) => flags[flagName],
    refresh: async () => {
      const response = await fetch('/api/feature-flags');
      const newFlags = await response.json();
      Object.assign(flags, newFlags);
    }
  };
});

// Example 4: Cache manager singleton with warm-up
const cacheManager = initOnce(async () => {
  console.log('Initializing cache manager...');

  const cache = new Map();

  // Warm up cache with frequently accessed data
  const warmUpData = await fetch('/api/cache/warmup').then(r => r.json());
  warmUpData.forEach(({ key, value }) => cache.set(key, value));

  console.log(`Cache warmed up with ${cache.size} entries`);

  return {
```

```
    get: (key) => cache.get(key),
    set: (key, value) => cache.set(key, value),
    has: (key) => cache.has(key),
    delete: (key) => cache.delete(key),
    clear: () => cache.clear(),
    size: () => cache.size
  };
});
```

10.1.3.2.5 5. initOnce(factory) - Singleton Initialization

```
/**
 * Caches function output with time-to-live expiration.
 * Automatically refreshes stale cache entries.
 *
 * Use cases:
 * - API response caching
 * - Temporary data storage
 * - Rate limit optimization
 */
function withCache(fn, ttl, options = {}) {
  const cache = new Map();
  const {
    refreshThreshold = 0,  // Refresh if within this ms of expiry
    onCacheHit = null,
    onCacheMiss = null,
    onRefresh = null
  } = options;

  return async function(...args) {
    const key = JSON.stringify(args);
    const now = Date.now();

    if (cache.has(key)) {
      const { value, expiry, loading } = cache.get(key);

      // Return cached value if not expired
      if (now < expiry) {
        if (onCacheHit) onCacheHit(key, value);

        // Background refresh if near expiry
        if (refreshThreshold > 0 && expiry - now < refreshThreshold && !loading) {
          console.log(`Background refresh for ${key}`);
          this.refreshCache(key, args);
        }
```

```
        return value;
      }

      // Cache expired
      console.log(`Cache expired for ${key}`);
    }

    // Cache miss or expired - fetch new value
    if (onCacheMiss) onCacheMiss(key);

    // Mark as loading to prevent duplicate requests
    cache.set(key, { loading: true, expiry: now + ttl });

    try {
      const value = await fn.apply(this, args);
      cache.set(key, {
        value,
        expiry: now + ttl,
        loading: false
      });
      return value;
    } catch (error) {
      cache.delete(key);
      throw error;
    }
  };

  async function refreshCache(key, args) {
    const entry = cache.get(key);
    entry.loading = true;

    try {
      const value = await fn.apply(this, args);
      cache.set(key, {
        value,
        expiry: Date.now() + ttl,
        loading: false
      });
      if (onRefresh) onRefresh(key, value);
    } catch (error) {
      entry.loading = false;
      console.error('Background refresh failed:', error);
    }
  }
}

// Example 1: Cache API responses
```

```
const fetchUserWithCache = withCache(
  async (userId) => {
    console.log(`Fetching user ${userId} from API...`);
    const response = await fetch(`/api/users/${userId}`);
    return response.json();
  },
  60000,  // Cache for 1 minute
  {
    refreshThreshold: 10000,  // Refresh if within 10s of expiry
    onCacheHit: (key) => console.log('Cache hit:', key),
    onCacheMiss: (key) => console.log('Cache miss:', key),
    onRefresh: (key) => console.log('Background refresh completed:', key)
  }
);

// Example 2: Cache expensive calculations
const calculateWithCache = withCache(
  (data) => {
    console.log('Performing expensive calculation...');
    return data.reduce((sum, val) => sum + Math.pow(val, 2), 0);
  },
  30000  // Cache for 30 seconds
);

const result1 = calculateWithCache([1, 2, 3, 4, 5]);  // Computes
const result2 = calculateWithCache([1, 2, 3, 4, 5]);  // Cached

// Example 3: Multi-level cache
function createMultiLevelCache(fn, l1TTL, l2TTL) {
  const l1Cache = new Map();  // Fast, short-lived
  const l2Cache = new Map();  // Slower, long-lived

  return async function(...args) {
    const key = JSON.stringify(args);
    const now = Date.now();

    // Check L1 cache
    if (l1Cache.has(key)) {
      const { value, expiry } = l1Cache.get(key);
      if (now < expiry) {
        console.log('L1 cache hit');
        return value;
      }
      l1Cache.delete(key);
    }

    // Check L2 cache
```

```
    if (l2Cache.has(key)) {
      const { value, expiry } = l2Cache.get(key);
      if (now < expiry) {
        console.log('L2 cache hit');
        // Promote to L1
        l1Cache.set(key, { value, expiry: now + l1TTL });
        return value;
      }
      l2Cache.delete(key);
    }

    // Fetch new value
    console.log('Cache miss, fetching...');
    const value = await fn.apply(this, args);

    // Store in both caches
    l1Cache.set(key, { value, expiry: now + l1TTL });
    l2Cache.set(key, { value, expiry: now + l2TTL });

    return value;
  };
}

// Usage: L1 cache for 30s, L2 cache for 5 minutes
const cachedFetch = createMultiLevelCache(
  (url) => fetch(url).then(r => r.json()),
  30000,   // L1: 30 seconds
  300000   // L2: 5 minutes
);
```

10.1.3.2.6 6. withCache(fn, ttl) - TTL-based Caching

10.1.3.3 C. Function Transformation Utilities

These functions create new functions with transformed behavior, enabling functional programming patterns like currying, composition, and function wrapping.

```
/**
 * Transforms a function to accept arguments one at a time.
 * Returns a new function until all arguments are provided.
 *
 * Use cases:
 * - Partial application
 * - Creating specialized functions
 * - Function composition
 */
```

```javascript
function curry(fn) {
  return function curried(...args) {
    if (args.length >= fn.length) {
      return fn.apply(this, args);
    } else {
      return function(...nextArgs) {
        return curried.apply(this, args.concat(nextArgs));
      };
    }
  };
}

// Example 1: Basic currying
const add = curry((a, b, c) => a + b + c);

console.log(add(1)(2)(3));        // 6
console.log(add(1, 2)(3));        // 6
console.log(add(1)(2, 3));        // 6

const add5 = add(5);              // Partially applied
console.log(add5(2)(3));          // 10
console.log(add5(10, 15));        // 30

// Example 2: Curried utility functions
const map = curry((fn, array) => array.map(fn));
const filter = curry((fn, array) => array.filter(fn));
const reduce = curry((fn, init, array) => array.reduce(fn, init));

const double = x => x * 2;
const isEven = x => x % 2 === 0;
const sum = (acc, val) => acc + val;

// Create specialized functions
const doubleAll = map(double);
const filterEvens = filter(isEven);
const sumAll = reduce(sum, 0);

const numbers = [1, 2, 3, 4, 5, 6];
console.log(doubleAll(numbers));        // [2, 4, 6, 8, 10, 12]
console.log(filterEvens(numbers));      // [2, 4, 6]
console.log(sumAll(numbers));           // 21

// Example 3: Curried DOM manipulation
const setAttribute = curry((attr, value, element) => {
  element.setAttribute(attr, value);
  return element;
});
```

```
const addClass = curry((className, element) => {
  element.classList.add(className);
  return element;
});

// Create specialized setters
const setDataId = setAttribute('data-id');
const setAriaLabel = setAttribute('aria-label');
const addActiveClass = addClass('active');

const button = document.querySelector('button');
setDataId('btn-1', button);
setAriaLabel('Submit button', button);
addActiveClass(button);

// Example 4: Advanced curry with placeholders
function curryWithPlaceholder(fn, placeholder = curry.placeholder) {
  return function curried(...args) {
    const hasPlaceholder = args.some(arg => arg === placeholder);

    if (args.length >= fn.length && !hasPlaceholder) {
      return fn.apply(this, args);
    }

    return function(...nextArgs) {
      const newArgs = args.map(arg =>
        arg === placeholder && nextArgs.length ? nextArgs.shift() : arg
      );
      return curried.apply(this, [...newArgs, ...nextArgs]);
    };
  };
}

curryWithPlaceholder.placeholder = Symbol('placeholder');
const _ = curryWithPlaceholder.placeholder;

const divide = curryWithPlaceholder((a, b, c) => a / b / c);
const divideBy2 = divide(_, 2);
console.log(divideBy2(100, 5));  // 10 (100 / 2 / 5)
```

10.1.3.3.1 1. curry(fn) - Transform to Curried Function

```
/**
 * Pre-fills some arguments of a function, returning a new function.
 * Different from curry - applies multiple args at once.
 *
```

```
 * Use cases:
 * - Event handlers with extra parameters
 * - Configuration functions
 * - Callback customization
 */
function partial(fn, ...fixedArgs) {
  return function(...remainingArgs) {
    return fn.apply(this, [...fixedArgs, ...remainingArgs]);
  };
}

// Example 1: Event handlers
function logEvent(level, category, message, event) {
  console.log(`[${level}] ${category}: ${message}`, event);
}

const logUserAction = partial(logEvent, 'INFO', 'USER');
const logError = partial(logEvent, 'ERROR', 'SYSTEM');

button.addEventListener('click', partial(logUserAction, 'Button clicked'));
input.addEventListener('error', partial(logError, 'Input validation failed'));

// Example 2: API calls with base configuration
async function apiCall(baseURL, auth, endpoint, options = {}) {
  const response = await fetch(`${baseURL}${endpoint}`, {
    ...options,
    headers: {
      'Authorization': auth,
      ...options.headers
    }
  });
  return response.json();
}

const callAPI = partial(apiCall, 'https://api.example.com', 'Bearer token123');

// Now easily make authenticated calls
const users = await callAPI('/users');
const posts = await callAPI('/posts', { method: 'POST', body: JSON.stringify({}) });

// Example 3: Partial from right
function partialRight(fn, ...fixedArgs) {
  return function(...remainingArgs) {
    return fn.apply(this, [...remainingArgs, ...fixedArgs]);
  };
}
```

```
function greet(greeting, name, punctuation) {
  return `${greeting}, ${name}${punctuation}`;
}

const greetWithExclamation = partialRight(greet, '!');
console.log(greetWithExclamation('Hello', 'Alice'));  // "Hello, Alice!"
console.log(greetWithExclamation('Hi', 'Bob'));       // "Hi, Bob!"

// Example 4: Partial with placeholder support
function partialWithPlaceholder(fn, ...args) {
  const placeholder = partialWithPlaceholder.placeholder;

  return function(...newArgs) {
    const finalArgs = args.map(arg =>
      arg === placeholder && newArgs.length ? newArgs.shift() : arg
    );
    return fn.apply(this, [...finalArgs, ...newArgs]);
  };
}

partialWithPlaceholder.placeholder = Symbol('_');
const _ = partialWithPlaceholder.placeholder;

function calculate(a, b, c, d) {
  return (a + b) * (c - d);
}

const calc = partialWithPlaceholder(calculate, 10, _, _, 5);
console.log(calc(20, 15));  // (10 + 20) * (15 - 5) = 300
```

10.1.3.3.2 2. partial(fn, ...args) - Partial Application

```

/**
 * Composes functions right-to-left.
 * compose(f, g, h)(x) = f(g(h(x)))
 *
 * Use cases:
 * - Data transformation pipelines
 * - Middleware chains
 * - Functional programming patterns
 */
function compose(...fns) {
  return function(initialValue) {
    return fns.reduceRight((acc, fn) => fn(acc), initialValue);
  };
}
```

```javascript
// Example 1: Data transformation
const trim = str => str.trim();
const lowercase = str => str.toLowerCase();
const removeSpaces = str => str.replace(/\s+/g, '-');
const addPrefix = str => `user-${str}`;

const createSlug = compose(addPrefix, removeSpaces, lowercase, trim);

console.log(createSlug('  John Doe  '));  // "user-john-doe"

// Example 2: Array transformations
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

const filterEvens = arr => arr.filter(x => x % 2 === 0);
const double = arr => arr.map(x => x * 2);
const sum = arr => arr.reduce((a, b) => a + b, 0);

const processNumbers = compose(sum, double, filterEvens);

console.log(processNumbers(numbers));  // 60 (2+4+6+8+10 doubled)

// Example 3: Async compose
function composeAsync(...fns) {
  return function(initialValue) {
    return fns.reduceRight(
      (acc, fn) => acc.then(fn),
      Promise.resolve(initialValue)
    );
  };
}

const fetchUser = async (id) => {
  const response = await fetch(`/api/users/${id}`);
  return response.json();
};

const addTimestamp = async (user) => ({
  ...user,
  fetchedAt: Date.now()
});

const validateUser = async (user) => {
  if (!user.email) throw new Error('Invalid user');
  return user;
};

const enrichUser = composeAsync(validateUser, addTimestamp, fetchUser);
```

```
// Usage
try {
  const user = await enrichUser(123);
  console.log(user);
} catch (error) {
  console.error('User processing failed:', error);
}

// Example 4: Middleware pattern
function createMiddleware(...middlewares) {
  return compose(...middlewares.reverse());
}

const logRequest = (next) => (req) => {
  console.log('Request:', req.url);
  return next(req);
};

const authenticate = (next) => (req) => {
  if (!req.headers.auth) {
    throw new Error('Unauthorized');
  }
  return next(req);
};

const parseJSON = (next) => (req) => {
  if (req.body) {
    req.body = JSON.parse(req.body);
  }
  return next(req);
};

const handleRequest = (req) => {
  console.log('Handling:', req);
  return { success: true };
};

const middleware = createMiddleware(logRequest, authenticate, parseJSON);
const processRequest = middleware(handleRequest);
```

10.1.3.3.3 3. compose(...fns) - Right-to-Left Composition

```
/**
 * Composes functions left-to-right (reverse of compose).
 * pipe(f, g, h)(x) = h(g(f(x)))
 *
```

```
 * Use cases:
 * - More intuitive data flow
 * - Sequential transformations
 * - Processing pipelines
 */
function pipe(...fns) {
  return function(initialValue) {
    return fns.reduce((acc, fn) => fn(acc), initialValue);
  };
}

// Example 1: Sequential data processing
const trim = str => str.trim();
const uppercase = str => str.toUpperCase();
const addExclamation = str => `${str}!`;
const repeat = n => str => str.repeat(n);

const shout = pipe(
  trim,
  uppercase,
  addExclamation,
  repeat(3)
);

console.log(shout('  hello  '));  // "HELLO!HELLO!HELLO!"

// Example 2: Form validation pipeline
const validateLength = min => value => {
  if (value.length < min) {
    throw new Error(`Minimum length is ${min}`);
  }
  return value;
};

const validateEmail = value => {
  if (!value.includes('@')) {
    throw new Error('Invalid email');
  }
  return value;
};

const normalizeEmail = value => value.toLowerCase().trim();

const validateEmailInput = pipe(
  normalizeEmail,
  validateLength(5),
  validateEmail
```

```
);

try {
  const email = validateEmailInput('  John@Example.COM  ');
  console.log('Valid email:', email);  // "john@example.com"
} catch (error) {
  console.error('Validation error:', error.message);
}

// Example 3: Async pipe
function pipeAsync(...fns) {
  return function(initialValue) {
    return fns.reduce(
      (acc, fn) => acc.then(fn),
      Promise.resolve(initialValue)
    );
  };
}

const fetchData = async (url) => {
  const response = await fetch(url);
  return response.json();
};

const extractItems = data => data.items || [];
const filterActive = items => items.filter(item => item.active);
const sortByDate = items => items.sort((a, b) => b.date - a.date);
const take = n => items => items.slice(0, n);

const getRecentActive = pipeAsync(
  fetchData,
  extractItems,
  filterActive,
  sortByDate,
  take(5)
);

const recentItems = await getRecentActive('/api/items');

// Example 4: Tap and log in pipeline
const tap = fn => value => {
  fn(value);
  return value;
};

const log = label => tap(value => console.log(`[${label}]`, value));
```

```
const processData = pipe(
  log('Input'),
  trim,
  log('After trim'),
  uppercase,
  log('After uppercase'),
  addExclamation,
  log('Final output')
);

processData('  test  ');
```

10.1.3.3.4 4. pipe(...fns) - Left-to-Right Composition

```
/**
 * Wraps a function with additional behavior while preserving original.
 * Allows adding cross-cutting concerns like logging, timing, etc.
 *
 * Use cases:
 * - Adding logging/monitoring
 * - Performance measurement
 * - Error handling
 * - Caching/memoization
 */
function wrap(fn, wrapper) {
  return function(...args) {
    return wrapper.call(this, fn, ...args);
  };
}

// Example 1: Add logging to function
function add(a, b) {
  return a + b;
}

const addWithLogging = wrap(add, (originalFn, a, b) => {
  console.log(`Calling add with: ${a}, ${b}`);
  const result = originalFn(a, b);
  console.log(`Result: ${result}`);
  return result;
});

console.log(addWithLogging(5, 3));
// Logs: Calling add with: 5, 3
// Logs: Result: 8
// Returns: 8
```

```javascript
// Example 2: Performance measurement
const measurePerformance = (fn, ...args) => {
  const start = performance.now();
  const result = fn(...args);
  const end = performance.now();
  console.log(`Execution time: ${(end - start).toFixed(2)}ms`);
  return result;
};

const expensiveCalculation = (n) => {
  let sum = 0;
  for (let i = 0; i < n; i++) {
    sum += Math.sqrt(i);
  }
  return sum;
};

const measuredCalculation = wrap(expensiveCalculation, measurePerformance);
measuredCalculation(1000000);  // Logs execution time

// Example 3: Add error handling
const withErrorHandling = (fn, ...args) => {
  try {
    return fn(...args);
  } catch (error) {
    console.error(`Error in ${fn.name}:`, error.message);
    return null;
  }
};

const riskyFunction = (data) => {
  if (!data) throw new Error('No data provided');
  return data.value;
};

const safeFunction = wrap(riskyFunction, withErrorHandling);
console.log(safeFunction(null));  // Logs error, returns null
console.log(safeFunction({ value: 42 }));  // Returns 42

// Example 4: Retry wrapper
const withRetry = (fn, ...args) => {
  const maxRetries = 3;
  for (let attempt = 0; attempt < maxRetries; attempt++) {
    try {
      return fn(...args);
    } catch (error) {
      if (attempt === maxRetries - 1) throw error;
```

```
      console.log(`Attempt ${attempt + 1} failed, retrying...`);
    }
  }
};

const flakyFunction = () => {
  if (Math.random() < 0.7) throw new Error('Random failure');
  return 'Success!';
};

const reliableFunction = wrap(flakyFunction, withRetry);

// Example 5: Caching wrapper
const withCache = (fn, ...args) => {
  if (!withCache.cache) {
    withCache.cache = new Map();
  }

  const key = JSON.stringify(args);
  if (withCache.cache.has(key)) {
    console.log('Cache hit!');
    return withCache.cache.get(key);
  }

  console.log('Cache miss, computing...');
  const result = fn(...args);
  withCache.cache.set(key, result);
  return result;
};

const slowFunction = (n) => {
  let sum = 0;
  for (let i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
};

const cachedFunction = wrap(slowFunction, withCache);
cachedFunction(1000);  // Computes
cachedFunction(1000);  // Returns from cache
```

10.1.3.3.5 5. wrap(fn, wrapper) - Function Wrapping

```
/**
 * Returns a function with reversed argument order.
```

```
 * Useful for adapting functions to different calling conventions.
 *
 * Use cases:
 * - Adapting library functions
 * - Point-free programming
 * - Functional composition
 */
function flip(fn) {
  return function(...args) {
    return fn.apply(this, args.reverse());
  };
}

// Example 1: Basic flip
const divide = (a, b) => a / b;
const flippedDivide = flip(divide);

console.log(divide(10, 2));         // 5
console.log(flippedDivide(10, 2));  // 0.2 (2 / 10)

// Example 2: Array operations
const append = (arr, item) => [...arr, item];
const prepend = flip(append);  // (item, arr) => [item, ...arr]

const numbers = [2, 3, 4];
console.log(append(numbers, 5));   // [2, 3, 4, 5]
console.log(prepend(numbers, 1));  // [1, 2, 3, 4]

// Example 3: Curried operations
const subtract = curry((a, b) => a - b);
const subtractFrom = flip(subtract);

const subtract5 = subtract(5);      // 5 - x
const from5 = subtractFrom(5);      // x - 5

console.log(subtract5(3));   // 2 (5 - 3)
console.log(from5(3));       // -2 (3 - 5)

// Example 4: Object property access
const getProp = (obj, key) => obj[key];
const getFromKey = flip(getProp);  // (key, obj) => obj[key]

const user = { name: 'Alice', age: 30 };
console.log(getProp(user, 'name'));      // "Alice"
console.log(getFromKey('name', user));   // "Alice"

// Useful for point-free style
```

```
const users = [
  { name: 'Alice', age: 30 },
  { name: 'Bob', age: 25 }
];

const getNames = users.map(getFromKey('name'));
console.log(getNames);  // Won't work directly

// Better approach with curry and flip
const prop = curry(flip(getProp));
const getName = prop('name');
console.log(users.map(getName));  // ['Alice', 'Bob']
```

10.1.3.3.6 6. flip(fn) - Reverse Argument Order

```
/**
 * Returns a function that negates the boolean result of fn.
 *
 * Use cases:
 * - Creating opposite predicates
 * - Filter inversions
 * - Validation logic
 */
function negate(fn) {
  return function(...args) {
    return !fn.apply(this, args);
  };
}

// Example 1: Filter predicates
const isEven = x => x % 2 === 0;
const isOdd = negate(isEven);

const numbers = [1, 2, 3, 4, 5, 6];
console.log(numbers.filter(isEven));  // [2, 4, 6]
console.log(numbers.filter(isOdd));   // [1, 3, 5]

// Example 2: Validation
const isEmpty = str => str.length === 0;
const isNotEmpty = negate(isEmpty);

const validateInput = (value) => {
  if (isNotEmpty(value)) {
    console.log('Valid input');
  } else {
    console.log('Input is empty');
```

```
  }
};

// Example 3: Array operations
const includes = curry((item, arr) => arr.includes(item));
const excludes = value => negate(includes(value));

const allowedTags = ['div', 'span', 'p'];
const isAllowed = includes(_, allowedTags);
const isForbidden = excludes(_, allowedTags);

console.log(isAllowed('div'));     // true
console.log(isForbidden('script')); // true

// Example 4: User permissions
const hasPermission = (user, permission) => {
  return user.permissions.includes(permission);
};

const lacksPermission = negate(hasPermission);

const user = { permissions: ['read', 'write'] };
console.log(hasPermission(user, 'read'));      // true
console.log(lacksPermission(user, 'delete'));  // true
```

10.1.3.3.7 7. negate(fn) - Logical Negation

```
/**
 * Executes fn only if predicate returns true.
 * Returns undefined if predicate fails.
 *
 * Use cases:
 * - Conditional logic
 * - Validation gates
 * - Permission checks
 */
function guard(fn, predicate) {
  return function(...args) {
    if (predicate.apply(this, args)) {
      return fn.apply(this, args);
    }
    return undefined;
  };
}

// Example 1: Age-gated function
```

```
const buyAlcohol = guard(
  (user) => {
    console.log(`Selling alcohol to ${user.name}`);
    return { success: true };
  },
  (user) => user.age >= 21
);

const user1 = { name: 'Alice', age: 25 };
const user2 = { name: 'Bob', age: 18 };

console.log(buyAlcohol(user1));  // Success
console.log(buyAlcohol(user2));  // undefined

// Example 2: Permission-based operations
const deleteUser = guard(
  (userId) => {
    console.log(`Deleting user ${userId}`);
    database.delete('users', userId);
  },
  (userId) => {
    const currentUser = getCurrentUser();
    return currentUser.role === 'admin';
  }
);

// Example 3: Input validation
const processPayment = guard(
  (amount, account) => {
    account.balance -= amount;
    console.log(`Processed payment of $${amount}`);
    return { success: true, newBalance: account.balance };
  },
  (amount, account) => {
    return amount > 0 && account.balance >= amount;
  }
);

const account = { balance: 100 };
console.log(processPayment(50, account));   // Success
console.log(processPayment(200, account));  // undefined (insufficient funds)

// Example 4: Rate limiting
const createRateLimitedFunction = (fn, maxCalls, windowMs) => {
  const calls = [];

  return guard(
```

```
    fn,
    () => {
      const now = Date.now();
      // Remove old calls
      while (calls.length && calls[0] < now - windowMs) {
        calls.shift();
      }

      if (calls.length < maxCalls) {
        calls.push(now);
        return true;
      }

      console.log('Rate limit exceeded');
      return false;
    }
  );
};

const limitedAPI = createRateLimitedFunction(
  () => fetch('/api/data'),
  5,    // 5 calls
  60000 // per minute
);
```

10.1.3.3.8 8. guard(fn, predicate) - Conditional Execution

```
/**
 * Executes fn for side effects, returns original value.
 * Useful for debugging and logging in pipelines.
 *
 * Use cases:
 * - Logging in pipelines
 * - Debugging transformations
 * - Side effect injection
 */
function tap(fn) {
  return function(value) {
    fn(value);
    return value;
  };
}

// Example 1: Debugging pipeline
const processData = pipe(
  tap(x => console.log('Input:', x)),
```

```
  x => x.trim(),
  tap(x => console.log('After trim:', x)),
  x => x.toUpperCase(),
  tap(x => console.log('After uppercase:', x)),
  x => x.split(''),
  tap(x => console.log('After split:', x))
);

const result = processData('  hello  ');

// Example 2: Logging with labels
const log = label => tap(value => {
  console.log(`[${label}]`, value);
});

const transform = pipe(
  log('Start'),
  x => x * 2,
  log('After double'),
  x => x + 10,
  log('After add'),
  x => x.toString(),
  log('Final')
);

// Example 3: Analytics tracking
const trackEvent = tap(data => {
  analytics.track('data_processed', {
    timestamp: Date.now(),
    data: JSON.stringify(data)
  });
});

const processWithTracking = pipe(
  trackEvent,
  transformData,
  trackEvent,
  saveToDatabase
);

// Example 4: Cache warming
const warmCache = tap(result => {
  cache.set(cacheKey, result);
  console.log('Cache warmed with result');
});

const fetchWithCacheWarming = pipe(
```

```
  fetchFromAPI,
  warmCache,
  transformResponse
);

// Example 5: Mutation for side effects
const updateUI = tap(data => {
  document.getElementById('result').textContent = JSON.stringify(data);
});

const displayResult = pipe(
  fetchData,
  processData,
  updateUI,  // Updates UI but returns data
  saveToLocalStorage
);
```

10.1.3.3.9 9. tap(fn) - Side Effects Without Modification

10.1.3.4 D. State & Encapsulation Utilities

These functions use closures to maintain private state, providing encapsulation and data hiding capabilities.

```

/**
 * Creates a counter with private state.
 * Demonstrates data encapsulation through closures.
 *
 * Use cases:
 * - ID generation
 * - Statistics tracking
 * - State management
 */
function counter(initial = 0) {
  let count = initial;

  return {
    increment: (by = 1) => {
      count += by;
      return count;
    },
    decrement: (by = 1) => {
      count -= by;
      return count;
    },
    get: () => count,
```

```
    reset: () => {
      count = initial;
      return count;
    },
    set: (value) => {
      count = value;
      return count;
    }
  };
}

// Example 1: Page view counter
const pageViews = counter(0);

// Track page views
window.addEventListener('load', () => {
  const views = pageViews.increment();
  console.log(`Page views: ${views}`);
});

// Example 2: Like counter with persistence
function createLikeCounter(postId, initialCount = 0) {
  const likeCtr = counter(initialCount);

  return {
    like: async () => {
      const newCount = likeCtr.increment();
      await fetch(`/api/posts/${postId}/like`, { method: 'POST' });
      updateUI(newCount);
      return newCount;
    },
    unlike: async () => {
      const newCount = likeCtr.decrement();
      await fetch(`/api/posts/${postId}/unlike`, { method: 'POST' });
      updateUI(newCount);
      return newCount;
    },
    getCount: () => likeCtr.get()
  };
}

// Example 3: Statistics tracker
function createStatsTracker() {
  const errors = counter(0);
  const successes = counter(0);
  const total = counter(0);
```

```
  return {
    recordSuccess: () => {
      successes.increment();
      total.increment();
    },
    recordError: () => {
      errors.increment();
      total.increment();
    },
    getStats: () => ({
      errors: errors.get(),
      successes: successes.get(),
      total: total.get(),
      successRate: (successes.get() / total.get() * 100).toFixed(2) + '%'
    }),
    reset: () => {
      errors.reset();
      successes.reset();
      total.reset();
    }
  };
}

const apiStats = createStatsTracker();

async function makeAPICall(endpoint) {
  try {
    const response = await fetch(endpoint);
    if (response.ok) {
      apiStats.recordSuccess();
    } else {
      apiStats.recordError();
    }
    return response;
  } catch (error) {
    apiStats.recordError();
    throw error;
  }
}

console.log(apiStats.getStats());  // { errors: 0, successes: 5, total: 5, successRate:
```

10.1.3.4.1 1. counter(initial) - Stateful Counter

```
/**
 * Creates a function that cycles through multiple states.
```

```
 * Each call returns the next state in the sequence.
 *
 * Use cases:
 * - Theme switching
 * - View mode toggling
 * - Carousel/slideshow navigation
 */
function toggle(...states) {
  let index = 0;

  return function() {
    const current = states[index];
    index = (index + 1) % states.length;
    return current;
  };
}

// Example 1: Theme toggler
const themeToggle = toggle('light', 'dark', 'auto');

button.addEventListener('click', () => {
  const theme = themeToggle();
  document.body.className = `theme-${theme}`;
  console.log(`Theme changed to: ${theme}`);
});

// Example 2: View mode toggle
const viewModeToggle = toggle('grid', 'list', 'compact');

viewButton.addEventListener('click', () => {
  const mode = viewModeToggle();
  gallery.className = `view-${mode}`;
  localStorage.setItem('viewMode', mode);
});

// Example 3: Sort order toggle
const sortToggle = toggle('asc', 'desc');

function sortTable(column) {
  const order = sortToggle();
  const rows = Array.from(table.querySelectorAll('tr'));

  rows.sort((a, b) => {
    const aVal = a.querySelector(`td:nth-child(${column})`).textContent;
    const bVal = b.querySelector(`td:nth-child(${column})`).textContent;

    if (order === 'asc') {
```

```
      return aVal.localeCompare(bVal);
    } else {
      return bVal.localeCompare(aVal);
    }
  });

  rows.forEach(row => table.appendChild(row));
}

// Example 4: Advanced toggle with state object
function createAdvancedToggle(...states) {
  let index = 0;

  return {
    next: () => {
      const current = states[index];
      index = (index + 1) % states.length;
      return current;
    },
    prev: () => {
      index = (index - 1 + states.length) % states.length;
      return states[index];
    },
    current: () => states[index],
    goto: (targetState) => {
      const targetIndex = states.indexOf(targetState);
      if (targetIndex !== -1) {
        index = targetIndex;
        return states[index];
      }
      throw new Error(`State "${targetState}" not found`);
    },
    reset: () => {
      index = 0;
      return states[index];
    }
  };
}

const carousel = createAdvancedToggle('slide1', 'slide2', 'slide3', 'slide4');
console.log(carousel.next());     // 'slide1'
console.log(carousel.next());     // 'slide2'
console.log(carousel.prev());     // 'slide1'
console.log(carousel.goto('slide4'));  // 'slide4'
```

10.1.3.4.2 2. toggle(...states) - State Cycler

```javascript
/**
 * Creates a flag that reports whether it's been triggered.
 * Useful for tracking one-time events or initialization.
 *
 * Use cases:
 * - First-time user experiences
 * - One-time notifications
 * - Initialization tracking
 */
function onceFlag() {
  let triggered = false;

  return {
    trigger: () => {
      if (!triggered) {
        triggered = true;
        return true;  // First time
      }
      return false;  // Already triggered
    },
    isTriggered: () => triggered,
    reset: () => {
      triggered = false;
    }
  };
}

// Example 1: First visit welcome message
const firstVisit = onceFlag();

function showWelcome() {
  if (firstVisit.trigger()) {
    showModal('Welcome to our site!');
    console.log('Welcome message shown');
  }
}

// Example 2: Feature introduction
const featureIntros = {
  dashboard: onceFlag(),
  analytics: onceFlag(),
  settings: onceFlag()
};

function showFeatureIntro(feature) {
  if (featureIntros[feature].trigger()) {
```

```
    showTutorial(feature);
    console.log(`Showing ${feature} tutorial`);
  }
}

// Example 3: One-time migration
const dataMigrated = onceFlag();

async function ensureDataMigration() {
  if (dataMigrated.trigger()) {
    console.log('Running data migration...');
    await migrateUserData();
    await migratePosts();
    console.log('Migration complete');
  } else {
    console.log('Data already migrated');
  }
}

// Example 4: Conditional initialization with dependencies
function createInitFlag(dependencies = []) {
  const flag = onceFlag();

  return {
    init: async (callback) => {
      if (flag.trigger()) {
        // Wait for dependencies
        await Promise.all(dependencies.map(dep => dep.init()));
        await callback();
        return true;
      }
      return false;
    },
    isInitialized: () => flag.isTriggered()
  };
}

const dbInit = createInitFlag();
const cacheInit = createInitFlag([dbInit]);
const apiInit = createInitFlag([dbInit, cacheInit]);

await apiInit.init(async () => {
  console.log('Initializing API...');
  await setupAPI();
});
```

10.1.3.4.3 3. onceFlag() - One-Time Flag

```
/**
 * Creates an accumulator that stores and aggregates values.
 *
 * Use cases:
 * - Running totals
 * - Log collection
 * - Event aggregation
 */
function accumulator(initial = []) {
  const values = [...initial];

  return {
    add: (value) => {
      values.push(value);
      return values.length;
    },
    addAll: (...items) => {
      values.push(...items);
      return values.length;
    },
    get: () => [...values],
    size: () => values.length,
    clear: () => {
      values.length = 0;
    },
    filter: (predicate) => values.filter(predicate),
    map: (fn) => values.map(fn),
    reduce: (fn, init) => values.reduce(fn, init)
  };
}

// Example 1: Error logger
const errorLog = accumulator();

window.addEventListener('error', (event) => {
  errorLog.add({
    message: event.message,
    timestamp: Date.now(),
    stack: event.error?.stack
  });

  console.log(`Total errors: ${errorLog.size()}`);
});

// Get all errors
console.log(errorLog.get());
```

```
// Example 2: Shopping cart
function createShoppingCart() {
  const items = accumulator();

  return {
    addItem: (product, quantity = 1) => {
      items.add({ product, quantity, addedAt: Date.now() });
      console.log(`Added ${quantity}x ${product.name} to cart`);
    },

    getItems: () => items.get(),

    getTotal: () => items.reduce(
      (sum, item) => sum + (item.product.price * item.quantity),
      0
    ),

    getItemCount: () => items.reduce(
      (sum, item) => sum + item.quantity,
      0
    ),

    clearCart: () => items.clear()
  };
}

const cart = createShoppingCart();
cart.addItem({ name: 'Book', price: 15 }, 2);
cart.addItem({ name: 'Pen', price: 2 }, 5);
console.log('Total:', cart.getTotal());  // 40
console.log('Items:', cart.getItemCount());  // 7

// Example 3: Performance metrics
function createMetricsCollector() {
  const metrics = accumulator();

  return {
    record: (name, value, unit = 'ms') => {
      metrics.add({ name, value, unit, timestamp: Date.now() });
    },

    getAverage: (name) => {
      const filtered = metrics.filter(m => m.name === name);
      if (filtered.length === 0) return 0;
      const sum = filtered.reduce((acc, m) => acc + m.value, 0);
      return sum / filtered.length;
    },
```

```
    getMax: (name) => {
      const filtered = metrics.filter(m => m.name === name);
      return Math.max(...filtered.map(m => m.value));
    },

    getStats: (name) => {
      const filtered = metrics.filter(m => m.name === name);
      if (filtered.length === 0) return null;

      const values = filtered.map(m => m.value);
      const sum = values.reduce((a, b) => a + b, 0);

      return {
        count: filtered.length,
        avg: sum / filtered.length,
        min: Math.min(...values),
        max: Math.max(...values)
      };
    },

    clear: () => metrics.clear()
  };
}

const perf = createMetricsCollector();
perf.record('api-call', 150);
perf.record('api-call', 200);
perf.record('render', 16);
console.log(perf.getStats('api-call'));  // { count: 2, avg: 175, min: 150, max: 200 }
```

10.1.3.4.4 4. accumulator() - Value Accumulator

```
/**
 * Creates a private data store with getter/setter methods.
 * Provides encapsulation and controlled access to data.
 *
 * Use cases:
 * - Application state
 * - Configuration management
 * - Data privacy
 */
function createStore(initialData = {}) {
  let data = { ...initialData };
  const listeners = [];

  return {
```

```
    get: (key) => {
      if (key === undefined) return { ...data };
      return data[key];
    },

    set: (key, value) => {
      const oldValue = data[key];
      data[key] = value;

      // Notify listeners
      listeners.forEach(listener => {
        listener(key, value, oldValue);
      });

      return value;
    },

    update: (key, updater) => {
      const oldValue = data[key];
      const newValue = updater(oldValue);
      return this.set(key, newValue);
    },

    delete: (key) => {
      const value = data[key];
      delete data[key];
      return value;
    },

    has: (key) => key in data,

    subscribe: (listener) => {
      listeners.push(listener);
      return () => {
        const index = listeners.indexOf(listener);
        if (index > -1) listeners.splice(index, 1);
      };
    },

    reset: () => {
      data = { ...initialData };
    }
  };
}

// Example 1: User preferences store
const preferences = createStore({
```

```
  theme: 'light',
  language: 'en',
  notifications: true
});

preferences.subscribe((key, newValue, oldValue) => {
  console.log(`${key} changed from ${oldValue} to ${newValue}`);
  localStorage.setItem(`pref_${key}`, JSON.stringify(newValue));
});

preferences.set('theme', 'dark');  // Logs: "theme changed from light to dark"

// Example 2: Application state store
function createAppStore(initial) {
  const store = createStore(initial);

  return {
    getState: () => store.get(),

    setState: (updates) => {
      Object.entries(updates).forEach(([key, value]) => {
        store.set(key, value);
      });
    },

    subscribe: (listener) => store.subscribe(listener),

    dispatch: (action) => {
      switch (action.type) {
        case 'SET_USER':
          store.set('user', action.payload);
          break;
        case 'SET_LOADING':
          store.set('loading', action.payload);
          break;
        case 'SET_ERROR':
          store.set('error', action.payload);
          break;
        default:
          console.warn('Unknown action:', action.type);
      }
    }
  };
}

const appStore = createAppStore({
  user: null,
```

```
  loading: false,
  error: null
});

appStore.subscribe((key, value) => {
  console.log(`State updated: ${key} =`, value);
  renderApp();
});

appStore.dispatch({ type: 'SET_USER', payload: { name: 'Alice' } });

// Example 3: Computed values store
function createComputedStore(initial, computed = {}) {
  const store = createStore(initial);

  return {
    get: (key) => {
      if (key in computed) {
        return computed[key](store.get());
      }
      return store.get(key);
    },
    set: store.set,
    subscribe: store.subscribe
  };
}

const cartStore = createComputedStore(
  { items: [], tax: 0.1 },
  {
    subtotal: (state) => state.items.reduce((sum, item) => sum + item.price * item.qty,
    taxAmount: (state, getters) => getters.subtotal * state.tax,
    total: (state, getters) => getters.subtotal + getters.taxAmount
  }
);
```

10.1.3.4.5 5. createStore(initial) - Private State Store

```
/**
 * Creates a sequential ID generator with optional prefix.
 *
 * Use cases:
 * - Unique identifiers
 * - Element IDs
 * - Transaction IDs
 */
```

```
function sequentialId(prefix = '') {
  let id = 0;

  return {
    next: () => `${prefix}${++id}`,
    current: () => `${prefix}${id}`,
    peek: () => `${prefix}${id + 1}`,
    reset: () => {
      id = 0;
    },
    setPrefix: (newPrefix) => {
      prefix = newPrefix;
    }
  };
}

// Example 1: Element ID generator
const elementIds = sequentialId('el-');

function createElement(tag) {
  const el = document.createElement(tag);
  el.id = elementIds.next();
  return el;
}

const div1 = createElement('div');  // id: "el-1"
const div2 = createElement('div');  // id: "el-2"

// Example 2: Transaction ID generator
const transactionIds = sequentialId('TX');

async function processPayment(amount) {
  const txId = transactionIds.next();
  console.log(`Processing transaction ${txId} for $${amount}`);

  await fetch('/api/payments', {
    method: 'POST',
    body: JSON.stringify({
      transactionId: txId,
      amount
    })
  });

  return txId;
}

// Example 3: Multiple ID generators
```

```
function createIdManager() {
  const generators = new Map();

  return {
    getGenerator: (name, prefix = '') => {
      if (!generators.has(name)) {
        generators.set(name, sequentialId(prefix));
      }
      return generators.get(name);
    },

    nextId: (name) => {
      return this.getGenerator(name).next();
    }
  };
}

const idManager = createIdManager();

const userId = idManager.nextId('user');        // "1"
const postId = idManager.nextId('post');        // "1"
const commentId = idManager.nextId('comment');  // "1"
const userId2 = idManager.nextId('user');       // "2"

// Example 4: UUID-style generator with closure
function createUuidGenerator() {
  let timestamp = Date.now();
  let sequence = 0;

  return function() {
    const now = Date.now();

    if (now === timestamp) {
      sequence++;
    } else {
      timestamp = now;
      sequence = 0;
    }

    return `${timestamp}-${sequence}-${Math.random().toString(36).substr(2, 9)}`;
  };
}

const generateUuid = createUuidGenerator();
console.log(generateUuid());  // "1699123456789-0-a1b2c3d4e"
console.log(generateUuid());  // "1699123456789-1-f5g6h7i8j"
```

10.1.3.4.6 6. sequentialId(prefix) - ID Generator

10.1.3.5 E. Asynchronous & Promise Utilities

These functions manage async state and concurrency through closures, providing advanced async programming patterns.

```
/**
 * Debounces async functions, cancelling pending promises.
 * Ensures only the latest call completes.
 *
 * Use cases:
 * - Async API calls
 * - Form validation
 * - Autosave functionality
 */
function debounceAsync(fn, wait) {
  let timeout;
  let pending = null;

  return function(...args) {
    clearTimeout(timeout);

    // Cancel previous pending promise
    if (pending) {
      pending.cancel = true;
    }

    return new Promise((resolve, reject) => {
      const promise = { cancel: false };
      pending = promise;

      timeout = setTimeout(async () => {
        try {
          if (!promise.cancel) {
            const result = await fn.apply(this, args);
            if (!promise.cancel) {
              resolve(result);
            }
          }
        } catch (error) {
          if (!promise.cancel) {
            reject(error);
          }
        }
      }, wait);
```

```
    });
  };
}

// Example 1: Debounced API search
const searchAPI = debounceAsync(async (query) => {
  console.log('Searching for:', query);
  const response = await fetch(`/api/search?q=${query}`);
  return response.json();
}, 500);

searchInput.addEventListener('input', async (e) => {
  try {
    const results = await searchAPI(e.target.value);
    displayResults(results);
  } catch (error) {
    console.error('Search failed:', error);
  }
});

// Example 2: Auto-save with debounce
const autoSave = debounceAsync(async (content) => {
  console.log('Saving...');
  const response = await fetch('/api/save', {
    method: 'POST',
    body: JSON.stringify({ content })
  });

  if (response.ok) {
    showSaveIndicator('Saved!');
  }

  return response.json();
}, 2000);

editor.addEventListener('input', (e) => {
  autoSave(e.target.value);
});
```

10.1.3.5.1 1. debounceAsync(fn, wait) - Debounce Async Functions

```
/**
 * Throttles async functions to run at most once per wait period.
 *
 * Use cases:
 * - Rate-limited API calls
```

```
 * - Scroll-triggered data fetching
 * - Real-time updates
 */
function throttleAsync(fn, wait) {
  let timeout;
  let lastRan = 0;
  let pending = null;

  return async function(...args) {
    const now = Date.now();

    if (now - lastRan >= wait) {
      lastRan = now;
      return fn.apply(this, args);
    } else if (!pending) {
      pending = new Promise((resolve, reject) => {
        timeout = setTimeout(async () => {
          lastRan = Date.now();
          pending = null;
          try {
            const result = await fn.apply(this, args);
            resolve(result);
          } catch (error) {
            reject(error);
          }
        }, wait - (now - lastRan));
      });
    }

    return pending;
  };
}

// Example: Throttled infinite scroll
const loadMore = throttleAsync(async () => {
  console.log('Loading more items...');
  const response = await fetch(`/api/items?page=${currentPage++}`);
  const items = await response.json();
  appendItems(items);
  return items;
}, 1000);

window.addEventListener('scroll', () => {
  if (window.innerHeight + window.scrollY >= document.body.offsetHeight - 500) {
    loadMore();
  }
});
```

10.1.3.5.2 2. throttleAsync(fn, wait) - Throttle Async Functions

```
/**
 * Controls async function concurrency.
 * Tracks running promises and queues additional calls.
 *
 * Use cases:
 * - API rate limiting
 * - Resource pool management
 * - Controlled parallel processing
 */
function queue(fn, concurrency = 1) {
  const waiting = [];
  let running = 0;

  async function processNext() {
    if (waiting.length === 0 || running >= concurrency) {
      return;
    }

    running++;
    const { args, resolve, reject, context } = waiting.shift();

    try {
      const result = await fn.apply(context, args);
      resolve(result);
    } catch (error) {
      reject(error);
    } finally {
      running--;
      processNext();
    }
  }

  return function(...args) {
    return new Promise((resolve, reject) => {
      waiting.push({
        args,
        resolve,
        reject,
        context: this
      });
      processNext();
    });
  };
}
```

```javascript
// Example 1: Concurrent API calls with limit
const fetchWithLimit = queue(async (url) => {
  console.log('Fetching:', url);
  const response = await fetch(url);
  return response.json();
}, 3);  // Max 3 concurrent requests

const urls = [
  '/api/user/1',
  '/api/user/2',
  '/api/user/3',
  '/api/user/4',
  '/api/user/5'
];

const results = await Promise.all(urls.map(url => fetchWithLimit(url)));

// Example 2: Image upload queue
const uploadImage = queue(async (file) => {
  console.log(`Uploading ${file.name}...`);
  const formData = new FormData();
  formData.append('file', file);

  const response = await fetch('/api/upload', {
    method: 'POST',
    body: formData
  });

  return response.json();
}, 2);  // Upload 2 images at a time

const files = Array.from(fileInput.files);
const uploads = files.map(file => uploadImage(file));
const results = await Promise.all(uploads);
```

10.1.3.5.3 3. queue(fn, concurrency) - Concurrency Control Queue

```javascript

/**
 * Limits concurrent executions of async function to n.
 *
 * Use cases:
 * - Database connection pooling
 * - API call throttling
 * - Resource management
 */
function limit(fn, n) {
```

```
  let running = 0;
  const waiting = [];

  async function tryRun(args, context) {
    if (running < n) {
      running++;
      try {
        return await fn.apply(context, args);
      } finally {
        running--;
        if (waiting.length > 0) {
          const next = waiting.shift();
          tryRun(next.args, next.context).then(next.resolve, next.reject);
        }
      }
    } else {
      return new Promise((resolve, reject) => {
        waiting.push({ args, context, resolve, reject });
      });
    }
  }

  return function(...args) {
    return tryRun(args, this);
  };
}

// Example: Database query limiter
const query = limit(async (sql, params) => {
  console.log('Executing query:', sql);
  const connection = await pool.getConnection();
  try {
    const result = await connection.query(sql, params);
    return result;
  } finally {
    connection.release();
  }
}, 10);  // Max 10 concurrent queries

// Multiple queries - only 10 run concurrently
const queries = Array.from({ length: 50 }, (_, i) =>
  query('SELECT * FROM users WHERE id = ?', [i + 1])
);

const results = await Promise.all(queries);
```

10.1.3.5.4 4. limit(fn, n) - Concurrent Execution Limit

```javascript
/**
 * Retries async function with exponential backoff.
 *
 * Use cases:
 * - Network request retries
 * - Flaky API handling
 * - Resource availability waiting
 */
function retryAsync(fn, attempts = 3, delay = 1000, backoff = 2) {
  return async function(...args) {
    let lastError;

    for (let i = 0; i < attempts; i++) {
      try {
        return await fn.apply(this, args);
      } catch (error) {
        lastError = error;

        if (i < attempts - 1) {
          const waitTime = delay * Math.pow(backoff, i);
          console.log(`Attempt ${i + 1} failed, retrying in ${waitTime}ms...`);
          await new Promise(resolve => setTimeout(resolve, waitTime));
        }
      }
    }

    throw lastError;
  };
}

// Example: Retry API call
const fetchWithRetry = retryAsync(async (url) => {
  console.log('Fetching:', url);
  const response = await fetch(url);

  if (!response.ok) {
    throw new Error(`HTTP ${response.status}`);
  }

  return response.json();
}, 5, 1000, 2);

try {
  const data = await fetchWithRetry('/api/data');
  console.log('Success:', data);
} catch (error) {
```

```
  console.error('Failed after retries:', error);
}
```

10.1.3.5.5 5. retryAsync(fn, attempts, delay) - Async Retry Logic

```
/**
 * Caches in-flight promises to prevent duplicate requests.
 *
 * Use cases:
 * - Prevent duplicate API calls
 * - Resource initialization
 * - Data fetching optimization
 */
function cachePromise(fn) {
  const cache = new Map();

  return function(...args) {
    const key = JSON.stringify(args);

    if (cache.has(key)) {
      const cached = cache.get(key);
      if (cached.pending) {
        console.log('Returning in-flight promise');
        return cached.promise;
      }
      if (Date.now() - cached.timestamp < 60000) {
        console.log('Returning cached result');
        return Promise.resolve(cached.value);
      }
    }

    console.log('Creating new promise');
    const promise = fn.apply(this, args)
      .then(value => {
        cache.set(key, {
          value,
          pending: false,
          timestamp: Date.now()
        });
        return value;
      })
      .catch(error => {
        cache.delete(key);
        throw error;
      });
```

```
    cache.set(key, { promise, pending: true });
    return promise;
  };
}

// Example: Prevent duplicate user fetches
const fetchUser = cachePromise(async (userId) => {
  console.log(`Fetching user ${userId}...`);
  const response = await fetch(`/api/users/${userId}`);
  return response.json();
});

// Multiple simultaneous calls return same promise
const user1Promise = fetchUser(123);
const user2Promise = fetchUser(123);  // Same promise
const user3Promise = fetchUser(123);  // Same promise

console.log(user1Promise === user2Promise);  // true
```

10.1.3.5.6 6. cachePromise(fn) - Promise Deduplication

```
/**
 * Defers async execution to next tick.
 *
 * Use cases:
 * - Breaking up heavy async operations
 * - Yielding to event loop
 * - Background task processing
 */
function deferAsync(fn) {
  return function(...args) {
    return new Promise((resolve) => {
      setTimeout(async () => {
        resolve(await fn.apply(this, args));
      }, 0);
    });
  };
}

// Example: Process large dataset in chunks
async function processLargeDataset(data) {
  const processChunk = deferAsync(async (chunk) => {
    return chunk.map(item => expensiveOperation(item));
  });

  const chunkSize = 1000;
```

```
  const results = [];

  for (let i = 0; i < data.length; i += chunkSize) {
    const chunk = data.slice(i, i + chunkSize);
    const result = await processChunk(chunk);
    results.push(...result);

    // Update progress
    updateProgress(i / data.length * 100);
  }

  return results;
}
```

10.1.3.5.7 7. deferAsync(fn) - Async Task Deferral

10.1.3.6 F. Utility Closures for Events & DOM

Functions that encapsulate event state and DOM behavior using closures.

```
/**
 * Adds an event listener that automatically removes itself after first execution.
 *
 * Use cases:
 * - One-time user interactions
 * - Initial load events
 * - Transition/animation end handlers
 */
function onceEvent(element, event, handler) {
  const wrapper = function(e) {
    handler.call(this, e);
    element.removeEventListener(event, wrapper);
  };

  element.addEventListener(event, wrapper);

  return () => element.removeEventListener(event, wrapper);
}

// Example 1: One-time welcome modal
onceEvent(document, 'DOMContentLoaded', () => {
  showWelcomeModal();
  console.log('Welcome modal shown once');
});

// Example 2: First click handler
```

```
const button = document.getElementById('subscribe-btn');
onceEvent(button, 'click', async () => {
  console.log('Processing subscription...');
  await subscribe();
  button.textContent = 'Subscribed!';
  button.disabled = true;
});

// Example 3: Animation end cleanup
const element = document.querySelector('.animate');
onceEvent(element, 'animationend', () => {
  element.classList.remove('animating');
  element.style.opacity = '1';
});
```

10.1.3.6.1 1. onceEvent(element, event, handler) - One-Time Event Listener

```
/**
 * Creates a debounced resize handler with cleanup.
 *
 * Use cases:
 * - Responsive layout recalculation
 * - Chart resizing
 * - Mobile orientation changes
 */
function debouncedResize(handler, wait = 250) {
  let timeout;

  const debouncedHandler = () => {
    clearTimeout(timeout);
    timeout = setTimeout(() => {
      handler();
    }, wait);
  };

  window.addEventListener('resize', debouncedHandler);

  // Return cleanup function
  return () => {
    clearTimeout(timeout);
    window.removeEventListener('resize', debouncedHandler);
  };
}

// Example 1: Responsive chart
const stopListening = debouncedResize(() => {
```

```
  console.log('Resizing chart to:', window.innerWidth, 'x', window.innerHeight);
  chart.resize(window.innerWidth, window.innerHeight);
  recalculateLayout();
}, 300);

// Clean up when component unmounts
// stopListening();

// Example 2: Mobile-aware resize
function createResponsiveHandler() {
  let isMobile = window.innerWidth < 768;

  return debouncedResize(() => {
    const wasMobile = isMobile;
    isMobile = window.innerWidth < 768;

    if (wasMobile !== isMobile) {
      console.log(`Switched to ${isMobile ? 'mobile' : 'desktop'} view`);
      reloadLayout();
    } else {
      adjustLayout();
    }
  }, 200);
}

const cleanup = createResponsiveHandler();
```

10.1.3.6.2 2. debouncedResize(handler, wait) - Debounced Resize Handler

```

/**
 * Buffers multiple rapid events into a single call.
 *
 * Use cases:
 * - Keyboard input handling
 * - Mouse movement tracking
 * - Touch gesture recognition
 */
function eventBuffer(fn, delay = 100) {
  const events = [];
  let timeout;

  return function(event) {
    events.push(event);

    clearTimeout(timeout);
    timeout = setTimeout(() => {
```

```
      fn.call(this, [...events]);
      events.length = 0;
    }, delay);
  };
}

// Example 1: Buffer mouse movements
const handleMouseMoves = eventBuffer((events) => {
  console.log(`Processed ${events.length} mouse movements`);
  const avgX = events.reduce((sum, e) => sum + e.clientX, 0) / events.length;
  const avgY = events.reduce((sum, e) => sum + e.clientY, 0) / events.length;
  updateCursor(avgX, avgY);
}, 50);

document.addEventListener('mousemove', handleMouseMoves);

// Example 2: Buffer keystrokes
const handleKeystrokes = eventBuffer((events) => {
  const text = events.map(e => e.key).join('');
  console.log('Text entered:', text);
  processInput(text);
}, 500);

input.addEventListener('keydown', handleKeystrokes);
```

10.1.3.6.3 3. eventBuffer(fn, delay) - Event Buffering

```
/**
 * Loads DOM content only once when needed.
 *
 * Use cases:
 * - Lazy image loading
 * - Infinite scroll
 * - Progressive content loading
 */
function lazyLoader(selector) {
  const loaded = new Set();

  const observer = new IntersectionObserver((entries) => {
    entries.forEach(entry => {
      if (entry.isIntersecting && !loaded.has(entry.target)) {
        loaded.add(entry.target);
        loadContent(entry.target);
      }
    });
  });
```

```
  function loadContent(element) {
    const src = element.dataset.src;
    if (src) {
      console.log('Loading:', src);

      if (element.tagName === 'IMG') {
        element.src = src;
      } else {
        fetch(src)
          .then(r => r.text())
          .then(html => {
            element.innerHTML = html;
          });
      }
    }
  }

  document.querySelectorAll(selector).forEach(el => {
    observer.observe(el);
  });

  return {
    observe: (element) => observer.observe(element),
    unobserve: (element) => observer.unobserve(element),
    disconnect: () => observer.disconnect()
  };
}

// Example: Lazy load images
const imageLoader = lazyLoader('img[data-src]');

// Dynamically add new images
const newImg = document.createElement('img');
newImg.dataset.src = '/images/photo.jpg';
document.body.appendChild(newImg);
imageLoader.observe(newImg);
```

10.1.3.6.4 4. lazyLoader(selector) - Lazy Content Loading

```
/**
 * Retains last right-clicked element for context menu.
 *
 * Use cases:
 * - Custom context menus
 * - Right-click actions
 * - Element-specific menus
```

```
 */
function withContextMenu(state = {}) {
  let currentElement = null;

  document.addEventListener('contextmenu', (e) => {
    currentElement = e.target;
    state.x = e.clientX;
    state.y = e.clientY;
  });

  return {
    getCurrentElement: () => currentElement,
    getPosition: () => ({ x: state.x, y: state.y }),

    showMenu: (menuElement) => {
      if (currentElement) {
        menuElement.style.left = `${state.x}px`;
        menuElement.style.top = `${state.y}px`;
        menuElement.style.display = 'block';
      }
    },

    hideMenu: (menuElement) => {
      menuElement.style.display = 'none';
      currentElement = null;
    }
  };
}

// Example: Custom context menu
const contextMenu = withContextMenu();

document.addEventListener('contextmenu', (e) => {
  e.preventDefault();
  const element = contextMenu.getCurrentElement();

  if (element.classList.contains('editable')) {
    const menu = document.getElementById('context-menu');
    contextMenu.showMenu(menu);

    // Set up menu actions
    document.getElementById('edit').onclick = () => {
      editElement(element);
      contextMenu.hideMenu(menu);
    };

    document.getElementById('delete').onclick = () => {
```

```
      deleteElement(element);
      contextMenu.hideMenu(menu);
    };
  }
});

document.addEventListener('click', () => {
  const menu = document.getElementById('context-menu');
  contextMenu.hideMenu(menu);
});
```

10.1.3.6.5 5. withContextMenu(state) - Context Menu State

10.1.3.7 G. Security / Privacy via Closure

Functions that use closures to hide internal data and provide secure interfaces.

```
/**
 * Creates a private data store using WeakMap or closure.
 * Prevents external access to sensitive data.
 *
 * Use cases:
 * - Sensitive user data
 * - Private class fields
 * - Encapsulated state
 */
function createPrivateStore() {
  const store = new WeakMap();

  return {
    set: (obj, key, value) => {
      if (!store.has(obj)) {
        store.set(obj, {});
      }
      store.get(obj)[key] = value;
    },

    get: (obj, key) => {
      const data = store.get(obj);
      return data ? data[key] : undefined;
    },

    has: (obj, key) => {
      const data = store.get(obj);
      return data ? key in data : false;
    },
```

```
    delete: (obj, key) => {
      const data = store.get(obj);
      if (data) {
        delete data[key];
      }
    }
  };
}

// Example: User with private data
const privateData = createPrivateStore();

class User {
  constructor(username, password) {
    this.username = username;
    privateData.set(this, 'password', hashPassword(password));
    privateData.set(this, 'apiKey', generateApiKey());
  }

  verifyPassword(password) {
    const stored = privateData.get(this, 'password');
    return hashPassword(password) === stored;
  }

  getApiKey(password) {
    if (this.verifyPassword(password)) {
      return privateData.get(this, 'apiKey');
    }
    throw new Error('Invalid password');
  }
}

const user = new User('alice', 'secret123');
console.log(user.username);  // 'alice'
console.log(user.password);  // undefined (private!)
console.log(user.verifyPassword('secret123'));  // true
```

10.1.3.7.1 1. createPrivateStore() - Private Data Storage

```
/**
 * Creates a counter with hidden internal value.
 * Only exposes methods, not state.
 *
 * Use cases:
 * - Rate limiting
 * - Usage tracking
```

```javascript
 * - Secure incrementing
 */
function makeCounter() {
  let count = 0;

  return {
    inc() {
      return ++count;
    },
    dec() {
      return --count;
    },
    value() {
      // Return copy, not reference
      return count;
    }
  };
}

// Example: API rate limit counter
function createRateLimiter(maxRequests, windowMs) {
  const counter = makeCounter();
  let windowStart = Date.now();

  return {
    canMakeRequest: () => {
      const now = Date.now();

      // Reset window
      if (now - windowStart > windowMs) {
        windowStart = now;
        // Reset counter (create new one)
        return true;
      }

      return counter.value() < maxRequests;
    },

    recordRequest: () => {
      if (this.canMakeRequest()) {
        counter.inc();
        return true;
      }
      return false;
    }
  };
}
```

10.1.3.7.2 2. makeCounter() - Hidden Counter

```
/**
 * Creates objects with private methods.
 *
 * Use cases:
 * - Internal helpers
 * - Protected operations
 * - Implementation hiding
 */
function privateMethodFactory() {
  // Private methods (not exposed)
  function validateInput(input) {
    if (!input || input.length < 3) {
      throw new Error('Invalid input');
    }
  }

  function sanitize(input) {
    return input.trim().toLowerCase();
  }

  function encrypt(data) {
    // Simplified encryption
    return btoa(data);
  }

  // Public interface
  return {
    processData(input) {
      validateInput(input);
      const clean = sanitize(input);
      return encrypt(clean);
    },

    safeProcess(input) {
      try {
        return this.processData(input);
      } catch (error) {
        console.error('Processing failed:', error.message);
        return null;
      }
    }
  };
}
```

```javascript
// Example: Secure API client
function createSecureClient(apiKey) {
  // Private methods
  function sign(data) {
    return hmacSha256(data, apiKey);
  }

  function encrypt(data) {
    return aesEncrypt(data, apiKey);
  }

  // Public methods
  return {
    async sendRequest(endpoint, data) {
      const encrypted = encrypt(JSON.stringify(data));
      const signature = sign(encrypted);

      const response = await fetch(endpoint, {
        method: 'POST',
        headers: {
          'X-Signature': signature
        },
        body: encrypted
      });

      return response.json();
    }
  };
}

const client = createSecureClient('secret-key');
// client.sign() - not accessible (private)
// client.encrypt() - not accessible (private)
client.sendRequest('/api/data', { message: 'hello' });  // Works
```

10.1.3.7.3 3. privateMethodFactory() - Private Methods

```javascript
/**
 * Validates a secret only once.
 *
 * Use cases:
 * - One-time passwords
 * - Single-use tokens
 * - Secure initialization
 */
function onceSecret(expectedKey) {
```

```
  let used = false;
  let attempts = 0;
  const maxAttempts = 3;

  return {
    validate: (key) => {
      if (used) {
        throw new Error('Secret already used');
      }

      if (attempts >= maxAttempts) {
        throw new Error('Too many attempts');
      }

      attempts++;

      if (key === expectedKey) {
        used = true;
        return true;
      }

      return false;
    },

    isUsed: () => used,
    getAttempts: () => attempts
  };
}

// Example: One-time initialization
const initSecret = onceSecret('init-token-12345');

async function initialize(token) {
  try {
    if (initSecret.validate(token)) {
      console.log('Initializing system...');
      await setupDatabase();
      await loadConfiguration();
      console.log('System initialized');
      return true;
    } else {
      console.error('Invalid initialization token');
      return false;
    }
  } catch (error) {
    console.error('Initialization error:', error.message);
    return false;
```

```
  }
}

// First call with correct token
await initialize('init-token-12345');  // Success

// Second call fails
await initialize('init-token-12345');  // Error: Secret already used
```

10.1.3.7.4 4. onceSecret(key) - One-Time Secret Validation

10.1.3.8 H. Miscellaneous Functional Tools

Other closure-driven helpers commonly found in functional programming libraries.

```
/**
 * Returns a function that always returns the same value.
 *
 * Use cases:
 * - Default values
 * - Functional composition
 * - Testing/mocking
 */
function constant(value) {
  return function() {
    return value;
  };
}

// Example 1: Default values
const getDefaultUser = constant({ name: 'Guest', role: 'visitor' });

function loadUser(id) {
  if (!id) return getDefaultUser();
  return fetchUser(id);
}

// Example 2: Functional composition
const numbers = [1, 2, 3, 4, 5];
const always10 = constant(10);

console.log(numbers.map(always10));  // [10, 10, 10, 10, 10]

// Example 3: Mocking
const mockFetch = constant(Promise.resolve({ data: 'mock data' }));
```

10.1.3.8.1 1. constant(value) - Always Return Same Value

```
/**
 * Returns its argument unchanged.
 * Useful in functional chains and as default callback.
 *
 * Use cases:
 * - Default transformers
 * - Functional composition
 * - Array filtering
 */
function identity(x) {
  return x;
}

// Example 1: Filter truthy values
const values = [0, 1, false, true, '', 'hello', null, 'world'];
console.log(values.filter(identity));  // [1, true, 'hello', 'world']

// Example 2: Default transformer
function transform(data, transformer = identity) {
  return data.map(transformer);
}

console.log(transform([1, 2, 3]));  // [1, 2, 3]
console.log(transform([1, 2, 3], x => x * 2));  // [2, 4, 6]

// Example 3: Flatten arrays
const nested = [[1], [2], [3]];
console.log(nested.flatMap(identity));  // [1, 2, 3]
```

10.1.3.8.2 2. identity(x) - Return Argument Unchanged

```
/**
 * A function that does nothing.
 * Useful as placeholder or default callback.
 *
 * Use cases:
 * - Default callbacks
 * - Event handler placeholders
 * - Testing
 */
function noop() {}

// Example 1: Default callback
```

```
function fetchData(url, onSuccess = noop, onError = noop) {
  fetch(url)
    .then(r => r.json())
    .then(onSuccess)
    .catch(onError);
}

// Example 2: Optional event handlers
class EventEmitter {
  constructor() {
    this.handlers = {};
  }

  on(event, handler = noop) {
    this.handlers[event] = handler;
  }

  emit(event, data) {
    const handler = this.handlers[event] || noop;
    handler(data);
  }
}
```

10.1.3.8.3 3. noop() - No Operation

```

/**
 * Generates unique IDs with internal counter.
 *
 * Use cases:
 * - Element IDs
 * - Temporary keys
 * - Testing
 */
function uniqueId(prefix = 'id') {
  let counter = 0;

  return function() {
    return `${prefix}_${++counter}`;
  };
}

// Example: Component ID generator
const generateComponentId = uniqueId('component');

class Component {
  constructor() {
```

```
    this.id = generateComponentId();
  }
}

const comp1 = new Component();  // id: "component_1"
const comp2 = new Component();  // id: "component_2"
```

10.1.3.8.4 4. uniqueId(prefix) - Unique ID Generator

```
/**
 * Executes a function n times, maintaining internal counter.
 *
 * Use cases:
 * - Batch operations
 * - Testing
 * - Data generation
 */
function times(n, fn) {
  const results = [];
  for (let i = 0; i < n; i++) {
    results.push(fn(i));
  }
  return results;
}

// Example 1: Generate test data
const users = times(10, i => ({
  id: i + 1,
  name: `User ${i + 1}`,
  email: `user${i + 1}@example.com`
}));

// Example 2: Create elements
const buttons = times(5, i => {
  const btn = document.createElement('button');
  btn.textContent = `Button ${i + 1}`;
  btn.onclick = () => console.log(`Clicked button ${i + 1}`);
  return btn;
});

buttons.forEach(btn => document.body.appendChild(btn));
```

10.1.3.8.5 5. times(n, fn) - Execute n Times

```javascript
/**
 * Runs multiple functions in order.
 *
 * Use cases:
 * - Initialization sequences
 * - Setup procedures
 * - Event chains
 */
function sequence(...fns) {
  return function(...args) {
    fns.forEach(fn => fn.apply(this, args));
  };
}

// Example: Initialization sequence
const initialize = sequence(
  () => console.log('Loading configuration...'),
  () => console.log('Connecting to database...'),
  () => console.log('Starting server...'),
  () => console.log('Application ready!')
);

initialize();
```

10.1.3.8.6 6. sequence(...fns) - Sequential Execution

```javascript

/**
 * Combines multiple boolean closures with AND logic.
 *
 * Use cases:
 * - Complex validation
 * - Filter combinations
 * - Authorization checks
 */
function predicateChain(...predicates) {
  return function(...args) {
    return predicates.every(pred => pred.apply(this, args));
  };
}

// Example: Complex validation
const isAdult = user => user.age >= 18;
const hasEmail = user => !!user.email;
const isVerified = user => user.verified === true;

const canAccessService = predicateChain(isAdult, hasEmail, isVerified);
```

```
const user1 = { age: 25, email: 'user@example.com', verified: true };
const user2 = { age: 16, email: 'teen@example.com', verified: true };

console.log(canAccessService(user1));  // true
console.log(canAccessService(user2));  // false (not adult)
```

10.1.3.8.7 7. predicateChain(...predicates) - Chain Predicates

```
/**
 * Repeats function until success or max tries.
 *
 * Use cases:
 * - Flaky operations
 * - Network retries
 * - Resource contention
 */
function withRetry(fn, maxTries = 3) {
  return function(...args) {
    let lastError;

    for (let i = 0; i < maxTries; i++) {
      try {
        return fn.apply(this, args);
      } catch (error) {
        lastError = error;
        console.log(`Attempt ${i + 1} failed`);
      }
    }

    throw lastError;
  };
}

// Example: Retry file read
const readFileWithRetry = withRetry((filename) => {
  const content = fs.readFileSync(filename, 'utf8');
  if (!content) throw new Error('Empty file');
  return content;
}, 5);

try {
  const data = readFileWithRetry('config.json');
  console.log('File read successfully');
} catch (error) {
  console.error('Failed to read file after retries');
}
```

10.1.3.8.8 8. withRetry(fn, maxTries) - Simple Retry

```
/**
 * Runs function at most once per event loop tick.
 *
 * Use cases:
 * - Batched DOM updates
 * - Render optimization
 * - Event coalescing
 */
function oncePerTick(fn) {
  let pending = false;
  let args = null;

  return function(...newArgs) {
    args = newArgs;

    if (!pending) {
      pending = true;

      Promise.resolve().then(() => {
        pending = false;
        fn.apply(this, args);
      });
    }
  };
}

// Example: Batch DOM updates
const updateUI = oncePerTick(() => {
  console.log('Updating UI (batched)');
  renderComponent();
});

// Multiple calls in same tick result in single update
updateUI();
updateUI();
updateUI();
// Only renders once after current tick completes
```

10.1.3.8.9 9. oncePerTick(fn) - Once Per Event Loop Tick

10.2 JavaScript Interview Questions - Comprehensive Solutions

This section contains in-depth solutions for 100 essential JavaScript interview questions, including variants, advanced modifications, and real-world applications.

10.2.1 Q1: Promise.all() Polyfill

Problem: Implement `Promise.all()` that resolves when all promises resolve, or rejects when any promise rejects.

Solution:

```
/**
 * Promise.all() polyfill
 * Waits for all promises to resolve or any to reject
 */
Promise.myAll = function(promises) {
  return new Promise((resolve, reject) => {
    // Handle non-array input
    if (!Array.isArray(promises)) {
      return reject(new TypeError('Argument must be an array'));
    }

    // Handle empty array
    if (promises.length === 0) {
      return resolve([]);
    }

    const results = [];
    let completedCount = 0;

    promises.forEach((promise, index) => {
      // Convert non-promise values to promises
      Promise.resolve(promise)
        .then(value => {
          results[index] = value;
          completedCount++;

          // All promises resolved
          if (completedCount === promises.length) {
            resolve(results);
          }
        })
        .catch(error => {
          // Any promise rejects, reject immediately
          reject(error);
        });
```

```
    });
  });
};

// Example usage
const p1 = Promise.resolve(1);
const p2 = Promise.resolve(2);
const p3 = new Promise((resolve) => setTimeout(() => resolve(3), 100));

Promise.myAll([p1, p2, p3])
  .then(results => console.log(results))  // [1, 2, 3]
  .catch(error => console.error(error));

// With rejection
const p4 = Promise.reject('Error!');
Promise.myAll([p1, p2, p4])
  .then(results => console.log(results))
  .catch(error => console.error(error));  // 'Error!'
```

Variant 1: With Progress Tracking

```
Promise.allWithProgress = function(promises, onProgress) {
  return new Promise((resolve, reject) => {
    if (!Array.isArray(promises)) {
      return reject(new TypeError('Argument must be an array'));
    }

    if (promises.length === 0) {
      return resolve([]);
    }

    const results = [];
    let completedCount = 0;
    const total = promises.length;

    promises.forEach((promise, index) => {
      Promise.resolve(promise)
        .then(value => {
          results[index] = value;
          completedCount++;

          // Report progress
          if (onProgress) {
            onProgress({
              completed: completedCount,
              total,
              percentage: (completedCount / total) * 100,
              index
```

```
            });
          }

          if (completedCount === total) {
            resolve(results);
          }
        })
        .catch(reject);
    });
  });
};

// Usage
const tasks = [
  fetch('/api/user'),
  fetch('/api/posts'),
  fetch('/api/comments')
];

Promise.allWithProgress(tasks, (progress) => {
  console.log(`Progress: ${progress.percentage.toFixed(0)}%`);
})
.then(results => console.log('All done:', results));
```

Variant 2: With Timeout

```
Promise.allWithTimeout = function(promises, timeoutMs) {
  const timeoutPromise = new Promise((_, reject) => {
    setTimeout(() => reject(new Error('Timeout exceeded')), timeoutMs);
  });

  return Promise.race([
    Promise.myAll(promises),
    timeoutPromise
  ]);
};

// Usage
Promise.allWithTimeout([p1, p2, p3], 5000)
  .then(results => console.log('Completed within timeout:', results))
  .catch(error => console.error('Timeout or error:', error));
```

Advanced Modification: Batched Execution with Concurrency Limit

```
Promise.allBatched = function(promises, batchSize = 5) {
  return new Promise(async (resolve, reject) => {
    const results = [];
    const batches = [];
```

```
    // Create batches
    for (let i = 0; i < promises.length; i += batchSize) {
      batches.push(promises.slice(i, i + batchSize));
    }

    try {
      // Execute batches sequentially
      for (const batch of batches) {
        const batchResults = await Promise.all(
          batch.map(p => Promise.resolve(p))
        );
        results.push(...batchResults);
      }
      resolve(results);
    } catch (error) {
      reject(error);
    }
  });
};

// Usage: Process 50 promises, 5 at a time
const manyPromises = Array.from({ length: 50 }, (_, i) =>
  fetch(`/api/item/${i}`)
);

Promise.allBatched(manyPromises, 5)
  .then(results => console.log('All batches completed:', results.length));
```

10.2.2 Q2: Promise.any() Polyfill

Problem: Resolve as soon as any promise resolves (opposite of Promise.all).

Solution:

```
/**
 * Promise.any() polyfill
 * Resolves with first successful promise, rejects if all fail
 */
Promise.myAny = function(promises) {
  return new Promise((resolve, reject) => {
    if (!Array.isArray(promises)) {
      return reject(new TypeError('Argument must be an array'));
    }

    if (promises.length === 0) {
      return reject(new AggregateError([], 'All promises were rejected'));
    }
```

```
    const errors = [];
    let rejectedCount = 0;

    promises.forEach((promise, index) => {
      Promise.resolve(promise)
        .then(value => {
          // First one to resolve wins
          resolve(value);
        })
        .catch(error => {
          errors[index] = error;
          rejectedCount++;

          // All promises rejected
          if (rejectedCount === promises.length) {
            reject(new AggregateError(errors, 'All promises were rejected'));
          }
        });
    });
  });
};

// Example usage
const slow = new Promise((resolve) => setTimeout(() => resolve('slow'), 1000));
const fast = new Promise((resolve) => setTimeout(() => resolve('fast'), 100));
const failed = Promise.reject('error');

Promise.myAny([slow, fast, failed])
  .then(result => console.log(result))  // 'fast' (first to resolve)
  .catch(error => console.error(error));

// All reject
Promise.myAny([
  Promise.reject('err1'),
  Promise.reject('err2'),
  Promise.reject('err3')
])
  .then(result => console.log(result))
  .catch(error => {
    console.error(error.message);  // 'All promises were rejected'
    console.error(error.errors);   // ['err1', 'err2', 'err3']
  });
```

Variant: With Fallback Value

```
Promise.anyWithFallback = function(promises, fallbackValue) {
  return Promise.myAny(promises)
```

```
    .catch(() => fallbackValue);
};

// Usage
Promise.anyWithFallback([
  fetch('/api/primary'),
  fetch('/api/secondary'),
  fetch('/api/tertiary')
], { data: 'default' })
  .then(result => console.log('Got result:', result));
```

Advanced: Fastest Response with Quality Check

```
Promise.anyWithValidation = function(promises, validator) {
  return new Promise((resolve, reject) => {
    if (!Array.isArray(promises)) {
      return reject(new TypeError('Argument must be an array'));
    }

    const errors = [];
    let settledCount = 0;

    promises.forEach((promise, index) => {
      Promise.resolve(promise)
        .then(value => {
          // Validate result
          if (validator(value)) {
            resolve(value);
          } else {
            errors[index] = new Error('Validation failed');
            settledCount++;

            if (settledCount === promises.length) {
              reject(new AggregateError(errors, 'No valid result found'));
            }
          }
        })
        .catch(error => {
          errors[index] = error;
          settledCount++;

          if (settledCount === promises.length) {
            reject(new AggregateError(errors, 'All promises were rejected'));
          }
        });
    });
  });
};
```

```javascript
// Usage: Get first valid API response
Promise.anyWithValidation(
  [
    fetch('/api/server1').then(r => r.json()),
    fetch('/api/server2').then(r => r.json()),
    fetch('/api/server3').then(r => r.json())
  ],
  (data) => data && data.status === 'ok' && data.value
)
  .then(result => console.log('Valid result:', result))
  .catch(error => console.error('No valid responses'));
```

10.2.3 Q3: Promise.race() Polyfill

Problem: Return first settled promise (resolved or rejected).

Solution:

```javascript
/**
 * Promise.race() polyfill
 * Returns first promise to settle (resolve or reject)
 */
Promise.myRace = function(promises) {
  return new Promise((resolve, reject) => {
    if (!Array.isArray(promises)) {
      return reject(new TypeError('Argument must be an array'));
    }

    if (promises.length === 0) {
      return;  // Never settles
    }

    promises.forEach(promise => {
      Promise.resolve(promise)
        .then(resolve)   // First to resolve
        .catch(reject);  // First to reject
    });
  });
};

// Example usage
const slow = new Promise((resolve) => setTimeout(() => resolve('slow'), 1000));
const fast = new Promise((resolve) => setTimeout(() => resolve('fast'), 100));

Promise.myRace([slow, fast])
  .then(result => console.log(result))  // 'fast'
  .catch(error => console.error(error));
```

```
// With rejection
const fastFail = Promise.reject('quick error');
Promise.myRace([slow, fastFail])
  .then(result => console.log(result))
  .catch(error => console.error(error));  // 'quick error'
```

Variant: Race with Timeout

```
function promiseWithTimeout(promise, timeoutMs, timeoutError = 'Timeout') {
  const timeout = new Promise((_, reject) =>
    setTimeout(() => reject(new Error(timeoutError)), timeoutMs)
  );

  return Promise.myRace([promise, timeout]);
}

// Usage
promiseWithTimeout(
  fetch('/api/slow-endpoint'),
  5000,
  'API request timed out'
)
  .then(response => console.log('Success:', response))
  .catch(error => console.error(error.message));
```

Advanced: Race with Cancellation

```
function promiseRaceWithCancel(promises) {
  let cancel;
  const cancelPromise = new Promise((_, reject) => {
    cancel = () => reject(new Error('Cancelled'));
  });

  const race = Promise.myRace([...promises, cancelPromise]);
  race.cancel = cancel;

  return race;
}

// Usage
const racePromise = promiseRaceWithCancel([
  fetch('/api/endpoint1'),
  fetch('/api/endpoint2'),
  fetch('/api/endpoint3')
]);

// Cancel if user navigates away
window.addEventListener('beforeunload', () => {
```

```
  racePromise.cancel();
});

racePromise
  .then(result => console.log('Winner:', result))
  .catch(error => console.error('Error or cancelled:', error));
```

10.2.4 Q4: Promise.finally() Polyfill

Problem: Execute cleanup logic after promise settles (resolved or rejected).

Solution:

```
/**
 * Promise.finally() polyfill
 * Always executes callback after promise settles
 */
Promise.prototype.myFinally = function(onFinally) {
  return this.then(
    value => Promise.resolve(onFinally()).then(() => value),
    reason => Promise.resolve(onFinally()).then(() => { throw reason; })
  );
};

// Example usage
fetch('/api/data')
  .then(response => response.json())
  .then(data => {
    console.log('Data:', data);
    return data;
  })
  .catch(error => {
    console.error('Error:', error);
    throw error;
  })
  .myFinally(() => {
    console.log('Cleanup: hiding loading spinner');
    hideLoadingSpinner();
  });
```

Variant: Finally with Async Cleanup

```
Promise.prototype.finallyAsync = function(onFinally) {
  return this.then(
    async value => {
      await onFinally();
      return value;
    },
    async reason => {
```

```
      await onFinally();
      throw reason;
    }
  );
};

// Usage
fetch('/api/data')
  .then(response => response.json())
  .finallyAsync(async () => {
    await saveToCache();
    await logMetrics();
    console.log('Async cleanup complete');
  });
```

Advanced: Finally with Error Handling

```
Promise.prototype.finallySafe = function(onFinally, onFinallyError) {
  return this.then(
    value => {
      try {
        const result = onFinally();
        return Promise.resolve(result).then(
          () => value,
          error => {
            if (onFinallyError) onFinallyError(error);
            return value;  // Preserve original value despite cleanup error
          }
        );
      } catch (error) {
        if (onFinallyError) onFinallyError(error);
        return value;
      }
    },
    reason => {
      try {
        const result = onFinally();
        return Promise.resolve(result).then(
          () => { throw reason; },
          error => {
            if (onFinallyError) onFinallyError(error);
            throw reason;  // Preserve original error
          }
        );
      } catch (error) {
        if (onFinallyError) onFinallyError(error);
        throw reason;
      }
```

```
    }
  );
};

// Usage
fetch('/api/data')
  .then(response => response.json())
  .finallySafe(
    () => {
      // May throw
      riskyCleanupOperation();
    },
    (cleanupError) => {
      console.error('Cleanup failed but continuing:', cleanupError);
    }
  );
```

10.2.5 Q5: Promise.allSettled() Polyfill

Problem: Wait for all promises to settle (fulfilled or rejected), return status of each.

Solution:

```
/**
 * Promise.allSettled() polyfill
 * Waits for all promises and returns array with status and value/reason
 */
Promise.myAllSettled = function(promises) {
  return new Promise((resolve) => {
    if (!Array.isArray(promises)) {
      return resolve([]);
    }

    if (promises.length === 0) {
      return resolve([]);
    }

    const results = [];
    let settledCount = 0;

    promises.forEach((promise, index) => {
      Promise.resolve(promise)
        .then(value => {
          results[index] = {
            status: 'fulfilled',
            value
          };
        })
```

```
        .catch(reason => {
          results[index] = {
            status: 'rejected',
            reason
          };
        })
        .finally(() => {
          settledCount++;
          if (settledCount === promises.length) {
            resolve(results);
          }
        });
    });
  });
};

// Example usage
const promises = [
  Promise.resolve('Success 1'),
  Promise.reject('Error 1'),
  Promise.resolve('Success 2'),
  new Promise((resolve) => setTimeout(() => resolve('Success 3'), 100))
];

Promise.myAllSettled(promises)
  .then(results => {
    results.forEach((result, index) => {
      if (result.status === 'fulfilled') {
        console.log(`Promise ${index}: ✓ ${result.value}`);
      } else {
        console.log(`Promise ${index}: ✗ ${result.reason}`);
      }
    });
  });

// Output:
// Promise 0: □ Success 1
// Promise 1: □ Error 1
// Promise 2: □ Success 2
// Promise 3: □ Success 3
```

Variant: With Statistics

```
Promise.allSettledWithStats = function(promises) {
  return Promise.myAllSettled(promises)
    .then(results => {
      const fulfilled = results.filter(r => r.status === 'fulfilled');
      const rejected = results.filter(r => r.status === 'rejected');
```

```
      return {
        results,
        stats: {
          total: results.length,
          fulfilled: fulfilled.length,
          rejected: rejected.length,
          successRate: (fulfilled.length / results.length) * 100
        }
      };
    });
};

// Usage
Promise.allSettledWithStats([
  fetch('/api/endpoint1'),
  fetch('/api/endpoint2'),
  fetch('/api/endpoint3')
])
  .then(({ results, stats }) => {
    console.log(`Success rate: ${stats.successRate.toFixed(1)}%`);
    console.log(`Fulfilled: ${stats.fulfilled}/${stats.total}`);
  });
```

Advanced: Retry Failed Promises

```
Promise.allSettledWithRetry = async function(promises, maxRetries = 3) {
  let results = await Promise.myAllSettled(promises);

  for (let attempt = 1; attempt <= maxRetries; attempt++) {
    const failedIndices = results
      .map((r, i) => r.status === 'rejected' ? i : -1)
      .filter(i => i !== -1);

    if (failedIndices.length === 0) break;

    console.log(`Retry attempt ${attempt}: retrying ${failedIndices.length} failed prom

    const retryPromises = failedIndices.map(i => promises[i]);
    const retryResults = await Promise.myAllSettled(retryPromises);

    failedIndices.forEach((originalIndex, retryIndex) => {
      if (retryResults[retryIndex].status === 'fulfilled') {
        results[originalIndex] = retryResults[retryIndex];
      }
    });
  }
```

```
  return results;
};

// Usage
Promise.allSettledWithRetry([
  fetch('/api/endpoint1'),
  fetch('/api/flaky-endpoint'),  // May fail
  fetch('/api/endpoint3')
], 3)
  .then(results => {
    console.log('Final results after retries:', results);
  });
```

10.2.6 Q6: Custom Promise Implementation

Problem: Implement your own minimal Promise class with then/catch/finally.

Solution:

```
/**
 * Custom Promise Implementation
 * Implements core Promise functionality from scratch
 */
class MyPromise {
  constructor(executor) {
    this.state = 'pending';  // pending, fulfilled, rejected
    this.value = undefined;
    this.reason = undefined;
    this.onFulfilledCallbacks = [];
    this.onRejectedCallbacks = [];

    const resolve = (value) => {
      if (this.state === 'pending') {
        this.state = 'fulfilled';
        this.value = value;
        this.onFulfilledCallbacks.forEach(fn => fn(value));
      }
    };

    const reject = (reason) => {
      if (this.state === 'pending') {
        this.state = 'rejected';
        this.reason = reason;
        this.onRejectedCallbacks.forEach(fn => fn(reason));
      }
    };

    try {
```

```
      executor(resolve, reject);
    } catch (error) {
      reject(error);
    }
  }

  then(onFulfilled, onRejected) {
    // Return new promise for chaining
    return new MyPromise((resolve, reject) => {
      const handleFulfilled = (value) => {
        try {
          if (typeof onFulfilled === 'function') {
            const result = onFulfilled(value);
            // Handle promise chaining
            if (result instanceof MyPromise) {
              result.then(resolve, reject);
            } else {
              resolve(result);
            }
          } else {
            resolve(value);
          }
        } catch (error) {
          reject(error);
        }
      };

      const handleRejected = (reason) => {
        try {
          if (typeof onRejected === 'function') {
            const result = onRejected(reason);
            if (result instanceof MyPromise) {
              result.then(resolve, reject);
            } else {
              resolve(result);
            }
          } else {
            reject(reason);
          }
        } catch (error) {
          reject(error);
        }
      };

      if (this.state === 'fulfilled') {
        setTimeout(() => handleFulfilled(this.value), 0);
      } else if (this.state === 'rejected') {
```

```
        setTimeout(() => handleRejected(this.reason), 0);
      } else {
        this.onFulfilledCallbacks.push(handleFulfilled);
        this.onRejectedCallbacks.push(handleRejected);
      }
    });
  }

  catch(onRejected) {
    return this.then(null, onRejected);
  }

  finally(onFinally) {
    return this.then(
      value => MyPromise.resolve(onFinally()).then(() => value),
      reason => MyPromise.resolve(onFinally()).then(() => { throw reason; })
    );
  }

  static resolve(value) {
    if (value instanceof MyPromise) {
      return value;
    }
    return new MyPromise((resolve) => resolve(value));
  }

  static reject(reason) {
    return new MyPromise((_, reject) => reject(reason));
  }
}

// Example usage
const promise = new MyPromise((resolve, reject) => {
  setTimeout(() => resolve('Success!'), 1000);
});

promise
  .then(result => {
    console.log(result);  // 'Success!'
    return result.toUpperCase();
  })
  .then(result => console.log(result))  // 'SUCCESS!'
  .catch(error => console.error('Error:', error))
  .finally(() => console.log('Done!'));
```

Advanced: Promise with State Inspection

```
class MyPromiseWithInspection extends MyPromise {
  getState() {
    return this.state;
  }

  getValue() {
    if (this.state === 'fulfilled') {
      return this.value;
    }
    throw new Error('Promise not fulfilled');
  }

  getReason() {
    if (this.state === 'rejected') {
      return this.reason;
    }
    throw new Error('Promise not rejected');
  }

  isPending() {
    return this.state === 'pending';
  }

  isFulfilled() {
    return this.state === 'fulfilled';
  }

  isRejected() {
    return this.state === 'rejected';
  }
}

// Usage
const p = new MyPromiseWithInspection((resolve) => {
  setTimeout(() => resolve(42), 100);
});

console.log(p.isPending());  // true
setTimeout(() => {
  console.log(p.isFulfilled());  // true
  console.log(p.getValue());     // 42
}, 150);
```

10.2.7 Q7: Execute Async Functions in Series

Problem: Run async tasks one by one sequentially.

Solution:

```
/**
 * Execute async functions in series
 * Each function waits for previous to complete
 */
async function executeInSeries(tasks) {
  const results = [];

  for (const task of tasks) {
    const result = await task();
    results.push(result);
  }

  return results;
}

// Example usage
const task1 = () => new Promise(resolve => setTimeout(() => resolve('Task 1'), 1000));
const task2 = () => new Promise(resolve => setTimeout(() => resolve('Task 2'), 500));
const task3 = () => new Promise(resolve => setTimeout(() => resolve('Task 3'), 200));

executeInSeries([task1, task2, task3])
  .then(results => console.log(results));  // ['Task 1', 'Task 2', 'Task 3']
```

Variant: With Error Handling

```
async function executeInSeriesWithErrorHandling(tasks, options = {}) {
  const {
    stopOnError = true,
    onProgress = null
  } = options;

  const results = [];
  const errors = [];

  for (let i = 0; i < tasks.length; i++) {
    try {
      const result = await tasks[i]();
      results.push({ success: true, value: result, index: i });

      if (onProgress) {
        onProgress({ completed: i + 1, total: tasks.length, result });
      }
    } catch (error) {
      errors.push({ index: i, error });
      results.push({ success: false, error, index: i });

      if (stopOnError) {
```

```
        break;
      }
    }
  }

  return { results, errors, allSucceeded: errors.length === 0 };
}

// Usage
executeInSeriesWithErrorHandling(
  [task1, task2, task3],
  {
    stopOnError: false,
    onProgress: ({ completed, total }) => {
      console.log(`Progress: ${completed}/${total}`);
    }
  }
).then(({ results, errors }) => {
  console.log('Completed with', errors.length, 'errors');
});
```

Advanced: Pipeline with Transformations

```
async function pipelineSeries(initialValue, ...operations) {
  let result = initialValue;

  for (const operation of operations) {
    result = await operation(result);
  }

  return result;
}

// Usage: Data processing pipeline
const fetchData = (id) => fetch(`/api/data/${id}`).then(r => r.json());
const validateData = (data) => {
  if (!data.valid) throw new Error('Invalid data');
  return data;
};
const transformData = (data) => ({ ...data, processed: true });
const saveData = (data) => fetch('/api/save', {
  method: 'POST',
  body: JSON.stringify(data)
}).then(r => r.json());

pipelineSeries(
  123,
  fetchData,
```

```
  validateData,
  transformData,
  saveData
)
  .then(result => console.log('Pipeline complete:', result))
  .catch(error => console.error('Pipeline failed:', error));
```

10.2.8 Q8: Execute Async Functions in Parallel

Problem: Run async tasks concurrently.

Solution:

```
/**
 * Execute async functions in parallel
 * All tasks start immediately
 */
async function executeInParallel(tasks) {
  const promises = tasks.map(task => task());
  return Promise.all(promises);
}

// Example usage
const task1 = () => new Promise(resolve => setTimeout(() => resolve('Task 1'), 1000));
const task2 = () => new Promise(resolve => setTimeout(() => resolve('Task 2'), 500));
const task3 = () => new Promise(resolve => setTimeout(() => resolve('Task 3'), 200));

executeInParallel([task1, task2, task3])
  .then(results => console.log(results));  // ['Task 1', 'Task 2', 'Task 3']
  // Completes in ~1000ms (not 1700ms)
```

Variant: With Concurrency Limit

```
async function executeInParallelWithLimit(tasks, limit) {
  const results = [];
  const executing = [];

  for (const [index, task] of tasks.entries()) {
    const promise = task().then(result => {
      results[index] = result;
      return result;
    });

    results[index] = promise;

    if (limit <= tasks.length) {
      const executing = promise.then(() =>
        executing.splice(executing.indexOf(promise), 1)
```

```
      );
      executing.push(executing);

      if (executing.length >= limit) {
        await Promise.race(executing);
      }
    }
  }

  return Promise.all(results);
}

// Usage: Run max 3 tasks at a time
executeInParallelWithLimit([task1, task2, task3, task4, task5], 3)
  .then(results => console.log('All done:', results));
```

Advanced: Parallel with Progress and Cancellation

```
class ParallelExecutor {
  constructor(tasks, options = {}) {
    this.tasks = tasks;
    this.limit = options.limit || Infinity;
    this.onProgress = options.onProgress;
    this.cancelled = false;
    this.results = [];
    this.errors = [];
  }

  async execute() {
    const executing = [];
    let completed = 0;

    for (let i = 0; i < this.tasks.length; i++) {
      if (this.cancelled) break;

      const promise = this.tasks[i]()
        .then(result => {
          this.results[i] = { success: true, value: result };
          completed++;

          if (this.onProgress) {
            this.onProgress({
              completed,
              total: this.tasks.length,
              index: i,
              result
            });
          }
```

```
        })
        .catch(error => {
          this.errors.push({ index: i, error });
          this.results[i] = { success: false, error };
          completed++;
        });

      executing.push(promise);

      if (executing.length >= this.limit) {
        await Promise.race(executing);
        executing.splice(executing.findIndex(p => p === promise), 1);
      }
    }

    await Promise.all(executing);
    return {
      results: this.results,
      errors: this.errors,
      cancelled: this.cancelled
    };
  }

  cancel() {
    this.cancelled = true;
  }
}

// Usage
const executor = new ParallelExecutor(
  [task1, task2, task3, task4, task5],
  {
    limit: 3,
    onProgress: ({ completed, total }) => {
      console.log(`Progress: ${completed}/${total}`);
      updateProgressBar(completed / total);
    }
  }
);

// Cancel after 2 seconds
setTimeout(() => executor.cancel(), 2000);

executor.execute()
  .then(({ results, errors, cancelled }) => {
    console.log('Results:', results);
    console.log('Errors:', errors);
```

```
    console.log('Was cancelled:', cancelled);
  });
```

10.2.9 Q9: Retry Promises N Times

Problem: Retry failed promises with exponential backoff.

Solution:

```
/**
 * Retry promise N times with delay
 */
async function retryPromise(promiseFactory, maxRetries = 3, delay = 1000) {
  let lastError;

  for (let attempt = 0; attempt < maxRetries; attempt++) {
    try {
      return await promiseFactory();
    } catch (error) {
      lastError = error;

      if (attempt < maxRetries - 1) {
        console.log(`Attempt ${attempt + 1} failed, retrying in ${delay}ms...`);
        await new Promise(resolve => setTimeout(resolve, delay));
        delay *= 2;  // Exponential backoff
      }
    }
  }

  throw new Error(`Failed after ${maxRetries} attempts: ${lastError.message}`);
}

// Example usage
const unreliableAPI = () => fetch('/api/flaky-endpoint');

retryPromise(unreliableAPI, 5, 1000)
  .then(response => response.json())
  .then(data => console.log('Success:', data))
  .catch(error => console.error('Failed after retries:', error));
```

Variant: With Custom Retry Logic

```
async function retryWithStrategy(promiseFactory, options = {}) {
  const {
    maxRetries = 3,
    initialDelay = 1000,
    maxDelay = 30000,
    backoffMultiplier = 2,
    shouldRetry = () => true,
```

```
    onRetry = null
  } = options;

  let delay = initialDelay;
  let lastError;

  for (let attempt = 0; attempt < maxRetries; attempt++) {
    try {
      return await promiseFactory();
    } catch (error) {
      lastError = error;

      // Check if should retry this error
      if (!shouldRetry(error, attempt)) {
        throw error;
      }

      if (attempt < maxRetries - 1) {
        if (onRetry) {
          onRetry(error, attempt + 1, delay);
        }

        await new Promise(resolve => setTimeout(resolve, delay));
        delay = Math.min(delay * backoffMultiplier, maxDelay);
      }
    }
  }

  throw lastError;
}

// Usage
retryWithStrategy(
  () => fetch('/api/data').then(r => {
    if (!r.ok) throw new Error(`HTTP ${r.status}`);
    return r.json();
  }),
  {
    maxRetries: 5,
    initialDelay: 1000,
    maxDelay: 10000,
    backoffMultiplier: 2,
    shouldRetry: (error, attempt) => {
      // Don't retry 4xx errors
      if (error.message.includes('HTTP 4')) {
        return false;
      }
```

```
      return true;
    },
    onRetry: (error, attempt, delay) => {
      console.log(`Retry ${attempt} after ${delay}ms. Error: ${error.message}`);
    }
  }
);
```

Advanced: Retry with Circuit Breaker

```
class RetryWithCircuitBreaker {
  constructor(options = {}) {
    this.maxRetries = options.maxRetries || 3;
    this.maxFailures = options.maxFailures || 5;
    this.resetTimeout = options.resetTimeout || 60000;

    this.failures = 0;
    this.state = 'closed';  // closed, open, half-open
    this.nextAttempt = Date.now();
  }

  async execute(promiseFactory) {
    // Circuit breaker is open
    if (this.state === 'open') {
      if (Date.now() < this.nextAttempt) {
        throw new Error('Circuit breaker is open');
      }
      // Try half-open state
      this.state = 'half-open';
    }

    try {
      const result = await retryPromise(promiseFactory, this.maxRetries);

      // Success - reset circuit breaker
      if (this.state === 'half-open') {
        this.state = 'closed';
      }
      this.failures = 0;

      return result;
    } catch (error) {
      this.failures++;

      // Open circuit breaker if too many failures
      if (this.failures >= this.maxFailures) {
        this.state = 'open';
        this.nextAttempt = Date.now() + this.resetTimeout;
```

```
        console.log('Circuit breaker opened');
      }

      throw error;
    }
  }

  getState() {
    return {
      state: this.state,
      failures: this.failures,
      nextAttempt: this.nextAttempt
    };
  }

  reset() {
    this.failures = 0;
    this.state = 'closed';
    this.nextAttempt = Date.now();
  }
}

// Usage
const breaker = new RetryWithCircuitBreaker({
  maxRetries: 3,
  maxFailures: 5,
  resetTimeout: 60000
});

async function callAPI() {
  try {
    return await breaker.execute(() => fetch('/api/data'));
  } catch (error) {
    console.error('API call failed:', error.message);
    console.log('Circuit breaker state:', breaker.getState());
  }
}
```

10.2.10 Q10: mapSeries Async Function

Problem: Process async items sequentially and map results.

Solution:

```
/**
 * Async map that processes items in series
 */
async function mapSeries(array, asyncFn) {
```

```
  const results = [];

  for (let i = 0; i < array.length; i++) {
    const result = await asyncFn(array[i], i, array);
    results.push(result);
  }

  return results;
}

// Example usage
const numbers = [1, 2, 3, 4, 5];

const results = await mapSeries(numbers, async (num) => {
  await new Promise(resolve => setTimeout(resolve, 100));
  return num * 2;
});

console.log(results);  // [2, 4, 6, 8, 10]
```

Variant: With Progress and Error Handling

```
async function mapSeriesWithProgress(array, asyncFn, options = {}) {
  const {
    onProgress = null,
    onError = null,
    continueOnError = false
  } = options;

  const results = [];
  const errors = [];

  for (let i = 0; i < array.length; i++) {
    try {
      const result = await asyncFn(array[i], i, array);
      results.push({ success: true, value: result, index: i });

      if (onProgress) {
        onProgress({
          completed: i + 1,
          total: array.length,
          currentItem: array[i],
          result
        });
      }
    } catch (error) {
      errors.push({ index: i, item: array[i], error });
      results.push({ success: false, error, index: i });
```

```
      if (onError) {
        onError(error, array[i], i);
      }

      if (!continueOnError) {
        throw error;
      }
    }
  }

  return { results, errors, allSucceeded: errors.length === 0 };
}

// Usage
const users = [1, 2, 3, 4, 5];

mapSeriesWithProgress(
  users,
  async (userId) => {
    const response = await fetch(`/api/users/${userId}`);
    return response.json();
  },
  {
    onProgress: ({ completed, total, result }) => {
      console.log(`Processing ${completed}/${total}`, result.name);
      updateProgressBar(completed / total);
    },
    continueOnError: true,
    onError: (error, userId) => {
      console.error(`Failed to fetch user ${userId}:`, error);
    }
  }
).then(({ results, errors }) => {
  console.log(`Completed: ${results.length - errors.length} succeeded`);
});
```

Advanced: mapSeries with Accumulator

```
async function mapSeriesWithAccumulator(array, asyncFn, initialAccumulator) {
  const results = [];
  let accumulator = initialAccumulator;

  for (let i = 0; i < array.length; i++) {
    const result = await asyncFn(accumulator, array[i], i, array);
    results.push(result.value);
    accumulator = result.accumulator;
  }
```

```
  return { results, finalAccumulator: accumulator };
}

// Usage: Running sum with async operations
const { results, finalAccumulator } = await mapSeriesWithAccumulator(
  [1, 2, 3, 4, 5],
  async (acc, num) => {
    await new Promise(resolve => setTimeout(resolve, 100));
    const squared = num * num;
    return {
      value: squared,
      accumulator: acc + squared
    };
  },
  0
);

console.log('Results:', results);          // [1, 4, 9, 16, 25]
console.log('Running sum:', finalAccumulator);  // 55
```

10.2.11 Q11: mapLimit Async Function

Problem: Run async tasks with a concurrency limit.

Solution:

```
/**
 * Async map with concurrency limit
 * Only N tasks run simultaneously
 */
async function mapLimit(array, limit, asyncFn) {
  const results = [];
  const executing = [];

  for (let i = 0; i < array.length; i++) {
    const promise = asyncFn(array[i], i, array).then(result => {
      executing.splice(executing.indexOf(promise), 1);
      return result;
    });

    results[i] = promise;
    executing.push(promise);

    if (executing.length >= limit) {
      await Promise.race(executing);
    }
  }
```

```
  return Promise.all(results);
}

// Example usage
const urls = Array.from({ length: 20 }, (_, i) => `/api/item/${i}`);

const data = await mapLimit(urls, 3, async (url) => {
  console.log('Fetching:', url);
  const response = await fetch(url);
  return response.json();
});

console.log('All data fetched:', data.length);  // 20 items, 3 at a time
```

Variant: With Error Handling and Retry

```
async function mapLimitWithRetry(array, limit, asyncFn, options = {}) {
  const { maxRetries = 3, onError = null, onProgress = null } = options;

  const results = [];
  const errors = [];
  const executing = [];
  let completed = 0;

  for (let i = 0; i < array.length; i++) {
    const executeWithRetry = async () => {
      let lastError;

      for (let attempt = 0; attempt < maxRetries; attempt++) {
        try {
          const result = await asyncFn(array[i], i, array);
          completed++;

          if (onProgress) {
            onProgress({ completed, total: array.length, index: i });
          }

          return { success: true, value: result, index: i };
        } catch (error) {
          lastError = error;
          if (attempt < maxRetries - 1) {
            await new Promise(resolve => setTimeout(resolve, 1000 * Math.pow(2, attempt
          }
        }
      }

      errors.push({ index: i, item: array[i], error: lastError });
```

```
      if (onError) {
        onError(lastError, array[i], i);
      }

      return { success: false, error: lastError, index: i };
    };

    const promise = executeWithRetry().then(result => {
      executing.splice(executing.indexOf(promise), 1);
      return result;
    });

    results[i] = promise;
    executing.push(promise);

    if (executing.length >= limit) {
      await Promise.race(executing);
    }
  }

  const finalResults = await Promise.all(results);
  return { results: finalResults, errors, allSucceeded: errors.length === 0 };
}

// Usage
mapLimitWithRetry(
  urls,
  5,
  async (url) => {
    const response = await fetch(url);
    if (!response.ok) throw new Error(`Failed: ${url}`);
    return response.json();
  },
  {
    maxRetries: 3,
    onProgress: ({ completed, total }) => {
      console.log(`Progress: ${completed}/${total}`);
    },
    onError: (error, url) => {
      console.error(`Failed after retries: ${url}`, error);
    }
  }
);
```

Advanced: Dynamic Concurrency Adjustment

```
class AdaptiveMapLimit {
  constructor(options = {}) {
    this.minLimit = options.minLimit || 1;
    this.maxLimit = options.maxLimit || 10;
    this.currentLimit = options.initialLimit || 3;
    this.successRate = 1;
    this.adjustInterval = options.adjustInterval || 5;
  }

  async execute(array, asyncFn) {
    const results = [];
    const executing = [];
    let completed = 0;
    let succeeded = 0;

    for (let i = 0; i < array.length; i++) {
      const promise = asyncFn(array[i], i, array)
        .then(result => {
          succeeded++;
          completed++;
          executing.splice(executing.indexOf(promise), 1);
          return { success: true, value: result };
        })
        .catch(error => {
          completed++;
          executing.splice(executing.indexOf(promise), 1);
          return { success: false, error };
        });

      results[i] = promise;
      executing.push(promise);

      // Adjust concurrency based on success rate
      if (completed > 0 && completed % this.adjustInterval === 0) {
        this.successRate = succeeded / completed;

        if (this.successRate > 0.9 && this.currentLimit < this.maxLimit) {
          this.currentLimit++;
          console.log(`Increasing concurrency to ${this.currentLimit}`);
        } else if (this.successRate < 0.7 && this.currentLimit > this.minLimit) {
          this.currentLimit--;
          console.log(`Decreasing concurrency to ${this.currentLimit}`);
        }
      }

      if (executing.length >= this.currentLimit) {
        await Promise.race(executing);
```

```
      }
    }

    return Promise.all(results);
  }
}

// Usage: Automatically adjusts concurrency based on success rate
const mapper = new AdaptiveMapLimit({
  minLimit: 1,
  maxLimit: 10,
  initialLimit: 3,
  adjustInterval: 5
});

const results = await mapper.execute(urls, async (url) => {
  const response = await fetch(url);
  return response.json();
});
```

10.2.12 Q12: Async Filter Function

Problem: Filter array asynchronously with predicate logic.

Solution:

```
/**
 * Async filter function
 * Filters array using async predicate
 */
async function asyncFilter(array, asyncPredicate) {
  const results = await Promise.all(
    array.map(async (item, index) => {
      const shouldInclude = await asyncPredicate(item, index, array);
      return shouldInclude ? item : null;
    })
  );

  return results.filter(item => item !== null);
}

// Example usage
const users = [
  { id: 1, name: 'Alice' },
  { id: 2, name: 'Bob' },
  { id: 3, name: 'Charlie' }
];
```

```
const activeUsers = await asyncFilter(users, async (user) => {
  const response = await fetch(`/api/users/${user.id}/status`);
  const { active } = await response.json();
  return active;
});

console.log('Active users:', activeUsers);
```

Variant: Sequential Filter

```
async function asyncFilterSeries(array, asyncPredicate) {
  const results = [];

  for (let i = 0; i < array.length; i++) {
    const shouldInclude = await asyncPredicate(array[i], i, array);
    if (shouldInclude) {
      results.push(array[i]);
    }
  }

  return results;
}

// Usage: Processes one item at a time
const filtered = await asyncFilterSeries(users, async (user) => {
  const status = await checkUserStatus(user.id);
  return status.active && status.verified;
});
```

Advanced: Filter with Concurrency Limit

```
async function asyncFilterLimit(array, limit, asyncPredicate) {
  const results = new Array(array.length);
  const executing = [];

  for (let i = 0; i < array.length; i++) {
    const promise = asyncPredicate(array[i], i, array)
      .then(shouldInclude => {
        results[i] = shouldInclude ? array[i] : null;
        executing.splice(executing.indexOf(promise), 1);
      })
      .catch(() => {
        results[i] = null;
        executing.splice(executing.indexOf(promise), 1);
      });

    executing.push(promise);
```

```
    if (executing.length >= limit) {
      await Promise.race(executing);
    }
  }

  await Promise.all(executing);
  return results.filter(item => item !== null);
}

// Usage: Max 5 concurrent checks
const verified = await asyncFilterLimit(users, 5, async (user) => {
  const response = await fetch(`/api/verify/${user.id}`);
  const data = await response.json();
  return data.verified;
});
```

10.2.13 Q13: Async Reject Function

Problem: Reject/filter out items asynchronously (opposite of filter).

Solution:

```
/**
 * Async reject function
 * Removes items that match async predicate
 */
async function asyncReject(array, asyncPredicate) {
  return asyncFilter(array, async (item, index, arr) => {
    const shouldReject = await asyncPredicate(item, index, arr);
    return !shouldReject;
  });
}

// Example usage
const allUsers = [
  { id: 1, name: 'Alice', email: 'alice@example.com' },
  { id: 2, name: 'Bob', email: 'bob@spam.com' },
  { id: 3, name: 'Charlie', email: 'charlie@example.com' }
];

const validUsers = await asyncReject(allUsers, async (user) => {
  const response = await fetch(`/api/check-spam/${user.email}`);
  const { isSpam } = await response.json();
  return isSpam;
});

console.log('Valid users:', validUsers);  // Alice and Charlie
```

Variant: With Reason Tracking

```
async function asyncRejectWithReasons(array, asyncPredicate) {
  const rejected = [];
  const accepted = [];

  await Promise.all(
    array.map(async (item, index) => {
      const result = await asyncPredicate(item, index, array);

      if (result.shouldReject) {
        rejected.push({ item, reason: result.reason, index });
      } else {
        accepted.push(item);
      }
    })
  );

  return { accepted, rejected };
}

// Usage
const { accepted, rejected } = await asyncRejectWithReasons(
  allUsers,
  async (user) => {
    const checks = await Promise.all([
      checkEmail(user.email),
      checkDomain(user.email),
      checkReputation(user.id)
    ]);

    const failures = checks.filter(c => !c.valid);

    return {
      shouldReject: failures.length > 0,
      reason: failures.map(f => f.reason).join(', ')
    };
  }
);

console.log('Accepted:', accepted.length);
console.log('Rejected:', rejected.length, 'reasons:', rejected.map(r => r.reason));
```

10.2.14 Q14: Execute Promises with Priority

Problem: Execute promises based on assigned priority.

Solution:

```javascript
/**
 * Priority Queue for Promises
 * Executes higher priority tasks first
 */
class PriorityPromiseQueue {
  constructor(concurrency = 1) {
    this.concurrency = concurrency;
    this.queue = [];
    this.running = 0;
  }

  add(promiseFactory, priority = 0) {
    return new Promise((resolve, reject) => {
      this.queue.push({
        promiseFactory,
        priority,
        resolve,
        reject
      });

      // Sort by priority (higher first)
      this.queue.sort((a, b) => b.priority - a.priority);

      this.process();
    });
  }

  async process() {
    if (this.running >= this.concurrency || this.queue.length === 0) {
      return;
    }

    this.running++;
    const { promiseFactory, resolve, reject } = this.queue.shift();

    try {
      const result = await promiseFactory();
      resolve(result);
    } catch (error) {
      reject(error);
    } finally {
      this.running--;
      this.process();
    }
  }
}
```

```
// Example usage
const queue = new PriorityPromiseQueue(2);

// Add tasks with different priorities
queue.add(() => fetch('/api/low-priority'), 1)
  .then(data => console.log('Low priority done'));

queue.add(() => fetch('/api/high-priority'), 10)
  .then(data => console.log('High priority done'));  // Executes first

queue.add(() => fetch('/api/medium-priority'), 5)
  .then(data => console.log('Medium priority done'));
```

Variant: With Dynamic Priority

```
class DynamicPriorityQueue extends PriorityPromiseQueue {
  add(promiseFactory, priority = 0, options = {}) {
    const task = {
      promiseFactory,
      priority,
      addedAt: Date.now(),
      deadline: options.deadline,
      agingRate: options.agingRate || 0.1,
      resolve: null,
      reject: null
    };

    return new Promise((resolve, reject) => {
      task.resolve = resolve;
      task.reject = reject;
      this.queue.push(task);
      this.updatePriorities();
      this.process();
    });
  }

  updatePriorities() {
    const now = Date.now();

    this.queue.forEach(task => {
      // Age-based priority boost
      const age = (now - task.addedAt) / 1000;  // seconds
      const agingBoost = age * task.agingRate;

      // Deadline urgency boost
      let deadlineBoost = 0;
      if (task.deadline) {
        const timeLeft = (task.deadline - now) / 1000;
```

```
        if (timeLeft < 60) {
          deadlineBoost = 100 / timeLeft;  // More urgent as deadline approaches
        }
      }

      task.effectivePriority = task.priority + agingBoost + deadlineBoost;
    });

    this.queue.sort((a, b) => b.effectivePriority - a.effectivePriority);
  }

  async process() {
    this.updatePriorities();
    super.process();
  }
}

// Usage
const dynamicQueue = new DynamicPriorityQueue(3);

// Task with deadline
dynamicQueue.add(
  () => fetch('/api/urgent'),
  5,
  { deadline: Date.now() + 10000 }  // 10 seconds deadline
);

// Task that gains priority over time
dynamicQueue.add(
  () => fetch('/api/patient'),
  1,
  { agingRate: 0.5 }  // Gains 0.5 priority per second
);
```

10.2.15 Q15: Dependent Async Tasks

Problem: Chain async tasks that depend on previous results.

Solution:

```
/**
 * Execute dependent async tasks
 * Each task receives results from previous tasks
 */
async function executeDependentTasks(tasks) {
  const results = [];

  for (const task of tasks) {
```

```
    const result = await task(results);
    results.push(result);
  }

  return results;
}

// Example usage
const dependentTasks = [
  async () => {
    const user = await fetch('/api/user').then(r => r.json());
    return user;
  },
  async ([user]) => {
    const posts = await fetch(`/api/users/${user.id}/posts`).then(r => r.json());
    return posts;
  },
  async ([user, posts]) => {
    const comments = await Promise.all(
      posts.map(post => fetch(`/api/posts/${post.id}/comments`).then(r => r.json()))
    );
    return comments.flat();
  }
];

const [user, posts, comments] = await executeDependentTasks(dependentTasks);
console.log('User:', user.name);
console.log('Posts:', posts.length);
console.log('Comments:', comments.length);
```

Variant: Dependency Graph Execution

```
class DependencyGraph {
  constructor() {
    this.tasks = new Map();
    this.results = new Map();
  }

  addTask(name, asyncFn, dependencies = []) {
    this.tasks.set(name, {
      fn: asyncFn,
      dependencies
    });
  }

  async execute(taskName) {
    // Return cached result if already executed
    if (this.results.has(taskName)) {
```

```
      return this.results.get(taskName);
    }

    const task = this.tasks.get(taskName);
    if (!task) {
      throw new Error(`Task ${taskName} not found`);
    }

    // Execute dependencies first
    const depResults = await Promise.all(
      task.dependencies.map(dep => this.execute(dep))
    );

    // Execute this task with dependency results
    const result = await task.fn(...depResults);
    this.results.set(taskName, result);

    return result;
  }

  async executeAll() {
    const allTasks = Array.from(this.tasks.keys());
    const results = await Promise.all(
      allTasks.map(name => this.execute(name))
    );

    return Object.fromEntries(
      allTasks.map((name, i) => [name, results[i]])
    );
  }
}

// Example usage
const graph = new DependencyGraph();

graph.addTask('fetchUser', async () => {
  return fetch('/api/user').then(r => r.json());
});

graph.addTask('fetchPosts', async (user) => {
  return fetch(`/api/users/${user.id}/posts`).then(r => r.json());
}, ['fetchUser']);

graph.addTask('fetchComments', async (user, posts) => {
  const comments = await Promise.all(
    posts.map(post => fetch(`/api/posts/${post.id}/comments`).then(r => r.json()))
  );
```

```
  return comments.flat();
}, ['fetchUser', 'fetchPosts']);

graph.addTask('generateReport', async (user, posts, comments) => {
  return {
    user: user.name,
    totalPosts: posts.length,
    totalComments: comments.length,
    avgCommentsPerPost: comments.length / posts.length
  };
}, ['fetchUser', 'fetchPosts', 'fetchComments']);

const results = await graph.executeAll();
console.log('Report:', results.generateReport);
```

10.2.16 Q16: Pausable Auto Incrementor

Problem: Timer-based counter that can pause/resume.

Solution:

```
/**
 * Pausable auto incrementor
 * Counter that automatically increments and can be paused
 */
class PausableIncrementor {
  constructor(interval = 1000, callback = null) {
    this.interval = interval;
    this.callback = callback;
    this.value = 0;
    this.timerId = null;
    this.isPaused = true;
  }

  start() {
    if (!this.isPaused) return;

    this.isPaused = false;
    this.timerId = setInterval(() => {
      this.value++;
      if (this.callback) {
        this.callback(this.value);
      }
    }, this.interval);
  }

  pause() {
    if (this.isPaused) return;
```

```
    this.isPaused = true;
    clearInterval(this.timerId);
    this.timerId = null;
  }

  resume() {
    this.start();
  }

  reset() {
    this.pause();
    this.value = 0;
  }

  getValue() {
    return this.value;
  }

  stop() {
    this.pause();
    this.value = 0;
  }
}

// Example usage
const counter = new PausableIncrementor(1000, (value) => {
  console.log('Counter:', value);
  document.getElementById('counter').textContent = value;
});

counter.start();  // Starts incrementing

setTimeout(() => counter.pause(), 5000);   // Pause after 5 seconds
setTimeout(() => counter.resume(), 8000);  // Resume after 8 seconds
setTimeout(() => counter.stop(), 12000);   // Stop after 12 seconds
```

Variant: With Step Control

```
class AdvancedIncrementor extends PausableIncrementor {
  constructor(options = {}) {
    super(options.interval, options.callback);
    this.step = options.step || 1;
    this.max = options.max;
    this.min = options.min || 0;
  }

  start() {
```

```
    if (!this.isPaused) return;

    this.isPaused = false;
    this.timerId = setInterval(() => {
      this.value += this.step;

      // Check boundaries
      if (this.max !== undefined && this.value >= this.max) {
        this.value = this.max;
        this.pause();
      }

      if (this.min !== undefined && this.value <= this.min) {
        this.value = this.min;
        this.pause();
      }

      if (this.callback) {
        this.callback(this.value);
      }
    }, this.interval);
  }

  setStep(newStep) {
    this.step = newStep;
  }

  increment() {
    this.value += this.step;
    if (this.callback) {
      this.callback(this.value);
    }
  }

  decrement() {
    this.value -= this.step;
    if (this.callback) {
      this.callback(this.value);
    }
  }
}

// Usage
const timer = new AdvancedIncrementor({
  interval: 100,
  step: 1,
  min: 0,
```

```
  max: 100,
  callback: (value) => {
    updateProgressBar(value / 100);
    if (value === 100) {
      console.log('Completed!');
    }
  }
});

timer.start();  // Auto-increments from 0 to 100, then stops
```

Advanced: Multi-Speed Incrementor

```
class MultiSpeedIncrementor extends AdvancedIncrementor {
  constructor(options = {}) {
    super(options);
    this.speeds = options.speeds || {
      slow: 2000,
      normal: 1000,
      fast: 500,
      turbo: 100
    };
    this.currentSpeed = 'normal';
  }

  setSpeed(speed) {
    if (!(speed in this.speeds)) {
      throw new Error(`Invalid speed: ${speed}`);
    }

    const wasRunning = !this.isPaused;
    if (wasRunning) {
      this.pause();
    }

    this.currentSpeed = speed;
    this.interval = this.speeds[speed];

    if (wasRunning) {
      this.start();
    }
  }

  getSpeed() {
    return this.currentSpeed;
  }
}
```

```
// Usage: Speed control for animations or progress
const animator = new MultiSpeedIncrementor({
  speeds: { slow: 2000, normal: 1000, fast: 500, turbo: 100 },
  callback: (frame) => {
    renderFrame(frame);
  }
});

animator.start();
document.getElementById('speed-slider').addEventListener('change', (e) => {
  animator.setSpeed(e.target.value);  // 'slow', 'normal', 'fast', 'turbo'
});
```

10.2.17 Q17: Queue Using Stacks

Problem: Implement queue data structure using two stacks.

Solution:

```
/**
 * Queue implementation using two stacks
 * FIFO behavior using two LIFO structures
 */
class QueueWithStacks {
  constructor() {
    this.stack1 = [];  // For enqueue
    this.stack2 = [];  // For dequeue
  }

  enqueue(item) {
    this.stack1.push(item);
  }

  dequeue() {
    // Move items from stack1 to stack2 if stack2 is empty
    if (this.stack2.length === 0) {
      while (this.stack1.length > 0) {
        this.stack2.push(this.stack1.pop());
      }
    }

    if (this.stack2.length === 0) {
      return undefined;
    }

    return this.stack2.pop();
  }
```

```
  peek() {
    if (this.stack2.length === 0) {
      while (this.stack1.length > 0) {
        this.stack2.push(this.stack1.pop());
      }
    }

    return this.stack2[this.stack2.length - 1];
  }

  isEmpty() {
    return this.stack1.length === 0 && this.stack2.length === 0;
  }

  size() {
    return this.stack1.length + this.stack2.length;
  }
}

// Example usage
const queue = new QueueWithStacks();

queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);

console.log(queue.dequeue());  // 1 (FIFO)
console.log(queue.dequeue());  // 2
console.log(queue.peek());     // 3
console.log(queue.size());     // 1
```

Variant: With Priority

```
class PriorityQueueWithStacks {
  constructor() {
    this.highPriority = new QueueWithStacks();
    this.normalPriority = new QueueWithStacks();
    this.lowPriority = new QueueWithStacks();
  }

  enqueue(item, priority = 'normal') {
    switch (priority) {
      case 'high':
        this.highPriority.enqueue(item);
        break;
      case 'low':
        this.lowPriority.enqueue(item);
        break;
```

```
      default:
        this.normalPriority.enqueue(item);
    }
  }

  dequeue() {
    if (!this.highPriority.isEmpty()) {
      return this.highPriority.dequeue();
    }
    if (!this.normalPriority.isEmpty()) {
      return this.normalPriority.dequeue();
    }
    if (!this.lowPriority.isEmpty()) {
      return this.lowPriority.dequeue();
    }
    return undefined;
  }

  isEmpty() {
    return this.highPriority.isEmpty() &&
           this.normalPriority.isEmpty() &&
           this.lowPriority.isEmpty();
  }
}

// Usage
const pQueue = new PriorityQueueWithStacks();

pQueue.enqueue('Task A', 'low');
pQueue.enqueue('Task B', 'high');
pQueue.enqueue('Task C', 'normal');

console.log(pQueue.dequeue());  // 'Task B' (high priority)
console.log(pQueue.dequeue());  // 'Task C' (normal priority)
console.log(pQueue.dequeue());  // 'Task A' (low priority)
```

10.2.18 Q18: Stack Using Queues

Problem: Implement stack using two queues.

Solution:

```
/**
 * Stack implementation using two queues
 * LIFO behavior using two FIFO structures
 */
class StackWithQueues {
  constructor() {
```

```
    this.queue1 = [];
    this.queue2 = [];
  }

  push(item) {
    // Add to queue2
    this.queue2.push(item);

    // Move all items from queue1 to queue2
    while (this.queue1.length > 0) {
      this.queue2.push(this.queue1.shift());
    }

    // Swap queues
    [this.queue1, this.queue2] = [this.queue2, this.queue1];
  }

  pop() {
    if (this.queue1.length === 0) {
      return undefined;
    }
    return this.queue1.shift();
  }

  peek() {
    if (this.queue1.length === 0) {
      return undefined;
    }
    return this.queue1[0];
  }

  isEmpty() {
    return this.queue1.length === 0;
  }

  size() {
    return this.queue1.length;
  }
}

// Example usage
const stack = new StackWithQueues();

stack.push(1);
stack.push(2);
stack.push(3);
```

```
console.log(stack.pop());   // 3 (LIFO)
console.log(stack.pop());   // 2
console.log(stack.peek());  // 1
console.log(stack.size());  // 1
```

Variant: Optimized with Single Queue

```
class OptimizedStackWithQueue {
  constructor() {
    this.queue = [];
  }

  push(item) {
    const size = this.queue.length;
    this.queue.push(item);

    // Rotate queue to put new item at front
    for (let i = 0; i < size; i++) {
      this.queue.push(this.queue.shift());
    }
  }

  pop() {
    return this.queue.shift();
  }

  peek() {
    return this.queue[0];
  }

  isEmpty() {
    return this.queue.length === 0;
  }

  size() {
    return this.queue.length;
  }

  toArray() {
    return [...this.queue];
  }
}

// Usage
const optimizedStack = new OptimizedStackWithQueue();

optimizedStack.push('A');
optimizedStack.push('B');
```

```
optimizedStack.push('C');

console.log(optimizedStack.toArray());  // ['C', 'B', 'A']
console.log(optimizedStack.pop());      // 'C'
console.log(optimizedStack.pop());      // 'B'
```

10.2.19 Question: Implement debounce and throttle.

Answer:

```
// Debounce: Execute after delay, reset timer on new calls
function debounce(func, delay) {
  let timeoutId;

  return function(...args) {
    clearTimeout(timeoutId);

    timeoutId = setTimeout(() => {
      func.apply(this, args);
    }, delay);
  };
}

// Usage
const search = debounce((query) => {
  console.log('Searching for:', query);
  // API call...
}, 300);

input.addEventListener('input', (e) => {
  search(e.target.value);
});

// Throttle: Execute at most once per delay period
function throttle(func, delay) {
  let lastCall = 0;

  return function(...args) {
    const now = Date.now();

    if (now - lastCall >= delay) {
      lastCall = now;
      func.apply(this, args);
    }
  };
}
```

```
// Usage
const handleScroll = throttle(() => {
  console.log('Scroll position:', window.scrollY);
}, 100);

window.addEventListener('scroll', handleScroll);

// Throttle with leading and trailing
function throttleAdvanced(func, delay, options = {}) {
  let timeoutId;
  let lastCall = 0;
  const { leading = true, trailing = true } = options;

  return function(...args) {
    const now = Date.now();

    if (!lastCall && !leading) {
      lastCall = now;
    }

    const remaining = delay - (now - lastCall);

    if (remaining <= 0) {
      if (timeoutId) {
        clearTimeout(timeoutId);
        timeoutId = null;
      }

      lastCall = now;
      func.apply(this, args);
    } else if (!timeoutId && trailing) {
      timeoutId = setTimeout(() => {
        lastCall = !leading ? 0 : Date.now();
        timeoutId = null;
        func.apply(this, args);
      }, remaining);
    }
  };
}
```

10.2.20 Question: Implement deep clone.

Answer:

```
function deepClone(obj, hash = new WeakMap()) {
  // Handle primitives and null
  if (obj === null || typeof obj !== 'object') {
```

```
  return obj;
}

// Handle circular references
if (hash.has(obj)) {
  return hash.get(obj);
}

// Handle Date
if (obj instanceof Date) {
  return new Date(obj);
}

// Handle RegExp
if (obj instanceof RegExp) {
  return new RegExp(obj.source, obj.flags);
}

// Handle Array
if (Array.isArray(obj)) {
  const arrCopy = [];
  hash.set(obj, arrCopy);

  for (let i = 0; i < obj.length; i++) {
    arrCopy[i] = deepClone(obj[i], hash);
  }

  return arrCopy;
}

// Handle Map
if (obj instanceof Map) {
  const mapCopy = new Map();
  hash.set(obj, mapCopy);

  obj.forEach((value, key) => {
    mapCopy.set(key, deepClone(value, hash));
  });

  return mapCopy;
}

// Handle Set
if (obj instanceof Set) {
  const setCopy = new Set();
  hash.set(obj, setCopy);
```

```
    obj.forEach((value) => {
      setCopy.add(deepClone(value, hash));
    });

    return setCopy;
  }

  // Handle Object
  const objCopy = Object.create(Object.getPrototypeOf(obj));
  hash.set(obj, objCopy);

  // Copy all properties (including symbols)
  Reflect.ownKeys(obj).forEach((key) => {
    objCopy[key] = deepClone(obj[key], hash);
  });

  return objCopy;
}

// Test
const original = {
  name: 'John',
  age: 30,
  date: new Date(),
  regex: /test/gi,
  arr: [1, 2, { nested: true }],
  map: new Map([['key', 'value']]),
  set: new Set([1, 2, 3])
};

original.self = original;  // Circular reference

const cloned = deepClone(original);
console.log(cloned);
console.log(cloned.self === cloned);  // true
```

10.3 React Interview Questions

10.3.1 Question: Explain React reconciliation and the Fiber architecture.

Answer:

```
Reconciliation is the process React uses to determine what needs to be updated in the DOM.

Old Reconciliation (Stack):
- Recursive process
```

```
- Synchronous (blocks the main thread)
- Can't be interrupted
- Performance issues with large trees

Fiber Architecture (React 16+):
- Incremental reconciliation
- Can pause, resume, abort work
- Priority-based scheduling
- Time slicing

Fiber is a JavaScript object representing a unit of work:

{
  type: 'div',           // Component type
  key: null,             // Unique key
  props: { ... },        // Props
  stateNode: DOMNode,    // Actual DOM node
  return: parentFiber,   // Parent fiber
  child: childFiber,     // First child
  sibling: siblingFiber, // Next sibling
  alternate: oldFiber,   // Previous fiber (for diffing)
  effectTag: 'UPDATE',   // What needs to be done
  nextEffect: nextFiber  // Next fiber with effects
}

Reconciliation phases:

1. Render Phase (interruptible):
   - Build work-in-progress tree
   - Diff with current tree
   - Mark effects (PLACEMENT, UPDATE, DELETION)
   - Can be paused and resumed

2. Commit Phase (synchronous):
   - Apply effects to DOM
   - Cannot be interrupted
   - Calls lifecycle methods (componentDidMount, useLayoutEffect)

Priority levels:
- Immediate (sync): User input, animations
- User-blocking: Hover, scroll
- Normal: Network responses
- Low: Data fetching
- Idle: Analytics

This enables:
- Concurrent Mode
- Suspense
```

```
- Smooth animations
- Better perceived performance
```

10.3.2 Question: When would you use `useMemo` vs `useCallback`?

Answer:

```javascript
// useMemo: Memoize a computed value
function ExpensiveComponent({ items }) {
  // Without useMemo: recalculates on every render
  const total = items.reduce((sum, item) => sum + item.price, 0);

  // With useMemo: only recalculates when items change
  const total = useMemo(() => {
    console.log('Calculating total...');
    return items.reduce((sum, item) => sum + item.price, 0);
  }, [items]);

  return <div>Total: {total}</div>;
}

// useCallback: Memoize a function
function Parent() {
  const [count, setCount] = useState(0);

  // Without useCallback: new function on every render
  // Child re-renders even if count didn't change
  const handleClick = () => {
    console.log('Clicked!');
  };

  // With useCallback: same function reference
  // Child only re-renders when dependencies change
  const handleClick = useCallback(() => {
    console.log('Clicked!');
  }, []);

  return (
    <div>
      <div>{count}</div>
      <Child onClick={handleClick} />
    </div>
  );
}

const Child = React.memo(({ onClick }) => {
  console.log('Child rendered');
  return <button onClick={onClick}>Click me</button>;
```

```
});

// When to use:

// useMemo:
// - Expensive calculations
// - Avoid recreating objects/arrays (for dependency arrays)
// - Avoid re-rendering children

// useCallback:
// - Pass callbacks to optimized children (React.memo)
// - Callbacks in dependency arrays
// - Event handlers passed to many children

// Don't overuse!
// Premature optimization is bad
// Memoization has cost (memory, comparison)
// Only use when profiling shows benefit

// Example where useMemo helps with dependencies
function SearchResults({ query }) {
  // Without useMemo: new object on every render
  // useEffect runs on every render!
  const filters = { query, type: 'user' };

  useEffect(() => {
    fetchResults(filters);
  }, [filters]);  // Different object every time!

  // With useMemo: same object if query doesn't change
  const filters = useMemo(() => ({
    query,
    type: 'user'
  }), [query]);

  useEffect(() => {
    fetchResults(filters);
  }, [filters]);  // Only runs when query changes

  return <div>...</div>;
}
```

10.3.3 Question: Explain the differences between controlled and uncontrolled components.

Answer:

```jsx
// Controlled: React state is the single source of truth
function ControlledForm() {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Name:', name);
    console.log('Email:', email);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
      <input
        value={email}
        onChange={(e) => setEmail(e.target.value)}
      />
      <button type="submit">Submit</button>
    </form>
  );
}

// Uncontrolled: DOM is the source of truth
function UncontrolledForm() {
  const nameRef = useRef();
  const emailRef = useRef();

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Name:', nameRef.current.value);
    console.log('Email:', emailRef.current.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input ref={nameRef} defaultValue="" />
      <input ref={emailRef} defaultValue="" />
      <button type="submit">Submit</button>
    </form>
  );
}

// When to use:
```

```
// Controlled:
// - Form validation
// - Conditional rendering
// - Format input (e.g., phone numbers)
// - Dynamic forms
// - Most cases (recommended)

// Uncontrolled:
// - Simple forms
// - File inputs (always uncontrolled)
// - Integration with non-React code
// - Performance (avoid re-renders on every keystroke)

// Hybrid: Controlled with debouncing
function HybridForm() {
  const [value, setValue] = useState('');
  const [debouncedValue, setDebouncedValue] = useState('');

  useEffect(() => {
    const timer = setTimeout(() => {
      setDebouncedValue(value);
    }, 300);

    return () => clearTimeout(timer);
  }, [value]);

  // Validate debouncedValue
  useEffect(() => {
    if (debouncedValue) {
      validateInput(debouncedValue);
    }
  }, [debouncedValue]);

  return (
    <input
      value={value}
      onChange={(e) => setValue(e.target.value)}
    />
  );
}
```

10.4 CSS Interview Questions

10.4.1 Question: Explain CSS specificity with examples.

[See Topic 9 for detailed answer]

10.4.2 Question: How does the Critical Rendering Path work?

Answer:

```
Critical Rendering Path: Steps browser takes to render a page

1. DOM Construction:
   - Parse HTML
   - Build DOM tree
   - Incremental (can start rendering before complete)

2. CSSOM Construction:
   - Parse CSS
   - Build CSSOM tree
   - Blocks rendering (render-blocking)
   - Must be complete before rendering

3. Render Tree Construction:
   - Combine DOM + CSSOM
   - Only visible elements
   - Skip display: none elements

4. Layout (Reflow):
   - Calculate position and size
   - Box model calculations
   - Expensive operation

5. Paint:
   - Fill in pixels
   - Text, colors, images, borders, shadows
   - Expensive operation

6. Composite:
   - Combine layers
   - GPU-accelerated
   - Cheap operation

Optimization strategies:

1. Minimize Render-Blocking Resources:
   - Inline critical CSS
   - Defer non-critical CSS
   - Use media queries to mark non-blocking CSS

   <link rel="stylesheet" href="critical.css">
   <link rel="stylesheet" href="print.css" media="print">
   <link rel="stylesheet" href="mobile.css" media="(max-width: 600px)">
```

```
2. Reduce DOM Size:
   - Smaller DOM = faster parsing, layout, paint
   - Keep tree depth < 32 levels
   - Keep children per element < 60

3. Minimize Reflows:
   - Batch DOM changes
   - Use classes instead of individual styles
   - Avoid layout thrashing (read then write)

4. Use Transform/Opacity for Animations:
   - Only composite, no layout or paint
   - GPU-accelerated
   - Smooth 60fps animations

5. Use will-change:
   - Tell browser about upcoming changes
   - Creates new layer
   - Use sparingly (memory cost)

   .animated {
     will-change: transform, opacity;
   }

6. Lazy Load Images:
   - Don't block initial render
   - Load as needed

   <img src="image.jpg" loading="lazy">

7. Use Resource Hints:
   <link rel="dns-prefetch" href="//example.com">
   <link rel="preconnect" href="//example.com">
   <link rel="prefetch" href="next-page.html">
   <link rel="preload" href="font.woff2" as="font">
```

10.5 Performance Interview Questions

10.5.1 Question: How would you optimize a slow React application?

Answer:

```javascript
// 1. Identify the problem (use React DevTools Profiler)
// - Which components re-render?
// - How long do renders take?
// - What triggers renders?

// 2. Prevent unnecessary re-renders
```

```jsx
// Use React.memo for functional components
const ExpensiveComponent = React.memo(({ data }) => {
  return <div>{/* ... */}</div>;
});

// Use PureComponent for class components
class ExpensiveComponent extends React.PureComponent {
  render() {
    return <div>{/* ... */}</div>;
  }
}

// 3. Optimize expensive calculations

// Use useMemo
function Component({ items }) {
  const sortedItems = useMemo(() => {
    return items.sort((a, b) => a.value - b.value);
  }, [items]);

  return <List items={sortedItems} />;
}

// 4. Optimize callbacks

// Use useCallback
function Parent() {
  const handleClick = useCallback((id) => {
    // Handle click...
  }, []);

  return items.map(item => (
    <ChildComponent key={item.id} onClick={handleClick} />
  ));
}

// 5. Split code with lazy loading

const HeavyComponent = lazy(() => import('./HeavyComponent'));

function App() {
  return (
    <Suspense fallback={<Spinner />}>
      <HeavyComponent />
    </Suspense>
  );
}
```

```jsx
// 6. Virtualize long lists

import { FixedSizeList } from 'react-window';

function VirtualList({ items }) {
  return (
    <FixedSizeList
      height={600}
      itemCount={items.length}
      itemSize={50}
      width="100%"
    >
      {({ index, style }) => (
        <div style={style}>{items[index]}</div>
      )}
    </FixedSizeList>
  );
}

// 7. Use proper key props

// Bad: index as key (causes issues with reordering)
items.map((item, index) => <Item key={index} {...item} />)

// Good: stable unique ID
items.map(item => <Item key={item.id} {...item} />)

// 8. Batch state updates

// React 18: automatic batching
function Component() {
  const [count, setCount] = useState(0);
  const [flag, setFlag] = useState(false);

  const handleClick = () => {
    // Batched automatically in React 18
    setCount(c => c + 1);
    setFlag(f => !f);
  };
}

// React 17: manual batching
import { unstable_batchedUpdates } from 'react-dom';

const handleClick = () => {
  unstable_batchedUpdates(() => {
    setCount(c => c + 1);
```

```
    setFlag(f => !f);
  });
};

// 9. Debounce expensive operations

function SearchComponent() {
  const [query, setQuery] = useState('');

  const debouncedSearch = useMemo(
    () => debounce((q) => performSearch(q), 300),
    []
  );

  useEffect(() => {
    debouncedSearch(query);
  }, [query, debouncedSearch]);

  return <input value={query} onChange={(e) => setQuery(e.target.value)} />;
}

// 10. Use production build
// - npm run build
// - Minified, optimized
// - No dev warnings

// 11. Use Web Workers for heavy computations

// worker.js
self.addEventListener('message', (e) => {
  const result = heavyComputation(e.data);
  self.postMessage(result);
});

// Component
function Component() {
  useEffect(() => {
    const worker = new Worker('worker.js');

    worker.postMessage(data);

    worker.addEventListener('message', (e) => {
      setResult(e.data);
    });

    return () => worker.terminate();
  }, []);
```

```
}

// 12. Optimize images
// - Use WebP format
// - Lazy load
// - Responsive images
// - CDN with compression

// 13. Monitor with Performance API

useEffect(() => {
  performance.mark('component-mount-start');

  return () => {
    performance.mark('component-mount-end');
    performance.measure(
      'component-mount',
      'component-mount-start',
      'component-mount-end'
    );

    const measure = performance.getEntriesByName('component-mount')[0];
    console.log('Component mount time:', measure.duration);
  };
}, []);
```

This completes the comprehensive frontend development guide covering all 10 topics with in-depth technical details, code examples, and advanced concepts.