# Part 1: Setting Up JSX Without React

Exercise 1.1: Project Setup and Configuration

Objective: Configure a TypeScript project to use JSX without React

Instructions:

1. Create a new directory for your project
2. Initialize npm and install dependencies
3. Configure TypeScript for custom JSX

Step 1: Package Setup

```json
// package.json - Complete this configuration
{
  "name": "jsx-without-react",
  "version": "1.0.0",
  "scripts": {
    // TODO: Add scripts for dev, build, and type-check
  },
  "devDependencies": {
    // TODO: Add TypeScript, Vite, and other necessary dependencies
  }
}
```

Step 2: TypeScript Configuration

```json
// tsconfig.json - Fill in the missing JSX configuration
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "ESNext",
    "moduleResolution": "node",
    // TODO: Configure JSX compilation
    // Hint: You need jsx, jsxFactory, and jsxFragmentFactory options
    "allowJs": true,
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist"]
}
```

Exercise 1.2: Basic JSX Runtime Implementation

Objective: Build the core JSX-to-DOM conversion system

Instructions:

Create a file `src/jsx-runtime.ts` and implement the basic JSX functions.

Starter Code:

```typescript
// src/jsx-runtime.ts

// TODO: Define the VNode interface
// A VNode should represent a virtual DOM node with:
// - type: string or Function
// - props: object with properties
// - children: array of VNodes, strings, or numbers
interface VNode {
  // Your interface definition here
}

// TODO: Define ComponentProps interface
// This should allow components to receive children and other props
interface ComponentProps {
  // Your interface definition here
}

// TODO: Define ComponentFunction type
// This should be a function that takes props and returns a VNode
type ComponentFunction = // Your type definition here

// TODO: Implement createElement function
// This is the heart of JSX - it converts JSX syntax to VNode objects
export function createElement(
  type: string | ComponentFunction,
  props: Record<string, any> | null,
  ...children: (VNode | string | number)[]
): VNode {
  // STEP 1: Handle props (use empty object if null)

  // STEP 2: Flatten and filter children (remove null/undefined)

  // STEP 3: Return VNode object

  // Your implementation here
}

// TODO: Implement createFragment function
// Fragments allow grouping elements without extra wrapper
export function createFragment(
  props: Record<string, any> | null,
  ...children: (VNode | string | number)[]
): VNode {
  // Hint: Use createElement with 'fragment' as type
  // Your implementation here
}
```

Your Task:

Complete the missing implementations. Here are some hints:

1. **VNode Interface**: Think about what information you need to represent a DOM element
2. **createElement**: This function is called for every JSX element like `<div>Hello</div>`
3. **Fragment**: Special case for grouping elements without a wrapper

Test Your Implementation:

```typescript
// Test in browser console or create a test file
const vnode = createElement('div', { className: 'test' }, 'Hello World');
console.log(vnode);
// Should output: { type: 'div', props: { className: 'test' }, children: ['Hello World'] }
```

Exercise 1.3: DOM Rendering System

Objective: Convert VNodes to actual DOM elements

Instructions:

Add these functions to your `jsx-runtime.ts` file:

Starter Code:

```typescript
// TODO: Implement renderToDOM function
// This converts VNode objects to actual DOM elements
export function renderToDOM(vnode: VNode | string | number): Node {
  // STEP 1: Handle text nodes (strings and numbers)
  if (/* condition for text nodes */) {
    // Return document.createTextNode()
  }

  // STEP 2: Handle fragments
  if (/* condition for fragments */) {
    // Create DocumentFragment and append children
  }

  // STEP 3: Handle component functions
  if (/* condition for function components */) {
    // Call the function with props and children, then render result
  }

  // STEP 4: Handle regular HTML elements
  // Create element, set attributes, append children

  // Your implementation here
}

// TODO: Implement mount function
// This attaches a VNode to a real DOM container
export function mount(vnode: VNode, container: HTMLElement): void {
  // Convert VNode to DOM and append to container
  // Your implementation here
}

// TODO: Implement basic state management
// Simple useState hook for components
export function useState<T>(initialValue: T): [() => T, (newValue: T) => void] {
  // STEP 1: Store the current value
  // STEP 2: Create getter function
  // STEP 3: Create setter function that updates value
  // STEP 4: Handle re-rendering (for now, just update the value)

  // Your implementation here
}
```

Implementation Guidelines:

1. **Text Nodes**: Use `document.createTextNode(String(vnode))`
2. **Elements**: Use `document.createElement(vnode.type as string)`
3. **Attributes**: Handle special cases like `className`, `style`, event handlers (`onClick`)
4. **Events**: Convert `onClick` to `addEventListener('click', handler)`

Challenge Questions:
- How will you handle the `style` prop when it's an object?
- How will you differentiate between HTML attributes and DOM properties?
- What happens with boolean attributes like `disabled`?

## Part 2: Building Your First JSX Application

Exercise 2.1: Simple Counter Component

Objective: Create your first functional component with state

Instructions:

Create a file `src/counter.tsx` and build a counter component.

Starter Template:

```typescript
// src/counter.tsx
/** @jsx createElement */
import { createElement, useState } from './jsx-runtime';

// TODO: Define ButtonProps interface
interface ButtonProps {
  // What props should a button accept?
}

// TODO: Create a Button component
const Button = (/* props parameter */) => {
  // Return JSX for a button element
  // Handle onClick, children, className props

  // Your implementation here
};

// TODO: Define CounterProps interface
interface CounterProps {
  // What props should the counter accept? (hint: initialCount?)
}

// TODO: Create Counter component
const Counter = (/* props parameter */) => {
  // STEP 1: Use useState for count value
  // STEP 2: Create increment, decrement, reset functions

  // STEP 3: Return JSX structure with:
  // - Display current count
  // - Three buttons (increment, decrement, reset)

  // Your implementation here
};

// TODO: Export Counter component
export { Counter };
```

Your Tasks:

1. Complete the Button component with proper TypeScript types
2. Implement the Counter component with state management
3. Handle button clicks to update the counter
4. Style the components with className props

Expected JSX Structure:

```jsx
// Your Counter component should render something like:
<div className="counter">
  <h2>Count: {currentCount}</h2>
  <div className="buttons">
    <Button onClick={increment}>+</Button>
    <Button onClick={decrement}>-</Button>
    <Button onClick={reset}>Reset</Button>
  </div>
</div>
```

Exercise 2.2: Todo List Application

Objective: Build a more complex application with multiple components

Instructions:

Create `src/todo-app.tsx` with a complete todo list application.

Component Structure to Build:

```typescript
// src/todo-app.tsx
/** @jsx createElement */
import { createElement, useState } from './jsx-runtime';

// TODO: Define TypeScript interfaces
interface Todo {
  // What properties should a todo item have?
  // Hint: id, text, completed, createdAt?
}

interface TodoItemProps {
  // What props does a todo item component need?
}

interface TodoListProps {
  // What props does the todo list need?
}

// TODO: Implement TodoItem component
const TodoItem = (/* props */) => {
  // STEP 1: Destructure props
  // STEP 2: Return JSX with:
  // - Checkbox for completion status
  // - Text display (with strikethrough if completed)
  // - Delete button

  // Your implementation here
};

// TODO: Implement AddTodoForm component
const AddTodoForm = (/* props */) => {
  // STEP 1: Use useState for input value
  // STEP 2: Handle form submission
  // STEP 3: Return JSX with input and submit button

  // Your implementation here
};

// TODO: Implement main TodoApp component
const TodoApp = () => {
  // STEP 1: State for todos array
  // STEP 2: Functions to add, toggle, delete todos
  // STEP 3: Return JSX structure with:
  // - Header
  // - Add todo form
  // - Todo list
  // - Summary (total/completed counts)

  // Your implementation here
};

export { TodoApp };
```

Required Functionality:

1. ✅ Add new todos
2. ✅ Mark todos as complete/incomplete
3. ✅ Delete todos
4. ✅ Display total and completed counts
5. ✅ Filter by status (optional challenge)

Implementation Hints:

- Use `Array.map()` to render todo items
- Generate unique IDs for todos (use `Date.now()` or counter)
- Handle form submission with `preventDefault()`
- Use conditional CSS classes for completed items

## Part 3: Advanced Features

Exercise 3.1: Enhanced JSX Runtime

Objective: Add advanced features to your JSX system

Instructions:

Extend your `jsx-runtime.ts` with these features:

Feature 1: Refs Support

```typescript
// TODO: Add ref support to your renderToDOM function
// Refs allow direct access to DOM elements

// In your renderToDOM function, add this check:
if (key === 'ref' && typeof value === 'function') {
  // Call the ref function with the element
  // Your implementation here
}
```

Feature 2: CSS-in-JS Support

```typescript
// TODO: Enhanced style handling
// Support both string and object styles

// Example usage:
const styles = {
  backgroundColor: 'blue',
  fontSize: '16px',
  marginTop: '10px'
};

<div style={styles}>Styled content</div>
```

Your Task:

Modify the style handling in `renderToDOM` to:

1. Handle string styles: `style="color: red;"`
2. Handle object styles: `style={{color: 'red', fontSize: '14px'}}`
3. Convert camelCase to kebab-case (backgroundColor → background-color)

Feature 3: Event Delegation

```typescript
// TODO: Implement better event handling
// Currently: element.addEventListener for each element
// Better: Use event delegation on document

// Challenge: Research event delegation and implement it
// Hint: Use data attributes to identify elements
```

Exercise 3.2: Component Library

Objective: Build reusable, typed components

Instructions:

Create `src/components.tsx` with a component library:

Template:

```typescript
// src/components.tsx
/** @jsx createElement */
import { createElement } from './jsx-runtime';

// TODO: Create a Card component
interface CardProps {
  // Define props: title?, children, className?, onClick?
}

const Card = (/* props */) => {
  // Return a card-like structure with title and content
  // Your implementation here
};
```

```
// TODO: Create a Modal component
interface ModalProps {
  // Define props: isOpen, onClose, title?, children
}

const Modal = (/* props */) => {
  // STEP 1: Return null if not open
  // STEP 2: Create overlay and modal content
  // STEP 3: Handle click outside to close

  // Your implementation here
};

// TODO: Create a Form component
interface FormProps {
  // Define props: onSubmit, children, className?
}

const Form = (/* props */) => {
  // Handle form submission and prevent default
  // Your implementation here
};

// TODO: Create an Input component
interface InputProps {
  // Define props: type?, value, onChange, placeholder?, className?
}

const Input = (/* props */) => {
  // Create a styled input with proper event handling
  // Your implementation here
};

export { Card, Modal, Form, Input };
```

Requirements:

1. All components must have proper TypeScript interfaces
2. Handle common props like `className`, `children`
3. Implement proper event handling
4. Make components reusable and composable

## Part 4: Real-World Application

Exercise 4.1: Dashboard Application

Objective: Build a complete dashboard using all learned concepts

Instructions:

Create a dashboard that displays data with charts and interactivity.

Project Structure:

```
src/
 ├──── jsx-runtime.ts        (your JSX implementation)
 ├──── components.tsx        (reusable components)
 ├──── dashboard.tsx         (main dashboard)
 ├──── chart.tsx             (chart component using Canvas)
 ├──── data-service.ts       (mock data service)
 └──── main.tsx              (app entry point)
```

Your Tasks:

1. Data Service (`src/data-service.ts`**)**

```typescript
// TODO: Create a mock data service
interface DataPoint {
  // Define structure for chart data
}

export class DataService {
```

  // TODO: Methods to:
  // - Generate mock data
  // - Simulate real-time updates
  // - Filter data by category/date
}

2. Chart Component (`src/chart.tsx`**)**

```typescript
// TODO: Build a chart component using HTML5 Canvas
// Requirements:
// - Support bar, line, and pie charts
// - Accept data and chart type as props
// - Implement basic drawing functions
// - Add interactive features (hover, click)

interface ChartProps {
  // Define props for chart component
}

// Implement drawing functions:
// - drawBarChart()
// - drawLineChart()
// - drawPieChart()
```

3. Dashboard Layout (`src/dashboard.tsx`**)**

```typescript
// TODO: Create main dashboard component
// Features to implement:
// - Header with title and controls
// - Chart type selector
// - Data filtering options
// - Real-time data updates
// - Responsive grid layout

// Use your component library for UI elements
```

Required Features:
- 📊 Multiple chart types (bar, line, pie)
- 🔄 Real-time data simulation
- 🎛️ Interactive controls
- 📱 Responsive design
- 🎨 Professional styling

## Part 5: Testing and Deployment

Exercise 5.1: Build System Setup

Objective: Configure build tools for production

Instructions:

1. Vite Configuration

Create `vite.config.ts`:
```typescript
// TODO: Configure Vite for your JSX setup
// Hints:
// - Set jsxFactory to your createElement function
// - Configure build options
// - Set up dev server
```

2. HTML Template

Create `index.html`:
```html
<!-- TODO: Create HTML template -->
<!-- Requirements: -->
<!-- - Basic CSS for styling -->
<!-- - Root div for mounting -->
<!-- - Script tag for your app -->
```

3. Main Entry Point

Create `src/main.tsx`:

```typescript
// TODO: Create main app file that:
// - Imports your components
// - Mounts the app to DOM
// - Handles routing if needed
```

Exercise 5.2: Performance Optimization

Objective: Optimize your JSX implementation

Your Tasks:

1. Benchmarking

```typescript
// TODO: Create performance tests
// - Measure createElement speed
// - Test rendering performance
// - Compare with React (optional)
```

2. Optimization Techniques

```typescript
// TODO: Implement optimizations:
// - Object pooling for VNodes
// - Batch DOM updates
// - Event delegation
// - Memory leak prevention
```

3. Bundle Size Analysis

```bash
# TODO: Analyze and optimize bundle size
```

Commands to research and implement

```
```