**Plan de Clases: Semana 1 - Fundamentos de React.js**

**Bootcamp Intensivo de React.js**  
**Academia**: AcademyCoder.com  
**Curso Personalizado para**: Javier y Naomy  
**Semana**: 1 (11 al 15 de agosto de 2025)  
**Horario**: Lunes a Viernes, 8:00 AM - 10:00 AM (90 min teoría + 30 min práctica)  
**Objetivo General**: Establecer una base sólida en React.js, cubriendo configuración del entorno, JSX, componentes funcionales y de clase, props, estado, eventos y un proyecto integrador (Todo List) para garantizar un aprendizaje profundo y práctico.  
**Metodología**: 90 minutos de presentación teórica (diapositivas detalladas con explicaciones exhaustivas, ejemplos de código, casos reales y demostraciones en vivo) + 30 minutos de práctica guiada (pair programming).  
**Herramientas**: Visual Studio Code, Node.js (LTS), Create React App, Git, GitHub, Google Chrome con React Developer Tools.  
**Enfoque Pedagógico**: Explicaciones profundas, orientadas a entrevistas técnicas y aplicaciones profesionales, con ejemplos prácticos del mundo real, analogías claras y resolución de errores comunes para preparar a los estudiantes para el mercado laboral.

**Índice**

1. Introducción General
2. Cronograma de la Semana
3. Material para la Presentación  
   3.1 Día 1: Introducción a React y Setup  
   3.2 Día 2: JSX y Componentes Funcionales  
   3.3 Día 3: Componentes de Clase y Props  
   3.4 Día 4: State y Manejo de Eventos  
   3.5 Día 5: Proyecto 1 - App de Todo List Simple
4. Recursos Adicionales

**Introducción General**

La **Semana 1** del **Bootcamp Intensivo de React.js** está diseñada para introducir a Javier y Naomy en los fundamentos de **React.js**, una de las bibliotecas de JavaScript más demandadas para construir interfaces de usuario modernas, eficientes y escalables. Este plan de clases está estructurado para proporcionar una comprensión profunda y práctica, con explicaciones exhaustivas que preparan a los estudiantes para entrevistas técnicas y proyectos profesionales. Cada día incluye:

* **Teoría (90 min)**: Diapositivas con contenido detallado, ejemplos de código comentados, casos reales (ej. aplicaciones como Netflix, Airbnb), analogías para clarificar conceptos, y explicaciones orientadas a entrevistas técnicas.
* **Práctica (30 min)**: Ejercicios guiados de pair programming para reforzar el aprendizaje, con soluciones detalladas y resolución de errores comunes.
* **Enfoque Profesional**: Cada tema se explica con un nivel de detalle que refleja la experiencia de un profesor con más de 10 años en desarrollo, abordando por qué los conceptos son críticos en la industria, cómo se aplican en proyectos reales, y cómo responder preguntas en entrevistas técnicas.

El curso usa **Create React App (CRA)** por su configuración robusta con Webpack, ideal para principiantes y entornos profesionales. Se compara con **Vite** para contextualizar decisiones técnicas, pero se prioriza CRA por su soporte completo y compatibilidad con el ecosistema React. Al final de la semana, los estudiantes construirán una **app de Todo List**, integrando todos los conceptos en un proyecto práctico que sirve como base para su portafolio.

**Objetivos de la Semana**

* Configurar un entorno profesional con **Create React App** y entender su estructura.
* Dominar **JSX** como la sintaxis declarativa de React, con un enfoque en su compilación y uso práctico.
* Comparar componentes funcionales y de clase, entendiendo su evolución y casos de uso.
* Implementar **props** para pasar datos, **estado** para datos dinámicos, y **eventos** para interactividad.
* Construir una aplicación funcional (Todo List) que combine todos los conceptos, lista para un portafolio profesional.

**Por qué Create React App vs. Vite?**

* **Create React App (CRA)**:
  + Herramienta oficial de React, usa **Webpack** para bundling, **Babel** para transpilar JSX/ES6+, y **ESLint** para linting.
  + Incluye soporte nativo para pruebas (Jest), CSS Modules, y configuraciones predefinidas.
  + Ideal para principiantes por su simplicidad y para proyectos empresariales por su robustez.
  + **Ventajas**: Configuración completa, documentación extensa, soporte para legacy code.
  + **Desventajas**: Builds más lentos, configuración menos flexible.
  + **Caso Real**: Usado en empresas como Meta y Shopify para proyectos que requieren estabilidad y herramientas integradas.
* **Vite (create vite@latest)**:
  + Usa **esbuild** para builds ultrarrápidos y soporte nativo para ES Modules.
  + Configuración ligera, ideal para proyectos modernos y pequeños.
  + **Ventajas**: Desarrollo más rápido, menor overhead.
  + **Desventajas**: Menos soporte para configuraciones complejas, comunidad más pequeña.
  + **Caso Real**: Usado en startups para prototipos rápidos o apps ligeras.
* **Decisión para el Curso**: **CRA** es la elección porque su configuración completa reduce la complejidad para principiantes, garantiza compatibilidad con herramientas empresariales (Redux, Jest), y prepara a los estudiantes para entornos donde Webpack es estándar. Vite se explorará en módulos avanzados.

**Cronograma de la Semana**

| **Día** | **Tema** | **Teoría (90 min)** | **Práctica (30 min)** |
| --- | --- | --- | --- |
| Lunes | Introducción a React y Setup | Qué es React, Virtual DOM, CRA vs. Vite, setup | Configurar entorno, app "Hello World" |
| Martes | JSX y Componentes Funcionales | Sintaxis JSX, componentes funcionales, casos reales | Componente <Greeting> personalizado |
| Miércoles | Componentes de Clase y Props | Clases vs. funcionales, props, flujo unidireccional | Familia de componentes con props |
| Jueves | State y Manejo de Eventos | useState, eventos (onClick), interactividad | Botón que actualiza estado |
| Viernes | Proyecto 1: Todo List Simple | Integrar conceptos, app funcional, revisión | App de Todo List (agregar/eliminar) |

**Material para la Presentación**

Cada día incluye diapositivas detalladas en formato Markdown, diseñadas para una presentación de **90 minutos** que combine teoría, ejemplos de código comentados, casos reales, y analogías. Las explicaciones para el profesor son extensas, reflejando la experiencia de un desarrollador con más de 10 años, y están orientadas a preparar a los estudiantes para entrevistas técnicas y proyectos profesionales. La práctica de **30 minutos** incluye ejercicios guiados con soluciones detalladas y resolución de errores comunes.

**Día 1: Introducción a React y Setup**

**Objetivo**: Comprender qué es React, su importancia en la industria, el rol del Virtual DOM, y configurar un entorno profesional con Create React App.  
**Duración**: 90 min teoría + 30 min práctica.  
**Recursos**: <https://react.dev/learn/installation>, <https://create-react-app.dev/>, <https://nodejs.org/>.

**Diapositivas**

**Diapositiva 1: Título - Bienvenidos al Bootcamp de React.js**

* **Contenido**:
  + **Bienvenidos, Javier y Naomy, al Bootcamp Intensivo de React.js!**
  + Tema de hoy: **Introducción a React y Configuración del Entorno**.
  + Objetivos:
    - Entender qué es React y por qué es líder en desarrollo front-end.
    - Configurar un entorno profesional con Create React App (CRA).
    - Crear y ejecutar una app "Hello World".
  + **Pregunta para Entrevistas**: ¿Qué es React y cómo difiere de un framework como Angular?
* **Explicación para el Profesor (5 min)**:
  + Saludar a los estudiantes con entusiasmo, destacando que este curso los convertirá en desarrolladores React listos para el mercado laboral.
  + Explicar que hoy se sentarán las bases técnicas y conceptuales, cruciales para el éxito del bootcamp.
  + Resaltar que CRA es la herramienta elegida por su robustez, y que se comparará con Vite para contextualizar decisiones técnicas.
  + Mencionar la pregunta de entrevista para motivar un aprendizaje profundo, útil para discusiones técnicas en el futuro.

**Diapositiva 2: ¿Qué es React.js?**

* **Contenido**:
  + **Definición**: React.js es una biblioteca de JavaScript de código abierto, creada por Meta (2013), diseñada para construir interfaces de usuario interactivas, modulares y escalables, especialmente para aplicaciones de una sola página (SPAs).
  + **Características Clave**:
    - **Componentes Reutilizables**: Divide la UI en bloques independientes (ej. un botón, una tarjeta de producto).
    - **Virtual DOM**: Representación en memoria del DOM para actualizaciones rápidas y eficientes.
    - **Flujo de Datos Unidireccional**: Datos fluyen de padres a hijos, simplificando debugging.
    - **Ecosistema Robusto**: Compatible con herramientas como Redux, React Router, Next.js.
  + **Casos Reales**:
    - **Facebook**: News feed dinámico con actualizaciones en tiempo real.
    - **Netflix**: Carrusel de películas, optimizado para rendimiento.
    - **Airbnb**: Formularios interactivos y mapas dinámicos.
  + **Ventajas en la Industria**:
    - Alta demanda laboral (70% de trabajos front-end requieren React).
    - Comunidad activa, extensa documentación, soporte para proyectos legacy y modernos.
  + **Pregunta para Entrevistas**: ¿Por qué usar React en lugar de manipular el DOM directamente con JavaScript?
* **Explicación para el Profesor (15 min)**:
  + Explicar que React no es un framework completo (como Angular, que incluye routing, estado, etc.), sino una biblioteca enfocada en la UI, lo que la hace flexible pero requiere herramientas adicionales.
  + Usar una analogía: React es como un chef especializado en platos principales (UI), mientras que un framework es un restaurante completo (todo incluido).
  + Detallar cómo empresas como Facebook usan React para interfaces dinámicas (ej. el feed que actualiza posts sin recargar la página).
  + Resaltar que en entrevistas, los candidatos deben mencionar la modularidad, el Virtual DOM, y el ecosistema como ventajas clave de React.
  + Mostrar una captura de pantalla de la UI de Netflix (carrusel) para ilustrar un componente React en acción.

**Diapositiva 3: El Virtual DOM Explicado**

* **Contenido**:
  + **Problema del DOM Real**: Manipular el DOM directamente (con document.getElementById) es lento, especialmente en apps con muchos elementos dinámicos (ej. lista de 1000 productos).
  + **Solución del Virtual DOM**:
    - React crea una copia ligera del DOM en memoria (Virtual DOM).
    - Compara cambios (diffing) entre el Virtual DOM y el DOM real.
    - Actualiza solo los nodos necesarios (reconciliation).
  + **Ejemplo Visual**:
  + // Estado inicial
  + <div><p>Hola</p></div>
  + // Estado nuevo
  + <div><p>Adiós</p></div>
  + // React solo actualiza el texto "Adiós", no el <div> entero.
  + **Beneficio**: Mejora el rendimiento, reduce operaciones costosas.
  + **Caso Real**: En Instagram, el Virtual DOM permite actualizar likes/comentarios sin recargar la página.
  + **Pregunta para Entrevistas**: ¿Cómo funciona el algoritmo de reconciliación en React?
* **Explicación para el Profesor (15 min)**:
  + Usar una analogía: El Virtual DOM es como un plano de una casa; en lugar de reconstruir toda la casa por un cambio (DOM real), solo se ajustan las piezas necesarias.
  + Explicar el proceso de diffing: React compara árboles de elementos y calcula el mínimo de cambios.
  + Demostración: Abrir Chrome DevTools, modificar un elemento del DOM manualmente (lento), y contrastar con React (rápido).
  + Resaltar que en entrevistas, se espera que los candidatos expliquen el Virtual DOM como un mecanismo de optimización, mencionando diffing y reconciliation.
  + Mencionar que el Virtual DOM es crítico para apps con actualizaciones frecuentes (ej. chats, feeds).

**Diapositiva 4: Create React App vs. Vite**

* **Contenido**:
  + **Create React App (CRA)**:
    - Herramienta oficial de React, usa **Webpack** para bundling.
    - Incluye **Babel** (transpila JSX/ES6+), **ESLint** (linting), **Jest** (pruebas), y soporte para CSS Modules.
    - Estructura preconfigurada, ideal para principiantes y proyectos empresariales.
    - **Ventajas**: Configuración completa, soporte para legacy, comunidad grande.
    - **Desventajas**: Builds más lentos, menos flexible para configuraciones avanzadas.
    - **Caso Real**: Usado en Shopify para apps e-commerce con múltiples dependencias.
  + **Vite (create vite@latest)**:
    - Usa **esbuild** para builds ultrarrápidos, soporta ES Modules nativos.
    - Configuración ligera, ideal para prototipos o apps modernas.
    - **Ventajas**: Desarrollo rápido, menor overhead.
    - **Desventajas**: Menos soporte para configuraciones complejas, comunidad más pequeña.
    - **Caso Real**: Usado en startups para MVPs rápidos (ej. app de notas).
  + **Nuestra Elección**: CRA por su robustez, compatibilidad con herramientas empresariales, y preparación para entornos profesionales.
  + **Pregunta para Entrevistas**: ¿Cuándo elegirías CRA sobre Vite, y viceversa?
* **Explicación para el Profesor (15 min)**:
  + Comparar CRA y Vite con una analogía: CRA es un camión pesado (lleva todo, pero más lento); Vite es un auto deportivo (rápido, pero menos carga).
  + Explicar que Webpack en CRA maneja tareas complejas (bundling, minificación, hot reloading), mientras que esbuild en Vite optimiza velocidad.
  + Resaltar que CRA es ideal para este curso porque reduce la complejidad de configuración y prepara a los estudiantes para proyectos con Redux, Jest, o TypeScript.
  + Mencionar que en entrevistas, los candidatos deben justificar la elección de herramientas según el contexto (ej. CRA para estabilidad, Vite para rapidez).
  + Demostración: Mostrar la diferencia en tiempo de npm start entre CRA y Vite (crear un proyecto pequeño con ambos).

**Diapositiva 5: Configuración del Entorno con CRA**

* **Contenido**:
  + **Requisitos Previos**:
    - Node.js (LTS, verificar con node -v, npm -v).
    - Visual Studio Code con extensiones: ESLint, Prettier, React Developer Tools.
    - Google Chrome con React Developer Tools (Chrome Web Store).
  + **Pasos para Crear un Proyecto**:
  + # Verificar Node.js
  + node -v # Debe mostrar v16 o superior
  + npm -v # Debe mostrar v8 o superior
  + # Crear proyecto
  + npx create-react-app my-first-react-app
  + cd my-first-react-app
  + npm start # Inicia en http://localhost:3000
  + **Resultado**: App con "Welcome to React" en el navegador.
  + **Caso Real**: Equipos en Amazon usan CRA para configurar rápidamente proyectos front-end con dependencias estándar.
  + **Pregunta para Entrevistas**: ¿Qué hace npx create-react-app bajo el capó?
* **Explicación para el Profesor (15 min)**:
  + Explicar cada comando: npx evita instalaciones globales, create-react-app genera una estructura estandarizada.
  + Detallar que CRA configura Webpack, Babel, y ESLint automáticamente, ahorrando tiempo.
  + Demostración en vivo: Ejecutar los comandos, abrir la app en el navegador, y mostrar React Developer Tools para inspeccionar el componente App.
  + Resaltar que en entrevistas, se espera que los candidatos mencionen que CRA abstrae Webpack y Babel, configurando un entorno listo para producción.
  + Mencionar errores comunes: Node.js no instalado, puerto 3000 ocupado (solucionar con PORT=3001 npm start).

**Diapositiva 6: Estructura de un Proyecto CRA**

* **Contenido**:
  + **Carpetas y Archivos Clave**:
    - src/: Código fuente (App.js, index.js).
    - public/: Archivos estáticos (index.html, favicon).
    - package.json: Dependencias (react, react-dom) y scripts (start, build).
    - node\_modules/: Librerías instaladas.
  + **Código Inicial (src/App.js)**:
  + import React from 'react';
  + import './App.css';
  + function App() {
  + return (
  + <div className="App">
  + <header className="App-header">
  + <h1>Welcome to React!</h1>
  + </header>
  + </div>
  + );
  + }
  + export default App;
  + **Cómo Funciona**:
    - index.js renderiza App en <div id="root"> (public/index.html).
    - App.css estiliza el componente.
  + **Caso Real**: Estructura similar usada en Twitter para componentes de UI modulares.
  + **Pregunta para Entrevistas**: ¿Qué papel juega index.js en un proyecto React?
* **Explicación para el Profesor (15 min)**:
  + Explicar la estructura: src contiene la lógica, public los assets estáticos.
  + Detallar que index.js es el punto de entrada, conectando React con el DOM real.
  + Demostración: Modificar App.js para cambiar el texto a "Hola, Bootcamp!" y mostrar el hot reloading.
  + Resaltar que className reemplaza class por conflictos con JavaScript; en entrevistas, esto es una pregunta común.
  + Mostrar cómo React Developer Tools inspecciona la jerarquía de componentes.

**Diapositiva 7: Práctica Guiada**

* **Contenido**:
  + **Ejercicio**: Configurar el entorno y crear una app "Hello World" con CRA.
  + **Pasos**:
    1. Instalar Node.js y verificar versiones.
    2. Ejecutar npx create-react-app my-first-react-app.
    3. Iniciar la app (npm start).
    4. Modificar src/App.js para mostrar "Hola, Javier y Naomy!".
  + **Solución Esperada**:
  + // src/App.js
  + import React from 'react';
  + import './App.css';
  + function App() {
  + return (
  + <div className="App">
  + <header className="App-header">
  + <h1>Hola, Javier y Naomy!</h1>
  + </header>
  + </div>
  + );
  + }
  + export default App;
  + **Errores Comunes**:
    1. Node.js no instalado (descargar desde <https://nodejs.org/>).
    2. Puerto ocupado (usar PORT=3001 npm start).
    3. Estructura de carpetas modificada accidentalmente.
  + **Pregunta para Entrevistas**: ¿Cómo resuelves un error de puerto ocupado en CRA?
* **Explicación para el Profesor (10 min)**:
  + Explicar la práctica: Los estudiantes configurarán su entorno y personalizarán App.js.
  + Guiar la instalación de Node.js si es necesario, supervisando cada comando.
  + Demostrar cómo abrir la app en el navegador y verificar cambios en tiempo real.
  + Resaltar que en entrevistas, los candidatos deben saber solucionar errores de setup (ej. puerto ocupado, dependencias corruptas).

**Diapositiva 8: Cierre y Tareas**

* **Contenido**:
  + **Resumen**:
    - React es una biblioteca para UI, optimizada por el Virtual DOM.
    - CRA simplifica el setup con Webpack, Babel, y ESLint.
    - La app "Hello World" es el primer paso hacia proyectos complejos.
  + **Tareas (1-2 horas)**:
    - Leer <https://react.dev/learn> (sección "Quick Start").
    - Explorar la estructura de carpetas de CRA (src, public).
    - Modificar App.js con un mensaje creativo (ej. agregar una imagen).
  + **Pregunta Reflexiva**: ¿Cómo mejora el Virtual DOM la experiencia del usuario en apps como Instagram?
  + **Pregunta para Entrevistas**: ¿Qué ventajas ofrece CRA para un equipo grande?
* **Explicación para el Profesor (5 min)**:
  + Resumir los puntos clave: React, Virtual DOM, CRA como base sólida.
  + Motivar a los estudiantes a leer la documentación oficial y experimentar con pequeños cambios en App.js.
  + Fomentar preguntas para aclarar dudas antes de la práctica.

**Práctica (30 min)**

* **Actividad**: Configurar el entorno y modificar App.js para mostrar un mensaje personalizado.
* **Instrucciones para el Profesor**:
  + Pair programming: Javier escribe los comandos, Naomy verifica la salida en el navegador.
  + Supervisar la instalación de Node.js y la ejecución de npx create-react-app.
  + Comprobar que ambos modifiquen App.js y vean los cambios en tiempo real.
  + Resolver errores: Node.js no encontrado, puerto ocupado, o problemas con VS Code.
* **Resultado Esperado**: App funcional en [http://localhost:3000](http://localhost:3000/) con mensaje personalizado.

**Día 2: JSX y Componentes Funcionales**

**Objetivo**: Dominar JSX como sintaxis declarativa, entender su compilación, y crear componentes funcionales reutilizables.  
**Duración**: 90 min teoría + 30 min práctica.  
**Recursos**: <https://react.dev/learn/writing-markup-with-jsx>, <https://react.dev/learn/your-first-component>.

**Diapositivas**

**Diapositiva 1: Título - JSX y Componentes Funcionales**

* **Contenido**:
  + **Tema de hoy**: JSX y Componentes Funcionales.
  + Objetivos:
    - Comprender JSX como la sintaxis declarativa de React.
    - Crear y reutilizar componentes funcionales.
    - Entender cómo JSX se compila y su rol en la industria.
  + **Importancia**: JSX es la base para escribir interfaces en React; los componentes funcionales son el estándar moderno.
  + **Pregunta para Entrevistas**: ¿Qué es JSX y cómo se diferencia del HTML tradicional?
* **Explicación para el Profesor (5 min)**:
  + Explicar que JSX es el núcleo de la experiencia de desarrollo en React, combinando la simplicidad de HTML con la potencia de JavaScript.
  + Resaltar que los componentes funcionales han reemplazado a las clases en proyectos modernos, gracias a los hooks (introducidos en React 16.8).
  + Mencionar que entender JSX es crucial para entrevistas, ya que los reclutadores suelen preguntar por su compilación y diferencias con HTML.

**Diapositiva 2: ¿Qué es JSX?**

* **Contenido**:
  + **Definición**: JSX (JavaScript XML) es una extensión de sintaxis que permite escribir código similar a HTML dentro de JavaScript, combinando la lógica de la UI con la programación.
  + **Cómo Funciona**:
    - JSX no es HTML, sino azúcar sintáctico que se compila a llamadas de React.createElement().
    - Ejemplo:
    - // JSX
    - const element = <h1>Hola, {name}!</h1>;
    - // Compilado por Babel
    - const element = React.createElement("h1", null, "Hola, ", name);
  + **Reglas Clave**:
    - **Elemento Raíz Único**: Todo JSX debe estar envuelto en un solo elemento (ej. <div> o <React.Fragment>).
    - **Atributos en camelCase**: className en lugar de class, onClick en lugar de onclick.
    - **Expresiones JS**: Usa {} para inyectar variables, funciones, o cálculos (ej. {2 + 2}).
    - **Auto-cierre de Etiquetas**: <img /> en lugar de <img>.
  + **Ventajas**:
    - Código más legible y declarativo que JavaScript puro.
    - Reduce errores al combinar UI y lógica.
    - Soporta validación de tipos con herramientas como TypeScript.
  + **Caso Real**: En Netflix, JSX se usa para renderizar carruseles de películas, con {movie.title} para datos dinámicos.
  + **Pregunta para Entrevistas**: ¿Por qué JSX requiere un elemento raíz único y cómo se soluciona con Fragments?
* **Explicación para el Profesor (20 min)**:
  + Explicar que JSX parece HTML pero es JavaScript, compilado por Babel para navegadores.
  + Usar una analogía: JSX es como un plano de construcción (UI) que el contratista (Babel) traduce en estructuras reales (React.createElement).
  + Detallar cada regla:
    - **Elemento raíz**: React necesita un solo nodo para el Virtual DOM; <React.Fragment> o <>...</> evitan <div> innecesarios.
    - **camelCase**: Evita conflictos con palabras reservadas de JS (class es una palabra clave).
    - **Expresiones**: {} permite lógica dinámica, pero no bloques como if (usar ternarios).
  + Demostración: Crear un componente JSX simple en CodeSandbox y mostrar su compilación en <https://babeljs.io/repl>.
  + Resaltar que en entrevistas, los candidatos deben explicar que JSX no se ejecuta directamente, sino que se transforma, y mencionar Fragments como solución moderna.
  + Ejemplo real: Mostrar cómo Airbnb usa JSX para renderizar listados de propiedades dinámicamente.

**Diapositiva 3: Componentes Funcionales**

* **Contenido**:
  + **Definición**: Funciones JavaScript que retornan JSX, representando un fragmento de UI.
  + **Estructura**:
  + import React from 'react';
  + function Greeting() {
  + return <h1>Bienvenidos al Bootcamp de React!</h1>;
  + }
  + export default Greeting;
  + **Uso**: <Greeting /> en otros componentes (ej. App.js).
  + **Ventajas**:
    - Simplicidad: Sin this, más concisos que clases.
    - Compatibilidad con hooks (useState, useEffect).
    - Estándar moderno (90% de proyectos nuevos usan funcionales).
  + **Caso Real**: En Spotify, los componentes funcionales renderizan listas de reproducción dinámicas.
  + **Pregunta para Entrevistas**: ¿Por qué los componentes funcionales han reemplazado a los de clase?
* **Explicación para el Profesor (15 min)**:
  + Comparar componentes con bloques de Lego: reutilizables, modulares, y combinables.
  + Explicar que los componentes funcionales son funciones puras que retornan UI, ideales para hooks (a introducir en la Semana 3).
  + Demostración: Crear Greeting.js, importarlo en App.js, y mostrar el resultado en el navegador.
  + Resaltar que los componentes deben empezar con mayúscula (convención de React para diferenciar de elementos HTML).
  + En entrevistas, los candidatos deben destacar la simplicidad de los funcionales y su integración con hooks como razones para su adopción.
  + Ejemplo real: Mostrar cómo Spotify usa componentes funcionales para botones de play/pausa.

**Diapositiva 4: JSX con Expresiones Dinámicas**

* **Contenido**:
  + **Expresiones JS**: Usa {} para inyectar variables, cálculos, funciones, o condicionales.
  + **Ejemplo Completo**:
  + function UserCard() {
  + const user = { name: "Naomy", age: 25, isActive: true };
  + return (
  + <div>
  + <h2>{user.name}</h2>
  + <p>Edad: {user.age}</p>
  + <p>Estado: {user.isActive ? 'Activo' : 'Inactivo'}</p>
  + <p>Días en el curso: {7 \* 5}</p>
  + </div>
  + );
  + }
  + **Reglas para {}**:
    - Solo expresiones (no bloques como if o for).
    - Ejemplo válido: {user.name || 'Anónimo'}.
    - Ejemplo inválido: {if (user) { user.name }}.
  + **Caso Real**: En Amazon, JSX renderiza precios dinámicos con {product.price.toFixed(2)}.
  + **Pregunta para Entrevistas**: ¿Cómo manejarías lógica condicional en JSX?
* **Explicación para el Profesor (15 min)**:
  + Explicar que {} permite inyectar cualquier expresión JS válida, pero no bloques (usar ternarios o funciones externas).
  + Demostración: Crear UserCard en vivo, modificar user y mostrar cambios dinámicos.
  + Resaltar que en entrevistas, los candidatos deben explicar cómo usar ternarios (?:) o && para condicionales en JSX.
  + Ejemplo real: Mostrar cómo Amazon usa {} para mostrar descuentos condicionalmente (ej. {product.discount ? product.price \* 0.9 : product.price}).
  + Mencionar errores comunes: Usar if dentro de {}, olvidar cerrar etiquetas.

**Diapositiva 5: Práctica Guiada**

* **Contenido**:
  + **Ejercicio**: Crear un componente Greeting personalizado con JSX.
  + **Pasos**:
    1. Crear src/Greeting.js como componente funcional.
    2. Usar JSX con una variable dinámica (ej. nombre del estudiante).
    3. Incluir un condicional en JSX (ej. mostrar un mensaje según el día).
    4. Importar y renderizar en App.js.
  + **Solución Esperada**:
  + // src/Greeting.js
  + import React from 'react';
  + function Greeting() {
  + const name = "Javier y Naomy";
  + const isFirstDay = true;
  + return (
  + <div>
  + <h1>¡Hola, {name}!</h1>
  + <p>{isFirstDay ? 'Bienvenidos al primer día!' : 'Sigan aprendiendo!'}</p>
  + </div>
  + );
  + }
  + export default Greeting;
  + // src/App.js
  + import React from 'react';
  + import Greeting from './Greeting';
  + function App() {
  + return (
  + <div className="App">
  + <Greeting />
  + </div>
  + );
  + }
  + export default App;
  + **Errores Comunes**:
    1. Olvidar import React from 'react' (necesario en CRA para JSX).
    2. Usar class en lugar de className.
    3. No cerrar etiquetas JSX.
  + **Pregunta para Entrevistas**: ¿Por qué React requiere importar React para usar JSX?
* **Explicación para el Profesor (15 min)**:
  + Explicar la práctica: Crear un componente reusable con JSX dinámico.
  + Guiar a los estudiantes para crear un archivo nuevo y usar {} correctamente.
  + Supervisar la importación y renderización en App.js.
  + Resaltar que en entrevistas, los candidatos deben saber que React es necesario porque JSX se compila a React.createElement.
  + Resolver errores: Importaciones incorrectas, sintaxis JSX inválida.

**Diapositiva 6: Cierre y Tareas**

* **Contenido**:
  + **Resumen**:
    - JSX combina HTML y JS, compilado por Babel a funciones React.
    - Componentes funcionales son el estándar para UI modular.
    - {} permite dinamismo en la UI.
  + **Tareas (1-2 horas)**:
    - Leer <https://react.dev/learn/writing-markup-with-jsx>.
    - Crear un componente WelcomeMessage con datos dinámicos (nombre, fecha actual).
    - Añadir un condicional para mostrar un mensaje diferente si es lunes.
  + **Pregunta Reflexiva**: ¿Cómo mejora JSX la productividad del desarrollador?
  + **Pregunta para Entrevistas**: ¿Cómo manejarías un componente que necesita múltiples condicionales en JSX?
* **Explicación para el Profesor (5 min)**:
  + Resumir: JSX simplifica la UI, componentes funcionales son la base de React moderno.
  + Motivar a experimentar con JSX dinámico y leer la documentación.
  + Resaltar que en entrevistas, los candidatos deben explicar cómo usar Fragments y ternarios para JSX complejo.

**Práctica (30 min)**

* **Actividad**: Crear el componente Greeting con JSX dinámico y renderizarlo.
* **Instrucciones para el Profesor**:
  + Pair programming: Naomy escribe el componente, Javier lo integra en App.js.
  + Verificar que usen {} para variables y condicionales correctamente.
  + Resolver errores: Importar React, olvidar elemento raíz, o sintaxis JSX inválida.
* **Resultado Esperado**: Componente funcional con mensaje dinámico y condicional.

**Día 3: Componentes de Clase y Props**

**Objetivo**: Comprender componentes de clase (legacy), compararlos con funcionales, y usar props para pasar datos dinámicos.  
**Duración**: 90 min teoría + 30 min práctica.  
**Recursos**: <https://react.dev/learn/passing-props-to-a-component>, <https://react.dev/learn/your-first-component>.

**Diapositivas**

**Diapositiva 1: Título - Componentes de Clase y Props**

* **Contenido**:
  + **Tema de hoy**: Componentes de Clase y Props.
  + Objetivos:
    - Entender componentes de clase y su rol en proyectos legacy.
    - Comparar con componentes funcionales.
    - Usar props para pasar datos dinámicos.
  + **Importancia**: Props son esenciales para la modularidad; las clases son comunes en código antiguo.
  + **Pregunta para Entrevistas**: ¿Cuándo usarías un componente de clase en lugar de uno funcional?
* **Explicación para el Profesor (5 min)**:
  + Explicar que los componentes de clase son menos comunes hoy, pero importantes para mantener proyectos antiguos.
  + Resaltar que props son la base del flujo de datos en React, un concepto clave en entrevistas.
  + Motivar a los estudiantes: Dominar ambos tipos de componentes los hará versátiles en cualquier codebase.

**Diapositiva 2: Componentes de Clase**

* **Contenido**:
  + **Definición**: Clases ES6 que extienden React.Component, con un método render() para devolver JSX.
  + **Estructura**:
  + import React from 'react';
  + class Welcome extends React.Component {
  + render() {
  + return <h1>Hola desde una clase!</h1>;
  + }
  + }
  + export default Welcome;
  + **Diferencias con Funcionales**:
    - Clases: Usan this, más verbosas, comunes en proyectos pre-2018.
    - Funcionales: Más simples, compatibles con hooks, estándar moderno.
  + **Casos de Uso**:
    - Proyectos legacy (ej. apps de bancos con código antiguo).
    - Cuando se necesita compatibilidad con bibliotecas antiguas.
  + **Caso Real**: En LinkedIn, los componentes de clase se usaban en versiones tempranas del feed (pre-hooks).
  + **Pregunta para Entrevistas**: ¿Por qué React introdujo hooks para reemplazar componentes de clase?
* **Explicación para el Profesor (20 min)**:
  + Explicar que las clases requieren this para acceder a métodos/props, lo que las hace más complejas.
  + Comparar con funcionales: Menos código, sin this, mejor para hooks.
  + Demostración: Crear un componente de clase Welcome y renderizarlo en App.js.
  + Resaltar que en entrevistas, los candidatos deben mencionar que los hooks (useState, useEffect) eliminaron la necesidad de clases, simplificando el código.
  + Ejemplo real: Mostrar cómo un banco usa clases para mantener un sistema de formularios antiguo.
  + Mencionar que el 80% de proyectos modernos usan funcionales, pero las clases son comunes en mantenimiento.

**Diapositiva 3: Introducción a Props**

* **Contenido**:
  + **Definición**: Props (propiedades) son datos inmutables pasados de un componente padre a un hijo.
  + **Ejemplo en Componente Funcional**:
  + function User({ name, age }) {
  + return (
  + <div>
  + <h2>{name}</h2>
  + <p>Edad: {age}</p>
  + </div>
  + );
  + }
  + // Uso: <User name="Naomy" age={25} />
  + **Ejemplo en Componente de Clase**:
  + import React from 'react';
  + class User extends React.Component {
  + render() {
  + const { name, age } = this.props;
  + return (
  + <div>
  + <h2>{name}</h2>
  + <p>Edad: {age}</p>
  + </div>
  + );
  + }
  + }
  + // Uso: <User name="Javier" age={30} />
  + **Reglas de Props**:
    - Inmutables: Los hijos no pueden modificar props.
    - Tipos: Strings, números, objetos, funciones, etc.
    - Desestructuración: Simplifica acceso en funcionales.
  + **Caso Real**: En Airbnb, props pasan datos de propiedades (precio, ubicación) a componentes de tarjetas.
  + **Pregunta para Entrevistas**: ¿Qué significa que los props son inmutables y cómo afecta el diseño?
* **Explicación para el Profesor (20 min)**:
  + Explicar que props son como parámetros de una función, pero no se pueden modificar en el hijo.
  + Demostración: Crear User (funcional y clase), pasar props desde App.js, y mostrar el resultado.
  + Resaltar que desestructurar props ({name, age}) mejora la legibilidad en funcionales.
  + En entrevistas, los candidatos deben explicar que la inmutabilidad de props asegura un flujo de datos predecible, facilitando debugging.
  + Ejemplo real: Mostrar cómo Airbnb usa props para renderizar detalles de una propiedad dinámicamente.
  + Mencionar errores comunes: Intentar modificar props, olvidar this.props en clases.

**Diapositiva 4: Flujo de Datos Unidireccional**

* **Contenido**:
  + **Concepto**: Los datos fluyen de padres a hijos vía props, nunca al revés.
  + **Ejemplo Completo**:
  + import React from 'react';
  + function App() {
  + const users = [
  + { name: "Javier", role: "Estudiante" },
  + { name: "Naomy", role: "Desarrolladora" }
  + ];
  + return (
  + <div>
  + {users.map(user => (
  + <UserCard key={user.name} name={user.name} role={user.role} />
  + ))}
  + </div>
  + );
  + }
  + function UserCard({ name, role }) {
  + return (
  + <div>
  + <h3>{name}</h3>
  + <p>Rol: {role}</p>
  + </div>
  + );
  + }
  + **Beneficios**:
    - Debugging más fácil (sabes de dónde vienen los datos).
    - Código predecible, escalable para apps grandes.
  + **Caso Real**: En Facebook, el flujo unidireccional asegura que los comentarios fluyan del componente padre al hijo.
  + **Pregunta para Entrevistas**: ¿Cómo manejarías comunicación de hijo a padre en React?
* **Explicación para el Profesor (15 min)**:
  + Explicar que el flujo unidireccional evita efectos secundarios inesperados.
  + Demostración: Crear App y UserCard, pasar props, y mostrar cómo map renderiza múltiples componentes.
  + Resaltar que en entrevistas, los candidatos deben mencionar que la comunicación hijo-padre se logra con funciones pasadas como props (callbacks, a introducir en la Semana 2).
  + Ejemplo real: Mostrar cómo Facebook usa props para pasar datos de posts a componentes de comentarios.
  + Mencionar que key es obligatorio en listas para optimizar el Virtual DOM.

**Diapositiva 5: Práctica Guiada**

* **Contenido**:
  + **Ejercicio**: Crear una familia de componentes con props.
  + **Pasos**:
    1. Crear UserCard.js (funcional) y UserProfile.js (clase).
    2. Pasar props name, role, y city desde App.js.
    3. Renderizar múltiples instancias con datos dinámicos.
  + **Solución Esperada**:
  + // src/UserCard.js
  + import React from 'react';
  + function UserCard({ name, role, city }) {
  + return (
  + <div>
  + <h3>{name}</h3>
  + <p>Rol: {role}</p>
  + <p>Ciudad: {city}</p>
  + </div>
  + );
  + }
  + export default UserCard;
  + // src/UserProfile.js
  + import React from 'react';
  + class UserProfile extends React.Component {
  + render() {
  + const { name, role, city } = this.props;
  + return (
  + <div>
  + <h3>{name}</h3>
  + <p>Rol: {role}</p>
  + <p>Ciudad: {city}</p>
  + </div>
  + );
  + }
  + }
  + export default UserProfile;
  + // src/App.js
  + import React from 'react';
  + import UserCard from './UserCard';
  + import UserProfile from './UserProfile';
  + function App() {
  + return (
  + <div className="App">
  + <UserCard name="Javier" role="Estudiante" city="Bogotá" />
  + <UserProfile name="Naomy" role="Desarrolladora" city="Medellín" />
  + </div>
  + );
  + }
  + export default App;
  + **Errores Comunes**:
    1. Olvidar this.props en clases.
    2. No desestructurar props en funcionales.
    3. Olvidar key en listas.
  + **Pregunta para Entrevistas**: ¿Por qué es importante el flujo de datos unidireccional en React?
* **Explicación para el Profesor (15 min)**:
  + Explicar la práctica: Comparar clases y funcionales usando props.
  + Guiar a los estudiantes para crear archivos separados y pasar múltiples props.
  + Supervisar que usen key si renderizan listas.
  + Resaltar que en entrevistas, los candidatos deben explicar cómo el flujo unidireccional evita bugs en apps grandes.

**Diapositiva 6: Cierre y Tareas**

* **Contenido**:
  + **Resumen**:
    - Componentes de clase son legacy, funcionales son el estándar.
    - Props permiten datos dinámicos con flujo unidireccional.
  + **Tareas (1-2 horas)**:
    - Leer <https://react.dev/learn/passing-props-to-a-component>.
    - Crear un componente ProfileCard con props adicionales (email, hobby).
    - Renderizar una lista de ProfileCard con map.
  + **Pregunta Reflexiva**: ¿Cómo facilita el flujo unidireccional el mantenimiento de una app?
  + **Pregunta para Entrevistas**: ¿Cómo optimiza React el renderizado de listas con props?
* **Explicación para el Profesor (5 min)**:
  + Resumir: Clases vs. funcionales, props como base de la modularidad.
  + Motivar a experimentar con listas y props dinámicos.

**Práctica (30 min)**

* **Actividad**: Crear UserCard y UserProfile, renderizarlos con props en App.js.
* **Instrucciones para el Profesor**:
  + Pair programming: Javier crea el componente funcional, Naomy el de clase.
  + Verificar que pasen múltiples props y los rendericen correctamente.
  + Resolver errores: Olvidar this, props mal escritos, o falta de key.
* **Resultado Esperado**: Dos componentes mostrando datos dinámicos con props.

**Día 4: State y Manejo de Eventos**

**Objetivo**: Introducir el estado con useState y manejar eventos como onClick para crear interactividad.  
**Duración**: 90 min teoría + 30 min práctica.  
**Recursos**: <https://react.dev/learn/state-a-components-memory>, <https://react.dev/learn/responding-to-events>.

**Diapositivas**

**Diapositiva 1: Título - State y Manejo de Eventos**

* **Contenido**:
  + **Tema de hoy**: State y Manejo de Eventos.
  + Objetivos:
    - Usar useState para manejar datos dinámicos.
    - Implementar eventos como onClick para interactividad.
    - Crear componentes dinámicos e interactivos.
  + **Importancia**: Estado y eventos son la base de la interactividad en React, esenciales para apps reales.
  + **Pregunta para Entrevistas**: ¿Cómo funciona useState y por qué es asíncrono?
* **Explicación para el Profesor (5 min)**:
  + Explicar que el estado permite que los componentes "recuerden" datos, y los eventos conectan la UI con el usuario.
  + Resaltar que useState es el hook fundamental para componentes funcionales.
  + Motivar: La interactividad es lo que hace que apps como Twitter o Instagram sean dinámicas.

**Diapositiva 2: Introducción al Estado (useState)**

* **Contenido**:
  + **Definición**: El estado es un objeto interno que un componente usa para almacenar datos que cambian (ej. un contador, un formulario).
  + **Hook useState**:
  + import React, { useState } from 'react';
  + function Counter() {
  + const [count, setCount] = useState(0);
  + return (
  + <div>
  + <p>Contador: {count}</p>
  + <button onClick={() => setCount(count + 1)}>Incrementar</button>
  + </div>
  + );
  + }
  + **Cómo Funciona**:
    - useState(valorInicial) retorna un array: [estado, setter].
    - setCount actualiza el estado y dispara un re-render.
    - Asíncrono: React agrupa actualizaciones para optimizar rendimiento.
  + **Reglas de useState**:
    - Solo en componentes funcionales o hooks personalizados.
    - No modificar el estado directamente (usar setCount).
    - Usar funciones de actualización para valores previos: setCount(prev => prev + 1).
  + **Caso Real**: En Twitter, el estado maneja el contador de likes en un tweet.
  + **Pregunta para Entrevistas**: ¿Por qué no debemos modificar el estado directamente en React?
* **Explicación para el Profesor (20 min)**:
  + Explicar que useState es el núcleo de la interactividad en React moderno.
  + Usar una analogía: El estado es como la memoria de un componente, y setCount es el lápiz que actualiza esa memoria.
  + Demostración: Crear un contador en vivo, mostrar cómo setCount re-renderiza el componente.
  + Resaltar que setCount es asíncrono; en entrevistas, los candidatos deben explicar que React batching optimiza actualizaciones.
  + Ejemplo real: Mostrar cómo Twitter usa el estado para actualizar likes sin recargar la página.
  + Mencionar errores comunes: Modificar estado directamente (count++), olvidar importar useState.

**Diapositiva 3: Manejo de Eventos**

* **Contenido**:
  + **Definición**: Funciones que responden a interacciones del usuario (clicks, inputs, teclas).
  + **Sintaxis en React**:
    - Usa camelCase: onClick, onChange, onSubmit.
    - Pasa funciones, no las invoques: onClick={handleClick} (correcto) vs. onClick={handleClick()} (invoca inmediatamente).
  + **Ejemplo**:
  + import React from 'react';
  + function Button() {
  + const handleClick = () => alert('¡Clic en el botón!');
  + return <button onClick={handleClick}>Haz clic</button>;
  + }
  + **Eventos Comunes**:
    - onClick: Clic en un elemento.
    - onChange: Cambio en un input.
    - onSubmit: Envío de formulario.
  + **Caso Real**: En Gmail, onClick maneja el envío de emails desde un botón.
  + **Pregunta para Entrevistas**: ¿Cómo evita React que los eventos se activen accidentalmente?
* **Explicación para el Profesor (15 min)**:
  + Explicar que los eventos en React son sintéticos, normalizando comportamientos entre navegadores.
  + Demostración: Crear un botón con onClick que muestre un alert.
  + Resaltar que en entrevistas, los candidatos deben explicar que pasar handleClick() ejecuta la función al renderizar, no al hacer clic.
  + Ejemplo real: Mostrar cómo Gmail usa onClick para acciones como "Enviar" o "Archivar".
  + Mencionar errores comunes: Invocar funciones directamente, olvidar camelCase.

**Diapositiva 4: Combinando Estado y Eventos**

* **Contenido**:
  + **Concepto**: Estado y eventos trabajan juntos para crear UI dinámica.
  + **Ejemplo Completo**:
  + import React, { useState } from 'react';
  + function Toggle() {
  + const [isOn, setIsOn] = useState(false);
  + const handleToggle = () => setIsOn(!isOn);
  + return (
  + <div>
  + <p>Estado: {isOn ? 'Encendido' : 'Apagado'}</p>
  + <button onClick={handleToggle}>Cambiar estado</button>
  + </div>
  + );
  + }
  + **Cómo Funciona**:
    - useState almacena el estado (isOn).
    - handleToggle actualiza el estado con setIsOn.
    - El re-render muestra el nuevo estado.
  + **Caso Real**: En Slack, el estado maneja el toggle de notificaciones (activadas/desactivadas).
  + **Pregunta para Entrevistas**: ¿Cómo manejarías múltiples actualizaciones de estado en un solo evento?
* **Explicación para el Profesor (15 min)**:
  + Explicar que la combinación de estado y eventos es la base de la interactividad en React.
  + Demostración: Crear Toggle en vivo, probar el cambio de estado.
  + Resaltar que en entrevistas, los candidatos deben mencionar el uso de funciones de actualización (prev) para manejar múltiples cambios de estado.
  + Ejemplo real: Mostrar cómo Slack usa estado para toggles de configuración.
  + Mencionar errores comunes: Actualizar estado directamente, no usar funciones de evento.

**Diapositiva 5: Práctica Guiada**

* **Contenido**:
  + **Ejercicio**: Crear un componente Counter con botones para incrementar/decrementar.
  + **Pasos**:
    1. Crear Counter.js con useState.
    2. Añadir botones para modificar el estado.
    3. Renderizar en App.js.
  + **Solución Esperada**:
  + // src/Counter.js
  + import React, { useState } from 'react';
  + function Counter() {
  + const [count, setCount] = useState(0);
  + return (
  + <div>
  + <p>Contador: {count}</p>
  + <button onClick={() => setCount(prev => prev + 1)}>Incrementar</button>
  + <button onClick={() => setCount(prev => prev - 1)}>Decrementar</button>
  + </div>
  + );
  + }
  + export default Counter;
  + // src/App.js
  + import React from 'react';
  + import Counter from './Counter';
  + function App() {
  + return (
  + <div className="App">
  + <Counter />
  + </div>
  + );
  + }
  + export default App;
  + **Errores Comunes**:
    1. Olvidar importar useState.
    2. Usar count++ en lugar de setCount.
    3. Invocar funciones directamente en onClick.
  + **Pregunta para Entrevistas**: ¿Por qué es importante usar funciones de actualización en useState?
* **Explicación para el Profesor (15 min)**:
  + Explicar la práctica: Crear un contador interactivo con botones.
  + Guiar a los estudiantes para usar prev en setCount para actualizaciones seguras.
  + Supervisar que los eventos estén correctamente definidos.
  + Resaltar que en entrevistas, los candidatos deben explicar que prev asegura consistencia en actualizaciones asíncronas.

**Diapositiva 6: Cierre y Tareas**

* **Contenido**:
  + **Resumen**:
    - useState maneja datos dinámicos.
    - Eventos como onClick conectan la UI con el usuario.
    - La combinación crea apps interactivas.
  + **Tareas (1-2 horas)**:
    - Leer <https://react.dev/learn/state-a-components-memory>.
    - Crear un componente ToggleLight (encendido/apagado con texto dinámico).
    - Añadir un botón para resetear el estado.
  + **Pregunta Reflexiva**: ¿Cómo afecta el estado a la experiencia del usuario?
  + **Pregunta para Entrevistas**: ¿Cómo optimiza React las actualizaciones de estado?
* **Explicación para el Profesor (5 min)**:
  + Resumir: Estado y eventos son la clave de la interactividad.
  + Motivar a experimentar con más eventos (onChange, onMouseOver).

**Práctica (30 min)**

* **Actividad**: Crear el componente Counter con botones de incremento/decremento.
* **Instrucciones para el Profesor**:
  + Pair programming: Naomy define el estado, Javier los eventos.
  + Verificar que usen useState y funciones de evento correctamente.
  + Resolver errores: Importaciones olvidadas, actualizaciones directas del estado.
* **Resultado Esperado**: Contador funcional con botones interactivos.

**Día 5: Proyecto 1 - App de Todo List Simple**

**Objetivo**: Integrar JSX, componentes, props, estado y eventos en una aplicación práctica.  
**Duración**: 90 min teoría + 30 min práctica.  
**Recursos**: <https://react.dev/learn>, <https://react.dev/learn/thinking-in-react>.

**Diapositivas**

**Diapositiva 1: Título - Proyecto 1: Todo List Simple**

* **Contenido**:
  + **Tema de hoy**: Proyecto 1 - App de Todo List Simple.
  + Objetivos:
    - Combinar JSX, componentes, props, estado y eventos.
    - Construir una app funcional para un portafolio.
    - Aplicar pensamiento React (dividir UI en componentes).
  + **Funcionalidades**:
    - Agregar tareas via formulario.
    - Mostrar lista de tareas.
    - Eliminar tareas individuales.
  + **Importancia**: Todo List es un proyecto clásico en entrevistas y portafolios.
  + **Pregunta para Entrevistas**: ¿Cómo dividirías una app como Todo List en componentes?
* **Explicación para el Profesor (5 min)**:
  + Explicar que este proyecto integra todos los conceptos de la semana, preparando a los estudiantes para apps más complejas.
  + Resaltar que Todo List es una pregunta común en entrevistas técnicas, evaluando diseño de componentes y manejo de estado.
  + Motivar: Este proyecto será parte de su portafolio profesional.

**Diapositiva 2: Diseño de la App (Pensamiento React)**

* **Contenido**:
  + **Enfoque React**: Dividir la UI en componentes reutilizables.
  + **Estructura**:
    - App: Maneja el estado (lista de tareas, input) y el formulario.
    - TodoItem: Muestra una tarea con botón de eliminar.
  + **Flujo de Datos**:
    - App almacena tareas en el estado.
    - Pasa tareas y funciones de eliminación a TodoItem via props.
    - Eventos (onClick, onChange) actualizan el estado.
  + **Diagrama**:
  + App
  + ├── Input + Botón (estado: input, todos)
  + └── TodoItem (props: todo, deleteTodo)
  + **Caso Real**: En Trello, la UI de tableros usa componentes similares (tablero → tarjeta).
  + **Pregunta para Entrevistas**: ¿Cómo estructurarías una app React para escalabilidad?
* **Explicación para el Profesor (15 min)**:
  + Explicar el "pensamiento React": Dividir la UI en componentes pequeños y reutilizables.
  + Dibujar el diagrama en una pizarra: App como padre, TodoItem como hijo.
  + Resaltar que en entrevistas, los candidatos deben explicar cómo dividen la UI en componentes basados en responsabilidad única.
  + Ejemplo real: Mostrar cómo Trello usa componentes para tarjetas individuales.

**Diapositiva 3: Código de App.js**

* **Contenido**:
* import React, { useState } from 'react';
* import TodoItem from './TodoItem';
* function App() {
* const [todos, setTodos] = useState([]);
* const [input, setInput] = useState('');
* const addTodo = () => {
* if (input.trim()) {
* setTodos([...todos, { id: Date.now(), text: input }]);
* setInput('');
* }
* };
* const deleteTodo = (id) => {
* setTodos(todos.filter(todo => todo.id !== id));
* };
* return (
* <div className="App">
* <h1>Todo List</h1>
* <input
* value={input}
* onChange={(e) => setInput(e.target.value)}
* placeholder="Nueva tarea"
* />
* <button onClick={addTodo}>Agregar</button>
* {todos.map(todo => (
* <TodoItem key={todo.id} todo={todo} deleteTodo={deleteTodo} />
* ))}
* </div>
* );
* }
* export default App;
  + **Explicación del Código**:
    - useState: todos almacena la lista, input el valor del formulario.
    - addTodo: Agrega tarea con ID único (Date.now()), limpia el input.
    - deleteTodo: Filtra tareas por ID.
    - input: Controlado con value y onChange.
    - map: Renderiza TodoItem con key para optimización.
  + **Caso Real**: En Asana, el estado maneja listas de tareas dinámicas.
  + **Pregunta para Entrevistas**: ¿Por qué es importante usar key en listas dinámicas?
* **Explicación para el Profesor (20 min)**:
  + Explicar cada línea:
    - useState para manejar estado dinámico.
    - ...todos (spread operator) para no mutar el estado directamente.
    - filter para eliminar tareas sin afectar el array original.
    - key para que React identifique elementos en listas.
  + Demostración: Crear App.js, agregar y eliminar tareas en vivo.
  + Resaltar que en entrevistas, los candidatos deben explicar que key optimiza el diffing del Virtual DOM.
  + Ejemplo real: Mostrar cómo Asana usa listas dinámicas para tareas.
  + Mencionar errores comunes: Olvidar key, mutar estado directamente.

**Diapositiva 4: Código de TodoItem.js**

* **Contenido**:
* import React from 'react';
* function TodoItem({ todo, deleteTodo }) {
* return (
* <div>
* <span>{todo.text}</span>
* <button onClick={() => deleteTodo(todo.id)}>Eliminar</button>
* </div>
* );
* }
* export default TodoItem;
  + **Explicación del Código**:
    - Recibe todo (objeto con id, text) y deleteTodo (función) como props.
    - onClick ejecuta deleteTodo con el ID de la tarea.
    - Componente simple, enfocado en mostrar y eliminar una tarea.
  + **Caso Real**: En Jira, los tickets se renderizan como componentes individuales.
  + **Pregunta para Entrevistas**: ¿Cómo pasarías una función como prop para comunicación hijo-padre?
* **Explicación para el Profesor (15 min)**:
  + Explicar que TodoItem es un componente presentacional, recibiendo datos y funciones via props.
  + Demostración: Crear TodoItem.js, integrarlo con App.js, y probar la eliminación.
  + Resaltar que en entrevistas, los candidatos deben explicar cómo las funciones como props habilitan la comunicación hijo-padre.
  + Ejemplo real: Mostrar cómo Jira usa componentes para tickets con botones de acción.

**Diapositiva 5: Práctica Guiada**

* **Contenido**:
  + **Ejercicio**: Construir la app Todo List en pareja.
  + **Pasos**:
    1. Crear TodoItem.js con props para mostrar y eliminar tareas.
    2. Implementar App.js con estado, formulario, y lista.
    3. Conectar ambos componentes con props.
    4. Probar agregar y eliminar tareas.
  + **Solución**: Usar el código de las diapositivas 3 y 4.
  + **Errores Comunes**:
    1. Olvidar key en map.
    2. Mutar estado directamente (todos.push()).
    3. No limpiar el input tras agregar.
  + **Pregunta para Entrevistas**: ¿Cómo manejarías el estado en una app Todo List escalable?
* **Explicación para el Profesor (15 min)**:
  + Explicar la práctica: Construir una app completa con pair programming.
  + Dividir tareas: Javier escribe App.js, Naomy escribe TodoItem.js.
  + Supervisar el uso de key, props, y eventos.
  + Resaltar que en entrevistas, los candidatos deben proponer Redux o Context para estados complejos.

**Diapositiva 6: Cierre y Tareas**

* **Contenido**:
  + **Resumen**:
    - Todo List combina JSX, componentes, props, estado y eventos.
    - Proyecto listo para el portafolio.
  + **Tareas (1-2 horas)**:
    - Mejorar la app: Agregar un botón para limpiar todas las tareas.
    - Leer <https://react.dev/learn/thinking-in-react>.
    - Añadir un condicional para mostrar "No hay tareas" si la lista está vacía.
  + **Pregunta Reflexiva**: ¿Cómo mejora la modularidad la mantenibilidad de la app?
  + **Pregunta para Entrevistas**: ¿Cómo optimizarías el rendimiento de una lista dinámica?
* **Explicación para el Profesor (5 min)**:
  + Resumir la semana: Fundamentos listos para módulos avanzados.
  + Felicitara los estudiantes por su primera app React funcional.

**Práctica (30 min)**

* **Actividad**: Construir la app Todo List.
* **Instrucciones para el Profesor**:
  + Pair programming: Naomy implementa el estado y formulario, Javier el componente TodoItem.
  + Verificar que usen key, spread operator, y eventos correctamente.
  + Resolver errores: Falta de key, mutación directa del estado, importaciones incorrectas.
* **Resultado Esperado**: App funcional para agregar y eliminar tareas.

**Recursos Adicionales**

* **Documentación**:
  + React Official Docs: <https://react.dev/>.
  + Create React App: <https://create-react-app.dev/>.
* **Videos**:
  + Traversy Media: "React Crash Course" (YouTube).
  + Net Ninja: "React Tutorial for Beginners" (YouTube).
  + freeCodeCamp: "React Tutorial" (YouTube).
* **Práctica Extra**:
  + freeCodeCamp: Sección de React (<https://www.freecodecamp.org/learn>).
  + CodeSandbox: Prototipar la Todo List (<https://codesandbox.io/>).
  + LeetCode: Problemas de JavaScript para reforzar lógica.

Repaso completo de semana 1

**Resumen Semana 1: Fundamentos de JavaScript y React.js para Todo List**

**Diapositiva 1: Título - Repaso para el Proyecto Todo List**

**Contenido:**

* **Tema de hoy**: Repaso de fundamentos de JavaScript y React.js para construir una app Todo List profesional.
* **Objetivos**:
  + Reforzar conceptos básicos de JavaScript: tipos de datos, variables, condicionales, funciones y arrays.
  + Repasar fundamentos de React.js: JSX, componentes funcionales, props, estado y eventos.
  + Preparar a Javier y Naomy para completar el proyecto Todo List con confianza.
* **Importancia**: Dominar JavaScript y React es crucial para construir aplicaciones modernas y aprobar entrevistas técnicas.
* **Pregunta para Entrevistas**: ¿Qué conceptos de JavaScript son esenciales para trabajar con React?

**Explicación para el Profesor (5 min):**

* Saludar a los estudiantes y destacar que este repaso unifica JavaScript y React para el proyecto final.
* Explicar que JavaScript es el pilar de React, y los conceptos de esta semana son suficientes para una app Todo List profesional.
* Motivar: Este repaso los preparará para entrevistas técnicas y proyectos reales.

**Diapositiva 2: Fundamentos de JavaScript - Tipos de Datos**

**Contenido:**

* **Definición**: Los tipos de datos determinan qué valores puede almacenar una variable y cómo se comportan.
* **Tipos Principales**:
  + **Primitivos**:
    - string: Texto (ej. "Hola, Naomy").
    - number: Números (ej. 42, 3.14).
    - boolean: true o false.
    - null: Valor nulo.
    - undefined: Variable sin valor asignado.
  + **No Primitivos**:
    - object: Colección de propiedades (ej. { name: "Javier", age: 30 }).
    - array: Lista ordenada (ej. ["tarea1", "tarea2"]).
* **Ejemplo para Todo List**:
* const todo = "Comprar leche"; // string
* const priority = 1; // number
* const isCompleted = false; // boolean
* const todos = ["Tarea 1", "Tarea 2"]; // array
* const task = { id: 1, text: "Estudiar React" }; // object
* **Caso Real**: En una app Todo List, los arrays almacenan listas de tareas, y los objetos representan tareas individuales.
* **Pregunta para Entrevistas**: ¿Cuál es la diferencia entre null y undefined en JavaScript?

**Explicación para el Profesor (10 min):**

* Explicar cada tipo con ejemplos: string para nombres de tareas, array para la lista de tareas.
* Analogía: Los tipos de datos son como ingredientes en una receta; cada uno tiene un propósito (texto para mostrar, números para IDs).
* Demostración: En la consola de Chrome, mostrar typeof "Hola", typeof 42, y typeof {}.
* Resaltar que en entrevistas, los candidatos deben diferenciar null (intencionalmente vacío) de undefined (sin asignar).
* Conectar con Todo List: Usaremos arrays para almacenar tareas y objetos para cada tarea con id y text.

**Diapositiva 3: Tipos de Variables y Alcance**

**Contenido:**

* **Tipos de Variables**:
  + var: Declaración antigua, alcance de función, propensa a errores.
  + let: Alcance de bloque, reasignable, ideal para valores que cambian.
  + const: Alcance de bloque, no reasignable (pero objetos/arrays mutables).
* **Ejemplo**:
* let taskCount = 0; // Cambia al agregar tareas
* const appName = "Todo List"; // No cambia
* const todos = []; // Constante, pero el contenido puede mutar
* todos.push("Estudiar JavaScript"); // Válido
* **Alcance**:
  + **Global**: Fuera de funciones/bloques.
  + **Bloque**: Dentro de {} (if, for).
  + **Función**: Dentro de una función.
* **Ejemplo para Todo List**:
* function addTask(task) {
* let id = Date.now(); // Alcance de bloque
* const newTodo = { id, task }; // Objeto constante
* todos.push(newTodo); // Mutamos el array
* }
* **Caso Real**: En apps como Trello, const se usa para datos fijos (nombres de tableros), y let para contadores dinámicos.
* **Pregunta para Entrevistas**: ¿Por qué preferimos const y let sobre var?

**Explicación para el Profesor (10 min):**

* Explicar que const es ideal para estados en React (useState usa const), pero permite mutar arrays/objetos.
* Analogía: const es como un cuaderno con páginas fijas, pero puedes escribir dentro; let es un borrador reescribible.
* Demostración: En la consola, mostrar cómo const todos = [] permite todos.push(), pero no todos = [].
* Resaltar que en entrevistas, los candidatos deben mencionar que let y const evitan problemas de alcance de var (hoisting).
* Conectar con Todo List: Usaremos const para estados y let para variables temporales en funciones.

**Diapositiva 4: Condicionales y Lógica**

**Contenido:**

* **Definición**: Condicionales controlan el flujo del programa basado en condiciones.
* **Estructuras**:
  + if/else:
  + if (todos.length === 0) {
  + console.log("No hay tareas");
  + } else {
  + console.log(`Tienes ${todos.length} tareas`);
  + }
  + Operador ternario (?:):
  + const message = todos.length === 0 ? "No hay tareas" : "Tareas pendientes";
  + Operador lógico &&:
  + todos.length > 0 && console.log("Mostrar lista");
* **Ejemplo para Todo List**:
* function renderTodos(todos) {
* return todos.length === 0 ? "No hay tareas" : todos.map(todo => todo.text);
* }
* **Caso Real**: En Asana, los condicionales muestran "Lista vacía" o la lista de tareas según el estado.
* **Pregunta para Entrevistas**: ¿Cómo usarías un operador ternario en JSX para renderizado condicional?

**Explicación para el Profesor (10 min):**

* Explicar que los condicionales son esenciales para mostrar contenido dinámico en React (ej. lista vacía).
* Analogía: Los condicionales son como un semáforo: deciden qué camino tomar según la situación.
* Demostración: En la consola, probar un if y un ternario con un array de tareas.
* Resaltar que en JSX, los ternarios y && son preferidos para renderizado condicional, ya que if no funciona dentro de {}.
* Conectar con Todo List: Usaremos ternarios en JSX para mostrar "No hay tareas" si el array está vacío.

**Diapositiva 5: Funciones y Arrays**

**Contenido:**

* **Funciones**:
  + Declaración: function addTodo(task) { ... }
  + Expresión: const addTodo = () => { ... }
  + Arrow functions: const addTodo = () => todos.push(task);
* **Métodos de Arrays**:
  + push: Añade al final (todos.push("Tarea")).
  + filter: Filtra elementos (todos.filter(todo => todo.id !== id)).
  + map: Transforma elementos (todos.map(todo => todo.text)).
* **Ejemplo para Todo List**:
* const todos = [];
* const addTodo = (text) => todos.push({ id: Date.now(), text });
* const deleteTodo = (id) => todos.filter(todo => todo.id !== id);
* const renderTodos = () => todos.map(todo => `${todo.text} (ID: ${todo.id})`);
* addTodo("Estudiar React"); // Añade tarea
* console.log(renderTodos()); // Muestra tareas
* **Caso Real**: En Jira, map renderiza listas de tickets, filter elimina tickets completados.
* **Pregunta para Entrevistas**: ¿Por qué usamos arrow functions en React?

**Explicación para el Profesor (10 min):**

* Explicar que las arrow functions son comunes en React por su concisión y manejo de this.
* Analogía: Los arrays son como una lista de compras; map la lee, filter elimina ítems, push añade nuevos.
* Demostración: En la consola, usar push, filter y map con un array de tareas.
* Resaltar que en entrevistas, los candidatos deben mencionar que map y filter son inmutables, ideales para el estado en React.
* Conectar con Todo List: Usaremos map para renderizar tareas, filter para eliminar, y arrow functions para eventos.

**Diapositiva 6: Repaso de React - JSX y Componentes Funcionales**

**Contenido:**

* **JSX**:
  + Sintaxis que combina HTML y JavaScript, compilada por Babel a React.createElement.
  + Reglas: Elemento raíz único, camelCase (className, onClick), expresiones con {}.
  + Ejemplo:
  + const name = "Javier";
  + const element = <h1>¡Hola, {name}!</h1>;
* **Componentes Funcionales**:
  + Funciones que retornan JSX, reutilizables y simples.
  + Ejemplo:
  + import React from 'react';
  + function Task({ text }) {
  + return <p>{text}</p>;
  + }
  + export default Task;
* **Ejemplo para Todo List**:
* function TodoItem({ todo }) {
* return <div><span>{todo.text}</span></div>;
* }
* **Caso Real**: En Netflix, JSX renderiza carruseles, componentes funcionales muestran películas.
* **Pregunta para Entrevistas**: ¿Por qué JSX requiere un elemento raíz único?

**Explicación para el Profesor (10 min):**

* Explicar que JSX es el núcleo de React, y los componentes funcionales son el estándar moderno.
* Analogía: JSX es como un plano de UI; los componentes son los bloques de construcción.
* Demostración: Crear un componente Task en CodeSandbox, renderizarlo con JSX dinámico.
* Resaltar que en entrevistas, los candidatos deben explicar que JSX se compila a funciones y usa Fragments (<>...</>) para evitar div innecesarios.
* Conectar con Todo List: Usaremos JSX para renderizar tareas y componentes funcionales para TodoItem.

**Diapositiva 7: Props y Flujo Unidireccional**

**Contenido:**

* **Props**: Datos inmutables pasados de padre a hijo.
  + Ejemplo:
  + function TodoItem({ text, id }) {
  + return <p>{text} (ID: {id})</p>;
  + }
  + // Uso: <TodoItem text="Estudiar" id={1} />
* **Flujo Unidireccional**: Datos fluyen de padre a hijo, no al revés.
  + Ejemplo:
  + function App() {
  + const todos = [{ id: 1, text: "Tarea 1" }];
  + return todos.map(todo => <TodoItem key={todo.id} text={todo.text} />);
  + }
* **Caso Real**: En Airbnb, props pasan datos de propiedades a componentes de tarjetas.
* **Pregunta para Entrevistas**: ¿Cómo manejarías comunicación hijo-padre en React?

**Explicación para el Profesor (10 min):**

* Explicar que props son como parámetros de función, y el flujo unidireccional asegura código predecible.
* Analogía: Props son como notas que un padre le pasa a un hijo; el hijo no puede cambiarlas.
* Demostración: Crear TodoItem y pasar props desde App en CodeSandbox.
* Resaltar que en entrevistas, los candidatos deben mencionar que funciones como props (callbacks) permiten comunicación hijo-padre.
* Conectar con Todo List: Pasaremos text y id a TodoItem, y una función deleteTodo para eliminar tareas.

**Diapositiva 8: Estado (useState) y Eventos**

**Contenido:**

* **Estado (useState)**:
  + Hook para manejar datos dinámicos en componentes funcionales.
  + Ejemplo:
  + import React, { useState } from 'react';
  + function Counter() {
  + const [count, setCount] = useState(0);
  + return <button onClick={() => setCount(count + 1)}>{count}</button>;
  + }
* **Eventos**:
  + Responden a interacciones (onClick, onChange).
  + Ejemplo:
  + function Input() {
  + const [value, setValue] = useState('');
  + return <input value={value} onChange={(e) => setValue(e.target.value)} />;
  + }
* **Ejemplo para Todo List**:
* function App() {
* const [todos, setTodos] = useState([]);
* const [input, setInput] = useState('');
* const addTodo = () => {
* if (input) setTodos([...todos, { id: Date.now(), text: input }]);
* };
* return (
* <div>
* <input value={input} onChange={(e) => setInput(e.target.value)} />
* <button onClick={addTodo}>Agregar</button>
* </div>
* );
* }
* **Caso Real**: En Twitter, el estado maneja el texto de un tweet, onChange actualiza el input.
* **Pregunta para Entrevistas**: ¿Por qué es asíncrono setState en React?

**Explicación para el Profesor (15 min):**

* Explicar que useState es la memoria de un componente, y los eventos conectan la UI con el usuario.
* Analogía: El estado es como un cuaderno de notas; setState escribe nuevas entradas.
* Demostración: Crear un input controlado y un botón para añadir tareas en CodeSandbox.
* Resaltar que en entrevistas, los candidatos deben explicar que setState es asíncrono para optimizar renders.
* Conectar con Todo List: Usaremos useState para la lista de tareas y el input, onChange para capturar texto, y onClick para agregar/eliminar.

**Diapositiva 9: Construyendo Todo List - Código Completo**

**Contenido:**

* **Estructura**:
  + App.js: Maneja estado (todos, input), formulario, y lista.
  + TodoItem.js: Muestra tarea y botón de eliminar.
* **Código de App.js**:
* import React, { useState } from 'react';
* import TodoItem from './TodoItem';
* function App() {
* const [todos, setTodos] = useState([]);
* const [input, setInput] = useState('');
* const addTodo = () => {
* if (input.trim()) {
* setTodos([...todos, { id: Date.now(), text: input }]);
* setInput('');
* }
* };
* const deleteTodo = (id) => {
* setTodos(todos.filter(todo => todo.id !== id));
* };
* return (
* <div className="App">
* <h1>Todo List</h1>
* <input
* value={input}
* onChange={(e) => setInput(e.target.value)}
* placeholder="Nueva tarea"
* />
* <button onClick={addTodo}>Agregar</button>
* {todos.length === 0 ? (
* <p>No hay tareas</p>
* ) : (
* todos.map(todo => (
* <TodoItem key={todo.id} todo={todo} deleteTodo={deleteTodo} />
* ))
* )}
* </div>
* );
* }
* export default App;
* **Código de TodoItem.js**:
* import React from 'react';
* function TodoItem({ todo, deleteTodo }) {
* return (
* <div>
* <span>{todo.text}</span>
* <button onClick={() => deleteTodo(todo.id)}>Eliminar</button>
* </div>
* );
* }
* export default TodoItem;
* **Caso Real**: En Trello, los tableros usan estado y eventos para gestionar tareas.
* **Pregunta para Entrevistas**: ¿Cómo optimizarías el rendimiento de una lista dinámica en React?

**Explicación para el Profesor (20 min):**

* Explicar el código línea por línea:
  + useState para todos e input.
  + addTodo usa spread operator (...todos) para no mutar el estado.
  + deleteTodo usa filter para eliminar tareas.
  + Input controlado con value y onChange.
  + Condicional ternario para mostrar "No hay tareas".
* Demostración: Crear la app en CodeSandbox, probar agregar y eliminar tareas.
* Resaltar que en entrevistas, los candidatos deben mencionar el uso de key para optimizar listas y evitar mutaciones directas.
* Conectar con Todo List: Este código es el núcleo del proyecto, listo para el portafolio.

**Diapositiva 10: Práctica Guiada**

**Contenido:**

* **Ejercicio**: Construir la app Todo List con pair programming.
* **Pasos**:
  1. Crear TodoItem.js para mostrar tareas y botón de eliminar.
  2. Implementar App.js con estado, input controlado, y lista dinámica.
  3. Conectar ambos componentes con props.
  4. Probar agregar y eliminar tareas.
* **Errores Comunes**:
  1. Olvidar key en map.
  2. Mutar estado directamente (todos.push()).
  3. No limpiar el input tras agregar.
  4. Olvidar importar useState o TodoItem.
* **Solución**: Usar el código de la Diapositiva 9.
* **Pregunta para Entrevistas**: ¿Cómo manejarías el estado en una app Todo List escalable?

**Explicación para el Profesor (10 min):**

* Explicar que esta práctica integra todos los conceptos: JavaScript (arrays, funciones), JSX, props, estado, eventos.
* Dividir tareas: Javier escribe App.js, Naomy escribe TodoItem.js.
* Supervisar el uso de key, spread operator, y eventos.
* Resolver errores comunes en vivo.

**Práctica (30 min):**

* **Actividad**: Construir la app Todo List.
* **Instrucciones**:
  + Pair programming: Naomy implementa el estado y formulario, Javier el componente TodoItem.
  + Verificar que usen key, spread operator, y eventos correctamente.
  + Resolver errores: Falta de key, mutación directa, importaciones incorrectas.
* **Resultado Esperado**: App funcional para agregar y eliminar tareas.

**Diapositiva 11: Cierre y Tareas**

**Contenido:**

* **Resumen**:
  + JavaScript: Tipos de datos, variables, condicionales, funciones, arrays.
  + React: JSX, componentes funcionales, props, estado, eventos.
  + Todo List: Integra todos los conceptos en una app profesional.
* **Tareas (1-2 horas)**:
  + Completar la app Todo List si no se terminó.
  + Añadir un botón para limpiar todas las tareas (setTodos([])).
  + Leer <https://react.dev/learn/thinking-in-react>.
  + Agregar un condicional para mostrar "Lista completa" si hay más de 5 tareas.
* **Pregunta Reflexiva**: ¿Cómo mejoran los componentes reutilizables la mantenibilidad?
* **Pregunta para Entrevistas**: ¿Cómo explicarías el flujo de datos en una app React?

**Explicación para el Profesor (5 min):**

* Resumir: Los estudiantes están listos para construir la app y avanzar a módulos avanzados.
* Felicitara Javier y Naomy por su progreso.
* Motivar a experimentar con mejoras (estilos, validaciones) para el portafolio.