Session 9 Instructor Guide: Adding Theme Music

Learning Outcomes

By the end of Session 9, students will be able to:

1. **Define custom hooks** as reusable functions that encapsulate component logic and start with “use”
2. **Apply the DRY principle** (Don’t Repeat Yourself) to eliminate code duplication through custom hooks
3. **Distinguish between built-in and custom hooks** and identify common use cases for each type
4. **Explain the HTMLAudioElement interface** and its role in programmatic audio control
5. **Compare refs and state** to understand when each should be used for data storage
6. **Use the useRef hook** to create persistent references that don’t trigger re-renders
7. **Access ref values** using the current property to interact with stored objects
8. **Create custom hooks** that combine multiple React features for complex functionality
9. **Integrate browser APIs** with React hooks for seamless component integration
10. **Implement error handling** in custom hooks to create robust, production-ready code
11. **Use AI assistance** effectively for code generation while maintaining code quality
12. **Build audio controls** that provide intuitive user interfaces for media playback
13. **Apply cleanup patterns** using useEffect to prevent memory leaks and resource conflicts

Instruction

Instructor introduces key concepts students need to succeed:

1. **Custom Hooks Philosophy** - Define custom hooks as React’s solution for logic reuse, emphasizing the DRY principle (Don’t Repeat Yourself) and “write once, use often” approach
2. **Browser Audio APIs** - Introduce HTMLAudioElement as the browser’s built-in audio interface, demonstrating basic audio control methods
3. **Refs vs State Distinction** - Explain when to use refs for non-rendering data versus state for UI-affecting data

4. **useRef Hook Mechanics** - Show how useRef creates persistent storage with the current property pattern, demonstrating common patterns like storing mutable values and accessing DOM elements
5. **Audio Integration Patterns** - Demonstrate how to wrap browser APIs in React hooks for clean component interfaces
6. **AI-Assisted Development** - Introduce GitHub Copilot workflow for complex functionality like error handling and cleanup
7. **Component Integration** - Show how custom hooks integrate with existing components through the useGame pattern
8. **Professional Error Handling** - Emphasize robust code that handles audio loading failures gracefully
9. **Memory Management** - Explain cleanup patterns to prevent audio conflicts and memory leaks
10. **User Experience Design** - Guide students through building intuitive audio controls with visual feedback
11. **Let's Add Music!** - Launch the hands-on mission: build complete audio system with custom hooks and AI assistance

---

Slide Deck Outline

Slide 1: Welcome to Advanced React Patterns! 🎵

- **Title:** "Session 9: Custom Hooks & Browser APIs — Adding Theme Music"
- **Session 8 Recap:** "Last time: You implemented scoring and victory with complex state management and built your first independent component"
- **Hook:** "Your game tracks progress — now let's make it sound professional!"
- **Today's Mission:**
  - **Create** custom hooks for reusable audio functionality
  - **Master** browser APIs with HTMLAudioElement integration
  - **Understand** refs vs state for different data storage needs
  - **Use** AI assistance for complex error handling and cleanup
  - **Build** professional audio controls with visual feedback
- **Visual:** Audio waveform with React hook icons
- **Connection:** "From interactive components to immersive audio experiences!"

Slide 2: Custom Hooks - React’s Logic Reuse System 🪝

- **Title:** “Building Your Own React Features”
- **What Are Custom Hooks?**
  - **Functions starting with “use”** that encapsulate component logic
  - **Embody DRY principle** - “Don’t Repeat Yourself” through reusable logic
  - **“Write once, use often”** - eliminate code duplication across components
  - **Combine built-in hooks** to create complex functionality
  - **Follow React rules** - only call at component top level

Built-in vs Custom Hooks:

Built-in Hooks	Custom Hooks
`useState`, `useEffect`, `useRef`	`useGame`, `useAudio`
Provided by React	Created by you
Basic React features	Complex, app-specific logic
Universal use cases	Tailored to your needs

- **Real-World Examples:**
  - **useLocalStorage** - Persist data in browser storage
  - **useFetch** - Handle API requests with loading states
  - **useTimer** - Manage countdown and interval logic
- **Today’s Hook:** `useAudio` - Complete audio playback control
- **Professional Context:** “Custom hooks are how React developers share complex logic across teams”

Slide 3: HTMLAudioElement - Browser’s Built-in Music Player 🔊

- **Title:** “Programmatic Audio Control in the Browser”
- **What is HTMLAudioElement?**
  - **Browser’s native audio interface** - no external libraries needed

    - **JavaScript API** for controlling audio playback
    - **Rich feature set** - play, pause, volume, looping, and more

Core Audio Methods:

```
const audio = new Audio('/path/to/music.mp3');
audio.play();        // Start playback
audio.pause();       // Stop playback
audio.volume = 0.5;  // Set volume (0-1)
audio.loop = true;   // Enable looping
```

Audio Properties: - **currentTime** - Playback position in seconds - **duration** - Total audio length - **paused** - Boolean playback state - **volume** - Audio level (0.0 to 1.0)

- **Promise-Based Play:** `audio.play()` returns a promise for error handling
- **Event System:** Listen for 'ended', 'error', 'loadstart' events
- **Student Application:** "Your useAudio hook will wrap this API in a clean React interface"

Slide 4: Refs vs State - Choosing the Right Storage 🔗

- **Title:** "When to Use Refs Instead of State"
- **Visual:** Split comparison showing different use cases

State (useState): - **Triggers re-renders** when changed - **For UI-affecting data** - what users see - **Examples:** Current screen, score, quiz progress - **Pattern:** `const [value, setValue] = useState()`

Refs (useRef): - **No re-renders** when changed - **For non-UI data** - behind-the-scenes storage - **Examples:** DOM elements, timers, audio objects - **Pattern:** `const ref = useRef(); ref.current = value`

Audio Storage Decision: - **Audio object** doesn't affect what's rendered - **Playback state** (`isPlaying`) does affect UI - **Solution:** Store audio in ref, playback state in state

- **Key Insight:** "Refs are perfect for browser API objects that need to persist but don't change the UI"
- **Student Preview:** "Your audio element will live in a ref, while isPlaying will be state"

Slide 5: useRef Hook - Persistent Storage Without Re-renders 📍

- **Title:** "Creating References That Survive Component Updates"

- **useRef Mechanics:**
    - **Container for mutable data** - holds values that can change
    - **No re-renders** - updates don't trigger component re-renders
    - **Persistent** - survives component updates
    - **Access via current** - `ref.current` holds the actual value

Common useRef Patterns:

Storing Mutable Values (Audio Use Case):

```
function useAudio(src) {
  const audioRef = useRef(null);

  const play = () => {
    if (!audioRef.current) {
      audioRef.current = new Audio(src);
    }
    audioRef.current.play();
  };
}
```

Accessing DOM Elements:

```
function MyComponent() {
  const inputRef = useRef(null);

  const focusInput = () => {
    inputRef.current.focus();
  };

  return <input ref={inputRef} />;
}
```

- **Memory Management:** Refs persist until component unmounts
- **Professional Usage:** "Essential for integrating with browser APIs and third-party libraries"

Slide 6: Component Integration - Adding Audio Controls 🎛️

- **Title:** "Building User-Friendly Audio Interfaces"
- **MusicToggle Component Design:**

    - **Visual feedback** - Different icons for play/pause states
    - **Accessibility** - Proper alt text and tooltips
    - **State-driven UI** - Appearance changes based on `music.isPlaying`

Conditional Rendering Pattern:

```
<img
  src={music.isPlaying ? "/images/playing.svg" : "/images/paused.svg"}
  alt={music.isPlaying ? "Pause" : "Play"}
/>
```

User Experience Principles: - **Clear visual states** - Users know if music is playing - **Intuitive controls** - Standard play/pause iconography - **Immediate feedback** - UI updates instantly on interaction

- **HUD Integration:** Audio controls fit naturally in game interface
- **Professional Polish:** “Good audio controls feel invisible when working correctly”

Slide 7: AI-Assisted Development - GitHub Copilot Workflow 🤖

- **Title:** “Using AI to Handle Complex Implementation Details”
- **When to Use AI Assistance:**
    - **Complex error handling** - Promise rejection, network failures
    - **Cleanup patterns** - useEffect cleanup functions
    - **Edge cases** - Scenarios you might not think of
    - **Boilerplate code** - Repetitive patterns

Effective AI Prompts: - **Be specific** - “Add error handling to the play function” - **Include context** - “in the useAudio hook” - **Specify behavior** - “catch errors, log warnings, update state”

AI Workflow: 1. **Write clear prompt** describing what you want 2. **Review generated code** - understand what it does 3. **Test functionality** - verify it works as expected 4. **Iterate if needed** - refine prompts for better results

- **Professional Skills:** “AI assistance accelerates development while you maintain code quality control”
- **Student Empowerment:** “You’re learning to collaborate with AI tools like professional developers”

Slide 8: Error Handling - Building Robust Audio Systems 🛡️

- **Title:** “Planning for When Audio Fails to Load”
- **Common Audio Failures:**
  - **File not found** - Invalid audio path
  - **Network issues** - Slow or failed downloads
  - **Format problems** - Unsupported audio formats
  - **Browser restrictions** - Autoplay policies

Promise-Based Error Handling:

```
audio.play()
  .then(() => setIsPlaying(true))
  .catch(error => {
    console.warn("Audio failed to play:", error);
    setIsPlaying(false);
  });
```

Graceful Degradation: - **Log warnings** instead of crashing - **Update UI state** to reflect reality - **Continue game functionality** even without audio

- **Professional Mindset:** “Always assume external resources might fail”
- **User Experience:** “Games should work even when audio doesn’t load”

Slide 9: Memory Management - Cleanup and Resource Management 🧹

- **Title:** “Preventing Memory Leaks with useEffect Cleanup”
- **The Problem:** Audio elements can continue playing after components unmount
- **The Solution:** useEffect cleanup functions

Cleanup Pattern:

```
useEffect(() => {
  return () => {
    if (audioRef.current) {
      audioRef.current.pause();
      audioRef.current = null;
    }
  };
}, []);
```

Why Cleanup Matters: - **Prevents memory leaks** - Audio objects are garbage collected - **Stops background audio** - No music playing after navigation - **Resource management** - Proper browser resource cleanup

- **Professional Practice:** "Always clean up resources in useEffect"
- **Student Application:** "Your audio will stop cleanly when navigating between screens"

Slide 10: Build Professional Audio Controls! 🚀

- **Today's Coding Mission:**
    1. **Add MusicToggle component** - Build audio controls in HUD with conditional rendering
    2. **Implement useRef storage** - Add audio reference to useAudio hook
    3. **Create audio playback** - Implement play function with HTMLAudioElement
    4. **Complete pause functionality** - Add pause method to stop audio playback
    5. **Add AI-assisted error handling** - Use Copilot for robust error management
    6. **Implement cleanup** - Add useEffect cleanup to prevent memory leaks
- **Success Criteria:**
    - Music toggle button appears in HUD
    - Clicking toggle starts/stops theme music
    - Audio controls show correct visual state
    - Error handling prevents crashes on invalid files
- **Professional Workflow:** "Custom hooks + browser APIs + AI assistance = production-quality features"

[HANDS-ON WORK HAPPENS HERE]

Slide 11: Custom Hook Architecture - Professional Patterns 🏗️

- **Title:** “How useAudio Demonstrates Professional Hook Design”
- **Hook Responsibilities:**
  - **State management** - Track playing/paused state
  - **Resource management** - Create and store audio elements
  - **API integration** - Wrap HTMLAudioElement in React interface
  - **Error handling** - Gracefully handle failures
  - **Cleanup** - Prevent memory leaks

Interface Design:

```
const { play, pause, toggle, isPlaying } = useAudio(src);
```

Benefits of This Pattern: - **Simple interface** - Components don’t need to know about audio complexity - **Reusable** - Any component can add audio with one line - **Testable** - Hook logic is isolated and testable - **Maintainable** - Audio logic centralized in one place

- **Professional Context:** “This is how React teams build scalable, maintainable applications”

Slide 12: What’s Next - Deployment and Sharing 🚀

- **Title:** “Preview of Session 10”
- **Today’s Achievement:** “You built custom hooks with browser API integration and AI-assisted development”
- **Next Challenge:** “Deploy your complete game to the internet for others to play”
- **Concepts Coming:**
  - **Git workflow** - Version control and commit practices
  - **GitHub Pages** - Free hosting for React applications
  - **Build process** - Optimizing your app for production
  - **Deployment automation** - CI/CD with GitHub Actions
- **Motivation:** “Your complete trivia game will be live on the internet!”
- **Visual:** Preview of deployed game with public URL