**Practical Hands-On with Real-World Examples**

**1. Using a Screen Reader (NVDA or VoiceOver Demo)**

**Explanation:**  
Screen readers are software applications that convert on-screen content into speech or Braille output, enabling users with visual impairments to interact with digital content. NVDA (NonVisual Desktop Access) is a popular free screen reader for Windows, while VoiceOver is built into macOS and iOS.

**Hands-On Steps:**

* Launch NVDA on Windows or VoiceOver on Mac/iOS.
* Open the web page or app you want to test.
* Use the screen reader commands to navigate through headings, links, buttons, and form fields. For example, NVDA users can press H to jump between headings, or Tab to move through focusable elements. VoiceOver users use the rotor gesture to navigate by elements.
* Listen carefully to how the screen reader announces UI elements. It should announce meaningful labels, roles (e.g., “button”, “checkbox”), and states (e.g., “checked”, “expanded”).

**Real-World Example:**  
In an e-commerce app, testers used NVDA to verify that product images had alt text describing the item (e.g., “Red cotton T-shirt, size M”). The screen reader correctly announced the product name and price when focused. However, the “Add to Cart” button was labeled only as “button,” causing confusion. Developers updated the accessible name to “Add Red T-shirt to cart,” improving clarity for screen reader users.

**Use Case:**  
Screen reader testing is essential to catch missing labels, improper reading order, or unclear control descriptions that automated tools cannot fully identify. It helps ensure the app is usable by visually impaired users.

**2. Keyboard Navigation Testing**

**Explanation:**  
Keyboard navigation testing ensures all interactive elements can be accessed and operated using only a keyboard (or equivalent input device). This benefits users who cannot use a mouse, including people with motor disabilities.

**Hands-On Steps:**

* Disable the mouse and use Tab, Shift + Tab, Enter, Space, arrow keys, and shortcuts to navigate through the app or website.
* Verify that the focus indicator (outline or highlight) is visible and moves logically through interactive controls (links, buttons, inputs).
* Check that all interactive elements are reachable and usable (e.g., dropdowns open with keyboard, modals can be closed).
* Confirm no keyboard traps exist — you should not get stuck in any component.

**Real-World Example:**  
A financial services app was tested for keyboard accessibility. It was found that a modal dialog was keyboard-focusable, but keyboard users could not close it because the close button was not reachable via Tab. The fix involved adding proper tab indexing and keyboard event handlers to close the modal with the Esc key.

**Use Case:**  
Keyboard navigation testing is crucial for users relying on assistive input devices or alternative keyboards. It ensures seamless access and control without a pointing device.

**3. Manual Code Inspection for Accessibility**

**Explanation:**  
Manual inspection involves reviewing the app’s or site’s source code (HTML, ARIA roles, labels, etc.) to verify proper use of accessibility features beyond automated detection.

**Hands-On Steps:**

* Open the code in an IDE or browser developer tools.
* Check for semantic HTML usage: Are headings (<h1>–<h6>), lists (<ul>, <ol>), and landmarks (<nav>, <main>, <footer>) properly used?
* Verify that all interactive elements have descriptive accessible names via aria-label, aria-labelledby, or alt attributes.
* Ensure ARIA roles and properties are used correctly without redundancy or conflicts. For example, avoid applying role="button" to a native <button>.
* Check for appropriate use of tabindex to control focus order.

**Real-World Example:**  
During code review of a healthcare portal, a developer discovered that checkboxes used for consent did not have associated <label> tags, making them inaccessible to screen readers. Adding explicit <label> elements tied to the inputs with for attributes solved the problem and improved compliance with WCAG 1.1.1 (Non-text Content).

**Use Case:**  
Manual code inspection catches subtle semantic and ARIA issues that automated tools miss, ensuring a solid foundation for accessibility.

**4. Using Browser Developer Tools to Inspect Accessibility Tree**

**Explanation:**  
The accessibility tree is an internal representation browsers create to communicate the page structure and elements to assistive technologies. Browser DevTools can expose this tree, allowing developers to see how accessible the page actually is.

**Hands-On Steps:**

* Open Chrome DevTools or Firefox Developer Tools.
* Go to the Elements panel, right-click on an element, and choose “Inspect Accessibility Properties” or open the Accessibility pane.
* Review the accessible name, role, states (checked, expanded), and properties assigned to the element.
* Check the tab order and focus behavior as reflected in the accessibility tree.
* Use the tool to identify missing roles or incorrect labels.

**Real-World Example:**  
A news website was tested using Chrome DevTools Accessibility pane. Developers found that some important navigation landmarks were missing, making it difficult for screen readers to quickly jump to main content. Adding <nav> and <main> landmarks improved the structure and allowed better screen reader navigation.

**Use Case:**  
Accessibility tree inspection helps developers debug and fix how browsers expose content to assistive technology, ensuring that the semantic meaning and navigational landmarks are correctly understood.

**Summary Table**

| **Technique** | **Purpose** | **Key Benefits** | **Example Outcome** |
| --- | --- | --- | --- |
| Screen Reader Testing | Verify spoken output & labeling | Real user experience with visual impairment | Improved button labels for clarity |
| Keyboard Navigation | Ensure all controls accessible via keyboard | Inclusive navigation for motor-impaired users | Modal dialog keyboard closable |
| Manual Code Inspection | Semantic and ARIA correctness | Catch subtle accessibility bugs | Proper label linking on checkboxes |
| Accessibility Tree Inspection | Verify browser's accessibility exposure | Debugging of roles, names, and states | Added missing navigation landmarks |