# JavaScript: DOM and Events

Lecture 26

# Objects are Everywhere

☐ Global variables in JavaScript are a lie

☐ Implicitly part of some "global object", provided by execution environment

  ■ See Developer Tools: Console
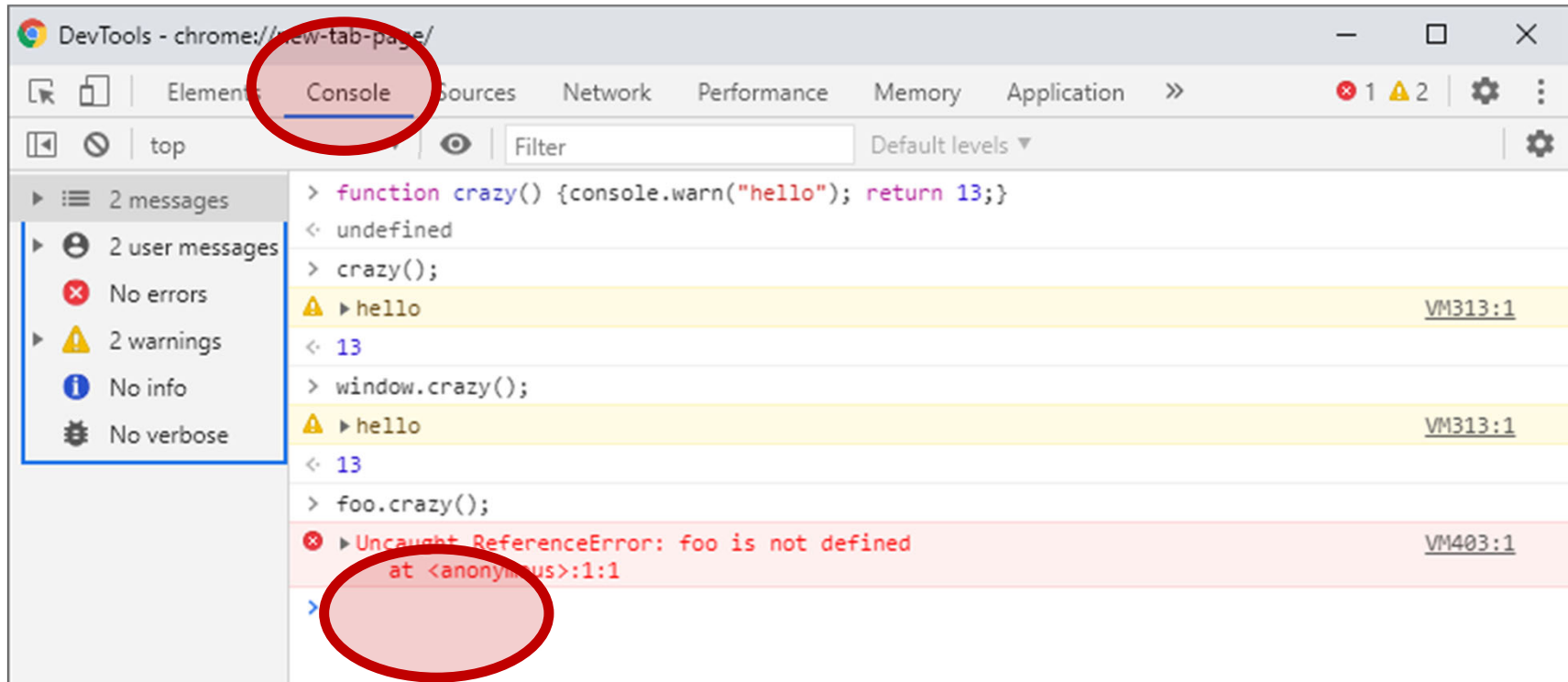
# Window Object

- For JavaScript running in a browser, implicit global object is the window
  - `>> this`
  - `<- Window`
- Many properties, including
  - `location` (url of displayed document)
  - `history`
  - `innerHeight, innerWidth`
  - `sessionStorage`
  - `alert(), prompt()`
  - `document` (tree of displayed document)
- For JavaScript in a different environment (eg node.js), the global object is different

# Document is a Tree
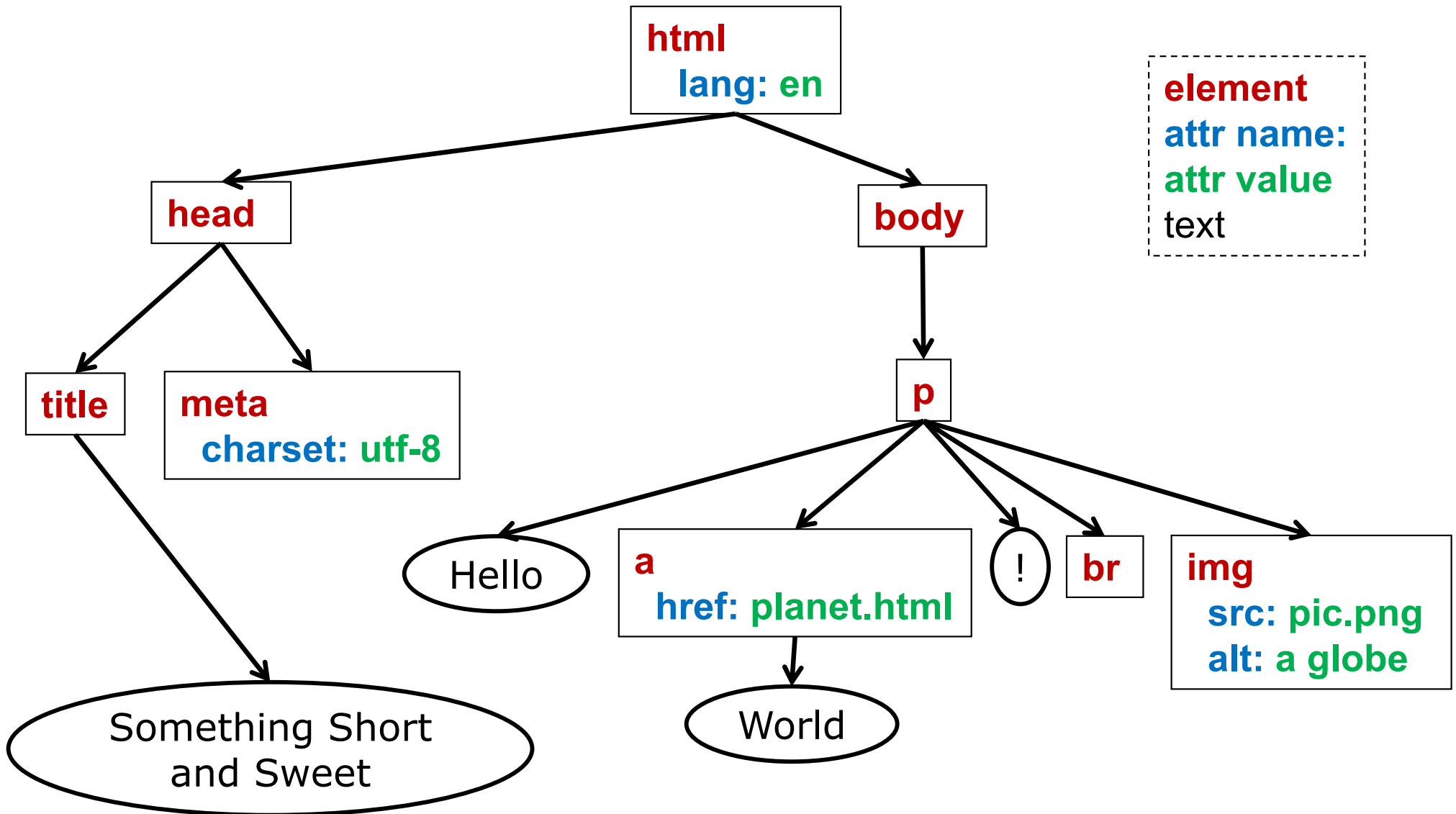
# DOM: "Document Object Model"

- ☐ DOM is a language-neutral API for working with HTML (and XML) documents
  - ■ Different programming languages have different bindings to this API
  - ■ But all are similar to JavaScript's API
- ☐ In JavaScript, tree nodes → objects
  - ■ A tree node (an HTML element, or text node)

    **`<input type="text" name="address">`**
  - ■ A JavaScript object with many properties

    **`{ tagName: "INPUT",`**

    **`type: "text",`**

    **`name: "address",`** *`/* lots more… */ `* **`}`**

# DOM History

- ☐ Ad hoc DOM existed from the beginning of JavaScript
  - ■ Core purpose of client-side execution: Enable user interaction with the document
  - ■ Need a connection between programming language (JavaScript) and the document
- ☐ DOM 1 specification (W3C) in '98
  - ■ Standardized mapping tree→objects and functions for *modifying* the tree
- ☐ DOM 2 ('00): added styles and event handling
- ☐ DOM 3 ('04): fancier tree traversal & indexing schemes
- ☐ DOM "4" ('15…):
  - ■ Actually just a "living document"
  - ■ Some non-backwards-compatible changes

# Accessing the DOM Tree

- ☐ The **document** property of **window**
- ☐ For accessing DOM
  - ■ **links**: all links in document
  - ■ **images**: all images in document
  - ■ **forms**: array of forms in a page
    - ☐ **elements[]**: array of widgets in a form
  - ■ **anchors**: all anchors in document
  - ■ **querySelectorAll(string)**: find matching nodes
  - ■ **getElementById(string)**: find a node
  - ■ **querySelector(string)**: find a node
- ☐ For mutating the DOM
  - ■ **write()**: outputs text to document body
    - ☐ Don't do this :-)
  - ■ **createElement(string)**: make new node
    - ☐ Then use Node's methods to insert into DOM

# Document is a Tree
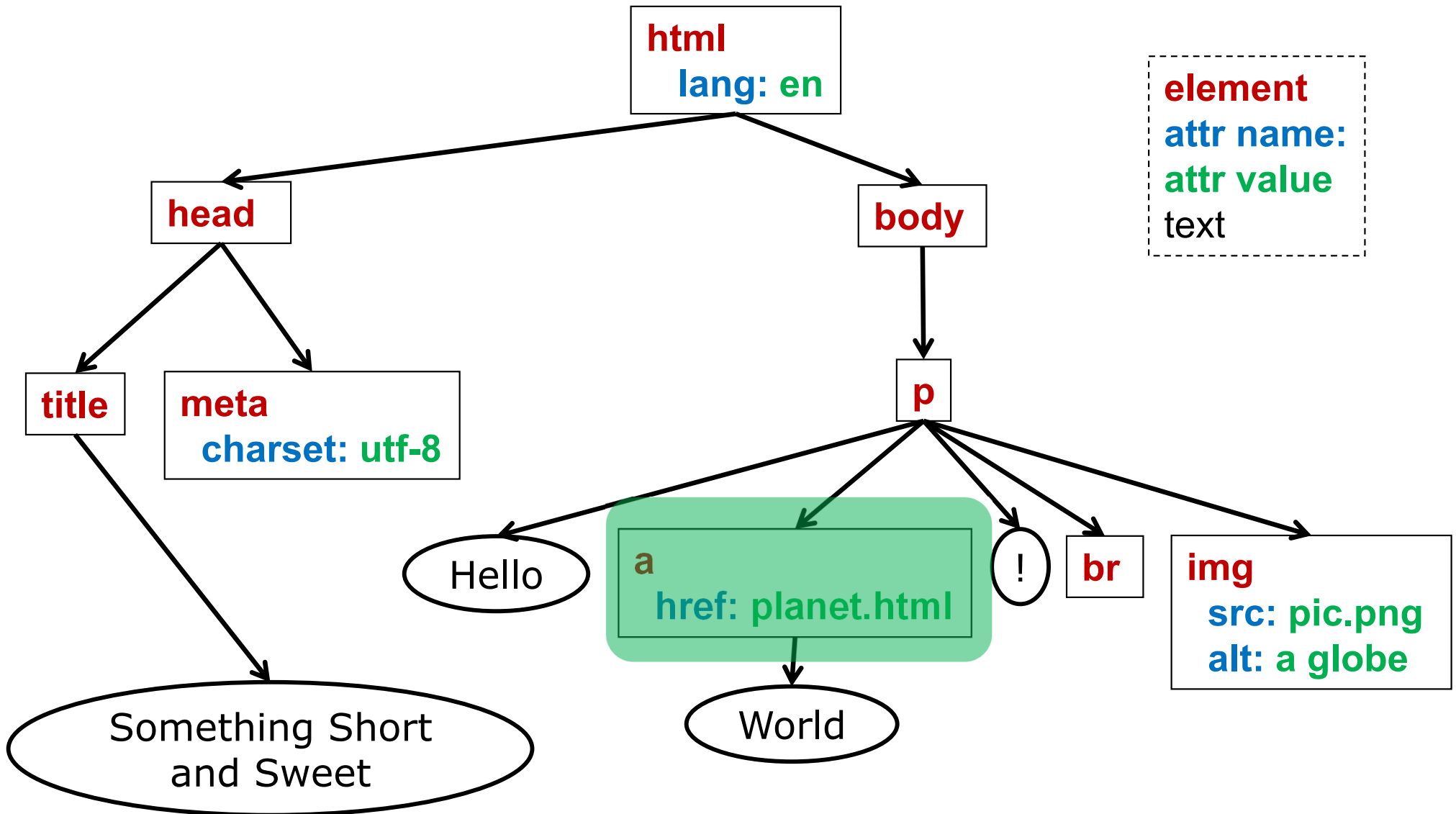
# Node is a JavaScript Object

- ☐ Properties
  - ■ **parentNode, childNodes, firstChild, lastChild, nextSibling, previousSibling**
  - ■ **innerHTML**
  - ■ **tagName**
    - ☐ HTML upper case (A), XML lower case (a)
  - ■ **attributes, name, id, class**
  - ■ **style**
    - ☐ Hyphenated property in CSS ("font-size") becomes camelCase in JavaScript ("fontSize")
- ☐ Methods
  - ■ **hasAttribute(attr), removeAttribute(attr), getAttribute(attr), setAttribute(attr)**
  - ■ **appendChild(node), removeChild(node), insertBefore(node)**
  - ■ **insertAdjacentHTML(position, html)**

# Demo: Web Console (Reading)

```
> let b = document.body;
> b.tagName;      // 'BODY'
> b.childNodes; // a NodeList
> for (let n of b.childNodes) {
     console.info(n.nodeName)
  }
> b.style; // inspect css properties


> let x = document.getElementById("footer");
> x.innerHTML;
> x.childNodes;
```

# Demo: Web Console (Writing)

```
> let b = document.body;
> b.style.backgroundColor;
> b.style.backgroundColor = "green";

> let x = document.getElementById("footer");

// bad
> x.innerHTML;
> x.innerHTML = "<h2>Hello</h2>";

//good
> let h = document.createElement("h2");
> h.className = "demo";
> h.textContent = "World";
> x.appendChild(h);
```

# How to Find a Node in Tree

1. Hard coding with "flat" techniques
   - Array of children
     ```
     document.forms[0].elements[0]
     ```
   - Downside: too brittle
   - If the document structure changes a little, everything breaks
2. Using an element's *name attribute*
   - In HTML:
     ```
     <form name="address"> …
     <input name="zip"... />  </form>
     ```
   - In JavaScript:
     ```
     document.address.zip
     ```
   - Downside: direct path still hard coded

# How to Find a Node in Tree

3. Using an element's *id attribute*
    - In HTML

        ```
        <td id="shipping">...</td>
        ```
    - In JavaScript

        ```
        document.getElementById("shipping")
        ```
    - Downside: element must have (unique) ID

4. Using a *CSS selector*
    - Can find one match or all matches

        ```
        document.querySelector("#shipping");
        document.querySelectorAll(".nav li");
        ```
    - Can search whole tree or subtree

        ```
        elt.querySelector("table"); // from elt
        ```

# Example

```html
<form id="wheels">
  <input type="checkbox" name="vehicles"
         value="car" /> Car
  <input type="checkbox" name="vehicles"
         value="truck" /> Truck
  <input type="checkbox" name="vehicles"
         value="bike" /> Bike
</form>
```

```javascript
let numChecked = 0;
let elt = document.getElementById("wheels");
for (const box of elt.vehicles) {
  if (box.checked)
    numChecked++;
}
```

# Interactive Documents

☐ To make a document interactive, you need:

- Widgets (ie HTML elements)
  - ☐ Buttons, windows, menus, etc.
- Events
  - ☐ Mouse clicked, window closed, button clicked, etc.
- Event listeners
  - ☐ Listen (ie wait) for events to be triggered, and then perform actions to handle them

# Events Drive the Flow of Control

☐ This style is *event driven* programming

☐ Event handling occurs as a loop:

- Program is idle
- User performs an action
  - ☐ Eg moves the mouse, clicks a button, types in a text box, selects an item from menu, …
- This action generates an event (object)
- That event is sent to the program, which responds
  - ☐ Code executes, could update document
- Program returns to being idle

# Handling Events Mechanism

- ☐ Three parts of the event-handling mechanism
  - ■ *Event source*: the widget with which the user interacts
  - ■ *Event object*: encapsulated information about the occurred event
  - ■ *Event listener*: a function that is called when an event occurs, and responds to the event

event object

HTML Element ┄┄┄┄┄┄┄┄> aHandler()
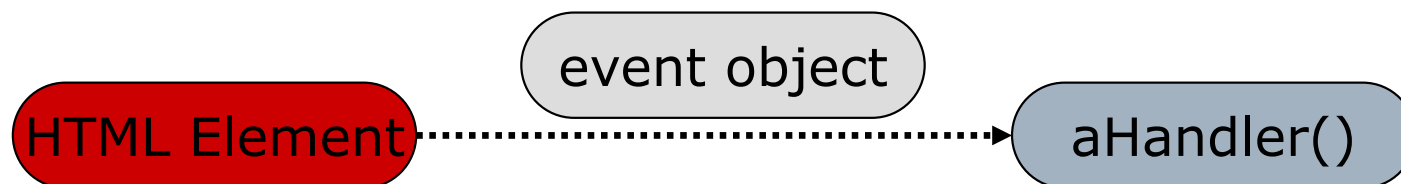
# Programmer Tasks

- ☐ Define an event handler
  - ■ Any function can be an event handler
  - ■ Often need information about the triggering event in order to know what response is needed
- ☐ Register handler with source element
- ☐ Detect event and invoke handler
  - ■ Ha!  Just kidding, you do NOT do this

# Simple Example: Color Swaps

```
<p>This page illustrates changing colors</p>
<form>
  <p>
    <label> background:
      <input type="text" name="back" size="10"
        onchange="foo('bg', this.value)" />
    </label> <br />
    <label> foreground:
      <input type="text" name="fore" size="10"
        onchange="foo('fg', this.value)" />
    </label>
  </p>
</form>
```

# Color Swaps (JavaScript)

```javascript
function foo(place, color) {
   if (place === "bg")
      document.body.style.backgroundColor =
          color;
   else
      document.body.style.color = color;
}
```

# Event Propagation

- ☐ Elements are nested in tree
- ☐ When an event occurs, which element's handler(s) is(are) notified?
- ☐ First, *propagation path* is calculated: from root to smallest element
- ☐ Then event dispatch occurs in 3 phases
  1. Capture (going *down* the path)
  2. Target (smallest element)
  3. Bubble (going *up* the path, reverse of 1)

# http://www.w3.org/TR/DOM-Level-3-Events/
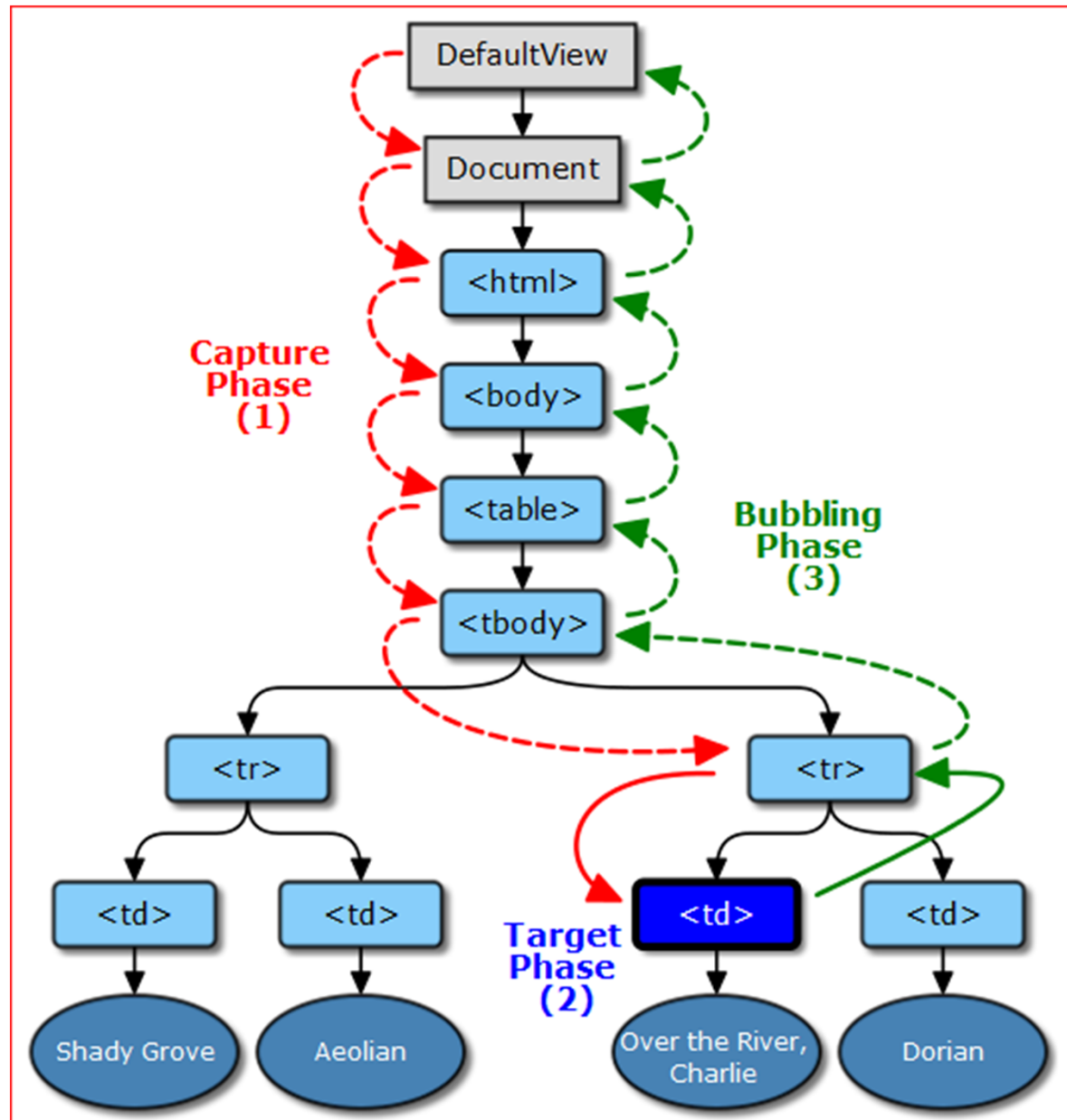
# Bubbling Up

- ☐ Handling is usually done in phase 2 and 3

- ☐ Example: mouse click on hyperlink
  - ■ Handler for `<a>` element displays a pop-up ("Are you sure you want to leave?")
  - ■ Once that is dismissed, event flows up to enclosing `<p>` element, then `<div>` then… *etc.* until it arrives at root element of DOM
  - ■ This root element (*i.e.* `window`) has a handler that loads the new document in the current window

# Programmer Tasks

- ☐ Define a handler
  - ■ Easy, any function will do
- ☐ Register handler
  - ■ Multiple ways to link (HTML) tree elements with (JavaScript) functions
- ☐ Be triggered by the event
  - ■ Ha!  Still kidding
- ☐ Get information about triggering event
  - ■ Multiple (incompatible) ways for handler to get the event object

# Registering an Event Handler

- ☐ Three techniques, ordered from:
  - ■ Oldest (most brittle, most universal) to
  - ■ Newest (most general, least standard)

1. Inline (link in HTML itself)

```
<a href="page.html" onclick="foo()">…
```

2. Direct property (link in JavaScript)

```
let e = … // find source element in tree

e.onclick = foo;
```

3. Chained (In JavaScript, browser differences)

```
let e = … // find source element in tree

e.addEventListener("click", foo, false);
```

# Inline Registration (pre DOM)

- ☐ Use HTML *attributes* (vary by element type)
    - ■ For window: `onload`, `onresize`, `onunload`,…
    - ■ Forms & elements: `onchange`, `onblur`, `onfocus`, `onsubmit`,…
    - ■ Mouse events: `onclick`, `onmouseover`, `onmouseout`,…
    - ■ Keyboard events: `onkeypress`, `onkeyup`,…
- ☐ The *value* of these attributes is JavaScript code to be executed
    - ■ Normally just a function invocation
- ☐ Example
    
    `<a href="page.html" onclick="foo()">`…
- ☐ Advantage: Quick, easy, universal
- ☐ Disadvantage: mixes code with content

# Direct Registration (DOM 1)

- ☐ Use *properties* of DOM element objects
  - ■ **onchange, onblur, onfocus,**…
  - ■ **onclick, onmouseover, onmouseout,**…
  - ■ **onkeypress, onkeyup,**…
- ☐ Set this property to appropriate handler

```
let e = … // find source element in tree
e.onclick = foo;
```

- ☐ Note: no parentheses!

```
e.onclick() = foo; // what does this do?
e.onclick = foo(); // what does this do?
```

- ☐ Disadvantage? No arguments to handler
  - ■ Not a problem, handler gets event object
- ☐ Real disadvantage: 1 handler/element

# Example

```javascript
let divs =
        document.querySelectorAll("div");
for (let d of divs) {
  d.onmouseover = function() {
    this.style.backgroundColor = "red"
  }
  d.onmouseout = function() {
    this.style.backgroundColor = "blue"
  } // *this* will be the element (div)
    // that listener is registered with
}
```

# Chained Registration (DOM 2)

- ☐ Each element has a *collection* of handlers
- ☐ Add/remove handler to this collection

```
let e = … // find source element in tree

e.addEventListener("click", foo);
```

- ☐ First parameter: event name
  - ■ Note: no "on" in event names, just "`click`"
- ☐ Second parameter: handler function
  - ■ This function takes an argument: event
- ☐ Third parameter: handling phase
  - ■ Default is `false` (target or bubbling phase)
  - ■ For capture phase (unusual) use `true`

# Example

```
let divs =
      document.querySelectorAll("div");
for (let d of divs) {
  d.addEventListener ("click",
    function(event) {
      this.act = this.act || false;
      this.act = !this.act;
      this.style.backgroundColor =
        (this.act ? "red" : "gray");
    });
}
```

# Broken: Wrong this with =>

```javascript
let divs =
      document.querySelectorAll("div");
for (let d of divs) {
  d.addEventListener ("click",
    (event) => { // wrong this
      this.act = this.act || false;
      this.act = !this.act;
      this.style.backgroundColor =
        (this.act ? "red" : "gray");
    });
}
```

# Alternative: Use event Argument

```
let divs =
      document.querySelectorAll("div");
for (let d of divs) {
  d.addEventListener ("click",
    (event) => { // use param, not this
      let t = event.currentTarget;
      t.act = t.act || false;
      t.act = !t.act;
      t.style.backgroundColor =
        (t.act ? "red" : "gray");
    });
}
```

# Summary

- ☐ DOM: Document Object Model
  - ■ Programmatic way to use document tree
  - ■ Get, create, delete, and modify nodes
- ☐ Event-driven programming
  - ■ Source: element in HTML (a node in DOM)
  - ■ Handler: JavaScript function
  - ■ Registration: in-line, direct, chained
  - ■ Event is available to handler for inspection