

Fundamentals of Web Development

Third Edition by Randy Connolly and Ricardo Hoar



Chapter 9

JavaScript 2:

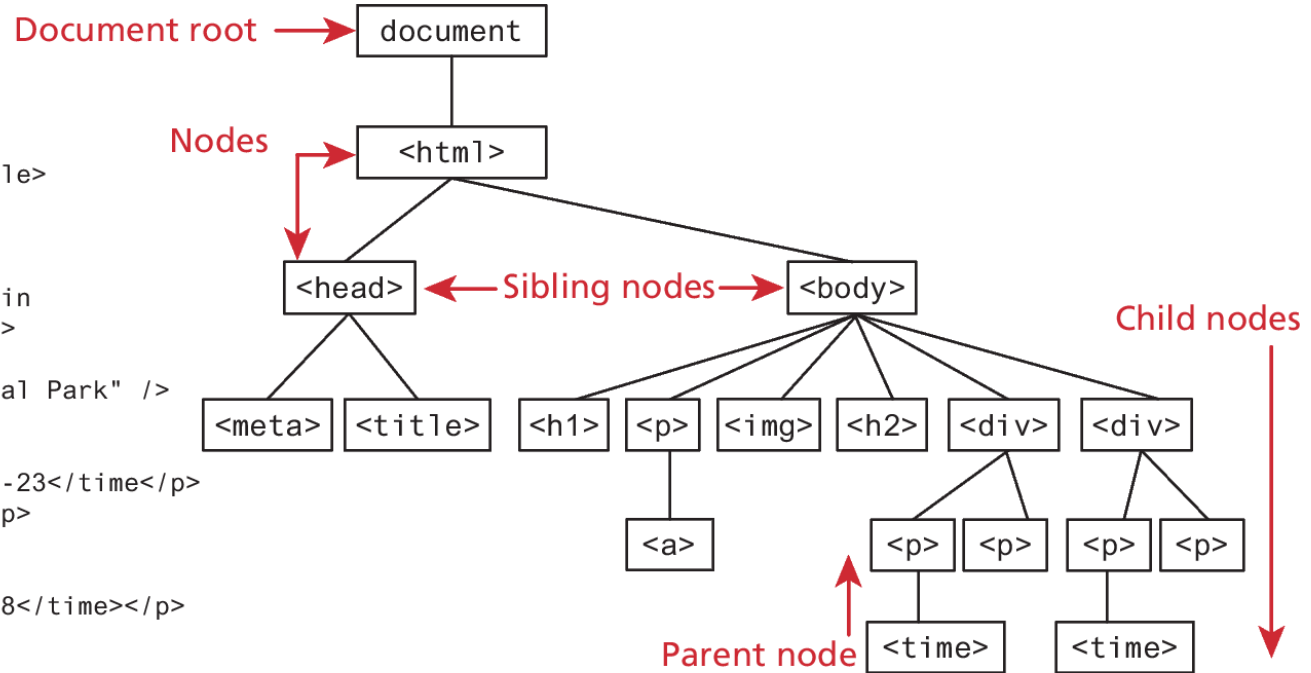
Using JavaScript

In this chapter you will learn . . .

- What is Document Object Model (DOM)
- How to use the DOM to dynamically manipulate the contents of a web page
- How to use the DOM and event handling to validate user input in a form
- What are regular expressions and how to use them in JavaScript.

The Document Object Model (DOM)

```
<html>
<head>
  <meta charset="utf-8">
  <title>Share Your Travels</title>
</head>
<body>
  <h1>Share Your Travels</h1>
  <p>Photo of Conservatory Pond in
    <a href="#">Central Park</a>
  </p>
  
  <h2>Reviews</h2>
  <div id="latestComment">
    <p>Ricardo on <time>2021-05-23</time></p>
    <p>Easy on the HDR buddy.</p>
  </div>
  <div>
    <p>Susan on <time>2021-11-18</time></p>
    <p>I love Central Park.</p>
  </div>
</body>
</html>
```



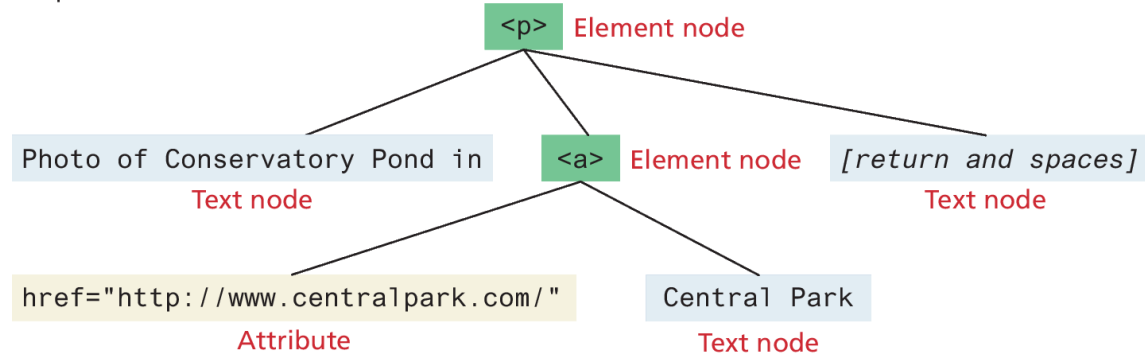
DOM Nodes and NodeLists

In the DOM, each element within the HTML document is called a **node**.

The DOM also defines a specialized object called a **NodeList** that represents a collection of nodes. It operates very similarly to an array.

Many programming tasks that we typically perform in JavaScript involve finding one or more nodes and then modifying them.

```
<p>Photo of Conservatory Pond in  
  <a href="http://www.centralpark.com/">Central Park</a>  
</p>
```



Some Essential Node Object Properties

- **childNodes** A NodeList of child nodes for this node
- **firstChild** First child node of this node
- **lastChild** Last child of this node
- **nextSibling** Next sibling node for this node
- **nodeName** Name of the node
- **nodeType** Type of the node
- **nodeValue** Value of the node
- **parentNode** Parent node for this node
- **previousSibling** Previous sibling node for this node
- **textContent** Represents the text content (stripped of any tags) of the node

Document Object

The **DOM document object** is the root JavaScript object representing the entire HTML document. It is globally accessible via the **document** object reference.

The properties of a document cover information about the page. Some are read-only, but others are modifiable. Like any JavaScript object, you can access its properties using either dot notation or square bracket notation

// retrieve the URL of the current page

let a = **document.URL**;

// retrieve the page encoding, for example ISO-8859-1

let b = **document["inputEncoding"]**;

Document Methods

In addition to these properties, there are several essential methods you will use all the time (We used **document.write()** last chapter). These methods fall into three categories

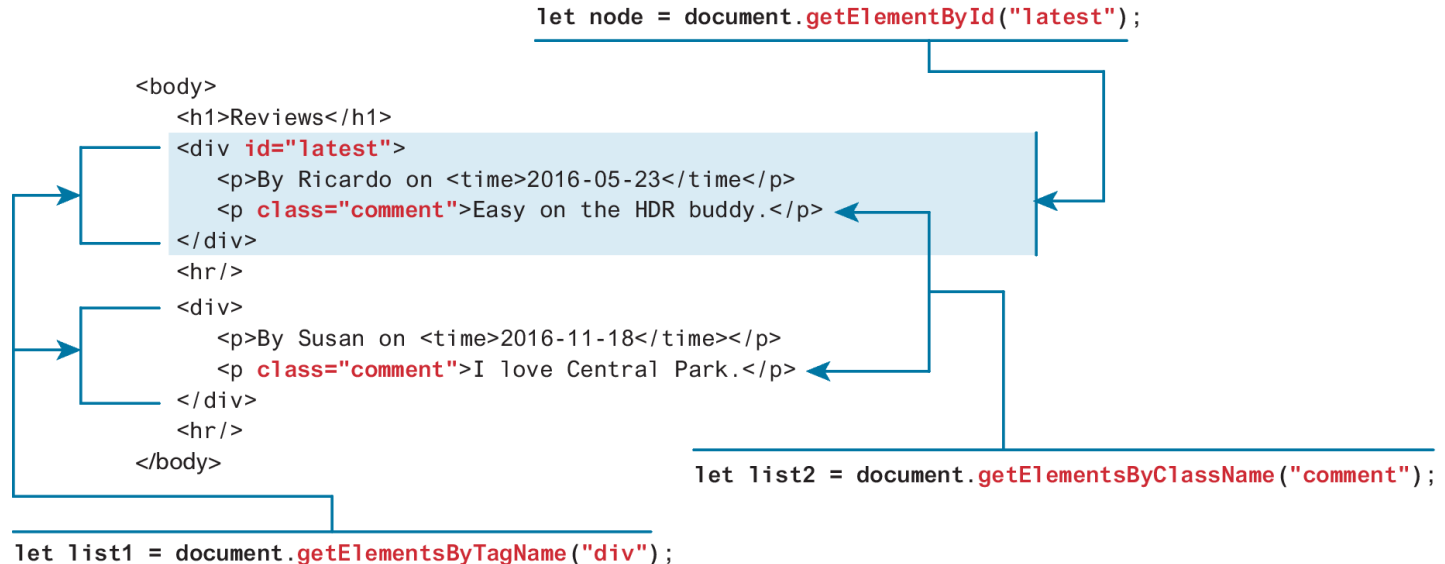
- Selection methods
- Family manipulation methods
- Event methods

Selection Methods

The most important DOM methods

They allow you to select one or more document elements. The oldest 3 are:

`getElementById("id")`, **`getElementsByClassName("name")`** and **`getElementsByTagName("name")`**



Query Selection Methods

The newer
querySelector()
and
querySelectorAll()
methods allow you
to query for DOM
elements much the
same way you
specify CSS styles

`querySelectorAll("nav ul a:link")`

`querySelectorAll("#main div time")`

`querySelector("#main>time")`

`querySelector("footer")`

```
<body>
  <nav>
    <ul>
      <li><a href="#">Canada</a></li>
      <li><a href="#">Germany</a></li>
      <li><a href="#">United States</a></li>
    </ul>
  </nav>
  <div id="main">
    Comments as of
    <time>November 15, 2012</time>
    <div>
      <p>By Ricardo on <time>September 15, 2012</time></p>
      <p>Easy on the HDR buddy.</p>
    </div>
    <div>
      <p>By Susan on <time>October 1, 2012</time></p>
      <p>I love Central Park.</p>
    </div>
  </div>
  <footer>
    <ul>
      <li><a href="#">Home</a> | </li>
      <li><a href="#">Browse</a> | </li>
    </ul>
  </footer>
</body>
```

Element Node Object

Element Node object represents an HTML element in the hierarchy, contained between the opening `<>` and closing `</>`.

An element can itself contain more elements

Every element node has the node properties shown in Table 9.1 (slide 5)

It also has a variety of additional properties, the most important of which are shown in Table 9.3 (next slide)

Some Essential Element Node Properties

- **classList** A read-only list of CSS classes assigned to this element. This list has a variety of helper methods for manipulating this list.
- **className** The current value for the class attribute of this HTML element.
- **id** The current value for the id of this element.
- **innerHTML** Represents all the content (text and tags) of the element.
- **style** The style attribute of an element. This returns a CSSStyleDeclaration object that contains sub-properties that correspond to the various CSS properties.
- **tagName** The tag name for the element.

Extra Properties for Certain Tag Types

Property	Description	Tags
href	Used in <a> tags to specify the linking URL.	a
name	Used to identify a tag. Unlike id which is available to all tags, name is limited to certain form-related tags.	a, input, textarea, form
src	Links to an external URL that should be loaded into the page (as opposed to href which is a link to follow when clicked).	img, input, iframe, script
value	Provides access to the value attribute of input tags. Typically used to access the user's input into a form field.	input, textarea, submit

TABLE 9.4 Some Specific HTML DOM Element Properties for Certain Tag Types

Accessing elements and their properties

```
<p id="here">hello <span>there</span></p>
<ul>
  <li>France</li>
  <li>Spain</li>
  <li>Thailand</li>
</ul>
<div id="main">
  <a href="somewhere.html">
    
  </a>
</div>

<script>

const node = document.getElementById("here");
console.log(node.innerHTML); // hello <span>there</span>
console.log(node.textContent); // "hello there"

const items = document.getElementsByTagName("li");
```

```
for (let i=0; i<items.length; i++) {
  // outputs: France, then Spain, then Thailand
  console.log(items[i].textContent);
}

const link = document.querySelector("#main a");

console.log(link.href); // outputs: somewhere.html

const img = document.querySelector("#main img");
console.log(img.src); // outputs: whatever.gif

console.log(img.className); // outputs: thumb

</script>
```

Modifying the DOM

Now that you can access some of the node and element properties you might be wondering how one can make use of some of these properties. Since most of the properties listed in the previous tables are all read and write, this means that they can be programmatically changed.

- Changing an Element's Style
- Changing the content of any given element
- DOM Manipulation Methods

Changing an Element's Style

To programmatically modify the styles associated with a particular element one must change the properties of the style property for that element

For instance, to change an element's background color and add a three pixel border, we could use the following code:






```
const node = document.getElementById("someId");  
node.style.backgroundColor = "#FFFF00";  
node.style.borderWidth = "3px";
```

How CSS styles can be programmatically manipulated in JavaScript

While you can directly change CSS style elements via this **style** property, it is generally preferable to change the appearance of an element instead using the **className** or **classList** properties

```
<style>
  .box {
    margin: 2em; padding: 0;
    border: solid 1pt black;
  }
  .yellowish { background-color: #EFE63F; }
  .hide { display: none; }
</style>
<main>
  <div class="box">
    ...
  </div>
</main>
```

```
var node = document.querySelector("main div");
```

- 1 `node.className = "yellowish";`  This replaces the existing class specification with this one. Thus the `<div>` no longer has the `box` class
- 2 `node.classList.remove("yellowish");`  `node.classList.add("box");` Removes the specified class specification and adds the `box` class
- 3 `node.classList.add("yellowish");`  Adds a new class to the existing class specification
- 4 `node.classList.toggle("hide");`  If it isn't in the class specification, then add it
- 5 `node.classList.toggle("hide");`  If it is in the class specification, then remove it

Equivalent to:

- 1 `<div class="yellowish">`
- 2 `<div class="">`
`<div class="box">`
- 3 `<div class="box yellowish">`
- 4 `<div class="box yellowish hide">`
- 5 `<div class="box yellowish">`

InnerHTML vs textContent vs DOM Manipulation

Listing 9.1 (slide 13) illustrated how you can programmatically access the content of an element node through its innerHTML or textContent property. These properties can also be used to modify the content of any given element.

For instance, you could change the content of the <div> in Listing 9.1 using the following:

```
const div = document.querySelector("#main");  
div.innerHTML = '<a href="#"></a>';
```

This replaces the existing content with the new content.

InnerHTML vs textContent vs DOM Manipulation (ii)

Using **innerHTML** is generally discouraged (even though you will likely see many examples online that use these approaches) because they are potentially vulnerable to Cross-Site Scripting (XSS) attacks

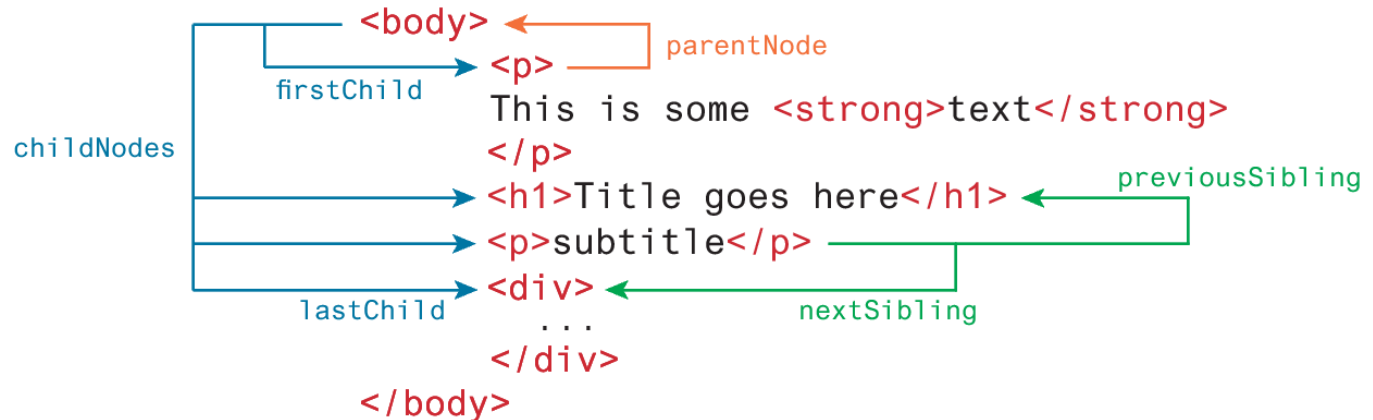
In practice, when you need to change the inner text of an element, it is preferable to use the **textContent** property instead of **innerHTML** since any markup is stripped from it.

In addition, when you need to generate HTML elements, it is better to use the appropriate DOM manipulation methods covered in the next section

DOM family relations

Each node in the DOM has a variety of “family relations” properties and methods for navigating between elements and for adding or removing elements from the document hierarchy.

Child and sibling properties can be an unreliable mechanism for selecting nodes and thus, in general, you will instead use selector methods



DOM Manipulation Methods

- **appendChild** Adds a new child node to the end of the current node.
- **createAttribute** Creates a new attribute node.
- **createElement** Creates an HTML element node.
- **createTextNode** Creates a text node.
- **insertAdjacentElement** Inserts a new child node at one of four positions relative to the current node.
- **insertAdjacentText** Inserts a new text node at one of four positions relative to the current node.
- **insertBefore** Inserts a new child node before a reference node in the current node.
- **removeChild** Removes a child from the current node.
- **replaceChild** Replaces a child node with a different child.

Visualizing the DOM modification

```
<div id="first">  
  <h1>DOM Example</h1>  
  <p>Existing element</p>  
</div>
```

Visualizing the DOM elements

```
<div>  
  <h1> "DOM Example" </h1>  
  <p> "Existing element" </p>  
</div>
```

- 1 Create a new text node

"this is dynamic"

```
const text = document.createTextNode("this is dynamic");
```

- 2 Create a new empty <p> element

```
const p = document.createElement("p");
```

<p></p>

Visualizing the DOM modification (ii)

- 3 Add the text node to new <p> element

```
p.appendChild(text);
```

```
<p> "this is dynamic" </p>
```

- 4 Add the <p> element to the <div>

```
const first = document.getElementById("first");  
first.appendChild(p);
```

```
<div id="first">  
  <h1>DOM Example</h1>  
  <p>Existing element</p>  
  <p>this is dynamic</p>  
</div>
```

```
<div>  
  <h1> "DOM Example" </h1>  
  <p> "Existing element" </p>  
  <p> "this is dynamic" </p>  
</div>
```

DOM Timing

Before finishing this section on using the DOM, it should be emphasized that the timing of any DOM code is very important.

You cannot access or modify the DOM until it has been loaded.

If the DOM programming is written *after* the markup as in Listing 9.2 that *should* ensure that the elements exist in the DOM before the code executes.

To wait until we know for sure that the DOM has been loaded requires knowledge from our next section on **event handling**.

JavaScript Event Handling



Implementing an Event Handler

An event handler is first defined, then registered to an element node object.

Registering an event handler requires passing a callback function to the **addEventListener()**

```
<input type="submit" id="btn">
```

```
<script>
```

```
function simpleHandler() {  
    alert("button was clicked");  
}
```

```
const btn = document.querySelector("#btn");
```

```
btn.addEventListener("click", simpleHandler);
```

```
</script>
```

1 Event handler defined

2 Event handler registered

Listening with an anonymous function

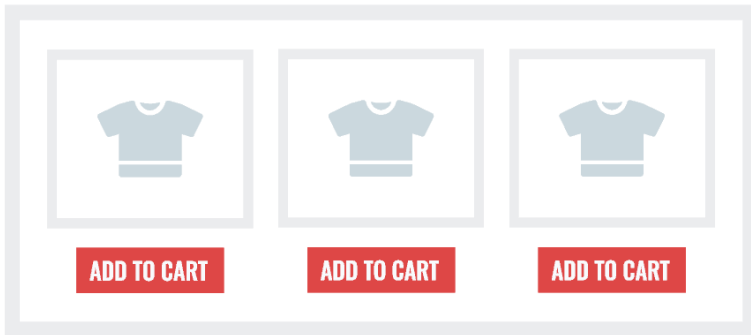
It is much more common to make use of an *anonymous function* passed to **addEventListener()**

```
const btn = document.getElementById("btn");  
  btn.addEventListener("click", function () {  
    alert("used an anonymous function");  
  });
```

```
document.querySelector("#btn").addEventListener("click", function (){  
  alert("a different approach but same result");  
});
```

```
document.querySelector("#btn").addEventListener("click", () => {  
  alert("arrow syntax but same result");  
});
```

Event handling with NodeList arrays



```
<ul id="list">
  <li>
    
    <button>Add To Cart</button>
  </li>
  <li>
    
    <button>Add To Cart</button>
  </li>
  <li>
    
    <button>Add To Cart</button>
  </li>
</ul>
```

```
// select all the buttons
const btns = document.querySelectorAll("#list button");

// this won't work and will generate error
btns.addEventListener("click", function () { ... });
```

```
// instead must loop through node list ...
for (let bt of btns) {
  // ...and assign event listener to each node
  bt.addEventListener("click", function () { ... });
}
```

Remember that a node list (i.e., array of nodes) doesn't support event listeners. Only individual node objects have the `addEventListener()` method defined.

Page Loading and the DOM

To ensure your DOM manipulation code *after* the page is loaded use one of the following two different page load events.

- **window.load** Fires when the entire page is loaded. This includes images and stylesheets, so on a slow connection or a page with a lot of images, the load event can take a long time to fire.
- **document.DOMContentLoaded** Fires when the HTML document has been completely downloaded and parsed. Generally, this is the event you want to use.

Using one of these, your DOM coding can now appear anywhere, including within the **<head>** element, which is the conventional place to add in your JavaScript code.

Wrapping DOM code within a DOMContentLoaded event handler

```
document.addEventListener('DOMContentLoaded', function() {  
  const menu = document.querySelectorAll("#menu li");  
  for (let item of menu) {  
    item.addEventListener("click", function () {  
      item.classList.toggle('shadow');  
    });  
  }  
  const heading = document.querySelector("h3");  
  heading.addEventListener('click', function() {  
    heading.classList.toggle('shadow');  
  });  
});
```

LISTING 9.4 Wrapping DOM code within a DOMContentLoaded event handler

Event Object

- When an event is triggered, the browser will construct an **event object** that contains information about the event.
- Your event handlers can access this event object simply by including it as an argument to the callback function (by convention, this event object parameter is often named **e**)

Event Object Example



```
<ul id="menu">
  <li>Home</li>
  <li>About</li>
  <li>Products</li>
  <li>Contact</li>
</ul>
```

```
const menu = document.querySelectorAll("#menu li");
for (let item of menu) {
  item.addEventListener("click", menuHandler );
}
```

```
function menuHandler(e) {
  const x = e.clientX;
  const y = e.clientY;
  displayArrow(x,y);
  e.target.classList.toggle("selected");
  performMenuAction(e.target.innerHTML);
}
```

By receiving the event object as a parameter and using it to reference the clicked item, the `menuHandler()` function will work no matter where it is located.

Click events include the on-screen pixel location of the mouse cursor.

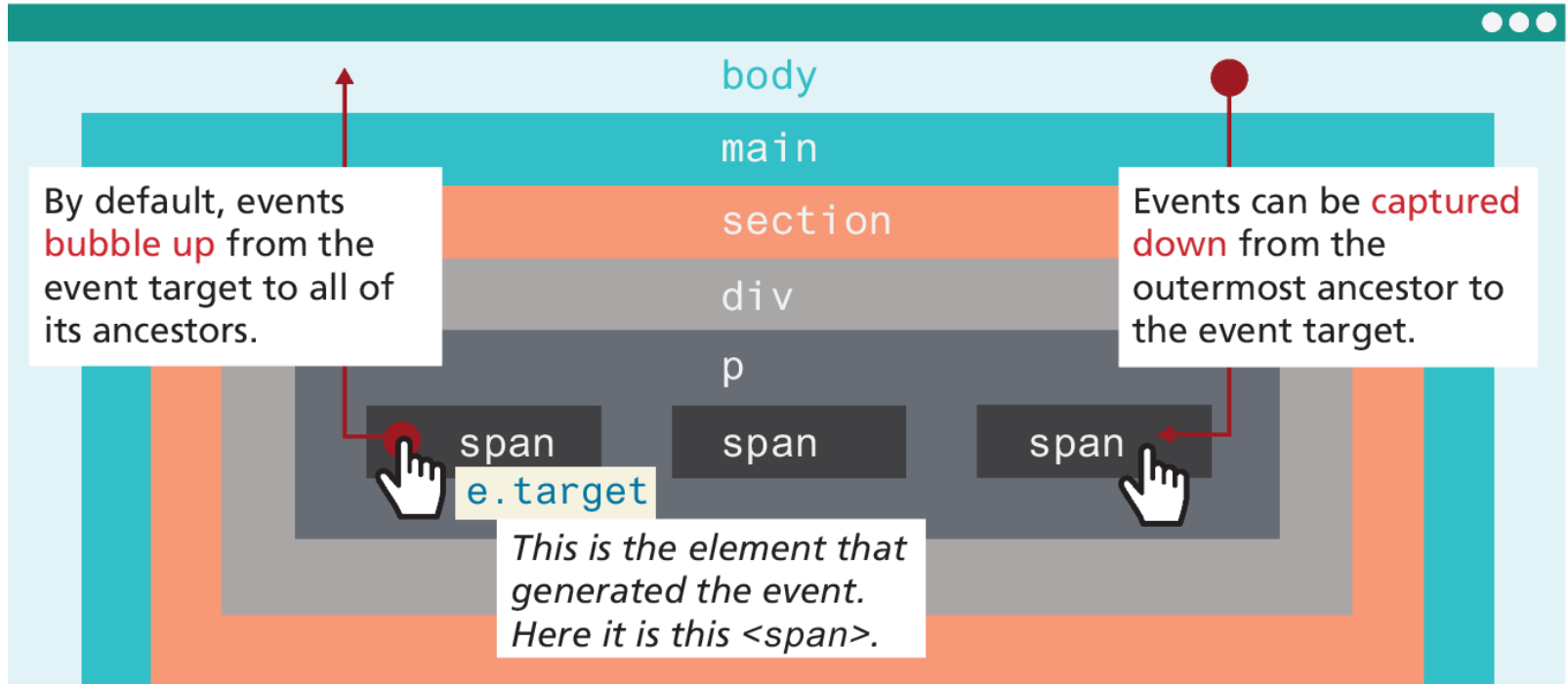
The `e.target` object in this case is referencing the clicked `` item.

Event Propagation

When an event fires on an element that has ancestor elements, the event propagates to those ancestors. There are two distinct phases of propagation:

- In the **event capturing phase**, the browser checks the outermost ancestor (the `<html>` element) to see if that element has an event handler registered for the triggered event, and if so, it is executed. It then proceeds to the next ancestor and performs the same steps; this continues until it reaches the element that triggered the event (that is, the **event target**).
- In the **event bubbling phase**, the opposite occurs. The browser checks if the element that triggered the event has an event handler registered for that event, and if so, it is executed.

Event capture and bubbling

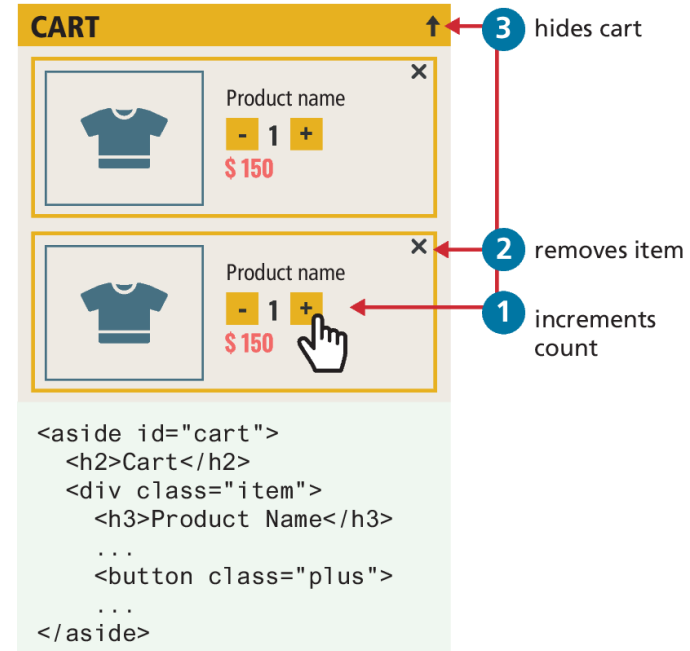


Problems with event propagation

Occasionally, the bubbling of events can cause problems. For instance consider elements nested within one another, each with its own on-click behaviors.

When the user clicks on the increment count button, the click handler for the increment `<button>` will trigger first. Unfortunately, it will then trigger the click event for the `<div>`, and the `<aside>` element!

Thankfully, there is a solution to such problems. The `stopPropagation()` method of the event argument object will stop event propagation.



Stopping event propagation

```
const btns = document.querySelectorAll(".plus");
for (let b of btns) {
  b.addEventListener("click", function (e) {
    e.stopPropagation();
    incrementCount(e);
  });
}
```

```
const items = document.querySelectorAll(".item");
for (let it of items) {
  it.addEventListener("click", function (e) {
    e.stopPropagation();
    removeItemFromCart(e);
  });
}
```

```
const aside = document.querySelector("aside#cart");
aside.addEventListener("click", function () {
  minimizeCart();
});
```

LISTING 9.5 Stopping event propagation

Event Delegation

To avoid creating duplicate event handlers for each element within a **NodeList**, an alternative is to use **event delegation** where we assign a single listener to the parent and make use of event bubbling

Suppose we have numerous image thumbnails within a parent element, similar to the following:

```
<body>
<header>...</header>
<main>
<section id="list">
<h2>Section Title</h2>
<img ... />
<img ... />
...
</section>
</main>
</body>
```

Event Delegation (ii)

Now what if you wanted to do something special when the user clicks the mouse on an ``

You would probably write something like the following:

```
const images = document.querySelectorAll("#list img");  
for (let img of images) {  
    img.addEventListener("click", someHandler);  
}
```

Notice that this solution adds an event listener to every `` element.

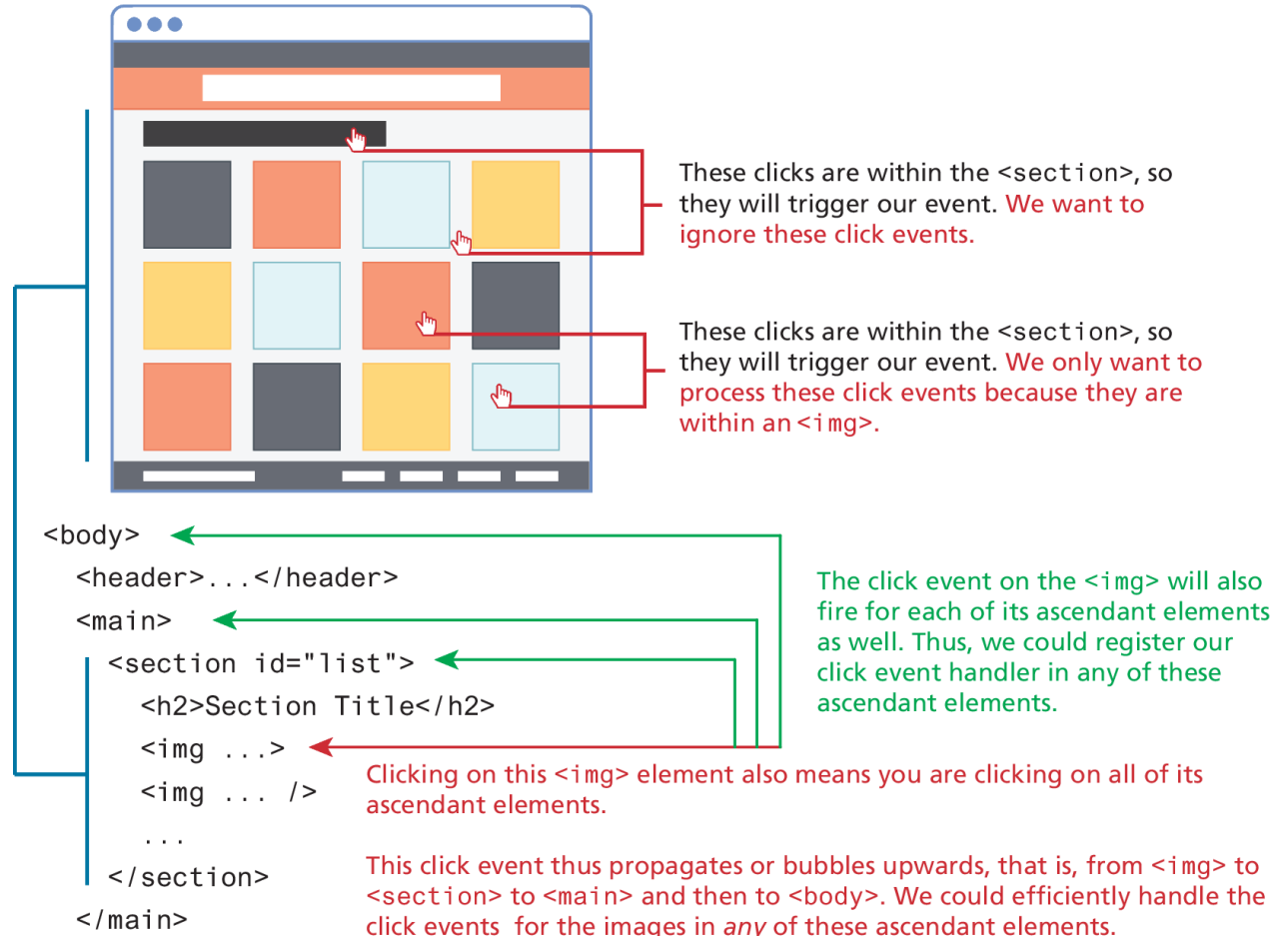
Event Delegation (iii)

Instead, we can add a single listener to the parent element, as shown in the following code

Since the user can click on all elements within the <section> element (as can be seen in Figure 9.15), the click event handler needs to determine if the user has clicked on one of the elements within it.

```
const parent = document.querySelector("#list");
parent.addEventListener("click", function (e) {
  // e.target is the object that generated the event.
  // to verify that e.target exists and that it is one of the
  // <img> elements. Note: nodeName always returns
  // upper case
  if (e.target && e.target.nodeName == "IMG") {
    doSomething(e.target);
  }
});
```

Event Delegation (iv)



Using the Dataset Property

- One of the more challenging aspects of writing JavaScript involves differences in timing between what variables are available to a function handler when it is being defined and what variables are available to that same function when it is being executed.
- The solution is to make use of the **dataset** property of the DOM element, which provides read/write access to custom data attributes (data-*) set on the element. For instance, you can make use of these via markup or via JavaScript. In markup, it can be added to any element as shown in the following:

```
<img src='file.png' id='a' data-id='5' data-country='Peru' />
```


Event Types

There are many different types of events that can be triggered in the browser. Perhaps the most obvious event is the click event, but JavaScript and the DOM support several others.

In actuality, there are several classes of event, with several types of events within each class specified by the W3C. Some of the most commonly used **event types** are:

- mouse events,
- keyboard events,
- touch events,
- form events, and
- frame events.

Mouse Events

- **click** The mouse was clicked on an element.
- **dblclick** The mouse was double clicked on an element.
- **mousedown** The mouse was pressed down over an element.
- **mouseup** The mouse was released over an element.
- **mouseover** The mouse was moved (not clicked) over an element.
- **mouseout** The mouse was moved off of an element.
- **mousemove** The mouse was moved while over an element.

Keyboard Events

Keyboard events are often overlooked by novice web developers, but are important tools for power users.

- **keydown** The user is pressing a key (this happens first).
- **keyup** The user releases a key that was down (this happens last).

```
document.getElementById("key").addEventListener("keydown", function (e) {  
    // get the raw key code  
    let keyPressed=e.key;  
    // convert to string  
    let character=String.fromCharCode(keyPressed);  
    alert("Key " + character + " was pressed");  
});
```

LISTING 9.7 Listener that hears and alerts key presses

Form Events

- **blur** Triggered when a form element has lost focus (i.e., control has moved to a different element), perhaps due to a click or Tab key press.
- **change** Some `<input>`, `<textarea>`, or `<select>` field had their value change. This could mean the user typed something, or selected a new choice.
- **focus** Complementing the blur event, this is triggered when an element gets focus (the user clicks in the field or tabs to it).
- **reset** HTML forms have the ability to be reset. This event is triggered when that happens.
- **select** When the users selects some text. This is often used to try and prevent copy/paste.
- **submit** When the form is submitted this event is triggered. We can do some prevalidation of the form in JavaScript before sending the data on to the server.

Handling the submit event

```
document.querySelector("#loginForm").addEventListener("submit",  
function(e) {  
    let pass = document.querySelector("#pw").value;  
    if (pass=="") {  
        alert ("enter a password");  
        e.preventDefault();  
    }  
});
```

LISTING 9.8 Handling the submit event

Media Events

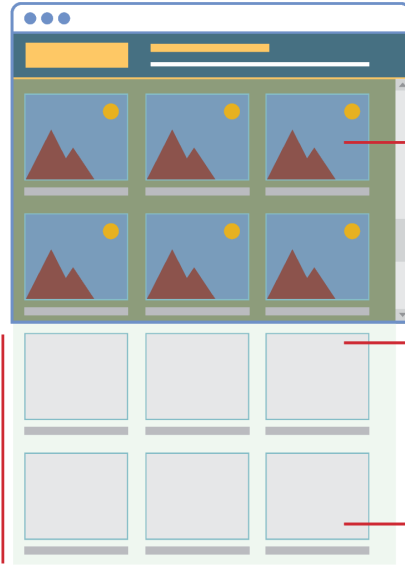
- **ended** Triggered when playback of audio or video element is completed.
- **pause** Triggered when playback is paused.
- **play** Triggered when playback is no longer paused.
- **ratechange** Triggered when playback speed changes.
- **volumechange** Triggered when audio volume has changed.

Frame Events

- **abort** An object was stopped from loading.
- **error** An object or image did not properly load.
- **load** When window content is fully loaded.
- **DOMContentLoaded** When DOM elements in document are loaded.
- **orientationchange** The device's orientation has changed from portrait to landscape, or vice-versa.
- **resize** The document view was resized.
- **scroll** The document view was scrolled.
- **unload** The document has unloaded.

Lazy Loading

1 Images that are not visible will not yet be downloaded. This will improve perceived responsiveness of the visible portion of the page.



``

``

Since no src attribute is provided, these images are not downloaded by the browser when HTML is received.

`img.lazy {
 width: 320px;
 height: 240px;
}`

CSS for images is sized correctly so that no content shifting occurs when real images are received.

2 Event listeners will be needed for scroll, resize, and orientationChange events.

```
document.addEventListener("scroll", lazyload);  
window.addEventListener("resize", lazyload);  
window.addEventListener("orientationChange", lazyload);
```

3 For each image, check if now visible. If it is, then change its src attribute to the correct one in data-src. This will make the browser request that file.

```
function lazyLoad() {  
  ...  
  const images = document.querySelectorAll("img.lazy");  
  for (let img of images) {  
    if (img.offsetTop < (window.innerHeight + window.pageYOffset)) {  
      img.src = img.dataset.src;  
      img.alt = img.dataset.alt;  
      img.classList.remove('lazy');  
    }  
  }  
}
```


Forms in JavaScript

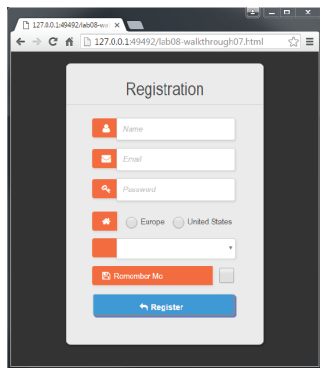
Chapter 5 covered the HTML for data entry forms.

JavaScript within forms is more than just the client-side validation of form data; JavaScript is also used to improve the user experience of the typical browser-based form.

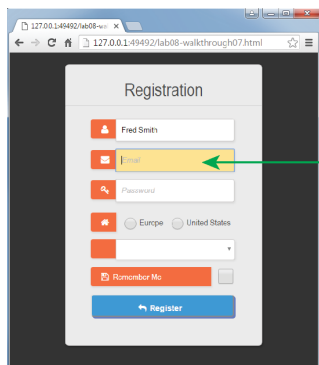
As a result, when working with forms in JavaScript, we are typically interested in three types of events:

- movement between elements,
- data being changed within a form element, and
- the final submission of the form.

Responding to form movement events



How form appears when no controls have the focus



When a control has the focus, then change its background color

```
// This function is going to get called every time the focus or blur events are triggered in one of our form's input elements.
```

```
function setBackground(e) {  
  if (e.type == "focus") {  
    e.target.style.backgroundColor = "#FFE393";  
  }  
  else if (e.type == "blur") {  
    e.target.style.backgroundColor = "white";  
  }  
}
```

Here we use the `style` property instead of the `classList` property because of specificity conflicts (that is, attribute selectors override class selectors).

```
// set up the event listeners only after the DOM is loaded
```

```
window.addEventListener("load", function() {  
  const cssSelector = "input[type=text],input[type=password]";  
  const fields = document.querySelectorAll(cssSelector);  
  for (let f of fields) {  
    f.addEventListener("focus", setBackground);  
    f.addEventListener("blur", setBackground);  
  }  
});
```

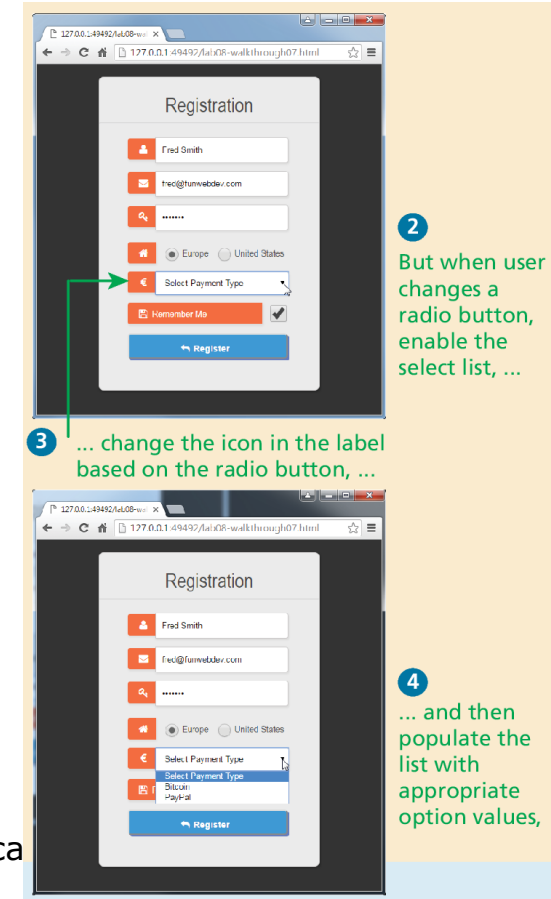
Selects the fields that will change.

Assigns the `setBackground()` function to change the background color of the control depending upon whether it has the focus.

Responding to Form Changes Events

We may want to change the options available within a form based on earlier user entry. For instance, we may want the payment options to be different based on the value of the region radio button.

Figure 9.19 demonstrates how we can add event listeners to the change event of the radio buttons; when one of these buttons changes its value, then the callback function will set the available payment options based on the selected region. The listing also changes the associated payment label as well.



Validating a Submitted Form

Form validation continues to be one of the most common applications of JavaScript.

Checking user inputs to ensure that they follow expected rules must happen on the server side for security reasons (in case JavaScript was circumvented); checking those same inputs on the client side using JavaScript will reduce server load and increase the perceived speed and responsiveness of the form.

Some of the more common validation activities include email validation, number validation, and data validation.

In practice, regular expressions (covered in Section 9.6) are used to concisely implement many of these validation checks.

Empty Field Validation

```
const form = document.querySelector("#loginForm");
form.addEventListener("submit", (e) => {
  const fieldValue = document.querySelector("#username").value;
  if (fieldValue == null || fieldValue == "") {
    // the field was empty. Stop form submission
    e.preventDefault();
    // Now tell the user something went wrong
    console.log("you must enter a username");
  }
});
```

LISTING 9.10 A simple validation script to check for empty fields

Determining which items in multiselect list are selected

```
const multi = document.querySelector("#listbox");  
// using the options technique loops through each option and check if it is selected  
for (let i=0; i < multi.options.length; i++) {  
    if (multi.options[i].selected) {  
        // this option was selected, do something with it ...  
        console.log(multi.options[i].textContent);  
    }  
}  
  
// the selectedOptions technique is simpler ... it only loops through the selected options  
for (let i=0; i < multi.selectedOptions.length; i++) {  
    console.log(multi.selectedOptions[i].textContent);  
}
```

LISTING 9.11 Determining which items in multiselect list are selected

Number Validation

Unfortunately, no simple functions exist for number validation like one might expect from a fullfledged library. Using `parseInt()`, `isNaN()`, and `isFinite()`, you can write your own number validation function.

Validating email, phone numbers, or social security numbers would include checking for blank fields and making use of `isNumeric` and regular expressions.

```
function isNumeric(n) {  
    return !isNaN(parseFloat(n)) && isFinite(n);  
}
```

LISTING 9.12 A function to test for a numeric value

Submitting Forms

Submitting a form using JavaScript requires having a node variable for the form element. Once the variable, say, *formExample* is acquired, one can simply call the **submit()** method:

```
const formExample = document.getElementById("loginForm");  
formExample.submit();
```

This is often done in conjunction with calling `preventDefault()` on the submit event.

It is possible to submit a form multiple times by clicking buttons quickly. The easiest way to protect against this is to simply disable the submit button immediately in the event handler for the submit event.

Regular Expressions

A **regular expression** is a set of special characters that define a pattern.

Their history predates the world of web development, as evidenced by the formal specification defined by the IEEE POSIX standard.

PHP, JavaScript, Java, the .NET environment, and most other modern languages support regular expressions.

Regular Expression Syntax

A regular expression consists of two types of characters: literals and metacharacters.

A **literal** is just a character you wish to match in the target (i.e., the text that you are searching within).

A **metacharacter** is a special symbol that acts as a command to the regular expression parser (there are 14, listed below).

. [] \ () ^ \$ | * ? { } +

Table 9.12 lists examples of typical metacharacter usage to create patterns;

Regular Expression Syntax (ii)

In JavaScript, regular expressions are case sensitive and contained within forward slashes. For instance

let pattern = /ran/;

will find matches in all three of the following strings:

'randy connolly'

'Sue ran to the store'

'I would like a cranberry'

Listing 9.13 contains a more complex regular expression whose development is described in textbook

Key Terms

blur	Element Node	event propagation	linter
Document Object Model (DOM)	event bubbling phase	event target	media events
document root	event capturing phase	event type	metacharacter
DOM document	event delegation	focus	mouse events
object	event handler	form events	node
DOM tree	event object	frame events	nodeList
		iteral	regular expression
		keyboard events	selection methods

Copyright



This work is protected by United States copyright laws and is provided solely for the use of instructors in teaching their courses and assessing student learning. Dissemination or sale of any part of this work (including on the World Wide Web) will destroy the integrity of the work and is not permitted. The work and materials from it should never be made available to students except by instructors using the accompanying text in their classes. All recipients of this work are expected to abide by these restrictions and to honor the intended pedagogical purposes and the needs of other instructors who rely on these materials.