1. *Web Programming Basics* -- Web browsers display HTML documents. If you want a web browser to execute JavaScript code, you must include (or reference) that code from an HTML document, and this is what the HTML <script> tag does. JavaScript code can appear inline within an HTML file between <script> and </script> tags. Here, for example, is an HTML file that includes a script tag with JavaScript code that dynamically updates one element of the document to make it behave like a digital clock:

<!DOCTYPE html>

<html>

<head>

<title>Digital Clock</title>

<style> #clock { font: bold 24px sans-serif; background: #ddf; padding: 15px; border: solid black 2px; border-radius: 10px; }

</style>

</head>

<body>

<h1>Digital Clock</h1>

<span id="clock"></span>

<script>

function displayTime() {

let clock = document.querySelector("#clock");

let now = new Date();

clock.textContent = now.toLocaleTimeString();}

displayTime()

setInterval(displayTime, 1000);

</script>

</body>

</html>

Although JavaScript code can be embedded directly within a <script> tag, it is more common to instead use the src attribute of the <script> tag to specify the URL (an absolute URL or a URL relative to the URL of the HTML file being displayed) of a file containing JavaScript code. If we took the JavaScript code out of this HTML file and stored it in its own scripts/digital\_clock.js file, then the <script> tag might reference that file of code like this: <script src="scripts/digital\_clock.js"></script>

1. *WHEN SCRIPTS RUN: ASYNC AND DEFERRED* -- When JavaScript was first added to web browsers, there was no API for traversing and manipulating the structure and content of an already rendered document. The only way that JavaScript code could affect the content of a document was to generate that content on the fly while the document was in the process of loading. It did this by using the document.write() method to inject HTML text into the document at the location of the script.

The use of document.write() is no longer considered good style, but the fact that it is possible means that when the HTML parser encounters a <script> element, it must, by default, run the script just to be sure that it doesn’t output any HTML before it can resume parsing and rendering the document. This can dramatically slow down parsing and rendering of the web page.

Fortunately, this default synchronous or blocking script execution mode is not the only option. The <script> tag can have defer and async attributes, which cause scripts to be executed differently. These are boolean attributes—they don’t have a value; they just need to be present on the <script> tag. Note that these attributes are only meaningful when used in conjunction with the src attribute:

<script defer src="deferred.js"></script>

<script async src="async.js"></script>

Both the defer and async attributes are ways of telling the browser that the linked script does not use document.write() to generate HTML output, and that the browser, therefore, can continue to parse and render the document while downloading the script. The defer attribute causes the browser to defer execution of the script until after the document has been fully loaded and parsed and is ready to be manipulated. The async attribute causes the browser to run the script as soon as possible but does not block document parsing while the script is being downloaded.

If a <script> tag has both attributes, the async attribute takes precedence. Note that deferred scripts run in the order in which they appear in the document. Async scripts run as they load, which means that they may execute out of order. Scripts with the type="module" attribute are, by default, executed after the document has loaded, as if they had a defer attribute. You can override this default with the async attribute, which will cause the code to be executed as soon as the module and all of its dependencies have loaded.

A simple alternative to the async and defer attributes—especially for code that is included directly in the HTML—is to simply put your scripts at the end of the HTML file. That way, the script can run knowing that the document content before it has been parsed and is ready to be manipulated.

1. *LOADING SCRIPTS ON DEMAND* -- Sometimes, you may have JavaScript code that is not used when a document first loads and is only needed if the user takes some action like clicking on a button or opening a menu. If you are developing your code using modules, you can load a module on demand with import(). If you are not using modules, you can load a file of JavaScript on demand simply by adding a <script> tag to your document when you want the script to load:

// Asynchronously load and execute a script from a specified URL

// Returns a Promise that resolves when the script has loaded.

function importScript(url) {

return new Promise((resolve, reject) => {

let s = document.createElement("script");

s.onload = () => { resolve(); };

s.onerror = (e) => { reject(e); };

s.src = url;

document.head.append(s); }); }

This importScript() function uses DOM APIs to create a new <script> tag and add it to the document <head>. And it uses event handlers to determine when the script has loaded successfully or when loading has failed.

1. *The Document Object Model* -- One of the most important objects in client-side JavaScript programming is the Document object—which represents the HTML document that is displayed in a browser window or tab. The API for working with HTML documents is known as the Document Object Model, or DOM.

The DOM API mirrors the tree structure of an HTML document. For each HTML tag in the document, there is a corresponding JavaScript Element object, and for each run of text in the document, there is a corresponding Text object. The Element and Text classes, as well as the Document class itself, are all subclasses of the more general Node class, and Node objects are organized into a tree structure that JavaScript can query and traverse using the DOM API.

There is a JavaScript class corresponding to each HTML tag type, and each occurrence of the tag in a document is represented by an instance of the class. The <body> tag, for example, is represented by an instance of HTMLBodyElement, and a <table> tag is represented by an instance of HTMLTableElement. The JavaScript element objects have properties that correspond to the HTML attributes of the tags. For example, instances of HTMLImageElement, which represent <img> tags, have a src property that corresponds to the src attribute of the tag. The initial value of the src property is the attribute value that appears in the HTML tag, and setting this property with JavaScript changes the value of the HTML attribute (and causes the browser to load and display a new image). Most of the JavaScript element classes just mirror the attributes of an HTML tag, but some define additional methods. The HTMLAudioElement and HTMLVideoElement classes, for example, define methods like play() and pause() for controlling playback of audio and video files.

1. *The Global Object in Web Browsers* -- There is one global object per browser window or tab. All of the JavaScript code (except code running in worker threads) running in that window shares this single global object. This is true regardless of how many scripts or modules are in the document: all the scripts and modules of a document share a single global object; if one script defines a property on that object, that property is visible to all the other scripts as well.

The global object is where JavaScript’s standard library is defined—the parseInt() function, the Math object, the Set class, and so on. In web browsers, the global object also contains the main entry points of various web APIs. For example, the document property represents the currently displayed document, the fetch() method makes HTTP network requests, and the Audio() constructor allows JavaScript programs to play sounds.

In web browsers, the global object does double duty: in addition to defining built-in types and functions, it also represents the current web browser window and defines properties like history, which represent the window’s browsing history, and innerWidth, which holds the window’s width in pixels. One of the properties of this global object is named window, and its value is the global object itself. This means that you can simply type window to refer to the global object in your client-side code. When using window-specific features, it is often a good idea to include a window. prefix: window.innerWidth is clearer than innerWidth, for example.

In modules, top-level declarations are scoped to the module and can be explicitly exported. In nonmodule scripts, however, top-level declarations are scoped to the containing document, and the declarations are shared by all scripts in the document. Older var and function declarations are shared via properties of the global object. Newer const, let, and class declarations are also shared and have the same document scope, but they do not exist as properties of any object that JavaScript code has access to.

1. Execution of JavaScript Programs -- You can think of JavaScript program execution as occurring in two phases. In the first phase, the document content is loaded, and the code from <script> elements (both inline scripts and external scripts) is run. Scripts generally run in the order in which they appear in the document, though this default order can be modified by the async and defer attributes we’ve described.

The JavaScript code within any single script is run from top to bottom, subject, of course, to JavaScript’s conditionals, loops, and other control statements. Some scripts don’t really do anything during this first phase and instead just define functions and classes for use in the second phase. Other scripts might do significant work during the first phase and then do nothing in the second. Imagine a script at the very end of a document that finds all <h1> and <h2> tags in the document and modifies the document by generating and inserting a table of contents at the beginning of the document. This could be done entirely in the first phase.

Once the document is loaded and all scripts have run, JavaScript execution enters its second phase. This phase is asynchronous and event-driven. If a script is going to participate in this second phase, then one of the things it must have done during the first phase is to register at least one event handler or other callback function that will be invoked asynchronously. During this event-driven second phase, the web browser invokes event handler functions and other callbacks in response to events that occur asynchronously. Event handlers are most commonly invoked in response to user input (mouse clicks, keystrokes, etc.) but may also be triggered by network activity, document and resource loading, elapsed time, or errors in JavaScript code.

Some of the first events to occur during the event-driven phase are the “DOMContentLoaded” and “load” events. “DOMContentLoaded” is triggered when the HTML document has been completely loaded and parsed. The “load” event is triggered when all of the document’s external resources—such as images—are also fully loaded. JavaScript programs often use one of these events as a trigger or starting signal. It is common to see programs whose scripts define functions but take no action other than registering an event handler function to be triggered by the “load” event at the beginning of the event-driven phase of execution. It is this “load” event handler that then manipulates the document and does whatever it is that the program is supposed to do. Note that it is common in JavaScript programming for an event handler function such as the “load” event handler described here to register other event handlers.

The loading phase of a JavaScript program is relatively short: ideally less than a second. Once the document is loaded, the event-driven phase lasts for as long as the document is displayed by the web browser. Because this phase is asynchronous and event-driven, there may be long periods of inactivity where no JavaScript is executed, punctuated by bursts of activity triggered by user or network events. We’ll cover these two phases in more detail next.

1. *CLIENT-SIDE JAVASCRIPT THREADING MODEL* -- JavaScript is a single-threaded language, and single-threaded execution makes for much simpler programming: you can write code with the assurance that two event handlers will never run at the same time. You can manipulate document content knowing that no other thread is attempting to modify it at the same time, and you never need to worry about locks, deadlock, or race conditions when writing JavaScript code.

Single-threaded execution means that web browsers stop responding to user input while scripts and event handlers are executing. This places a burden on JavaScript programmers: it means that JavaScript scripts and event handlers must not run for too long. If a script performs a computationally intensive task, it will introduce a delay into document loading, and the user will not see the document content until the script completes. If an event handler performs a computationally intensive task, the browser may become nonresponsive, possibly causing the user to think that it has crashed.

The web platform defines a controlled form of concurrency called a “web worker.” A web worker is a background thread for performing computationally intensive tasks without freezing the user interface. The code that runs in a web worker thread does not have access to document content, does not share any state with the main thread or with other workers, and can only communicate with the main thread and other workers through asynchronous message events, so the concurrency is not detectable to the main thread, and web workers do not alter the basic single-threaded execution model of JavaScript programs.

1. *CLIENT-SIDE JAVASCRIPT TIMELINE* -- We’ve already seen that JavaScript programs begin in a script execution phase and then transition to an event-handling phase. These two phases can be further broken down into the following steps:

* The web browser creates a Document object and begins parsing the web page, adding Element objects and Text nodes to the document as it parses HTML elements and their textual content. The document.readyState property has the value “loading” at this stage.
* When the HTML parser encounters a <script> tag that does not have any of the async, defer, or type="module" attributes, it adds that script tag to the document and then executes the script. The script is executed synchronously, and the HTML parser pauses while the script downloads (if necessary) and runs. A script like this can use document.write() to insert text into the input stream, and that text will become part of the document when the parser resumes. A script like this often simply defines functions and registers event handlers for later use, but it can traverse and manipulate the document tree as it exists at that time. That is, non-module scripts that do not have an async or defer attribute can see their own <script> tag and document content that comes before it.
* When the parser encounters a <script> element that has the async attribute set, it begins downloading the script text (and if the script is a module, it also recursively downloads all of the script’s dependencies) and continues parsing the document. The script will be executed as soon as possible after it has downloaded, but the parser does not stop and wait for it to download. Asynchronous scripts must not use the document.write() method. They can see their own <script> tag and all document content that comes before it, and may or may not have access to additional document content.
* When the document is completely parsed, the document.readyState property changes to “interactive.”
* Any scripts that had the defer attribute set (along with any module scripts that do not have an async attribute) are executed in the order in which they appeared in the document. Async scripts may also be executed at this time. Deferred scripts have access to the complete document and they must not use the document.write() method.
* The browser fires a “DOMContentLoaded” event on the Document object. This marks the transition from synchronous script-execution phase to the asynchronous, event-driven phase of program execution. Note, however, that there may still be async scripts that have not yet executed at this point.
* The document is completely parsed at this point, but the browser may still be waiting for additional content, such as images, to load. When all such content finishes loading, and when all async scripts have loaded and executed, the document.readyState property changes to “complete” and the web browser fires a “load” event on the Window object.
* From this point on, event handlers are invoked asynchronously in response to user input events, network events, timer expirations, and so on.

1. *Program Input and Output* -- Like any program, client-side JavaScript programs process input data to produce output data. There are a variety of inputs available:

* The content of the document itself, which JavaScript code can access with the DOM API.
* User input, in the form of events, such as mouse clicks (or touch-screen taps) on HTML <button> elements, or text entered into HTML <textarea> elements.
* The URL of the document being displayed is available to client-side JavaScript as document.URL. If you pass this string to the URL() constructor, you can easily access the path, query, and fragment sections of the URL.
* The content of the HTTP “Cookie” request header is available to client-side code as document.cookie. Cookies are usually used by server-side code for maintaining user sessions, but client-side code can also read (and write) them if necessary.
* The global navigator property provides access to information about the web browser, the OS it’s running on top of, and the capabilities of each. For example, navigator.userAgent is a string that identifies the web browser, navigator.language is the user’s preferred language, and navigator.hardwareConcurrency returns the number of logical CPUs available to the web browser. Similarly, the global screen property provides access to the user’s display size via the screen.width and screen.height properties. In a sense, these navigator and screen objects are to web browsers what environment variables are to Node programs.

Client-side JavaScript typically produces output, when it needs to, by manipulating the HTML document with the DOM API or by using a higher-level framework such as React or Angular to manipulate the document. Client-side code can also use console.log() and related methods to produce output. But this output is only visible in the web developer console, so it is useful when debugging, but not for user-visible output.

1. *Program Errors* -- Unlike applications (such as Node applications) that run directly on top of the OS, JavaScript programs in a web browser can’t really “crash.” If an exception occurs while your JavaScript program is running, and if you do not have a catch statement to handle it, an error message will be displayed in the developer console, but any event handlers that have been registered keep running and responding to events.

If you would like to define an error handler of last resort to be invoked when this kind of uncaught exception occurs, set the onerror property of the Window object to an error handler function. When an uncaught exception propagates all the way up the call stack and an error message is about to be displayed in the developer console, the window.onerror function will be invoked with three string arguments. The first argument to window.onerror is a message describing the error. The second argument is a string that contains the URL of the JavaScript code that caused the error. The third argument is the line number within the document where the error occurred. If the onerror handler returns true, it tells the browser that the handler has handled the error and that no further action is necessary—in other words, the browser should not display its own error message.

When a Promise is rejected and there is no .catch() function to handle it, that is a situation much like an unhandled exception: an unanticipated error or a logic error in your program. You can detect this by defining a window.onunhandledrejection function or by using window.addEventListener() to register a handler for “unhandledrejection” events. The event object passed to this handler will have a promise property whose value is the Promise object that rejected and a reason property whose value is what would have been passed to a .catch() function. As with the error handlers described earlier, if you call preventDefault() on the unhandled rejection event object, it will be considered handled and won’t cause an error message in the developer console.

1. *Events* -- In client-side JavaScript, events can occur on any element within an HTML document, and this fact makes the event model of web browsers significantly more complex than Node’s event model. We begin this section with some important definitions that help to explain that event model:

* *event type* -- This string specifies what kind of event occurred. Examples, mousemove, keydown, load etc.
* *event target* -- This is the object on which the event occurred or with which the event is associated. When we speak of an event, we must specify both the type and the target. A load event on a Window, for example, or a click event on a <button> Element. Window, Document, and Element objects are the most common event targets in client-side JavaScript applications, but some events are triggered on other kinds of objects. For example, a Worker object is a target for “message” events that occur when the worker thread sends a message to the main thread.
* *event handler, or event listener* -- This function handles or responds to an event. Applications register their event handler functions with the web browser, specifying an event type and an event target. When an event of the specified type occurs on the specified target, the browser invokes the handler function. When event handlers are invoked for an object, we say that the browser has “fired,” “triggered,” or “dispatched” the event.
* *event object* -- This object is associated with a particular event and contains details about that event. Event objects are passed as an argument to the event handler function. All event objects have a type property that specifies the event type and a target property that specifies the event target. Each event type defines a set of properties for its associated event object. The object associated with a mouse event includes the coordinates of the mouse pointer, for example, and the object associated with a keyboard event contains details about the key that was pressed and the modifier keys that were held down. Many event types define only a few standard properties—such as type and target—and do not carry much other useful information. For those events, it is the simple occurrence of the event, not the event details, that matter.
* *event propagation* -- This is the process by which the browser decides which objects to trigger event handlers on. For events that are specific to a single object—such as the “load” event on the Window object or a “message” event on a Worker object—no propagation is required. But when certain kinds of events occur on elements within the HTML document, however, they propagate or “bubble” up the document tree. If the user moves the mouse over a hyperlink, the mousemove event is first fired on the <a> element that defines that link. Then it is fired on the containing elements: perhaps a <p> element, a <section> element, and the Document object itself. It is sometimes more convenient to register a single event handler on a Document or other container element than to register handlers on each individual element you’re interested in. An event handler can stop the propagation of an event so that it will not continue to bubble and will not trigger handlers on containing elements. Handlers do this by invoking a method of the event object. In another form of event propagation, known as event capturing, handlers specially registered on container elements have the opportunity to intercept (or “capture”) events before they are delivered to their actual target.

Some events have default actions associated with them. When a click event occurs on a hyperlink, for example, the default action is for the browser to follow the link and load a new page. Event handlers can prevent this default action by invoking a method of the event object. This is sometimes called “canceling” the event.

1. *Event Categories* -- Client-side JavaScript supports such a large number of event types that there is no way this chapter can cover them all. It can be useful, though, to group events into some general categories, to illustrate the scope and wide variety of supported events:

* *Device-dependent input events* -- These events are directly tied to a specific input device, such as the mouse or keyboard. They include event types such as “mousedown,” “mousemove,” “mouseup,” “touchstart,” “touchmove,” “touchend,” “keydown,” and “keyup.”
* Device-independent input events These input events are not directly tied to a specific input device. The “click” event, for example, indicates that a link or button (or other document element) has been activated. This is often done via a mouse click, but it could also be done by keyboard or (on touchsensitive devices) with a tap. The “input” event is a deviceindependent alternative to the “keydown” event and supports keyboard input as well as alternatives such as cut-and-paste and input methods used for ideographic scripts. The “pointerdown,” “pointermove,” and “pointerup” event types are device-independent alternatives to mouse and touch events. They work for mouse-type pointers, for touch screens, and for pen- or stylus-style input as well.
* User interface events UI events are higher-level events, often on HTML form elements that define a user interface for a web application. They include the “focus” event (when a text input field gains keyboard focus), the “change” event (when the user changes the value displayed by a form element), and the “submit” event (when the user clicks a Submit button in a form).
* *State-change events* -- Some events are not triggered directly by user activity, but by network or browser activity, and indicate some kind of life-cycle or state-related change. The “load” and “DOMContentLoaded” events —fired on the Window and Document objects, respectively, at the end of document loading—are probably the most commonly used of these events. Browsers fire “online” and “offline” events on the Window object when network connectivity changes. The browser’s history management mechanism fires the “popstate” event in response to the browser’s Back button.
* *API-specific events* -- A number of web APIs defined by HTML and related specifications include their own event types. The HTML <video> and <audio> elements define a long list of associated event types such as “waiting,” “playing,” “seeking,” “volumechange,” and so on, and you can use them to customize media playback.

1. *Registering Event Handlers* -- There are two basic ways to register event handlers. The first, from the early days of the web, is to set a property on the object or document element that is the event target. The second (newer and more general) technique is to pass the handler to the addEventListener() method of the object or element.

*SETTING EVENT HANDLER PROPERTIES* -- The simplest way to register an event handler is by setting a property of the event target to the desired event handler function. By convention, event handler properties have names that consist of the word “on” followed by the event name: onclick, onchange, onload, onmouseover, and so on.

window.onload = function() {

let form = document.querySelector("form#shipping");

form.onsubmit = function(event) {

if (!isFormValid(this)) { event.preventDefault();} }; };

The shortcoming of event handler properties is that they are designed around the assumption that event targets will have at most one handler for each type of event. It is often better to register event handlers using addEventListener() because that technique does not overwrite any previously registered handlers.

1. *ADDEVENTLISTENER()* -- Any object that can be an event target—this includes the Window and Document objects and all document Elements—defines a method named addEventListener() that you can use to register an event handler for that target. addEventListener() takes three arguments. The first is the event type for which the handler is being registered. The event type (or name) is a string that does not include the “on” prefix used when setting event handler properties. The second argument to addEventListener() is the function that should be invoked when the specified type of event occurs. The third argument is optional and is explained below. The following code registers two handlers for the “click” event on a <button> element. Note the differences between the two techniques used:

<button id="mybutton">Click me</button>

<script>

let b = document.querySelector("#mybutton");

b.onclick = function() { console.log("Thanks for clicking me!"); };

b.addEventListener("click", () => { console.log("Thanks again!"); });

</script>

Calling addEventListener() with “click” as its first argument does not affect the value of the onclick property. In this code, a button click will log two messages to the developer console. And if we called addEventListener() first and then set onclick, we would still log two messages, just in the opposite order. More importantly, you can call addEventListener() multiple times to register more than one handler function for the same event type on the same object. When an event occurs on an object, all of the handlers registered for that type of event are invoked in the order in which they were registered. Invoking addEventListener() more than once on the same object with the same arguments has no effect—the handler function remains registered only once, and the repeated invocation does not alter the order in which handlers are invoked.

addEventListener() is paired with a removeEventListener() method that expects the same two arguments (plus an optional third) but removes an event handler function from an object rather than adding it. It is often useful to temporarily register an event handler and then remove it soon afterward. For example, when you get a “mousedown” event, you might register temporary event handlers for “mousemove” and “mouseup” events so that you can see if the user drags the mouse. You’d then deregister these handlers when the “mouseup” event arrives. In such a situation, your event handler removal code might look like this:

document.removeEventListener("mousemove", handleMouseMove);

document.removeEventListener("mouseup", handleMouseUp);

The optional third argument to addEventListener() is a boolean value or object. If you pass true, then your handler function is registered as a capturing event handler and is invoked at a different phase of event dispatch. If you pass a third argument of true when you register an event listener, then you must also pass true as the third argument to removeEventListener() if you want to remove the handler. Registering a capturing event handler is only one of the three options that addEventListener() supports, and instead of passing a single boolean value, you can also pass an object that explicitly specifies the options you want:

document.addEventListener("click", handleClick, { capture: true, once: true, passive: true });

If the Options object has a capture property set to true, then the event handler will be registered as a capturing handler. If that property is false or is omitted, then the handler will be non-capturing. If the Options object has a once property set to true, then the event listener will be automatically removed after it is triggered once. If this property is false or is omitted, then the handler is never automatically removed. If the Options object has a passive property set to true, it indicates that the event handler will never call preventDefault() to cancel the default action. This is particularly important for touch events on mobile devices—if event handlers for “touchmove” events can prevent the browser’s default scrolling action, then the browser cannot implement smooth scrolling. This passive property provides a way to register a potentially disruptive event handler of this sort but lets the web browser know that it can safely begin its default behavior —such as scrolling—while the event handler is running. Smooth scrolling is so important for a good user experience that Firefox and Chrome make “touchmove” and “mousewheel” events passive by default. So if you actually want to register a handler that calls preventDefault() for one of these events, you should explicitly set the passive property to false. You can also pass an Options object to removeEventListener(), but the capture property is the only one that is relevant. There is no need to specify once or passive when removing a listener, and these properties are ignored.

1. *EVENT HANDLER ARGUMENT* -- Event handlers are invoked with an Event object as their single argument. The properties of the Event object provide details about the event:

* *type* -- The type of the event that occurred.
* *target* -- The object on which the event occurred.
* currentTarget -- For events that propagate, this property is the object on which the current event handler was registered.
* *timestamp* -- A timestamp (in milliseconds) that represents when the event occurred but that does not represent an absolute time. You can determine the elapsed time between two events by subtracting the timestamp of the first event from the timestamp of the second.
* *isTrusted* -- This property will be true if the event was dispatched by the web browser itself and false if the event was dispatched by JavaScript code.

Specific kinds of events have additional properties. Mouse and pointer events, for example, have clientX and clientY properties that specify the window coordinates at which the event occurred.

*EVENT HANDLER CONTEXT* -- When you register an event handler by setting a property, it looks as if you are defining a new method on the target object: target.onclick = function() { /\* handler code \*/ };

It isn’t surprising, therefore, that event handlers are invoked as methods of the object on which they are defined. That is, within the body of an event handler, the this keyword refers to the object on which the event handler was registered. Handlers are invoked with the target as their this value, even when registered using addEventListener(). This does not work for handlers defined as arrow functions, however: arrow functions always have the same this value as the scope in which they are defined.

*HANDLER RETURN VALUE* -- In modern JavaScript, event handlers should not return anything. You may see event handlers that return values in older code, and the return value is typically a signal to the browser that it should not perform the default action associated with the event. If the onclick handler of a Submit button in a form returns false, for example, then the web browser will not submit the form (usually because the event handler determined that the user’s input fails client-side validation). The standard and preferred way to prevent the browser from performing a default action is to call the preventDefault() method on the Event object.

*INVOCATION ORDER* -- An event target may have more than one event handler registered for a particular type of event. When an event of that type occurs, the browser invokes all of the handlers in the order in which they were registered. Interestingly, this is true even if you mix event handlers registered with addEventListener() with an event handler registered on an object property like onclick.

1. *Event Propagation* -- When the target of an event is the Window object or some other standalone object, the browser responds to an event simply by invoking the appropriate handlers on that one object. When the event target is a Document or document Element, however, the situation is more complicated.

After the event handlers registered on the target element are invoked, most events “bubble” up the DOM tree. The event handlers of the target’s parent are invoked. Then the handlers registered on the target’s grandparent are invoked. This continues up to the Document object, and then beyond to the Window object. Event bubbling provides an alternative to registering handlers on lots of individual document elements: instead, you can register a single handler on a common ancestor element and handle events there. Most events that occur on document elements bubble. Notable exceptions are the “focus,” “blur,” and “scroll” events. The “load” event on document elements bubbles, but it stops bubbling at the Document object and does not propagate on to the Window object. (The “load” event handlers of the Window object are triggered only when the entire document has loaded.)

Event bubbling is the third “phase” of event propagation. The invocation of the event handlers of the target object itself is the second phase. The first phase, which occurs even before the target handlers are invoked, is called the “capturing” phase. Recall that addEventListener() takes an optional third argument. If that argument is true, or {capture:true}, then the event handler is registered as a capturing event handler for invocation during this first phase of event propagation.

The capturing phase of event propagation is like the bubbling phase in reverse. The capturing handlers of the Window object are invoked first, then the capturing handlers of the Document object, then of the body object, and so on down the DOM tree until the capturing event handlers of the parent of the event target are invoked. Capturing event handlers registered on the event target itself are not invoked.

Event capturing provides an opportunity to peek at events before they are delivered to their target. A capturing event handler can be used for debugging, or it can be used along with the event cancellation technique to filter events so that the target event handlers are never actually invoked. One common use for event capturing is handling mouse drags, where mouse motion events need to be handled by the object being dragged, not the document elements over which it is dragged.

1. *Event Cancellation* -- Browsers respond to many user events, even if your code does not: when the user clicks the mouse on a hyperlink, the browser follows the link. If an HTML text input element has the keyboard focus and the user types a key, the browser will enter the user’s input. If the user moves their finger across a touch-screen device, the browser scrolls. If you register an event handler for events like these, you can prevent the browser from performing its default action by invoking the preventDefault() method of the event object. (Unless you registered the handler with the passive option, which makes preventDefault() ineffective.)

Canceling the default action associated with an event is only one kind of event cancellation. We can also cancel the propagation of events by calling the stopPropagation() method of the event object. If there are other handlers defined on the same object, the rest of those handlers will still be invoked, but no event handlers on any other object will be invoked after stopPropagation() is called. stopPropagation() works during the capturing phase, at the event target itself, and during the bubbling phase. stopImmediatePropagation() works like stopPropagation(), but it also prevents the invocation of any subsequent event handlers registered on the same object.

*Dispatching Custom Events* -- Client-side JavaScript’s event API is a relatively powerful one, and you can use it to define and dispatch your own events. Suppose, for example, that your program periodically needs to perform a long calculation or make a network request and that, while this operation is pending, other operations are not possible. You want to let the user know about this by displaying “spinners” to indicate that the application is busy. But the module that is busy should not need to know where the spinners should be displayed. Instead, that module might just dispatch an event to announce that it is busy and then dispatch another event when it is no longer busy. Then, the UI module can register event handlers for those events and take whatever UI actions are appropriate to notify the user.

If a JavaScript object has an addEventListener() method, then it is an “event target,” and this means it also has a dispatchEvent() method. You can create your own event object with the CustomEvent() constructor and pass it to dispatchEvent(). The first argument to CustomEvent() is a string that specifies the type of your event, and the second argument is an object that specifies the properties of the event object. Set the detail property of this object to a string, object, or other value that represents the content of your event. If you plan to dispatch your event on a document element and want it to bubble up the document tree, add bubbles:true to the second argument:

document.dispatchEvent(new CustomEvent("busy", { detail: true }));

fetch(url) .then(handleNetworkResponse) .catch(handleNetworkError) .finally(() => { document.dispatchEvent(new CustomEvent("busy", { detail: false })); });

// Elsewhere, in your program you can register a handler for "busy" events

// and use it to show or hide the spinner to let the user know.

document.addEventListener("busy", (e) => { if (e.detail) { showSpinner(); } else { hideSpinner(); } });

1. *Selecting Document Elements* -- Client-side JavaScript programs often need to manipulate one or more elements within the document. The global document property refers to the Document object, and the Document object has head and body properties that refer to the Element objects for the <head> and <body> tags, respectively. But a program that wants to manipulate an element embedded more deeply in the document must somehow obtain or select the Element objects that refer to those document elements.

*SELECTING ELEMENTS WITH CSS SELECTORS* -- CSS stylesheets have a very powerful syntax, known as selectors, for describing elements or sets of elements within a document. The DOM methods querySelector() and querySelectorAll() allow us to find the element or elements within a document that match a specified CSS selector. Before we cover the methods, we’ll start with a quick tutorial on CSS selector syntax. CSS selectors can describe elements by tag name, the value of their id attribute, or the words in their class attribute:

div // Any <div> element

#nav // The element with id="nav"

.warning // Any element with "warning" in its class attribute

The # character is used to match based on the id attribute, and the . character is used to match based on the class attribute. Elements can also be selected based on more general attribute values:

p[lang="fr"] // A paragraph written in French: <p lang="fr">

\*[name="x"] // Any element with a name="x" attribute

Note that these examples combine a tag name selector (or the \* tag name wildcard) with an attribute selector. More complex combinations are also possible:

span.fatal.error // Any <span> with "fatal" and "error" in its class span[lang="fr"].warning // Any <span> in French with class "warning"

Selectors can also specify document structure:

#log span // Any <span> descendant of the element with id="log"

#log>span // Any <span> child of the element with id="log"

body>h1:first-child // The first <h1> child of the <body>

img + p.caption // A <p> with class "caption" immediately after an <img>

h2 ~ p // Any <p> that follows an <h2> and is a sibling of it

If two selectors are separated by a comma, it means that we’ve selected elements that match either one of the selectors:

button,input[type="button"] // All <button> and <input type="button"> elements

As you can see, CSS selectors allow us to refer to elements within a document by type, ID, class, attributes, and position within the document. The querySelector() method takes a CSS selector string as its argument and returns the first matching element in the document that it finds, or returns null if none match:

// Find the document element for the HTML tag with attribute id="spinner"

let spinner = document.querySelector("#spinner");

querySelectorAll() is similar, but it returns all matching elements in the document rather than just returning the first:

// Find all Element objects for <h1>, <h2>, and <h3> tags

let titles = document.querySelectorAll("h1, h2, h3");

The return value of querySelectorAll() is not an array of Element objects. Instead, it is an array-like object known as a NodeList. NodeList objects have a length property and can be indexed like arrays, so you can loop over them with a traditional for loop. NodeLists are also iterable, so you can use them with for/of loops as well. If you want to convert a NodeList into a true array, simply pass it to Array.from(). The NodeList returned by querySelectorAll() will have a length property set to 0 if there are not any elements in the document that match the specified selector.

querySelector() and querySelectorAll() are implemented by the Element class as well as by the Document class. When invoked on an element, these methods will only return elements that are descendants of that element. Note that CSS defines ::first-line and ::first-letter pseudoelements. In CSS, these match portions of text nodes rather than actual elements. They will not match if used with querySelectorAll() or querySelector(). Also, many browsers will refuse to return matches for the :link and :visited pseudoclasses, as this could expose information about the user’s browsing history.

Another CSS-based element selection method is closest(). This method is defined by the Element class and takes a selector as its only argument. If the selector matches the element it is invoked on, it returns that element. Otherwise, it returns the closest ancestor element that the selector matches, or returns null if none matched. In a sense, closest() is the opposite of querySelector(): closest() starts at an element and looks for a match above it in the tree, while querySelector() starts with an element and looks for a match below it in the tree. closest() can be useful when you have registered an event handler at a high level in the document tree. If you are handling a “click” event, for example, you might want to know whether it is a click a hyperlink. The event object will tell you what the target was, but that target might be the text inside a link rather than the hyperlink’s <a> tag itself. Your event handler could look for the nearest containing hyperlink like this:

// Find the closest enclosing <a> tag that has an href attribute.

let hyperlink = event.target.closest("a[href]");

Here is another way you might use closest():

// Return true if the element e is inside of an HTML list element

function insideList(e) { return e.closest("ul,ol,dl") !== null; }

The related method matches() does not return ancestors or descendants: it simply tests whether an element is matched by a CSS selector and returns true if so and false otherwise: // Return true if e is an HTML heading element

function isHeading(e) { return e.matches("h1,h2,h3,h4,h5,h6"); }

*PRESELECTED ELEMENTS* -- For historical reasons, the Document class defines shortcut properties to access certain kinds of nodes. The images, forms, and links properties, for example, provide easy access to the <img>, <form>, and <a> elements (but only <a> tags that have an href attribute) of a document. These properties refer to HTMLCollection objects, which are much like NodeList objects, but they can additionally be indexed by element ID or name. With the document.forms property, for example, you can access the <form id="address"> tag as: document.forms.address;

1. *Document Structure and Traversal* -- Once you have selected an Element from a Document, you sometimes need to find structurally related portions (parent, siblings, children) of the document. When we are primarily interested in the Elements of a document instead of the text within them (and the whitespace between them, which is also text), there is a traversal API that allows us to treat a document as a tree of Element objects, ignoring Text nodes that are also part of the document. This traversal API does not involve any methods; it is simply a set of properties on Element objects that allow us to refer to the parent, children, and siblings of a given element:

* *parentNode* -- This property of an element refers to the parent of the element, which will be another Element or a Document object.
* *children* -- This NodeList contains the Element children of an element, but excludes non-Element children like Text nodes (and Comment nodes).
* *childElementCount* -- The number of Element children. Returns the same value as children.length.
* *firstElementChild, lastElementChild* -- These properties refer to the first and last Element children of an Element. They are null if the Element has no Element children.
* *nextElementSibling, previousElementSibling* -- These properties refer to the sibling Elements immediately before or immediately after an Element, or null if there is no such sibling.

Using these Element properties, the second child Element of the first child Element of the Document can be referred to with either of these expressions: document.children[0].children[1] document.firstElementChild.firstElementChild.nextElementSibling

(In a standard HTML document, both of those expressions refer to the <body> tag of the document.)

*DOCUMENTS AS TREES OF NODES* -- If you want to traverse a document or some portion of a document and do not want to ignore the Text nodes, you can use a different set of properties defined on all Node objects. This will allow you to see Elements, Text nodes, and even Comment nodes (which represent HTML comments in the document). All Node objects define the following properties:

* *parentNode* -- The node that is the parent of this one, or null for nodes like the Document object that have no parent.
* *childNodes* -- A read-only NodeList that that contains all children (not just Element children) of the node.
* *firstChild*, *lastChild* -- The first and last child nodes of a node, or null if the node has no children.
* *nextSibling*, *previousSibling* -- The next and previous sibling nodes of a node. These properties connect nodes in a doubly linked list.
* *nodeType* -- A number that specifies what kind of node this is. Document nodes have value 9. Element nodes have value 1. Text nodes have value 3. Comment nodes have value 8.
* *nodeValue* -- The textual content of a Text or Comment node.
* *nodeName* -- The HTML tag name of an Element, converted to uppercase.

Using these Node properties, the second child node of the first child of the Document can be referred to with expressions like these:

document.childNodes[0].childNodes[1]

document.firstChild.firstChild.nextSibling

Suppose the document in question is the following:

<html><head><title>Test</title></head><body>Hello World! </body></html> Then the second child of the first child is the <body> element. It has a nodeType of 1 and a nodeName of “BODY”. Note, however, that this API is extremely sensitive to variations in the document text. If the document is modified by inserting a single newline between the <html> and the <head> tag, for example, the Text node that represents that newline becomes the first child of the first child, and the second child is the <head> element instead of the <body> element.

1. *HTML ATTRIBUTES AS ELEMENT PROPERTIES* -- The Element objects that represent the elements of an HTML document usually define read/write properties that mirror the HTML attributes of the elements. Element defines properties for the universal HTML attributes such as id, title, lang, and dir and event handler properties like onclick. Element-specific subtypes define attributes specific to those elements. To query the URL of an image, for example, you can use the src property of the HTMLElement that represents the <img> element
2. *THE CLASS ATTRIBUTE* -- The class attribute of an HTML element is a particularly important one. Its value is a space-separated list of CSS classes that apply to the element and affect how it is styled with CSS. Because class is a reserved word in JavaScript, the value of this attribute is available through the className property on Element objects. The className property can set and return the value of the class attribute as a string. But the class attribute is poorly named: its value is a list of CSS classes, not a single class, and it is common in clientside JavaScript programming to want to add and remove individual class names from this list rather than work with the list as a single string. For this reason, Element objects define a classList property that allows you to treat the class attribute as a list. The value of the classList property is an iterable Array-like object. Although the name of the property is classList, it behaves more like a set of classes, and defines add(), remove(), contains(), and toggle() methods:

let spinner = document.querySelector("#spinner");

spinner.classList.remove("hidden"); spinner.classList.add("animated");

1. *DATASET ATTRIBUTES* -- It is sometimes useful to attach additional information to HTML elements, typically when JavaScript code will be selecting those elements and manipulating them in some way. In HTML, any attribute whose name is lowercase and begins with the prefix “data-” is considered valid, and you can use them for any purpose. These “dataset attributes” will not affect the presentation of the elements on which they appear, and they define a standard way to attach additional data without compromising document validity. In the DOM, Element objects have a dataset property that refers to an object that has properties that correspond to the data- attributes with their prefix removed. Thus, dataset.x would hold the value of the data-x attribute. Hyphenated attributes map to camelCase property names: the attribute data-section-number becomes the property dataset.sectionNumber.
2. *ELEMENT CONTENT AS HTML* -- Reading the innerHTML property of an Element returns the content of that element as a string of markup. Setting this property on an element invokes the web browser’s parser and replaces the element’s current content with a parsed representation of the new string. Note, however, that appending text to the innerHTML property with the += operator is not efficient because it requires both a serialization step to convert element content to a string and then a parsing step to convert the new string back into element content.

The outerHTML property of an Element is like innerHTML except that its value includes the element itself. When you query outerHTML, the value includes the opening and closing tags of the element. And when you set outerHTML on an element, the new content replaces the element itself. A related Element method is insertAdjacentHTML(), which allows you to insert a string of arbitrary HTML markup “adjacent” to the specified element. The markup is passed as the second argument to this method, and the precise meaning of “adjacent” depends on the value of the first argument. This first argument should be a string with one of the values “beforebegin,” “afterbegin,” “beforeend,” or “afterend.”

1. *ELEMENT CONTENT AS PLAIN TEXT* -- Sometimes you want to query the content of an element as plain text or to insert plain text into a document (without having to escape the angle brackets and ampersands used in HTML markup). The standard way to do this is with the textContent property. The textContent property is defined by the Node class, so it works for Text nodes as well as Element nodes. For Element nodes, it finds and returns all text in all descendants of the element.
2. *Creating, Inserting, and Deleting Nodes* -- Create a new element with the createElement() method of the Document class and append strings of text or other elements to it with its append() and prepend() methods:

let paragraph = document.createElement("p");

let emphasis = document.createElement("em");

emphasis.append("World");

paragraph.append("Hello ", emphasis, "!");

paragraph.prepend("¡");

paragraph.innerHTML // => "¡Hello <em>World</em>!"

append() and prepend() take any number of arguments, which can be Node objects or strings. String arguments are automatically converted to Text nodes. append() adds the arguments to the element at the end of the child list. prepend() adds the arguments at the start of the child list. If you want to insert an Element or Text node into the middle of the containing element’s child list, you should obtain a reference to a sibling node and call before() to insert the new content before that sibling or after() to insert it after that sibling.

Like append() and prepend(), after() and before() take any number of string and element arguments and insert them all into the document after converting strings to Text nodes. append() and prepend() are only defined on Element objects, but after() and before() work on both Element and Text nodes: you can use them to insert content relative to a Text node.

Note that elements can only be inserted at one spot in the document. If an element is already in the document and you insert it somewhere else, it will be moved to the new location, not copied. If you do want to make a copy of an element, use the cloneNode() method, passing true to copy all of its content. You can remove an Element or Text node from the document by calling its remove() method, or you can replace it by calling replaceWith() instead. remove() takes no arguments, and replaceWith() takes any number of strings and elements just like before() and after() do.

1. *Scripting CSS* –

*CSS Classes --* The simplest way to use JavaScript to affect the styling of document content is to add and remove CSS class names from the class attribute of HTML tags. This is easy to do with the classList property of Element objects, as explained in “The class attribute”. Suppose, for example, that your document’s stylesheet includes a definition for a “hidden” class: .hidden { display:none; }

With this style defined, you can hide/show an element with code like this:

document.querySelector("#tooltip").classList.remove("hidden");

*Inline Styles* -- To continue with the preceding tooltip example, suppose that the document is structured with only a single tooltip element, and we want to dynamically position it before displaying it. In general, we can’t create a different stylesheet class for each possible position of the tooltip, so the classList property won’t help us with positioning. In this case, we need to script the style attribute of the tooltip element to set inline styles that are specific to that one element. The DOM defines a style property on all Element objects that correspond to the style attribute. Unlike most such properties, however, the style property is not a string. Instead, it is a CSSStyleDeclaration object: a parsed representation of the CSS styles that appear in textual form in the style attribute. To display and set the position of our hypothetical tooltip with JavaScript, we might use code like this:

function displayAt(tooltip, x, y) {

tooltip.style.display = "block";

tooltip.style.position = "absolute";

tooltip.style.left = `${x}px`; tooltip.style.top = `${y}px`; }

*Computed Styles* -- The computed style for an element is the set of property values that the browser derives (or computes) from the element’s inline style plus all applicable style rules in all stylesheets: it is the set of properties actually used to display the element. Like inline styles, computed styles are represented with a CSSStyleDeclaration object. Unlike inline styles, however, computed styles are read-only. You can’t set these styles, but the computed CSSStyleDeclaration object for an element lets you determine what style property values the browser used when rendering that element. Obtain the computed style for an element with the getComputedStyle() method of the Window object. The first argument to this method is the element whose computed style is desired. The optional second argument is used to specify a CSS pseudoelement, such as “::before” or “::after”:

let title = document.querySelector("#section1title");

let styles = window.getComputedStyle(title);

let beforeStyles = window.getComputedStyle(title, "::before");

*Scripting Stylesheets* -- In addition to scripting class attributes and inline styles, JavaScript can also manipulate stylesheets themselves. Stylesheets are associated with an HTML document with a <style> tag or with a <link rel="stylesheet"> tag. Both of these are regular HTML tags, so you can give them both id attributes and then look them up with document.querySelector(). The Element objects for both <style> and <link> tags have a disabled property that you can use to disable the entire stylesheet. You might use it with code like this:

function toggleTheme() {

let lightTheme = document.querySelector("#light-theme");

let darkTheme = document.querySelector("#dark-theme");

if (darkTheme.disabled) {

lightTheme.disabled = true; darkTheme.disabled = false; }

else { lightTheme.disabled = false; darkTheme.disabled = true; } }

Another simple way to script stylesheets is to insert new ones into the document using DOM manipulation techniques we’ve already seen. For example:

function setTheme(name) {

let link = document.createElement("link");

link.id = "theme";

link.rel = "stylesheet";

link.href = `themes/${name}.css`;

let currentTheme = document.querySelector("#theme");

if (currentTheme) currentTheme.replaceWith(link);

else document.head.append(link); }

Less subtly, you can also just insert a string of HTML containing a <style> tag into your document. This is a fun trick, for example:

document.head.insertAdjacentHTML( "beforeend",

"<style>body{transform:rotate(180deg)}</style>" );

1. *CSS Animations and Events* -- Suppose you have the following two CSS classes defined in a stylesheet:

.transparent { opacity: 0; }

.fadeable { transition: opacity .5s ease-in }

If you apply the first style to an element, it will be fully transparent and therefore invisible. But if you apply the second style that tells the browser that when the opacity of the element changes, that change should be animated over a period of 0.5 seconds, “ease-in” specifies that the opacity change animation should start off slow and then accelerate. Now suppose that your HTML document contains an element with the “fadeable” class:

<div id="subscribe" class="fadeable notification">...</div>

In JavaScript, you can add the “transparent” class:

document.querySelector("#subscribe").classList.add("transparent");

This element is configured to animate opacity changes. Adding the “transparent” class changes the opacity and triggers an animate: the browser “fades out” the element so that it becomes fully transparent over the period of half a second. This works in reverse as well. JavaScript can also be used to monitor the progress of a CSS transition because the web browser fires events at the start and end of a transition

1. *Document Coordinates and Viewport Coordinates* -- The position of a document element is measured in CSS pixels, with the x coordinate increasing to the right and the y coordinate increasing as we go down. There are two different points we can use as the coordinate system origin, however: the x and y coordinates of an element can be relative to the top-left corner of the document or relative to the top-left corner of the viewport in which the document is displayed. In top-level windows and tabs, the “viewport” is the portion of the browser that actually displays document content.

*Querying the Geometry of an Element* -- You can determine the size (including CSS border and padding, but not the margin) and position (in viewport coordinates) of an element by calling its getBoundingClientRect() method. It takes no arguments and returns an object with properties left, right, top, bottom, width, and height. The left and top properties give the x and y coordinates of the upper-left corner of the element, and the right and bottom properties give the coordinates of the lower-right corner. The differences between these values are the width and height properties.

Block elements, such as images, paragraphs, and <div> elements are always rectangular when laid out by the browser. Inline elements, such as <span>, <code>, and <b> elements, however, may span multiple lines and may therefore consist of multiple rectangles. Imagine, for example, some text within <em> and </em> tags that happens to be displayed so that it wraps across two lines. Its rectangles consist of the end of the first line and beginning of the second line. If you call getBoundingClientRect() on this element, the bounding rectangle would include the entire width of both lines. If you want to query the individual rectangles of inline elements, call the getClientRects() method to obtain a read-only, array-like object whose elements are rectangle objects like those returned by getBoundingClientRect(). Sometimes we want to go in the other direction and determine which element is at a given location in the viewport. You can determine this with the elementFromPoint() method of the Document object.

1. *Scrolling* -- The scrollTo() method of the Window object takes the x and y coordinates of a point (in document coordinates) and sets these as the scrollbar offsets. That is, it scrolls the window so that the specified point is in the upper-left corner of the viewport. The following code scrolls the browser so that the bottom-most page of the document is visible:

let documentHeight = document.documentElement.offsetHeight;

let viewportHeight = window.innerHeight;

window.scrollTo(0, documentHeight - viewportHeight);

The scrollBy() method of the Window is similar to scrollTo(), but its arguments are relative and are added to the current scroll position:

// Scroll 50 pixels down every 500 ms. Note there is no way to turn this off! setInterval(() => { scrollBy(0,50)}, 500);

If you want to scroll smoothly with scrollTo() or scrollBy(), pass a single object argument instead of two numbers, like this: window.scrollTo({ left: 0, top: documentHeight - viewportHeight, behavior: "smooth" });

Often, instead of scrolling to a numeric location in a document, we just want to scroll so that a certain element in the document is visible. You can do this with the scrollIntoView() method on the desired HTML element. This method ensures that the element on which it is invoked is visible in the viewport. By default, it tries to put the top edge of the element at or near the top of the viewport. If false is passed as the only argument, it tries to put the bottom edge of the element at the bottom of the viewport. The browser will also scroll the viewport horizontally as needed to make the element visible. You can also pass an object to scrollIntoView(), setting the behavior:"smooth" property for smooth scrolling. You can set the block property to specify where the element should be positioned vertically and the inline property to specify how it should be positioned horizontally if horizontal scrolling is needed. Legal values for both of these properties are start, end, nearest, and center.

*Custom Elements* -- The second web browser feature that enables web components is “custom elements”: the ability to associate a JavaScript class with an HTML tag name so that any such tags in the document are automatically turned into instances of the class in the DOM tree. The customElements.define() method takes a web component tag name as its first argument (remember that the tag name must include a hyphen) and a subclass of HTMLElement as its second argument. Any existing elements in the document with that tag name are “upgraded” to newly created instances of the class. And if the browser parses any HTML in the future, it will automatically create an instance of the class for each of the tags it encounters. The class passed to customElements.define() should extend HTMLElement.

1. *Nodejs* -- You can then execute the program and see output like this: $ node --trace-uncaught argv.js --arg1 --arg2 filename

[ '/usr/local/bin/node', '/private/tmp/argv.js', '--arg1', '--arg2', 'filename' ]

There are a couple of things to note here:

* The first and second elements of process.argv will be fully qualified filesystem paths to the Node executable and the file of JavaScript that is being executed, even if you did not type them that way.
* Command-line arguments that are intended for and interpreted by the Node executable itself are consumed by the Node executable and do not appear in process.argv. (The -- trace-uncaught command-line argument isn’t actually doing anything useful in the previous example; it is just there to demonstrate that it does not appear in the output.) Any arguments (such as --arg1 and filename) that appear after the name of the JavaScript file will appear in process.argv.

A program can force itself to exit by calling process.exit(). Users can usually terminate a Node program by typing Ctrl-C in the terminal window where the program is running. A program can ignore Ctrl-C by registering a signal handler function with process.on("SIGINT" , ()=>{}).

If you don’t want these exceptions to cause your program to completely crash, register a global handler function that will be invoked instead of crashing: process.setUncaughtExceptionCaptureCallback(e => { console.error("Uncaught exception:", e); });

If you do not want unhandled rejections, to print error messages or terminate your program, register a global handler function:

process.on("unhandledRejection", (reason, promise) => { // reason is whatever value would have been passed to a .catch() function // promise is the Promise object that rejected });

1. *Node Modules* -- Node 13 adds support for standard ES6 modules as well as requirebased modules (which Node calls “CommonJS modules”). The two module systems are not fully compatible, so this is somewhat tricky to do. Node needs to know—before it loads a module—whether that module will be using require() and module.exports or if it will be using import and export. When Node loads a file of JavaScript code as a CommonJS module, it automatically defines the require() function along with identifiers exports and module, and it does not enable the import and export keywords.

On the other hand, when Node loads a file of code as an ES6 module, it must enable the import and export declarations, and it must not define extra identifiers like require, module, and exports. The simplest way to tell Node what kind of module it is loading is to encode this information in the file extension. If you save your JavaScript code in a file that ends with .mjs, then Node will always load it as an ES6 module, will expect it to use import and export, and will not provide a require() function. And if you save your code in a file that ends with .cjs, then Node will always treat it as a CommonJS module, will provide a require() function, and will throw a SyntaxError if you use import or export declarations. For files that do not have an explicit .mjs or .cjs extension, Node looks for a file named package.json in the same directory as the file and then in each of the containing directories. Once the nearest package.json file is found, Node checks for a top-level type property in the JSON object. If the value of the type property is “module”, then Node loads the file as an ES6 module. If the value of that property is “commonjs”, then Node loads the file as a CommonJS module.

Note that you do not need to have a package.json file to run Node programs: when no such file is found (or when the file is found but it does not have a type property), Node defaults to using CommonJS modules. This package.json trick only becomes necessary if you want to use ES6 modules with Node and do not want to use the .mjs file extension. Because there is an enormous amount of existing Node code written using CommonJS module format, Node allows ES6 modules to load CommonJS modules using the import keyword. The reverse is not true, however: a CommonJS module cannot use require() to load an ES6 module.

1. *Node Is Asynchronous by Default* -- Node achieves high levels of concurrency while maintaining a singlethreaded programming model by making its API asynchronous and nonblocking by default. Node takes its nonblocking approach very seriously and to an extreme that may surprise you. You probably expect functions that read from and write to the network to be asynchronous, but Node goes further and defines nonblocking asynchronous functions for reading and writing files from the local filesystem. Node takes it still further: the default functions for initiating a network connection or looking up a file modification time, for example, are also nonblocking.

Now that we’ve discussed Node’s aggressively nonblocking API, let’s turn back to the topic of concurrency. Node’s built-in nonblocking functions work using the operating system’s version of callbacks and event handlers. When you call one of these functions, Node takes action to get the operation started, then registers some kind of event handler with the operating system so that it will be notified when the operation is complete. The callback you passed to the Node function gets stored internally so that Node can invoke your callback when the operating system sends the appropriate event to Node.

This kind of concurrency is often called event-based concurrency. At its core, Node has a single thread that runs an “event loop.” When a Node program starts, it runs whatever code you’ve told it to run. This code presumably calls at least one nonblocking function causing a callback or event handler to be registered with the operating system. (If not, then you’ve written a synchronous Node program, and Node simply exits when it reaches the end.) When Node reaches the end of your program, it blocks until an event happens, at which time the OS starts it running again. Node maps the OS event to the JavaScript callback you registered and then invokes that function. Your callback function may invoke more nonblocking Node functions, causing more OS event handlers to be registered. Once your callback function is done running, Node goes back to sleep again and the cycle repeats.

For web servers and other I/O-intensive applications that spend most of their time waiting for input and output, this style of event-based concurrency is efficient and effective. A web server can concurrently handle requests from 50 different clients without needing 50 different threads as long as it uses nonblocking APIs and there is some kind of internal mapping from network sockets to JavaScript functions to invoke when activity occurs on those sockets.

1. *Buffers* -- One of the datatypes you’re likely to use frequently in Node— especially when reading data from files or from the network—is the Buffer class. A Buffer is a lot like a string, except that it is a sequence of bytes instead of a sequence of characters. Node was created before core JavaScript supported typed arrays and there was no Uint8Array to represent an array of unsigned bytes. Node defined the Buffer class to fill that need. Now that Uint8Array is part of the JavaScript language, Node’s Buffer class is a subclass of Uint8Array. What distinguishes Buffer from its Uint8Array superclass is that it is designed to interoperate with JavaScript strings: the bytes in a buffer can be initialized from character strings or converted to character strings. A character encoding maps each character in some set of characters to an integer. Given a string of text and a character encoding, we can encode the characters in the string into a sequence of bytes. And given a (properly encoded) sequence of bytes and a character encoding, we can decode those bytes into a sequence of characters. Node’s Buffer class has methods that perform both encoding and decoding, and you can recognize these methods because they expect an encoding argument that specifies the encoding to be used. Encodings in Node are specified by name, as strings. The supported encodings are: "utf8", "utf16le", "latin1", "ascii", "base64".
2. Streams -- Node supports four basic stream types:

* Readable -- Readable streams are sources of data. The stream returned by fs.createReadStream(), for example, is a stream from which the content of a specified file can be read. process.stdin is another Readable stream that returns data from standard input.
* Writable -- Writable streams are sinks or destinations for data. The return value of fs.createWriteStream(), for example, is a Writable stream: it allows data to be written to it in chunks, and outputs all of that data to a specified file.
* Duplex -- Duplex streams combine a Readable stream and a Writable stream into one object. The Socket objects returned by net.connect() and other Node networking APIs, for example, are Duplex streams. If you write to a socket, your data is sent across the network to whatever computer the socket is connected to. And if you read from a socket, you access the data written by that other computer.
* Transform -- Transform streams are also readable and writable, but they differ from Duplex streams in an important way: data written to a Transform stream becomes readable—usually in some transformed form—from the same stream. The zlib.createGzip() function, for example, returns a Transform stream that compresses (with the gzip algorithm) the data written to it. In a similar way, the crypto.createCipheriv() function returns a Transform stream that encrypts or decrypts data that is written to it.