
CHAPTER 22

HTML5 APIs

The term HTML5 refers to the latest version of the HTML specification, of course, but it has also come to refer to an entire suite of web application technologies that are being developed and specified as part of or alongside HTML. A more formal term for these technologies is the Open Web Platform. In practice, however, “HTML5” is a convenient shorthand, and this chapter uses it in that way. Some of the new HTML5 APIs are documented elsewhere in this book:

- Chapter 15 covers the `getElementsByClassName()` and `querySelectorAll()` methods and the `dataset` attribute of document elements.
- Chapter 16 covers the `classList` property of elements.
- Chapter 18 covers XMLHttpRequest Level 2, cross-origin HTTP requests, and the EventSource API defined by the Server-Sent Events specification.
- Chapter 20 documents the Web Storage API and the application cache for offline web apps.
- Chapter 21 covers the `<audio>`, `<video>`, and `<canvas>` elements and SVG graphics.

This chapter covers a number of other HTML5 APIs:

- §22.1 covers the Geolocation API, which allows browsers to (with permission) determine the user’s physical location.
- §22.2 covers history management APIs that allow web applications to save and update their state in response to the browser’s Back and Forward buttons without having to reload themselves from the web server.
- §22.3 describes a simple API for passing messages between documents with different origins. This API safely works around the same-origin security policy (§13.6.2) that prevents documents from different web servers from interacting directly with each other.
- §22.4 covers a major new feature of HTML5: the ability to run JavaScript code in an isolated background thread and to safely communicate with those “worker” threads.

- §22.5 describes special-purpose memory-efficient types for working with arrays of bytes and numbers.
- §22.6 covers Blobs: opaque chunks of data that serve as the central data exchange format for a variety of new binary data APIs. This section also covers a number of Blob-related types and APIs: File and FileReader objects, the BlobBuilder type, and Blob URLs.
- §22.7 demonstrates the Filesystem API by which web applications can read and write files within a private sandboxed filesystem. This is one of the APIs that is still in flux and is not documented in the reference section.
- §22.8 demonstrates the IndexedDB API for storing and retrieving objects in simple databases. Like the Filesystem API, IndexedDB is unstable and is not documented in the reference section.
- Finally, §22.9 covers the Web Sockets API that allows web applications to connect to servers using bidirectional stream-based networking instead of the stateless request/response networking model supported by XMLHttpRequest.

The features documented in this chapter either do not fit naturally into any of the previous chapters or are not yet stable and mature enough to integrate into the main chapters of the book. Some of the APIs seem stable enough to document in the reference section, while in other cases the API is still in flux and is not covered in Part IV. All but one of the examples in this chapter (Example 22-9), worked in at least one browser when this book went to press. Because the specifications being covered here are still evolving, some of these examples may no longer work when you read this chapter.

22.1 Geolocation

The Geolocation API (<http://www.w3.org/TR/geolocation-API/>) allows JavaScript programs to ask the browser for the user's real-world location. Location-aware applications can display maps, directions, and other information relevant to the user's current position. There are, of course, significant privacy concerns here, and browsers that support the Geolocation API always ask the user before allowing a JavaScript program to access the user's physical location.

Browsers that support the Geolocation API define `navigator.geolocation`. This property refers to an object with three methods:

`navigator.geolocation.getCurrentPosition()`

Request the user's current position.

`navigator.geolocation.watchPosition()`

Request the current position, but also continue to monitor position and invoke the specified callback when the user's position changes.

`navigator.geolocation.clearWatch()`

Stop watching the user's location. The argument to this method should be the number returned by the corresponding call to `watchPosition()`.

In devices that include GPS hardware, very precise location information can be obtained from the GPS unit. More commonly, however, location information comes via the Web. If a browser submits your Internet IP address to a web service, it can usually determine (based on ISP records) what city you are in (and it is common for advertisers to do this on the server side). A browser can often obtain an even more precise location by asking the operating system for the list of nearby wireless networks and their signal strengths. This information, when submitted to a sophisticated web service allows your location to be computed with surprising accuracy (usually within one city block).

These geolocation technologies involve either an exchange over the network or communication with multiple satellites, so the Geolocation API is asynchronous: `getCurrentPosition()` and `watchPosition()` return immediately but accept a callback argument that the browser invokes when it has determined the user's position (or when the position has changed). The simplest form of location request looks like this:

```
navigator.geolocation.getCurrentPosition(function(pos) {
    var latitude = pos.coords.latitude;
    var longitude = pos.coords.longitude;
    alert("Your position: " + latitude + ", " + longitude);
});
```

In addition to latitude and longitude, every successful geolocation request also returns an accuracy value (in meters) that specifies how closely the position is known. Example 22-1 demonstrates: it calls `getCurrentPosition()` to determine the current position and uses the resulting information to display a map (from Google Maps) of the current location, zoomed approximately to the location accuracy.

Example 22-1. Using geolocation to display a map

```
// Return a newly created <img> element that will (once geolocation succeeds)
// be set to display a Google map of the current location. Note that the caller
// must insert the returned element into the document in order to make it
// visible. Throws an error if geolocation is not supported in the browser
function getmap() {
    // Check for geolocation support
    if (!navigator.geolocation) throw "Geolocation not supported";

    // Create a new <img> element, start a geolocation request to make the img
    // display a map of where we are, and then return the image.
    var image = document.createElement("img");
    navigator.geolocation.getCurrentPosition(setMapURL);
    return image;

    // This function will be invoked after we return the image object, when
    // (and if) the geolocation request succeeds.
    function setMapURL(pos) {
        // Get our position information from the argument object
        var latitude = pos.coords.latitude;    // Degrees N of equator
        var longitude = pos.coords.longitude;  // Degrees E of Greenwich
        var accuracy = pos.coords.accuracy;   // Meters

        // Construct a URL for a static Google map image of this location
        var url = "http://maps.google.com/maps/api/staticmap" +
```

```

        "?center=" + latitude + "," + longitude +
        "&size=640x640&sensor=true";

    // Set the map zoom level using a rough heuristic
    var zoomlevel=20;    // Start zoomed in almost all the way
    if (accuracy > 80)    // Zoom out for less accurate positions
        zoomlevel -= Math.round(Math.log(accuracy/50)/Math.LN2);
    url += "&zoom=" + zoomlevel;    // Add zoom level to the URL

    // Now display the map in the image object. Thanks, Google!
    image.src = url;
}
}

```

The Geolocation API has several features that are not demonstrated by Example 22-1:

- In addition to the first callback argument, `getCurrentPosition()` and `watchPosition()` accept an optional second callback that is invoked if the geolocation request fails.
- In addition to the success and error callbacks, those two methods also accept an options object as an optional third argument. The properties of this object specify whether a high accuracy position is desired, how “stale” the position is allowed to be, and how long the system is allowed to take to determine the position.
- The object passed to the success callback also includes a timestamp and may (on some devices) include additional information such as altitude, speed, and heading.

Example 22-2 demonstrates these additional features.

Example 22-2. A demonstration of all geolocation features

```

// Determine my location asynchronously and display it in the specified element.
function whereami(elt) {
    // Pass this object as the 3rd argument to getCurrentPosition()
    var options = {
        // Set to true to get a higher accuracy reading (from GPS, for example)
        // if available. Note, however that this can affect battery life.
        enableHighAccuracy: false, // Approximate is okay: this is the default

        // Set this property if a cached location is good enough.
        // The default is 0, which forces location to be checked anew.
        maximumAge: 300000, // A fix from the last 5 minutes is okay

        // How long are you willing to wait to get the location?
        // The default is Infinity and getCurrentPosition() never times out
        timeout: 15000 // Don't take more than 15 seconds
    };

    if (navigator.geolocation) // Request position, if supported
        navigator.geolocation.getCurrentPosition(success, error, options);
    else
        elt.innerHTML = "Geolocation not supported in this browser";

    // This function will be invoked if geolocation fails
}

```

```
function error(e) {
    // The error object has a numeric code and a text message. Code values:
    // 1: the user did not give permission to share his or her location
    // 2: the browser was unable to determine the position
    // 3: a timeout occurred
    elt.innerHTML = "Geolocation error " + e.code + "; " + e.message;
}

// This function will be invoked if geolocation succeeds
function success(pos) {
    // These are the fields that we always get. Note that the timestamp
    // is in the outer object, not the inner, coords object.
    var msg = "At " +
        new Date(pos.timestamp).toLocaleString() + " you were within " +
        pos.coords.accuracy + " meters of latitude " +
        pos.coords.latitude + " longitude " +
        pos.coords.longitude + ".";

    // If our device returns altitude, add that information.
    if (pos.coords.altitude) {
        msg += " You are " + pos.coords.altitude + " ± " +
            pos.coords.altitudeAccuracy + "meters above sea level.";
    }

    // if our device returns speed and heading, add that, too.
    if (pos.coords.speed) {
        msg += " You are travelling at " +
            pos.coords.speed + "m/s on heading " +
            pos.coords.heading + ".";
    }

    elt.innerHTML = msg; // Display all the position information
}
}
```

22.2 History Management

Web browsers keep track of what documents have been loaded into a window and display Back and Forward buttons that allow the user to navigate among those documents. This browser history model dates back to the days in which documents were passive and all computation was performed on the server. Today, web applications often generate or load content dynamically and display new application states without performing new document loads. Applications like these must perform their own history management if they want to user to be able to use the Back and Forward buttons to navigate from one application state to another in an intuitive way. HTML5 defines two mechanisms for history management.

The simpler history management technique involves `location.hash` and the `hashchange` event. This technique is also somewhat more widely implemented at the time of this writing: browsers were beginning to implement it even before HTML5 standardized it. In most browsers (but not older versions of IE), setting the `location.hash`

property updates the URL displayed in the location bar and adds an entry to the browser's history. The hash property sets the fragment identifier of the URL and is traditionally used to specify the ID of a document section to scroll to. But `location.hash` does not have to be an element ID: you can set it to any string. If you can encode your application state as a string, you can use that string as a fragment identifier.

By setting the `location.hash` property, then, you allow the user to use the Back and Forward buttons to navigate between document states. For this to work, your application must have some way to detect these changes of state, so that it can read the state stored in the fragment identifier and update itself accordingly. In HTML5, the browser fires a `hashchange` event at the Window whenever the fragment identifier changes. In browsers that support the `hashchange` event, you can set `window.onhashchange` to a handler function that will be called whenever the fragment identifier changes as a result of history navigation. When this handler function is called, your function would parse the `location.hash` value and redisplay the application using the state information it contains.

HTML5 also defines a somewhat more complex and robust method of history management involving the `history.pushState()` method and the `popstate` event. When a web app enters a new state, it calls `history.pushState()` to add that state to the browsing history. The first argument is an object that contains all the state information necessary to restore the current state of the document. Any object that can be converted to a string with `JSON.stringify()` will work, and certain other native types such as `Date` and `RegExp` should also work as well (see the sidebar below). The second argument is an optional title (a plain text string) that the browser can use (in a <Back> menu, for example) to identify the saved state in the browsing history. The third argument is an optional URL that will be displayed as the location of the current state. Relative URLs are resolved against the current location of the document, and it is common to simply specify a hash (or “fragment identifier”) portion of the URL, such as `#state`. Associating a URL with each state allows the user to bookmark internal states of your application, and if you include sufficient information in the URL, your application can restore its state when loaded from a bookmark.

Structured Clones

As noted above, the `pushState()` method accepts a state object and makes a private copy of it. This is a deep copy or deep clone of the object: it recursively copies the contents of any nested objects or arrays. The HTML5 standard calls this kind of copy a *structured clone*. The process of creating a structured clone is something like passing the object to `JSON.stringify()` and then passing the resulting string to `JSON.parse()` (see §6.9). But JSON only supports JavaScript primitives plus objects and arrays. The HTML5 standard says that the structured clone algorithm must also be able to clone `Date` and `RegExp` objects, `ImageData` objects (from the <canvas> element: see §21.4.14), and `FileList`, `File`, and `Blob` objects (described in §22.6). JavaScript functions and errors are explicitly excluded from the structured clone algorithm, as are most host objects such as windows, documents, elements, and so on.

You may not have any reason to store files or image data as part of your history state, but structured clones are also used by a number of other HTML5-related standards, and we'll see them again throughout this chapter.

In addition to the `pushState()` method, the History object also defines `replaceState()`, which takes the same arguments but replaces the current history state instead of adding a new state to the browsing history.

When the user navigates to saved history states using the Back or Forward buttons, the browser fires a `popstate` event on the Window object. The event object associated with the event has a property named `state`, which contains a copy (another structured clone) of the state object you passed to `pushState()`.

Example 22-3 is a simple web application—the number guessing game pictured in Figure 22-1—that uses these HTML5 techniques to save its history, allowing the user to “go back” to review or redo her guesses.

As this book goes to press, Firefox 4 has made two modifications to the History API that other browsers may follow. First, Firefox 4 makes the current state available through the `state` property of the History object itself, which means that newly loaded pages do not need to wait for a `popstate` event. Second, Firefox 4 no longer fires a `popstate` event for newly loaded pages that do not have any saved state. This second change means that the example below does not work quite right in Firefox 4.

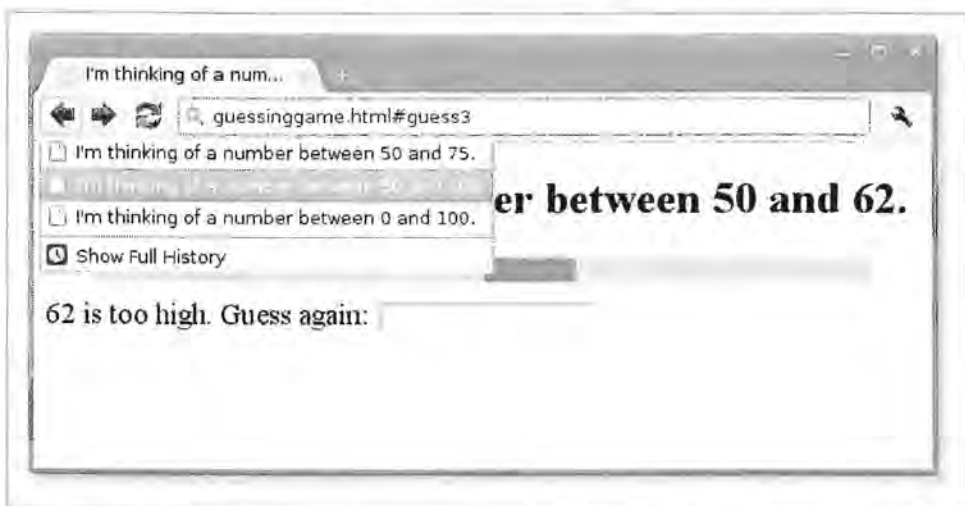


Figure 22-1. A number guessing game

Example 22-3. History management with `pushState()`

```
<!DOCTYPE html>
<html><head><title>I'm thinking of a number...</title>
<script>
```

```

window.onload = newgame;           // Start a new game when we load
window.onpopstate = popState;      // Handle history events
var state, ui;                     // Globals initialized in newgame()

function newgame(playagain) {      // Begin a new game of guess-the-number
    // Set up an object to hold document elements we care about
    ui = {
        heading: null, // The <h1> at the top of the document.
        prompt: null,  // Ask the user to enter a guess.
        input: null,   // Where the user enters the guess.
        low: null,     // Three table cells for the visual representation
        mid: null,     // ...of the range of numbers to guess.
        high: null
    };
    // Look up each of these element ids
    for(var id in ui) ui[id] = document.getElementById(id);

    // Define an event handler for the input field
    ui.input.onchange = handleGuess;

    // Pick a random number and initialize game state
    state = {
        n: Math.floor(99 * Math.random()) + 1, // An integer: 0 < n < 100
        low: 0,                               // The lower bound (exclusive) on guesses
        high: 100,                             // The upper bound (exclusive) on guesses
        guessnum: 0,                           // How many guesses have been made
        guess: undefined                       // What the last guess was
    };

    // Modify document content to display this initial state
    display(state);

    // This function is called as the onload event handler, and is also called
    // by the Play Again button displayed at the end of a game. The playagain
    // argument will be true in that second case. If it is true, then we save
    // the new game state. But if we were called in response to a load event,
    // we don't save the state. This is because load events will also occur
    // when we step backwards through the browser history from some other
    // document into the existing state of a game. If we were to save a new
    // initial state, in that case we would overwrite the actual historical
    // state of the game. In browsers that support pushState(), the load event
    // is always followed by a popstate event. So rather than saving state here,
    // we wait for the popstate. If it gives us a state object, we just use
    // that. Otherwise, if the popstate has a null state, we know this is
    // really a new game and we use replaceState to save the new game state.
    if (playagain === true) save(state);
}

// Save game state into browser history with pushState(), if it is supported
function save(state) {
    if (!history.pushState) return; // Do nothing if pushState() not defined

    // We'll associate a URL with the saved state. This URL displays the
    // guess number, but does not encode the game state, so it is not useful
    // to bookmark. We can't easily put game state in the URL because it would
    // make the secret number visible in the location bar.

```

```

var url = "#guess" + state.guessnum;
// Now save the state object and the URL
history.pushState(state, // State object to save
    "", // State title: current browsers ignore this
    url); // State URL: not useful to bookmark
}

// This is the onpopstate event handler that restores historical states.
function popState(event) {
    if (event.state) { // If the event has a state object, restore that state
        // Note that event.state is a deep copy of the saved state object
        // so we can modify it without altering the saved value.
        state = event.state; // Restore the historical state
        display(state);      // Display the restored state
    }
    else {
        // When we load the page for the first time, we'll get a popstate event
        // with no state. Replace that null state with our real state: see the
        // comment in newgame(). No need to call display() here.
        history.replaceState(state, "", "#guess" + state.guessnum);
    }
};

// This event handler is invoked each time the user guesses a number.
// It updates the game state, saves it, and displays it.
function handleGuess() {
    // Get the user's guess from the input field
    var g = parseInt(this.value);
    // If it is a number and is in the right range
    if ((g > state.low) && (g < state.high)) {
        // Update the state object based on this guess
        if (g < state.n) state.low = g;
        else if (g > state.n) state.high = g;
        state.guess = g;
        state.guessnum++;
        // Now save the new state in the browser's history
        save(state);
        // Modify the document to respond to the user's guess
        display(state);
    }
    else { // An invalid guess: don't push a new history state
        alert("Please enter a number greater than " + state.low +
            " and less than " + state.high);
    }
}

// Modify the document to display the current state of the game.
function display(state) {
    // Display document heading and title
    ui.heading.innerHTML = document.title =
        "I'm thinking of a number between " +
        state.low + " and " + state.high + ".";

    // Display a visual representation of the range of numbers using a table
    ui.low.style.width = state.low + "%";
    ui.mid.style.width = (state.high-state.low) + "%";
}

```

```

    ui.high.style.width = (100-state.high) + "%";

    // Make sure the input field is visible, empty, and focused
    ui.input.style.visibility = "visible";
    ui.input.value = "";
    ui.input.focus();

    // Set the prompt based on the user's most recent guess
    if (state.guess === undefined)
        ui.prompt.innerHTML = "Type your guess and hit Enter: ";
    else if (state.guess < state.n)
        ui.prompt.innerHTML = state.guess + " is too low. Guess again: ";
    else if (state.guess > state.n)
        ui.prompt.innerHTML = state.guess + " is too high. Guess again: ";
    else {
        // When correct, hide the input field and show a Play Again button.
        ui.input.style.visibility = "hidden"; // No more guesses now
        ui.heading.innerHTML = document.title = state.guess + " is correct! ";
        ui.prompt.innerHTML =
            "You Win! <button onclick='newgame(true)''>Play Again</button>";
    }
}
</script>
<style> /* CSS styles to make the game look good */
#prompt { font-size: 16pt; }
table { width: 90%; margin: 10px; margin-left: 5%; }
#low, #high { background-color: lightgray; height: 1em; }
#mid { background-color: green; }
</style>
</head>
<body><!-- The HTML elements below are the game UI -->
<!-- Game title and textual representation of the range of numbers -->
<h1 id="heading">I'm thinking of a number...</h1>
<!-- a visual representation of the numbers that haven't been ruled out -->
<table><tr><td id="low"></td><td id="mid"></td><td id="high"></td></tr></table>
<!-- Where the user enters their guess -->
<label id="prompt"></label><input id="input" type="text">
</body></html>

```

22.3 Cross-Origin Messaging

As noted in §14.8, some browser windows and tabs are completely isolated from each other, and the code running in one is completely unaware of the others. In other cases, when a script explicitly opens new windows or works with nested frames, the multiple windows and frames are aware of each other. If they contain documents from the same web server, scripts in these windows and frames can interact with each other and manipulate each other's documents.

Sometimes, however, a script can refer to another Window object, but because the content in that window is from a different origin, the web browser (following the same-origin policy) will not allow the script to see the document content of that other window. For the most part, the browser won't allow the script to read properties or invoke

methods of that other window, either. One window method that scripts from different origins *are* allowed to invoke is called `postMessage()`, and this method enables a limited kind of communication—in the form of asynchronous message passing—between scripts from different origins. This kind of communication is defined by HTML5 and is implemented by all current browsers (including IE8 and later). The technique is known as “cross-document messaging,” but since the API is defined on the Window object instead of the document, it might be better known as “inter-window message passing” or “cross-origin messaging.”

The `postMessage()` method expects two arguments. The first is the message to be sent. The HTML5 specification says that this can be any primitive value or object that can be cloned (see “Structured Clones” on page 672), but some current browser implementations (including Firefox 4 beta) expect strings, so if you want to pass an object or array as a message you should serialize it with `JSON.stringify()` (§6.9) first.

The second argument is a string that specifies the expected origin of the destination window. Include the protocol, hostname, and (optionally) the port portions of a URL (you can pass a complete URL, but anything other than the protocol, host, and port will be ignored). This is a security feature: malicious code or ordinary users can navigate windows to new documents that you don’t expect, so `postMessage()` won’t deliver your message if the window contains a document from a different origin than the one you specified. If the message you are passing doesn’t contain any sensitive information and you are willing to pass it to code from any origin, you can pass the string “*” as a wildcard instead. If you want to specify the same origin as the current window, you can simply use “/”.

If the origins match, the call to `postMessage()` will result in a message event being fired at the target Window object. A script in that window can define an event handler function to be notified of message events. This handler is passed an event object with the following properties:

data

This is a copy of the message that was passed as the first argument to `postMessage()`.

source

The Window object from which the message was sent.

origin

A string that specifies the origin (as a URL) from which the message was sent.

Most `onmessage()` handlers should first check the `origin` property of their argument and should ignore messages from unexpected domains.

Cross-origin messaging via `postMessage()` and the message event can be useful when you want to include a module or “gadget” from another site within your web page. If the gadget is simple and self-contained, you can simply isolate it in an `<iframe>`. Suppose, however, that it is a more complex gadget that defines an API and your web page has to control it or interact with it somehow. If the gadget is defined as a `<script>` element, it can expose a normal JavaScript API, but including in your page allows it to

take complete control of the page and its content. It is not uncommon to do this on the Web today (particularly for web advertising), but it is not really a good idea, even when you trust the other site.

Cross-origin messaging provides an alternative: the gadget author can package the gadget within an HTML file that listens for message events and dispatches those events to the appropriate JavaScript functions. Then the web page that includes the gadget can interact with it by sending messages with `postMessage()`. Examples 22-4 and 22-5 demonstrate this. Example 22-4 is a simple gadget, included via `<iframe>`, that searches Twitter and displays tweets that match a specified search term. To make this gadget search for something, the containing page simply sends it the desired search term as a message.

Example 22-4. A Twitter search gadget, controlled by `postMessage()`

```
<!DOCTYPE html>
<!--
  This is a Twitter search gadget. Include it in any webpage, inside an
  iframe, and ask it to search for things by sending it a query string with
  postMessage(). Since it is in an <iframe> and not a <script>, it can't
  mess around with the containing document.
-->
<html>
<head>
<style>body { font: 9pt sans-serif; }</style>
<!-- Use jQuery for its jQuery.getJSON() utility -->
<script src="http://code.jquery.com/jquery-1.4.4.min.js"/></script>
<script>
// We ought to just be able to use window.onmessage, but some older browsers
// (e.g., Firefox 3) don't support it, so we do it this way instead.
if (window.addEventListener)
  window.addEventListener("message", handleMessage, false);
else
  window.attachEvent("onmessage", handleMessage); // For IE8

function handleMessage(e) {
  // We don't care what the origin of this message is: we're willing
  // to search Twitter for anyone who asks us. We do expect the message
  // to come from the window that contains us, however.
  if (e.source !== window.parent) return;

  var searchterm = e.data; // This is what we were asked to search for

  // Use jQuery Ajax utilities and the Twitter search API to find
  // tweets matching the message.
  jQuery.getJSON("http://search.twitter.com/search.json?callback=?",
    { q: searchterm },
    function(data) { // Called with request results
      var tweets = data.results;
      // Build an HTML document to display these results
      var escaped = searchterm.replace("<", "&lt;");
      var html = "<h2>" + escaped + "</h2>";
      if (tweets.length == 0) {
        html += "No tweets found";
      }
    }
  );
}
```

```

    }
    else {
        html += "<dl>"; // <dl> list of results
        for(var i = 0; i < tweets.length; i++) {
            var tweet = tweets[i];
            var text = tweet.text;
            var from = tweet.from_user;
            var tweeturl = "http://twitter.com/#!/" +
                from + "/status/" + tweet.id_str;
            html += "<dt><a target='_blank' href='" +
                tweeturl + "'>" + tweet.from_user +
                "</a></dt><dd>" + tweet.text + "</dd>";
        }
        html += "</dl>";
    }
    // Set the <iframe> document
    document.body.innerHTML = html;
});
}

$(function() {
    // Let our container know we're here and ready to search.
    // The container can't send any messages to us before it gets this message
    // from us because we won't be here to receive the message yet.
    // Normally, containers can just wait for an onload event to know that all
    // of their <iframe>s have loaded. We send this message for containers that
    // want to start searching Twitter even before they get their onload event.
    // We don't know the origin of our container, so use * so that the browser
    // will deliver it to anyone.
    window.parent.postMessage("Twitter Search v0.1", "*");
});
</script>
</head>
<body>
</body>
</html>

```

Example 22-5 is a simple JavaScript file that can be inserted into any web page that wants to use the Twitter search gadget. It inserts the gadget into the document and then adds an event handler to all links in the document so that moving the mouse over a link calls `postMessage()` on the gadget's frame to make the gadget search for the URL of the link. This allows a user to see what, if anything, people are tweeting about a website before visiting the site.

Example 22-5. Using the Twitter search gadget with `postMessage()`

```

// This file of JS code inserts the Twitter Search Gadget into the document
// and adds an event handler to all links in the document so that when the
// use moves the mouse over them, the gadget searches for the link's URL.
// This allows the user to see what people are tweeting about the link
// destination before clicking on it.
window.addEventListener("load", function() {
    var origin = "http://davidflanagan.com"; // Won't work in IE < 9
    var gadget = "/demos/TwitterSearch.html"; // Gadget origin
    var iframe = document.createElement("iframe"); // Gadget path
    // Create the iframe

```

```

iframe.src = origin + gadget;           // Set its URL
iframe.width = "250";                   // 250 pixels wide
iframe.height = "100%";                 // Full document height
iframe.style.cssFloat = "right";        // Flush right

// Insert the iframe at the start of the document
document.body.insertBefore(iframe, document.body.firstChild);

// Now find all links and hook them up to the gadget
var links = document.getElementsByTagName("a");
for(var i = 0; i < links.length; i++) {
    // addEventListener doesn't work in IE8 and before
    links[i].addEventListener("mouseover", function() {
        // Send the url as the search term, and only deliver it if the
        // iframe is still displaying a document from davidflanagan.com
        iframe.contentWindow.postMessage(this.href, origin);
    }, false);
}
}, false);

```

22.4 Web Workers

One of the fundamental features of client-side JavaScript is that it is single-threaded: a browser will never run two event handlers at the same time, and it will never trigger a timer while an event handler is running, for example. Concurrent updates to application state or to the document are simply not possible, and client-side programmers do not need to think about, or even understand, concurrent programming. A corollary is that client-side JavaScript functions must not run too long: otherwise they will tie up the event loop and the web browser will become unresponsive to user input. This is the reason that Ajax APIs are always asynchronous and the reason that client-side JavaScript cannot have a simple, synchronous `load()` or `require()` function for loading JavaScript libraries.

The Web Workers specification¹ very carefully relaxes the single-threaded requirement for client-side JavaScript. The “workers” it defines are effectively parallel threads of execution. Web workers live in a self-contained execution environment, however, with no access to the Window or Document object and can communicate with the main thread only through asynchronous message passing. This means that concurrent modifications of the DOM are still not possible, but it also means that there is now a way to use synchronous APIs and write long-running functions that do not stall the event loop and hang the browser. Creating a new worker is not a heavyweight operation like opening a new browser window, but workers are not flyweight threads either, and it does not make sense to create new workers to perform trivial operations. Complex web

1. Web workers were originally part of the HTML5 specification, but they were broken off into an independent, but closely related specification. At the time of this writing, specification drafts are available at <http://dev.w3.org/html5/workers/> and <http://whatwg.org/ww/>.

applications may find it useful to create tens of workers, but it is unlikely that an application with hundreds or thousands of workers would be practical.

As with any threading API, there are two pieces to the Web Workers specification. The first is the `Worker` object: this is what a worker looks like from the outside, to the thread that creates it. The second is the `WorkerGlobalScope`: this is the global object for a new worker, and it is what a worker thread looks like, on the inside, to itself. The subsections that follow explain both. They are followed by a section of examples.

22.4.1 Worker Objects

To create a new worker, just use the `Worker()` constructor, passing a URL that specifies the JavaScript code that the worker is to run:

```
var loader = new Worker("utils/loader.js");
```

If you specify a relative URL, it is resolved relative to the URL of the document that contains the script that called the `Worker()` constructor. If you specify an absolute URL, it must have the same origin (same protocol, host, and port) as that containing document.

Once you have a `Worker` object, you can send data to it with `postMessage()`. The value you pass to `postMessage()` will be cloned (see “Structured Clones” on page 672), and the resulting copy will be delivered to the worker via a message event:

```
loader.postMessage("file.txt");
```

Note that the `postMessage()` method of a `Worker` does not have the origin argument that the `postMessage()` method of a `Window` does (§22.3). Also, the `postMessage()` method of a `Worker` correctly clones the message in current browsers, unlike `Window.postMessage()`, which is still restricted to string messages in some important browsers.

You can receive messages from a worker by listening for message events on the `Worker` object:

```
worker.onmessage = function(e) {
    var message = e.data;           // Get message from event
    console.log("URL contents: " + message); // Do something with it
}
```

If a worker throws an exception and does not catch or handle it itself, that exception propagates as an event that you can listen for:

```
worker.onerror = function(e) {
    // Log the error message, including worker filename and line number
    console.log("Error at " + e.filename + ":" + e.lineno + ": " +
        e.message);
}
```

Like all event targets, `Worker` objects define the standard `addEventListener()` and `removeEventListener()` methods, and you can use these in place of the `onmessage` and `onerror` properties if you want to manage multiple event handlers.

The Worker object has just one other method, `terminate()`, which forces a worker thread to stop running.

22.4.2 Worker Scope

When you create a new worker with the `Worker()` constructor, you specify the URL of a file of JavaScript code. That code is executed in a new, pristine JavaScript execution environment, completely isolated from the script that created the worker. The global object for that new execution environment is a `WorkerGlobalScope` object. A `WorkerGlobalScope` is something more than the core JavaScript global object, but less than a full-blown client-side `Window` object.

The `WorkerGlobalScope` object has a `postMessage()` method and an `onmessage` event handler property that are just like those of the `Worker` object but work in the opposite direction: calling `postMessage()` inside a worker generates a message event outside the worker, and messages sent from outside the worker are turned into events and delivered to the `onmessage` handler. Note that since the `WorkerGlobalScope` is the global object for a worker, `postMessage()` and `onmessage` look like a global function and global variable to worker code.

The `close()` function allows a worker to terminate itself, and it is similar in effect to the `terminate()` method of a `Worker` object. Note, however, that there is no API on the `Worker` object to test whether a worker has closed itself, and there is no `onclose` event handler property, either. If you call `postMessage()` on a worker that has closed, your message will be discarded silently and no error will be raised. In general, if a worker is going to `close()` itself, it may be a good idea to first post some kind of “closing” message.

The most interesting global function defined by `WorkerGlobalScope` is `importScripts()`: workers use this function to load any library code they require. For example:

```
// Before we start working, load the classes and utilities we'll need
importScripts("collections/Set.js", "collections/Map.js", "utils/base64.js");
```

`importScripts()` takes one or more URL arguments, each of which should refer to a file of JavaScript code. Relative URLs are resolved relative to the URL that was passed to the `Worker()` constructor. It loads and executes these files one after the other, in the order in which they were specified. If loading a script causes a network error, or if executing throws an error of any sort, none of the subsequent scripts are loaded or executed. A script loaded with `importScripts()` can itself call `importScripts()` to load the files it depends on. Note, however, that `importScripts()` does not try to keep track of what scripts have already loaded and does nothing to prevent dependency cycles.

`importScripts()` is a synchronous function: it does not return until all of the scripts have loaded and executed. You can start using the scripts you loaded as soon as `importScripts()` returns: there is no need for a callback or event handler. Once you have internalized the asynchronous nature of client-side JavaScript, it can seem strange to go back to simple, synchronous programming again. But that is the beauty of threads: you can use a blocking function call in a worker without blocking the event loop in the

main thread, and without blocking the computations being concurrently performed in other workers.

Worker Execution Model

Worker threads run their code (and all imported scripts) synchronously from top to bottom, and then enter an asynchronous phase in which they respond to events and timers. If a worker registers an `onmessage` event handler, it will never exit as long as there is a possibility that message events will still arrive. But if a worker doesn't listen for messages, it will run until there are no further pending tasks (such as download and timers) and all task-related callbacks have been called. Once all registered callbacks have been called, there is no way a worker can begin a new task, so it is safe for the thread to exit. Imagine a worker with no `onmessage` event handler that downloads a file using `XMLHttpRequest`. If the `onload` handler for that download begins a new download or registers a timeout with `setTimeout()`, the thread has new tasks and keeps running. Otherwise, the thread exits.

Since `WorkerGlobalScope` is the global object for workers, it has all of the properties of the core JavaScript global object, such as the `JSON` object, the `isNaN()` function, and the `Date()` constructor. (Look up `Global` in the core language reference section for a complete list.) In addition, however, `WorkerGlobalScope` also has the following properties of the client-side `Window` object:

- `self` is a reference to the global object itself. Note, however, that `WorkerGlobalScope` does not have the synonymous `window` property that `Window` objects have.
- The timer methods `setTimeout()`, `clearTimeout()`, `setInterval()` and `clearInterval()`.
- A `location` property that describes the URL that was passed to the `Worker()` constructor. This property refers to a `Location` object, just as the `location` property of a `Window` does. The `Location` object has properties `href`, `protocol`, `host`, `hostname`, `port`, `pathname`, `search`, and `hash`. In a worker, these properties are read-only.
- A `navigator` property that refers to an object with properties like those of the `Navigator` object of a window. A worker's `navigator` object has properties `appName`, `appVersion`, `platform`, `userAgent`, and `onLine`.
- The usual event target methods `addEventListener()` and `removeEventListener()`.
- An `onerror` property that you can set to an error handler function like the `Window.onerror` handler described in §14.6. An error handler, if you register one, is passed the error message, URL, and line number as three string arguments. It can return `false` to indicate that the error has been handled and should not be propagated as an error event on the `Worker` object. (At the time of this writing, however, error handling within a worker is not implemented interoperably across browsers.)

Finally, the `WorkerGlobalScope` object includes important client-side JavaScript constructor objects. These include `XMLHttpRequest()` so that workers can perform scripted

HTTP (see Chapter 18) and the `Worker()` constructor so that workers can create their own worker threads. (At the time of this writing, the `Worker()` constructor is not available to workers in Chrome and Safari, however.)

A number of the HTML5 APIs described later in this chapter define features that are available through both an ordinary Window object and also in workers through the `WorkerGlobalScope`. Often, the Window object will define an asynchronous API and the `WorkerGlobalScope` will add a synchronous version of the same basic API. These “worker-enabled” APIs will be described when we come to them later in the chapter.

Advanced Worker Features

The worker threads described in this section are *dedicated workers*; they are associated with, or dedicated to, a single parent thread. The Web Workers specification defines another type of worker, the *shared worker*. As I write this, browsers do not yet implement shared workers. The intent, however, is that a shared worker is a kind of named resource that can provide a computational service to any thread that cares to connect to it. In practice, interacting with a shared worker is like communicating with a server over a network socket.

The “socket” for a shared worker is known as a `MessagePort`. `MessagePorts` define a message-passing API like we’ve seen for dedicated workers and cross-document messaging: they have a `postMessage()` method and an `onmessage` event handler attribute. HTML5 allows you to create connected pairs of `MessagePort` objects with the `MessageChannel()` constructor. You can pass `MessagePorts` (via a special `postMessage()` argument) to other windows or other workers and use them as dedicated communication channels. `MessagePorts` and `MessageChannels` are an advanced API that is not yet supported by many browsers and is not covered here.

22.4.3 Web Worker Examples

We’ll end this section with two Web Worker examples. The first demonstrates how to perform long computations in a worker thread so that they don’t affect the UI responsiveness of the main thread. The second example demonstrates how worker threads can use simpler synchronous APIs.

Example 22-6 defines a `smear()` function that expects an `<img>` element as its argument. It applies a motion blur effect to “smear” the image to the right. It uses techniques from Chapter 21 to copy the image to an offscreen `<canvas>` element and then to extract the image’s pixels to an `ImageData` object. You cannot pass an `<img>` or a `<canvas>` element to a worker via `postMessage()`, but you can pass an `ImageData` object (details are in “Structured Clones” on page 672). Example 22-6 creates a `Worker` object and calls `postMessage()` to send it the pixels to be smeared. When the worker sends the processed pixels back, the code copies them back into the `<canvas>`, extracts them as a `data://` URL, and sets that URL on the `src` property of the original `<img>` element.

Example 22-6. Creating a Web Worker for image processing

```
// Asynchronously replace the contents of the image with a smeared version.
// Use it like this: 
function smear(img) {
    // Create an offscreen <canvas> the same size as the image
    var canvas = document.createElement("canvas");
    canvas.width = img.width;
    canvas.height = img.height;

    // Copy the image into the canvas, then extract its pixels
    var context = canvas.getContext("2d");
    context.drawImage(img, 0, 0);
    var pixels = context.getImageData(0,0,img.width,img.height)

    // Send the pixels to a worker thread
    var worker = new Worker("SmearWorker.js");    // Create worker
    worker.postMessage(pixels);                  // Copy and send pixels

    // Register a handler to get the worker's response
    worker.onmessage = function(e) {
        var smeared_pixels = e.data;              // Pixels from worker
        context.putImageData(smeared_pixels, 0, 0); // Copy them to the canvas
        img.src = canvas.toDataURL();              // And then to the img
        worker.terminate();                       // Stop the worker thread
        canvas.width = canvas.height = 0;         // Don't keep pixels around
    }
}
```

Example 22-7 is the code used by the worker thread created in Example 22-6. The bulk of this example is the image processing function: a modified version of the code from Example 21-10. Note that this example sets up its message-passing infrastructure in a single line of code: the onmessage event handler simply smears the image it is passed and posts it right back.

Example 22-7. Image processing in a Web Worker

```
// Get an ImageData object from the main thread, process it, send it back
onmessage = function(e) { postMessage(smear(e.data)); }

// Smear the ImageData pixels to the right, producing a motion blur.
// For large images, this function does a lot of computation and would
// cause UI responsiveness issues if it was used on the main thread.
function smear(pixels) {
    var data = pixels.data, width = pixels.width, height = pixels.height;
    var n = 10, m = n-1; // Make n bigger for more smearing
    for(var row = 0; row < height; row++) { // For each row
        var i = row*width*4 + 4;           // 2nd pixel offset
        for(var col = 1; col < width; col++, i += 4) { // For each column
            data[i] = (data[i] + data[i-4]*m)/n; // Red pixel component
            data[i+1] = (data[i+1] + data[i-3]*m)/n; // Green
            data[i+2] = (data[i+2] + data[i-2]*m)/n; // Blue
            data[i+3] = (data[i+3] + data[i-1]*m)/n; // Alpha component
        }
    }
}
```

```

    return pixels;
}

```

Note that the code in Example 22-7 can process any number of images that are sent to it. For simplicity, however, Example 22-6 creates a new Worker object for each image it processes. To ensure that the worker does not just sit around waiting for messages, it kills the thread with `terminate()` when done.

Debugging Workers

One of the APIs not available (at least as I write this) in `WorkerGlobalScope` is the console API and its invaluable `console.log()` function. Worker threads can't log output and can't interact with the document at all, so they can be tricky to debug. If a worker throws an error, the main thread will receive an error event on the Worker object. But often, you need a way for a worker to output debugging messages that are visible in the browser's web console. One straightforward way to do this is to modify the message passing protocol you use with the worker so that the worker can send debugging messages somehow. In Example 22-6, for example, we could insert the following code at the start of the `onmessage` event handler:

```

if (typeof e.data === "string") {
    console.log("Worker: " + e.data);
    return;
}

```

With that additional code in place, the Worker thread could display debugging messages simply by passing strings to `postMessage()`.

The next example demonstrates how Web Workers allow you to write synchronous code and use it safely in client-side JavaScript. §18.1.2.1 showed how to make synchronous HTTP requests with `XMLHttpRequest`, but it warned that doing so on the main browser thread was a very bad practice. In a worker thread, however, it is perfectly reasonable to make synchronous requests, and Example 22-8 demonstrates worker code that does just that. Its `onmessage` event handler expects an array of URLs to be fetched. It uses the synchronous `XMLHttpRequest` API to fetch them, and then posts the textual content of the URLs as an array of strings back to the main thread. Or, if any of the HTTP requests fail, it throws an error that propagates to the `onerror` handler of the Worker.

Example 22-8. Making synchronous XMLHttpRequests in a Web Worker

```

// This file will be loaded with new Worker(), so it runs as an independent
// thread and can safely use the synchronous XMLHttpRequest API.
// Messages are expected to be arrays of URLs. Synchronously fetch the
// contents of each URL as a string and send back an array of those strings.
onmessage = function(e) {
    var urls = e.data;    // Our input: the URLs to fetch
    var contents = [];    // Our output: the contents of those URLs

    for(var i = 0; i < urls.length; i++) {

```

```

var url = urls[i];           // For each URL
var xhr = new XMLHttpRequest(); // Begin an HTTP request
xhr.open("GET", url, false); // false makes this synchronous
xhr.send();                 // Blocks until response is complete
if (xhr.status !== 200)      // Throw an error if request failed
    throw Error(xhr.status + " " + xhr.statusText + ": " + url);
contents.push(xhr.responseText); // Otherwise, store the URL contents
}

// Finally, send the array of URL contents back to the main thread
postMessage(contents);
}

```

22.5 Typed Arrays and ArrayBuffers

As you know from Chapter 7, JavaScript arrays are general-purpose objects with numeric properties and a special `length` property. Array elements can be any JavaScript value. Arrays can grow or shrink dynamically and can be sparse. JavaScript implementations perform lots of optimizations so that typical uses of JavaScript arrays are very fast. *Typed arrays* are array-like objects (§7.11) that differ from regular arrays in some important ways:

- The elements of a typed array are all numbers. The constructor used to create the typed array determines the type (signed or unsigned integers or floating point) and size (in bits) of the numbers.
- Typed arrays have a fixed length.
- The elements of a typed array are always initialized to 0 when the array is created.

There are eight kinds of typed arrays, each with a different element type. You can create them with the following constructors:

Constructor	Numeric type
<code>Int8Array()</code>	signed bytes
<code>Uint8Array()</code>	unsigned bytes
<code>Int16Array()</code>	signed 16-bit short integers
<code>Uint16Array()</code>	unsigned 16-bit short integers
<code>Int32Array()</code>	signed 32-bit integers
<code>Uint32Array()</code>	unsigned 32-bit integers
<code>Float32Array()</code>	32-bit floating-point value
<code>Float64Array()</code>	64-bit floating-point value: a regular JavaScript number

Typed Arrays, <canvas>, and Core JavaScript

Typed arrays are an essential part of the WebGL 3D graphics API for the <canvas> element, and browsers have implemented them as part of WebGL. WebGL is not covered in this book, but typed arrays are generally useful and are covered here. You may recall from Chapter 21 that the Canvas API defines a `getImageData()` method that returns an `ImageData` object. The `data` property of an `ImageData` is an array of bytes. The HTML standard calls this a `CanvasPixelArray`, but it is essentially the same as the `Uint8Array` described here, except for the way it handles values outside of the range 0 to 255.

Note that these types are not part of the core language. A future version of the JavaScript language is likely to include support for typed arrays like these, but at the time of this writing, it is unclear whether the language will adopt the API described here or will create a new API.

When you create a typed array, you pass the array size to the constructor or pass an array or typed array to initialize the array elements with. Once you have created a typed array, you can read and write its elements with regular square-bracket notation, just as you would with any other array-like object:

```
var bytes = new Uint8Array(1024); // One kilobyte of bytes
for(var i = 0; i < bytes.length; i++) // For each element of the array
    bytes[i] = i & 0xFF; // Set it to the low 8 bits of index
var copy = new Uint8Array(bytes); // Make a copy of the array
var ints = new Int32Array([0,1,2,3]); // A typed array holding these 4 ints
```

Modern JavaScript implementations optimize arrays to make them very efficient. Nevertheless, typed arrays can be even more efficient in both execution time and memory use. The following function computes the largest prime number less than the value you specify. It uses the Sieve of Eratosthenes algorithm, which requires a large array to keep track of which numbers are prime and which are composite. Since only a single bit of information is required for each array element, an `Int8Array` can be used more efficiently than a regular JavaScript array:

```
// Return the largest prime smaller than n, using the sieve of Eratosthenes
function sieve(n) {
    var a = new Int8Array(n+1); // a[x] will be 1 if x is composite
    var max = Math.floor(Math.sqrt(n)); // Don't do factors higher than this
    var p = 2; // 2 is the first prime
    while(p <= max) { // For primes less than max
        for(var i = 2*p; i <= n; i += p) // Mark multiples of p as composite
            a[i] = 1;
        while(a[p]) /* empty */; // The next unmarked index is prime
    }
    while(a[n]) n--; // Loop backward to find the last prime
    return n; // And return it
}
```

The `sieve()` function continues to work if you replace the `Int8Array()` constructor with the traditional `Array()` constructor, but it runs two to three times more slowly and

requires substantially more memory for large values of the parameter *n*. You might also find typed arrays useful when working with numbers for graphics or mathematics:

```
var matrix = new Float64Array(9); // A 3x3 matrix
var 3dPoint = new Int16Array(3); // A point in 3D space
var rgba = new Uint8Array(4); // A 4-byte RGBA pixel value
var sudoku = new Uint8Array(81); // A 9x9 sudoku board
```

JavaScript square-bracket notation allows you to get and set individual elements of a typed array. But typed arrays also define methods for setting and querying entire regions of the array. The `set()` method copies the elements of regular or typed arrays into a typed array:

```
var bytes = new Uint8Array(1024) // A 1K buffer
var pattern = new Uint8Array([0,1,2,3]); // An array of 4 bytes
bytes.set(pattern); // Copy them to the start of another byte array
bytes.set(pattern, 4); // Copy them again at a different offset
bytes.set([0,1,2,3], 8); // Or just copy values direct from a regular array
```

Typed arrays also have a `subarray` method that returns a portion of the array on which it is called:

```
var ints = new Int16Array([0,1,2,3,4,5,6,7,8,9]); // 10 short integers
var last3 = ints.subarray(ints.length-3, ints.length); // Last 3 of them
last3[0] // => 7: this is the same as ints[7]
```

Note that `subarray()` does not make a copy of the data. It just returns a new view of the same underlying values:

```
ints[9] = -1; // Change a value in the original array and...
last3[2] // => -1: it also changes in the subarray
```

The fact that the `subarray()` method returns a new view of an existing array reveals something important about typed arrays: they are all views on an underlying chunk of bytes known as an `ArrayBuffer`. Every typed array has three properties that relate to the underlying buffer:

```
last3.buffer // => returns an ArrayBuffer object
last3.buffer == ints.buffer // => true: both are views of the same buffer
last3.byteOffset // => 14: this view starts at byte 14 of the buffer
last3.byteLength // => 6: this view is 6 bytes (3 16-bit ints) long
```

The `ArrayBuffer` object itself has only a single property that returns its length:

```
last3.byteLength // => 6: this view is 6 bytes long
last3.buffer.byteLength // => 20: but the underlying buffer has 20 bytes
```

`ArrayBuffers` are just opaque chunks of bytes. You can access those bytes with typed arrays, but an `ArrayBuffer` is not itself a typed array. Be careful, however: you can use numeric array indexing with `ArrayBuffers` just as you can with any JavaScript object. Doing so does not give you access to the bytes in the buffer, however:

```
var bytes = new Uint8Array(8); // Allocate 8 bytes
bytes[0] = 1; // Set the first byte to 1
bytes.buffer[0] // => undefined: buffer doesn't have index 0
bytes.buffer[1] = 255; // Try incorrectly to set a byte in the buffer
```

```

bytes.buffer[1]           // => 255: this just sets a regular JS property
bytes[1]                  // => 0: the line above did not set the byte

```

You can create `ArrayBuffers` directly with the `ArrayBuffer()` constructor, and, given an `ArrayBuffer` object, you can create any number of typed array views of that buffer:

```

var buf = new ArrayBuffer(1024*1024); // One megabyte
var asbytes = new Uint8Array(buf);    // Viewed as bytes
var asints = new Int32Array(buf);     // Viewed as 32-bit signed integer
var lastK = new Uint8Array(buf,1023*1024); // Last kilobyte as bytes
var ints2 = new Int32Array(buf, 1024, 256); // 2nd kilobyte as 256 integers

```

Typed arrays allow you to view the same sequence of bytes in chunks of 8, 16, 32, or 64 bits. This exposes the “endianness”: the order in which bytes are arranged into longer words. For efficiency, typed arrays use the native endianness of the underlying hardware. On little-endian systems, the bytes of a number are arranged in an `ArrayBuffer` from least significant to most significant. On big-endian platforms, the bytes are arranged from most significant to least significant. You can determine the endianness of the underlying platform with code like this:

```

// If the integer 0x00000001 is arranged in memory as 01 00 00 00, then
// we're on a little endian platform. On a big-endian platform we'd get
// get bytes 00 00 00 01 instead.
var little_endian = new Int8Array(new Int32Array([1]).buffer)[0] === 1;

```

Today, the most common CPU architectures are little-endian. Many network protocols, however, and some binary file formats, require big-endian byte ordering. In §22.6, you'll learn how you can use `ArrayBuffers` to hold bytes read from files or downloaded from the network. When you do this, you can't just assume that the platform endianness matches the byte order of the data. In general, when working with external data, you can use `Int8Array` and `Uint8Array` to view the data as an array of individual bytes, but you should not use the other typed arrays with multibyte word sizes. Instead, you can use the `DataView` class, which defines methods for reading and writing values from an `ArrayBuffer` with explicitly specified byte ordering:

```

var data; // Assume this is an ArrayBuffer from the network
var view = DataView(data); // Create a view of it
var int = view.getInt32(0); // Big-endian 32-bit signed int from byte 0
int = view.getInt32(4,false); // Next 32-bit int is also big-endian
int = view.getInt32(8,true) // Next 4 bytes as a little-endian signed int
view.setInt32(8,int,false); // Write it back in big-endian format

```

`DataView` defines eight `get` methods for each of the eight typed array formats. They have names like `getInt16()`, `getUint32()`, and `getFloat64()`. The first argument is the byte offset within the `ArrayBuffer` at which the value begins. All of these getter methods, other than `getInt8()` and `getUint8()`, accept an optional boolean value as their second argument. If the second argument is omitted or is false, big-endian byte ordering is used. If the second argument is true, little-endian ordering is used.

`DataView` defines eight corresponding `set` methods that write values into the underlying `ArrayBuffer`. The first argument is the offset at which the value begins. The second argument is the value to write. Each of the methods, except `setInt8()` and

`setUint8()`, accepts an optional third argument. If the argument is omitted or is `false`, the value is written in big-endian format with most significant byte first. If the argument is `true`, the value is written in little-endian format with the least significant byte first.

22.6 Blobs

A Blob is an opaque reference to, or handle for, a chunk of data. The name comes from SQL databases, where it means “Binary Large Object.” In JavaScript, Blobs often represent binary data, and they can be large, but neither is required: a Blob could also represent the contents of a small text file. Blobs are opaque: all you can do with them directly is determine their size in bytes, ask for their MIME type, and chop them up into smaller Blobs:

```
var blob = ...    // We'll see how to obtain a Blob later
blob.size        // Size of the Blob in bytes
blob.type        // MIME type of the Blob, or "" if unknown
var subblob = blob.slice(0,1024, "text/plain"); // First 1K of the Blob as text
var last = blob.slice(blob.size-1024, 1024);    // Last 1K of the Blob, untyped
```

The web browser can store Blobs in memory or on disk, and Blobs can represent really enormous chunks of data (such as video files) that are too large to fit in main memory without first being broken into smaller pieces with `slice()`. Because Blobs can be so large and may require disk access, the APIs that work with them are asynchronous (with synchronous versions available for use by worker threads).

Blobs are not terribly interesting by themselves, but they serve as a critical data interchange mechanism for various JavaScript APIs that work with binary data. Figure 22-2 illustrates how Blobs can be read from and written to the Web, the local filesystem, local databases, and also other windows and workers. It also shows how Blob content can be accessed as text, as typed arrays, or as URLs.

Before you can work with a Blob, you must obtain one somehow. There are a number of ways to do this, some involving APIs we’ve already covered and some involving APIs that are described later in this chapter:

- Blobs are supported by the structured clone algorithm (see “Structured Clones” on page 672), which means that you can obtain one from another window or thread via the message event. See §22.3 and §22.4.
- Blobs can be retrieved from client-side databases, as described in §22.8.
- Blobs can be downloaded from the web via scripted HTTP, using cutting-edge features of the XHR2 specification. This is covered in §22.6.2.
- You can create your own blobs, using a `BlobBuilder` object to build them out of strings, `ArrayBuffer` objects (§22.5), and other Blobs. The `BlobBuilder` object is demonstrated in §22.6.3.

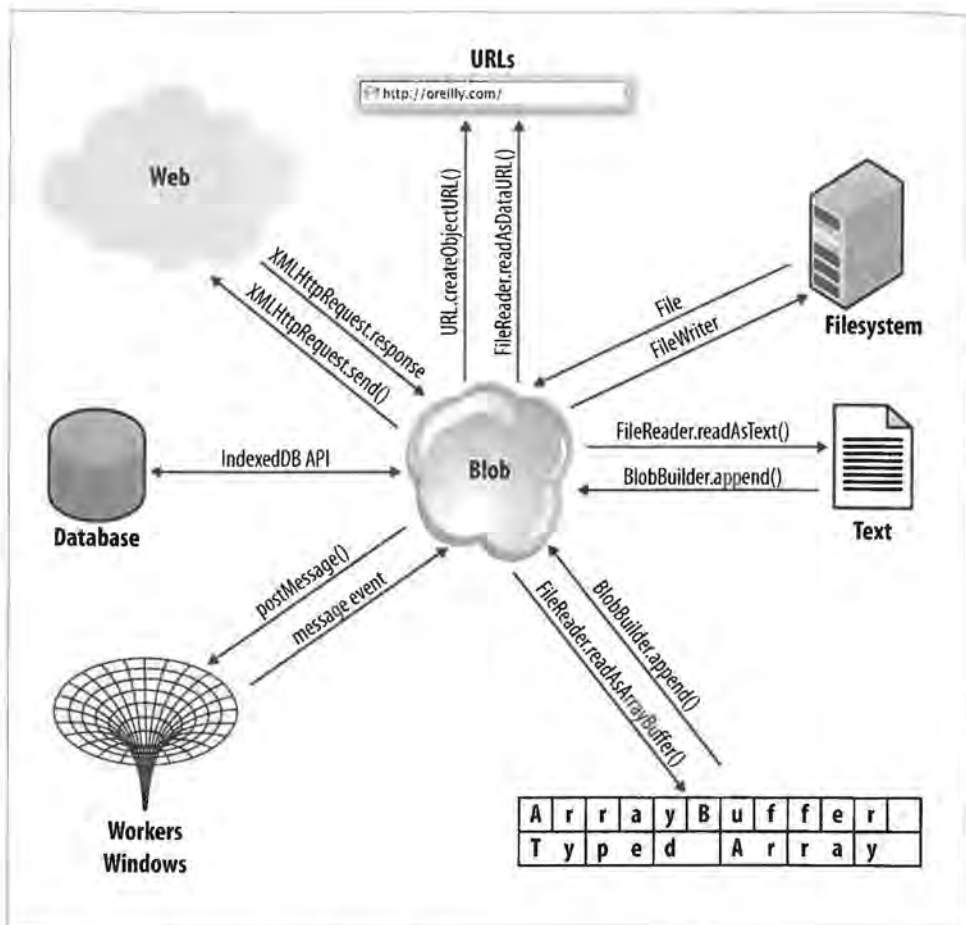


Figure 22-2. Blobs and the APIs that use them

- Finally, and most importantly, the client-side JavaScript File object is a subtype of Blob: a File is just a Blob of data with a name and a modification date. You can obtain File objects from `<input type="file">` elements and from the drag-and-drop API, as explained in §22.6.1. File objects can also be obtained using the Filesystem API, which is covered in §22.7.

Once you have a Blob, there are various things you can do with it, many of them symmetrical to the items above:

- You can send a Blob to another window or worker thread with `postMessage()`. See §22.3 and §22.4.
- You can store a Blob in a client-side database. See §22.8.
- You can upload a Blob to a server by passing it to the `send()` method of an XMLHttpRequest object. The file upload example (Example 18-9) demonstrated how to do this (remember, a File object is just a specialized kind of Blob).

- You can use the `createObjectURL()` function to obtain a special `blob://` URL that refers to the content of a Blob, and then use this URL with the DOM or with CSS. §22.6.4 demonstrates this.
- You can use a `FileReader` object to asynchronously (or synchronously, in a worker thread) extract the content of a Blob into a string or `ArrayBuffer`. §22.6.5 demonstrates the basic technique.
- You can use the `Filesystem` API and the `FileWriter` object described in §22.7 to write a Blob into a local file.

The subsections below demonstrate simple ways to obtain and use Blobs. The more complicated techniques involving the local filesystem and client-side databases are covered later, in sections of their own.

22.6.1 Files As Blobs

The `<input type="file">` element was originally intended to enable file uploads in HTML forms. Browsers have always been careful to implement this element so that it only allows the upload of files explicitly selected by the user. Scripts cannot set the `value` property of this element to a filename, so they cannot go uploading arbitrary files from the user's computer. More recently, browser vendors have extended this element to allow client-side access to user-selected files. Note that allowing a client-side script to read the contents of selected files is no more or less secure than allowing those files to be uploaded to the server.

In browsers that support local file access, the `files` property of an `<input type="file">` element will be a `FileList` object. This is an array-like object whose elements are zero or more user-selected `File` objects. A `File` object is a Blob that also has `name` and `lastModifiedDate` properties:

```
<script>
// log information about a list of selected files
function fileinfo(files) {
  for(var i = 0; i < files.length; i++) { // files is an array-like object
    var f = files[i];
    console.log(f.name,                // Name only: no path
               f.size, f.type,         // size and type are Blob properties
               f.lastModifiedDate);    // another File property
  }
}
</script>
<!-- Allow selection of multiple image files and pass them to fileinfo()-->
<input type="file" accept="image/*" multiple onchange="fileinfo(this.files)"/>
```

Being able to display the names, types, and sizes of selected files isn't terribly interesting. In §22.6.4 and §22.6.5, we'll see how you can actually make use of the content of the file.

In addition to selecting files with an `<input>` element, a user can also give a script access to local files by dropping them into the browser. When an application receives a drop

event, the `dataTransfer.files` property of the event object will be the `FileList` associated with the drop, if there was one. The drag-and-drop API was covered in §17.7 and Example 22-10 below demonstrates its use with files.

22.6.2 Downloading Blobs

Chapter 18 covered scripted HTTP with the `XMLHttpRequest` object and also documented some of the new features of the `XMLHttpRequest Level 2 (XHR2)` draft specification. At the time of this writing, XHR2 defines a way to download the contents of a URL as a Blob, but browser implementations do not yet support it. Because the code cannot yet be tested, this section is only a simple sketch of the XHR2 API for working with Blobs.

Example 22-9 shows the basic technique for downloading a Blob from the Web. Contrast this example with Example 18-2, which downloads the contents of a URL as plain text:

Example 22-9. Downloading a Blob with XMLHttpRequest

```
// GET the contents of the url as a Blob and pass it to the specified callback.
// This code is untested: no browsers supported this API when it was written.
function getBlob(url, callback) {
    var xhr = new XMLHttpRequest(); // Create new XHR object
    xhr.open("GET", url);           // Specify URL to fetch
    xhr.responseType = "blob";      // We'd like a Blob, please
    xhr.onload = function() {       // onload is easier than onreadystatechange
        callback(xhr.response);     // Pass the blob to our callback
    }                               // Note .response, not .responseText
    xhr.send(null);                 // Send the request now
}
```

If the Blob you're downloading is quite large and you want to start processing it while it is downloading, you can use an `onprogress` event handler, along with the Blob reading techniques demonstrated in §22.6.5.

22.6.3 Building Blobs

Blobs often represent chunks of data from an external source such as a local file, a URL, or a database. But sometimes a web application wants to create its own Blobs to be uploaded to the Web or stored in a file or database or passed to another thread. To create a Blob from your own data, use a `BlobBuilder`:

```
// Create a new BlobBuilder
var bb = new BlobBuilder();
// Append a string to the blob, and mark the end of the string with a NUL char
bb.append("This blob contains this text and 10 big-endian 32-bit signed ints.");
bb.append("\0"); // NUL-terminate the string to mark its end
// Store some data into an ArrayBuffer
var ab = new ArrayBuffer(4*10);
var dv = new DataView(ab);
for(var i = 0; i < 10; i++) dv.setInt32(i*4,i);
// Append the ArrayBuffer to the Blob
```

```
bb.append(ab);
// Now get the blob from the builder, specifying a made-up MIME type
var blob = bb.getBlob("x-optional/mime-type-here");
```

We saw at the beginning of this section that Blobs have a `slice()` method that breaks them into pieces. You can join Blobs together by passing Blobs to the `append()` method of a `BlobBuilder`.

22.6.4 Blob URLs

The preceding sections have shown how you can obtain or create Blobs. We'll now shift gears and start talking about what you can actually *do* with the Blobs you obtain or create. One of the simplest things you can do with a Blob is create a URL that refers to the Blob. You can then use this URL anywhere you'd use a regular URL: in the DOM, in a stylesheet, or even as the target of an XMLHttpRequest.

Create a Blob URL with the function `createObjectURL()`. At the time of this writing, the draft specification and Firefox 4 put this function in a global object named `URL`, and Chrome and Webkit prefix that new global, calling it `webkitURL`. Earlier versions of the specification (and earlier browser implementations) put the function directly on the Window object. To create Blob URLs portably across browsers, you can define a utility like this:

```
var getBlobURL = (window.URL && URL.createObjectURL.bind(URL)) ||
  (window.webkitURL && webkitURL.createObjectURL.bind(webkitURL)) ||
  window.createObjectURL;
```

Web workers are also allowed to use this API and have access to these same functions, in the same `URL` (or `webkitURL`) object.

Pass a blob to `createObjectURL()` and it returns a URL (as an ordinary string). The URL will begin with `blob://`, and that URL scheme will be followed by a short string of text that identifies the Blob with some kind of opaque unique identifier. Note that this is very different than a `data://` URL, which encodes its own contents. A Blob URL is simply a reference to a Blob that is stored by the browser in memory or on the disk. `blob://` URLs are also quite different from `file://` URLs, which refer directly to a file in the local filesystem, exposing the path of the file, allowing directory browsing, and otherwise raising security issues.

Example 22-10 demonstrates two important techniques. First, it implements a “drop target” that listens for drag-and-drop events involving files. Then, when the user drops one or more files on the drop target, it uses `createObjectURL()` to obtain a URL for each one and then creates `<img>` elements to display thumbnails of the images referenced by those URLs.

Example 22-10. Displaying dropped image files with Blob URLs

```
<!DOCTYPE html>
<html><head>
<script>
```

```

// At the time of this writing, Firefox and Webkit disagree on the
// name of the createObjectURL() function
var getBlobURL = (window.URL && URL.createObjectURL.bind(URL)) ||
  (window.webkitURL && webkitURL.createObjectURL.bind(webkitURL)) ||
  window.createObjectURL;
var revokeBlobURL = (window.URL && URL.revokeObjectURL.bind(URL)) ||
  (window.webkitURL && webkitURL.revokeObjectURL.bind(webkitURL)) ||
  window.revokeObjectURL;

// When the document is loaded, add event handlers to the droptarget element
// so that it can handle drops of files
window.onload = function() {
  // Find the element we want to add handlers to.
  var droptarget = document.getElementById("droptarget");

  // When the user starts dragging files over the droptarget, highlight it.
  droptarget.ondragenter = function(e) {
    // If the drag is something other than files, ignore it.
    // The HTML5 dropzone attribute will simplify this when implemented.
    var types = e.dataTransfer.types;
    if (!types ||
      (types.contains && types.contains("Files")) ||
      (types.indexOf && types.indexOf("Files") != -1)) {
      droptarget.classList.add("active"); // Highlight droptarget
      return false;                      // We're interested in the drag
    }
  };
  // Unhighlight the drop zone if the user moves out of it
  droptarget.ondragleave = function() {
    droptarget.classList.remove("active");
  };

  // This handler just tells the browser to keep sending notifications
  droptarget.ondragover = function(e) { return false; };

  // When the user drops files on us, get their URLs and display thumbnails.
  droptarget.ondrop = function(e) {
    var files = e.dataTransfer.files;           // The dropped files
    for(var i = 0; i < files.length; i++) {    // Loop through them all
      var type = files[i].type;
      if (type.substring(0,6) != "image/")     // Skip any nonimages
        continue;
      var img = document.createElement("img"); // Create an <img> element
      img.src = getBlobURL(files[i]);          // Use Blob URL with <img>
      img.onload = function() {                // When it loads
        this.width = 100;                      // adjust its size and
        document.body.appendChild(this);      // insert into document.
        revokeBlobURL(this.src);              // But don't leak memory!
      }
    }

    droptarget.classList.remove("active");     // Unhighlight droptarget
    return false;                             // We've handled the drop
  }
};
</script>

```

```

<style> /* Simple styles for the file drop target */
#droptarget { border: solid black 2px; width: 200px; height: 200px; }
#droptarget.active { border: solid red 4px; }
</style>
</head>
<body> <!-- The document starts off with just the file drop target -->
<div id="droptarget">Drop Image Files Here</div>
</body>
</html>

```

Blob URLs have the same origin (§13.6.2) as the script that creates them. This makes them much more versatile than `file://` URLs, which have a distinct origin and are therefore difficult to use within a web application. A Blob URL is only valid in documents of the same origin. If, for example, you passed a Blob URL via `postMessage()` to a window with a different origin, the URL would be meaningless to that window.

Blob URLs are not permanent. A Blob URL is no longer valid once the user has closed or navigated away from the document whose script created the URL. It is not possible, for example, to save a Blob URL to local storage and then reuse it when the user begins a new session with a web application.

It is also possible to manually “revoke” the validity of a Blob URL by calling `URL.revokeObjectURL()` (or `webkitURL.revokeObjectURL()`) and you may have noticed that Example 22-10 does this. This is a memory management issue. Once the thumbnail image has been displayed, the Blob is no longer needed and it should be allowed to be garbage collected. But if the web browser is maintaining a mapping from the Blob URL we’ve created to the Blob, that Blob cannot be garbage collected even if we’re not using it. The JavaScript interpreter cannot track the usage of strings, and if the URL is still valid, it has to assume that it might still be used. This means that it cannot garbage collect the Blob until the URL has been revoked. Example 22-10 uses local files that don’t require any cleanup, but you can imagine a more serious memory management issue if the Blob in question were one that had been built in memory with a `BlobBuilder` or one that had been downloaded with `XMLHttpRequest` and stored in a temporary file.

The `blob://` URL scheme is explicitly designed to work like a simplified `http://` URL, and browsers are required to act like mini HTTP servers when `blob://` URLs are requested. If a Blob URL that is no longer valid is requested, the browser must send a 404 Not Found status code. If a Blob URL from a different origin is requested, the browser must respond with 403 Not Allowed. Blob URLs only work with GET requests, and when one is successfully requested, the browser sends an HTTP 200 OK status code and also sends a `Content-Type` header that uses the `type` property of the Blob. Because Blob URLs work like simple HTTP URLs, you can “download” their content with `XMLHttpRequest`. (As we’ll see in the next section, however, you can read the content of a Blob more directly using a `FileReader` object.)

22.6.5 Reading Blobs

So far, Blobs have been opaque chunks of data that allow only indirect access, through Blob URLs, to their contents. The `FileReader` object allows us read access to the characters or bytes contained in a Blob, and you can think of it as the opposite of a `BlobBuilder`. (A better name would be `BlobReader`, since it works with any Blob, not just Files.) Since Blobs can be very large objects stored in the filesystem, the API for reading them is asynchronous, much like the `XMLHttpRequest` API. A synchronous version of the API, `FileReaderSync`, is available in worker threads, although workers can also use the asynchronous version.

To use a `FileReader`, first create an instance with the `FileReader()` constructor. Next, define event handlers. Typically you'll define handlers for load and error events and possibly also for progress events. You can do this with `onload`, `onerror`, and `onprogress` or with the standard `addEventListener()` method. `FileReader` objects also trigger load-start, loadend, and abort events, which are like the `XMLHttpRequest` events with the same names: see §18.1.4.

Once you've created a `FileReader` and registered suitable event handlers, you must pass the Blob you want to read to one of four methods: `readAsText()`, `readAsArrayBuffer()`, `readAsDataURL()`, and `readAsBinaryString()`. (You can, of course, call one of these methods first and then register event handlers—the single-threaded nature of JavaScript, described in §22.4, means that event handlers will never be called until your function has returned and the browser is back in its event loop.) The first two methods are the most important and are the ones covered here. Each of these read methods takes a Blob as its first argument. `readAsText()` takes an optional second argument that specifies the name of a text encoding. If you omit the encoding, it will automatically work with ASCII and UTF-8 text (and also UTF-16 text with a byte-order mark or BOM).

As the `FileReader` reads the Blob you've specified, it updates its `readyState` property. The value starts off at 0, indicating that nothing has been read. It changes to 1 when some data is available, and changes to 2 when the read has completed. The `result` property holds a partial or complete result as a string or `ArrayBuffer`. You do not normally poll the state and `result` properties, but instead use them from your `onprogress` or `onload` event handler.

Example 22-11 demonstrates how to use the `readAsText()` method to read local text files that the user selects.

Example 22-11. Reading text files with `FileReader`

```
<script>
// Read the specified text file and display it in the <pre> element below
function readfile(f) {
    var reader = new FileReader(); // Create a FileReader object
    reader.readAsText(f);          // Read the file
    reader.onload = function() {    // Define an event handler
        var text = reader.result;   // This is the file contents
    }
}
```

```

    var out = document.getElementById("output"); // Find output element
    out.innerHTML = ""; // Clear it
    out.appendChild(document.createTextNode(text)); // Display file contents
}
reader.onerror = function(e) { // If anything goes wrong
    console.log("Error", e); // Just log it
};
}
</script>
Select the file to display:


```

The `readAsArrayBuffer()` method is similar to `readAsText()`, except that it generally takes a little more effort to make use of an `ArrayBuffer` result than a string result. Example 22-12 is an example that uses `readAsArrayBuffer()` to read the first four bytes of a file as a big-endian integer.

Example 22-12. Reading the first four bytes of a file

```

<script>
// Examine the first 4 bytes of the specified blob. If this "magic number"
// identifies the type of the file, asynchronously set a property on the Blob.
function typefile(file) {
    var slice = file.slice(0,4); // Only read the start of the file
    var reader = new FileReader(); // Create an asynchronous FileReader
    reader.readAsArrayBuffer(slice); // Read the slice of the file
    reader.onload = function(e) {
        var buffer = reader.result; // The result ArrayBuffer
        var view = new DataView(buffer); // Get access to the result bytes
        var magic = view.getUint32(0, false); // Read 4 bytes, big-endian
        switch(magic) { // Determine file type from them
            case 0x89504E47: file.verified_type = "image/png"; break;
            case 0x47494638: file.verified_type = "image/gif"; break;
            case 0x25504446: file.verified_type = "application/pdf"; break;
            case 0x504b0304: file.verified_type = "application/zip"; break;
        }
        console.log(file.name, file.verified_type);
    };
}
</script>


```

In worker threads, you can use `FileReaderSync` instead of `FileReader`. The synchronous API defines the same `readAsText()` and `readAsArrayBuffer()` methods that take the same arguments as the asynchronous methods. The difference is that the synchronous methods block until the operation is complete and return the resulting string or `ArrayBuffer` directly, with no need for event handlers. Example 22-14 below uses `FileReaderSync`.

22.7 The Filesystem API

In §22.6.5, you saw the `FileReader` class used to read the content of user-selected files, or of any `Blob`. The `File` and `Blob` types are defined by a draft specification known as the File API. Another draft specification, even newer than the File API, gives web applications controlled access to a private local filesystem “sandbox” in which they can write files, read files, create directories, list directories, and so on. At the time of this writing, this Filesystem API is implemented only by Google’s Chrome browser, but it is a powerful and important form of local storage, so it is covered here, even though the API is even less stable than most of the other APIs described in this chapter. This section covers basic filesystem tasks but does not demonstrate all features of the API. Because the API is new and unstable, it is not documented in the reference section of this book.

Working with files in the local filesystem is a multistep process. First, you must obtain an object that represents the filesystem itself. There is a synchronous API for doing this in worker threads and an asynchronous API for use on the main thread:

```
// Obtaining a filesystem synchronously. Pass filesystem lifetime and size.
// Returns a filesystem object or raises an exception.
var fs = requestFileSystemSync(PERSISTENT, 1024*1024);

// The asynchronous version uses callback functions for success and error
requestFileSystem(TEMPORARY,           // lifetime
                  50*1024*1024,        // size: 50Mb
                  function(fs) {       // called with the filesystem object
                    // Use fs here
                  },
                  function(e) {        // called with an error object onerror
                    console.log(e);     // Or handle it some other way
                  });
```

In both the synchronous and asynchronous versions of the API, you specify the lifetime and the size of the filesystem you want. A `PERSISTENT` filesystem is suitable for web apps that want to store user data permanently. The browser won’t delete it except at the user’s explicit request. A `TEMPORARY` filesystem is appropriate for web apps that want to cache data but can still operate if the web browser deletes the filesystem. The size of the filesystem is specified in bytes and should be a reasonable upper bound on the amount of data you need to store. A browser may enforce this as a quota.

The filesystem obtained with these functions depends on the origin of the containing document. All documents or web apps from the same origin (host, port, and protocol) share a filesystem. Two documents or applications from different origins have completely distinct and disjoint filesystems. The filesystem is also walled off from the rest of the files on the user’s hard drive: there is no way for a web application to “break out” beyond the local root directory or otherwise access arbitrary files.

Note that these functions have “request” in their names. The first time you call one, the browser may ask the user for permission before creating a filesystem and granting

access.² Once permission has been granted, subsequent calls to the request method should simply return an object that represents the already existing local filesystem.

The filesystem object you obtain with one of the methods above has a `root` property that refers to the root directory of the filesystem. This is a `DirectoryEntry` object, and it may have nested directories that are themselves represented by `DirectoryEntry` objects. Each directory in the file system may contain files, represented by `FileEntry` objects. The `DirectoryEntry` object defines methods for obtaining `DirectoryEntry` and `FileEntry` objects by pathname (they will optionally create new directories or files if you specify a name that doesn't exist). `DirectoryEntry` also defines a `createReader()` factory method that returns a `DirectoryReader` for listing the contents of a directory.

The `FileEntry` class defines a method for obtaining the `File` object (a `Blob`) that represents the contents of a file. You can then use a `FileReader` object (as shown in §22.6.5) to read the file. `FileEntry` defines another method to return a `FileWriter` object that you can use to write content into a file.

Reading or writing a file with this API is a multistep process. First you obtain the filesystem object. Then you use the root directory of that object to look up (and optionally create) the `FileEntry` object for the file you're interested in. Then you use the `FileEntry` object to obtain the `File` or `FileWriter` object for reading or writing. This multistep process is particularly complex when using the asynchronous API:

```
// Read the text file "hello.txt" and log its contents.
// The asynchronous API nests functions four deep.
// This example doesn't include any error callbacks.
requestFileSystem(PERSISTENT, 10*1024*1024, function(fs) { // Get filesystem
  fs.root.getFile("hello.txt", {}, function(entry) {      // Get FileEntry
    entry.file(function(file) {                            // Get File
      var reader = new FileReader();
      reader.readAsText(file);
      reader.onload = function() {                          // Get file content
        console.log(reader.result);
      };
    });
  });
});
```

Example 22-13 is a more fully fleshed-out example. It demonstrates how to use the asynchronous API to read files, write files, delete files, create directories, and list directories.

Example 22-13. Using the asynchronous filesystem API

```
/*
 * These functions have been tested in Google Chrome 10.0 dev.
 * You may need to launch Chrome with these options:
 * --unlimited-quota-for-files      : enables filesystem access
 * --allow-file-access-from-files : allows testing from file:// URLs
 */
```

2. At the time of this writing, Chrome doesn't ask for permission, but it requires you to launch it with the `--unlimited-quota-for-files` command-line flag.

```

*/

// Lots of the asynchronous functions we use accept an optional error callback.
// This one just logs the error.
function logerr(e) { console.log(e); }

// requestFileSystem() gets us a sandboxed local filesystem accessible only
// to apps from this origin. We can read and write files at will, but
// can't get out of the sandbox to access the rest of the system.
var filesystem; // Assume this is initialized before the funcs below are called.
requestFileSystem(PERSISTENT, // Or TEMPORARY for cache files
    10*1024*1024, // We'd like 10 megabytes, please
    function(fs) { // When done, call this function
        filesystem = fs; // Just save the filesystem into
    }, // a global variable.
    logerr); // Call this if an error occurs

// Read the contents of the specified file as text and pass them to callback.
function readTextFile(path, callback) {
    // Call getFile() to find the FileEntry for the specified filename
    filesystem.root.getFile(path, {}, function(entry) {
        // This function is called with the FileEntry for the file
        // Now we call the FileEntry.file() method to get the File object
        entry.file(function(file) { // Call this with the File
            var reader = new FileReader(); // Create a FileReader
            reader.readAsText(file); // And read the file
            reader.onload = function() { // When read successful
                callback(reader.result); // Pass it to the callback
            }
            reader.onerror = logerr; // Log readAsText() errors
        }, logerr); // Log file() errors
    }, logerr); // Log getFile() errors
}

// Append the specified contents to the file at the specified path, creating
// a new file if no file by that name already exists. Call callback when done.
function appendToFile(path, contents, callback) {
    // filesystem.root is the root directory.
    filesystem.root.getFile( // Get a FileEntry object
        path, // The name and path of the file we want
        {create:true}, // Create it if it doesn't already exist
        function(entry) { // Call this when it has been found
            entry.createWriter( // Create a FileWriter object for the file
                function(writer) { // Call this function when created
                    // By default a writer starts at the beginning of the file.
                    // We want to start writing at the end of the file
                    writer.seek(writer.length); // Move to end of file

                    // Convert file contents to a Blob. The contents argument
                    // can be a string or a Blob or an ArrayBuffer.
                    var bb = new BlobBuilder()
                    bb.append(contents);
                    var blob = bb.getBlob();
                }
            )
        }
    )
}

```

```

        // Now write the blob to the file
        writer.write(blob);
        writer.onerror = logerr; // Log errors from write()
        if (callback)           // If there is a callback
            writer.onwrite = callback; // call it on success
    },
    logerr); // Log errors from createWriter()
},
logerr); // Log errors from getFile()
}

// Delete the named file, calling the optional callback when done
function deleteFile(name, callback) {
    filesystem.root.getFile(name, {}, // Get FileEntry for named file
        function(entry) { // Pass the FileEntry here
            entry.remove(callback, // Delete the FileEntry
                logerr); // Or log remove() error
        },
        logerr); // Log a getFile() error
}

// Create a new directory with the specified name
function makeDirectory(name, callback) {
    filesystem.root.getDirectory(name, // Name of directory to create
        { // Options
            create: true, // Create, if doesn't exist
            exclusive: true // Error if it does exist
        },
        callback, // Call this when done
        logerr); // Log any errors
}

// Read the contents of the specified directory, and pass them, as an array
// of strings, to the specified callback function
function listFiles(path, callback) {
    // If no directory specified, list the root directory. Otherwise, look up
    // the named directory and list it (or log an error looking it up).
    if (!path) getFiles(filesystem.root);
    else filesystem.root.getDirectory(path, {}, getFiles, logerr);

    function getFiles(dir) { // This function is used above
        var reader = dir.createReader(); // A DirectoryReader object
        var list = []; // Where we store filenames
        reader.readEntries(handleEntries, // Pass entries to function below
            logerr); // or log an error.

        // Reading directories can be a multistep process. We have to keep
        // calling readEntries() until we get an empty array. Then we're done
        // and we can pass the full list to the user's callback function.
        function handleEntries(entries) {
            if (entries.length == 0) callback(list); // We're done
            else {
                // Otherwise, add these entries to the list and ask for more
                // The array-like object contains FileEntry objects and
                // we need to get the name of each one.
                for(var i = 0; i < entries.length; i++) {

```

```

        var name = entries[i].name;           // Get entry name
        if (entries[i].isDirectory) name += "/"; // Mark directories
        list.push(name);                     // Add to list
    }
    // Now get the next batch of entries
    reader.readEntries(handleEntries, logerr);
}
}
}
}
}

```

Working with files and the filesystem is quite a bit easier in worker threads, where it is okay to make blocking calls and we can use the synchronous API. Example 22-14 defines the same filesystem utility functions as Example 22-13 does, but it uses the synchronous API and is quite a bit shorter.

Example 22-14. The synchronous filesystem API

```

// Filesystem utilities using the synchronous API in a worker thread
var filesystem = requestFileSystemSync(PERSISTENT, 10*1024*1024);

function readTextFile(name) {
    // Get a File from a FileEntry from the root DirectoryEntry
    var file = filesystem.root.getFile(name).file();
    // Use the synchronous FileReader API to read it
    return new FileReaderSync().readAsText(file);
}

function appendToFile(name, contents) {
    // Get a FileWriter from a FileEntry from the root DirectoryEntry
    var writer = filesystem.root.getFile(name, {create:true}).createWriter();
    writer.seek(writer.length); // Start at the end of the file
    var bb = new BlobBuilder() // Build the file contents into a Blob
    bb.append(contents);
    writer.write(bb.getBlob()); // Now write the blob to the file
}

function deleteFile(name) {
    filesystem.root.getFile(name).remove();
}

function makeDirectory(name) {
    filesystem.root.getDirectory(name, { create: true, exclusive:true });
}

function listFiles(path) {
    var dir = filesystem.root;
    if (path) dir = dir.getDirectory(path);

    var lister = dir.createReader();
    var list = [];
    do {
        var entries = lister.readEntries();
        for(var i = 0; i < entries.length; i++) {
            var name = entries[i].name;

```

```

        if (entries[i].isDirectory) name += "/";
        list.push(name);
    }
    } while(entries.length > 0);

    return list;
}

// Allow the main thread to use these utilities by sending a message
onmessage = function(e) {
    // We expect the message to be an object like this:
    // { function: "appendToFile", args: ["test", "testing, testing"]}
    // We invoke the specified function with the specified args and
    // post the message back
    var f = self[e.data.function];
    var result = f.apply(null, e.data.args);
    postMessage(result);
};

```

22.8 Client-Side Databases

Web application architecture has traditionally featured HTML, CSS, and JavaScript on the client and a database on the server. Among the most surprising HTML5 APIs, therefore, are client-side databases. These are not just client-side APIs for accessing database servers across the network, but actual client-side databases stored on the user's computer and directly accessed by JavaScript code in the browser.

The Web Storage API described in §20.1 can be thought of as a particularly simple kind of database that persists simple key/value pairs. But in addition, there are two client-side database APIs that are “real” databases. One, known as Web SQL Database, is a simple relational database that supports basic SQL queries. Chrome, Safari, and Opera implemented this API, but Firefox and IE have not, and likely never will. Work on the official specification of this API has stopped and this full-featured SQL database will probably never become an official standard nor an unofficial but interoperable feature of the web platform.

Standardization efforts are now focused on another database API, known as IndexedDB. It is too early to document this API in any detail (it is not covered in Part IV), but Firefox 4 and Chrome 11 include implementations, and this section includes a working example that demonstrates some of the most important features of the IndexedDB API.

IndexedDB is an object database, not a relational database, and it is much simpler than databases that support SQL queries. It is more powerful, efficient, and robust than the key/value storage provided by the Web Storage API, however. Like the Web Storage and the Filesystem API, IndexedDB databases are scoped to the origin of the containing document: two web pages with the same origin can access each other's data, but web pages from different origins cannot.

Each origin can have any number of IndexedDB databases. Each one has a name that must be unique within the origin. In the IndexedDB API, a database is simply a collection of named *object stores*. As the name implies, an object store stores objects (or any value that can be cloned—see “Structured Clones” on page 672). Each object must have a *key* by which it can be sorted and retrieved from the store. Keys must be unique—two objects in the same store may not have the same key—and they must have a natural ordering so that they can be sorted. JavaScript strings, numbers, and Date objects are valid keys. An IndexedDB database can automatically generate a unique key for each object you insert into the database. Often, though, the objects you insert into an object store will already have a property that is suitable for use as a key. In this case, you specify a “key path” for that property when you create the object store. Conceptually, a key path is a value that tells the database how to extract an object’s key from the object.

In addition to retrieving objects from an object store by their primary key value, you may want to be able to search based on the value of other properties in the object. In order to be able to do this, you can define any number of *indexes* on the object store. (The ability to index an object store explains the name “IndexedDB.”) Each index defines a secondary key for the stored objects. These indexes are not generally unique and multiple objects may match a single key value. So when querying an object store via an index, you generally use a *cursor*, which defines an API for retrieving streaming query results one at a time. Cursors can also be used when querying an object store (or index) for a range of keys, and the IndexedDB API includes an object for describing ranges (upper-bounded and/or lower-bounded, inclusive bounds or exclusive bounds) of keys.

IndexedDB provides atomicity guarantees: queries and updates to the database are grouped within a *transaction* so that they all succeed together or all fail together and never leave the database in an undefined partially updated state. Transactions in IndexedDB are simpler than in many database APIs; we’ll mention them again below.

Conceptually, the IndexedDB API is quite simple. To query or update a database, you first open the database you want (specifying it by name). Next, you create a transaction object and use that object to look up the desired object store within the database, also by name. Finally, you look up an object by calling the `get()` method of the object store or store a new object by calling `put()`. (Or by calling `add()`, if you want to avoid overwriting existing objects.) If you want to look up the objects for a range of keys, you create an `IDBRange` object and pass it to the `openCursor()` method of the object store. Or, if you want to make a query using a secondary key, you look up the named index of the object store, and then call the `get()` or `openCursor()` method of the index object.

This conceptual simplicity is complicated, however, by the fact that the API must be asynchronous so that web apps can use it without blocking the browser’s main UI thread. (The IndexedDB specification defines a synchronous version of the API for use by worker threads, but at the time of this writing, no browser has yet implemented that version of the API, so it is not covered here.) Creating transactions and looking up object stores and indexes are easy synchronous operations. But opening a database,

updating an object store with `put()`, and querying a store or index with `get()` or `openCursor()` are all asynchronous operations. These asynchronous methods all immediately return a request object. The browser triggers a success or error event on the request object when the request succeeds or fails, and you can define handlers with the `onsuccess` and `onerror` properties. Inside an `onsuccess` handler, the result of the operation is available as the `result` property of the request object.

One convenient feature of this asynchronous API is that it simplifies transaction management. In a typical use of the IndexedDB API, you first open the database. This is an asynchronous operation, so it triggers an `onsuccess` handler. In that event handler, you create a transaction object, and then use that transaction object to look up the object store or stores you'll be using. Then you make any number of `get()` and `put()` calls on the object stores. They are asynchronous, so nothing happens right away, but the requests generated by those `get()` and `put()` calls are automatically associated with the transaction object. If you need to, you can cancel all the pending operations (and undo any that have already completed) in the transaction by calling the `abort()` method of the transaction object. In many other database APIs, you'd expect the transaction object to have a `commit()` method to finalize the transaction. In IndexedDB, however, the transaction is committed after the original `onsuccess` event handler that created the transaction exits and the browser returns to its event loop and after all the operations pending on that transaction complete (without starting any new operations in their callback functions). This sounds complicated, but in practice, it is straightforward. The IndexedDB API forces you to create transaction objects in order to look up object stores, but in common use cases, you really don't have to think about the transactions very much.

Finally, there is one special kind of transaction that enables a very important part of the IndexedDB API. Creating a new database in the IndexedDB API is easy: you just pick a name and request that that database be opened. But a new database is completely empty, and it is useless unless you add one or more object stores (and possibly some indexes as well) to it. The only way to create object stores and indexes is within the `onsuccess` event handler of the request object returned by a call to the `setVersion()` method of the database object. Calling `setVersion()` allows you to specify a version number for the database—in typical usage, you update the version number each time you alter the structure of the database. More importantly, however, `setVersion()` implicitly begins a special kind of transaction that enables you to call the `createObjectStore()` method of the database object and the `createIndex()` method of an object store.

With this high-level overview of IndexedDB in mind, you should now be able to understand Example 22-15. That example uses IndexedDB to create and query a database that maps US postal codes (zipcodes) to US cities. It demonstrates many, but not all, of the basic features of IndexedDB. At the time of this writing, the example works in Firefox 4 and Chrome 11, but because the specification is still in flux and implementations are still quite preliminary, there is a chance that it won't work exactly as written when you read this. Nevertheless, the overall structure of the example should still be

useful to you. Example 22-15 is long, but it has lots of comments that make it easy to understand.

Example 22-15. A IndexedDB database of US postal codes

```
<!DOCTYPE html>
<html>
<head>
<title>Zipcode Database</title>
<script>
// IndexedDB implementations still use API prefixes
var indexedDB = window.indexedDB || // Use the standard DB API
    window.mozIndexedDB || // Or Firefox's early version of it
    window.webkitIndexedDB; // Or Chrome's early version
// Firefox does not prefix these two:
var IDBTransaction = window.IDBTransaction || window.webkitIDBTransaction;
var IDBKeyRange = window.IDBKeyRange || window.webkitIDBKeyRange;

// We'll use this function to log any database errors that occur
function logerr(e) {
    console.log("IndexedDB error" + e.code + ": " + e.message);
}

// This function asynchronously obtains the database object (creating and
// initializing the database if necessary) and passes it to the function f().
function withDB(f) {
    var request = indexedDB.open("zipcodes"); // Request the zipcode database
    request.onerror = logerr; // Log any errors
    request.onsuccess = function() { // Or call this when done
        var db = request.result; // The result of the request is the database

        // You can always open a database, even if it doesn't exist.
        // We check the version to find out whether the DB has been created and
        // initialized yet. If not, we have to go do that. But if the db is
        // already set up, we just pass it to the callback function f().
        if (db.version === "1") f(db); // If db is init'd, pass it to f()
        else initdb(db, f); // Otherwise initialize it first
    }
}

// Given a zip code, find out what city it belongs to and asynchronously
// pass the name of that city to the specified callback function.
function lookupCity(zip, callback) {
    withDB(function(db) {
        // Create a transaction object for this query
        var transaction = db.transaction(["zipcodes"], // Object stores we need
            IDBTransaction.READ_ONLY, // No updates
            0); // No timeout

        // Get the object store from the transaction
        var objects = transaction.objectStore("zipcodes");

        // Now request the object that matches the specified zipcode key.
        // The lines above were synchronous, but this one is async
        var request = objects.get(zip);
    });
}
```

```

request.onerror = logerr;           // Log any errors that occur
request.onsuccess = function() {    // Pass the result to this function
    // The result object is now in the request.result
    var object = request.result;
    if (object) // If we found a match, pass city and state to callback
        callback(object.city + ", " + object.state);
    else       // Otherwise, tell the callback that we failed
        callback("Unknown zip code");
}
});
}

// Given the name of a city find all zipcodes for all cities (in any state)
// with that name (case-sensitive). Asynchronously pass the results, one at
// a time, to the specified callback function
function lookupZipcodes(city, callback) {
    withDB(function(db) {
        // As above, we create a transaction and get the object store
        var transaction = db.transaction(["zipcodes"],
                                         IDBTransaction.READ_ONLY, 0);
        var store = transaction.objectStore("zipcodes");
        // This time we get the city index of the object store
        var index = store.index("cities");

        // This query is likely to have many results, so we have to use a
        // cursor object to retrieve them all. To create a cursor, we need
        // a range object that represents the range of keys
        var range = new IDBKeyRange.only(city); // A range with only() one key

        // Everything above has been synchronous.
        // Now we request a cursor, which will be returned asynchronously.
        var request = index.openCursor(range); // Request the cursor
        request.onerror = logerr;             // Log errors
        request.onsuccess = function() {      // Pass cursor to this function
            // This event handler will be invoked multiple times, once
            // for each record that matches the query, and then once more
            // with a null cursor to indicate that we're done.
            var cursor = request.result       // The cursor is in request.result
            if (!cursor) return;             // No cursor means no more results
            var object = cursor.value         // Get the matching record
            callback(object);                // Pass it to the callback
            cursor.continue();               // Ask for the next matching record
        }
    });
}

// This function is used by an onchange callback in the document below
// It makes a DB request and displays the result
function displayCity(zip) {
    lookupCity(zip, function(s) { document.getElementById('city').value = s; });
}

// This is another onchange callback used in the document below.
// It makes a DB request and displays the results
function displayZipcodes(city) {
    var output = document.getElementById("zipcodes");

```

```

    output.innerHTML = "Matching zipcodes:";
    lookupZipcodes(city, function(o) {
        var div = document.createElement("div");
        var text = o.zipcode + ": " + o.city + ", " + o.state;
        div.appendChild(document.createTextNode(text));
        output.appendChild(div);
    });
}

// Set up the structure of the database and populate it with data, then pass
// the database to the function f(). withDB() calls this function if the
// database has not been initialized yet. This is the trickiest part of the
// program, so we've saved it for last.
function initdb(db, f) {
    // Downloading zipcode data and storing it in the database can take
    // a while the first time a user runs this application. So we have to
    // provide notification while that is going on.
    var statusline = document.createElement("div");
    statusline.style.cssText =
        "position:fixed; left:0px; top:0px; width:100%;" +
        "color:white; background-color: black; font: bold 18pt sans-serif;" +
        "padding: 10px; ";
    document.body.appendChild(statusline);
    function status(msg) { statusline.innerHTML = msg.toString(); }

    status("Initializing zipcode database");

    // The only time you can define or alter the structure of an IndexedDB
    // database is in the onsuccess handler of a setVersion request.
    var request = db.setVersion("1"); // Try to update the DB version
    request.onerror = status; // Display status on fail
    request.onsuccess = function() { // Otherwise call this function
        // Our zipcode database includes only one object store.
        // It will hold objects that look like this: {
        //   zipcode: "02134", // send it to Zoom! :-}
        //   city: "Allston",
        //   state: "MA",
        //   latitude: "42.355147",
        //   longitude: "-71.13164"
        // }
        //
        // We'll use the "zipcode" property as the database key
        // And we'll also create an index using the city name

        // Create the object store, specifying a name for the store and
        // an options object that includes the "key path" specifying the
        // property name of the key field for this store. (If we omit the
        // key path, IndexedDB will define its own unique integer key.)
        var store = db.createObjectStore("zipcodes", // store name
            { keyPath: "zipcode" });

        // Now index the object store by city name as well as by zipcode.
        // With this method the key path string is passed directly as a
        // required argument rather than as part of an options object.
        store.createIndex("cities", "city");
    }
}

```

```

// Now we need to download our zipcode data, parse it into objects
// and store those objects in object store we created above.
//
// Our file of raw data contains lines formatted like this:
//
// 02130,Jamaica Plain,MA,42.309998,-71.11171
// 02131,Roslindale,MA,42.284678,-71.13052
// 02132,West Roxbury,MA,42.279432,-71.1598
// 02133,Boston,MA,42.338947,-70.919635
// 02134,Allston,MA,42.355147,-71.13164
//
// Surprisingly, the US Postal Service does not make this data freely
// available, so we use out-of-date census-based zipcode data from:
// http://mappinghacks.com/2008/04/28/civicspace-zip-code-database/

// We use XMLHttpRequest to download the data. But use the new XHR2
// onload and onprogress events to process it as it arrives
var xhr = new XMLHttpRequest(); // An XHR to download the data
xhr.open("GET", "zipcodes.csv"); // HTTP GET for this URL
xhr.send(); // Start right away
xhr.onerror = status; // Display any error codes
var lastChar = 0, numlines = 0; // How much have we already processed?

// Handle the database file in chunks as it arrives
xhr.onprogress = xhr.onload = function(e) { // Two handlers in one!
    // We'll process the chunk between lastChar and the last newline
    // that we've received. (We need to look for newlines so we don't
    // process partial records)
    var lastNewline = xhr.responseText.lastIndexOf("\n");
    if (lastNewline > lastChar) {
        var chunk = xhr.responseText.substring(lastChar, lastNewline);
        lastChar = lastNewline + 1; // Where to start next time

        // Now break the new chunk of data into individual lines
        var lines = chunk.split("\n");
        numlines += lines.length;

        // In order to insert zipcode data into the database we need a
        // transaction object. All the database insertions we make
        // using this object will be committed to the database when this
        // function returns and the browser goes back to the event
        // loop. To create our transaction object, we need to specify
        // which object stores we'll be using (we only have one) and we
        // need to tell it that we'll be doing writes to the database,
        // not just reads:
        var transaction = db.transaction(["zipcodes"], // object stores
                                         IDBTransaction.READ_WRITE);

        // Get our object store from the transaction
        var store = transaction.objectStore("zipcodes");

        // Now loop through the lines of the zipcode file, create
        // objects for them, and add them to the object store.
        for(var i = 0; i < lines.length; i++) {
            var fields = lines[i].split(","); // Comma-separated values
            var record = { // This is the object we'll store

```

```

        zipcode: fields[0], // All properties are string
        city: fields[1],
        state: fields[2],
        latitude: fields[3],
        longitude: fields[4]
    };

    // The best part about the IndexedDB API is that object
    // stores are really simple. Here's how we add a record:
    store.put(record); // Or use add() to avoid overwriting
}

status("Initializing zipcode database: loaded "
      + numlines + " records.");
}

if (e.type == "load") {
    // If this was the final load event, then we've sent all our
    // zipcode data to the database. But since we've just blasted
    // it with some 40,000 records, it may still be processing.
    // So we'll make a simple query. When it succeeds, we know
    // that the database is ready to go, and we can then remove
    // the status line and finally call the function f() that was
    // passed to withDB() so long ago
    lookupCity("02134", function(s) { // Allston, MA
        document.body.removeChild(statusline);
        withDB(f);
    });
}
}
}
}
</script>
</head>
<body>
<p>Enter a zip code to find its city:</p>
Zipcode: <input onchange="displayCity(this.value)"></input>
City: <output id="city"></output>
</div>
<div>
<p>Enter a city name (case sensitive, without state) to find cities and their zipcodes:</p>
City: <input onchange="displayZipcodes(this.value)"></input>
<div id="zipcodes"></div>
</div>
<p><i>This example is only known to work in Firefox 4 and Chrome 11.</i></p>
<p><i>Your first query may take a very long time to complete.</i></p>
<p><i>You may need to start Chrome with --unlimited-quota-for-indexeddb</i></p>
</body>
</html>

```

22.9 Web Sockets

Chapter 18 showed how client-side JavaScript code can communicate over the network. The examples in that chapter all used HTTP, which means that they were all

constrained by the fundamental nature of the HTTP: it is a stateless protocol that consists of client requests and server responses. HTTP is actually a fairly specialized network protocol. More general network connections over the Internet (or over local intranets) often involve longer-lived connections and bidirectional message exchange over TCP sockets. It is not safe to give untrusted client-side JavaScript code access to low-level TCP sockets, but the WebSocket API defines a secure alternative: it allows client-side code to create bidirectional socket-type connections to servers that support the WebSocket protocol. This makes it much easier to perform certain kinds of networking tasks.

The WebSocket Protocol

In order to use WebSockets from JavaScript, you only need to understand the client-side WebSocket API described here. There is no equivalent server-side API for writing WebSocket servers, but this section does include a simple server example that uses Node (§12.2) along with a third-party WebSocket server library. The client and server communicate over a long-lived TCP socket following rules defined by the WebSocket protocol. The details of the protocol are not relevant here, but it is worth noting that the WebSocket protocol is carefully designed so that web servers can easily handle both HTTP and WebSocket connections over the same port.

WebSockets enjoy wide support among web browser vendors. An early draft version of the WebSocket protocol was found to have an important security hole, however, and at the time of this writing, some browsers have disabled their WebSocket support until a secure version of the protocol has been standardized. In Firefox 4, for example, you may need to explicitly enable WebSockets by visiting the `about:config` page and setting the configuration variable `network.websocket.override-security-block` to `true`.

The WebSocket API is surprisingly easy to use. First, create a socket with the `WebSocket()` constructor:

```
var socket = new WebSocket("ws://ws.example.com:1234/resource");
```

The argument to the `WebSocket()` constructor is a URL that uses the `ws://` protocol (or `wss://` for a secure connection like that used by `https://`). The URL specifies the host to connect to, and may also specify a port (WebSockets use the same default ports as HTTP and HTTPS) and a path or resource.

Once you have created a socket, you generally register event handlers on it:

```
socket.onopen = function(e) { /* The socket is now connected. */ };
socket.onclose = function(e) { /* The socket closed. */ };
socket.onerror = function(e) { /* Something went wrong! */ };
socket.onmessage = function(e) {
    var message = e.data;      /* The server sent us a message. */
};
```

In order to send data to the server over the socket, you call the `send()` method of the socket:

```
socket.send("Hello, server!");
```

The current version of the WebSocket API supports only textual messages, and sends them as UTF-8 encoded strings. The current WebSocket protocol includes support for binary messages, however, and a future version of the API may allow binary data to be exchanged with a WebSocket server.

When your code is done communicating with the server, you can close a WebSocket by calling its `close()` method.

WebSocket is completely bidirectional, and once a WebSocket connection has been established, the client and server can send messages to each other at any time, and that communication does not have to take the form of requests and responses. Each WebSocket-based service will define its own “subprotocol” for transferring data between client and server. Over time, these “subprotocols” may evolve, and you may end up with clients and servers that need to support more than one version of a subprotocol. Fortunately, the WebSocket protocol includes a negotiation mechanism for choosing a subprotocol that both client and server can understand. You can pass an array of strings to the `WebSocket()` constructor. The server will take these as a list of the subprotocols that the client understands. It will pick one to use and send it back to the client. Once the connection is established, the client can determine which subprotocol is in use by checking the `protocol` property of the socket.

§18.3 explained the EventSource API and demonstrated it with an online chat client and server. WebSockets make it even easier to write this kind of application. Example 22-16 is a very simple chat client: it is a lot like Example 18-15, but it uses a WebSocket for bidirectional communication instead of using an EventSource to receive messages and an XMLHttpRequest to send them.

Example 22-16. A WebSocket-based chat client

```
<script>
window.onload = function() {
    // Take care of some UI details
    var nick = prompt("Enter your nickname"); // Get user's nickname
    var input = document.getElementById("input"); // Find the input field
    input.focus(); // Set keyboard focus

    // Open a WebSocket to send and receive chat messages on.
    // Assume that the HTTP server we were downloaded from also functions as
    // a websocket server, and use the same host name and port, but change
    // from the http:// protocol to ws://
    var socket = new WebSocket("ws://" + location.host + "/");

    // This is how we receive messages from the server through the web socket
    socket.onmessage = function(event) { // When a new message arrives
        var msg = event.data; // Get text from event object
        var node = document.createTextNode(msg); // Make it into a text node
        var div = document.createElement("div"); // Create a <div>
        div.appendChild(node); // Add text node to div
        document.body.insertBefore(div, input); // And add div before input
        input.scrollIntoView(); // Ensure input elt is visible
    }
}
```

```

    }

    // This is how we send messages to the server through the web socket
    input.onChange = function() {
        var msg = nick + ": " + input.value;
        socket.send(msg);
        input.value = "";
    }
    // When user strikes return
    // Username plus user's input
    // Send it through the socket
    // Get ready for more input
};
</script>
<!-- The chat UI is just a single, wide text input field -->
<!-- New chat messages will be inserted before this element -->


```

Example 22-17 is a WebSocket-based chat server intended to be run in Node (§12.2). You can compare this example to Example 18-17 to see that WebSockets simplify the server-side of the chat application as well as the client-side.

Example 22-17. A chat server using WebSockets and Node

```

/*
 * This is server-side JavaScript, intended to be run with NodeJS.
 * It runs a WebSocket server on top of an HTTP server, using an external
 * websocket library from https://github.com/miksago/node-websocket-server/
 * If it gets an HTTP request for "/" it returns the chat client HTML file.
 * Any other HTTP requests return 404. Messages received via the
 * WebSocket protocol are simply broadcast to all active connections.
 */
var http = require('http'); // Use Node's HTTP server API
var ws = require('websocket-server'); // Use an external WebSocket library

// Read the source of the chat client at startup. Used below.
var clientui = require('fs').readFileSync("wschatclient.html");

// Create an HTTP server
var httpserver = new http.Server();

// When the HTTP server gets a new request, run this function
httpserver.on("request", function (request, response) {
    // If the request was for "/", send the client-side chat UI.
    if (request.url === "/") { // A request for the chat UI
        response.writeHead(200, {"Content-Type": "text/html"});
        response.write(clientui);
        response.end();
    }
    else { // Send a 404 "Not Found" code for any other request
        response.writeHead(404);
        response.end();
    }
});

// Now wrap a WebSocket server around the HTTP server
var wsserver = ws.createServer({server: httpserver});

// Call this function when we receive a new connection request

```

```
wsserver.on("connection", function(socket) {  
    socket.send("Welcome to the chat room."); // Greet the new client  
    socket.on("message", function(msg) {      // Listen for msgs from the client  
        wsserver.broadcast(msg);              // And broadcast them to everyone  
    });  
});
```

```
// Run the server on port 8000. Starting the WebSocket server starts the  
// HTTP server as well. Connect to http://localhost:8000/ to use it.  
wsserver.listen(8000);
```

Core JavaScript Reference

This part of the book is a reference that documents the classes, methods, and properties defined by the core JavaScript language. This reference is arranged alphabetically by class or object name:

Arguments	EvalError	Number	String
Array	Function	Object	SyntaxError
Boolean	Global	RangeError	TypeError
Date	JSON	ReferenceError	URIError
Error	Math	RegExp	

The reference pages for the methods and properties of classes are alphabetized by their full names, which include the names of the classes that define them. For example, if you want to read about the `replace()` method of the `String` class, you would look under `String.replace()`, not just `replace`.

Core JavaScript defines some global functions and properties, such as `eval()` and `NaN`. Technically, these are properties of the global object. Since the global object has no name, however, they are listed in this reference section under their own unqualified names. For convenience, the full set of global functions and properties in core JavaScript is summarized in a special reference page named “Global” (even though there is no object or class by that name).

Core JavaScript Reference

arguments[]

an array of function arguments

Synopsis

`arguments`

Description

The `arguments[ ]` array is defined only within a function body. Within the body of a function, `arguments` refers to the Arguments object for the function. This object has numbered properties and serves as an array containing all arguments passed to the function. The `arguments` identifier is essentially a local variable automatically declared and initialized within every function. It refers to an Arguments object only within the body of a function and is undefined in global code.

See Also

Arguments; Chapter 8

Arguments

arguments and other properties of a function

Object → Arguments

Synopsis

`arguments`
`arguments[n]`

Elements

The Arguments object is defined only within a function body. Although it is not technically an array, the Arguments object has numbered properties that function as array elements and a `length` property that specifies the number of array elements. Its elements are the values that are passed as arguments to the function. Element 0 is the first argument, element 1 is the second argument, and so on. All values passed as arguments become array elements of the Arguments object, whether or not those arguments are given names in the function declaration.

Properties

callee

A reference to the function that is currently executing.

length

The number of arguments passed to the function and the number of array elements in the Arguments object.

Description

When a function is invoked, an Arguments object is created for it, and the local variable `arguments` is automatically initialized to refer to that Arguments object. The main purpose of the Arguments object is to provide a way to determine how many arguments are passed to the function and to refer to unnamed arguments. In addition to the array elements and `length` property, however, the `callee` property allows an unnamed function to refer to itself.

For most purposes, the Arguments object can be thought of as an array with the addition of the `callee` property. However, it is not an instance of Array, and the `Arguments.length` property does not have any of the special behaviors of the `Array.length` property and cannot be used to change the size of the array.

In non-strict mode, the Arguments object has one *very* unusual feature. When a function has named arguments, the array elements of the Arguments object are synonyms for the local variables that hold the function arguments. The Arguments object and the argument names provide two different ways of referring to the same variable. Changing the value of an argument with an argument name changes the value that is retrieved through the Arguments object, and changing the value of an argument through the Arguments object changes the value that is retrieved by the argument name.

See Also

Function; Chapter 8

Arguments.callee

not defined in strict mode

the function that is currently running

Synopsis

```
arguments.callee
```

Description

`arguments.callee` refers to the function that is currently running. It provides a way for an unnamed function to refer to itself. This property is defined only within a function body.

Example

```
// An unnamed function literal uses the callee property to refer
// to itself so that it can be recursive
var factorial = function(x) {
    if (x < 2) return 1;
```

```

    else return x * arguments.callee(x-1);
  }
  var y = factorial(5); // Returns 120

```

Arguments.length

the number of arguments passed to a function

Synopsis

`arguments.length`

Description

The `length` property of the `Arguments` object specifies the number of arguments passed to the current function. This property is defined only within a function body.

Note that this property specifies the number of arguments actually passed, not the number expected. See `Function.length` for the number of declared arguments. Note also that this property does not have any of the special behavior of the `Array.length` property.

Example

```

// Use an Arguments object to check that correct # of args were passed
function check(args) {
  var actual = args.length;           // The actual number of arguments
  var expected = args.callee.length; // The expected number of arguments
  if (actual !== expected) {          // Throw exception if they don't match
    throw new Error("Wrong number of arguments: expected: " +
                    expected + "; actually passed " + actual);
  }
}
// A function that demonstrates how to use the function above
function f(x, y, z) {
  check(arguments); // Check for correct number of arguments
  return x + y + z; // Now do the rest of the function normally
}

```

See Also

`Array.length`, `Function.length`

Array

built-in support for arrays

Object → Array

Constructor

```

new Array()
new Array(size)
new Array(element0, element1, ..., elementn)

```

Arguments

size

The desired number of elements in the array. The returned array has its `length` field set to *size*.

element0, ... elementn

An argument list of two or more arbitrary values. When the `Array()` constructor is invoked with these arguments, the newly created array is initialized with the specified argument values as its elements and its `length` field set to the number of arguments.

Returns

The newly created and initialized array. When `Array()` is invoked with no arguments, the returned array is empty and has a `length` field of 0. When invoked with a single numeric argument, the constructor returns an array with the specified number of undefined elements. When invoked with any other arguments, the constructor initializes the array with the values specified by the arguments. When the `Array()` constructor is called as a function, without the `new` operator, it behaves exactly as it does when called with the `new` operator.

Throws

`RangeError`

When a single integer *size* argument is passed to the `Array()` constructor, a `RangeError` exception is thrown if *size* is negative or is larger than $2^{32}-1$.

Literal Syntax

ECMAScript v3 specifies an array literal syntax. You may also create and initialize an array by placing a comma-separated list of expressions within square brackets. The values of these expressions become the elements of the array. For example:

```
var a = [1, true, 'abc'];
var b = [a[0], a[0]*2, f(x)];
```

Properties

`length`

A read/write integer specifying the number of elements in the array or, when the array does not have contiguous elements, a number one larger than the index of the last element in the array. Changing the value of this property truncates or extends the array.

Methods

The methods `every()`, `filter()`, `forEach()`, `indexOf()`, `lastIndexOf()`, `map()`, `reduce()`, `reduceRight()`, and `some()` are new in ECMAScript 5 but were implemented by browsers other than IE before ES5 was standardized.

`concat()`

Concatenates elements to an array.

`every()`

Test whether a predicate is true for every array element.

- filter()**
Return array elements that satisfy a predicate function.
- forEach()**
Invoke a function for each element of the array.
- indexOf()**
Search an array for a matching element.
- join()**
Converts all array elements to strings and concatenates them.
- lastIndexOf()**
Search backward through an array.
- map()**
Compute new array elements from the elements of this array.
- pop()**
Removes an item from the end of an array.
- push()**
Pushes an item to the end of an array.
- reduce()**
Compute a value from the elements of this array.
- reduceRight()**
Reduce this array from right-to-left.
- reverse()**
Reverses, in place, the order of the elements of an array.
- shift()**
Shifts an element off the beginning of an array.
- slice()**
Returns a subarray slice of an array.
- some()**
Test whether a predicate is true for at least one element of this array.
- sort()**
Sorts, in place, the elements of an array.
- splice()**
Inserts, deletes, or replaces array elements.
- toLocaleString()**
Converts an array to a localized string.
- toString()**
Converts an array to a string.
- unshift()**
Inserts elements at the beginning of an array.

`Array.concat()`

Description

Arrays are a basic feature of JavaScript and are documented in detail in Chapter 7.

See Also

Chapter 7

`Array.concat()`

concatenate arrays

Synopsis

```
array.concat(value, ...)
```

Arguments

value, ...

Any number of values to be concatenated with *array*.

Returns

A new array, which is formed by concatenating each of the specified arguments to *array*.

Description

`concat()` creates and returns a new array that is the result of concatenating each of its arguments to *array*. It does not modify *array*. If any of the arguments to `concat()` is itself an array, the elements of that array are concatenated, rather than the array itself.

Example

```
var a = [1,2,3];  
a.concat(4, 5)           // Returns [1,2,3,4,5]  
a.concat([4,5]);         // Returns [1,2,3,4,5]  
a.concat([4,5],[6,7])    // Returns [1,2,3,4,5,6,7]  
a.concat(4, [5,[6,7]])  // Returns [1,2,3,4,5,[6,7]]
```

See Also

`Array.join()`, `Array.push()`, `Array.splice()`

`Array.every()`

ECMAScript 5

test whether a predicate is true for every element

Synopsis

```
array.every(predicate)  
array.every(predicate, o)
```

Arguments

predicate

A predicate function to test array elements

o

The optional this value for invocations of *predicate*.

Returns

true if *predicate* returns true (or any truthy value) for every element of *array* or false if the predicate returns false (or a falsy value) for any element.

Description

The `every()` method tests whether some condition holds for all elements of an array. It loops through the elements of *array*, in ascending order, and invokes the specified *predicate* function on each element in turn. If *predicate* returns false (or any value that converts to false), then `every()` stops looping and returns false immediately. If every invocation of *predicate* returns true, then `every()` returns true. When invoked on an empty array, `every()` returns true.

For each array index *i*, *predicate* is invoked with three arguments:

```
predicate(array[i], i, array)
```

The return value of *predicate* is interpreted as a boolean value. true and all truthy values indicate that the array element passes the test or meets the condition described by that function. A return value of false or any falsy value means that the array element does not pass the test.

See `Array.forEach()` for further details.

Example

```
[1,2,3].every(function(x) { return x < 5; }) // => true: all elts are < 5
[1,2,3].every(function(x) { return x < 3; }) // => false: all elts are not < 3
[].every(function(x) { return false; });    // => true: always true for []
```

See Also

`Array.filter()`, `Array.forEach()`, `Array.some()`

Array.filter()

ECMAScript 5

return array elements that pass a predicate

Synopsis

```
array.map(predicate)
array.map(predicate, o)
```

`Array.forEach()`

Arguments

predicate

The function to invoke to determine whether an element of *array* will be included in the returned array.

o

An optional value on which *predicate* is invoked.

Returns

A new array containing only those elements of *array* for which *predicate* returned `true` (or a truthy value).

Description

`filter()` creates a new array and then populates it with the elements of *array* for which the *predicate* function returns `true` (or a truthy value). The `filter()` method does not modify *array* itself (though the *predicate* function may do so).

`filter()` loops through the indexes of *array*, in ascending order, and invokes *predicate* once for each element. For an index *i*, *predicate* is invoked with three arguments,

`predicate(array[i], i, array)`

If *predicate* returns `true` or a truthy value, then the element at index *i* of *array* is appended to the newly created array. Once `filter()` has tested each element of *array* it returns the new array.

See `Array.forEach()` for further details.

Example

```
[1,2,3].filter(function(x) { return x > 1; }); // => [2,3]
```

See Also

`Array.every()`, `Array.forEach()`, `Array.indexOf()`, `Array.map()`, `Array.reduce()`

Array.forEach()

ECMAScript 5

invoke a function for each array element

Synopsis

```
array.forEach(f)  
array.forEach(f, o)
```

Arguments

f

The function to invoke for each element of *array*.

o

An optional value on which *f* is invoked.

Returns

This method returns nothing.

Description

`forEach()` loops through the indexes of *array*, in ascending order, and invokes *f* once for each element. For an index *i*, *f* is invoked with three arguments:

```
f(array[i], i, array)
```

The return value, if any, of *f* is ignored. Note that `forEach()` does not have a return value. In particular, it does not return *array*.

Array Method Details

The following details apply to the `forEach()` method, and also to the related methods `map()`, `filter()`, `every()`, and `some()`.

Each of these methods expects a function as its first argument and accepts an optional second argument. If a second argument *o* is specified, then the function is invoked as if it was a method of *o*. That is, within the body of the function, `this` will evaluate to *o*. If the second argument is not specified, then the function is invoked as a function (not a method) and `this` will be the global object in non-strict code or `null` in strict code.

Each of these methods notes the length of *array* before it begins looping. If the invoked function appends new elements to *array*, those newly-added elements will not be looped over. If the function alters existing elements that have not yet been looped over, it is the altered values that will be passed.

When invoked on sparse arrays, these methods do not invoke the function for indexes at which no element actually exists.

Example

```
var a = [1,2,3];
a.forEach(function(x,i,a) { a[i]++; }); // a is now [2,3,4]
```

See Also

`Array.every()`, `Array.filter()`, `Array.indexOf()`, `Array.map()`, `Array.reduce()`

Array.indexOf()

ECMAScript 5

search an array

Synopsis

```
array.indexOf(value)
array.indexOf(value, start)
```

Arguments

value

The value to search *array* for.

start

An optional array index at which to begin the search. If omitted, 0 is used.

Returns

The lowest index $\geq$ *start* of *array* at which the element is `===` to *value*, or -1 if no such matching element exists.

Description

This method searches *array* for an element that is equal to *value*, and returns the index of the first element it finds. The search begins at the array index specified by *start*, or at index 0, and continues with successively higher indexes until a match is found or all elements have been checked. The `===` operator is used to test equality. The return value is the index of the first matching element found, or -1 if no match is found.

Example

```
[ 'a', 'b', 'c' ].indexOf('b')    // => 1
[ 'a', 'b', 'c' ].indexOf('d')    // => -1
[ 'a', 'b', 'c' ].indexOf('a', 1) // => -1
```

See Also

Array.lastIndexOf(), String.indexOf()

Array.join()

concatenate array elements to form a string

Synopsis

```
array.join()
array.join(separator)
```

Arguments**separator**

An optional character or string used to separate one element of the array from the next in the returned string. If this argument is omitted, a comma is used.

Returns

The string that results from converting each element of *array* to a string and then concatenating them together, with the *separator* string between elements.

Description

join() converts each element of an array to a string and then concatenates those strings, inserting the specified *separator* string between the elements. It returns the resulting string.

You can perform a conversion in the opposite direction—splitting a string into array elements—with the split() method of the String object. See String.split() for details.

Example

```
a = new Array(1, 2, 3, "testing");
s = a.join("+"); // s is the string "1+2+3+testing"
```

See Also

String.split()

Array.lastIndexOf()

ECMAScript 5

search backward through an array

Synopsis

```
array.lastIndexOf(value)
array.lastIndexOf(value, start)
```

Arguments

value

The value to search *array* for.

start

An optional array index at which to begin the search. If omitted, the search begins with the last element of the array.

Returns

The highest index $\leq$ *start* of *array* at which the element is `===` to *value*, or -1 if no such matching element exists.

Description

This method searches backward through successively lower elements of *array* for an element that is equal to *value*, and returns the index of the first such element it finds. If *start* is specified, it is used as the starting position for the search; otherwise the search begins at the end of the array. The `===` operator is used to test equality. The return value is the index of the first matching element found, or -1 if no match is found.

See Also

Array.indexOf(), String.lastIndexOf()

Array.length

the size of an array

Synopsis

```
array.length
```

Description

The `length` property of an array is always one larger than the index of the highest element defined in the array. For traditional “dense” arrays that have contiguous elements and begin with element 0, the `length` property specifies the number of elements in the array.

The `length` property of an array is initialized when the array is created with the `Array()` constructor method. Adding new elements to an array updates the `length`, if necessary:

```
a = new Array();           // a.length initialized to 0
b = new Array(10);         // b.length initialized to 10
c = new Array("one", "two", "three"); // c.length initialized to 3
c[3] = "four";             // c.length updated to 4
c[10] = "blastoff";        // c.length becomes 11
```

You can set the value of the `length` property to change the size of an array. If you set `length` to be smaller than its previous value, the array is truncated, and elements at the end are lost. If you set `length` to be larger than its previous value, the array becomes bigger, and the new elements added at the end of the array have the `undefined` value.

Array.map()

ECMAScript 5

compute new array elements from old

Synopsis

```
array.map(f)
array.map(f, o)
```

Arguments

f

The function to invoke for each element of *array*. Its return values become elements of the returned array.

o

An optional value on which *f* is invoked.

Returns

A new array with elements computed by the function *f*.

Description

`map()` creates a new array with the same length as *array*, and computes the elements of this new array by passing the elements of *array* to the function *f*. `map()` loops through the indexes of *array*, in ascending order, and invokes *f* once for each element. For an index *i*, *f* is invoked with three arguments, and its return value is stored at index *i* of the newly created array:

```
a[i] = f(array[i], i, array)
```

Once `map()` has passed each element of *array* to *f* and has stored the results in the new array, it returns that new array.

See `Array.forEach()` for further details.

Example

```
[1,2,3].map(function(x) { return x*x; }); // => [1,4,9]
```

See Also

Array.every(), Array.filter(), Array.forEach(), Array.indexOf(), Array.reduce()

Array.pop()

remove and return the last element of an array

Synopsis

```
array.pop()
```

Returns

The last element of *array*.

Description

`pop()` deletes the last element of *array*, decrements the array length, and returns the value of the element that it deleted. If the array is already empty, `pop()` does not change the array and returns the undefined value.

Example

`pop()`, and its companion method `push()`, provide the functionality of a first-in, last-out stack. For example:

```
var stack = [];           // stack: []
stack.push(1, 2);         // stack: [1,2]      Returns 2
stack.pop();              // stack: [1]        Returns 2
stack.push([4,5]);        // stack: [1,[4,5]]   Returns 2
stack.pop();              // stack: [1]        Returns [4,5]
stack.pop();              // stack: []         Returns 1
```

See Also

Array.push()

Array.push()

append elements to an array

Synopsis

```
array.push(value, ...)
```

Arguments

value, ...

One or more values to be appended to the end of *array*.

Returns

The new length of the array, after the specified values are appended to it.

Description

`push()` appends its arguments, in order, to the end of *array*. It modifies *array* directly, rather than creating a new array. `push()`, and its companion method `pop()`, use arrays to provide the functionality of a first in, last out stack. See `Array.pop()` for an example.

See Also

`Array.pop()`

Array.reduce()

ECMAScript 5

compute a value from the elements of an array

Synopsis

```
array.reduce(f)
array.reduce(f, initial)
```

Arguments

f

A function that combines two values (such as two array elements) and returns a new “reduced” value.

initial

An optional initial value to seed the array reduction with. If this argument is specified, `reduce()` behaves as if it had been inserted at the start of *array*.

Returns

The reduced value of the array, which is the return value of the last invocation of *f*.

Description

The `reduce()` method expects a function *f* as its first argument. This function should behave like a binary operator: it takes two values, performs some operation on them and returns a result. If *array* has *n* elements, the `reduce()` method invokes *f* *n*-1 times to reduce those elements to a single combined value. (You may be familiar with this array reduction operation from other programming languages where it is sometimes called “fold” or “inject”.)

The first invocation of *f* is passed the first two elements of *array*. Each subsequent invocation of *f* is passed the return value of the previous invocation and the next element (in ascending order) of *array*. The return value of the final invocation of *f* becomes the return value of the `reduce()` method itself.

`reduce()` may be invoked with an optional second argument, *initial*. If *initial* is specified, then `reduce()` behaves as if that argument has been inserted at the beginning of *array* (it does not actually modify *array*, however). Another way to say this is that if `reduce()` is invoked with two arguments, then *initial* is used as if it had previously been returned from *f*. In this

case, the first invocation of *f* is passed *initial* and the first element of *array*. When *initial* is specified, there are *n*+1 elements to reduce (*n* elements of *array*, plus the *initial* value) and *f* is invoked *n* times.

If *array* is empty and *initial* is not specified, `reduce()` throws a `TypeError`. If *array* is empty and *initial* is specified, then `reduce()` returns *initial* and never calls *f*. If *array* has only a single element and *initial* is not specified, then `reduce()` returns the single element of *array* without calling *f*.

The paragraphs above describe two arguments to *f*, but `reduce()` actually invokes that function with four arguments. The third argument is the array index of the second argument. The fourth argument is *array* itself. *f* is always invoked as a function, not as a method.

Example

```
[1,2,3,4].reduce(function(x,y) { return x*y; }) // => 24: ((1*2)*3)*4
```

See Also

`Array.forEach()`, `Array.map()`, `Array.reduceRight()`

Array.reduceRight()

ECMAScript 5

reduce an array from right-to-left

Synopsis

```
array.reduceRight(f)
array.reduceRight(f, initial)
```

Arguments

f

A function that combines two values (such as two array elements) and returns a new “reduced” value.

initial

An optional initial value to seed the array reduction with. If this argument is specified, `reduceRight()` behaves as if it had been inserted at the end of *array*.

Returns

The reduced value of the array, which is the return value of the last invocation of *f*.

Description

`reduceRight()` works like the `reduce()` method: it invokes the function *f* *n*-1 times to reduce the *n* elements of *array* to a single value. `reduceRight()` differs from `reduce()` only in that it enumerates array elements from right to left (from highest index to lowest) rather than left to right. See `Array.reduce()` for details.

Example

```
[2, 10, 60].reduceRight(function(x,y) { return x/y }) // => 3: (60/10)/2
```

`Array.reverse()`

See Also

`Array.reduce()`

Array.reverse()

reverse the elements of an array

Synopsis

```
array.reverse()
```

Description

The `reverse()` method of an Array object reverses the order of the elements of an array. It does this *in place*: it rearranges the elements of the specified *array* without creating a new array. If there are multiple references to *array*, the new order of the array elements is visible through all references.

Example

```
a = new Array(1, 2, 3);    // a[0] == 1, a[2] == 3;
a.reverse();               // Now a[0] == 3, a[2] == 1;
```

Array.shift()

shift array elements down

Synopsis

```
array.shift()
```

Returns

The former first element of the array.

Description

`shift()` removes and returns the first element of *array*, shifting all subsequent elements down one place to occupy the newly vacant space at the start of the array. If the array is empty, `shift()` does nothing and returns the undefined value. Note that `shift()` does not create a new array; instead, it modifies *array* directly.

`shift()` is similar to `Array.pop()`, except it operates on the beginning of an array rather than the end. `shift()` is often used in conjunction with `unshift()`.

Example

```
var a = [1, [2,3], 4]
a.shift(); // Returns 1; a = [[2,3], 4]
a.shift(); // Returns [2,3]; a = [4]
```

See Also

`Array.pop()`, `Array.unshift()`

Array.slice()

return a portion of an array

Synopsis

```
array.slice(start, end)
```

Arguments

start

The array index at which the slice is to begin. If negative, this argument specifies a position measured from the end of the array. That is, -1 indicates the last element, -2 indicates the next from the last element, and so on.

end

The array index immediately after the end of the slice. If not specified, the slice includes all array elements from the *start* to the end of the array. If this argument is negative, it specifies an array element measured from the end of the array.

Returns

A new array that contains the elements of *array* from the element specified by *start*, up to, but not including, the element specified by *end*.

Description

`slice()` returns a slice, or subarray, of *array*. The returned array contains the element specified by *start* and all subsequent elements up to, but not including, the element specified by *end*. If *end* is not specified, the returned array contains all elements from the *start* to the end of *array*.

Note that `slice()` does not modify the array. If you want to actually remove a slice of an array, use `Array.splice()`.

Example

```
var a = [1,2,3,4,5];
a.slice(0,3);    // Returns [1,2,3]
a.slice(3);      // Returns [4,5]
a.slice(1,-1);   // Returns [2,3,4]
a.slice(-3,-2);  // Returns [3]; buggy in IE 4; returns [1,2,3]
```

Bugs

start can't be a negative number in Internet Explorer 4. This is fixed in later versions of IE.

See Also

`Array.splice()`

Array.some()

test whether a predicate is true for any element

Synopsis

```
array.some(predicate)  
array.some(predicate, o)
```

Arguments

predicate

A predicate function to test array elements

o

The optional this value for invocations of *predicate*.

Returns

true if *predicate* returns true (or a truthy value) for at least one element of *array* or false if the predicate returns false (or a falsy value) for all elements.

Description

The `some()` method tests whether a condition holds for at least one element of an array. It loops through the elements of *array*, in ascending order, and invokes the specified *predicate* function on each element in turn. If *predicate* returns true (or a value that converts to true), then `some()` stops looping and returns true immediately. If every invocation of *predicate* returns false (or a value that converts to false), then `some()` returns false. When invoked on an empty array, `some()` returns false.

This method is very much like `every()`. See `Array.every()` and `Array.forEach()` for further details.

Example

```
[1,2,3].some(function(x) { return x > 5; }) // => false: no elts are > 5  
[1,2,3].some(function(x) { return x > 2; }) // => true: some elts are > 3  
[].some(function(x) { return true; });      // => false: always false for []
```

See Also

`Array.every()`, `Array.filter()`, `Array.forEach()`

Array.sort()

sort the elements of an array

Synopsis

```
array.sort()  
array.sort(orderfunc)
```

Arguments

orderfunc

An optional function used to specify the sorting order.

Returns

A reference to the array. Note that the array is sorted in place, and no copy is made.

Description

The `sort()` method sorts the elements of *array* in place: no copy of the array is made. If `sort()` is called with no arguments, the elements of the array are arranged in alphabetical order (more precisely, the order determined by the character encoding). To do this, elements are first converted to strings, if necessary, so that they can be compared.

If you want to sort the array elements in some other order, you must supply a comparison function that compares two values and returns a number indicating their relative order. The comparison function should take two arguments, *a* and *b*, and should return one of the following:

- A value less than zero, if, according to your sort criteria, *a* is less than *b* and should appear before *b* in the sorted array.
- Zero, if *a* and *b* are equivalent for the purposes of this sort.
- A value greater than zero, if *a* is greater than *b* for the purposes of the sort.

Note that undefined elements of an array are always sorted to the end of the array. This is true even if you provide a custom ordering function: undefined values are never passed to the *orderfunc* you supply.

Example

The following code shows how you might write a comparison function to sort an array of numbers in numerical, rather than alphabetical order:

```
// An ordering function for a numerical sort
function numberorder(a, b) { return a - b; }
a = new Array(33, 4, 1111, 222);
a.sort();           // Alphabetical sort: 1111, 222, 33, 4
a.sort(numberorder); // Numerical sort: 4, 33, 222, 1111
```

Array.splice()

insert, remove, or replace array elements

Synopsis

```
array.splice(start, deleteCount, value, ...)
```

Arguments

start

The array element at which the insertion and/or deletion is to begin.

deleteCount

The number of elements, starting with and including *start*, to be deleted from *array*. Specify 0 to insert elements without deleting any.

value, ...

Zero or more values to be inserted into *array*, beginning at the index specified by *start*.

Returns

An array containing the elements, if any, deleted from *array*.

Description

`splice()` deletes zero or more array elements starting with and including the element *start* and replaces them with zero or more values specified in the argument list. Array elements that appear after the insertion or deletion are moved as necessary so that they remain contiguous with the rest of the array. Note that, unlike the similarly named `slice()`, `splice()` modifies *array* directly.

Example

The operation of `splice()` is most easily understood through an example:

```
var a = [1,2,3,4,5,6,7,8]
a.splice(1,2);    // Returns [2,3]; a is [1,4]
a.splice(1,1);    // Returns [4]; a is [1]
a.splice(1,0,2,3); // Returns []; a is [1 2 3]
```

See Also

`Array.slice()`

Array.toLocaleString()

convert an array to a localized string

Overrides `Object.toLocaleString()`

Synopsis

```
array.toLocaleString()
```

Returns

A localized string representation of *array*.

Throws

`TypeError`

If this method is invoked on an object that is not an Array.

Description

The `toLocaleString()` method of an array returns a localized string representation of an array. It does this by calling the `toLocaleString()` method of all of the array elements, then concatenating the resulting strings using a locale-specific separator character.

See Also

Array.toString(), Object.toLocaleString()

Array.toString()

convert an array to a string

Overrides Object.toString()

Synopsis

```
array.toString()
```

Returns

A string representation of *array*.

Throws

TypeError

If this method is invoked on an object that is not an Array.

Description

The `toString()` method of an array converts an array to a string and returns the string. When an array is used in a string context, JavaScript automatically converts it to a string by calling this method. On some occasions, however, you may want to call `toString()` explicitly.

`toString()` converts an array to a string by first converting each array element to strings (by calling its `toString()` method). Once each element is converted to a string, `toString()` outputs them in a comma-separated list. This return value is the same string that would be returned by the `join()` method with no arguments.

See Also

Array.toLocaleString(), Object.toString()

Array.unshift()

Insert elements at the beginning of an array

Synopsis

```
array.unshift(value, ...)
```

Arguments

value, ...

One or more values that are inserted at the start of *array*.

Returns

The new length of the array

Description

`unshift()` inserts its arguments at the beginning of *array*, shifting the existing elements to higher indexes to make room. The first argument to `shift()` becomes the new element 0 of the array; the second argument, if any, becomes the new element 1; and so on. Note that `unshift()` does not create a new array; it modifies *array* directly.

Example

`unshift()` is often used in conjunction with `shift()`. For example:

```
var a = [];           // a:[]
a.unshift(1);         // a:[1]      Returns: 1
a.unshift(22);        // a:[22,1]   Returns: 2
a.shift();            // a:[1]      Returns: 22
a.unshift(33,[4,5]);  // a:[33,[4,5],1] Returns: 3
```

See Also

`Array.shift()`

Boolean

support for boolean values

Object → Boolean

Constructor

```
new Boolean(value) // Constructor function
Boolean(value)     // Conversion function
```

Arguments

value

The value to be held by the Boolean object or to be converted to a boolean value.

Returns

When invoked as a constructor with the `new` operator, `Boolean()` converts its argument to a boolean value and returns a Boolean object that contains that value. When invoked as a function, without the `new` operator, `Boolean()` simply converts its argument to a primitive boolean value and returns that value.

The values 0, NaN, null, the empty string "", and the undefined value are all converted to false. All other primitive values, except false (but including the string "false"), and all objects and arrays are converted to true.

Methods

`toString()`

Returns "true" or "false", depending on the boolean value represented by the Boolean object.

`valueOf()`

Returns the primitive boolean value contained in the Boolean object.

Description

Boolean values are a fundamental datatype in JavaScript. The Boolean object is an object wrapper around the boolean value. This Boolean object type exists primarily to provide a `toString()` method to convert boolean values to strings. When the `toString()` method is invoked to convert a boolean value to a string (and it is often invoked implicitly by JavaScript), JavaScript internally converts the boolean value to a transient Boolean object, on which the method can be invoked.

See Also

Object

Boolean.toString()

convert a boolean value to a string

Overrides `Object.toString()`

Synopsis

`b.toString()`

Returns

The string "true" or "false", depending on the value of the primitive boolean value or Boolean object *b*.

Throws

`TypeError`

If this method is invoked on an object that is not a Boolean.

Boolean.valueOf()

the boolean value of a Boolean object

Overrides `Object.valueOf()`

Synopsis

`b.valueOf()`

Returns

The primitive boolean value held by the Boolean object *b*.

Throws

`TypeError`

If this method is invoked on an object that is not a Boolean.

Date

manipulate dates and times

Object → Date

Constructor

```
new Date()  
new Date(milliseconds)  
new Date(datestring)  
new Date(year, month, day, hours, minutes, seconds, ms)
```

With no arguments, the `Date()` constructor creates a `Date` object set to the current date and time. When one numeric argument is passed, it is taken as the internal numeric representation of the date in milliseconds, as returned by the `getTime()` method. When one string argument is passed, it is a string representation of a date, in the format accepted by the `Date.parse()` method. Otherwise, the constructor is passed between two and seven numeric arguments that specify the individual fields of the date and time. All but the first two arguments—the year and month fields—are optional. Note that these date and time fields are specified using local time, not Coordinated Universal Time (UTC) (which is similar to Greenwich Mean Time [GMT]). See the static `Date.UTC()` method for an alternative.

`Date()` may also be called as a function, without the `new` operator. When invoked in this way, `Date()` ignores any arguments passed to it and returns a string representation of the current date and time.

Arguments

milliseconds

The number of milliseconds between the desired date and midnight on January 1, 1970 (UTC). For example, passing the argument 5000 creates a date that represents five seconds past midnight on 1/1/70.

datestring

A single argument that specifies the date and, optionally, the time as a string. The string should be in a format accepted by `Date.parse()`.

year

The year, in four-digit format. For example, specify 2001 for the year 2001. For compatibility with early implementations of JavaScript, if this argument is between 0 and 99, 1900 is added to it.

month

The month, specified as an integer from 0 (January) to 11 (December).

day

The day of the month, specified as an integer from 1 to 31. Note that this argument uses 1 as its lowest value, while other arguments use 0 as their lowest value. Optional.

hours

The hour, specified as an integer from 0 (midnight) to 23 (11 p.m.). Optional.

minutes

The minutes in the hour, specified as an integer from 0 to 59. Optional.

seconds

The seconds in the minute, specified as an integer from 0 to 59. Optional.

ms

The milliseconds in the second, specified as an integer from 0 to 999. Optional.

Methods

The Date object has no properties that can be read and written directly; instead, all access to date and time values is done through methods. Most methods of the Date object come in two forms: one that operates using local time and one that operates using universal (UTC or GMT) time. If a method has “UTC” in its name, it operates using universal time. These pairs of methods are listed together below. For example, the listing for `get[UTC]Day()` refers to both the methods `getDay()` and `getUTCDay()`.

Date methods may be invoked only on Date objects, and they throw a `TypeError` exception if you attempt to invoke them on any other type of object:

`get[UTC]Date()`

Returns the day of the month of a Date object, in local or universal time.

`get[UTC]Day()`

Returns the day of the week of a Date object, in local or universal time.

`get[UTC]FullYear()`

Returns the year of the date in full four-digit form, in local or universal time.

`get[UTC]Hours()`

Returns the hours field of a Date object, in local or universal time.

`get[UTC]Milliseconds()`

Returns the milliseconds field of a Date object, in local or universal time.

`get[UTC]Minutes()`

Returns the minutes field of a Date object, in local or universal time.

`get[UTC]Month()`

Returns the month field of a Date object, in local or universal time.

`get[UTC]Seconds()`

Returns the seconds field of a Date object, in local or universal time.

`getTime()`

Returns the internal, millisecond representation of a Date object. Note that this value is independent of time zone, and therefore, there is not a separate `getUTCTime()` method.

`getTimezoneOffset()`

Returns the difference, in minutes, between the local and UTC representations of this date. Note that the value returned depends on whether daylight saving time is or would be in effect at the specified date.

`getYear()`

Returns the year field of a Date object. Deprecated in favor of `getFullYear()`.

`set[UTC]Date()`

Sets the day of the month field of the date, using local or universal time.

`set[UTC]FullYear()`

Sets the year (and optionally month and day) field of the date, using local or universal time.

`set[UTC]Hours()`

Sets the hour field (and optionally the minutes, seconds, and milliseconds fields) of the date, using local or universal time.

`set[UTC]Milliseconds()`

Sets the milliseconds field of a date, using local or universal time.

`set[UTC]Minutes()`

Sets the minutes field (and optionally the seconds and milliseconds fields) of a date, using local or universal time.

`set[UTC]Month()`

Sets the month field (and optionally the day of the month) of a date, using local or universal time.

`set[UTC]Seconds()`

Sets the seconds field (and optionally the milliseconds field) of a date, using local or universal time.

`setTime()`

Sets the fields of a Date object using the millisecond format.

`setYear()`

Sets the year field of a Date object. Deprecated in favor of `setFullYear()`.

`toString()`

Returns a string that represents the date portion of the date, expressed in the local time zone.

`toGMTString()`

Converts a Date to a string, using the GMT time zone. Deprecated in favor of `toUTCString()`.

`toISOString()`

Converts a Date to a string, using the ISO-8601 standard combined date/time format and UTC.

`toJSON()`

JSON serializes a Date object, using `toISOString()`.

`toLocaleDateString()`

Returns a string that represents the date portion of the date, expressed in the local time zone, using the local date formatting conventions.

`toLocaleString()`

Converts a Date to a string, using the local time zone and the local date formatting conventions.

toLocaleTimeString()

Returns a string that represents the time portion of the date, expressed in the local time zone, using the local time formatting conventions.

toString()

Converts a Date to a string using the local time zone.

toTimeString()

Returns a string that represents the time portion of the date, expressed in the local time zone.

toUTCString()

Converts a Date to a string, using universal time.

valueOf()

Converts a Date to its internal millisecond format.

Static Methods

In addition to the many instance methods listed previously, the Date object also defines three static methods. These methods are invoked through the Date() constructor itself, not through individual Date objects:

Date.now()

Returns the current time, as milliseconds since the epoch.

Date.parse()

Parses a string representation of a date and time and returns the internal millisecond representation of that date.

Date.UTC()

Returns the millisecond representation of the specified UTC date and time.

Description

The Date object is a datatype built into the JavaScript language. Date objects are created with the new Date() syntax shown earlier.

Once a Date object is created, a number of methods allow you to operate on it. Most methods simply allow you to get and set the year, month, day, hour, minute, second, and millisecond fields of the object, using either local time or UTC (universal, or GMT) time. The toString() method and its variants convert dates to human-readable strings. getTime() and setTime() convert to and from the internal representation of the Date object—the number of milliseconds since midnight (GMT) on January 1, 1970. In this standard millisecond format, a date and time are represented by a single integer, which makes date arithmetic particularly easy. The ECMAScript standard requires the Date object to be able to represent any date and time, to millisecond precision, within 100 million days before or after 1/1/1970. This is a range of plus or minus 273,785 years, so the JavaScript clock will not “roll over” until the year 275755.

Examples

Once you create a Date object, there are a variety of methods you can use to operate on it:

```
d = new Date(); // Get the current date and time
document.write('Today is: ' + d.toLocaleDateString() + '. '); // Display date
document.write('The time is: ' + d.toLocaleTimeString()); // Display time
var dayOfWeek = d.getDay(); // What weekday is it?
var weekend = (dayOfWeek == 0) || (dayOfWeek == 6); // Is it a weekend?
```

Another common use of the Date object is to subtract the millisecond representations of the current time from some other time to determine the difference between the two times. The following client-side example shows two such uses:

```
<script language="JavaScript">
today = new Date(); // Make a note of today's date
christmas = new Date(); // Get a date with the current year
christmas.setMonth(11); // Set the month to December...
christmas.setDate(25); // and the day to the 25th
// If Christmas hasn't already passed, compute the number of
// milliseconds between now and Christmas, convert this
// to a number of days and print a message
if (today.getTime() < christmas.getTime()) {
    difference = christmas.getTime() - today.getTime();
    difference = Math.floor(difference / (1000 * 60 * 60 * 24));
    document.write('Only ' + difference + ' days until Christmas!<p>');
}
</script>
// ... rest of HTML document here ...
<script language="JavaScript">
// Here we use Date objects for timing
// We divide by 1000 to convert milliseconds to seconds
now = new Date();
document.write('<p>It took ' +
    (now.getTime() - today.getTime()) / 1000 +
    'seconds to load this page.');
```

See Also

Date.parse(), Date.UTC()

Date.getDate()

return the day-of-the-month field of a Date

Synopsis

```
date.getDate()
```

Returns

The day of the month of the specified Date object *date*, using local time. Return values are between 1 and 31.

Date.getDay()

return the day-of-the-week field of a Date

Synopsis

```
date.getDay()
```

Returns

The day of the week of the specified Date object *date*, using local time. Return values are between 0 (Sunday) and 6 (Saturday).

Date.getFullYear()

return the year field of a Date

Synopsis

```
date.getFullYear()
```

Returns

The year that results when *date* is expressed in local time. The return value is a full four-digit year, including the century, not a two-digit abbreviation.

Date.getHours()

return the hours field of a Date

Synopsis

```
date.getHours()
```

Returns

The hours field, expressed in local time, of the specified Date object *date*. Return values are between 0 (midnight) and 23 (11 p.m.).

Date.getMilliseconds()

return the milliseconds field of a Date

Synopsis

```
date.getMilliseconds()
```

Returns

The milliseconds field, expressed in local time, of *date*.

Date.getMinutes()

return the minutes field of a Date

Synopsis

date.getMinutes()

Returns

The minutes field, expressed in local time, of the specified Date object *date*. Return values are between 0 and 59.

Date.getMonth()

return the month field of a Date

Synopsis

date.getMonth()

Returns

The month field, expressed in local time, of the specified Date object *date*. Return values are between 0 (January) and 11 (December).

Date.getSeconds()

return the seconds field of a Date

Synopsis

date.getSeconds()

Returns

The seconds field, expressed in local time, of the specified Date object *date*. Return values are between 0 and 59.

Date.getTime()

return a Date in milliseconds

Synopsis

date.getTime()

Returns

The millisecond representation of the specified Date object *date*—that is, the number of milliseconds between midnight (GMT) on 1/1/1970 and the date and time specified by *date*.

Description

`getTime()` converts a date and time to a single integer. This is useful when you want to compare two `Date` objects or to determine the time elapsed between two dates. Note that the millisecond representation of a date is independent of the time zone, so there is no `getUTCTime()` method in addition to this one. Don't confuse this `getTime()` method with the `getDay()` and `getDate()` methods, which return the day of the week and the day of the month, respectively.

`Date.parse()` and `Date.UTC()` allow you to convert a date and time specification to a millisecond representation without going through the overhead of first creating a `Date` object.

See Also

`Date`, `Date.parse()`, `Date.setTime()`, `Date.UTC()`

Date.getTimezoneOffset()

determine the offset from GMT

Synopsis

```
date.getTimezoneOffset()
```

Returns

The difference, in minutes, between GMT and local time.

Description

`getTimezoneOffset()` returns the number of minutes difference between the GMT or UTC time and the local time. In effect, this function tells you what time zone the JavaScript code is running in and whether or not daylight saving time is (or would be) in effect at the specified *date*.

The return value is measured in minutes, rather than hours, because some countries have time zones that are not at even one-hour intervals.

Date.getUTCDate()

return the day-of-the-month field of a `Date` (universal time)

Synopsis

```
date.getUTCDate()
```

Returns

The day of the month (a value between 1 and 31) that results when *date* is expressed in universal time.

Date.getUTCDay()

return the day-of-the-week field of a Date (universal time)

Synopsis

date.getUTCDay()

Returns

The day of the week that results when *date* is expressed in universal time. Return values are between 0 (Sunday) and 6 (Saturday).

Date.getUTCFullYear()

return the year field of a Date (universal time)

Synopsis

date.getUTCFullYear()

Returns

The year that results when *date* is expressed in universal time. The return value is a full four-digit year, not a two-digit abbreviation.

Date.getUTCHours()

return the hours field of a Date (universal time)

Synopsis

date.getUTCHours()

Returns

The hours field, expressed in universal time, of *date*. The return value is an integer between 0 (midnight) and 23 (11 p.m.).

Date.getUTCMilliseconds()

return the milliseconds field of a Date (universal time)

Synopsis

date.getUTCMilliseconds()

Returns

The milliseconds field, expressed in universal time, of *date*.

Date.getUTCMinutes()

return the minutes field of a Date (universal time)

Synopsis

date.getUTCMinutes()

Returns

The minutes field, expressed in universal time, of *date*. The return value is an integer between 0 and 59.

Date.getUTCMonth()

return the month-of-the-year field of a Date (universal time)

Synopsis

date.getUTCMonth()

Returns

The month of the year that results when *date* is expressed in universal time. The return value is an integer between 0 (January) and 11 (December). Note that the Date object represents the first day of the month as 1 but represents the first month of the year as 0.

Date.getUTCSeconds()

return the seconds field of a Date (universal time)

Synopsis

date.getUTCSeconds()

Returns

The seconds field, expressed in universal time, of *date*. The return value is an integer between 0 and 59.

Date.getYear()

deprecated

return the year field of a Date

Synopsis

date.getYear()

Returns

The year field of the specified Date object *date* minus 1900.

`Date.now()`

Description

`getFullYear()` returns the year field of a specified `Date` object minus 1900. As of ECMAScript v3, it is not required in conforming JavaScript implementations; use `getFullYear()` instead.

`Date.now()`

ECMAScript 5

return the current time in milliseconds

Synopsis

```
Date.now()
```

Returns

The current time, in milliseconds since midnight GMT on January 1, 1970.

Description

Prior to ECMAScript 5, you can implement this method like this:

```
Date.now = function() { return (new Date()).getTime(); }
```

See Also

`Date`, `Date.getTime()`

`Date.parse()`

parse a date/time string

Synopsis

```
Date.parse(date)
```

Arguments

date

A string containing the date and time to be parsed.

Returns

The number of milliseconds between the specified date and time and midnight GMT on January 1, 1970.

Description

`Date.parse()` is a static method of `Date`. It parses the date specified by its single string argument and returns it as the number of milliseconds since the epoch. This return value can be used directly, used to create a new `Date` object, or used to set the date in an existing `Date` object with `Date.setTime()`.

ECMAScript 5 requires this method to be able to parse strings returned by the `Date.toISOString()` method. In ECMAScript 5 and before, this method is also required to be able to

parse the implementation-dependent strings returned by the `toUTCString()` and `toString()` methods.

See Also

`Date`, `Date.setTime()`, `Date.toISOString()`, `Date.toString()`

Date.setDate()

set the day-of-the-month field of a `Date`

Synopsis

```
date.setDate(day_of_month)
```

Arguments

day_of_month

An integer between 1 and 31 that is used as the new value (in local time) of the day-of-the-month field of *date*.

Returns

The millisecond representation of the adjusted date. Prior to ECMAScript standardization, this method returns nothing.

Date.setFullYear()

set the year and, optionally, the month and date fields of a `Date`

Synopsis

```
date.setFullYear(year)
date.setFullYear(year, month)
date.setFullYear(year, month, day)
```

Arguments

year

The year, expressed in local time, to be set in *date*. This argument should be an integer that includes the century, such as 1999; it should not be an abbreviation, such as 99.

month

An optional integer between 0 and 11 that is used as the new value (in local time) of the month field of *date*.

day

An optional integer between 1 and 31 that is used as the new value (in local time) of the day-of-the-month field of *date*.

Returns

The internal millisecond representation of the adjusted date.

Date.setHours()

set the hours, minutes, seconds, and milliseconds fields of a Date

Synopsis

```
date.setHours(hours)
date.setHours(hours, minutes)
date.setHours(hours, minutes, seconds)
date.setHours(hours, minutes, seconds, millis)
```

Arguments

hours

An integer between 0 (midnight) and 23 (11 p.m.) local time that is set as the new hours value of *date*.

minutes

An optional integer, between 0 and 59, that is used as the new value (in local time) of the minutes field of *date*. This argument is not supported prior to ECMAScript standardization.

seconds

An optional integer, between 0 and 59, that is used as the new value (in local time) of the seconds field of *date*. This argument is not supported prior to ECMAScript standardization.

millis

An optional integer, between 0 and 999, that is used as the new value (in local time) of the milliseconds field of *date*. This argument is not supported prior to ECMAScript standardization.

Returns

The millisecond representation of the adjusted date. Prior to ECMAScript standardization, this method returns nothing.

Date.setMilliseconds()

set the milliseconds field of a Date

Synopsis

```
date.setMilliseconds(millis)
```

Arguments

millis

The milliseconds field, expressed in local time, to be set in *date*. This argument should be an integer between 0 and 999.

Returns

The millisecond representation of the adjusted date.

Date.setMinutes()

set the minutes, seconds, and milliseconds fields of a Date

Synopsis

```
date.setMinutes(minutes)
date.setMinutes(minutes, seconds)
date.setMinutes(minutes, seconds, millis)
```

Arguments

minutes

An integer between 0 and 59 that is set as the minutes value (in local time) of the Date object *date*.

seconds

An optional integer, between 0 and 59, that is used as the new value (in local time) of the seconds field of *date*. This argument is not supported prior to ECMAScript standardization.

millis

An optional integer, between 0 and 999, that is used as the new value (in local time) of the milliseconds field of *date*. This argument is not supported prior to ECMAScript standardization.

Returns

The millisecond representation of the adjusted date. Prior to ECMAScript standardization, this method returns nothing.

Date.setMonth()

set the month and day fields of a Date

Synopsis

```
date.setMonth(month)
date.setMonth(month, day)
```

Arguments

month

An integer between 0 (January) and 11 (December) that is set as the month value (in local time) for the Date object *date*. Note that months are numbered beginning with 0, while days within the month are numbered beginning with 1.

day

An optional integer between 1 and 31 that is used as the new value (in local time) of the day-of-the-month field of *date*. This argument is not supported prior to ECMAScript standardization.

`Date.setSeconds()`

Returns

The millisecond representation of the adjusted date. Prior to ECMAScript standardization, this method returns nothing.

Date.setSeconds()

set the seconds and milliseconds fields of a Date

Synopsis

```
date.setSeconds(seconds)  
date.setSeconds(seconds, millis)
```

Arguments

seconds

An integer between 0 and 59 that is set as the seconds value for the Date object *date*.

millis

An optional integer, between 0 and 999, that is used as the new value (in local time) of the milliseconds field of *date*. This argument is not supported prior to ECMAScript standardization.

Returns

The millisecond representation of the adjusted date. Prior to ECMAScript standardization, this method returns nothing.

Date.setTime()

set a Date in milliseconds

Synopsis

```
date.setTime(milliseconds)
```

Arguments

milliseconds

The number of milliseconds between the desired date and time and midnight GMT on January 1, 1970. A millisecond value of this type may also be passed to the `Date()` constructor and may be obtained by calling the `Date.UTC()` and `Date.parse()` methods. Representing a date in this millisecond format makes it independent of time zone.

Returns

The *milliseconds* argument. Prior to ECMAScript standardization, this method returns nothing.

Date.setUTCDate()

set the day-of-the-month field of a Date (universal time)

Synopsis

```
date.setUTCDate(day_of_month)
```

Arguments

day_of_month

The day of the month, expressed in universal time, to be set in *date*. This argument should be an integer between 1 and 31.

Returns

The internal millisecond representation of the adjusted date.

Date.setUTCFullYear()

set the year, month, and day fields of a Date (universal time)

Synopsis

```
date.setUTCFullYear(year)  
date.setSeconds(seconds, millis)  
date.setUTCFullYear(year, month, day)
```

Arguments

year

The year, expressed in universal time, to be set in *date*. This argument should be an integer that includes the century, such as 1999, not an abbreviation, such as 99.

month

An optional integer between 0 and 11 that is used as the new value (in universal time) of the month field of *date*. Note that months are numbered beginning with 0, while days within the month are numbered beginning with 1.

day

An optional integer between 1 and 31 that is used as the new value (in universal time) of the day-of-the-month field of *date*.

Returns

The internal millisecond representation of the adjusted date.

Date.setUTCHours()

set the hours, minutes, seconds, and milliseconds fields of a Date (universal time)

Synopsis

```
date.setUTCHours(hours)  
date.setUTCHours(hours, minutes)
```

Date.setUTCMilliseconds()

```
date.setUTCHours(hours, minutes, seconds)
date.setUTCHours(hours, minutes, seconds, millis)
```

Arguments

hours

The hours field, expressed in universal time, to be set in *date*. This argument should be an integer between 0 (midnight) and 23 (11 p.m.).

minutes

An optional integer, between 0 and 59, that is used as the new value (in universal time) of the minutes field of *date*.

seconds

An optional integer, between 0 and 59, that is used as the new value (in universal time) of the seconds field of *date*.

millis

An optional integer, between 0 and 999, that is used as the new value (in universal time) of the milliseconds field of *date*.

Returns

The millisecond representation of the adjusted date.

Date.setUTCMilliseconds()

set the milliseconds field of a Date (universal time)

Synopsis

```
date.setUTCMilliseconds(millis)
```

Arguments

millis

The milliseconds field, expressed in universal time, to be set in *date*. This argument should be an integer between 0 and 999.

Returns

The millisecond representation of the adjusted date.

Date.setUTCMinutes()

set the minutes, seconds, and milliseconds fields of a Date (universal time)

Synopsis

```
date.setUTCMinutes(minutes)
date.setUTCMinutes(minutes, seconds)
date.setUTCMinutes(minutes, seconds, millis)
```

Arguments

minutes

The minutes field, expressed in universal time, to be set in *date*. This argument should be an integer between 0 and 59.

seconds

An optional integer between 0 and 59 that is used as the new value (in universal time) of the seconds field of *date*.

millis

An optional integer between 0 and 999 that is used as the new value (in universal time) of the milliseconds field of *date*.

Returns

The millisecond representation of the adjusted date.

Date.setUTCMonth()

set the month and day fields of a Date (universal time)

Synopsis

```
date.setUTCMonth(month)
```

```
date.setUTCMonth(month, day)
```

Arguments

month

The month, expressed in universal time, to be set in *date*. This argument should be an integer between 0 (January) and 11 (December). Note that months are numbered beginning with 0, while days within the month are numbered beginning with 1.

day

An optional integer between 1 and 31 that is used as the new value (in universal time) of the day-of-the-month field of *date*.

Returns

The millisecond representation of the adjusted date.

Date.setUTCSeconds()

set the seconds and milliseconds fields of a Date (universal time)

Synopsis

```
date.setUTCSeconds(seconds)
```

```
date.setUTCSeconds(seconds, millis)
```

`Date.setYear()`

Arguments

seconds

The seconds field, expressed in universal time, to be set in *date*. This argument should be an integer between 0 and 59.

millis

An optional integer between 0 and 999 that is used as the new value (in universal time) of the milliseconds field of *date*.

Returns

The millisecond representation of the adjusted date.

Date.setYear()

deprecated

set the year field of a Date

Synopsis

date.setYear(*year*)

Arguments

year

An integer that is set as the year value (in local time) for the Date object *date*. If this value is between 0 and 99, inclusive, 1900 is added to it and it is treated as a year between 1900 and 1999.

Returns

The millisecond representation of the adjusted date. Prior to ECMAScript standardization, this method returns nothing.

Description

`setYear()` sets the year field of a specified Date object, with special behavior for years between 1900 and 1999.

As of ECMAScript v3, this function is no longer required in conforming JavaScript implementations; use `setFullYear()` instead.

Date.toString()

return the date portion of a Date as a string

Synopsis

date.toString()

Returns

An implementation-dependent, human-readable string representation of the date portion of *date*, expressed in the local time zone.

See Also

Date.toString()
Date.toTimeString()

Date.toGMTString()

deprecated

convert a Date to a universal time string

Synopsis

date.toGMTString()

Returns

A string representation of the date and time specified by the Date object *date*. The date is converted from the local time zone to the GMT time zone before being converted to a string.

Description

toGMTString() is deprecated in favor of the identical method Date.toUTCString().

As of ECMAScript v3, conforming implementations of JavaScript are no longer required to provide this method; use toUTCString() instead.

See Also

Date.toUTCString()

Date.toISOString()

ECMAScript 5

convert a Date to an ISO8601-formatted string

Synopsis

date.toISOString()

Returns

A string representation of *date*, formatted according to the ISO-8601 standard and expressed as a full-precision combined date and time in UTC with a timezone of "Z". The returned string has this format:

yyyy-mm-ddThh:mm:ss.sssZ

See Also

Date.parse(), Date.toString()

Date.toJSON

JSON-serialize a Date object

Synopsis

```
date.toJSON(key)
```

Arguments

key

JSON.stringify() passes this argument, but the toJSON() method ignores it.

Returns

A string representation of the date, obtained by calling its toISOString() method.

Description

This method is used by JSON.stringify() to convert a Date object to a string. It is not intended for general use.

See Also

Date.toISOString(), JSON.stringify()

Date.toLocaleDateString()

return the date portion of a Date as a locally formatted string

Synopsis

```
date.toLocaleDateString()
```

Returns

An implementation-dependent, human-readable string representation of the date portion of *date*, expressed in the local time zone and formatted according to local conventions.

See Also

Date.toDateString(), Date.toLocaleString(), Date.toLocaleTimeString(), Date.toString(), Date.toTimeString()

Date.toLocaleString()

convert a Date to a locally formatted string

Synopsis

```
date.toLocaleString()
```

Returns

A string representation of the date and time specified by *date*. The date and time are represented in the local time zone and formatted using locally appropriate conventions.

Usage

`toLocaleString()` converts a date to a string, using the local time zone. This method also uses local conventions for date and time formatting, so the format may vary from platform to platform and from country to country. `toLocaleString()` returns a string formatted in what is likely the user's preferred date and time format.

See Also

`Date.toISOString()`, `Date.toLocaleDateString()`, `Date.toLocaleTimeString()`, `Date.toString()`, `Date.toUTCString()`

Date.toLocaleTimeString()

return the time portion of a Date as a locally formatted string

Synopsis

```
date.toLocaleTimeString()
```

Returns

An implementation-dependent, human-readable string representation of the time portion of *date*, expressed in the local time zone and formatted according to local conventions.

See Also

`Date.toDateString()`, `Date.toLocaleDateString()`, `Date.toLocaleString()`, `Date.toString()`, `Date.toTimeString()`

Date.toString()

convert a Date to a string

Overrides `Object.toString()`

Synopsis

```
date.toString()
```

Returns

A human-readable string representation of *date*, expressed in the local time zone.

Description

`toString()` returns a human-readable, implementation-dependent string representation of *date*. Unlike `toUTCString()`, `toString()` expresses the date in the local time zone. Unlike `toLocaleString()`, `toString()` may not represent the date and time using locale-specific formatting.

`Date.toString()`

See Also

`Date.parse()`
`Date.toDateString()`
`Date.toISOString()`
`Date.toLocaleString()`
`Date.toString()`
`Date.toUTCString()`

`Date.toString()`

return the time portion of a `Date` as a string

Synopsis

date.toString()

Returns

A implementation-dependent, human-readable string representation of the time portion of *date*, expressed in the local time zone.

See Also

`Date.toString()`, `Date.toDateString()`, `Date.toLocaleTimeString()`

`Date.toUTCString()`

convert a `Date` to a string (universal time)

Synopsis

date.toUTCString()

Returns

A human-readable string representation, expressed in universal time, of *date*.

Description

`toUTCString()` returns an implementation-dependent string that represents *date* in universal time.

See Also

`Date.toISOString()`, `Date.toLocaleString()`, `Date.toString()`

`Date.UTC()`

convert a `Date` specification to milliseconds

Synopsis

`Date.UTC(year, month, day, hours, minutes, seconds, ms)`

Arguments

year

The year in four-digit format. If this argument is between 0 and 99, inclusive, 1900 is added to it and it is treated as a year between 1900 and 1999.

month

The month, specified as an integer from 0 (January) to 11 (December).

day

The day of the month, specified as an integer from 1 to 31. Note that this argument uses 1 as its lowest value, while other arguments use 0 as their lowest value. This argument is optional.

hours

The hour, specified as an integer from 0 (midnight) to 23 (11 p.m.). This argument is optional.

minutes

The minutes in the hour, specified as an integer from 0 to 59. This argument is optional.

seconds

The seconds in the minute, specified as an integer from 0 to 59. This argument is optional.

ms

The number of milliseconds, specified as an integer from 0 to 999. This argument is optional and is ignored prior to ECMAScript standardization.

Returns

The millisecond representation of the specified universal time. That is, this method returns the number of milliseconds between midnight GMT on January 1, 1970 and the specified time.

Description

`Date.UTC()` is a static method; it is invoked through the `Date()` constructor, not through an individual `Date` object.

The arguments to `Date.UTC()` specify a date and time and are understood to be in UTC; they are in the GMT time zone. The specified UTC time is converted to the millisecond format, which can be used by the `Date()` constructor method and by the `Date.setTime()` method.

The `Date()` constructor method can accept date and time arguments identical to those that `Date.UTC()` accepts. The difference is that the `Date()` constructor assumes local time, while `Date.UTC()` assumes universal time (GMT). To create a `Date` object using a UTC time specification, you can use code like this:

```
d = new Date(Date.UTC(1996, 4, 8, 16, 30));
```

See Also

`Date`, `Date.parse()`, `Date.setTime()`

Date.valueOf()

convert a Date to millisecond representation

Overrides Object.valueOf()

Synopsis

```
date.valueOf()
```

Returns

The millisecond representation of *date*. The value returned is the same as that returned by `Date.getTime()`.

decodeURI()

unescape characters in a URI

Synopsis

```
decodeURI(uri)
```

Arguments

uri

A string that contains an encoded URI or other text to be decoded.

Returns

A copy of *uri*, with any hexadecimal escape sequences replaced with the characters they represent.

Throws

`URIError`

Indicates that one or more of the escape sequences in *uri* is malformed and cannot be correctly decoded.

Description

`decodeURI()` is a global function that returns a decoded copy of its *uri* argument. It reverses the encoding performed by `encodeURIComponent()`; see that function's reference page for details.

See Also

`decodeURIComponent()`, `encodeURIComponent()`, `escape()`, `unescape()`

decodeURIComponent()

unescape characters in a URI component

Synopsis

```
decodeURIComponent(s)
```

Arguments*s*

A string that contains an encoded URI component or other text to be decoded.

Returns

A copy of *s*, with any hexadecimal escape sequences replaced with the characters they represent.

Throws**URIError**

Indicates that one or more of the escape sequences in *s* is malformed and cannot be correctly decoded.

Description

`decodeURIComponent()` is a global function that returns a decoded copy of its *s* argument. It reverses the encoding performed by `encodeURIComponent()`. See that function's reference page for details.

See Also

`decodeURI()`, `encodeURIComponent()`, `escape()`, `unescape()`

encodeURIComponent()

escape characters in a URI

Synopsis

```
encodeURIComponent(uri)
```

Arguments*uri*

A string that contains the URI or other text to be encoded.

Returns

A copy of *uri*, with certain characters replaced by hexadecimal escape sequences.

Throws**URIError**

Indicates that *uri* contains malformed Unicode surrogate pairs and cannot be encoded.

Description

`encodeURIComponent()` is a global function that returns an encoded copy of its *uri* argument. ASCII letters and digits are not encoded, nor are the following ASCII punctuation characters:

```
- _ . ! ~ * ' ( )
```

Because `encodeURIComponent()` is intended to encode complete URIs, the following ASCII punctuation characters, which have special meaning in URIs, are not escaped either:

`; / ? : @ & = + $ , #`

Any other characters in *uri* are replaced by converting each character to its UTF-8 encoding and then encoding each of the resulting one, two, or three bytes with a hexadecimal escape sequence of the form `%xx`. In this encoding scheme, ASCII characters are replaced with a single `%xx` escape, characters with encodings between `\u0080` and `\u07ff` are replaced with two escape sequences, and all other 16-bit Unicode characters are replaced with three escape sequences.

If you use this method to encode a URI, you should be certain that none of the components of the URI (such as the query string) contain URI separator characters such as `?` and `#`. If the components have to contain these characters, you should encode each component separately with `encodeURIComponent()`.

Use `decodeURI()` to reverse the encoding applied by this method. Prior to ECMAScript v3, you can use `escape()` and `unescape()` methods (which are now deprecated) to perform a similar kind of encoding and decoding.

Example

```
// Returns http://www.isp.com/app.cgi?arg1=1&arg2=hello%20world
encodeURIComponent("http://www.isp.com/app.cgi?arg1=1&arg2=hello world");
encodeURIComponent("\u00a9"); // The copyright character encodes to %C2%A9
```

See Also

`decodeURI()`, `decodeURIComponent()`, `encodeURIComponent()`, `escape()`, `unescape()`

encodeURIComponent()

escape characters in a URI component

Synopsis

```
encodeURIComponent(s)
```

Arguments

s

A string that contains a portion of a URI or other text to be encoded.

Returns

A copy of *s*, with certain characters replaced by hexadecimal escape sequences.

Throws

URIError

Indicates that *s* contains malformed Unicode surrogate pairs and cannot be encoded.

Description

`encodeURIComponent()` is a global function that returns an encoded copy of its *s* argument. ASCII letters and digits are not encoded, nor are the following ASCII punctuation characters:

`- _ . ! ~ * ' ( )`

All other characters, including punctuation characters such as `/`, `:`, and `#` that serve to separate the various components of a URI, are replaced with one or more hexadecimal escape sequences. See `encodeURIComponent()` for a description of the encoding scheme used.

Note the difference between `encodeURIComponent()` and `encodeURIComponent()`; `encodeURIComponent()` assumes that its argument is a portion (such as the protocol, hostname, path, or query string) of a URI. Therefore it escapes the punctuation characters that are used to separate the portions of a URI.

Example

```
encodeURIComponent("hello world?"); // Returns hello%20world%3F
```

See Also

`decodeURI()`, `decodeURIComponent()`, `encodeURIComponent()`, `escape()`, `unescape()`

Error

a generic exception

Object → Error

Constructor

```
new Error()
new Error(message)
```

Arguments

message

An optional error message that provides details about the exception.

Returns

A newly constructed Error object. If the *message* argument is specified, the Error object uses it as the value of its *message* property; otherwise, it uses an implementation-defined default string as the value of that property. When the `Error()` constructor is called as a function, without the `new` operator, it behaves just as it does when called with the `new` operator.

Properties

message

An error message that provides details about the exception. This property holds the string passed to the constructor or an implementation-defined default string.

name

A string that specifies the type of the exception. For instances of the Error class and all of its subclasses, this property specifies the name of the constructor used to create the instance.

Methods

toString()

Returns an implementation-defined string that represents this Error object.

Description

Instances of the Error class represent errors or exceptions and are typically used with the `throw` and `try/catch` statements. The `name` property specifies the type of the exception, and the `message` property can provide human-readable details about the exception.

The JavaScript interpreter never throws Error objects directly; instead, it throws instances of one of the Error subclasses, such as `SyntaxError` or `RangeError`. In your own code, you may find it convenient to throw Error objects to signal exceptions, or you may prefer to simply throw an error message or error code as a primitive string or number value.

Note that the ECMAScript specification defines a `toString()` method for the Error class (it is inherited by each of the subclasses of Error) but that it does not require this `toString()` method to return a string that contains the contents of the `message` property. Therefore, you should not expect the `toString()` method to convert an Error object to a meaningful, human-readable string. To display an error message to a user, you should explicitly use the `name` and `message` properties of the Error object.

Examples

You might signal an exception with code like the following:

```
function factorial(x) {
    if (x < 0) throw new Error("factorial: x must be >= 0");
    if (x <= 1) return 1; else return x * factorial(x-1);
}
```

And if you catch an exception, you might display its to the user with code like the following (which uses the client-side `Window.alert()` method):

```
try { /*(&*/ an error is thrown here */ }
catch(e) {
    if (e instanceof Error) { // Is it an instance of Error or a subclass?
        alert(e.name + ": " + e.message);
    }
}
```

See Also

`EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, `URIError`

Error.message

a human-readable error message

Synopsis

```
error.message
```

Description

The `message` property of an Error object (or of an instance of any subclass of Error) is intended to contain a human-readable string that provides details about the error or exception that occurred. If a `message` argument is passed to the `Error()` constructor, this message becomes the value of the `message` property. If no `message` argument is passed, an Error object inherits an implementation-defined default value (which may be the empty string) for this property.

Error.name

the type of an error

Synopsis

```
error.name
```

Description

The `name` property of an Error object (or of an instance of any subclass of Error) specifies the type of error or exception that occurred. All Error objects inherit this property from their constructor. The value of the property is the same as the name of the constructor. Thus `SyntaxError` objects have a `name` property of `"SyntaxError"`, and `EvalError` objects have a `name` of `"EvalError"`.

Error.toString()

convert an Error object to a string

Overrides `Object.toString()`

Synopsis

```
error.toString()
```

Returns

An implementation-defined string. The ECMAScript standard does not specify anything about the return value of this method, except that it is a string. Notably, it does not require the returned string to contain the error name or the error message.

escape()

deprecated

encode a string

Synopsis

```
escape(s)
```

Arguments

`s`

The string that is to be "escaped" or encoded.

`eval()`

Returns

An encoded copy of *s* in which certain characters have been replaced by hexadecimal escape sequences.

Description

`escape()` is a global function. It returns a new string that contains an encoded version of *s*. The string *s* itself is not modified.

`escape()` returns a string in which all characters of *s* other than ASCII letters, digits, and the punctuation characters `@`, `*`, `_`, `+`, `~`, `.`, and `/` have been replaced by escape sequences of the form `%xx` or `%u xxxx` (where *x* represents a hexadecimal digit). Unicode characters `\u0000` to `\u00ff` are replaced with the `%xx` escape sequence, and all other Unicode characters are replaced with the `%u xxxx` sequence.

Use the `unescape()` function to decode a string encoded with `escape()`.

Although the `escape()` function was standardized in the first version of ECMAScript, it was deprecated and removed from the standard by ECMAScript v3. Implementations of ECMAScript are likely to implement this function, but they are not required to. You should use `encodeURIComponent()` and `encodeURIComponent()` instead of `escape()`.

Example

```
escape("Hello World!"); // Returns "Hello%20World%21"
```

See Also

`encodeURIComponent()`, `encodeURIComponent()`

`eval()`

execute JavaScript code from a string

Synopsis

```
eval(code)
```

Arguments

code

A string that contains the JavaScript expression to be evaluated or the statements to be executed.

Returns

The value of the evaluated *code*, if any.

Throws

`eval()` throws a `SyntaxError` if *code* is not legal JavaScript code. If the evaluation of *code* raises an error, `eval()` propagates that error.

Description

`eval()` is a global method that evaluates a string of JavaScript code. If `code` contains an expression, `eval` evaluates the expression and returns its value. (Some expressions, such as object and function literals look like statements and must be enclosed in parentheses when passed to `eval()` in order to resolve the ambiguity.) If `code` contains a JavaScript statement or statements, `eval()` executes those statements and returns the value, if any, returned by the last statement. If `code` does not return any value, `eval()` returns `undefined`. Finally, if `code` throws an exception, `eval()` passes that exception on to the caller.

`eval()` behaves different in ECMAScript 3 and ECMAScript 5, and in ECMAScript 5, it behaves differently in strict mode and non-strict mode, and a minor digression is necessary in order to explain these differences. It is much easier to implement efficient interpreters when a programming language defines `eval` as an operator instead of as a function. JavaScript's `eval` is a function, but for the sake of efficiency, the language draws a distinction between *direct*, operator-like calls to `eval()` and *indirect* calls. A direct call uses the identifier `eval` directly and, if you removed the parentheses, it would look like `eval` was an operator. Any other invocation of `eval()` is an indirect call. If you assign the `eval()` function to a variable with a different name and invoke it through that variable, it is an indirect call. Similarly, if you invoke `eval()` as a method of the global object, it is an indirect call.

With that distinction between direct and indirect calls made, we can document the behavior of `eval()` like this:

Direct call, ES3 and ES5 non-strict mode

`eval()` evaluates `code` in the current lexical scope. If `code` contains variable or function declarations they are defined in the local scope. This is the normal use-case for `eval()`.

Indirect call, ES3

The ECMAScript 3 specification allows interpreters to throw an `EvalError` for any indirect call to `eval()`. Implementations of ES3 don't generally do this in practice, but indirect calls should be avoided.

Indirect call, ES5

In ECMAScript 5, indirect calls to `eval()` must not throw an `EvalError`, and instead must evaluate `code` in the global scope, ignoring any local variables in the current lexical scope. In ES5, we can assign `var geval = eval;`, then we can use `geval()` to evaluate `code` in the global scope.

Direct or Indirect call, strict mode

In strict mode variable and function definitions in `code` are defined in a private scope that lasts only for the duration of the `eval()` call. This means that direct calls to `eval()` in strict mode cannot alter the lexical scope and indirect calls in strict mode cannot alter the global scope. These rules apply if the invocation of `eval()` is in strict mode, or if `code` begins with a "use strict" directive.

`eval()` provides a very powerful capability to the JavaScript language, but its use is infrequent in real-world programs. Obvious uses are to write programs that act as recursive JavaScript interpreters and to write programs that dynamically generate and evaluate JavaScript code.

Most JavaScript functions that expect string arguments convert whatever value they are passed to a string before proceeding. `eval()` does not behave like this: if `code` is not a primitive string

value, it is simply returned unchanged. Be careful, therefore, that you do not inadvertently pass a String object to `eval()` when you intended to pass a primitive string value.

Example

```
eval("1+2");           // Returns 3
// This code uses client-side JavaScript methods to prompt the user to
// enter an expression and to display the results of evaluating it.
// See the client-side methods Window.alert() and Window.prompt() for details.
try {
    alert("Result: " + eval(prompt("Enter an expression:", "")));
}
catch(exception) {
    alert(exception);
}
```

EvalError

thrown when `eval()` is used improperly

Object → Error → EvalError

Constructor

```
new EvalError()
new EvalError(message)
```

Arguments

message

An optional error message that provides details about the exception. If specified, this argument is used as the value for the `message` property of the EvalError object.

Returns

A newly constructed EvalError object. If the *message* argument is specified, the Error object uses it as the value of its `message` property; otherwise, it uses an implementation-defined default string as the value of that property. When the `EvalError()` constructor is called as a function without the `new` operator, it behaves just as it does when called with the `new` operator.

Properties

message

An error message that provides details about the exception. This property holds the string passed to the constructor or an implementation-defined default string. See `Error.message` for details.

name

A string that specifies the type of the exception. All EvalError objects inherit the value "EvalError" for this property.

Description

An instance of the EvalError class may be thrown when the global function `eval()` is invoked under any other name. See `eval()` for an explanation of the restrictions on how this function may be invoked. See `Error` for details about throwing and catching exceptions.

See Also

Error, Error.message, Error.name

Function

a JavaScript function

Object → Function

Synopsis

```

functionfunctionname(argument_name_list) // Function definition statement
{
    body
}
function (argument_name_list) {body}      // Unnamed function literal
functionname(argument_value_list)        // Function invocation

```

Constructor

`new Function(argument_names..., body)`

Arguments

argument_names...

Any number of string arguments, each naming one or more arguments of the Function object being created.

body

A string that specifies the body of the function. It may contain any number of JavaScript statements, separated with semicolons, and may refer to any of the argument names specified by previous arguments to the constructor.

Returns

A newly created Function object. Invoking the function executes the JavaScript code specified by *body*.

Throws

SyntaxError

Indicates that there was a JavaScript syntax error in the *body* argument or in one of the *argument_names* arguments.

Properties

`arguments[]`

An array of arguments that were passed to the function. Deprecated.

`caller`

A reference to the Function object that invoked this one, or `null` if the function was invoked from top-level code. Deprecated.

`length`

The number of named arguments specified when the function was declared.

prototype

An object which, for a constructor function, defines properties and methods shared by all objects created with that constructor function.

Methods

apply()

Invokes a function as a method of a specified object, passing a specified array of arguments.

bind()

Return a new function that invokes this one as a method of the specified object with the optionally specified arguments.

call()

Invokes a function as a method of a specified object, passing the specified arguments.

toString()

Returns a string representation of the function.

Description

A function is a fundamental datatype in JavaScript. Chapter 8 explains how to define and use functions, and Chapter 9 covers the related topics of methods, constructors, and the `prototype` property of functions. See those chapters for complete details. Note that although function objects may be created with the `Function()` constructor described here, this is not efficient, and the preferred way to define functions, in most cases, is with a function definition statement or a function literal.

In JavaScript 1.1 and later, the body of a function is automatically given a local variable, named `arguments`, that refers to an Arguments object. This object is an array of the values passed as arguments to the function. Don't confuse this with the deprecated `arguments[]` property listed earlier. See the `Arguments` reference page for details.

See Also

`Arguments`; Chapter 8, Chapter 9

Function.apply()

invoke a function as a method of an object

Synopsis

```
function.apply(thisobj, args)
```

Arguments

thisobj

The object to which *function* is to be applied. In the body of the function, *thisobj* becomes the value of the `this` keyword. If this argument is `null`, the global object is used.

args

An array of values to be passed as arguments to *function*.

Returns

Whatever value is returned by the invocation of *function*.

Throws**TypeError**

If this method is invoked on an object that is not a function or if this method is invoked with an *args* argument that is not an array or an Arguments object.

Description

`apply()` invokes the specified *function* as if it were a method of *thisobj*, passing it the arguments contained in the *args* array. It returns the value returned by the function invocation. Within the body of the function, the `this` keyword refers to the *thisobj* object.

The *args* argument must be an array or an Arguments object. Use `Function.call()` instead if you want to specify the arguments to pass to the function individually instead of as array elements.

Example

```
// Apply the default Object.toString() method to an object that
// overrides it with its own version of the method. Note no arguments.
Object.prototype.toString.apply(o);
// Invoke the Math.max() method with apply to find the largest
// element in an array. Note that first argument doesn't matter
// in this case.
var data = [1,2,3,4,5,6,7,8];
Math.max.apply(null, data);
```

See Also

`Function.call()`

Function.arguments[]

deprecated

arguments passed to a function

Synopsis

```
function.arguments[i]
function.arguments.length
```

Description

The `arguments` property of a Function object is an array of the arguments that are passed to a function. It is defined only while the function is executing. `arguments.length` specifies the number of elements in the array.

This property is deprecated in favor of the Arguments object; it should never be used in new JavaScript code.

See Also

Arguments

Function.bind()

ECMAScript 5

return a function that invokes this as a method

Synopsis

```
function.bind(o)
function.bind(o, args...)
```

Arguments

o

The object to which this function should be bound.

args...

Zero or more argument values that will also be bound.

Returns

A new function which invokes this function as a method of *o* and passes it the arguments *args*.

Description

The `bind()` method returns a new function which invokes this function as a method of the object *o*. The arguments passed to this function consist of the *args* passed to `bind()` followed by whatever values are passed to the new function.

Example

Suppose that *f* is a function and we call the `bind()` method like this:

```
var g = f.bind(o, 1, 2);
```

Now *g* is a new function and the invocation `g(3)` is equivalent to:

```
f.call(o, 1, 2, 3);
```

See Also

`Function.apply()`, `Function.call()`, §8.7.4

Function.call()

invoke a function as a method of an object

Synopsis

```
function.call(thisobj, args...)
```

Arguments**thisobj**

The object on which *function* is to be invoked. In the body of the function, *thisobj* becomes the value of the *this* keyword. If this argument is *null*, the global object is used.

args...

Any number of arguments, which will be passed as arguments to *function*.

Returns

Whatever value is returned by the invocation of *function*.

Throws**TypeError**

If this method is invoked on an object that is not a function.

Description

call() invokes the specified *function* as if it were a method of *thisobj*, passing it any arguments that follow *thisobj* in the argument list. The return value of *call()* is the value returned by the function invocation. Within the body of the function, the *this* keyword refers to the *thisobj* object, or to the global object if *thisobj* is *null*.

Use *Function.apply()* instead if you want to specify the arguments to pass to the function in an array.

Example

```
// Call the default Object.toString() method on an object that
// overrides it with its own version of the method. Note no arguments.
Object.prototype.toString.call(o);
```

See Also

Function.apply()

Function.caller

deprecated; not defined in strict mode

the function that called this one

Synopsis

function.caller

Description

In early versions of JavaScript, the *caller* property of a *Function* object is a reference to the function that invoked the current one. If the function is invoked from the top level of a JavaScript program, *caller* is *null*. This property may be used only from within the function (i.e., the *caller* property is only defined for a function while that function is executing).

Function.caller is not part of the ECMAScript standard and is not required in conforming implementations. It should not be used.

Function.length

the number of declared arguments

Synopsis

function.length

Description

The **length** property of a function specifies the number of named arguments declared when the function was defined. The function may actually be invoked with more than or fewer than this number of arguments. Don't confuse this property of a Function object with the **length** property of the Arguments object, which specifies the number of arguments actually passed to the function. See **Arguments.length** for an example.

See Also

Arguments.length

Function.prototype

the prototype for a class of objects

Synopsis

function.prototype

Description

The **prototype** property is used when a function is used as a constructor. It refers to an object that serves as the prototype for an entire class of objects. Any object created by the constructor inherits all properties of the object referred to by the **prototype** property.

See Chapter 9 for a full discussion of constructor functions, the **prototype** property, and the definition of classes in JavaScript.

See Also

Chapter 9

Function.toString()

convert a function to a string

Synopsis

function.toString()

Returns

A string that represents the function.

Throws**TypeError**

If this method is invoked on an object that is not a Function.

Description

The `toString()` method of the Function object converts a function to a string in an implementation-dependent way. In most implementations, such as the implementations in Firefox and IE, this method returns a string of valid JavaScript code—code that includes the `function` keyword, argument list, the complete body of the function, and so on. In these implementations, the output of this `toString()` method is valid input for the global `eval()` function. This behavior is not required by the specification, however, and should not be relied upon.

Global

the global object

Object → Global

Synopsis

`this`

Global Properties

The global object is not a class, so the following global properties have individual reference entries under their own names. That is, you can find details on the `undefined` property listed under the name `undefined`, not under `Global.undefined`. Note that all top-level variables are also properties of the global object:

Infinity

A numeric value that represents positive infinity.

NaN

The not-a-number value.

undefined

The undefined value.

Global Functions

The global object is an object, not a class. The global functions listed here are not methods of any object, and their reference entries appear under the function name. For example, you'll find details on the `parseInt()` function under `parseInt()`, not `Global.parseInt()`.

decodeURI()

Decodes a string escaped with `encodeURIComponent()`.

decodeURIComponent()

Decodes a string escaped with `encodeURIComponent()`.

encodeURIComponent

Encodes a URI by escaping certain characters.

Global

`encodeURIComponent`

Encodes a URI component by escaping certain characters.

`escape()`

Encodes a string by replacing certain characters with escape sequences.

`eval()`

Evaluates a string of JavaScript code and returns the result.

`isFinite()`

Tests whether a value is a finite number.

`isNaN()`

Tests whether a value is the not-a-number value.

`parseFloat()`

Parses a number from a string.

`parseInt()`

Parses an integer from a string.

`unescape()`

Decodes a string encoded with `escape()`.

Global Objects

In addition to the global properties and functions listed earlier, the global object also defines properties that refer to all the other predefined JavaScript objects. Most of these properties are constructor functions:

`Array`

The `Array()` constructor.

`Boolean`

The `Boolean()` constructor.

`Date`

The `Date()` constructor.

`Error`

The `Error()` constructor.

`EvalError`

The `EvalError()` constructor.

`Function`

The `Function()` constructor.

`JSON`

A reference to an object that defines JSON parsing and serialization functions.

`Math`

A reference to an object that defines mathematical functions.

Number

The `Number()` constructor.

Object

The `Object()` constructor.

RangeError

The `RangeError()` constructor.

ReferenceError

The `ReferenceError()` constructor.

RegExp

The `RegExp()` constructor.

String

The `String()` constructor.

SyntaxError

The `SyntaxError()` constructor.

TypeError

The `TypeError()` constructor.

URIError

The `URIError()` constructor.

Description

The global object is a predefined object that serves as a placeholder for the global properties and functions of JavaScript. All other predefined objects, functions, and properties are accessible through the global object. The global object is not a property of any other object, so it does not have a name. (The title of this reference page was chosen simply for organizational convenience and does not indicate that the global object is named "Global"). In top-level JavaScript code, you can refer to the global object with the keyword `this`. It is rarely necessary to refer to the global object in this way, however, because the global object serves as the top of the scope chain, which means that unqualified variable and function names are looked up as properties of the object. When JavaScript code refers to the `parseInt()` function, for example, it is referring to the `parseInt` property of the global object. The fact that the global object is the top of the scope chain also means that all variables declared in top-level JavaScript code become properties of the global object.

The global object is simply an object, not a class. There is no `Global()` constructor, and there is no way to instantiate a new global object.

When JavaScript is embedded in a particular environment, the global object is usually given additional properties that are specific to that environment. In fact, the type of the global object is not specified by the ECMAScript standard, and an implementation or embedding of JavaScript may use an object of any type as the global object, as long as the object defines the basic properties and functions listed here. In client-side JavaScript, for example, the global object is a `Window` object and represents the web browser window within which the JavaScript code is running.

Example

In core JavaScript, none of the predefined properties of the global object are enumerable, so you can list all implicitly and explicitly declared global variables with a `for/in` loop like this:

```
var variables = ""
for(var name in this)
    variables += name + "\n";
```

See Also

Window in Part IV ; Chapter 3

Infinity

a numeric property that represents infinity

Synopsis

Infinity

Description

Infinity is a global property that contains the special numeric value representing positive infinity. The Infinity property is not enumerated by `for/in` loops and cannot be deleted with the `delete` operator. Note that Infinity is not a constant and can be set to any other value, something that you should take care not to do. (`Number.POSITIVE_INFINITY` is a constant, however.)

See Also

`isFinite()`, `NaN`, `Number.POSITIVE_INFINITY`

isFinite()

determine whether a number is finite

Synopsis

`isFinite(n)`

Arguments

n

The number to be tested.

Returns

true if *n* is (or can be converted to) a finite number, or false if *n* is NaN (not a number) or positive or negative infinity.

See Also

Infinity, `isNaN()`, `NaN`, `Number.NaN`, `Number.NEGATIVE_INFINITY`, `Number.POSITIVE_INFINITY`

isNaN()

check for not-a-number

Synopsis

```
isNaN(x)
```

Arguments

x

The value to be tested.

Returns

true if *x* is not a number or if it is the special numeric value NaN. It returns false if *x* is any other number.

Description

"NaN" is an acronym for "not-a-number". The global variable NaN holds a special numeric value (also known as NaN) that represents an illegal number (such as the result of zero divided by zero). isNaN() tests whether its argument is not a number. This function returns false if *x* is, or can be converted to, a number other than NaN. It returns true if *x* is not and cannot be converted to a number, or if it is equal to NaN.

NaN has the special property that it is not equal to any value including itself. So if you want to test specifically for the NaN value, rather than generically for any non-number, do not write *x* === NaN: that will always be false. Instead use the expression *x* !== *x*: this will evaluate to true only if *x* is NaN.

A common use of isNaN() is to test the results of parseFloat() and parseInt() to determine if they represent legal numbers.

Example

```

isNaN(0);           // => false
isNaN(0/0);         // => true
isNaN(parseInt("3")); // => false
isNaN(parseInt("hello")); // => true
isNaN("3");         // => false
isNaN("hello");     // => true
isNaN(true);        // => false
isNaN(undefined);   // => true

```

See Also

isFinite(), NaN, Number.NaN, parseFloat(), parseInt()

JSON

ECMAScript 5

JSON parsing and stringification

Description

JSON is a simple object that serves as the namespace for the global ECMAScript 5 functions `JSON.parse()` and `JSON.stringify()`. JSON is not a constructor. Prior to ECMAScript 5, compatible JSON parsing and serialization functions are available from <http://json.org/json2.js>.

“JSON” stands for JavaScript Object Notation. JSON is a data serialization format based on JavaScript literals, and can represent the `null` value, the boolean values `true` and `false`, floating-point numbers (using JavaScript numeric literals), strings (using JavaScript string literals), arrays of values (using JavaScript array literal syntax) and string to value mappings (using JavaScript object literal syntax). The primitive value `undefined` as well as the numbers `NaN` and `Infinity` are not representable in JSON. JavaScript functions, Dates, RegExps and Errors are not supported either.

Example

```
// Make a deep copy of any object or array that can be JSON-serialized
function deepcopy(o) { return JSON.parse(JSON.stringify(o)); }
```

See Also

`JSON.parse()`, `JSON.stringify()`, §6.9, <http://json.org>

JSON.parse()

ECMAScript 5

parse a JSON-formatted string

Synopsis

```
JSON.parse(s)
JSON.parse(s, reviver)
```

Arguments

s

The string to be parsed

reviver

An optional function that can transform parsed values.

Returns

An object, array, or primitive value parsed from *s* (and optionally modified by *reviver*).

Description

`JSON.parse()` is a global function for parsing JSON-formatted strings. Typically, you pass a single string argument and `JSON.parse()` returns the JavaScript value that the string represents.

You can use the optional *reviver* argument to filter or post-process the parsed value before it is returned. If it is specified, the *reviver* function is invoked once for each primitive value (but not the objects or arrays that contain those primitive values) parsed from *s*. *reviver* is invoked with two arguments. The first is a property name—either an object property name or an array index converted to a string. The second argument is the primitive value of that object property or array element. *reviver* is invoked as a method of the object or array that contains the primitive value. As a special case, if the string *s* represents a primitive value rather than the more typical object or array, then that primitive value will be stored in a newly-created object using a property whose name is the empty string. In this case, *reviver* will be invoked once on that newly created object, with an empty string as its first argument and the primitive value as its second.

The return value of the *reviver* function becomes the new value of the named property. If *reviver* returns its second argument, then the property will remain unchanged. If *reviver* returns *undefined* (or returns no value at all) then the named property will be deleted from the object or array before `JSON.parse()` returns to the user.

Example

Many uses of `JSON.parse()` are trivial:

```
var data = JSON.parse(text);
```

The `JSON.stringify()` function converts Date objects to strings, and you can use a *reviver* function to reverse this transformation. The example below also filters property names and returns `undefined` to remove certain properties from the result object:

```
var data JSON.parse(text, function(name, value) {
    // Remove any values whose property name begins with an underscore
    if (name[0] == '_') return undefined;
    // If the value is a string in ISO 8601 date format convert it to a Date.
    if (typeof value === "string" &&
        /\d\d\d\d\d\d-\d\d-\d\dT\d\d:\d\d:\d\d.\d\d\dZ$/.test(value))
        return new Date(value);
    // Otherwise, return the value unchanged
    return value
});
```

See Also

JSON.stringify(), §6.9

JSON.stringify()

ECMAScript 5

serialize an object, array or primitive value

Synopsis

```
JSON.stringify(o)
JSON.stringify(o, filter)
JSON.stringify(o, filter, indent)
```

Arguments

o

The object, array, or primitive value to be converted to a JSON string.

filter

An optional function that can replace values before stringification, or an array that contains the names of properties to be stringified.

indent

An optional argument that specifies an indentation string or the number of spaces to use for indentation when formatted human-readable output is desired. If omitted, the returned string contains no extraneous spaces and is machine-readable, but not easily human-readable.

Returns

A JSON-formatted string representing the value *o*, as filtered by *filter* and formatted according to *indent*.

Description

JSON.stringify() converts a primitive value, object or array to a JSON-formatted string that can later be parsed with JSON.parse(). Usually, this function is called with a single argument and returns the corresponding string.

When JSON.stringify() is called with a single argument, and when that value consists only of objects, arrays, strings, numbers, booleans and the null value the stringification is completely straightforward. When the value to be stringified contains objects that are instances of a class, however, the stringification process is more complex. If JSON.stringify() encounters any object (or array) with a method named toJSON(), it invokes that method on the object and stringifies the return value instead of the object itself. It invokes toJSON() with a single string argument which is the property name or array index of the object. The Date class defines a toJSON() method that converts Dates to strings using the Date.toISOString() method. No other built-in JavaScript class defines a toJSON() method, but you can define them for your own classes. Remember that, despite its name, toJSON() does not have to stringify the object on which it is invoked: it merely has to return a value that will be stringified in place of the original object.

The second argument to JSON.stringify() allows a second layer of filtering for the stringification process. This optional argument may be a function or an array and the two cases provide completely different filtering functionality. If you pass a function, it is a replacer function, and works something like the toJSON() method described above. If specified, the replacer function is invoked for each value to be stringified. The this value is the object or array within which the value is defined. The first argument to the replacer function is the object property name or array index of the value within that object, and the second argument is the value itself. That value is replaced by the return value of the replacer function. If the replacer returns undefined or returns nothing at all, then that value (and its array element or object property) are omitted from the stringification.

If an array of strings (or numbers—they are converted to strings) is passed instead as the second argument, these are used as the names of object properties. Any property whose name

is not in the array will be omitted from stringification. Furthermore, the returned string will include properties in the same order that they appear in the array.

`JSON.stringify()` normally returns a machine-readable string without any whitespace or newlines inserted. If you want the output to be more human readable, specify a third argument. If you specify a number between 1 and 10, `JSON.stringify()` will insert newlines and use the specified number of spaces to indent each “level” of output. If you specify a non-empty string instead, `JSON.stringify()` will insert newlines and use that string (or the first 10 characters of it) to indent each level.

Examples

```
// Basic serialization
var text = JSON.stringify(data);

// Specify exactly what fields to serialize
var text = JSON.stringify(address, ["city", "state", "country"]);

// Specify a replacer function so that RegExp objects can be serialized
var text = JSON.stringify(patterns, function(key, value) {
    if (value.constructor === RegExp) return value.toString();
    return value;
});

// Or achieve the same replacement like this:
RegExp.prototype.toJSON = function() { return this.toString(); }
```

See Also

`JSON.parse()`, §6.9

Math

mathematical functions and constants

Synopsis

```
Math.constant
Math.function()
```

Constants

`Math.E`

The constant e , the base of the natural logarithm.

`Math.LN10`

The natural logarithm of 10.

`Math.LN2`

The natural logarithm of 2.

`Math.LOG10E`

The base-10 logarithm of e .

Math

Math.LOG2E

The base-2 logarithm of e .

Math.PI

The constant π .

Math.SQRT1_2

The number 1 divided by the square root of 2.

Math.SQRT2

The square root of 2.

Static Functions

Math.abs()

Computes an absolute value.

Math.acos()

Computes an arccosine.

Math.asin()

Computes an arcsine.

Math.atan()

Computes an arctangent.

Math.atan2()

Computes the angle from the X axis to a point.

Math.ceil()

Rounds a number up.

Math.cos()

Computes a cosine.

Math.exp()

Computes a power of e .

Math.floor()

Rounds a number down.

Math.log()

Computes a natural logarithm.

Math.max()

Returns the larger of two numbers.

Math.min()

Returns the smaller of two numbers.

Math.pow()

Computes x^y

Math.random()

Computes a random number.

Math.round()
Rounds to the nearest integer.

Math.sin()
Computes a sine.

Math.sqrt()
Computes a square root.

Math.tan()
Computes a tangent.

Description

Math is an object that defines properties that refer to useful mathematical functions and constants. These functions and constants are invoked with syntax like this:

```
y = Math.sin(x);  
area = radius * radius * Math.PI;
```

Math is not a class of objects as Date and String are. There is no Math() constructor, and functions like Math.sin() are simply functions, not methods that operate on an object.

See Also

Number

Math.abs()

compute an absolute value

Synopsis

```
Math.abs(x)
```

Arguments

x
Any number.

Returns

The absolute value of x.

Math.acos()

compute an arccosine

Synopsis

```
Math.acos(x)
```

`Math.asin()`

Arguments

x

A number between -1.0 and 1.0 .

Returns

The arccosine, or inverse cosine, of the specified value *x*. This return value is between 0 and π radians.

Math.asin()

compute an arcsine

Synopsis

`Math.asin(x)`

Arguments

x

A number between -1.0 and 1.0 .

Returns

The arcsine of the specified value *x*. This return value is between $-\pi/2$ and $\pi/2$ radians.

Math.atan()

compute an arctangent

Synopsis

`Math.atan(x)`

Arguments

x

Any number.

Returns

The arc tangent of the specified value *x*. This return value is between $-\pi/2$ and $\pi/2$ radians.

Math.atan2()

compute the angle from the X axis to a point

Synopsis

`Math.atan2(y, x)`

Arguments

y

The Y coordinate of the point.

x

The X coordinate of the point.

Returns

A value between $-\pi$ and π radians that specifies the counterclockwise angle between the positive X axis and the point (*x*, *y*).

Description

The `Math.atan2()` function computes the arc tangent of the ratio *y/x*. The *y* argument can be considered the Y coordinate (or “rise”) of a point, and the *x* argument can be considered the X coordinate (or “run”) of the point. Note the unusual order of the arguments to this function: the Y coordinate is passed before the X coordinate.

Math.ceil()

round a number up

Synopsis

```
Math.ceil(x)
```

Arguments

x

Any numeric value or expression.

Returns

The closest integer greater than or equal to *x*.

Description

`Math.ceil()` computes the ceiling function—i.e., it returns the closest integer value that is greater than or equal to the function argument. `Math.ceil()` differs from `Math.round()` in that it always rounds up, rather than rounding up or down to the closest integer. Also note that `Math.ceil()` does not round negative numbers to larger negative numbers; it rounds them up toward zero.

Example

```
a = Math.ceil(1.99); // Result is 2.0
b = Math.ceil(1.01); // Result is 2.0
c = Math.ceil(1.0);  // Result is 1.0
d = Math.ceil(-1.99); // Result is -1.0
```

Math.cos()

Math.cos()

compute a cosine

Synopsis

Math.cos(*x*)

Arguments

x

An angle, measured in radians. To convert degrees to radians, multiply the degree value by 0.017453293 ($2\pi/360$).

Returns

The cosine of the specified value *x*. This return value is between -1.0 and 1.0.

Math.E

the mathematical constant *e*

Synopsis

Math.E

Description

Math.E is the mathematical constant *e*, the base of the natural logarithm, with a value of approximately 2.71828.

Math.exp()

compute e^x

Synopsis

Math.exp(*x*)

Arguments

x

A numeric value or expression to be used as the exponent.

Returns

e^x , *e* raised to the power of the specified exponent *x*, where *e* is the base of the natural logarithm, with a value of approximately 2.71828.

Math.floor()

round a number down

Synopsis

`Math.floor(x)`

Arguments

x

Any numeric value or expression.

Returns

The closest integer less than or equal to *x*.

Description

`Math.floor()` computes the floor function; in other words, it returns the nearest integer value that is less than or equal to the function argument.

`Math.floor()` rounds a floating-point value down to the closest integer. This behavior differs from that of `Math.round()`, which rounds up or down to the nearest integer. Also note that `Math.floor()` rounds negative numbers down (i.e., to be more negative), not up (i.e., closer to zero).

Example

```
a = Math.floor(1.99); // Result is 1.0
b = Math.floor(1.01); // Result is 1.0
c = Math.floor(1.0);  // Result is 1.0
d = Math.floor(-1.01); // Result is -2.0
```

Math.LN10

the mathematical constant $\log_e 10$

Synopsis

`Math.LN10`

Description

`Math.LN10` is $\log_e 10$, the natural logarithm of 10. This constant has a value of approximately 2.3025850929940459011.

Math.LN2

the mathematical constant $\log_e 2$

Synopsis

`Math.LN2`

`Math.log()`

Description

`Math.LN2` is $\log_e 2$, the natural logarithm of 2. This constant has a value of approximately 0.69314718055994528623.

Math.log()

compute a natural logarithm

Synopsis

`Math.log(x)`

Arguments

x

Any numeric value or expression greater than zero.

Returns

The natural logarithm of *x*.

Description

`Math.log()` computes $\log_e x$, the natural logarithm of its argument. The argument must be greater than zero.

You can compute the base-10 and base-2 logarithms of a number with these formulas:

$$\begin{aligned}\log_{10}^x &= \log_{10}^e \cdot \log_e^x \\ \log_2^x &= \log_2^e \cdot \log_e^x\end{aligned}$$

These formulas translate into the following JavaScript functions:

```
function log10(x) { return Math.LOG10E * Math.log(x); }  
function log2(x) { return Math.LOG2E * Math.log(x); }
```

Math.LOG10E

the mathematical constant $\log_{10}e$

Synopsis

`Math.LOG10E`

Description

`Math.LOG10E` is $\log_{10}^e$ the base-10 logarithm of the constant *e*. It has a value of approximately 0.43429448190325181667.

Math.LOG2E

the mathematical constant $\log_2e$

Synopsis

`Math.LOG2E`

Description

`Math.LOG2E` is $\log_2 e$ the base-2 logarithm of the constant e . It has a value of approximately 1.442695040888963387.

Math.max()

return the largest argument

Synopsis

```
Math.max(args...)
```

Arguments

args...

Zero or more values.

Returns

The largest of the arguments. Returns `-Infinity` if there are no arguments. Returns `NaN` if any of the arguments is `NaN` or is a nonnumeric value that cannot be converted to a number.

Math.min()

return the smallest argument

Synopsis

```
Math.min(args...)
```

Arguments

args...

Any number of arguments.

Returns

The smallest of the specified arguments. Returns `Infinity` if there are no arguments. Returns `NaN` if any argument is `NaN` or is a nonnumeric value that cannot be converted to a number.

Math.PI

the mathematical constant π

Synopsis

```
Math.PI
```

Description

`Math.PI` is the constant π or pi, the ratio of the circumference of a circle to its diameter. It has a value of approximately 3.14159265358979.

Math.pow()

compute x^y

Synopsis

`Math.pow(x, y)`

Arguments

x

The number to be raised to a power.

y

The power that *x* is to be raised to.

Returns

x to the power of *y*, x^y

Description

`Math.pow()` computes *x* to the power of *y*. Any values of *x* and *y* may be passed to `Math.pow()`. However, if the result is an imaginary or complex number, `Math.pow()` returns NaN. In practice, this means that if *x* is negative, *y* should be a positive or negative integer. Also, bear in mind that large exponents can easily cause floating-point overflow and return a value of Infinity.

Math.random()

return a pseudorandom number

Synopsis

`Math.random()`

Returns

A pseudorandom number greater than or equal to 0.0 and less than 1.0.

Math.round()

round to the nearest integer

Synopsis

`Math.round(x)`

Arguments

x

Any number.

Returns

The integer closest to *x*.

Description

`Math.round()` rounds its argument up or down to the nearest integer. It rounds .5 up. For example, it rounds 2.5 to 3 and rounds -2.5 to -2.

Math.sin()

compute a sine

Synopsis

```
Math.sin(x)
```

Arguments

`x`

An angle, in radians. To convert degrees to radians, multiply by 0.017453293 ($2\pi/360$).

Returns

The sine of `x`. This return value is between -1.0 and 1.0.

Math.sqrt()

compute a square root

Synopsis

```
Math.sqrt(x)
```

Arguments

`x`

A numeric value greater than or equal to zero.

Returns

The square root of `x`. Returns NaN if `x` is less than zero.

Description

`Math.sqrt()` computes the square root of a number. Note, however, that you can compute arbitrary roots of a number with `Math.pow()`. For example:

```
Math.cuberoot = function(x){ return Math.pow(x,1/3); }
Math.cuberoot(8); // Returns 2
```

Math.SQRT1_2

the mathematical constant $1/\sqrt{2}$

Synopsis

```
Math.SQRT1_2
```

Description

Math.SQRT1_2 is $1/\sqrt{2}$ the reciprocal of the square root of 2. This constant has a value of approximately 0.7071067811865476.

Math.SQRT2

the mathematical constant $\sqrt{2}$

Synopsis

Math.SQRT2

Description

Math.SQRT2 is the constant $\sqrt{2}$, the square root of 2. This constant has a value of approximately 1.414213562373095.

Math.tan()

compute a tangent

Synopsis

Math.tan(x)

Arguments

x

An angle, measured in radians. To convert degrees to radians, multiply the degree value by 0.017453293 ($2\pi/360$).

Returns

The tangent of the specified angle x.

NaN

the not-a-number property

Synopsis

NaN

Description

NaN is a global property that refers to the special numeric not-a-number value. The NaN property is not enumerated by `for/in` loops and cannot be deleted with the `delete` operator. Note that NaN is not a constant and can be set to any other value, something that you should take care not to do.

To determine if a value is not a number, use `isNaN()`, because NaN always compares as nonequal to any other value, including itself!

See Also

Infinity, isNaN(), Number.NaN

Number

support for numbers

Object → Number

Constructor

`new Number(value)`
`Number(value)`

Arguments

value

The numeric value of the Number object being created or a value to be converted to a number.

Returns

When `Number()` is used with the `new` operator as a constructor, it returns a newly constructed Number object. When `Number()` is invoked as a function without the `new` operator, it converts its argument to a primitive numeric value and returns that value (or NaN if the conversion failed).

Constants

`Number.MAX_VALUE`

The largest representable number.

`Number.MIN_VALUE`

The smallest representable number.

`Number.NaN`

Not-a-number value.

`Number.NEGATIVE_INFINITY`

Negative infinite value; returned on overflow.

`Number.POSITIVE_INFINITY`

Infinite value; returned on overflow.

Methods

`toString()`

Converts a number to a string using a specified radix (base).

`toLocaleString()`

Converts a number to a string using local number-formatting conventions.

`toFixed()`

Converts a number to a string that contains a specified number of digits after the decimal place.

toExponential()

Converts a number to a string using exponential notation with the specified number of digits after the decimal place.

toPrecision()

Converts a number to a string using the specified number of significant digits. Uses exponential or fixed-point notation depending on the size of the number and the number of significant digits specified.

valueOf()

Returns the primitive numeric value of a Number object.

Description

Numbers are a basic, primitive datatype in JavaScript. JavaScript also supports the Number object, which is a wrapper object around a primitive numeric value. JavaScript automatically converts between the primitive and object forms as necessary. You can explicitly create a Number object with the `Number()` constructor, although there is rarely any need to do so.

The `Number()` constructor can also be used without the `new` operator, as a conversion function. When invoked in this way, it attempts to convert its argument to a number and returns the primitive numeric value (or `NaN`) that results from the conversion.

The `Number()` constructor is also used as a placeholder for five useful numeric constants: the largest and smallest representable numbers, positive and negative infinity, and the special `NaN` value. Note that these values are properties of the `Number()` constructor function itself, not of individual number objects. For example, you can use the `MAX_VALUE` property as follows:

```
var biggest = Number.MAX_VALUE
```

but *not* like this:

```
var n = new Number(2);
var biggest = n.MAX_VALUE
```

By contrast, the `toString()` and other methods of the Number object are methods of each Number object, not of the `Number()` constructor function. As noted earlier, JavaScript automatically converts from primitive numeric values to Number objects whenever necessary. This means that you can use the Number methods with primitive numeric values as well as with Number objects.

```
var value = 1234;
var binary_value = n.toString(2);
```

See Also

Infinity, Math, NaN

Number.MAX_VALUE

the maximum numeric value

Synopsis

```
Number.MAX_VALUE
```

Description

`Number.MAX_VALUE` is the largest number representable in JavaScript. Its value is approximately $1.79E+308$.

Number.MIN_VALUE

the minimum numeric value

Synopsis

`Number.MIN_VALUE`

Description

`Number.MIN_VALUE` is the smallest (closest to zero, not most negative) number representable in JavaScript. Its value is approximately $5E-324$.

Number.NaN

the special not-a-number value

Synopsis

`Number.NaN`

Description

`Number.NaN` is a special value that indicates that the result of some mathematical operation (such as taking the square root of a negative number) is not a number. `parseInt()` and `parseFloat()` return this value when they cannot parse the specified string, and you might use `Number.NaN` in a similar way to indicate an error condition for some function that normally returns a valid number.

JavaScript prints the `Number.NaN` value as `NaN`. Note that the `NaN` value always compares as unequal to any other number, including `NaN` itself. Thus, you cannot check for the not-a-number value by comparing to `Number.NaN`; use the `isNaN()` function instead. In ECMAScript v1 and later, you can also use the predefined global property `NaN` instead of `Number.NaN`.

See Also

`isNaN()`, `NaN`

Number.NEGATIVE_INFINITY

negative infinity

Synopsis

`Number.NEGATIVE_INFINITY`

Description

`Number.NEGATIVE_INFINITY` is a special numeric value that is returned when an arithmetic operation or mathematical function generates a negative value greater than the largest representable number in JavaScript (i.e., more negative than `-Number.MAX_VALUE`).

JavaScript displays the `NEGATIVE_INFINITY` value as `-Infinity`. This value behaves mathematically like infinity; for example, anything multiplied by infinity is infinity, and anything divided by infinity is zero. In ECMAScript v1 and later, you can also use `-Infinity` instead of `Number.NEGATIVE_INFINITY`.

See Also

`Infinity`, `isFinite()`

Number.POSITIVE_INFINITY

`infinity`

Synopsis

`Number.POSITIVE_INFINITY`

Description

`Number.POSITIVE_INFINITY` is a special numeric value returned when an arithmetic operation or mathematical function overflows or generates a value greater than the largest representable number in JavaScript (i.e., greater than `Number.MAX_VALUE`). Note that when numbers “underflow,” or become less than `Number.MIN_VALUE`, JavaScript converts them to zero.

JavaScript displays the `POSITIVE_INFINITY` value as `Infinity`. This value behaves mathematically like infinity; for example, anything multiplied by infinity is infinity, and anything divided by infinity is zero. In ECMAScript v1 and later, you can also use the predefined global property `Infinity` instead of `Number.POSITIVE_INFINITY`.

See Also

`Infinity`, `isFinite()`

Number.toExponential()

format a number using exponential notation

Synopsis

`number.toExponential(digits)`

Arguments

digits

The number of digits that appears after the decimal point. This may be a value between 0 and 20, inclusive, and implementations may optionally support a larger range of values. If this argument is omitted, as many digits as necessary are used.

Returns

A string representation of *number*, in exponential notation, with one digit before the decimal place and *digits* digits after the decimal place. The fractional part of the number is rounded, or padded with zeros, as necessary, so that it has the specified length.

Throws*RangeError*

If *digits* is too small or too large. Values between 0 and 20, inclusive, will not cause a *RangeError*. Implementations are allowed to support larger and smaller values as well.

TypeError

If this method is invoked on an object that is not a *Number*.

Example

```
var n = 12345.6789;
n.toExponential(1); // Returns 1.2e+4
n.toExponential(5); // Returns 1.23457e+4
n.toExponential(10); // Returns 1.2345678900e+4
n.toExponential(); // Returns 1.23456789e+4
```

See Also

`Number.toFixed()`, `Number.toLocaleString()`, `Number.toPrecision()`, `Number.toString()`

Number.toFixed()

format a number using fixed-point notation

Synopsis

```
number.toFixed(digits)
```

Arguments*digits*

The number of digits to appear after the decimal point; this may be a value between 0 and 20, inclusive, and implementations may optionally support a larger range of values. If this argument is omitted, it is treated as 0.

Returns

A string representation of *number* that does not use exponential notation and has exactly *digits* digits after the decimal place. The number is rounded if necessary, and the fractional part is padded with zeros if necessary so that it has the specified length. If *number* is greater than $1e+21$, this method simply calls `Number.toString()` and returns a string in exponential notation.

Throws*RangeError*

If *digits* is too small or too large. Values between 0 and 20, inclusive, will not cause a *RangeError*. Implementations are allowed to support larger and smaller values as well.

`Number.toLocaleString()`

TypeError

If this method is invoked on an object that is not a `Number`.

Example

```
var n = 12345.6789;  
n.toFixed();           // Returns 12346: note rounding, no fractional part  
n.toFixed(1);          // Returns 12345.7: note rounding  
n.toFixed(6);          // Returns 12345.678900: note added zeros  
(1.23e+20).toFixed(2); // Returns 12300000000000000000.00  
(1.23e-10).toFixed(2)  // Returns 0.00
```

See Also

`Number.toExponential()`, `Number.toLocaleString()`, `Number.toPrecision()`, `Number.toString()`

Number.toLocaleString()

convert a number to a locally formatted string

Synopsis

```
number.toLocaleString()
```

Returns

An implementation-dependent string representation of the number, formatted according to local conventions, which may affect such things as the punctuation characters used for the decimal point and the thousands separator.

Throws

TypeError

If this method is invoked on an object that is not a `Number`.

See Also

`Number.toExponential()`, `Number.toFixed()`, `Number.toPrecision()`, `Number.toString()`

Number.toPrecision()

format the significant digits of a number

Synopsis

```
number.toPrecision(precision)
```

Arguments

precision

The number of significant digits to appear in the returned string. This may be a value between 1 and 21, inclusive. Implementations are allowed to optionally support larger

and smaller values of *precision*. If this argument is omitted, the `toString()` method is used instead to convert the number to a base-10 value.

Returns

A string representation of *number* that contains *precision* significant digits. If *precision* is large enough to include all the digits of the integer part of *number*, the returned string uses fixed-point notation. Otherwise, exponential notation is used with one digit before the decimal place and *precision*-1 digits after the decimal place. The number is rounded or padded with zeros as necessary.

Throws

RangeError

If *digits* is too small or too large. Values between 1 and 21, inclusive, will not cause a *RangeError*. Implementations are allowed to support larger and smaller values as well.

TypeError

If this method is invoked on an object that is not a *Number*.

Example

```
var n = 12345.6789;
n.toPrecision(1); // Returns 1e+4
n.toPrecision(3); // Returns 1.23e+4
n.toPrecision(5); // Returns 12346: note rounding
n.toPrecision(10); // Returns 12345.67890: note added zero
```

See Also

`Number.toExponential()`, `Number.toFixed()`, `Number.toLocaleString()`, `Number.toString()`

Number.toString()

convert a number to a string

Overrides `Object.toString()`

Synopsis

```
number.toString(radix)
```

Arguments

radix

An optional argument that specifies the radix, or base, between 2 and 36, in which the number should be represented. If omitted, base 10 is used. Note, however, that the ECMAScript specification allows an implementation to return any value if this argument is specified as any value other than 10.

Returns

A string representation of the number, in the specified base.

`Number.valueOf()`

Throws

TypeError

If this method is invoked on an object that is not a `Number`.

Description

The `toString()` method of the `Number` object converts a number to a string. When the *radix* argument is omitted or is specified as 10, the number is converted to a base-10 string. Although the ECMAScript specification does not require implementations to honor any other values for *radix*, all implementations in common use accept values between 2 and 36.

See Also

`Number.toExponential()`, `Number.toFixed()`, `Number.toLocaleString()`, `Number.toPrecision()`

Number.valueOf()

return the primitive number value

Overrides `Object.valueOf()`

Synopsis

number.`valueOf()`

Returns

The primitive number value of this `Number` object. It is rarely necessary to call this method explicitly.

Throws

TypeError

If this method is invoked on an object that is not a `Number`.

See Also

`Object.valueOf()`

Object

a superclass that contains features of all JavaScript objects

Constructor

`new Object()`
`new Object(value)`

Arguments

value

This optional argument specifies a primitive JavaScript value—a number, boolean, or string—that is to be converted to a `Number`, `Boolean`, or `String` object.

Returns

If no *value* argument is passed, this constructor returns a newly created Object instance. If a primitive *value* argument is specified, the constructor creates and returns a Number, Boolean, or String object wrapper for the primitive value. When the `Object()` constructor is called as a function, without the `new` operator, it behaves just as it does when used with the `new` operator.

Properties

`constructor`

A reference to the JavaScript function that was the constructor for the object.

Methods

`hasOwnProperty()`

Checks whether an object has a locally defined (noninherited) property with a specified name.

`isPrototypeOf()`

Checks whether this object is the prototype object of a specified object.

`propertyIsEnumerable()`

Checks whether a named property exists and would be enumerated by a `for/in` loop.

`toLocaleString()`

Returns a localized string representation of the object. The default implementation of this method simply calls `toString()`, but subclasses may override it to provide localization.

`toString()`

Returns a string representation of the object. The implementation of this method provided by the Object class is quite generic and does not provide much useful information. Subclasses of Object typically override this method by defining their own `toString()` method, which produces more useful output.

`valueOf()`

Returns the primitive value of the object, if any. For objects of type Object, this method simply returns the object itself. Subclasses of Object, such as Number and Boolean, override this method to return the primitive value associated with the object.

Static Methods

In ECMAScript 5, the Object constructor serves as a namespace for the following global functions:

`Object.create()`

Create a new object with specified prototype and properties.

`Object.defineProperties()`

Create or configure one or more properties of a specified object.

`Object.defineProperty()`

Create or configure a property of a specified object.

`Object.freeze()`

Make the specified object immutable.

`Object.getOwnPropertyDescriptor()`

Query the attributes of the specified property of the specified object.

`Object.getOwnPropertyNames()`

Return an array of the names of all non-inherited properties of the specified object, including non-enumerable properties.

`Object.getPrototypeOf()`

Return the prototype of the specified object.

`Object.isExtensible()`

Determine whether new properties can be added to the specified object.

`Object.isFrozen()`

Determine whether the specified object is frozen.

`Object.isSealed()`

Determine whether the specified object is sealed.

`Object.keys()`

Return an array of the names of the non-inherited enumerable properties of the specified object.

`Object.preventExtensions()`

Prevent the future addition of properties to the specified object.

`Object.seal()`

Prevent the addition of new properties and the deletion of existing properties for the specified object.

Description

The `Object` class is a built-in datatype of the JavaScript language. It serves as the superclass for all other JavaScript objects; therefore, methods and behavior of the `Object` class are inherited by all other objects. The basic behavior of objects in JavaScript is explained in Chapter 6.

In addition to the `Object()` constructor shown above, objects can also be created and initialized using the `Object` literal syntax described in §6.1.

See Also

`Array`, `Boolean`, `Function`, `Function.prototype`, `Number`, `String`; Chapter 6

`Object.constructor`

an object's constructor function

Synopsis

object.`constructor`

Description

The `constructor` property of any object is a reference to the function that was used as the constructor for that object. For example, if you create an array `a` with the `Array()` constructor, `a.constructor` is an `Array`:

```
a = new Array(1,2,3); // Create an object
a.constructor == Array // Evaluates to true
```

One common use of the `constructor` property is to determine the type of unknown objects. Given an unknown value, you can use the `typeof` operator to determine whether it is a primitive value or an object. If it is an object, you can use the `constructor` property to determine what type of object it is. For example, the following function determines whether a given value is an array:

```
function isArray(x) {
    return ((typeof x == "object") && (x.constructor == Array));
}
```

Note, however, that while this technique works for the objects built into core JavaScript, it is not guaranteed to work with host objects such as the `Window` object of client-side JavaScript. The default implementation of the `Object.toString()` method provides another way to determine the type of an unknown object.

See Also

`Object.toString()`

Object.create()

ECMAScript 5

create an object with specified prototype and properties

Synopsis

```
Object.create(proto)
Object.create(proto, descriptors)
```

Arguments

proto

The prototype of the newly-created object, or `null`.

descriptors

An optional object that maps property names to property descriptors.

Returns

A newly created object that inherits from *proto* and has the properties described by *descriptors*.

Throws

TypeError

If *proto* is not an object or `null`, or if *descriptors* is specified and causes `Object.defineProperty()` to throw a `TypeError`.

Description

`Object.create()` creates and returns a new object with *proto* as its prototype. This means that the new object inherits properties from *proto*.

If the optional *descriptors* argument is specified, `Object.create()` adds properties to the new object as if by calling `Object.defineProperty()`. That is the two-argument invocation `Object.create(p,d)` is equivalent to:

```
Object.defineProperty(Object.create(p), d);
```

See `Object.defineProperty()` for more on the *descriptors* argument, and see `Object.getOwnPropertyDescriptor()` for an explanation of property descriptor objects.

Note that this is not a method to be invoked on an object: it is a global function and you must pass an object to it.

Example

```
// Create an object that has own properties x and y and inherits property z
var p = Object.create({z:0}, {
  x: { value: 1, writable: false, enumerable:true, configurable: true},
  y: { value: 2, writable: false, enumerable:true, configurable: true},
});
```

See Also

`Object.defineProperty()`, `Object.defineProperties()`, `Object.getOwnPropertyDescriptor()`, §6.1, §6.7

Object.defineProperties()

ECMAScript 5

create or configure multiple object properties

Synopsis

```
Object.defineProperties(o, descriptors)
```

Arguments

o

The object on which properties are to be created or configured.

descriptors

An object that maps property names to property descriptors.

Returns

The object *o*.

Throws

TypeError

If *o* is not an object, or if any of the specified properties cannot be created or configured. This function is not atomic: it may create or configure certain properties and then throw

an error, before even attempting to create or configure other properties. See §6.7 for a list of property configuration errors that can cause a `TypeError`.

Description

`Object.defineProperty()` creates or configures on the object *o* the properties named and described by *descriptors*. The names of the properties in *descriptors* are the names of the properties to be created or configured on *o*, and the values of those properties are the property descriptor objects that specify the attributes of the properties to be created or configured.

`Object.defineProperty()` works much like `Object.defineProperty()` does; see that function for more details. See `Object.getOwnPropertyDescriptor()` for more on property descriptor objects.

Example

```
// Add read-only properties x and y to a newly-created object
var p = Object.defineProperty({}, {
  x: { value: 0, writable: false, enumerable:true, configurable: true},
  y: { value: 1, writable: false, enumerable:true, configurable: true},
});
```

See Also

`Object.create()`, `Object.defineProperty()`, `Object.getOwnPropertyDescriptor()`, §6.7

Object.defineProperty()

ECMAScript 5

create or configure an object property

Synopsis

`Object.defineProperty(o, name, desc)`

Arguments

o

The object on which a property is to be created or configured.

name

The name of the property to be created or configured.

desc

A property descriptor object that describes the new property or describes the changes to be made to an existing property.

Returns

The object *o*.

`Object.freeze()`

Throws

TypeError

If *o* is not an object, or if the property cannot be created (because *o* is not extensible) or configured (because it already exists and is non-configurable, for example). See §6.7 for a list of property configuration errors that can cause this function to throw a *TypeError*.

Description

`Object.defineProperty()` creates or configures the property named *name* of the object *o*, using the property descriptor *desc*. See `Object.getOwnPropertyDescriptor()` for an explanation of property descriptor objects.

If *o* does not already have a property named *name*, then this function simply creates a new property with the attributes and value specified in *desc*. If any properties are missing from *desc*, then the corresponding attributes are set to *false* or *undefined*.

If *name* is the name of an existing property of *o*, then `Object.defineProperty()` configures that property by altering its value or attributes. In this case, *desc* only needs to contain the attributes to be changed: attributes not mentioned in *desc* will not be altered.

Note that this is not a method to be invoked on an object: it is a global function and you must pass an object to it.

Example

```
function constant(o, n, v) { // Define a constant o.n with value v
    Object.defineProperty(o, n, { value: v, writable: false
                                enumerable: true, configurable:false});
}
```

See Also

`Object.create()`, `Object.defineProperties()`, `Object.getOwnPropertyDescriptor()`, §6.7

Object.freeze()

ECMAScript 5

make an object immutable

Synopsis

```
Object.freeze(o)
```

Arguments

o

The object to be frozen

Returns

The now-frozen argument object *o*.

Description

`Object.freeze()` makes *o* non-extensible (see `Object.preventExtensions()`) and makes all of its own properties non-configurable, like `Object.seal()` does. In addition, however, it also makes all non-inherited data properties read-only. This means that new properties cannot be added to *o* and that existing properties cannot be set or deleted. Freezing an object is a permanent change: once frozen and object cannot be unfrozen.

Note that `Object.freeze()` only sets the writable attribute of data properties. Properties that have a setter function defined are not affected. Also note that `Object.freeze()` does not affect inherited properties.

Note that this is not a method to be invoked on an object: it is a global function and you must pass an object to it.

See Also

`Object.defineProperty()`, `Object.isFrozen()`, `Object.preventExtensions()`, `Object.seal()`, §6.8.3

Object.getOwnPropertyDescriptor()

ECMAScript 5

query property attributes

Synopsis

```
Object.getOwnPropertyDescriptor(o, name)
```

Arguments

o

The object that is to have its property attributes queried.

name

The name of the property (or index of the array element) to query.

Returns

A property descriptor object for the specified property of the specified object, or *undefined* if no such property exists.

Description

`Object.getOwnPropertyDescriptor()` returns a property descriptor for the specified property of the specified object. A property descriptor is an object that describes the attributes and value of a property. See the sub-section below for complete details. Note that this is not a method to be invoked on an object: it is a global function and you must pass an object to it.

Property Descriptors

A property descriptor is an ordinary JavaScript object that describes the attributes (and sometimes the value) of a property. There are two kinds of JavaScript properties. A *data property* has a value and three attributes: enumerable, writable, and configurable. An *accessor property* has a getter and/or a setter method as well as enumerable and configurable attributes.

The descriptor for a data property looks like this:

```
{
  value:      /* any JavaScript value */,
  writable:   /* true or false */,
  enumerable: /* true or false */,
  configurable: /* true or false */
}
```

The descriptor for an accessor property looks like this:

```
{
  get:        /* function or undefined: replaces the property value */,
  set:        /* function or undefined: replaces the writable attribute */,
  enumerable: /* true or false */,
  configurable: /* true or false */
}
```

See Also

`Object.defineProperty()`, §6.7

Object.getOwnPropertyNames()

ECMAScript 5

return the names of non-inherited properties

Synopsis

```
Object.getOwnPropertyNames(o)
```

Arguments

o

An object

Returns

An array that contains the names of all non-inherited properties of *o*, including non-enumerable properties.

Description

`Object.getOwnPropertyNames()` returns an array that contains the names of all non-inherited properties of *o*, including non-enumerable properties. See `Object.keys()` for a function that returns only the names of enumerable properties.

Note that this is not a method to be invoked on an object: it is a global function and you must pass an object to it.

Example

```
Object.getOwnPropertyNames([]) // => ["length"]; "length" is non-enumerable
```

See Also

`Object.keys()`, §6.5

Object.getPrototypeOf()

ECMAScript 5

return the prototype of an object

Synopsis

`Object.getPrototypeOf(o)`

Arguments

o

An object.

Returns

The prototype object of *o*.

Description

`Object.getPrototypeOf()` returns the prototype of its argument. Note that this is a global function, and you must pass an object to it. It is not a method that is invoked on an object.

Example

```

var p = {};           // An ordinary object
Object.getPrototypeOf(p) // => Object.prototype
var o = Object.create(p) // An object that inherits from p
Object.getPrototypeOf(o) // => p

```

See Also

`Object.create()`; Chapter 6

Object.hasOwnProperty()

check whether a property is inherited

Synopsis

`object.hasOwnProperty(propname)`

Arguments

*propname*A string that contains the name of a property of *object*.

Returns

true if *object* has a noninherited property with the name specified by *propname*; false if *object* does not have a property with the specified name or if it inherits that property from its prototype object.

Description

As explained in Chapter 9, JavaScript objects may have properties of their own, and they may also inherit properties from their prototype object. The `hasOwnProperty()` method provides a way to distinguish between inherited properties and noninherited local properties.

Example

```
var o = new Object();           // Create an object
o.x = 3.14;                     // Define a noninherited local property
o.hasOwnProperty("x");          // Returns true: x is a local property of o
o.hasOwnProperty("y");          // Returns false: o doesn't have a property y
o.hasOwnProperty("toString");   // Returns false: toString property is inherited
```

See Also

`Function.prototype`, `Object.propertyIsEnumerable()`; Chapter 9

Object.isExtensible()

ECMAScript 5

can new properties be added to an object?

Synopsis

```
Object.isExtensible(o)
```

Arguments

o

The object to be checked for extensibility

Returns

true if the object can be extended with new properties, or false if it cannot.

Description

An object is extensible (or extendable) if it can have new properties added to it. All objects are extendable when they are created and remain that way unless they are passed to `Object.preventExtensions()`, `Object.seal()`, or `Object.freeze()`.

Note that this is not a method to be invoked on an object: it is a global function and you must pass an object to it.

Example

```
var o = {};                     // Start with a newly-created object
Object.isExtensible(o)          // => true: it is extendable
Object.preventExtensions(o);    // Make it non-extendable
Object.isExtensible(o)          // => false: now it is not extendable
```

See Also

`Object.isFrozen()`, `Object.isSealed()`, `Object.preventExtensions()`, §6.8.3

Object.isFrozen()

ECMAScript 5

is an object immutable?

Synopsis

`Object.isFrozen(o)`

Arguments

o

The object to be checked

Returns

true if *o* is frozen and immutable, or false if it is not.

Description

An object is frozen if all of its non-inherited properties (except those with setter methods) are read-only, and if it is sealed. An object is sealed if no new (non-inherited) properties can be added to it, no existing (non-inherited) properties can be deleted from it. `Object.isFrozen()` tests whether its argument is frozen or not. Once frozen, an object can never be unfrozen.

The usual way to freeze an object is by passing it to `Object.freeze()`. It is also possible to freeze an object by passing it to `Object.preventExtensions()` and then using `Object.defineProperty()` to make all of its properties read-only and nondeletable.

Note that this is not a method to be invoked on an object: it is a global function and you must pass an object to it.

See Also

`Object.defineProperty()`, `Object.freeze()`, `Object.isExtensible()`, `Object.isSealed()`, `Object.preventExtensions()`, `Object.seal()`, §6.8.3

Object.isPrototypeOf()

is one object the prototype of another?

Synopsis

`object.isPrototypeOf(o)`

Arguments

o

Any object.

Returns

true if *object* is the prototype of *o*; false if *o* is not an object or if *object* is not the prototype of *o*.

Description

As explained in Chapter 9, JavaScript objects inherit properties from their prototype object. The prototype of an object is referred to by the `prototype` property of the constructor function that creates and initializes the object. The `isPrototypeOf()` method provides a way to determine if one object is the prototype of another. This technique can be used to determine the class of an object.

Example

```
var o = new Object();           // Create an object
Object.prototype.isPrototypeOf(o) // true: o is an object
Function.prototype.isPrototypeOf(o.toString); // true: toString is a function
Array.prototype.isPrototypeOf([1,2,3]); // true: [1,2,3] is an array
// Here is a way to perform a similar test
(o.constructor == Object); // true: o was created with Object() constructor
(o.toString.constructor == Function); // true: o.toString is a function
// Prototype objects themselves have prototypes. The following call
// returns true, showing that function objects inherit properties
// from Function.prototype and also from Object.prototype.
Object.prototype.isPrototypeOf(Function.prototype);
```

See Also

`Function.prototype`, `Object.constructor`; Chapter 9

Object.isSealed()

ECMAScript 5

can properties be added to or deleted from an object?

Synopsis

```
Object.isSealed(o)
```

Arguments

o

The object to be checked

Returns

true if *o* is sealed, or false if it is not.

Description

An object is sealed if no new (non-inherited) properties can be added to it and no existing (non-inherited) properties can be deleted from it. `Object.isSealed()` tests whether its argument is sealed or not. Once sealed, an object can never be unsealed. The usual way to seal an object is by passing it to `Object.seal()` or `Object.freeze()`. It is also possible to seal an object by passing it to `Object.preventExtensions()` and then using `Object.defineProperty()` to make all of its properties non-deletable.

Note that this is not a method to be invoked on an object: it is a global function and you must pass an object to it.

See Also

Object.defineProperty(), Object.freeze(), Object.isExtensible(), Object.isFrozen(), Object.preventExtensions(), Object.seal(), §6.8.3

Object.keys()

ECMAScript 5

return own enumerable property names

Synopsis

```
Object.keys(o)
```

Arguments

o
An object

Returns

An array that contains the names of all enumerable own (non-inherited) properties of *o*.

Description

Object.keys() returns an array of property names for the object *o*. The array only includes the names of properties that are enumerable and are defined directly on *o*: inherited properties are not included. (See Object.getOwnPropertyNames() for a way to obtain the names of non-enumerable properties.) Property names appear in the returned array in the same order they would be enumerated by a for/in loop.

Note that this is not a method to be invoked on an object: it is a global function and you must pass an object to it.

Example

```
Object.keys({x:1, y:2}) // => ["x", "y"]
```

See Also

Object.getOwnPropertyNames(), §5.5.4, §6.5

Object.preventExtensions()

ECMAScript 5

don't allow new properties on an object

Synopsis

```
Object.preventExtensions(o)
```

Arguments

o
The object that is to have its extensible attribute set

Returns

The argument object *o*.

Description

`Object.preventExtensions()` sets the *extensible* attribute of *o* to false so that no new properties can be added to it. This is a permanent change: once an object has been made non-extensible, it cannot be made extensible again.

Note that `Object.preventExtensions()` does not affect the prototype chain, and a nonextensible object can still gain new inherited properties.

Note that this is not a method to be invoked on an object: it is a global function and you must pass an object to it.

See Also

`Object.freeze()`, `Object.isExtensible()`, `Object.seal()`, §6.8.3

Object.propertyIsEnumerable()

will property be seen by a for/in loop?

Synopsis

```
object.propertyIsEnumerable(propname)
```

Arguments

propname

A string that contains the name of a property of *object*.

Returns

true if *object* has a noninherited property with the name specified by *propname* and if that property is *enumerable*, which means that it would be enumerated by a *for/in* loop on *object*.

Description

The *for/in* statement loops through the enumerable properties of an object. Not all properties of an object are enumerable, however: properties added to an object by JavaScript code are enumerable, but the predefined properties (such as methods) of built-in objects are not usually enumerable. The `propertyIsEnumerable()` method provides a way to distinguish between enumerable and nonenumerable properties. Note, however, that the ECMAScript specification states that `propertyIsEnumerable()` does not examine the prototype chain, which means it works only for local properties of an object and does not provide any way to test the enumerability of inherited properties.

Example

```
var o = new Object();           // Create an object
o.x = 3.14;                     // Define a property
o.propertyIsEnumerable("x");    // true: property x is local and enumerable
o.propertyIsEnumerable("y");    // false: o doesn't have a property y
o.propertyIsEnumerable("toString"); // false: toString property is inherited
Object.prototype.propertyIsEnumerable("toString"); // false: nonenumerable
```

See Also

Function.prototype, Object.hasOwnProperty(); Chapter 6

Object.seal()

ECMAScript 5

prevent the addition or deletion of properties

Synopsis

```
Object.seal(o)
```

Arguments

o

The object to be sealed

Returns

The now-sealed argument object *o*.

Description

`Object.seal()` makes *o* non-extensible (see `Object.preventExtensions()`) and makes all of its own properties non-configurable. This has the effect of preventing the addition of new properties and preventing the deletion of existing properties. Sealing an object is permanent: once sealed, an object cannot be unsealed.

Note that `Object.seal()` does not make properties read-only; see `Object.freeze()` for that. Also note that `Object.seal()` does not affect inherited properties. If a sealed object has a non-sealed object in its prototype chain, then inherited properties may be added or removed.

Note that this is not a method to be invoked on an object: it is a global function and you must pass an object to it.

See Also

`Object.defineProperty()`, `Object.freeze()`, `Object.isSealed()`, `Object.preventExtensions()`, §6.8.3

Object.toLocaleString()

return an object's localized string representation

Synopsis

```
object.toString()
```

Returns

A string representing the object.

Description

This method is intended to return a string representation of the object, localized as appropriate for the current locale. The default `toLocaleString()` method provided by the `Object` class simply calls the `toString()` method and returns the nonlocalized string that it returns. Note, however, that other classes, including `Array`, `Date`, and `Number`, define their own versions of this method to perform localized string conversions. When defining your own classes, you may want to override this method as well.

See Also

`Array.toLocaleString()`, `Date.toLocaleString()`, `Number.toLocaleString()`, `Object.toLocaleString()`

Object.toString()

define an object's string representation

Synopsis

```
object.toString()
```

Returns

A string representing the object.

Description

The `toString()` method is not one you often call explicitly in your JavaScript programs. Instead, you define this method in your objects, and the system calls it whenever it needs to convert your object to a string.

The JavaScript system invokes the `toString()` method to convert an object to a string whenever the object is used in a string context. For example, an object is converted to a string when it is passed to a function that expects a string argument:

```
alert(my_object);
```

Similarly, objects are converted to strings when they are concatenated to strings with the `+` operator:

```
var msg = 'My object is: ' + my_object;
```

The `toString()` method is invoked without arguments and should return a string. To be useful, the string you return should be based, in some way, on the value of the object for which the method was invoked.

When you define a custom class in JavaScript, it is good practice to define a `toString()` method for the class. If you do not, the object inherits the default `toString()` method from the `Object` class. This default method returns a string of the form:

```
[objectclass]
```

where *class* is the class of the object: a value such as `"Object"`, `"String"`, `"Number"`, `"Function"`, `"Window"`, `"Document"`, and so on. This behavior of the default `toString()` method is occasionally useful to determine the type or class of an unknown object. Because most

objects have a custom version of `toString()`, however, you must explicitly invoke the `Object.toString()` method on an object `o` with code like this:

```
Object.prototype.toString.apply(o);
```

Note that this technique for identifying unknown objects works only for built-in objects. If you define your own object class, it will have a *class* of "Object". In this case, you can use the `Object.constructor` property to obtain more information about the object.

The `toString()` method can be quite useful when you are debugging JavaScript programs; it allows you to print objects and see their value. For this reason alone, it is a good idea to define a `toString()` method for every object class you create.

Although the `toString()` method is usually invoked automatically by the system, there are times when you may invoke it yourself. For example, you might want to do an explicit conversion of an object to a string in a situation where JavaScript does not do it automatically for you:

```
y = Math.sqrt(x);      // Compute a number
ystr = y.toString();    // Convert it to a string
```

Note in this example that numbers have a built-in `toString()` method you can use to force a conversion.

In other circumstances, you can choose to use a `toString()` call even in a context where JavaScript does the conversion automatically. Using `toString()` explicitly can help to make your code clearer:

```
alert(my_obj.toString());
```

See Also

`Object.constructor`, `Object.toLocaleString()`, `Object.valueOf()`

Object.valueOf()

the primitive value of the specified object

Synopsis

```
object.valueOf()
```

Returns

The primitive value associated with the *object*, if any. If there is no value associated with *object*, returns the object itself.

Description

The `valueOf()` method of an object returns the primitive value associated with that object, if there is one. For objects of type `Object`, there is no primitive value, and this method simply returns the object itself.

For objects of type `Number`, however, `valueOf()` returns the primitive numeric value represented by the object. Similarly, it returns the primitive boolean value associated with a `Boolean` object and the string associated with a `String` object.

It is rarely necessary to invoke the `valueOf()` method yourself. JavaScript does this automatically whenever an object is used where a primitive value is expected. In fact, because of this automatic invocation of the `valueOf()` method, it is difficult to even distinguish between primitive values and their corresponding objects. The `typeof` operator shows you the difference between strings and String objects for example, but in practical terms, you can use them equivalently in your JavaScript code.

The `valueOf()` methods of the Number, Boolean, and String objects convert these wrapper objects to the primitive values they represent. The `Object()` constructor performs the opposite operation when invoked with a number, boolean, or string argument: it wraps the primitive value in an appropriate object wrapper. JavaScript performs this primitive-to-object conversion for you in almost all circumstances, so it is rarely necessary to invoke the `Object()` constructor in this way.

In some circumstances, you may want to define a custom `valueOf()` method for your own objects. For example, you might define a JavaScript object type to represent complex numbers (a real number plus an imaginary number). As part of this object type, you would probably define methods for performing complex addition, multiplication, and so on (see Example 9-3). But you might also want to treat your complex numbers like ordinary real numbers by discarding the imaginary part. To achieve this, you might do something like the following:

```
Complex.prototype.valueOf = new Function("return this.real");
```

With this `valueOf()` method defined for your Complex object type, you can, for example, pass one of your complex number objects to `Math.sqrt()`, which computes the square root of the real portion of the complex number.

See Also

`Object.toString()`

parseFloat()

convert a string to a number

Synopsis

```
parseFloat(s)
```

Arguments

s

The string to be parsed and converted to a number.

Returns

The parsed number, or NaN if *s* does not begin with a valid number. In JavaScript 1.0, `parseFloat()` returns 0 instead of NaN when *s* cannot be parsed as a number.

Description

`parseFloat()` parses and returns the first number that occurs in *s*. Parsing stops, and the value is returned, when `parseFloat()` encounters a character in *s* that is not a valid part of the

number. If *s* does not begin with a number that `parseFloat()` can parse, the function returns the not-a-number value `NaN`. Test for this return value with the `isNaN()` function. If you want to parse only the integer portion of a number, use `parseInt()` instead of `parseFloat()`.

See Also

`isNaN()`, `parseInt()`

parseInt()

convert a string to an integer

Synopsis

```
parseInt(s)
parseInt(s, radix)
```

Arguments

s

The string to be parsed.

radix

An optional integer argument that represents the radix (i.e., base) of the number to be parsed. If this argument is omitted or is 0, the number is parsed in base 10—or in base 16 if it begins with 0x or 0X. If this argument is less than 2 or greater than 36, `parseInt()` returns `NaN`.

Returns

The parsed number, or `NaN` if *s* does not begin with a valid integer. In JavaScript 1.0, `parseInt()` returns 0 instead of `NaN` when it cannot parse *s*.

Description

`parseInt()` parses and returns the first number (with an optional leading minus sign) that occurs in *s*. Parsing stops, and the value is returned, when `parseInt()` encounters a character in *s* that is not a valid digit for the specified *radix*. If *s* does not begin with a number that `parseInt()` can parse, the function returns the not-a-number value `NaN`. Use the `isNaN()` function to test for this return value.

The *radix* argument specifies the base of the number to be parsed. Specifying 10 makes `parseInt()` parse a decimal number. The value 8 specifies that an octal number (using digits 0 through 7) is to be parsed. The value 16 specifies a hexadecimal value, using digits 0 through 9 and letters A through F. *radix* can be any value between 2 and 36.

If *radix* is 0 or is not specified, `parseInt()` tries to determine the radix of the number from *s*. If *s* begins (after an optional minus sign) with 0x, `parseInt()` parses the remainder of *s* as a hexadecimal number. Otherwise `parseInt()` parses it as a decimal number.

Example

```
parseInt("19", 10); // Returns 19 (10 + 9)
parseInt("11", 2);  // Returns 3 (2 + 1)
```

```

parseInt("17", 8); // Returns 15 (8 + 7)
parseInt("1f", 16); // Returns 31 (16 + 15)
parseInt("10");    // Returns 10
parseInt("0x10");  // Returns 16

```

See Also

`isNaN()`, `parseFloat()`

RangeError

thrown when a number is out of its legal range

Object → Error → RangeError

Constructor

```

new RangeError()
new RangeError(message)

```

Arguments

message

An optional error message that provides details about the exception. If specified, this argument is used as the value for the `message` property of the RangeError object.

Returns

A newly constructed RangeError object. If the *message* argument is specified, the Error object uses it as the value of its `message` property; otherwise, it uses an implementation-defined default string as the value of that property. When the `RangeError()` constructor is called as a function, without the `new` operator, it behaves just as it would when called with the `new` operator.

Properties

`message`

An error message that provides details about the exception. This property holds the string passed to the constructor or an implementation-defined default string. See `Error.message` for details.

`name`

A string that specifies the type of the exception. All RangeError objects inherit the value "RangeError" for this property.

Description

An instance of the RangeError class is thrown when a numeric value is not in its legal range. For example, setting the length of an array to a negative number causes a RangeError to be thrown. See `Error` for details about throwing and catching exceptions.

See Also

`Error`, `Error.message`, `Error.name`

ReferenceError

thrown when reading a variable that does not exist

Object → Error → ReferenceError

Constructor

```
new ReferenceError()
new ReferenceError(message)
```

Arguments

message

An optional error message that provides details about the exception. If specified, this argument is used as the value for the `message` property of the `ReferenceError` object.

Returns

A newly constructed `ReferenceError` object. If the *message* argument is specified, the `Error` object uses it as the value of its `message` property; otherwise, it uses an implementation-defined default string as the value of that property. When the `ReferenceError()` constructor is called as a function, without the `new` operator, it behaves just as it would with the `new` operator.

Properties

`message`

An error message that provides details about the exception. This property holds the string passed to the constructor or an implementation-defined default string. See `Error.message` for details.

`name`

A string that specifies the type of the exception. All `ReferenceError` objects inherit the value "ReferenceError" for this property.

Description

An instance of the `ReferenceError` class is thrown when you attempt to read the value of a variable that does not exist. See `Error` for details about throwing and catching exceptions.

See Also

`Error`, `Error.message`, `Error.name`

RegExp

regular expressions for pattern matching

Object → RegExp

Literal Syntax

```
/pattern/attributes
```

Constructor

```
new RegExp(pattern, attributes)
```

Arguments

pattern

A string that specifies the pattern of the regular expression or another regular expression.

attributes

An optional string containing any of the “g”, “i”, and “m” attributes that specify global, case-insensitive, and multiline matches, respectively. The “m” attribute is not available prior to ECMAScript standardization. If the *pattern* argument is a regular expression instead of a string, this argument must be omitted.

Returns

A new RegExp object, with the specified pattern and flags. If the *pattern* argument is a regular expression rather than a string, the `RegExp()` constructor creates a new RegExp object using the same pattern and flags as the specified RegExp. If `RegExp()` is called as a function without the `new` operator, it behaves just as it would with the `new` operator, except when *pattern* is a regular expression; in that case, it simply returns *pattern* instead of creating a new RegExp object.

Throws

SyntaxError

If *pattern* is not a legal regular expression, or if *attributes* contains characters other than “g”, “i”, and “m”.

TypeError

If *pattern* is a RegExp object, and the *attributes* argument is not omitted.

Instance Properties

global

Whether the RegExp has the “g” attribute.

ignoreCase

Whether the RegExp has the “i” attribute.

lastIndex

The character position of the last match; used for finding multiple matches in a string.

multiline

Whether the RegExp has the “m” attribute.

source

The source text of the regular expression.

Methods

`exec()`

Performs powerful, general-purpose pattern matching.

`test()`

Tests whether a string contains a pattern.

Description

The `RegExp` object represents a regular expression, a powerful tool for performing pattern matching on strings. See Chapter 10 for complete details on regular-expression syntax and use.

See Also

Chapter 10

RegExp.exec()

general-purpose pattern matching

Synopsis

```
regexp.exec(string)
```

Arguments

string

The string to be searched.

Returns

An array containing the results of the match or `null` if no match was found. The format of the returned array is described below.

Throws

`TypeError`

If this method is invoked on an object that is not a `RegExp`.

Description

`exec()` is the most powerful of all the `RegExp` and `String` pattern-matching methods. It is a general-purpose method that is somewhat more complex to use than `RegExp.test()`, `String.search()`, `String.replace()`, and `String.match()`.

`exec()` searches *string* for text that matches *regexp*. If it finds a match, it returns an array of results; otherwise, it returns `null`. Element 0 of the returned array is the matched text. Element 1 is the text that matched the first parenthesized subexpression, if any, within *regexp*. Element 2 contains the text that matched the second subexpression, and so on. The array `length` property specifies the number of elements in the array, as usual. In addition to the array elements and the `length` property, the value returned by `exec()` also has two other properties. The `index` property specifies the character position of the first character of the matched text. The `input` property refers to *string*. This returned array is the same as the array that is returned by the `String.match()` method, when invoked on a nonglobal `RegExp` object.

When `exec()` is invoked on a nonglobal pattern, it performs the search and returns the result described earlier. When *regexp* is a global regular expression, however, `exec()` behaves in a slightly more complex way. It begins searching *string* at the character position specified by the `lastIndex` property of *regexp*. When it finds a match, it sets `lastIndex` to the position of the first character after the match. This means that you can invoke `exec()` repeatedly in order to loop through all matches in a string. When `exec()` cannot find any more matches, it returns

null and resets `lastIndex` to zero. If you begin searching a new string immediately after successfully finding a match in another string, you must be careful to manually reset `lastIndex` to zero.

Note that `exec()` always includes full details of every match in the array it returns, whether or not *regex* is a global pattern. This is where `exec()` differs from `String.match()`, which returns much less information when used with global patterns. Calling the `exec()` method repeatedly in a loop is the only way to obtain complete pattern-matching information for a global pattern.

Example

You can use `exec()` in a loop to find all matches within a string. For example:

```

var pattern = /\bJava\w*\b/g;
var text = "JavaScript is more fun than Java or JavaBeans!";
var result;
while((result = pattern.exec(text)) != null) {
    alert("Matched '" + result[0] +
        "' at position " + result.index +
        " next search begins at position " + pattern.lastIndex);
}
    
```

See Also

[RegExp.lastIndex](#), [RegExp.test\(\)](#), [String.match\(\)](#), [String.replace\(\)](#), [String.search\(\)](#), [Chapter 10](#)

RegExp.global

whether a regular expression matches globally

Synopsis

```

regex.global
    
```

Description

`global` is a read-only boolean property of `RegExp` objects. It specifies whether a particular regular expression performs global matching—i.e., whether it was created with the “g” attribute.

RegExp.ignoreCase

whether a regular expression is case-insensitive

Synopsis

```

regex.ignoreCase
    
```

Description

`ignoreCase` is a read-only boolean property of `RegExp` objects. It specifies whether a particular regular expression performs case-insensitive matching—i.e., whether it was created with the “i” attribute.

RegExp.lastIndex

the starting position of the next match

Synopsis

```
regexp.lastIndex
```

Description

`lastIndex` is a read/write property of `RegExp` objects. For regular expressions with the “g” attribute set, it contains an integer that specifies the character position immediately following the last match found by the `RegExp.exec()` and `RegExp.test()` methods. These methods use this property as the starting point for the next search they conduct. This allows you to call those methods repeatedly, to loop through all matches in a string. Note that `lastIndex` is not used by `RegExp` objects that do not have the “g” attribute set and do not represent global patterns.

This property is read/write, so you can set it at any time to specify where in the target string the next search should begin. `exec()` and `test()` automatically reset `lastIndex` to 0 when they fail to find a match (or another match). If you begin to search a new string after a successful match of some other string, you have to explicitly set this property to 0.

See Also

`RegExp.exec()`, `RegExp.test()`

RegExp.source

the text of the regular expression

Synopsis

```
regexp.source
```

Description

`source` is a read-only string property of `RegExp` objects. It contains the text of the `RegExp` pattern. This text does not include the delimiting slashes used in regular-expression literals, and it does not include the “g”, “i”, and “m” attributes.

RegExp.test()

test whether a string matches a pattern

Synopsis

```
regexp.test(string)
```

Arguments

string

The string to be tested.

Returns

true if *string* contains text that matches *regexp*; false otherwise.

Throws

TypeError

If this method is invoked on an object that is not a RegExp.

Description

test() tests *string* to see if it contains text that matches *regexp*. If so, it returns true; otherwise, it returns false. Calling the test() method of a RegExp *r* and passing it the string *s* is equivalent to the following expression:

```
(r.exec(s) != null)
```

Example

```
var pattern = /java/i;
pattern.test("JavaScript"); // Returns true
pattern.test("ECMAScript"); // Returns false
```

See Also

RegExp.exec(), RegExp.lastIndex, String.match(), String.replace(), String.substring();
Chapter 10

RegExp.toString()

convert a regular expression to a string

Overrides Object.toString()

Synopsis

```
regexp.toString()
```

Returns

A string representation of *regexp*.

Throws

TypeError

If this method is invoked on an object that is not a RegExp.

Description

The `RegExp.toString()` method returns a string representation of a regular expression in the form of a regular-expression literal.

Note that implementations are not required to add escape sequences to ensure that the returned string is a legal regular-expression literal. Consider the regular expression created by the expression `new RegExp("/", "g")`. An implementation of `RegExp.toString()` could return `///g` for this regular expression; it could also add an escape sequence and return `/\\/g`.

String

support for strings

Object → String

Constructor

```
new String(s) // Constructor function
String(s)     // Conversion function
```

Arguments

s

The value to be stored in a `String` object or converted to a primitive string.

Returns

When `String()` is used as a constructor with the `new` operator, it returns a `String` object, which holds the string *s* or the string representation of *s*. When the `String()` constructor is used without the `new` operator, it simply converts *s* to a primitive string and returns the converted value.

Properties

`length`

The number of characters in the string.

Methods

`charAt()`

Extracts the character at a given position from a string.

`charCodeAt()`

Returns the encoding of the character at a given position in a string.

`concat()`

Concatenates one or more values to a string.

`indexOf()`

Searches the string for a character or substring.

`lastIndexOf()`

Searches the string backward for a character or substring.

localeCompare()

Compares strings using locale-specific ordering.

match()

Performs pattern matching with a regular expression.

replace()

Performs a search-and-replace operation with a regular expression.

search()

Searches a string for a substring that matches a regular expression.

slice()

Returns a slice or substring of a string.

split()

Splits a string into an array of strings, breaking at a specified delimiter string or regular expression.

substr()

Extracts a substring of a string; a variant of `substring()`.

substring()

Extracts a substring of a string.

toLowerCase()

Returns a copy of the string, with all characters converted to lowercase.

toString()

Returns the primitive string value.

toUpperCase()

Returns a copy of the string, with all characters converted to uppercase.

trim()

Returns a copy of the string with all leading and trailing whitespace removed.

valueOf()

Returns the primitive string value.

Static Methods**String.fromCharCode()**

Creates a new string using the character codes passed as arguments.

HTML Methods

Since the earliest days of JavaScript, the `String` class has defined a number of methods that return a string modified by placing it within HTML tags. These methods have never been standardized by ECMAScript but can be useful in both client- and server-side JavaScript code that dynamically generates HTML. If you are willing to use nonstandard methods, you might create the HTML source for a bold, red hyperlink with code like this:

```
var s = "click here!";  
var html = s.bold().link("javascript:alert('hello')").fontcolor("red");
```

Because these methods are not standardized, they do not have individual reference entries in the pages that follow:

anchor(*name*)

Returns a copy of the string, in an `<a name=>` environment.

big()

Returns a copy of the string, in a `<big>` environment.

blink()

Returns a copy of the string, in a `<blink>` environment.

bold()

Returns a copy of the string, in a `<b>` environment.

fixed()

Returns a copy of the string, in a `<tt>` environment.

fontcolor(*color*)

Returns a copy of the string, in a `<font color=>` environment.

fontsize(*size*)

Returns a copy of the string, in a `<font size=>` environment.

italics()

Returns a copy of the string, in an `<i>` environment.

link(*url*)

Returns a copy of the string, in an `<a href=>` environment.

small()

Returns a copy of the string, in a `<small>` environment.

strike()

Returns a copy of the string, in a `<strike>` environment.

sub()

Returns a copy of the string, in a `<sub>`

sup()

Returns a copy of the string, in a `<sup>` environment.

Description

Strings are a primitive datatype in JavaScript. The String class type exists to provide methods for operating on primitive string values. The `length` property of a String object specifies the number of characters in the string. The String class defines a number of methods for operating on strings; for example, there are methods for extracting a character or a substring from the string or searching for a character or a substring. Note that JavaScript strings are *immutable*: none of the methods defined by the String class allows you to change the contents of a string. Instead, methods such as `String.toUpperCase()` return an entirely new string, without modifying the original.

In ECMAScript 5, and in many JavaScript implementations prior to ES5, strings behave like read-only arrays in which each element is a single-character string. For example, to extract

the third character from a string *s*, you can write *s*[2] instead of *s.charAt*(2). When the *for/in* statement is applied to a string, it enumerates these array indexes for each character in the string.

See Also

Chapter 3

String.charAt()

get the *n*'th character from a string

Synopsis

```
string.charAt(n)
```

Arguments

n

The index of the character that should be returned from *string*.

Returns

The *n*th character of *string*.

Description

String.charAt() returns the *n*th character of the string *string*. The first character of the string is numbered 0. If *n* is not between 0 and *string.length*-1, this method returns an empty string. Note that JavaScript does not have a character data type that is distinct from the string type, so the returned character is a string of length 1.

See Also

String.charCodeAt(), *String.indexOf()*, *String.lastIndexOf()*

String.charCodeAt()

get the *n*th character code from a string

Synopsis

```
string.charCodeAt(n)
```

Arguments

n

The index of the character whose encoding is to be returned.

Returns

The Unicode encoding of the *n*th character within *string*. This return value is a 16-bit integer between 0 and 65535.

Description

`charCodeAt()` is like `charAt()`, except that it returns the character encoding at a specific location, rather than returning a substring that contains the character itself. If *n* is negative or greater than or equal to the string length, `charCodeAt()` returns NaN.

See `String.fromCharCode()` for a way to create a string from Unicode encodings.

See Also

`String.charAt()`, `String.fromCharCode()`

String.concat()

concatenate strings

Synopsis

```
string.concat(value, ...)
```

Arguments

value, ...

One or more values to be concatenated to *string*.

Returns

A new string that results from concatenating each of the arguments to *string*.

Description

`concat()` converts each of its arguments to a string (if necessary) and appends them, in order, to the end of *string*. It returns the resulting concatenation. Note that *string* itself is not modified.

`String.concat()` is an analog to `Array.concat()`. Note that it is often easier to use the `+` operator to perform string concatenation.

See Also

`Array.concat()`

String.fromCharCode()

create a string from character encodings

Synopsis

```
String.fromCharCode(c1, c2, ...)
```

Arguments

c1, *c2*, ...

Zero or more integers that specify the Unicode encodings of the characters in the string to be created.

Returns

A new string containing characters with the specified encodings.

Description

This static method provides a way to create a string by specifying the individual numeric Unicode encodings of its characters. Note that as a static method, `fromCharCode()` is a property of the `String()` constructor and is not actually a method of strings or `String` objects.

`String.charCodeAt()` is a companion instance method that provides a way to obtain the encodings of the individual characters of a string.

Example

```
// Create the string "hello"
var s = String.fromCharCode(104, 101, 108, 108, 111);
```

See Also

`String.charCodeAt()`

String.indexOf()

search a string

Synopsis

```
string.indexOf(substring)
string.indexOf(substring, start)
```

Arguments***substring***

The substring that is to be searched for within *string*.

start

An optional integer argument that specifies the position within *string* at which the search is to start. Legal values are 0 (the position of the first character in the string) to *string*.length-1 (the position of the last character in the string). If this argument is omitted, the search begins at the first character of the string.

Returns

The position of the first occurrence of *substring* within *string* that appears after the *start* position, if any, or -1 if no such occurrence is found.

Description

`String.indexOf()` searches the string *string* from beginning to end to see if it contains an occurrence of *substring*. The search begins at position *start* within *string*, or at the beginning of *string* if *start* is not specified. If an occurrence of *substring* is found, `String.indexOf()` returns the position of the first character of the first occurrence of *substring* within *string*. Character positions within *string* are numbered starting with zero.

If no occurrence of *substring* is found within *string*, `String.indexOf()` returns `-1`.

See Also

`String.charAt()`, `String.lastIndexOf()`, `String.substring()`

String.lastIndexOf()

search a string backward

Synopsis

```
string.lastIndexOf(substring)
string.lastIndexOf(substring, start)
```

Arguments

substring

The substring to be searched for within *string*.

start

An optional integer argument that specifies the position within *string* where the search is to start. Legal values are from 0 (the position of the first character in the string) to *string*.length-1 (the position of the last character in the string). If this argument is omitted, the search begins with the last character of the string.

Returns

The position of the last occurrence of *substring* within *string* that appears before the *start* position, if any, or `-1` if no such occurrence is found within *string*.

Description

`String.lastIndexOf()` searches the string from end to beginning to see if it contains an occurrence of *substring*. The search begins at position *start* within *string*, or at the end of *string* if *start* is not specified. If an occurrence of *substring* is found, `String.lastIndexOf()` returns the position of the first character of that occurrence. Since this method searches from end to beginning of the string, the first occurrence found is the last one in the string that occurs before the *start* position.

If no occurrence of *substring* is found, `String.lastIndexOf()` returns `-1`.

Note that although `String.lastIndexOf()` searches *string* from end to beginning, it still numbers character positions within *string* from the beginning. The first character of the string has position 0, and the last has position *string*.length-1.

See Also

`String.charAt()`, `String.indexOf()`, `String.substring()`

String.length

the length of a string

Synopsis

string.length

Description

The `String.length` property is a read-only integer that indicates the number of characters in the specified *string*. For any string *s*, the index of the last character is *s*.length-1. The length property of a string is not enumerated by a `for/in` loop and may not be deleted with the delete operator.

String.localeCompare()

compare one string to another, using locale-specific ordering

Synopsis

string.localeCompare(*target*)

Arguments

target

A string to be compared, in a locale-sensitive fashion, with *string*.

Returns

A number that indicates the result of the comparison. If *string* is “less than” *target*, `localeCompare()` returns a number less than zero. If *string* is “greater than” *target*, the method returns a number greater than zero. And if the strings are identical or indistinguishable according to the locale ordering conventions, the method returns 0.

Description

When the `<` and `>` operators are applied to strings, they compare those strings using only the Unicode encodings of those characters and do not consider the collation order of the current locale. The ordering produced in this way is not always correct. Consider Spanish, for example, in which the letters “ch” are traditionally sorted as if they were a single letter that appeared between the letters “c” and “d”.

`localeCompare()` provides a way to compare strings that does take the collation order of the default locale into account. The ECMAScript standard does not specify how the locale-specific comparison is done; it merely specifies that this function utilize the collation order provided by the underlying operating system.

Example

You can use code like the following to sort an array of strings into a locale-specific ordering:

```
var strings; // The array of strings to sort; initialized elsewhere
strings.sort(function(a,b) { return a.localeCompare(b) });
```

String.match()

find one or more regular-expression matches

Synopsis

```
string.match(regex)
```

Arguments

regex

A RegExp object that specifies the pattern to be matched. If this argument is not a RegExp, it is first converted to one by passing it to the `RegExp()` constructor.

Returns

An array containing the results of the match. The contents of the array depend on whether *regex* has the global “g” attribute set. Details on this return value are given in the Description.

Description

`match()` searches *string* for one or more matches of *regex*. The behavior of this method depends significantly on whether *regex* has the “g” attribute or not (see Chapter 10 for full details on regular expressions).

If *regex* does not have the “g” attribute, `match()` searches *string* for a single match. If no match is found, `match()` returns `null`. Otherwise, it returns an array containing information about the match that it found. Element 0 of the array contains the matched text. The remaining elements contain the text that matches any parenthesized subexpressions within the regular expression. In addition to these normal array elements, the returned array also has two object properties. The `index` property of the array specifies the character position within *string* of the start of the matched text. Also, the `input` property of the returned array is a reference to *string* itself.

If *regex* has the “g” flag, `match()` does a global search, searching *string* for all matching substrings. It returns `null` if no match is found, and it returns an array if one or more matches are found. The contents of this returned array are quite different for global matches, however. In this case, the array elements contain each of the matched substrings within *string*. The returned array does not have `index` or `input` properties in this case. Note that for global matches, `match()` does not provide information about parenthesized subexpressions, nor does it specify where within *string* each match occurred. If you need to obtain this information for a global search, you can use `RegExp.exec()`.

Example

The following global match finds all numbers within a string:

```
"1 plus 2 equals 3".match(/\d+/g) // Returns ["1", "2", "3"]
```

The following nonglobal match uses a more complex regular expression with several parenthesized subexpressions. It matches a URL, and its subexpressions match the protocol, host, and path portions of the URL:

```
var url = /(\w+):\/\/([\\w.]+)\/(\\S*)/;
var text = "Visit my home page at http://www.isp.com/~david";
```

```

var result = text.match(url);
if (result != null) {
    var fullurl = result[0]; // Contains "http://www.isp.com/~david"
    var protocol = result[1]; // Contains "http"
    var host = result[2]; // Contains "www.isp.com"
    var path = result[3]; // Contains "~david"
}

```

See Also

RegExp, RegExp.exec(), RegExp.test(), String.replace(), String.search(); Chapter 10

String.replace()

replace substring(s) matching a regular expression

Synopsis

string.replace(regex, replacement)

Arguments

regex

The RegExp object that specifies the pattern to be replaced. If this argument is a string, it is used as a literal text pattern to be searched for; it is not first converted to a RegExp object.

replacement

A string that specifies the replacement text, or a function that is invoked to generate the replacement text. See the Description section for details.

Returns

A new string, with the first match, or all matches, of *regex* replaced with *replacement*.

Description

replace() performs a search-and-replace operation on *string*. It searches *string* for one or more substrings that match *regex* and replaces them with *replacement*. If *regex* has the global "g" attribute specified, *replace()* replaces all matching substrings. Otherwise, it replaces only the first matching substring.

replacement may be a string or a function. If it is a string, each match is replaced by the string. Note that the *\$* character has special meaning within the *replacement* string. As shown in the following table, it indicates that a string derived from the pattern match is used in the replacement.

Characters	Replacement
\$1, \$2, ..., \$99	The text that matched the 1st through 99th parenthesized subexpression within <i>regexp</i>
\$&	The substring that matched <i>regexp</i>
\$'	The text to the left of the matched substring
\$'	The text to the right of the matched substring
\$\$	A literal dollar sign

ECMAScript v3 specifies that the *replacement* argument to `replace()` may be a function instead of a string. In this case, the function is invoked for each match, and the string it returns is used as the replacement text. The first argument to the function is the string that matches the pattern. The next arguments are the strings that match any parenthesized subexpressions within the pattern; there may be zero or more of these arguments. The next argument is an integer that specifies the position within *string* at which the match occurred, and the final argument to the *replacement* function is *string* itself.

Example

To ensure that the capitalization of the word "JavaScript" is correct:

```
text.replace(/javascript/i, "JavaScript");
```

To convert a single name from "Doe, John" format to "John Doe" format:

```
name.replace(/(\w+)\s*,\s*(\w+)/, "$2 $1");
```

To replace all double quotes with double back and forward single quotes:

```
text.replace(/"([^"]*)"/g, "'$1'");
```

To capitalize the first letter of all words in a string:

```
text.replace(/\b\w+\b/g, function(word) {
    return word.substring(0,1).toUpperCase() +
           word.substring(1);
});
```

See Also

`RegExp`, `RegExp.exec()`, `RegExp.test()`, `String.match()`, `String.search()`; Chapter 10

String.search()

search for a regular expression

Synopsis

```
string.search(regexp)
```

Arguments

regexp

A `RegExp` object that specifies the pattern to be searched for in *string*. If this argument is not a `RegExp`, it is first converted to one by passing it to the `RegExp()` constructor.

Returns

The position of the start of the first substring of *string* that matches *regexp*, or -1 if no match is found.

Description

`search()` looks for a substring matching *regexp* within *string* and returns the position of the first character of the matching substring, or -1 if no match was found.

`search()` does not do global matches; it ignores the *g* flag. It also ignores the `lastIndex` property of *regexp* and always searches from the beginning of the string, which means that it always returns the position of the first match in *string*.

Example

```
var s = "JavaScript is fun";
s.search(/script/i) // Returns 4
s.search(/a(.)/)    // Returns 1
```

See Also

RegExp, RegExp.exec(), RegExp.test(), String.match(), String.replace(); Chapter 10

String.slice()

extract a substring

Synopsis

```
string.slice(start, end)
```

Arguments**start**

The string index where the slice is to begin. If negative, this argument specifies a position measured from the end of the string. That is, -1 indicates the last character, -2 indicates the second from last character, and so on.

end

The string index immediately after the end of the slice. If not specified, the slice includes all characters from *start* to the end of the string. If this argument is negative, it specifies a position measured from the end of the string.

Returns

A new string that contains all the characters of *string* from and including *start*, and up to but not including *end*.

Description

`slice()` returns a string containing a slice, or substring, of *string*. It does not modify *string*.

The String methods `slice()`, `substring()`, and the deprecated `substr()` all return specified portions of a string. `slice()` is more flexible than `substring()` because it allows negative ar-

gument values. `slice()` differs from `substr()` in that it specifies a substring with two character positions, while `substr()` uses one position and a length. Note also that `String.slice()` is an analog of `Array.slice()`.

Example

```
var s = "abcdefg";
s.slice(0,4)    // Returns "abcd"
s.slice(2,4)    // Returns "cd"
s.slice(4)      // Returns "efg"
s.slice(3,-1)   // Returns "def"
s.slice(3,-2)   // Returns "de"
s.slice(-3,-1)  // Should return "ef"; returns "abcdef" in IE 4
```

Bugs

Negative values for *start* do not work in Internet Explorer 4 (but they do in later versions of IE). Instead of specifying a character position measured from the end of the string, they specify character position 0.

See Also

`Array.slice()`, `String.substring()`

String.split()

break a string into an array of strings

Synopsis

```
string.split(delimiter, limit)
```

Arguments

delimiter

The string or regular expression at which the *string* splits.

limit

This optional integer specifies the maximum length of the returned array. If specified, no more than this number of substrings will be returned. If not specified, the entire string will be split, regardless of its length.

Returns

An array of strings, created by splitting *string* into substrings at the boundaries specified by *delimiter*. The substrings in the returned array do not include *delimiter* itself, except in the case noted in the Description.

Description

The `split()` method creates and returns an array of as many as *limit* substrings of the specified string. These substrings are created by searching the string from start to end for text that matches *delimiter* and breaking the string before and after that matching text. The delimiting text is not included in any of the returned substrings, except as noted at the end of this section.

Note that if the delimiter matches the beginning of the string, the first element of the returned array will be an empty string—the text that appears before the delimiter. Similarly, if the delimiter matches the end of the string, the last element of the array (assuming no conflicting *limit*) will be the empty string.

If no *delimiter* is specified, the string is not split at all, and the returned array contains only a single, unbroken string element. If *delimiter* is the empty string or a regular expression that matches the empty string, the string is broken between each character, and the returned array has the same length as the string does, assuming no smaller *limit* is specified. (Note that this is a special case because the empty strings before the first character and after the last character are not matched.)

As noted earlier, the substrings in the array returned by this method do not contain the delimiting text used to split the string. However, if *delimiter* is a regular expression that contains parenthesized subexpressions, the substrings that match those parenthesized subexpressions (but not the text that matches the regular expression as a whole) are included in the returned array.

Note that the `String.split()` method is the inverse of the `Array.join()` method.

Example

The `split()` method is most useful when you are working with highly structured strings. For example:

```
"1:2:3:4:5".split(":"); // Returns ["1","2","3","4","5"]
"|a|b|c|".split("|"); // Returns ["", "a", "b", "c", ""]
```

Another common use of the `split()` method is to parse commands and similar strings by breaking them down into words delimited by spaces:

```
var words = sentence.split(' ');
```

It is easier to split a string into words using a regular expression as a delimiter:

```
var words = sentence.split(/\s+/);
```

To split a string into an array of characters, use the empty string as the delimiter. Use the *limit* argument if you only want to split a prefix of the string into an array of characters:

```
"hello".split(""); // Returns ["h","e","l","l","o"]
"hello".split("", 3); // Returns ["h","e","l"]
```

If you want the delimiters or one or more portions of the delimiter included in the returned array, use a regular expression with parenthesized subexpressions. For example, the following code breaks a string at HTML tags and includes those tags in the returned array:

```
var text = "hello <b>world</b>";
text.split(/(<[>]*)/); // Returns ["hello ", "<b>", "world", "</b>", ""]
```

See Also

`Array.join()`, `RegExp`; Chapter 10

String.substr()

deprecated

extract a substring

Synopsis

string.substr(start, length)

Arguments

start

The start position of the substring. If this argument is negative, it specifies a position measured from the end of the string: -1 specifies the last character, -2 specifies the second-to-last character, and so on.

length

The number of characters in the substring. If this argument is omitted, the returned substring includes all characters from the starting position to the end of the string.

Returns

A copy of the portion of *string* starting at and including the character specified by *start* and continuing for *length* characters, or to the end of the string if *length* is not specified.

Description

substr() extracts and returns a substring of *string*. It does not modify *string*.

Note that *substr()* specifies the desired substring with a character position and a length. This provides a useful alternative to *String.substring()* and *String.splice()*, which specify a substring with two character positions. Note, however, that this method has not been standardized by ECMAScript and is therefore deprecated.

Example

```
var s = "abcdefg";
s.substr(2,2); // Returns "cd"
s.substr(3);   // Returns "defg"
s.substr(-3,2); // Should return "ef"; returns "ab" in IE 4
```

Bugs

Negative values for *start* do not work in IE. Instead of specifying a character position measured from the end of the string, they specify character position 0.

See Also

String.slice(), *String.substring()*

String.substring()

return a substring of a string

Synopsis

string.substring(from, to)

Arguments

from

A nonnegative integer that specifies the position within *string* of the first character of the desired substring.

to

A nonnegative optional integer that is one greater than the position of the last character of the desired substring. If this argument is omitted, the returned substring runs to the end of the string.

Returns

A new string, of length *to-from*, which contains a substring of *string*. The new string contains characters copied from positions *from* to *to*-1 of *string*.

Description

String.substring() returns a substring of *string* consisting of the characters between positions *from* and *to*. The character at position *from* is included, but the character at position *to* is not included.

If *from* equals *to*, this method returns an empty (length 0) string. If *from* is greater than *to*, this method first swaps the two arguments and then returns the substring between them.

It is important to remember that the character at position *from* is included in the substring but that the character at position *to* is not included in the substring. While this may seem arbitrary or counterintuitive, a notable feature of this system is that the length of the returned substring is always equal to *to-from*.

Note that String.slice() and the nonstandard String.substr() can also extract substrings from a string. Unlike those methods, String.substring() does not accept negative arguments.

See Also

String.charAt(), String.indexOf(), String.lastIndexOf(), String.slice(), String.substr()

String.toLocaleLowerCase()

convert a string to lowercase

Synopsis

```
string.toLocaleLowerCase()
```

Returns

A copy of *string*, converted to lowercase letters in a locale-specific way. Only a few languages, such as Turkish, have locale-specific case mappings, so this method usually returns the same value as toLowerCase().

See Also

String.toLocaleUpperCase(), String.toLowerCase(), String.toUpperCase()

String.toLocaleUpperCase()

convert a string to uppercase

Synopsis

```
string.toLocaleUpperCase()
```

Returns

A copy of *string*, converted to uppercase letters in a locale-specific way. Only a few languages, such as Turkish, have locale-specific case mappings, so this method usually returns the same value as `toUpperCase()`.

See Also

`String.toLocaleLowerCase()`, `String.toLowerCase()`, `String.toUpperCase()`

String.toLowerCase()

convert a string to lowercase

Synopsis

```
string.toLowerCase()
```

Returns

A copy of *string*, with each uppercase letter converted to its lowercase equivalent, if it has one.

String.toString()

return the string

Overrides `Object.toString()`

Synopsis

```
string.toString()
```

Returns

The primitive string value of *string*. It is rarely necessary to call this method.

Throws

TypeError

If this method is invoked on an object that is not a `String`.

See Also

`String.valueOf()`

String.toUpperCase()

String.toUpperCase()

convert a string to uppercase

Synopsis

string.toUpperCase()

Returns

A copy of *string*, with each lowercase letter converted to its uppercase equivalent, if it has one.

String.trim()

ECMAScript 5

strip leading and trailing whitespace

Synopsis

string.trim()

Returns

A copy of *string*, with all leading and trailing whitespace removed.

See Also

String.replace()

String.valueOf()

return the string

Overrides Object.valueOf()

Synopsis

string.valueOf()

Returns

The primitive string value of *string*.

Throws

TypeError

If this method is invoked on an object that is not a String.

See Also

String.toString()

SyntaxError

thrown to signal a syntax error

Object → Error → SyntaxError

Constructor

new SyntaxError()
new SyntaxError(*message*)

Arguments

message

An optional error message that provides details about the exception. If specified, this argument is used as the value for the `message` property of the `SyntaxError` object.

Returns

A newly constructed `SyntaxError` object. If the *message* argument is specified, the `Error` object uses it as the value of its `message` property; otherwise, it uses an implementation-defined default string as the value of that property. When the `SyntaxError()` constructor is called as a function, without the `new` operator, it behaves just as it does when called with the `new` operator.

Properties

message

An error message that provides details about the exception. This property holds the string passed to the constructor, or an implementation-defined default string. See `Error.message` for details.

name

A string that specifies the type of the exception. All `SyntaxError` objects inherit the value "SyntaxError" for this property.

Description

An instance of the `SyntaxError` class is thrown to signal a syntax error in JavaScript code. The `eval()` method, the `Function()` constructor, and the `RegExp()` constructor may all throw exceptions of this type. See `Error` for details about throwing and catching exceptions.

See Also

`Error`, `Error.message`, `Error.name`

TypeError

thrown when a value is of the wrong type

Object → Error → TypeError

Constructor

```
new TypeError()  
new TypeError(message)
```

Arguments

message

An optional error message that provides details about the exception. If specified, this argument is used as the value for the `message` property of the `TypeError` object.

Returns

A newly constructed `TypeError` object. If the *message* argument is specified, the `Error` object uses it as the value of its `message` property; otherwise, it uses an implementation-defined

default string as the value of that property. When the `TypeError()` constructor is called as a function, without the `new` operator, it behaves just as it does when called with the `new` operator.

Properties

message

An error message that provides details about the exception. This property holds the string passed to the constructor, or an implementation-defined default string. See `Error.message` for details.

name

A string that specifies the type of the exception. All `TypeError` objects inherit the value “`TypeError`” for this property.

Description

An instance of the `TypeError` class is thrown when a value is not of the type expected. This happens most often when you attempt to access a property of a `null` or `undefined` value. It can also occur if you invoke a method defined by one class on an object that is an instance of some other class, or if you use the `new` operator with a value that is not a constructor function, for example. JavaScript implementations are also permitted to throw `TypeError` objects when a built-in function or method is called with more arguments than expected. See `Error` for details about throwing and catching exceptions.

See Also

`Error`, `Error.message`, `Error.name`

undefined

the undefined value

Synopsis

`undefined`

Description

`undefined` is a global property that holds the JavaScript `undefined` value. This is the same value that is returned when you attempt to read the value of a nonexistent object property. The `undefined` property is not enumerated by `for/in` loops and cannot be deleted with the `delete` operator. Note that `undefined` is not a constant and can be set to any other value, something that you should take care not to do.

When testing a value to see whether it is `undefined`, use the `===` operator, because the `==` operator treats the `undefined` value as equal to `null`.

unescape()

deprecated

decode an escaped string

Synopsis

`unescape(s)`

Arguments

s

The string that is to be decoded or “unescaped.”

Returns

A decoded copy of *s*.

Description

`unescape()` is a global function that decodes a string encoded with `escape()`. It decodes *s* by finding and replacing character sequences of the form `%xx` and `%u xxxx` (where *x* represents a hexadecimal digit) with the Unicode characters `\u00xx` and `\u xxxx`.

Although `unescape()` was standardized in the first version of ECMAScript, it has been deprecated and removed from the standard by ECMAScript v3. Implementations of ECMAScript are likely to implement this function, but they are not required to. You should use `decodeURI()` and `decodeURIComponent()` instead of `unescape()`. See `escape()` for more details and an example.

See Also

`decodeURI()`, `decodeURIComponent()`, `escape()`, `String`

URIError

thrown by URI encoding and decoding methods

Object → Error → URIError

Constructor

`new URIError()``new URIError(message)`

Arguments

message

An optional error message that provides details about the exception. If specified, this argument is used as the value for the `message` property of the `URIError` object.

Returns

A newly constructed `URIError` object. If the *message* argument is specified, the `Error` object uses it as the value of its `message` property; otherwise, it uses an implementation-defined default string as the value of that property. When the `URIError()` constructor is called as a function without the `new` operator, it behaves just as it does when called with the `new` operator.

Properties

message

An error message that provides details about the exception. This property holds the string passed to the constructor, or an implementation-defined default string. See `Error.message` for details.

name

A string that specifies the type of the exception. All `URIError` objects inherit the value "URIError" for this property.

Description

An instance of the `URIError` class is thrown by `decodeURI()` and `decodeURIComponent()` if the specified string contains illegal hexadecimal escapes. It can also be thrown by `encodeURI()` and `encodeURIComponent()` if the specified string contains illegal Unicode surrogate pairs. See `Error` for details about throwing and catching exceptions.

See Also

`Error`, `Error.message`, `Error.name`