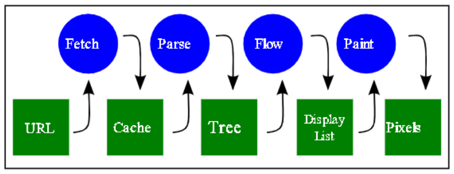
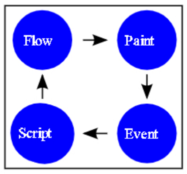
Overview of events and handlers

This overview of events and event handling explains the code design pattern used to react to incidents occurring when a browser accesses a web page, and it summarizes the types of such incidents modern web browsers can handle.

Events and event handling provide a core technique in JavaScript for reacting to incidents occurring when a browser accesses a web page, including events from preparing a web page for display, from interacting with the content of the web page, relating to the device on which the browser is running, and from many other causes such as media stream playback or animation timing.

Events and event handling become central to web programming with the addition of the language to browsers, accompanying a switch in the rendering architecture of browsers from fetch and load page rendering to event driven, reflow based, page rendering. Initially, browsers wait, until they receive all of the resources associated with a page, to parse, process, draw, and present the page to the user. The displayed page remains unchanged until the browser requests a new page. With the change to dynamic page rendering, browsers loop continuously between processing, drawing, presenting content, and waiting for some new event trigger. Event triggers include the completion of the loading of a resource on the network *e.g.,* downloads an image that can now be drawn on the screen, the completion of parsing a resource by the browser *e.g.*, processes the HTML content of a page, the interaction of a user with the contents of the page *e.g.,* clicks a button. Douglas Crockford explains this change effectively in several lectures, notably his talk, *An Inconvenient API: The Theory of the DOM,* which shows the change in flow from the original browser flow





to the event driven browser. The latter approach changes the last steps from a single flow into a perpetual loop, where waiting for and handling the incidence of new events follows painting. The innovation of the dynamic approach allows for a page to be partially rendered even when the browser has not finished retrieving all resources; this approach also allows for event driven actions, which JavaScript leverages. (The talk is available from several sources, including this one.) Currently, all execution environments for JavaScript code use events and event handling.

The event design pattern

The event system, at its core, is simply a programming design pattern. The pattern starts with an agreement over a kind of event and:

- the name String used for the event,
- the type of the data structure used to represent the key properties of that event, and
- the JavaScript object which will 'emit' that event.

The pattern is implemented by

- defining a JavaScript function which takes as an argument the data structure which was agreed upon, and
- registering the function using the name String with the object which will emit the event.

The function is said to be a 'listener' or a 'handler' with both names used interchangeably. This pattern can easily be followed using completely custom code, as explained in the article on custom events. The pattern is also used by modern web browsers which define many events emitted in response to user input or browser activity.

Modern web browsers follow the event pattern using a standardized approach. Browsers use as the data structure for the properties of the event, an object derived from the `EventPrototype` object. Browsers use as the registration method for the function which will handle those data structures a method called `addEventListener` which expects as arguments a string event type name and the handler function. Finally, browsers define a large number of objects as event emitters and define a wide variety of event types generated by the objects.

Button event handler demo

For example, browser `<button>` elements are intended to emit events named `'click'` in response to a mouse click or, when displayed in touch sensitive surfaces, to a finger tap. We could define in the HTML page a `button` as:

```
<button id="buttonOne">Click here to emit a 'click' event</button>
```

and, in our JavaScript code, we could first define a function to listen to that `'click'` event:

```
var example_click_handler = function (ev){
    var objKind = (ev instanceof Event) ? "EventPrototype" : "ObjectPrototype";
    alert("We got a click event at " + ev.timeStamp + " with an argument object derived from: " + objKind );
};
```

and second register our function with the the `Button` object, either on the scripting side using the DOM (Document Object Model) representation of the HTML page:

```
var buttonDOMElement = document.querySelector('#buttonOne');
buttonDOMElement.addEventListener('click', example_click_handler);
```

or within the HTML page by adding the function as the value of the `'onclick'` attribute, although this second approach is usually only used in very simple web pages.

This code relies on the agreement that there is a kind of event called `'click'` which will call any listener (or 'handler') function with an `Event` object argument (actually, in this case a derivative `MouseEvent` object) and which will be fired by HTML `button` elements after user interaction. The code has no visible effect until a user interacts with the button either by placing the mouse pointer over the HTML button and clicking on the left mouse button or by placing a finger or stylus of some kind on the screen above the HTML button; when that happens, the `buttonDOMElement` JavaScript object would call the `example_click_handler` function with an `Event` object as an argument. The function, in turn, would perform whatever action was chosen by the programmer, in this case to open an HTML alert dialog. Note that the handler has access to the `ev` object since it is passed as an argument; the object has information about the event, notably the time at which the event occurred.

As a second example, much modern JavaScript integrated into web pages is wrapped into an event function call to ensure that the code is only executed when the HTML has been processed and is available for alteration or decoration. For example, code might be attached as:

```
var funcInit = function(){
    // user code goes here and can safely address all the HTML elements from the page
    // since the document has successfully been 'loaded'
}
document.addEventListener('DOMContentLoaded', funcInit);
```

so that this code will only be executed after the `document` object emits the `'DOMContentLoaded'` event because the HTML has been parsed and Javasript objects created representing each of the nodes of the HTML document. Note that in this example, the code does not even name the event argument to the function because the code never needs to use the data structure describing the event; rather, the code merely needs to wait to run until after then event has happened.

The pattern is therefore easy to learn and implement. The difficulty with events comes from learning the wide variety of events which are generated in modern web browsers. There is also some subtlety in learning how to write the handler functions since such code works asynchronously and potentially will run repeatedly but in slightly different situations.

Notable events

Web browsers define a large number of events so it is not practical to list them all. The Event Reference attempts to maintain a list of the standard Events used in modern web browsers.

In general, we can distinguish events of different kinds based on the object emitting the event including:

- the `window` object, such as due to resizing the browser,
- the `window.screen` object, such as due to changes in device orientation,
- the `document` object, including the loading, modification, user interaction, and unloading of the page,
- the objects in the DOM (document object model) tree including user interactions or modifications,
- the `XMLHttpRequest` objects used for network requests, and
- the media objects such as `audio` and `video`, when the media stream players change state.

although this list is currently incomplete.

Some notable events are:

Note: This list of events will need work to make relevant; that work is awaiting some global reorganization work on the documents. This will also need finding a good explanation of the events involved during page loading, such as discussed partially in *this web page* or in *this Stack Overflow question*.

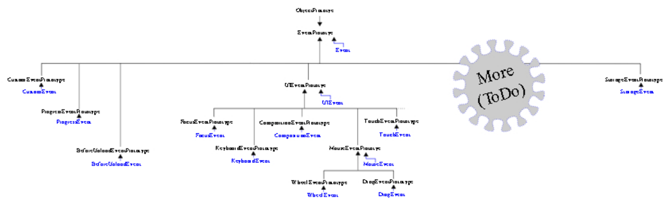
- the global object `window` emits an event called `'load'` when the page has finished rendering, meaning that all resources have been downloaded and acted upon, so that the scripts have been run and the images displayed,
- the global object `window` emits an event called `'resize'` when the height or the width of the browser window is changed by a user,
- the DOM object `document` representing the HTML document emits an event called `'DOMContentLoaded'` when the document has finished loading,
- the DOM node objects such as `div` or `button` emit an event called `'click'` when the user presses the mouse button while the mouse pointer is on top of the DOM node in the HTML page.

The Event object hierarchy

The web browser defines many events of different kinds. Each definition includes, as the data structure passed to the handler function, an object which inherits from the `EventPrototype` object.

A partial diagram of the class hierarchy of event objects is:

Note: This diagram is incomplete.



The Web API Documentation contains a page defining the Event object which also includes the known DOM event subclasses of the `Event` object.

Documents

Three sources on the MDN (Mozilla Developer Network) web site are particularly useful for programmers wanting to work with events:

- this Event Guide which is part of the Web Developers' Guide,
- the Event Reference,
- the Web API documentation for the `Event` object.

Last modified: Apr 10, 2020, by MDN contributors

Learn the best of web development

Get the latest and greatest from MDN delivered straight to your inbox.

you@example.com

Sign up now