The `EventTarget` method **`addEventListener()`** sets up a function that will be called whenever the specified event is delivered to the target. Common targets are `Element`, `Document`, and `Window`, but the target may be any object that supports events (such as `XMLHttpRequest`).

`addEventListener()` works by adding a function or an object that implements `EventListener` to the list of event listeners for the specified event type on the `EventTarget` on which it's called.

Syntax

```
target.addEventListener(type, listener [, options]);
target.addEventListener(type, listener [, useCapture]);
target.addEventListener(type, listener [, useCapture, wantsUntrusted  ]);
// Gecko/Mozilla only
```

Parameters

type
A case-sensitive string representing the event type to listen for.

listener
The object that receives a notification (an object that implements the `Event` interface) when an event of the specified type occurs. This must be an object implementing the `EventListener` interface, or a JavaScript function. See The event listener callback for details on the callback itself.

options Optional
An options object specifies characteristics about the event listener. The available options are:

`capture`
A `Boolean` indicating that events of this type will be dispatched to the registered *`listener`* before being dispatched to any `EventTarget` beneath it in the DOM tree.

`once`
A `Boolean` indicating that the *`listener`* should be invoked at most once after being added. If `true`, the *`listener`* would be automatically removed when invoked.

`passive`
A `Boolean` that, if `true`, indicates that the function specified by *`listener`* will never call `preventDefault()`. If a passive listener does call `preventDefault()`, the user

agent will do nothing other than generate a console warning. See Improving scrolling performance with passive listeners to learn more.

`mozSystemGroup`
A `Boolean` indicating that the listener should be added to the system group. Available only in code running in XBL or in the chrome of the Firefox browser.

`useCapture` Optional
A `Boolean` indicating whether events of this type will be dispatched to the registered *`Listener`* *before* being dispatched to any `EventTarget` beneath it in the DOM tree. Events that are bubbling upward through the tree will not trigger a listener designated to use capture. Event bubbling and capturing are two ways of propagating events that occur in an element that is nested within another element, when both elements have registered a handle for that event. The event propagation mode determines the order in which elements receive the event. See DOM Level 3 Events and JavaScript Event order for a detailed explanation. If not specified, *`useCapture`* defaults to `false`.

> **Note:** For event listeners attached to the event target, the event is in the target phase, rather than the capturing and bubbling phases. Events in the target phase will trigger all listeners on an element in the order they were registered, regardless of the *`useCapture`* parameter.

> **Note:** *`useCapture`* has not always been optional. Ideally, you should include it for the widest possible browser compatibility.

`wantsUntrusted`
A Firefox (Gecko)-specific parameter. If `true`, the listener receives synthetic events dispatched by web content (the default is `false` for browser chrome and `true` for regular web pages). This parameter is useful for code found in add-ons, as well as the browser itself.

Return value

`undefined`

Usage notes

The event listener callback

The event listener can be specified as either a callback function or an object that implements `EventListener`, whose `handleEvent()` method serves as the callback function.

The callback function itself has the same parameters and return value as the `handleEvent()` method; that is, the callback accepts a single parameter: an object based on `Event` describing the event that has occurred, and it returns nothing.

For example, an event handler callback that can be used to handle both `fullscreenchange` and `fullscreenerror` might look like this:

```
function eventHandler(event) {
  if (event.type == 'fullscreenchange') {
    /* handle a full screen toggle */
  } else /* fullscreenerror */ {
    /* handle a full screen toggle error */
  }
}
```

Safely detecting option support

In older versions of the DOM specification, the third parameter of `addEventListener()` was a Boolean value indicating whether or not to use capture. Over time, it became clear that more options were needed. Rather than adding more parameters to the function (complicating things enormously when dealing with optional values), the third parameter was changed to an object that can contain various properties defining the values of options to configure the process of removing the event listener.

Because older browsers (as well as some not-too-old browsers) still assume the third parameter is a Boolean, you need to build your code to handle this scenario intelligently. You can do this by using feature detection for each of the options you're interested in.

For example, if you want to check for the `passive` option:

```
let passiveSupported = false;

try {
  const options = {
    get passive() { // This function will be called when the browser
                    //   attempts to access the passive property.
      passiveSupported = true;
      return false;
    }
  };
```

```
      window.addEventListener("test", null, options);
      window.removeEventListener("test", null, options);
    } catch(err) {
      passiveSupported = false;
    }
```

This creates an *options* object with a getter function for the `passive` property; the getter sets a flag, *passiveSupported*, to `true` if it gets called. That means that if the browser checks the value of the `passive` property on the *options* object, *passiveSupported* will be set to `true`; otherwise, it will remain `false`. We then call `addEventListener()` to set up a fake event handler, specifying those options, so that the options will be checked if the browser recognizes an object as the third parameter. Then, we call `removeEventListener()` to clean up after ourselves. (Note that `handleEvent()` is ignored on event listeners that aren't called.)

You can check whether any option is supported this way. Just add a getter for that option using code similar to what is shown above.

Then, when you want to create an actual event listener that uses the options in question, you can do something like this:

```
someElement.addEventListener("mouseup", handleMouseUp, passiveSupported
                             ? { passive: true } : false);
```

Here we're adding a listener for the `mouseup` event on the element *someElement*. For the third parameter, if *passiveSupported* is `true`, we're specifying an *options* object with `passive` set to `true`; otherwise, we know that we need to pass a Boolean, and we pass `false` as the value of the *useCapture* parameter.

If you'd prefer, you can use a third-party library like Modernizr or Detect It to do this test for you.

You can learn more from the article about `EventListenerOptions` from the Web Incubator Community Group.

Examples

Add a simple listener

This example demonstrates how to use `addEventListener()` to watch for mouse clicks on an element.

HTML

```
<table id="outside">
  <tr><td id="t1">one</td></tr>
  <tr><td id="t2">two</td></tr>
</table>
```

JavaScript

```
// Function to change the content of t2
function modifyText() {
  const t2 = document.getElementById("t2");
  if (t2.firstChild.nodeValue == "three") {
    t2.firstChild.nodeValue = "two";
  } else {
    t2.firstChild.nodeValue = "three";
  }
}

// Add event listener to table
const el = document.getElementById("outside");
el.addEventListener("click", modifyText, false);
```

In this code, `modifyText()` is a listener for `click` events registered using `addEventListener()`. A click anywhere in the table bubbles up to the handler and runs `modifyText()`.

Result

Event listener with anonymous function

Here, we'll take a look at how to use an anonymous function to pass parameters into the event listener.

HTML

```
<table id="outside">
  <tr><td id="t1">one</td></tr>
  <tr><td id="t2">two</td></tr>
</table>
```

JavaScript

```
// Function to change the content of t2
function modifyText(new_text) {
  const t2 = document.getElementById("t2");
  t2.firstChild.nodeValue = new_text;
}

// Function to add event listener to table
const el = document.getElementById("outside");
el.addEventListener("click", function(){modifyText("four")}, false);
```

Notice that the listener is an anonymous function that encapsulates code that is then, in turn, able to send parameters to the `modifyText()` function, which is responsible for actually responding to the event.

Result

Event listener with an arrow function

This example demonstrates a simple event listener implemented using arrow function notation.

HTML

```
<table id="outside">
  <tr><td id="t1">one</td></tr>
  <tr><td id="t2">two</td></tr>
</table>
```

JavaScript

```
// Function to change the content of t2
function modifyText(new_text) {
  const t2 = document.getElementById("t2");
  t2.firstChild.nodeValue = new_text;
}

// Add event listener to table with an arrow function
const el = document.getElementById("outside");
el.addEventListener("click", () => { modifyText("four"); }, false);
```

Result

Please note that while anonymous and arrow functions are similar, they have different `this` bindings. While anonymous (and all traditional JavaScript functions) create their own `this` bindings, arrow functions inherit the `this` binding of the containing function.

That means that the variables and constants available to the containing function are also available to the event handler when using an arrow function.

Example of options usage

HTML

```
<div class="outer">
  outer, once & none-once
  <div class="middle" target="_blank">
    middle, capture & none-capture
    <a class="inner1" href="https://www.mozilla.org" target="_blank">
      inner1, passive & preventDefault(which is not allowed)
    </a>
    <a class="inner2" href="https://developer.mozilla.org/" target="_bl
      inner2, none-passive & preventDefault(not open new page)
    </a>
  </div>
</div>
```

CSS

```
.outer, .middle, .inner1, .inner2 {
  display: block;
  width:   520px;
  padding: 15px;
  margin:  15px;
  text-decoration: none;
}
.outer {
  border: 1px solid red;
  color:  red;
}
.middle {
  border: 1px solid green;
  color:  green;
  width:  460px;
}
.inner1, .inner2 {
  border: 1px solid purple;
  color:  purple;
  width:  400px;
}
```

JavaScript

```
const outer  = document.querySelector('.outer');
const middle = document.querySelector('.middle');
const inner1 = document.querySelector('.inner1');
const inner2 = document.querySelector('.inner2');

const capture = {
  capture : true
};
const noneCapture = {
  capture : false
};
const once = {
  once : true
```

```
};
const noneOnce = {
  once : false
};
const passive = {
  passive : true
};
const nonePassive = {
  passive : false
};

outer.addEventListener('click', onceHandler, once);
outer.addEventListener('click', noneOnceHandler, noneOnce);

middle.addEventListener('click', captureHandler, capture);
middle.addEventListener('click', noneCaptureHandler, noneCapture);
inner1.addEventListener('click', passiveHandler, passive);
inner2.addEventListener('click', nonePassiveHandler, nonePassive);

function onceHandler(event) {
  alert('outer, once');
}
function noneOnceHandler(event) {
  alert('outer, none-once, default');
}
function captureHandler(event) {
  //event.stopImmediatePropagation();
  alert('middle, capture');
}
function noneCaptureHandler(event) {
  alert('middle, none-capture, default');
}
function passiveHandler(event) {
  // Unable to preventDefault inside passive event listener invocation.
  event.preventDefault();
  alert('inner1, passive, open new page');
}
function nonePassiveHandler(event) {
  event.preventDefault();
  //event.stopPropagation();
  alert('inner2, none-passive, default, not open new page');
```

```
    }
```

Result

Click the outer, middle, inner containers respectively to see how the options work.

Before using a particular value in the *`options`* object, it's a good idea to ensure that the user's browser supports it, since these are an addition that not all browsers have supported historically. See Safely detecting option support for details.

Other notes

Why use addEventListener()?

`addEventListener()` is the way to register an event listener as specified in W3C DOM. The benefits are as follows:

- It allows adding more than a single handler for an event. This is particularly useful for AJAXlibraries, JavaScript modules, or any other kind of code that needs to work well with other libraries/extensions.
- It gives you finer-grained control of the phase when the listener is activated (capturing vs. bubbling).
- It works on any DOM element, not just HTML elements.

The alternative, older way to register event listeners, is described below.

Adding a listener during event dispatch

If an `EventListener` is added to an `EventTarget` while it is processing an event, that event does not trigger the listener. However, that same listener may be triggered during a later stage of event flow, such as the bubbling phase.

Multiple identical event listeners

If multiple identical `EventListener`s are registered on the same `EventTarget` with the same parameters, the duplicate instances are discarded. They do not cause the `EventListener` to be

called twice, and they do not need to be removed manually with the `removeEventListener()` method.

Note, however that when using an anonymous function as the handler, such listeners will NOT be identical, because anonymous functions are not identical even if defined using the SAME unchanging source-code simply called repeatedly, even if in a loop.

However, repeatedly defining the same named function in such cases can be more problematic. (See Memory issues, below.)

The value of "this" within the handler

It is often desirable to reference the element on which the event handler was fired, such as when using a generic handler for a set of similar elements.

If attaching a handler function to an element using `addEventListener()`, the value of `this` inside the handler is a reference to the element. It is the same as the value of the `currentTarget` property of the event argument that is passed to the handler.

```
my_element.addEventListener('click', function (e) {
  console.log(this.className)           // logs the className of my_ele
  console.log(e.currentTarget === this) // logs `true`
})
```

As a reminder, arrow functions do not have their own `this` context.

```
my_element.addEventListener('click', (e) => {
  console.log(this.className)           // WARNING: `this` is not `my_e
  console.log(e.currentTarget === this) // logs `false`
})
```

If an event handler (for example, `onclick`) is specified on an element in the HTML source, the JavaScript code in the attribute value is effectively wrapped in a handler function that binds the value of `this` in a manner consistent with the `addEventListener()`; an occurrence of `this` within the code represents a reference to the element.

```
<table id="my_table" onclick="console.log(this.id);"><!-- `this` refers
  ...
</table>
```

Note that the value of `this` inside a function, *called by* the code in the attribute value, behaves as per standard rules. This is shown in the following example:

```
<script>
  function logID() { console.log(this.id); }
</script>
<table id="my_table" onclick="logID();"><!-- when called, `this` will r
  ...
</table>
```

The value of `this` within `logID()` is a reference to the global object `Window` (or `undefined` in the case of strict mode.

Specifying "this" using bind()

The `Function.prototype.bind()` method lets you specify the value that should be used as `this` for all calls to a given function. This lets you easily bypass problems where it's unclear what `this` will be, depending on the context from which your function was called. Note, however, that you'll need to keep a reference to the listener around so you can remove it later.

This is an example with and without `bind()`:

```
const Something = function(element) {
  // |this| is a newly created object
  this.name = 'Something Good';
  this.onclick1 = function(event) {
    console.log(this.name); // undefined, as |this| is the element
  };
  this.onclick2 = function(event) {
    console.log(this.name); // 'Something Good', as |this| is bound to
  };
  element.addEventListener('click', this.onclick1, false);
  element.addEventListener('click', this.onclick2.bind(this), false); /
}
const s = new Something(document.body);
```

Another solution is using a special function called `handleEvent()` to catch any events:

```
const Something = function(element) {
  // |this| is a newly created object
  this.name = 'Something Good';
  this.handleEvent = function(event) {
    console.log(this.name); // 'Something Good', as this is bound to ne
    switch(event.type) {
      case 'click':
        // some code here...
        break;
      case 'dblclick':
        // some code here...
        break;
    }
  };

  // Note that the listeners in this case are |this|, not this.handleEv
  element.addEventListener('click', this, false);
  element.addEventListener('dblclick', this, false);

  // You can properly remove the listeners
  element.removeEventListener('click', this, false);
  element.removeEventListener('dblclick', this, false);
}
const s = new Something(document.body);
```

Another way of handling the reference to *this* is to pass to the `EventListener` a function that calls the method of the object that contains the fields that need to be accessed:

```
class SomeClass {

  constructor() {
    this.name = 'Something Good';
  }

  register() {
    const that = this;
    window.addEventListener('keydown', function(e) { that.someMethod(e)
  }
```

```
    someMethod(e) {
      console.log(this.name);
      switch(e.keyCode) {
        case 5:
          // some code here...
          break;
        case 6:
          // some code here...
          break;
      }

    }

}

const myObject = new SomeClass();
myObject.register();
```

Getting data into and out of an event listener

It may seem that event listeners are like islands, and that it is extremely difficult to pass them any data, much less to get any data back from them after they execute. Event listeners only take one argument, the Event Object, which is automatically passed to the listener, and the return value is ignored. So how can we get data in and back out of them again? There are a number of good methods for doing this.

Getting data into an event listener using "this"

As mentioned above, you can use `Function.prototype.bind()` to pass a value to an event listener via the `this` reference variable.

```
const myButton = document.getElementById('my-button-id');
const someString = 'Data';

myButton.addEventListener('click', function () {
  console.log(this);  // Expected Value: 'Data'
}.bind(someString));
```

This method is suitable when you don't need to know which HTML element the event listener fired on programmatically from within the event listener. The primary benefit to doing this is that the event listener receives the data in much the same way that it would if you were to actually pass it through its argument list.

Getting data into an event listener using the outer scope property

When an outer scope contains a variable declaration (with `const`, `let`), all the inner functions declared in that scope have access to that variable (look here for information on outer/inner functions, and here for information on variable scope). Therefore, one of the simplest ways to access data from outside of an event listener is to make it accessible to the scope in which the event listener is declared.

```
const myButton = document.getElementById('my-button-id');
const someString = 'Data';

myButton.addEventListener('click', function() {
  console.log(someString);  // Expected Value: 'Data'

  someString = 'Data Again';
});

console.log(someString);  // Expected Value: 'Data' (will never output
```

> **Note:** Although inner scopes have access to `const`, `let` variables from outer scopes, you cannot expect any changes to these variables to be accessible after the event listener definition, within the same outer scope. Why? Simply because by the time the event listener would execute, the scope in which it was defined would have already finished executing.

Getting data into and out of an event listener using objects

Unlike most functions in JavaScript, objects are retained in memory as long as a variable referencing them exists in memory. This, and the fact that objects can have properties, and that they can be passed around by reference, makes them likely candidates for sharing data among scopes. Let's explore this.

> **Note:** Functions in JavaScript are actually objects. (Hence they too can have properties, and will be retained in memory even after they finish executing if assigned to a variable that persists in memory.)

Because object properties can be used to store data in memory as long as a variable referencing the object exists in memory, you can actually use them to get data into an event listener, and any changes to the data back out after an event handler executes. Consider this example.

```
const myButton = document.getElementById('my-button-id');
const someObject = {aProperty: 'Data'};

myButton.addEventListener('click', function() {
  console.log(someObject.aProperty);  // Expected Value: 'Data'

  someObject.aProperty = 'Data Again';  // Change the value
});

window.setInterval(function() {
  if (someObject.aProperty === 'Data Again') {
    console.log('Data Again: True');
    someObject.aProperty = 'Data';  // Reset value to wait for next eve
  }
}, 5000);
```

In this example, even though the scope in which both the event listener and the interval function are defined would have finished executing before the original value of `someObject.aProperty` would have changed, because `someObject` persists in memory (by *reference*) in both the event listener and interval function, both have access to the same data (i.e. when one changes the data, the other can respond to the change).

> **Note:** Objects are stored in variables by reference, meaning only the memory location of the actual data is stored in the variable. Among other things, this means variables that "store" objects can actually affect other variables that get assigned ("store") the same object reference. When two variables reference the same object (e.g., `let a = b = {aProperty: 'Yeah'};`), changing the data in either variable will affect the other.

> **Note:** Because objects are stored in variables by reference, you can return an object from a function to keep it alive (preserve it in memory so you don't lose the data) after that function stops executing.

Legacy Internet Explorer and attachEvent

In Internet Explorer versions before IE 9, you have to use `attachEvent()`, rather than the standard `addEventListener()`. For IE, we modify the preceding example to:

```
if (el.addEventListener) {
  el.addEventListener('click', modifyText, false);
} else if (el.attachEvent)  {
  el.attachEvent('onclick', modifyText);
}
```

There is a drawback to `attachEvent()`: The value of `this` will be a reference to the `window` object, instead of the element on which it was fired.

The `attachEvent()` method could be paired with the `onresize` event to detect when certain elements in a web page were resized. The proprietary `mselementresize` event, when paired with the `addEventListener` method of registering event handlers, provides similar functionality as `onresize`, firing when certain HTML elements are resized.

Polyfill

You can work around `addEventListener()`, `removeEventListener()`, `Event.preventDefault()`, and `Event.stopPropagation()` not being supported by Internet Explorer 8 by using the following code at the beginning of your script. The code supports the use of `handleEvent()` and also the `DOMContentLoaded` event.

> **Note:** `useCapture` is not supported, as IE 8 does not have any alternative method. The following code only adds IE 8 support. This IE 8 polyfill only works in standards mode: a doctype declaration is required.

```
(function() {
  if (!Event.prototype.preventDefault) {
    Event.prototype.preventDefault=function() {
      this.returnValue=false;
    };
  }
  if (!Event.prototype.stopPropagation) {
    Event.prototype.stopPropagation=function() {
      this.cancelBubble=true;
    };
  }
  if (!Element.prototype.addEventListener) {
    var eventListeners=[];
```

```
var addEventListener=function(type,listener /*, useCapture (will be
  var self=this;
  var wrapper=function(e) {
    e.target=e.srcElement;
    e.currentTarget=self;
    if (typeof listener.handleEvent != 'undefined') {
      listener.handleEvent(e);
    } else {
      listener.call(self,e);
    }
  };
  if (type=="DOMContentLoaded") {
    var wrapper2=function(e) {
      if (document.readyState=="complete") {
        wrapper(e);
      }
    };
    document.attachEvent("onreadystatechange",wrapper2);
    eventListeners.push({object:this,type:type,listener:listener,wr

    if (document.readyState=="complete") {
      var e=new Event();
      e.srcElement=window;
      wrapper2(e);
    }
  } else {
    this.attachEvent("on"+type,wrapper);
    eventListeners.push({object:this,type:type,listener:listener,wr
  }
};
var removeEventListener=function(type,listener /*, useCapture (will
  var counter=0;
  while (counter<eventListeners.length) {
    var eventListener=eventListeners[counter];
    if (eventListener.object==this && eventListener.type==type && e
      if (type=="DOMContentLoaded") {
        this.detachEvent("onreadystatechange",eventListener.wrapper
      } else {
        this.detachEvent("on"+type,eventListener.wrapper);
      }
      eventListeners.splice(counter, 1);
```

```
            break;
          }
          ++counter;
        }
      };
      Element.prototype.addEventListener=addEventListener;
      Element.prototype.removeEventListener=removeEventListener;
      if (HTMLDocument) {
        HTMLDocument.prototype.addEventListener=addEventListener;
        HTMLDocument.prototype.removeEventListener=removeEventListener;
      }
      if (Window) {
        Window.prototype.addEventListener=addEventListener;
        Window.prototype.removeEventListener=removeEventListener;

      }
    }
  })();
```

Older way to register event listeners

`addEventListener()` was introduced with the DOM 2 Events specification. Before then, event listeners were registered as follows:

```
// Passing a function reference — do not add '()' after it, which would
el.onclick = modifyText;

// Using a function expression
element.onclick = function() {
  // ... function logic ...
};
```

This method replaces the existing `click` event listener(s) on the element if there are any. Other events and associated event handlers such as `blur` (`onblur`) and `keypress` (`onkeypress`) behave similarly.

Because it was essentially part of DOM 0, this technique for adding event listeners is very widely supported and requires no special cross-browser code. It is used to register event listeners

dynamically when very old browsers (like IE <=8) must be supported; see the table below for details on browser support for `addEventListener`.

Memory issues

```
const els = document.getElementsByTagName('*');

// Case 1
for(let i=0 ; i < els.length; i++){
  els[i].addEventListener("click", function(e){/*do something*/}, false
}

// Case 2
function processEvent(e){
  /* do something */
}

for(let i=0 ; i < els.length; i++){
  els[i].addEventListener("click", processEvent, false);
}
```

In the first case above, a new (anonymous) handler function is created with each iteration of the loop. In the second case, the same previously declared function is used as an event handler, which results in smaller memory consumption because there is only one handler function created. Moreover, in the first case, it is not possible to call `removeEventListener()` because no reference to the anonymous function is kept (or here, not kept to any of the multiple anonymous functions the loop might create.) In the second case, it's possible to do *`myElement`*`.removeEventListener("click", processEvent, false)` because `processEvent` is the function reference.

Actually, regarding memory consumption, the lack of keeping a function reference is not the real issue; rather it is the lack of keeping a STATIC function reference. In both problem-cases below, a function reference is kept, but since it is redefined on each iteration, it is not static. In the third case, the reference to the anonymous function is being reassigned with each iteration. In the fourth case, the entire function definition is unchanging, but it is still being repeatedly defined as if new (unless it was [[promoted]] by the compiler) and so is not static. Therefore, though appearing to be simply [[Multiple identical event listeners]], in both cases each iteration will instead create a new listener with its own unique reference to the handler function. However, since the function definition itself does not change, the SAME function may still be called for every duplicate listener (especially if the code gets optimized.)

Also in both cases, because the function reference was kept but repeatedly redefined with each add, the remove-statement from above can still remove a listener, but now only the last one added.

```
// For illustration only: Note "MISTAKE" of [j] for [i] thus causing de

// Case 3
for(let i=0, j=0 ; i<els.length ; i++){
  /* do lots of stuff with j */
  els[j].addEventListener("click", processEvent = function(e){/*do some
}

// Case 4
for(let i=0, j=0 ; i<els.length ; i++){
  /* do lots of stuff with j */
  function processEvent(e){/*do something*/};
  els[j].addEventListener("click", processEvent, false);
}
```

Improving scrolling performance with passive listeners

According to the specification, the default value for the `passive` option is always `false`. However, this introduces the potential for event listeners handling certain touch events (among others) to block the browser's main thread while it is attempting to handle scrolling, resulting in possibly enormous reduction in performance during scroll handling.

To prevent this problem, some browsers (specifically, Chrome and Firefox) have changed the default value of the `passive` option to `true` for the `touchstart` and `touchmove` events on the document-level nodes `Window`, `Document`, and `Document.body`. This prevents the event listener from being called, so it can't block page rendering while the user is scrolling.

> **Note:** See the compatibility table below if you need to know which browsers (and/or which versions of those browsers) implement this altered behavior.

You can override this behavior by explicitly setting the value of `passive` to `false`, as shown here:

```
/* Feature detection */
let passiveIfSupported = false;

try {
```

```
    window.addEventListener("test", null,
      Object.defineProperty(
        {},
        "passive",
        {
          get: function() { passiveIfSupported = { passive: false }; }
        }
      )
    );
  } catch(err) {}

  window.addEventListener('scroll', function(event) {
    /* do something */
    // can't use event.preventDefault();
  }, passiveIfSupported );
```

On older browsers that don't support the `options` parameter to `addEventListener()`, attempting to use it prevents the use of the `useCapture` argument without proper use of feature detection.

You don't need to worry about the value of `passive` for the basic `scroll` event. Since it can't be canceled, event listeners can't block page rendering anyway.