|  |
| --- |
| [[https://myetudes.org/etudes-melete-tool/images/printer.png](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385321) Send to Printer](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385321) | [Close Window](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385321) |
|  | |
| 10. Event Driven Programming  10.1. Events, Targets & Listeners  *Copyright (c) 2014, Rula Khayrallah*  A client side JavaScript application is for the most part **event driven**.  It sits and waits for events to happen.  Events can come from various sources.  These sources include **user activity** such as key presses and clicks as well as **network** or **browser**state changes.  When a significant event occurs, the application responds to it (we say it **handles it)**then goes back to waiting for the next event.  So we can think of an event as anything significant that happens to the document, to the browser or to some element or object associated with it.  The following entities are associated with a given event:  **The event type** (or event name) describes the event.  The type 'click', for example, means that the user clicked on an element.  The type 'keydown' means that a key on the keyboard was pushed down.  The type 'load' means that a document has finished loading from the network.  For a list of event types, visit <https://developer.mozilla.org/en-US/docs/Web/Reference/Events>  **The event target** is the object on which the event happened. The target could be the window, the document or a specific element in the document.  When a user clicks on a button, for example, the target is that button element.  When the user enters text in an input field, the target is that input element.  **The event handler** or **event listener** is the function that responds to the event.  **An application needs to register the event handler function with the browser, specifying an event type and an event target**. When an event of the specified type occurs on the specified target, the browser invokes the handler.  An event object is an object that contains details about that event. **Event objects are passed as an argument to the event handler function.**  10.2. Registering Event Listeners  *Copyright (c) 2014, Rula Khayrallah*  **The recommended way to register events for current browsers is by using the addEventListener() method.**  The addEventListener() method is invoked on an event target.  It**registers the specified listener on that event target**. The event target may be any object that supports events such as an element in a document, the document itself or the window object.  addEventListener() takes three arguments.  The first is the **event type**for which the handler is being registered.  The second argument is the function that should be invoked when the specified type of event occurs.  That is the **event listener** or handler.  The final argument is **a boolean value**. For now we’ll set it to false.  We’ll revisit this argument a little later.  Note that the addEventListener() method  allows us to register more than on event listener for a given event and a given target.  Let’s go back to our simple calculator example from the last module and illustrate the use of event listeners:  calculator.html  <!DOCTYPE html>  <html>          <head>                  <meta charset="utf-8">                  <title>My Simple Calculator</title>          </head>          <body>                  <h2>Let's add some numbers! </h2>                  <p>Please enter two numbers:  </p>                  <input id = "first" type="number">                  <br>                  <input id = "second" type="number">                  <p> And the answer is:</p>                  <p id="answer"></p>          </body>  </html>  Instead of updating the web page manually from the console like we did in the previous module, we are ready to write a script that will do that as soon as the user enters or modifies one of the input values.  The first step is to add the script as an external file to the source HTML document.  calculator.html  <!DOCTYPE html>  <html>          <head>                  <meta charset="utf-8">                  <title>My Simple Calculator</title>          </head>          <body>                  <h2>Let's add some numbers! </h2>                  <p>Please enter two numbers:  </p>                  <input id = "first" type="number">                  <br>                  <input id = "second" type="number">                  <p> And the answer is:</p>                  <p id="answer"></p>  **<script defer src="../scripts/add.js"></script>**          </body>  </html>  Now we are ready to create our script.  First let’s create the function that will add the two input elements and display the answer.  We’ll call it update.  To write the update function, we basically reuse the code that we wrote at the console prompt in module 9.15.  function update() {        // Get the two input numbers      var firstNumber = Number(document.getElementById('first').value);      var secondNumber = Number(document.getElementById('second').value);        // Then  compute the sum      var myAnswer = firstNumber + secondNumber;        // And write it in the 'answer' element      document.getElementById('answer').textContent = myAnswer;  }  Once we have defined the update function, we only need to add the following two lines to our main program.  document.getElementById('first').addEventListener('input', **update**, false); document.getElementById('second').addEventListener('input', **update**, false);  And we have registered our function update with 'input' events on the two input elements.  The **event type** here is '**input'**.  We have two event targets:  One **target** is the element whose id is 'first'.  That’s the **first input**.  The other **target** is the element whose id is 'second'.  That’s the **second input**.  The **update** function is our**event listener**(or handler).  Let’s save our complete program add.js in our scripts folder and test it with our web page.  add.js  function update() {        // Get the two input numbers      var firstNumber = Number(document.getElementById('first').value);      var secondNumber = Number(document.getElementById('second').value);        // Then  compute the sum      var myAnswer = firstNumber + secondNumber;        // And write it in the 'answer' element      document.getElementById('answer').textContent = myAnswer;  }    document.getElementById('first').addEventListener('input', update, false);  document.getElementById('second').addEventListener('input', update, false);  Let’s open the updated calculator.html file in Firefox.  The following is displayed:  As soon as we enter a number,  the input event is triggered, the function update() is called and the answer is updated.  As soon as we enter the second number the result is updated again:  The input event has been triggered again (on the second input this time) and the function update has been called again.  Every time we change one of the input fields, an input event will be triggered, the function will be called and the answer will be updated.  When we don’t change any input, no change event is triggered and our application is not doing anything:  it’s just sitting there and waiting…  Let’s add one more event listener to our calculator web page.  We’ll register a help function that will be invoked **when the mouse is over the answer**.  It will provide some obvious explanation as to how the answer was computed.  We can do that by adding the following line to our add.js script:  document.getElementById('answer').addEventListener('mouseover', help, false);  The **event type** here is '**mouseover'**.  The event **target** is the **element whose id is 'answer'.**  The **help** function is our**event listener**(or handler).  And now we need to define our help function in add.js:  add.js:  function update() {        // Get the two input numbers      var firstNumber = Number(document.getElementById('first').value);      var secondNumber = Number(document.getElementById('second').value);        // Then  compute the sum      var myAnswer = firstNumber + secondNumber;        // And write it in the 'answer' element      document.getElementById('answer').textContent = myAnswer;  }    function help() {        // check that there is an answer currently displayed      var currentAnswer = Number(document.getElementById('answer').textContent);      if (currentAnswer) {          document.getElementById('answer').textContent =              currentAnswer +              '=' +              document.getElementById('first').value +              '+' +              document.getElementById('second').value;      }  }    // Define our event listeners  document.getElementById('first').addEventListener('input', update, false);  document.getElementById('second').addEventListener('input', update, false);  document.getElementById('answer').addEventListener('mouseover', help, false);    Once we refresh our page in the browser,  we can see this new feature in action when we move our mouse over the answer.  Note that the removeEventListener may be used to de-register an event listener on a given target.  Example:  document.getElementById('answer').removeEventListener('mouseover', help, false);  10.3. The Event Object  *Copyright (c) 2014, Rula Khayrallah*  **The event object is always implicitly passed as an argument to the event listener function.**  Its properties provide details about the event. The type property, for example, specifies the type of the event that occurred and the target property specifies the target of the event.  The update function that we saw earlier did not use the event object.  However there are instances where a listener is registered for several event types and on several targets and it is useful for the function to be able to access the specifics for a given invocation.  There are also some other details that may be relevant for a subset of events and/or targets.  We’ll illustrate that with a simple example that uses the**HTML5 canvas element**.  Here’s some background on the <canvas> element that will make it easier to understand what’s going on.  For more details, check out the HTML5 reference.  The <canvas> element is used to draw graphics, on the fly, on a web page.  It basically gives us a drawing surface inside the web page.  Every canvas has a **drawing context object**.  That is where the drawing methods and properties are defined. To access them, we first call getContext("2d") to get the context object.  We can then invoke the various drawing methods on the context object.  In our example, we'll invoke the fillRect() method on the context object to draw a rectangle at a given position on the canvas.  The **origin of the coordinate system, for the canvas as well as the web page is at the upper left corner**, with the x coordinate increasing toward the right, and the y coordinate increasing toward the bottom:  The **base unit is the pixel**, with the top left pixel having coordinates (0,0).  We’ll first create a source document, draw.html that defines a canvas element as follows:  draw.html  <!DOCTYPE html>  <html>      <head>              <meta charset="utf-8">              <title>Let's Draw</title>      </head>      <body>              <h2> Just click inside the box </h2>  **<canvas id="myCanvas" width="300" height="300" style="border:1px solid #c3c3c3;">**  **</canvas>**              <script defer src="../scripts/draw.js"></script>     </body>  </html>  Note that the height and width attributes specify the size of the canvas.  We added a border so that we can see where we are supposed to draw.  In the next module we'll see how to specify these attributes in an external style sheet.  We would like to **capture the user clicks** on the canvas as events in JavaScript and draw a small red square at the position of each click.  In order to do that, we have to somehow have access to the position of the click event.  The event object has that information.  **The** **event object clientX property contains the horizontal coordinate** within the application's client area at which the event occurred.  To obtain the x coordinate with respect to the canvas we’ll have to adjust for the canvas leftOffset.  Similarly **the event object clientY property contains the vertical coordinate** within the application's client area at which the event occurred.  To obtain the y coordinate with respect to the canvas we’ll have to adjust for the canvas topOffset.  Here’s our program:  draw.js  function drawSquare(event) {      // Our function takes the event object as a parameter      var myCanvas = document.getElementById('myCanvas');      // compute the coordinates with respect to the canvas      var x = event.clientX - myCanvas.offsetLeft;      var y = event.clientY - myCanvas.offsetTop;      // to access the canvas, we need its context      var myContext = myCanvas.getContext('2d');      // set the color to red      myContext.fillStyle = '#FF0000';      // draw a 10 by 10 square starting at the click event position      myContext.fillRect(x, y, 10, 10);  }  document.getElementById('myCanvas') .addEventListener('click', drawSquare, false);  Note that when we specify drawsquare as an argument to the addEventListener method, we don't specify event as an argument:  we only specify the function name:  ...addEventListener('click', **drawSquare**, false);  However in the function definition corresponding to drawsquare, we specify event as a parameter:  function drawSquare(**event**) { ...  Initially, the following is displayed:  Then every time we click (or on a touch screen, touch) inside the box, a little red square is added at the location we clicked (or touched).  10.4. Event Bubbling  *Copyright (c) 2014, Rula Khayrallah*  So far we have encountered simple examples where the event listener is registered and invoked directly on the target that triggers it.  When the first input was modified in our simple calculator example, the event registered on that first input element was invoked.  However in the general case, **most** **events bubble up the DOM tree**.  So when an event is triggered on an element, it is also triggered on its ancestors in the DOM.  The event listener(s) registered on the target element (if any) are invoked first but then the listeners registered on the element’s parent are invoked and then the listeners registered on the element’s grandparent are invoked.   And so on up the DOM tree…  Event bubbling allows us to **register a single listener on a common ancestor** element and handle events there.  So going back to our simple calculator example, in our add.js script, instead of registering the update listener on both target inputs, we could have registered it once on a common ancestor, say body.  So instead of:  document.getElementById('first').addEventListener('input', update, false);  document.getElementById('second').addEventListener('input', update, false);  We could have written:  document.**body**.addEventListener('input', update, false);  Note that in this case, the listener function can still figure out the specific descendant element that the event was triggered on by using the **target property of the event object** passed to the function. The element that the function is currently invoked on is accessible via both the **currentTarget property of the event object** as well as via '**this'.** Let’s go back to our update function and add some logging statements to illustrate that:  function update(**event**) {    **console.log('this:', this.nodeName);**  **console.log('event.target:', event.target.id);**  **console.log('event.currentTarget: ', event.currentTarget.nodeName);**        // Get the two input numbers      var firstNumber = Number(document.getElementById('first').value);      var secondNumber = Number(document.getElementById('second').value);      // Then  compute the sum      var myAnswer = firstNumber + secondNumber;      // And write it in the 'answer' element if it is a valid answer      document.getElementById('answer').textContent = myAnswer;  }     When the event is triggered by entering a number in the first input, we get:  this: BODY  event.target: first  event.currentTarget: BODY  And when the event is triggered by entering a number in the second input, we get:  this: BODY  event.target: second  event.currentTarget: BODY  The listener may also stop further bubbling by invoking the **stopPropagation()** method on the event object:  event.stopPropagation()  Finally,  we said **most events bubble**.  There are some events such as the focus, blur and scroll events that don’t.  In that case, the event listener has to be registered directly on the element where the event will be triggered.  10.5. Event Capturing  *Copyright (c) 2014, Rula Khayrallah*  Event capturing is an event propagation method that goes in the opposite order as event bubbling.  Remember the third argument to the addEventListener() method?  We have been setting it to false in our examples.  It is actually a 'useCapture' parameter.  When set to true, events are propagated down the DOM tree, from parent to child.  Let’s say an event is triggered on a target element.  The event listeners registered on the highest ancestor of that target element are invoked first and then the listeners registered on the target element second highest ancestor and so on down the DOM tree…  Let’s illustrate the difference between the two propagation methods with an example:  <!DOCTYPE HTML>  <html>         <head>                <meta charset="utf-8">                <title>JavaScript for Programmers</title>         </head>         <body>                <div id = "grandParent">                       <div id = "parent">                               <p>Just enter anything below to see event propagation in action</p>                               <input id = "myInput" type = "text">                       </div>                </div>                <script defer src="../scripts/propagate.js"></script>         </body>  </html>  We’ll write some event listener functions that simply include some logging statements to identify the function that was invoked and the element it was invoked on.  We’ll first register all these functions with the useCapture parameter set to false.  propagate.js  function grandParentHandler(event) {      console.log('Grandparent handler');      console.log('this:', this.id);      console.log('event.target:', event.target.id);  };  function parentHandler(event) {      console.log('Parent handler');      console.log('this:', this.id);      console.log('event.target:', event.target.id);  };  function elementHandler(event) {      console.log('Event handler');      console.log('this:', this.id);      console.log('event.target:', event.target.id);  };  document.getElementById('grandParent') .addEventListener('input', grandParentHandler, false);  document.getElementById('parent') .addEventListener('input', parentHandler, false);  document.getElementById('myInput') .addEventListener('input', elementHandler, false);  With the useCapture parameter set to false, events bubble up:  the target event handler in invoked first, then the parent’s handler and then the grandparent’s handler.  Event handler  this: myInput  event.target: myInput  Parent handler  this: parent  event.target: myInput  Grandparent handler  this: grandParent  event.target: myInput    Let’s see what happens when the useCapture parameter is set to true.  First we change the last 3 lines in propagate.js to:  document.getElementById('grandParent') .addEventListener('input', grandParentHandler,**true**);  document.getElementById('parent') .addEventListener('input', parentHandler, **true**);  document.getElementById('myInput') .addEventListener('input', elementHandler, **true**);    And the invocation order is reversed as we can see in our console output:  Grandparent handler  this: grandParent  event.target: myInput  Parent handler  this: parent  event.target: myInput  Event handler  this: myInput  event.target: myInput  Note that bubbling is much more common than capturing.  10.6. Anonymous Function as the Event Listener  *Copyright (c) 2014, Rula Khayrallah*  It is very common in JavaScript to use an anonymous function expression to specify the event listener.  Let's go back to our calculator example, and our add.js program:  function update() {        // Get the two input numbers      var firstNumber = Number(document.getElementById('first').value);      var secondNumber = Number(document.getElementById('second').value);        // Then  compute the sum      var myAnswer = firstNumber + secondNumber;        // And write it in the 'answer' element      document.getElementById('answer').textContent = myAnswer;  }  document.body.addEventListener('input', **update**, false);  Since we are registering the event listener only once on the body element we don't have to define the function separately.  We can just define it in an anonymous function expression as follows:    document.body.addEventListener('input', **function () {**  **// Get the two input numbers**  **var firstNumber = Number(document.getElementById('first').value);**  **var secondNumber = Number(document.getElementById('second').value);**  **// Then  compute the sum**  **var myAnswer = firstNumber + secondNumber;**  **// And write it in the 'answer' element**  **document.getElementById('answer').textContent = myAnswer;**  **}**, false);  10.7. Event Listeners and 'this'  *Copyright (c) 2014, Rula Khayrallah*  We have seen that inside the event listener, the event target is accessible via both the currentTarget property of the event object as well as via 'this'.    As a result, when the event listener is a method that is invoked on an object, the value of 'this' inside the method no longer points to the object.    Consider the following example:    car.drive = function () {       this.mileage = this.mileage + 10;       return this;    };    Now suppose that we want to invoke the drive method on the object myCar whenever the user clicks on an 'action' button. myCar inherits the drive method from the prototype car. However the following will NOT work because the value of 'this' passed to the drive method is NOT the object myCar.    document.getElementById('action').addEventListener('click', **myCar.drive**, false);  // this will not work.    As an alternative, we can define an intermediary function that will in turn invoke the drive method with the correct context:    function driveFunction () {       myCar.drive();    }    document.getElementById('action').addEventListener('click', **driveFunction**, false);    This can also be achieved with an anonymous function expression:    document.getElementById('action') .addEventListener('click',   **function () { myCar.drive();}**, false);    10.8. Load Events  *Copyright (c) 2014, Rula Khayrallah*  In the previous sections, we have seen events that are triggered by the user: user input, clicks and so on.  There are also events that are independent of the user:  the DOMContentLoaded event and the load event are two such events.  The**DOMContentLoaded** event is triggered **on the document object**when the source document has been parsed and **the DOM is ready** to be manipulated.  Note that the images, stylesheets and any other associated resources may not have been loaded yet.  The**load event is triggered when a resource and all its dependent resources have finished loading**.  It is triggered on the window object when the web page, the images, stylesheets and any other resources included in the source document have finished loading.  While older programs used to check for the window load event before starting to execute, in modern browsers there is usually no need to wait for all resources to load before executing our JavaScript code.  Note that event listeners may be added for these events just like any other user events:  window.addEventListener('load', function () { console.log ('Loading is complete!');}, false);  document.addEventListener('DOMContentLoaded', function () { console.log ('DOM is ready!');}, false); |  |