SNS COLLEGE OF ENGINEERING

An Autonomous Institution

Coimbatore-107

19TS601-FULL STACK DEVELOPMENT

UNIT-1

JAVASCRIPT AND BASICS OF MERN STACK

UI Events -Forms, controls

UI Events-Mouse Events

- Mouse events belong to the most common and significant event types.
- Mouse event object can be defined as a unity of events that happen when the mouse interacts with the HTML document.
- The Types of Mouse Events
  - Simple
  - Complex

UI Events-Mouse Events

Simple events

- mousedown/mouseup: the mouse button is clicked or released over an element.
- mouseover/mouseout: the pointer of the mouse comes over or out of an element.
- mousemove: each move of the mouse triggers the event.
- contextmenu: it triggers when an attempt of opening a context menu is detected. Commonly, it happens when the user presses the right button of the mouse.However, there are other means of opening the context menu (for instance, using a specific keyboard key but it doesn't belong to mouse events).

UI Events-Mouse Events

- Complex events
- The complex events are the following:
- click: it is activated after mousedown and then mouseup over the same element when the left button of the mouse is used.
- dblclick: it triggers when a double click over an element is performed.
- The basis of the complex events are simple ones.

UI Events-Event order

- Multiple events can be triggered by an action. For example, an initial click triggers mousedown at the moment the button is pressed. Afterward, comes the mouseup and click when it is released.
- In the cases when multiple events are initiated by a single action, their order is fixed. It means that the handlers are called in the following sequence: mousedown → mouseup → click .

Getting the button: which

- The click-related events always contain the which property, allowing to get a particular mouse button. You needn’t use it for the events related to click and contextmenu .
- But, in the event of tracking mousedown and mouseup ,it will be necessary. The reason is that such events trigger on any button. So, which will allow distinguishing between “right-mousedown” and “left-mousedown” .

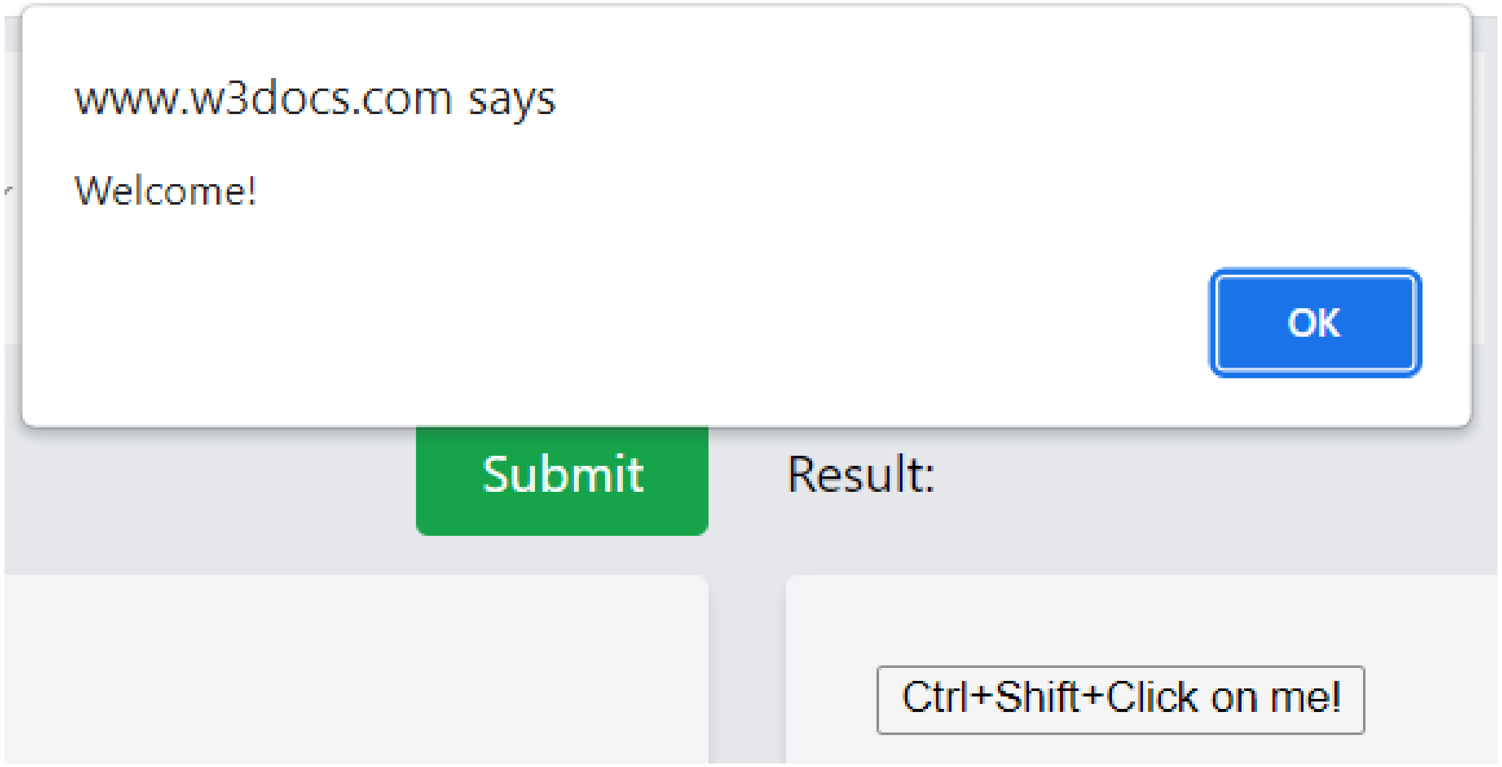
Getting the button: which

- The possible three values are as follows:
  - The left button - event.which == 1 .
  - The middle button- event.which == 2 .
  - The right button - event.which == 3 .
- However, the middle button is rarely used.

Modifiers: shift, alt, ctrl and meta

- All the mouse events have information about pressed modifier keys.
- The properties of the events are:
  - shiftKey: Shift
  - altKey: Alt (or Opt for Mac)
  - ctrlKey: Ctrl
  - metaKey:for Mac - Cmd
- When the matching key is pressed during the event, the properties above are true.
- Let's take an example, where the button works on Ctrl+Shift+click:

```
<html>
 <head>
  <title>Title of the Document</title>
 </head>
 <body>
  <button id="button">Ctrl+Shift+Click on me!</button>
  <script>
   button.onclick = function(event) {
    if(event.ctrlKey && event.shiftKey) {
     alert('Welcome!');
    }
   };
  </script>
 </body>
</html>
```



www.w3docs.com says

Welcome!

- consider that for Mac, Cmd is used instead of Ctrl.On Mac, there is another Cmd that corresponds to the property metaKey.
- So, in places where Windows user presses Ctrl+Enter or Ctrl+A, a user of MAC should press Cmd+Enter or Cmd+A.
- So, anytime it is necessary to support combinations, such as Ctrl+click, for MAC it makes sense using Cmd+click.

Coordinates: clientX/Y, pageX/Y

- The overall mouse events have coordinates in the following two flavours:
  - clientX and clientY that are window-relative.
  - pageX and pageY that are document-relative.
- Let's say you have a window with the size 500x500.
- The mouse is in the left-upper corner. In this case, clientX and clientY are equal to 0.
- When the mouse is in the center, clientX and clientY are equal to 250, no matter how far the document has been scrolled. It is like position:fixed .

- The document related coordinates, such as pageX , pageY should be counted from the left-upper corner of the document.
- Other examples include “Disabling selection” and “prevent copying”

Keyboard Event-keydown and keyup

- A keyboard event is generated by pressing a key: no matter it's a symbol key or a special key, such as Shift, Ctrl, and more.
- Other ways of inputting on modern devices. For instance, speech recognition or copy/paste using the mouse.

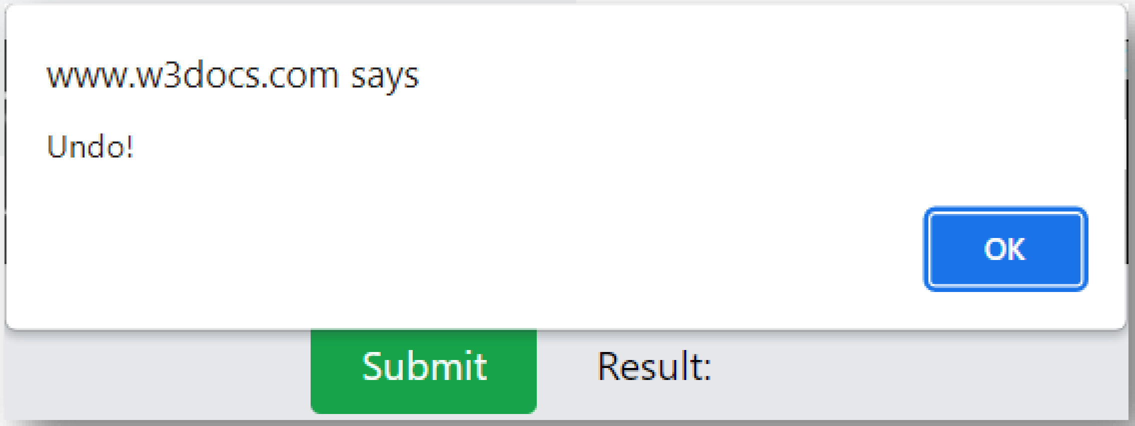
- The keydown and keyup events occur whenever the user presses a key.
- The keydown happens at the moment the user presses the key down.
- It repeats as long as the user keeps it down.
- The keyup event happens when the user releases the key after the default action has been performed.

Key board event-Event.code and Event.key

- The key property of the event object allows getting the character, while the code property - the “physical key code”.
- For instance, the same “S” can be pressed both with the Shift button or without. It will give different characters: lowercase z and uppercase Z.
- The event.key is the character itself: hence, it will be different.

- In case the user works with different languages, then switching to another language will bring a completely different character instead of “S”.
- It will be the value of the event.key, while event.code is constantly the same.

```
<html>
 <head>
  <title>Title of the Document</title>
 </head>
 <body>
  <script>
   document.addEventListener('keydown', function(event) {
    if(event.code == 'KeyZ' && (event.ctrlKey || event.metaKey)) {
     alert('Undo!')
    }
   })
  </script>
 </body>
</html>
```



```
www.w3docs.com says
Undo!
```

Key board event:Auto-repeat

- When any key is being pressed for a long enough time, it gets “auto-repeating”.
- The keydown event happens again and again, and when it is released, the keyup is got.
- So, it is typical to have many keydowns and only one keyup.
- For events that happen by auto-repeat, the event.repeat property is set to true.

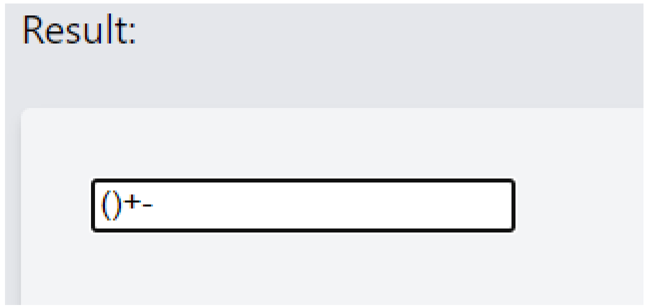
Key board event:Default Actions

- There can be different default actions. It’s because many possible things can be initiated by the keyboard, as described below:
  - A character comes out on the screen.
  - A character is deleted with the Delete key.
  - The scrolling of the page with the PageDown key.
  - Opening of the “Save Page” dialog by Ctrl+S.

- If you prevent the default action on keydown, it may cancel most of them, except for the OS-based specific keys.
- For instance, Alt+F4 closes the current browser window on Windows. And, in JavaScript, no way exists to stop it by preventing the default action.

- Let’s check out an example where the <input> expects a phone number, not accepting keys except digits, such as +, - or ():

```
<html>
 <head>
  <title>Title of the Document</title>
 </head>
 <body>
  <script>
   function checkedPhoneKey(key) {
    return(key >= '0' && key <= '9') || key == '+' || key == '(' || key == ')' || key == '-';
   }
  </script>
  <input onkeydown="return checkedPhoneKey(event.key)" placeholder="Phone number"
type="tel">
 </body>
</html>
```


Result:

```
()+-
```

Also, take into account that special keys, like Backspace, Left, Right, Ctrl+V will not work in such an output. That is the side-effect of the checkedPhoneKey strict filter.

Key board event: Legacy

- Previously, there was a keypress event, along with keyCode, charCode, which properties of the event object.
- Anyway, they caused so many incompatibilities that developers started deprecating all of them and making new, modern events (described above). Of course, the old code still works but there is no use in them.

Summary

- The primary keyboard event properties are the following:
  - code: it is the key code (for example, "KeyA"), specific to the key location on the keyboard.
  - key: the character ("A", "a") for non-character keys. Normally, its value is equivalent to the value of the code.

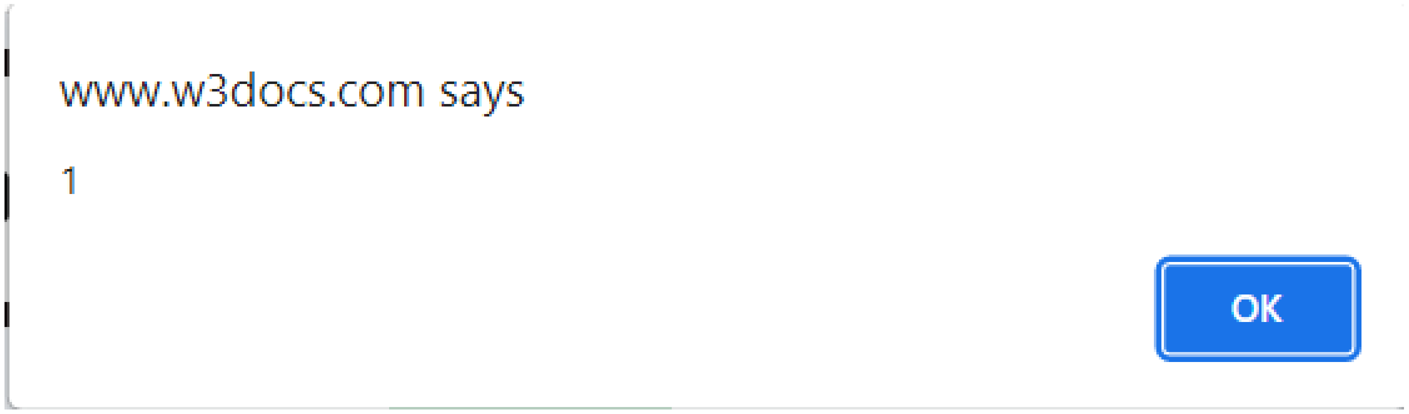
Forms, controls :JavaScript Form Properties and Methods

- The method property is targeted at setting or returning the value of the method attribute in a form.
- The method attribute indicates the way of sending form-data. The latter is sent to the page, specified in the action attribute.

Navigation: Form and Elements

- Document forms are the components of the specific collection, known as document.forms. It is a so-called “named collection”, both named and ordered. For getting the form both the name and the number can be used, like here:
  - document.forms.myForm - the form with name="myForm"
  - document.forms[0] - it's the first form in the document
- While having a form, any element is available in the named collection form.elements, like in this example:

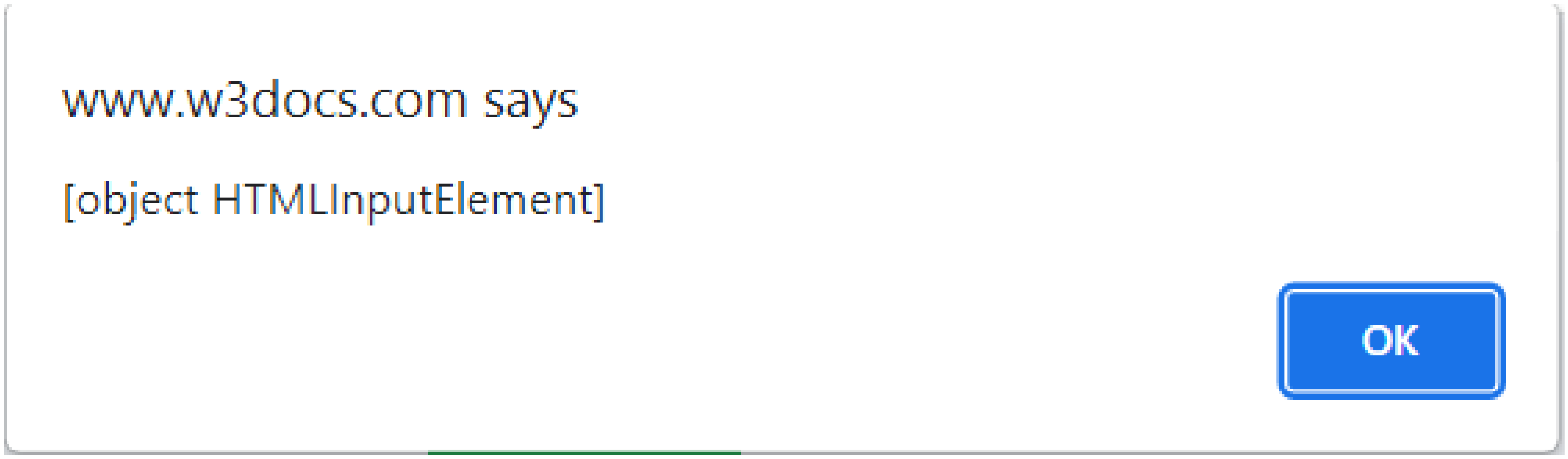
```
<html>
  <head>
    <title>Title of the Document</title>
  </head>
  <body>
    <form name="myForm">
      <input name="firstName" value="1">
      <input name="lastName" value="2">
    </form>
    <script>
     // get the form
     let form = document.forms.myForm; // <form name="myForm"> element
     // get the element
     let elem = form.elements.firstName; // <input name="firstName"> element
     alert(elem.value); // 1
    </script>
  </body>
</html>
```


www.w3docs.com says

```
1
```

- Multiple elements can exist with the same name:
- for example, the case with radio buttons. In such a case, form.elements[name] is a collection, for example:

```
<html>
 <head>
  <title>Title of the Document</title>
 </head>
 <body>
  <form>
   <input type="radio" name="age" value="1">Value1</input>
   <input type="radio" name="age" value="2">Value2</input>
  </form>
  <script>
   let form = document.forms[0];
   let ageElems = form.elements.age;
   alert(ageElems[0]); // [object HTMLInputElement]
  </script>
 </body>
</html>
```



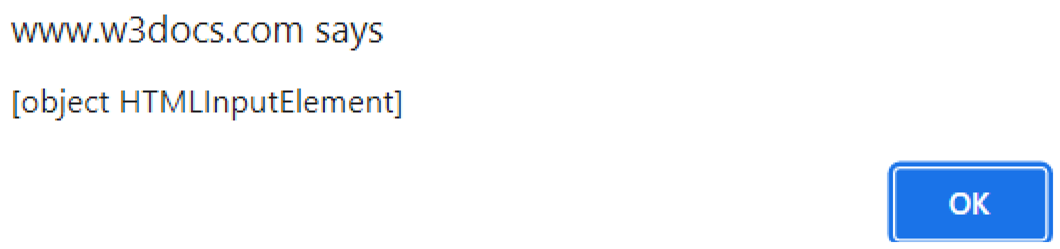
```
www.w3docs.com says

[object HTMLInputElement]
```

- The navigation properties don't rely upon the tag structure.
- All the control elements are available inside form.elements.

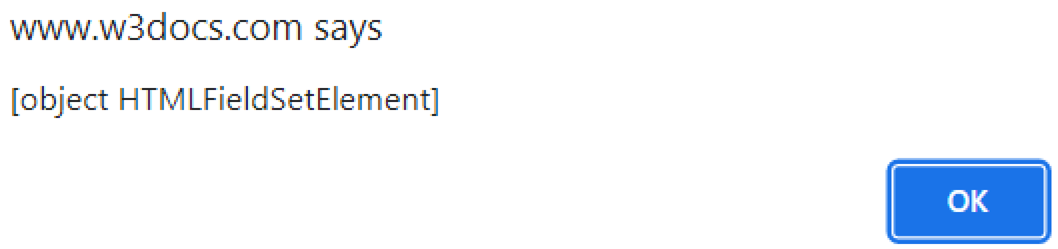
Form,Controls: Fieldsets as “Subforms

- A form can contain one or many <fieldset> elements.
- Also, they have elements property, which is controlled by the lists form inside them. Here is an example:

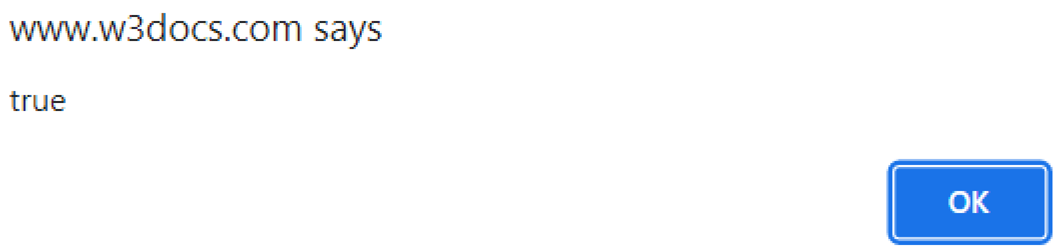
```
<html>
 <head>
  <title>Title of the Document</title>
 </head>
 <body>
  <form id="formId">
   <fieldset name="infoFields">
    <legend>info</legend>
    <input name="login" type="text">
   </fieldset>
  </form>
  <script>
   alert(formId.elements.login); // <input name="login">
   let fieldset = formId.elements.infoFields;
   alert(fieldset); // HTMLFieldSetElement
   // we can get input by name both from the form and from a set of fields
   alert(fieldset.elements.login == formId.elements.login); // true
  </script>
 </body>
</html>
```


www.w3docs.com says

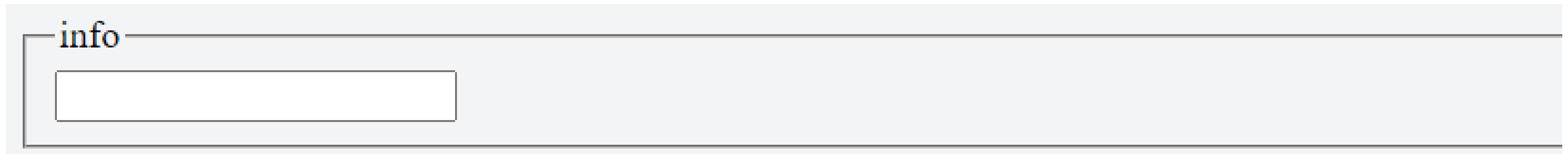
[object HTMLInputElement]


www.w3docs.com says

[object HTMLFieldSetElement]

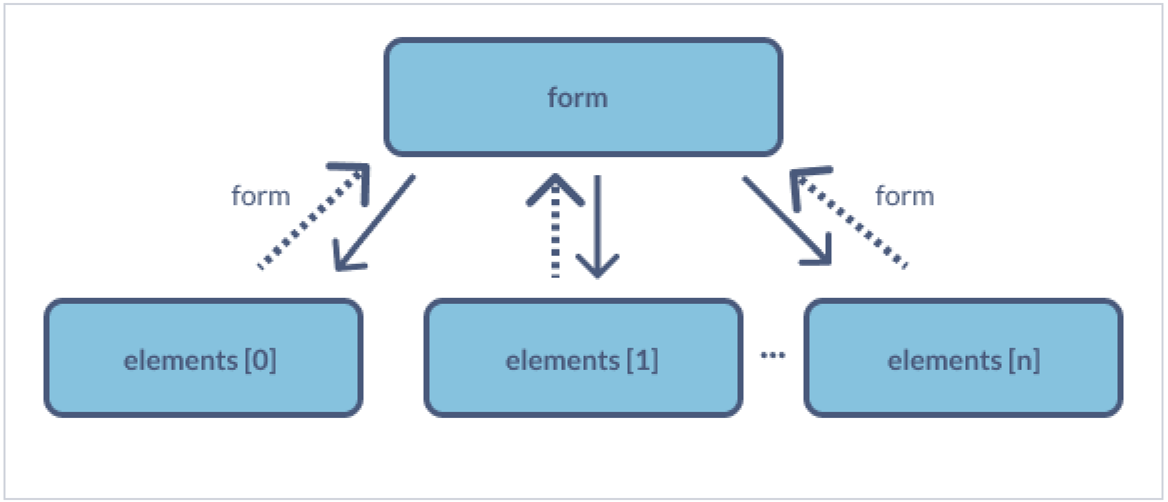

www.w3docs.com says

true

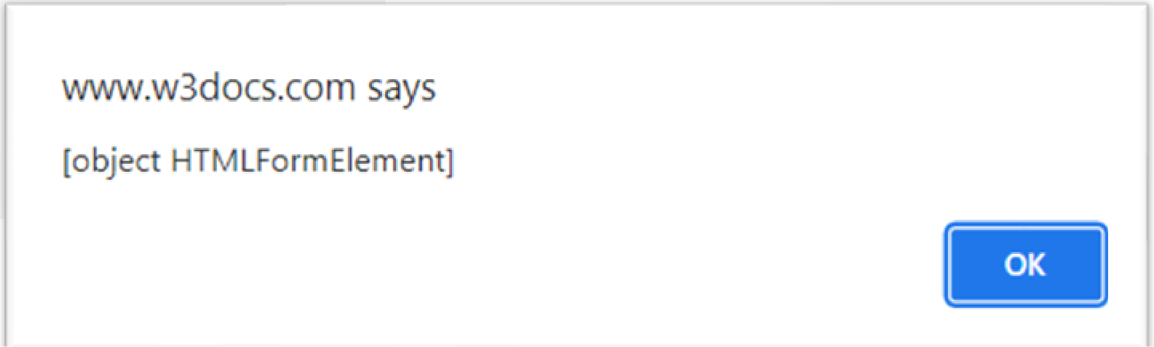

info

Backreference: element.form

- The form is available as element.form for any element. So, elements reference the form and the form references all elements, as demonstrated in the picture below:
- And, here is an example:



```
<html>
 <head>
  <title>Title of the Document</title>
 </head>
 <body>
  <form id="form">
   <input type="text" name="login">
  </form>
  <script>
   // form -> element
   let login = form.login;
   // element -> form
   alert(login.form); // HTMLFormElement
  </script>
 </body>
</html>
```



www.w3docs.com says

[object HTMLFormElement]

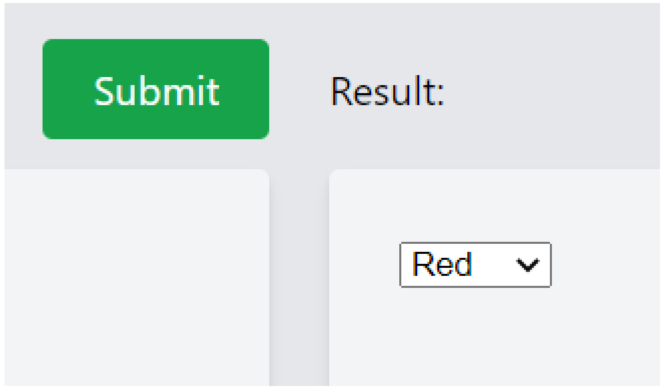
Form Elements

- input and text area
  - Their value can be accessed as input.value (string) or input.checked (boolean) for checkboxes, as shown in the example below:
  - input.value = "The Value";
  - textarea.value = "The text"
  - input.checked = true; // checkbox or radio button
- Also, take into consideration that although <textarea>...</textarea> keeps its value as nested HTML, textarea.innerHTML should never be used for accessing it.
- It includes only the HTML that was initially on the page and not the current value.

- select and option
- A <select> element has three significant properties:
  - The group of <option> subelements is known as select.options.
  - The value of the currently selected <option> is select.value.
  - The number of the currently selected <option> .

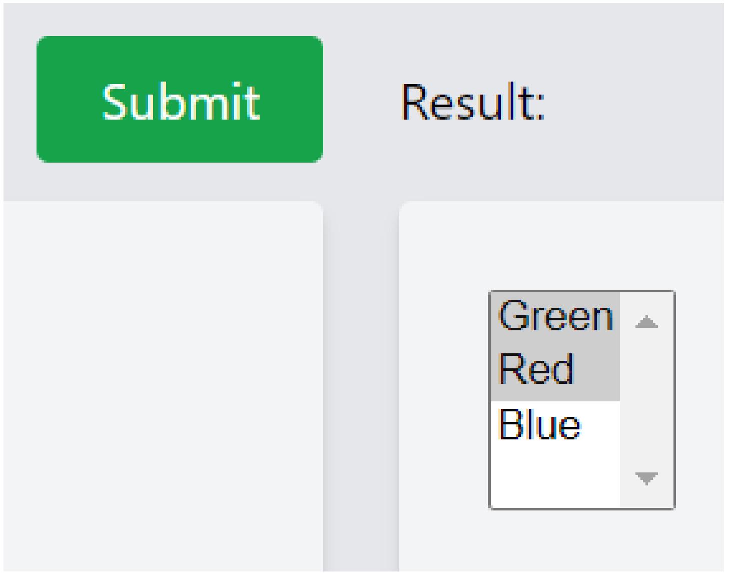
- Accordingly, they offer three different ways of setting a value for the <select>:
  - Finding the matching <option> element and set option.selected to true.
  - Setting select.value to the value.
  - Setting select.selectedIndex to the number of the option.
- The 2nd and the 3rd ways are more convenient, but the first is the most obvious one.
- Here is an example:

```
<html>
  <head>
    <title>Title of the Document</title>
  </head>
  <body>
    <select id="select">
      <option value="green">Green</option>
      <option value="red">Red</option>
      <option value="blue">Blue</option>
    </select>
    <script>
     // all three lines do the same thing
     select.options[2].selected = true;
     select.selectedIndex = 2;
     select.value = 'red';
    </script>
  </body>
</html>
```



- In contrast to most other controls, <select> helps to control multiple options simultaneously, if it has the attribute multiple. However, this feature is rarely used. Hence, you can use the first way that is add/remove the selected property <option> from subelements.
- In the example, we demonstrate how to get their collection as select.options:

```
<html>
  <head>
    <title>Title of the Document</title>
  </head>
  <body>
    <select id="select" multiple>
      <option value="green" selected>Green</option>
      <option value="red" selected>Red</option>
      <option value="blue">Blue</option>
    </select>
    <script>
      // get all selected values from multiple selection
      let selected = Array.from(select.options).filter(option => option.selected).map(option =>
option.value);
      alert(selected); // green, red
    </script>
  </body>
</html>
```


Submit
Result:
Green
Red
Blue

- new option
- This element is also used rarely on its own. But in its specification there is a pretty short syntax for creating <option> elements, like this:

- let option = new Option(text, value, defaultSelected, selected);

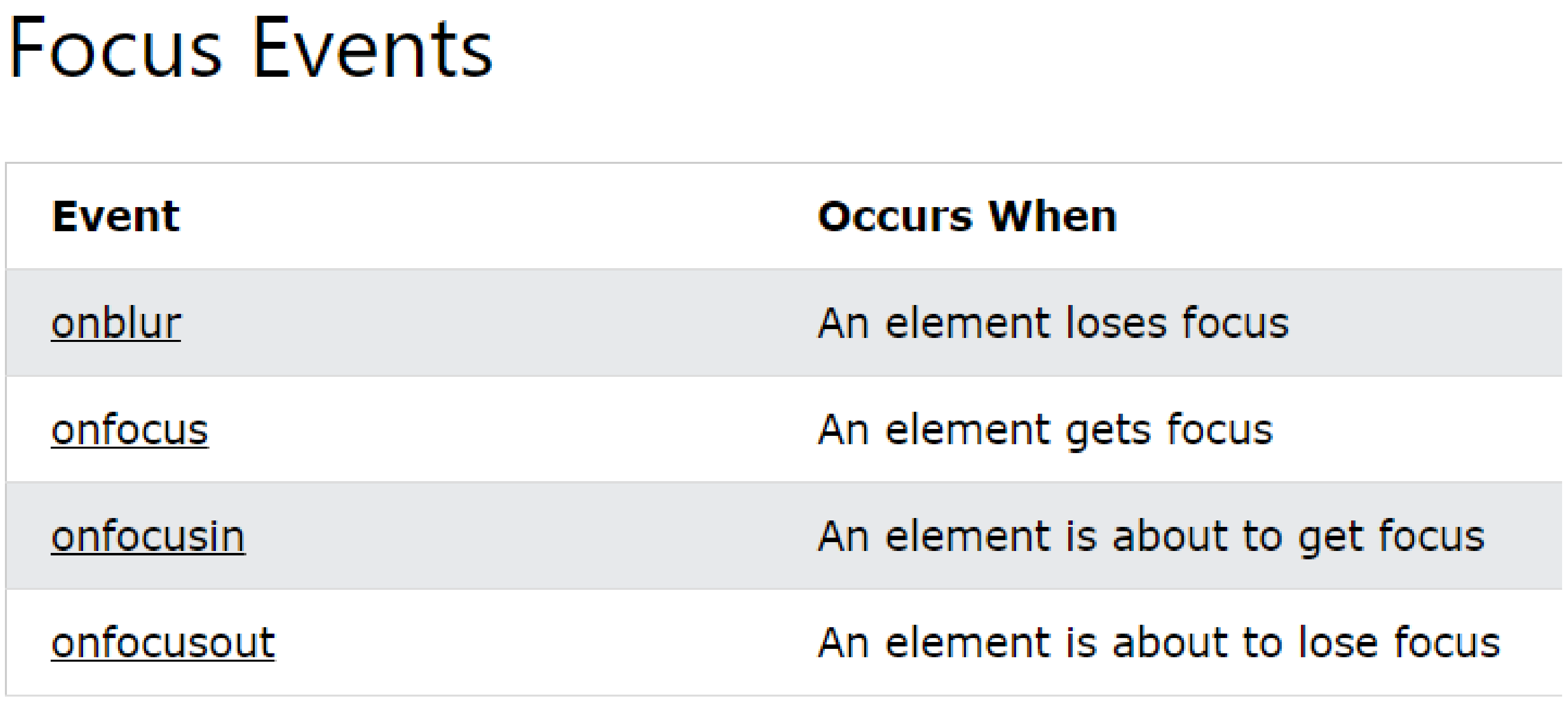
- The parameters are the following:
  - text is the text within the option,
  - value is the value of the option,
  - defaultSelected – if it's true, then selected HTML-attribute is generated,
  - selected – if it's true, then the option is picked out.

- Sometimes there is a confusion over defaultSelected and selected. The difference between them is the following: defaultSelecte specifies the HTML-attribute, which you can get using option.getAttribute('selected'), and the selected sets whether the is selected or not. It is more important. In principle, both values can be set to true or false. For example:

- let option = new Option('text', 'value');
- // creates <option value="value"> text </option>
- In the example below, the same element is selected:
- let option = new Option('text', 'value', true, true);
- Option elements include the following properties:
  - option.selected - the option is selected.
  - option.index - the number of the option amid the others in its kbd class="highlighted"><select>.
  - option.text -the option’s text content.

JavaScript Focusing: focus/blur

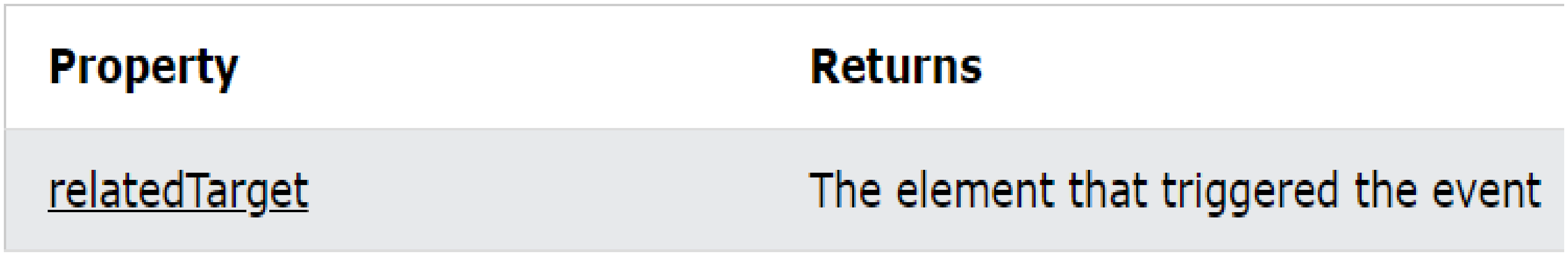
- The FocusEvent Object handles events that occur when elements gets or loses focus.



Focus Events

Event	Occurs When
onblur	An element loses focus
onfocus	An element gets focus
onfocusin	An element is about to get focus
onfocusout	An element is about to lose focus

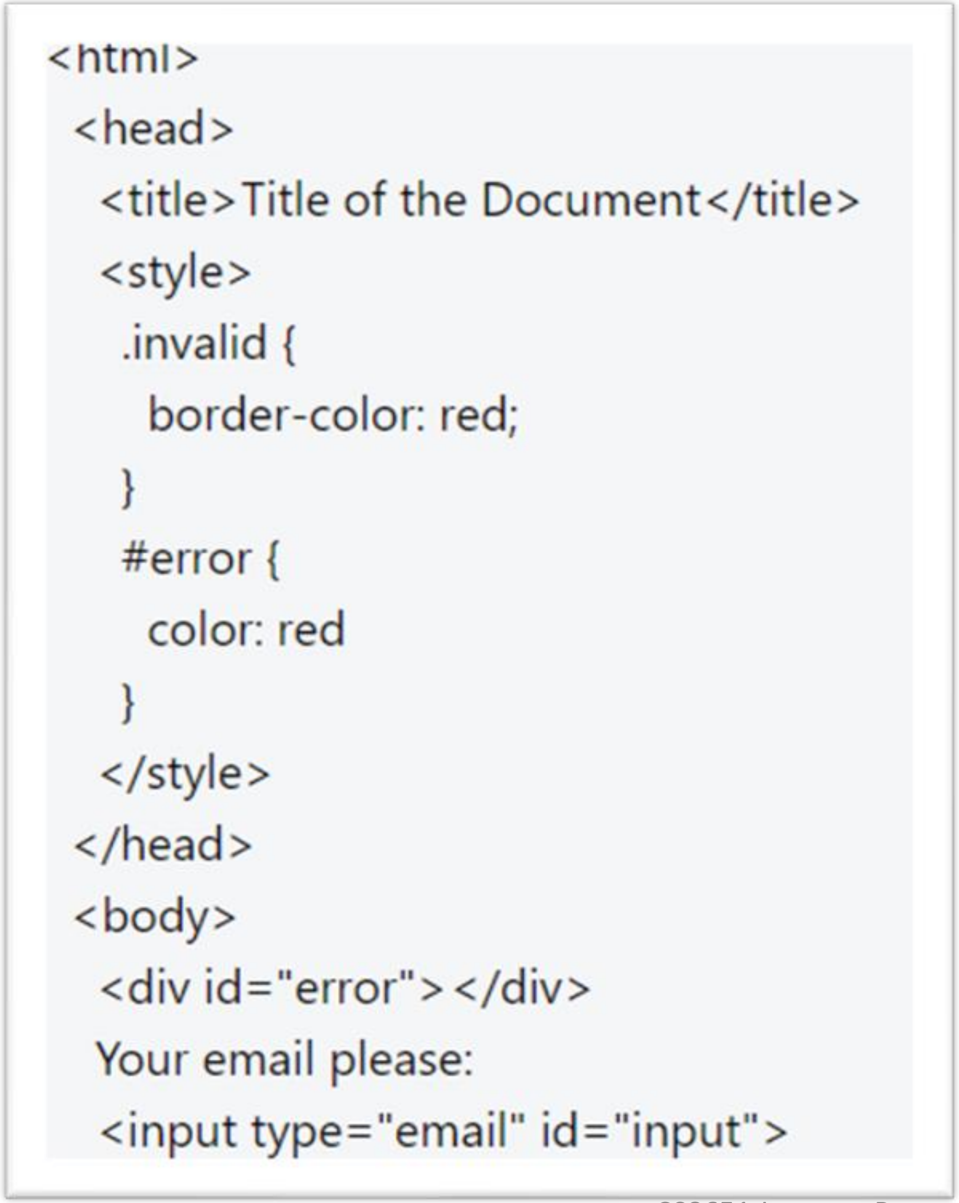
FocusEvent Properties



Property	Returns
relatedTarget	The element that triggered the event

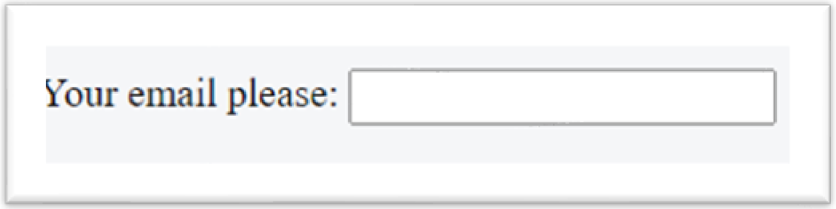
Events:focus/blur

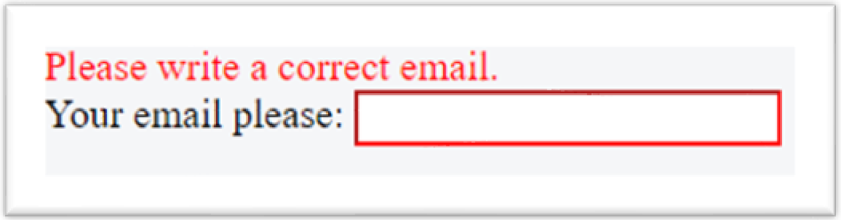
- On focusing, the focus event is called, and when the element loses the focus, the blur event is called.
- To be more accurate, let’s apply them for validation of an input field.
  – The blur handler checks if the email is entered. If it’s not entered- an error occurs.
  – The focus handler hides the error message:

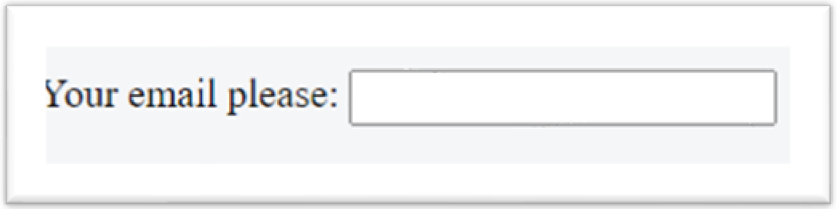


```
<html>
 <head>
  <title>Title of the Document</title>
  <style>
   .invalid {
     border-color: red;
   }
   #error {
     color: red
   }
  </style>
 </head>
 <body>
  <div id="error"></div>
  Your email please:
  <input type="email" id="input">
```

```
    <script>
     input.onblur = function() {
       if(!input.value.includes('@')) { // not email
         input.classList.add('invalid');
         error.innerHTML = 'Please write a correct email.'
       }
     };
     input.onfocus = function() {
       if(this.classList.contains('invalid')) {
         // remove the "error" indication, if the user wants to re-enter something
         this.classList.remove('invalid');
         error.innerHTML = "";
       }
     };
    </script>
  </body>
</html>
```


Your email please:

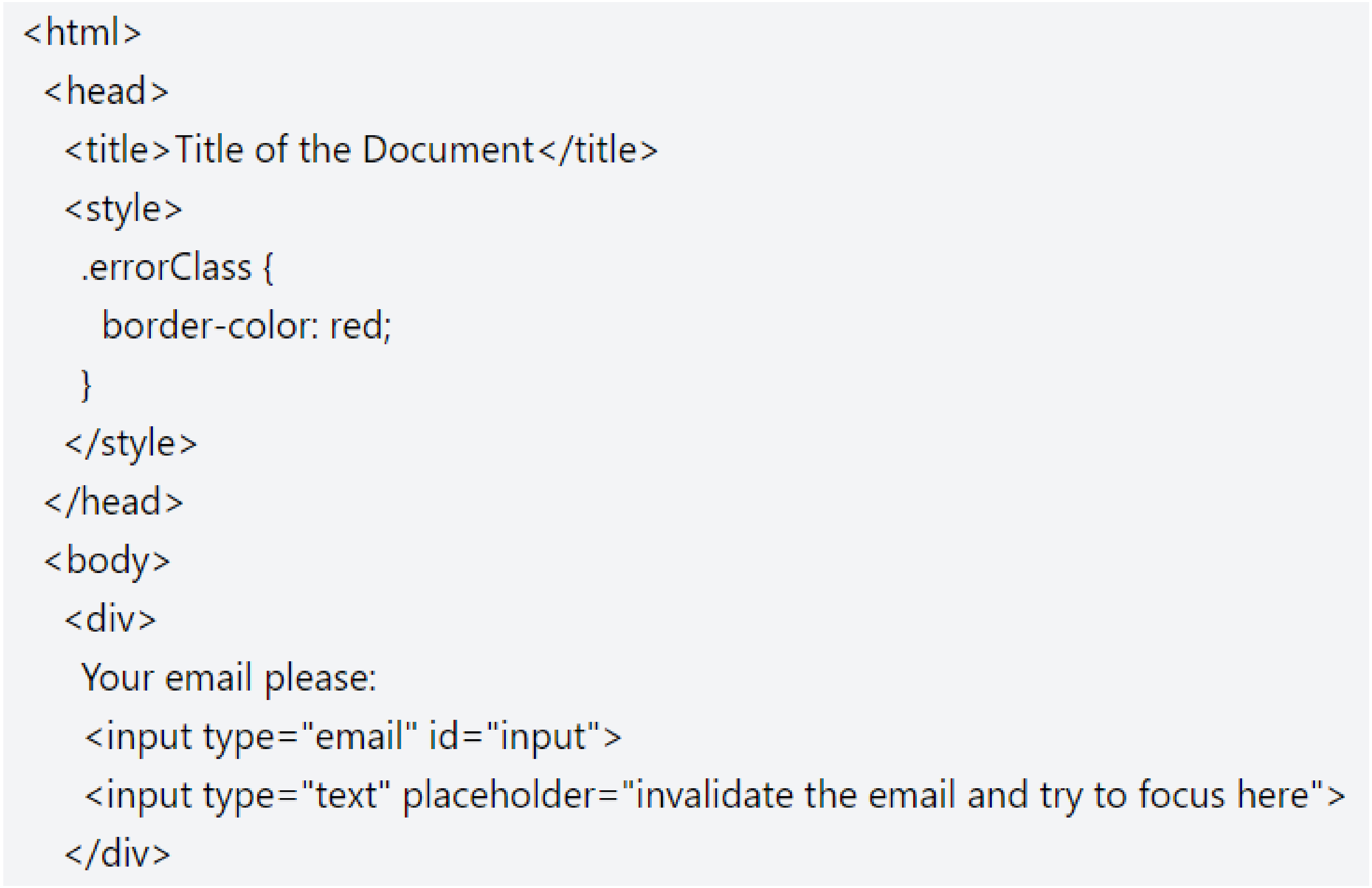

Please write a correct email.
Your email please:


Your email please:

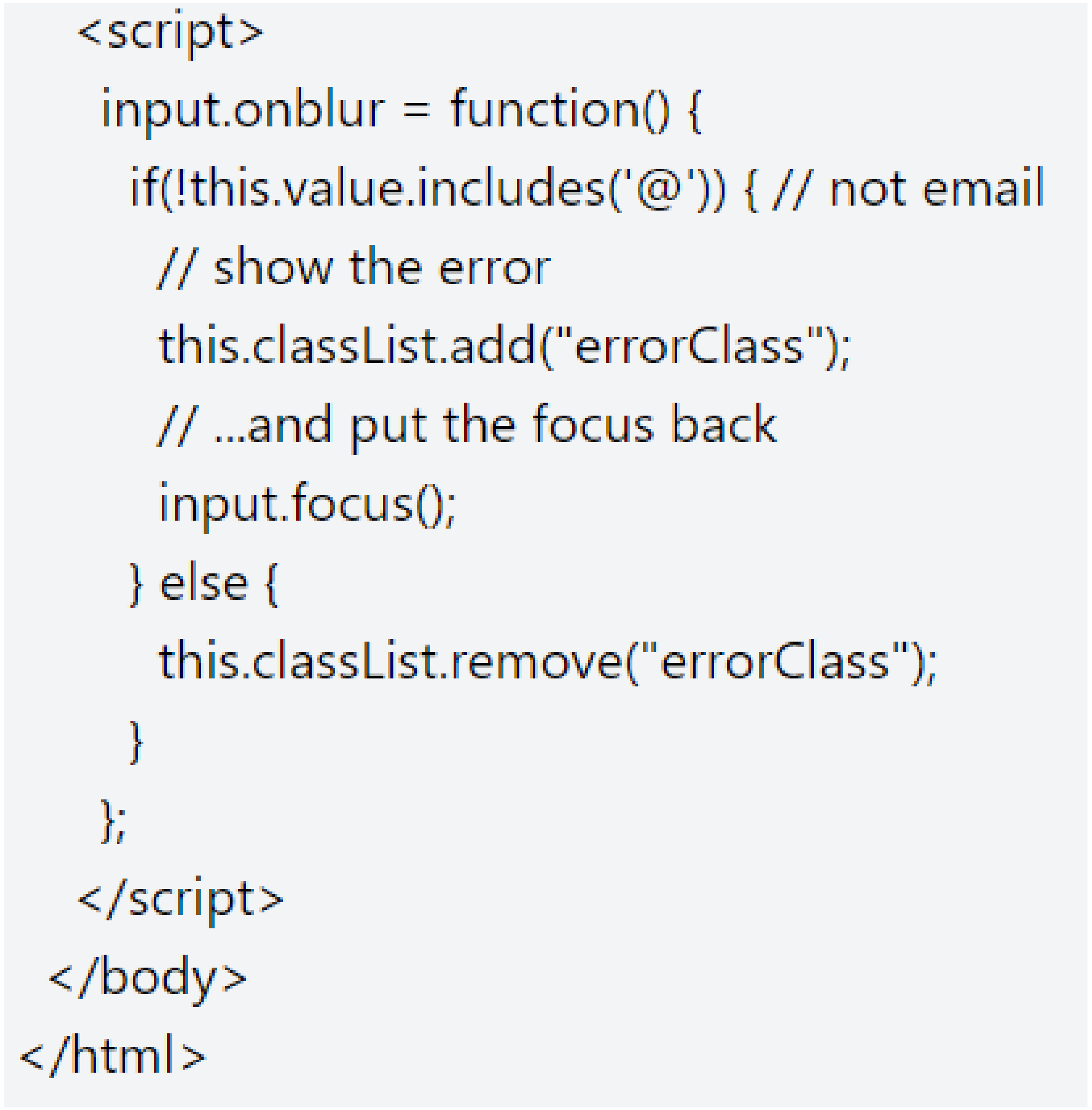
- Modern HTML is capable of doing much validation, applying input attributes: required, pattern, and more.
- JavaScript may be used for getting more flexibility. Also, the changed value can be sent to the server, in case it’s correct.

Methods focus/blur

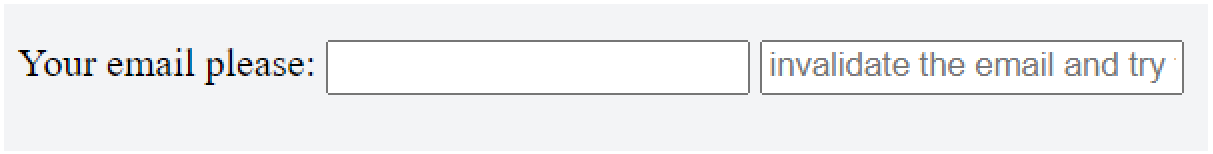
- The elem.focus() and elem.blur() methods are used for setting/unsetting the focus on the element.
- Let’s make the user unable to leave the input, if the value is invalid, like this:



```
<html>
 <head>
  <title>Title of the Document</title>
  <style>
   .errorClass {
     border-color: red;
   }
  </style>
 </head>
 <body>
  <div>
   Your email please:
   <input type="email" id="input">
   <input type="text" placeholder="invalidate the email and try to focus here">
  </div>
```

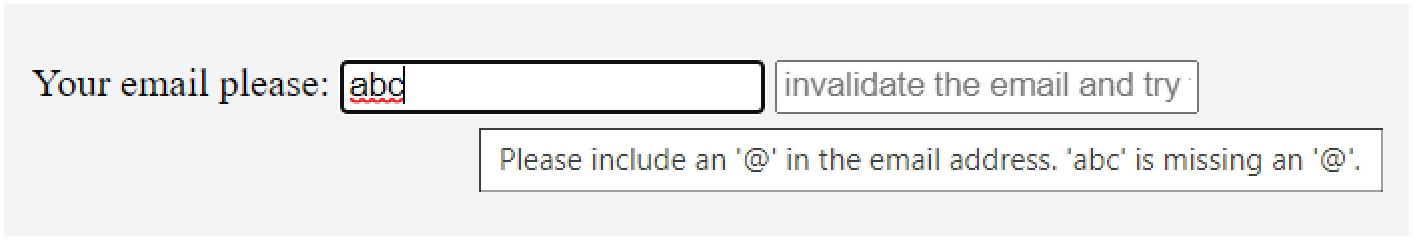


```
    <script>
     input.onblur = function() {
      if(!this.value.includes('@')) { // not email
       // show the error
       this.classList.add("errorClass");
       // ...and put the focus back
       input.focus();
      } else {
       this.classList.remove("errorClass");
      }
     };
    </script>
  </body>
</html>
```



Your email please:

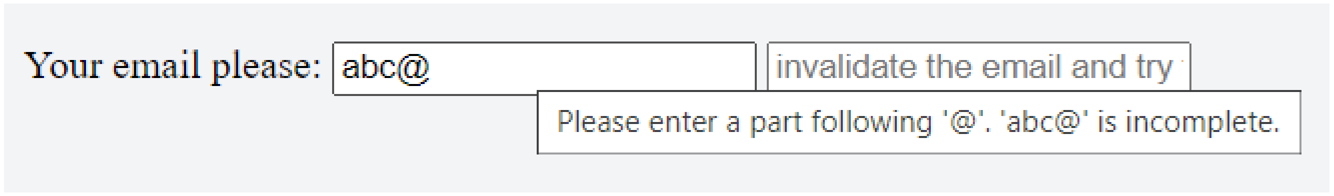
invalidate the email and try



Your email please: abc

invalidate the email and try

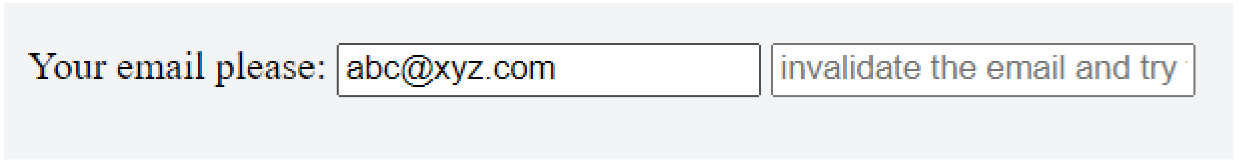
Please include an '@' in the email address. 'abc' is missing an '@'.



Your email please: abc@

invalidate the email and try

Please enter a part following '@'. 'abc@' is incomplete.



Your email please: abc@xyz.com

invalidate the email and try

- If you enter something into the input and then try to apply Tab or click away from the <input>, and then onblur brings the focus back.
- Another important note: it’s not possible to prevent losing focus by calling event.preventDefault() in onblur, as the latter works the element lost the focus. A focus loss can happen for different reasons.

- One of the reasons is when the user clicks somewhere else. But, JavaScript itself can lead to it, for example:
- An alert moves the focus to itself, causing the focus loss at the element (it’s a blur event). When the alert is discarded, the focus returns (focus event).
- In case an element is removed from DOM, it may also cause a focus loss. But, if it is reinserted later, the focus won’t return.

- The features above sometimes cause focus/blur handlers to misbehave: they trigger when it’s not necessary.
- The best solution is to be careful while using those events.

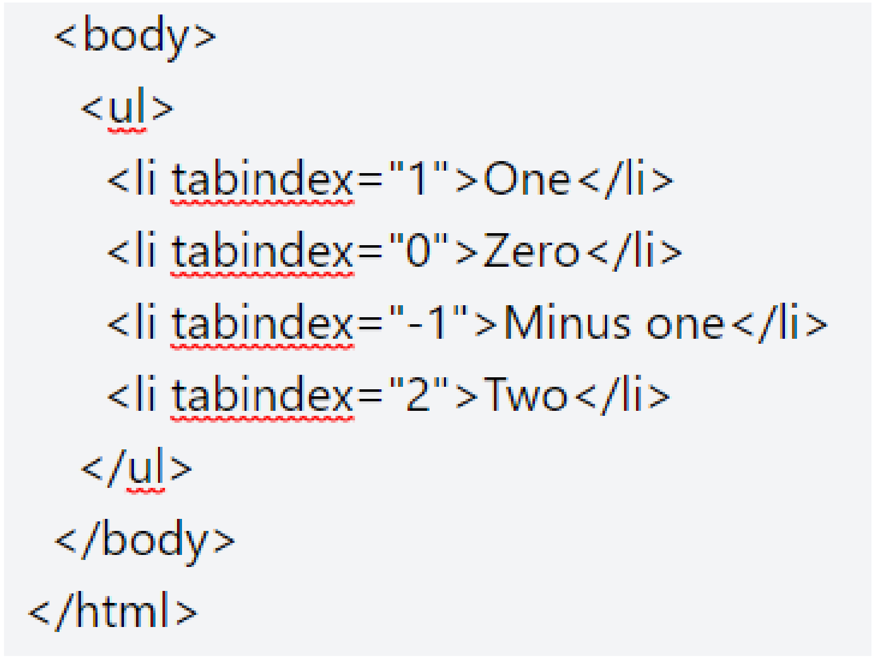
Allow Focusing on any Element: tabindex

- Focusing is not supported by many elements by default.
- The list can vary a little between browsers, but a thing is constantly correct: focus/blur support is guaranteed for elements that a user can interact with: <input>, <button>, <a>, and more.
- On the other hand, elements that exist for formatting something, such as <span>, <div>, <table> - are non focusable by default. The elem.focus() method doesn’t operate on them, and focus/blur events never trigger.

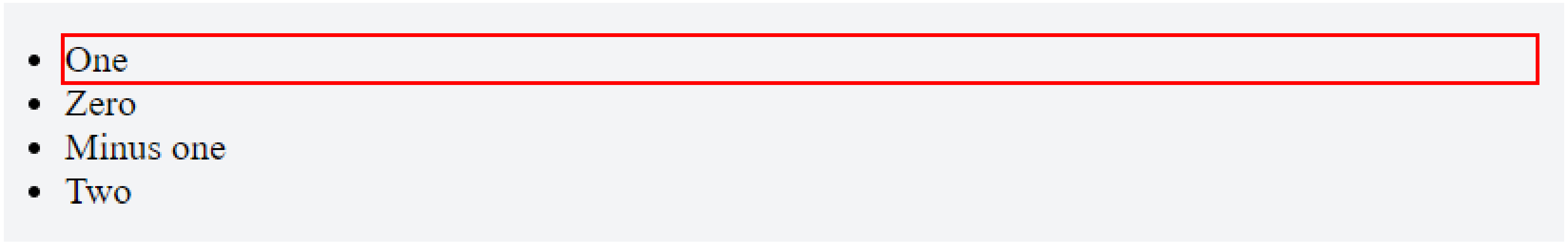
- That situation can be changed with HTML-attribute tabindex.
- When an element has tabindex, it becomes non focusable. The attribute value is the order number of the element when Tab is applied for switching between them. In other words: if there are two elements and the first hastabindex="1", the second- tabindex="2", using Tab while in the first one, moves the focus into the second.
- The switch order is as follows: elements with tabindex from 1 and above go first and only then go the elements without. Elements that have matching tabindex are switched in the document source order.

- Two special values exist:
- tabindex="0" that places an element amid the ones without tabindex. In other words, when you switch elements, the ones with tabindex=0 go after those with tabindex ≥ 1.
- As a rule, it is used for making an element focusable, keeping the default switching order.
- tabindex="-1" is used only for programmatic focusing on an element. The Tab key ignores elements like that, but the elem.focus() method works.
- Let’s take a look at the example below:

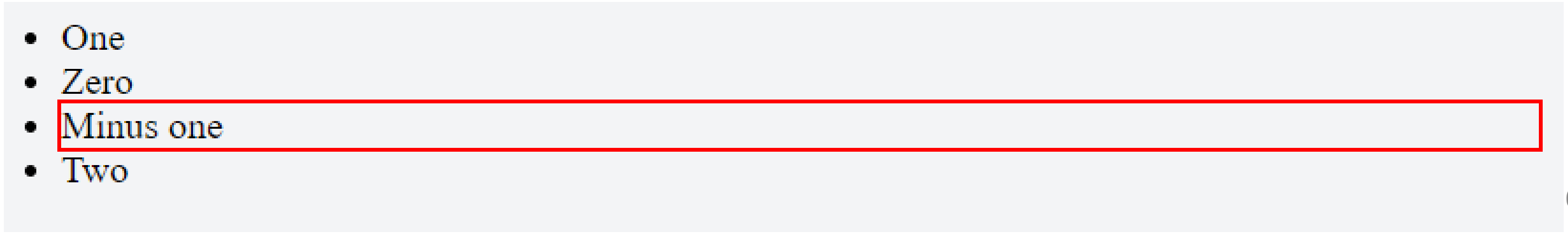
```
<!-- You should click the first item pressing the Tab. Keep track of the order. Please consider
that many subsequent Tabs are capable of moving the focus out of the iframe with the
example.-->
<!DOCTYPE HTML>
<html>
  <head>
    <title>Title of the Document</title>
    <style>
      li {
        cursor: pointer;
      }
      :focus {
        outline: 2px solid red;
      }
    </style>
```



```
  <body>
    <ul>
      <li tabindex="1">One</li>
      <li tabindex="0">Zero</li>
      <li tabindex="-1">Minus one</li>
      <li tabindex="2">Two</li>
    </ul>
  </body>
</html>
```



- One
- Zero
- Minus one
- Two



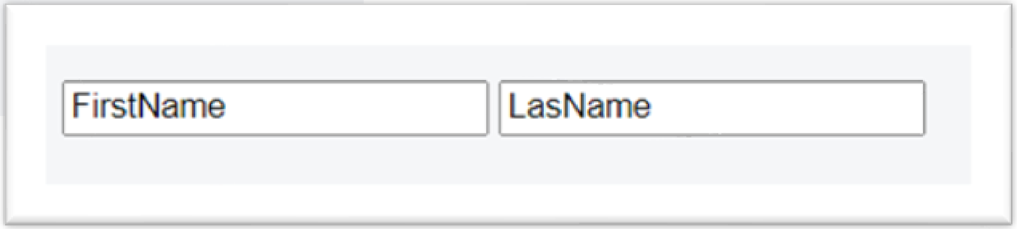
- One
- Zero
- Minus one
- Two

- Also, you can add tabindex from JavaScript by applying the elem.tabIndex property. The effect will be the same.

Delegation: focusin/focusout

- The focus and blur events don’t bubble.
- For example, it is not possible to put onfocus on the <form> to highlight it, as follows:

```
<html>
  <head>
    <title>Title of the Document</title>
    <style>
     .focused {
       outline: 1px solid red;
     }
    </style>
  </head>
  <body>
    <!-- on focusing in the form - add the class -->
    <form onfocus="this.className='focused'">
      <input type="text" name="firstname" value="FirstName">
      <input type="text" name="lastname" value="LasName">
    </form>
  </body>
</html>
```



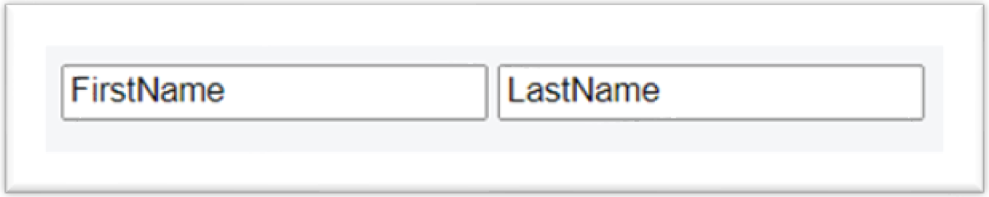
- The reason that the code above doesn’t work is that when the user focuses on the <input>, the focus event triggers on that input only. As it doesn’t bubble up, the form.onfocus doesn’t trigger either.

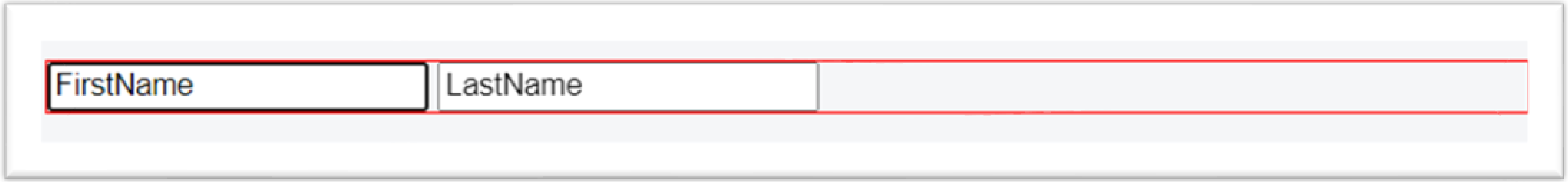
- In such a case, we can suggest two solutions.

- Within the first solution, focus/blur don’t bubble up but propagate down on the capturing phase. Hence, this is a working example:

- Focusin/focusout: This type of event can bubble.

```
<html>
  <head>
    <title>Title of the Document</title>
    <style>
     .focused {
       outline: 1px solid red;
     }
    </style>
  </head>
  <body>
    <form id="form">
      <input type="text" name="firstname" value="FirstName">
      <input type="text" name="lastname" value="LastName">
    </form>
    <script>
     form.addEventListener("focusin", () => form.classList.add('focused'));
     form.addEventListener("focusout", () => form.classList.remove('focused'));
    </script>
  </body>
</html>
```


FirstName
LastName


FirstName
LastName

Focusing:focus/blur-Summary

- The focus and blur events are a crucial part of any programming activity.
- The focus event triggers on focusing, while the blur event happens when the focus is lost.
- They have several specific features that are described below:
  - They never bubble. But, instead, you can use focusin/focusout that can bubble.
  - By default, the focus is not supported by most elements. The good thing is that you can use tabindex for making anything focusable.
- And, finally, the currently focused element may be available in document.activeElement.

Forms,controls-Events:change,input,cut,copy,paste

- the change event: it generally occurs on the focus loss for text input. So, this event triggers when a value was changed.
- the input event: it occurs for text inputs on every change. Unlike the change event, it triggers immediately.
- the cut/copy/paste events: these events occur while cutting/copying/pasting a value. Their actions can’t be prevented. The property event.clipboardData allows reading/writing access to the clipboard.

Forms: event and method submit

- The submit event triggers when the form is already submitted.
- Normally, developers use it to validate the form before sending it to the server or to cancel the submission and process it in JavaScript.
- The method form.submit() is useful for launching the form to send from JavaScript.
- It can be efficiently used to create and send own forms to the server.

Thank You