JavaScript Best Practices

Assoc. Prof. Dr. Kanda Runapongsa Saikaew
(krunapon@kku.ac.th)
Department of Computer Engineering
Khon Kaen University

Agenda

- JS Debugging
- JS Style Guide
- JS Mistakes
- JS Performance
- JS Forms
- Forms API

Code Debugging

- Errors can (will) happen, every time you write some new computer code
- Programming code might contain syntax errors, or logical errors
- Searching for (and fixing) errors in programming code is called code debugging

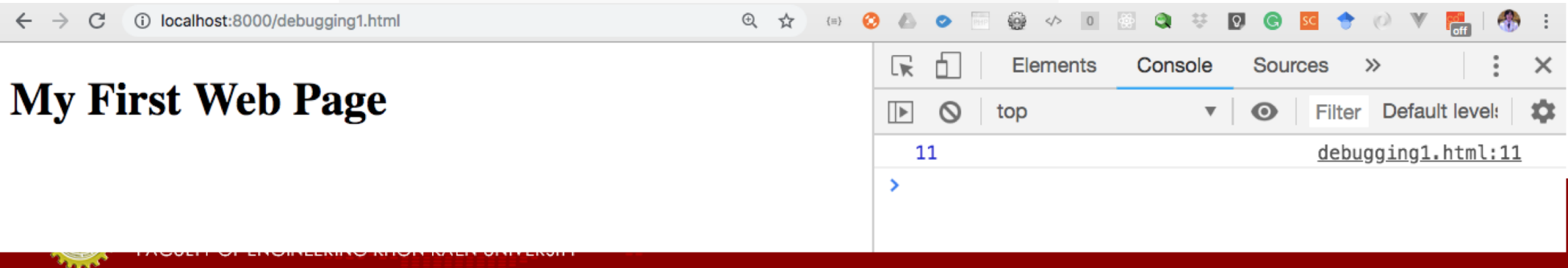
JavaScript Debuggers

- Debugging is not easy.
- But fortunately, all modern browsers have a built-in JavaScript debugger
- With a debugger, you can also set breakpoints, and examine variables while the code is executing
- You activate debugging your browser with the F12 key, and select “Console” in the debugger menu

The console.log() Method

- If your browser support debugging, you can use console.log() to display JavaScript values in the debugger window

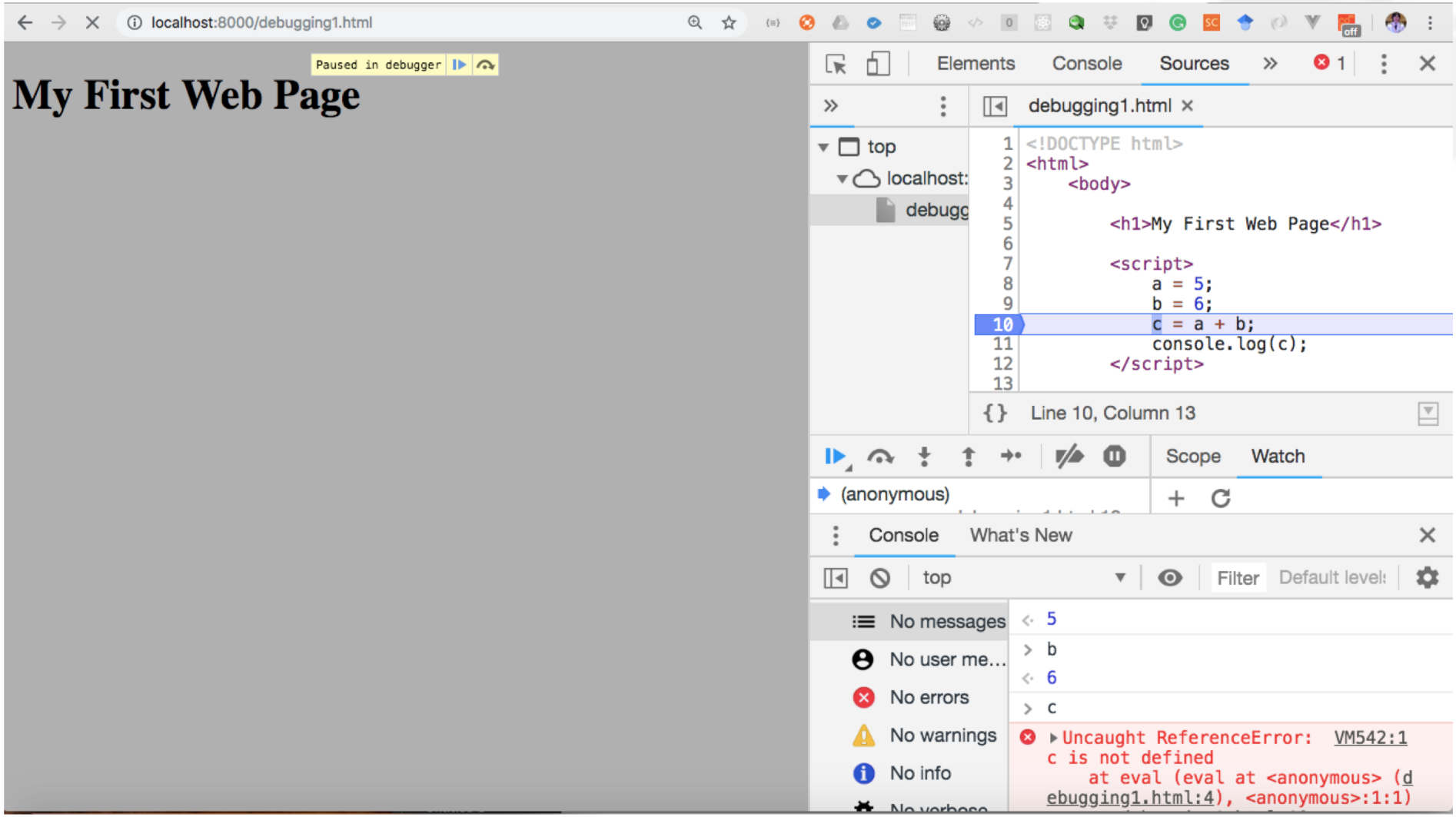
```
1   <!DOCTYPE html>
2   <html>
3       <body>
4
5           <h1>My First Web Page</h1>
6
7           <script>
8               a = 5;
9               b = 6;
10              c = a + b;
11              console.log(c);
12          </script>
```



Setting Breakpoints

- In the debugger window, you can set breakpoints in the JavaScript code.
- At each breakpoint, JavaScript will stop executing, and let you examine JavaScript values.
- After examining values, you can resume the execution of code (typically with a play button).

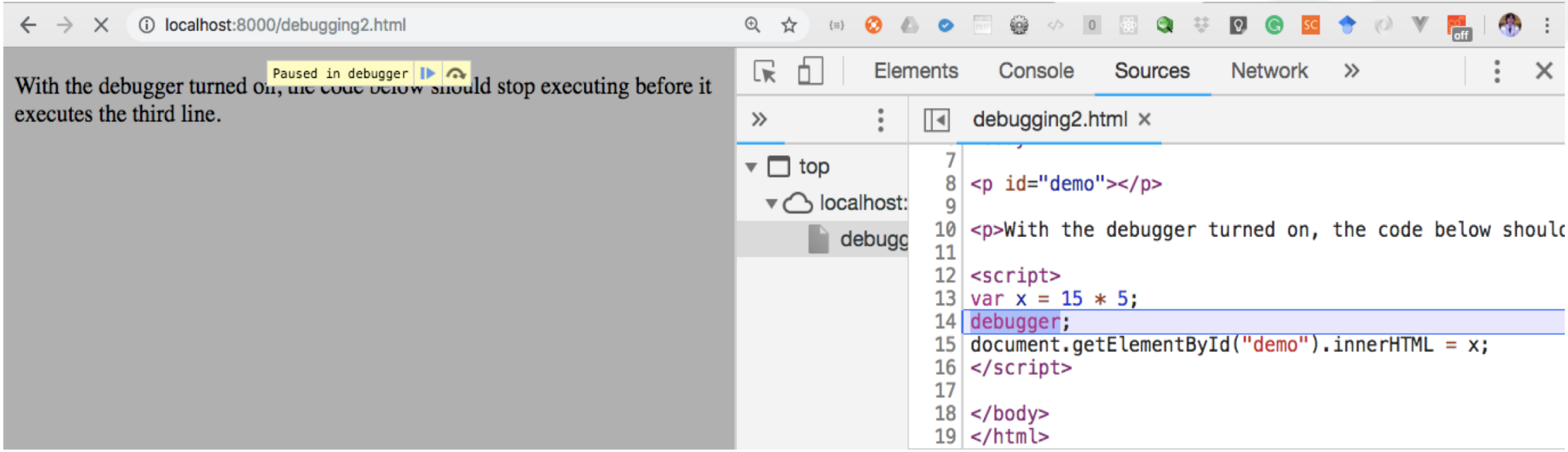
Setting Breakpoints Example



The debugger Keyword

- The **debugger** keyword stops the execution of JavaScript, and calls (if available) the debugging function.
- This has the same function as setting a breakpoint in the debugger.
- If no debugging is available, the debugger statement has no effect.
- With the debugger turned on, this code will stop executing before it executes the third line

Using debugger Keyword Example



Major Browser's Debugging Tools

- Normally, you activate debugging in your browser with F12, and select "Console" in the debugger menu
- Otherwise
  - Chrome
    - Open the browser.
    - From the menu, select "More tools".
    - From tools, choose "Developer tools".
    - Finally, select Console.

Agenda

- JS Debugging
- **JS Style Guide**
- JS Mistakes
- JS Performance
- JS Forms
- Forms API

JavaScript Coding Conventions

- Coding conventions are **style guidelines for programming**. They typically cover:
  - Naming and declaration rules for variables and functions.
  - Rules for the use of white space, indentation, and comments.
  - Programming practices and principles
- Coding conventions **secure quality**:
  - Improves code readability
  - Make code maintenance easier

Variable Names

- Use camelCase for identifier names (variables and functions)
- All names start with a letter
- Constants (Like PI) written in UPPERCASE
- camelCase is used by JavaScript itself, by jQuery, and other JavaScript libraries
- Hyphens can be mistaken as subtraction attempts. Hyphens are not allowed in JavaScript names.

Spaces Around Operators

- Always put spaces around operators (= + - * /), and after commas

Examples:

```
var x = y + z;
var values = ["Volvo", "Saab", "Fiat"];
```

Code Indentation

- Always use 4 spaces for indentation of code blocks

Functions:

```
function toCelsius(fahrenheit) {
    return (5 / 9) * (fahrenheit - 32);
}
```

Statement Rules

- Always end a statement with a semicolon

Examples:

```
var values = ["Volvo", "Saab", "Fiat"];

var person = {
    firstName: "John",
    lastName: "Doe",
    age: 50,
    eyeColor: "blue"
};
```

Examples of Functions, Loops, and Conditionals

Functions:

```
function toCelsius(fahrenheit) {
    return (5 / 9) * (fahrenheit - 32);
}
```

Loops:

```
for (i = 0; i < 5; i++) {
    x += i;
}
```

Conditionals:

```
if (time < 20) {
    greeting = "Good day";
} else {
    greeting = "Good evening";
}
```

Object Rules

- General rules for object definitions:
- Place the opening bracket on the same line as the object name.
- Use colon plus one space between each property and its value.
- Use quotes around string values, not around numeric values.
- Do not add a comma after the last property-value pair.
- Place the closing bracket on a new line, without leading spaces.
- Always end an object definition with a semicolon.

Examples of Object Rules

Example

```
var person = {
    firstName: "John",
    lastName: "Doe",
    age: 50,
    eyeColor: "blue"
};
```

- Short objects can be written compressed, on one line, using spaces only between properties, like this

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

Line Length < 80

- For readability, avoid lines longer than 80 characters.
- If a JavaScript statement does not fit on one line, the best place to break it, is after an operator or a comma.

Example

```
document.getElementById("demo").innerHTML =
    "Hello Dolly.";
```

Use Lower Case File Names

- Most web servers (Apache, Unix) are case sensitive about file names:
  - london.jpg cannot be accessed as London.jpg.
- Other web servers (Microsoft, IIS) are not case sensitive:
  - london.jpg can be accessed as London.jpg or london.jpg.

Avoid Global Variables

- Minimize the use of global variables.
- This includes all data types, objects, and functions.
- Global variables and functions can be overwritten by other scripts.
- Use local variables instead, and learn how to use closures

Always Declare Local Variables

- All variables used in a function should be declared as **local** variables.
- Local variables **must** be declared with the **var** keyword, otherwise they will become global variables.

Declarations on Top

- It is a good coding practice to put all declarations at the top of each script or function.
- This will:
  - Give cleaner code
  - Provide a single place to look for local variables
  - Make it easier to avoid unwanted (implied) global variables
  - Reduce the possibility of unwanted re-declarations

Examples of Variable Declarations

```
// Declare at the beginning
var firstName, lastName, price, discount, fullPrice;

// Use later
firstName = "John";
lastName = "Doe";

price = 19.90;
discount = 0.10;

fullPrice = price * 100 / discount;
```

Initialize Variables

- It is a good coding practice to initialize variables when you declare them.
- This will:
  - Give cleaner code
  - Provide a single place to initialize variables

```
// Declare and initiate at the beginning
var firstName = "",
    lastName = "",
    price = 0,
    discount = 0,
    fullPrice = 0,
    myArray = [],
    myObject = {};
```

Never Declare Number, String, or Boolean Objects

- Always treat numbers, strings, or booleans as primitive values. Not as objects.
- Declaring these types as objects, slows down execution speed, and produces nasty side effects

```
11  var x = "John";                    // x is a string
12  var y = new String("John");  // y is an object
13  document.getElementById("demo").innerHTML = x===y;
14  </script>
```

```
10 ▼ <script>
11   var x = new String("John");
12   var y = new String("John");
13   document.getElementById("demo").innerHTML = x==y;
14   </script>
```

Beware of Automatic Type Conversions

- Beware that numbers can accidentally be converted to strings or NaN (Not a Number).
- JavaScript is loosely typed. A variable can contain different data types, and a variable can change its data type:

```
var x = 5 + 7;       // x.valueOf() is 12,  typeof x is a number
var x = 5 + "7";     // x.valueOf() is 57,  typeof x is a string
var x = "5" + 7;     // x.valueOf() is 57,  typeof x is a string
var x = 5 - 7;       // x.valueOf() is -2,  typeof x is a number
var x = 5 - "7";     // x.valueOf() is -2,  typeof x is a number
var x = "5" - 7;     // x.valueOf() is -2,  typeof x is a number
var x = 5 - "x";     // x.valueOf() is NaN, typeof x is a number
```

Use === Comparison

- The == comparison operator always converts (to matching types) before comparison.
- The === operator forces comparison of values and type:

```
0 == "";        // true
1 == "1";       // true
1 == true;      // true

0 === "";       // false
1 === "1";      // false
1 === true;     // false
```

Use Parameter Defaults

- If a function is called with a missing argument, the value of the missing argument is set to **undefined**.
- Undefined values can break your code. It is a good habit to assign default values to arguments.

```
function myFunction(x, y) {
    if (y === undefined) {
        y = 0;
    }
}
```

ECMAScript 2015 allows default parameters in the function call:

```
function (a=1, b=1) { // function code }
```

Avoid Using eval()

- The eval() function is used to run text as code. In almost all cases, it should not be necessary to use it.
- Because it allows arbitrary code to be run, it also represents a security problem
- It makes your code slower and harder to maintain

```
var property = 'bar';
var value = eval('foo.' + property);
```

```
var property = 'bar';
var value = foo[property];
```

Agenda

- JS Debugging
- JS Style Guide
- **JS Mistakes**
- JS Performance
- JS Forms
- Forms API

Confusing Addition & Concatenation

- **Addition** is about adding **numbers**.
- **Concatenation** is about adding **strings**.
- In JavaScript both operations use the same + operator.
- Because of this, adding a number as a number will produce a different result from adding a number as a string:

```
var x = 10 + 5;          // the result in x is 15
var x = 10 + "5";        // the result in x is "105"
```

Accessing Arrays with Named Indexes

- Many programming languages support arrays with named indexes.
- Arrays with named indexes are called associative arrays (or hashes).
- JavaScript does **not** support arrays with named indexes.
- In JavaScript, **arrays** use **numbered indexes**
- In JavaScript, **objects** use **named indexes**.
- If you use a named index, when accessing an array, JavaScript will redefine the array to a standard object.
- After the automatic redefinition, array methods and properties will produce undefined or incorrect results:

Examples of Array Indexes

```
var person = [];
person[0] = "John";
person[1] = "Doe";
person[2] = 46;
var x = person.length;          // person.length will return 3
var y = person[0];              // person[0] will return "John"
```

```
var person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
var x = person.length;          // person.length will return 0
var y = person[0];              // person[0] will return undefined
```

Agenda

- JS Debugging
- JS Style Guide
- JS Mistakes
- **JS Performance**
- JS Forms
- Forms API

Reduce Activity in Loops

- Loops are often used in programming.
- Each statement in a loop, including the for statement, is executed for each iteration of the loop.
- Statements or assignments that can be placed outside the loop will make the loop run faster

Bad:

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

Reduce DOM Access

- Accessing the HTML DOM is very slow, compared to other JavaScript statements.
- If you expect to access a DOM element several times, access it once, and use it as a local variable
- Bad

```
1  document.getElementById("demo").innerHTML = "<h1>Hello</h1>";
2  document.getElementById("demo").textContent = "Hello";
```

Reduce DOM Size

- Keep the number of elements in the HTML DOM small.
- This will always improve page loading, and speed up rendering (page display), especially on smaller devices.
- Every attempt to search the DOM (like getElementsByTagName) will benefit from a smaller DOM

Avoid Unnecessary Variables

- Don't create new variables if you don't plan to save values.
- Often you can replace code like this:
- Bad

```
var fullName = firstName + " " + lastName;
document.getElementById("demo").innerHTML = fullName;
```

Delay JavaScript Loading

- Putting your scripts at the bottom of the page body lets the browser load the page first.
- While a script is downloading, the browser will not start any other downloads. In addition all parsing and rendering activity might be blocked.
- An alternative is to use **defer="true"** in the script tag. The defer attribute specifies that the script should be executed after the page has finished parsing, but it only works for external scripts.

Agenda

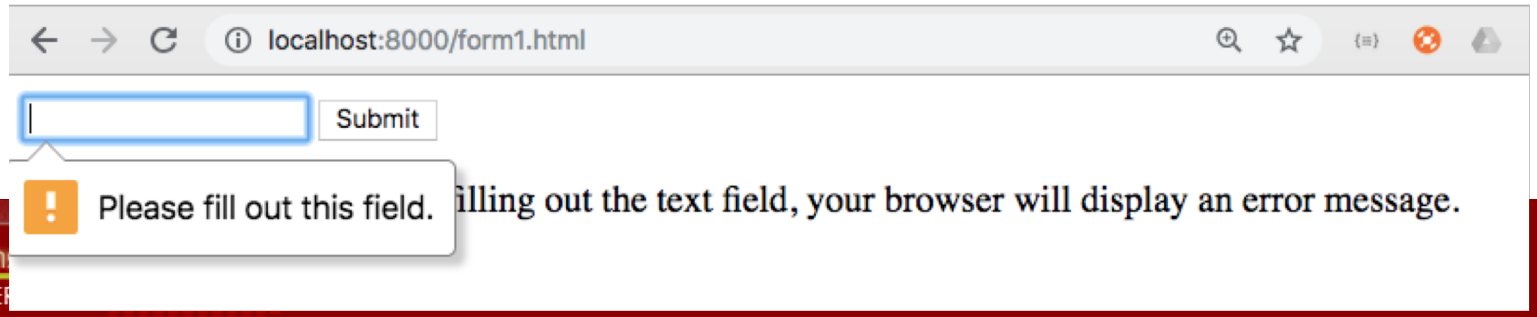
- JS Debugging
- JS Style Guide
- JS Mistakes
- JS Performance
- **JS Forms**
- Forms API

Automatic HTML Form Validation

- HTML form validation can be performed automatically by the browser:
- If a form field (fname) is empty, the **required** attribute prevents this form from being submitted:

HTML Form Example

```
<form action="/action_page.php" method="post">
  <input type="text" name="fname" required>
  <input type="submit" value="Submit">
</form>
```

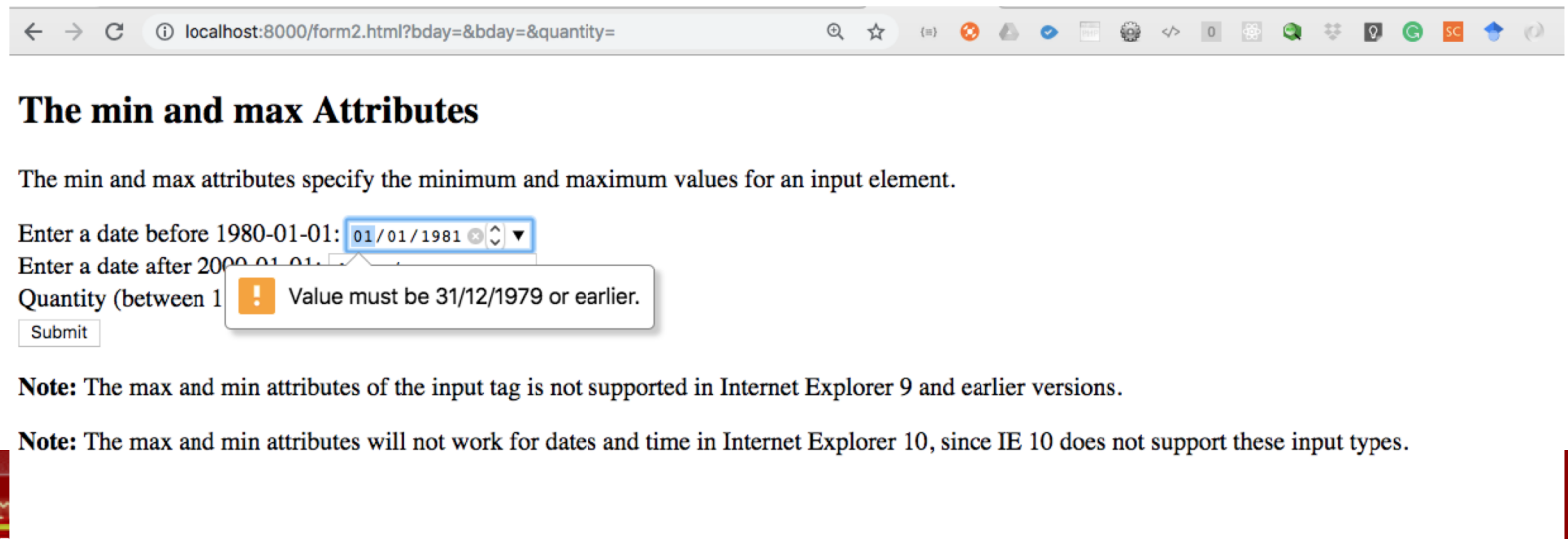


Constraint Validation HTML Input Attributes

Attribute	Description
disabled	Specifies that the input element should be disabled
max	Specifies the maximum value of an input element
min	Specifies the minimum value of an input element
pattern	Specifies the value pattern of an input element
required	Specifies that the input field requires an element
type	Specifies the type of an input element

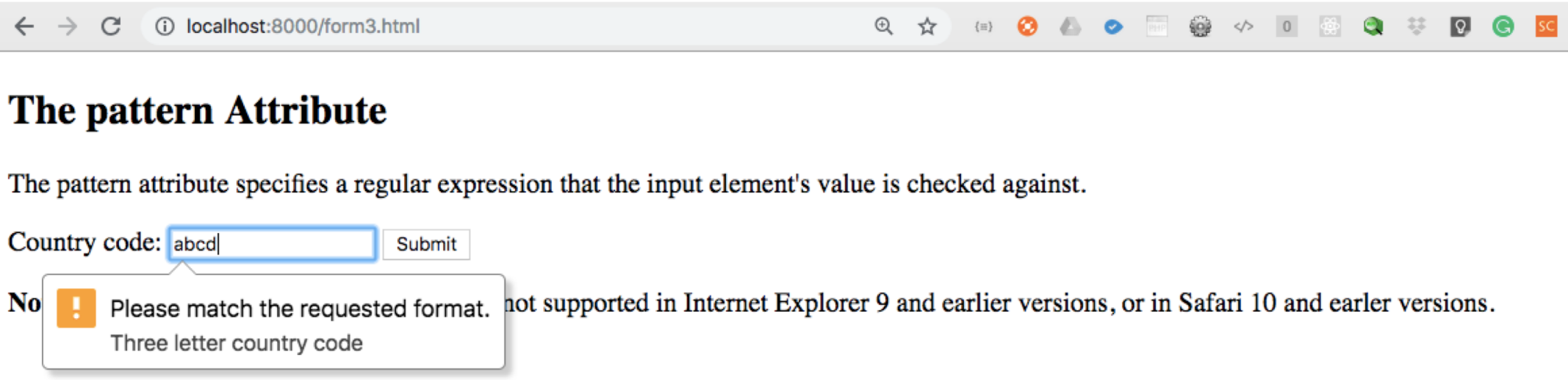
Example of Using min and max

```
<form>

  Enter a date before 1980-01-01:
  <input type="date" name="bday" max="1979-12-31"><br>

  Enter a date after 2000-01-01:
  <input type="date" name="bday" min="2000-01-02"><br>

  Quantity (between 1 and 5):
  <input type="number" name="quantity" min="1" max="5"><br>

  <input type="submit">

</form>
```



The pattern Attribute

- The pattern attribute specifies a regular expression that the <input> element's value is checked against
- The pattern attribute works with the following input types: text, search, url, tel, email, and password



Agenda

- JS Debugging
- JS Style Guide
- JS Mistakes
- JS Performance
- JS Forms
- **Forms API**

JavaScript Validation API

Constraint Validation DOM Methods

Property	Description
checkValidity()	Returns true if an input element contains valid data.
setCustomValidity()	Sets the validationMessage property of an input element.

The checkValidity() Method

```
<!DOCTYPE html>
<html>
<body>

<p>Enter a number and click OK:</p>

<input id="id1" type="number" min="100" max="300" required>
<button onclick="myFunction()">OK</button>

<p>If the number is less than 100 or greater than 300, an error message will be displayed.</p>

<p id="demo"></p>

<script>
function myFunction() {
    var inpObj = document.getElementById("id1");
    if (!inpObj.checkValidity()) {
        document.getElementById("demo").innerHTML = inpObj.validationMessage;
    } else {
        document.getElementById("demo").innerHTML = "Input OK";
    }
}
</script>

</body>
</html>
```

Example of the checkValidity() Method

```
1  <!DOCTYPE html>
2  <html>
3  <body>
4
5  <p>Enter a number and click OK:</p>
6
7  <input id="id1" type="number" min="100" max="300" required>
8  <button onclick="myFunction()">OK</button>
9
10 <p>If the number is less than 100 or greater than 300, an error message will be
   displayed.</p>
11
12 <p id="demo"></p>
13
14 <script>
15 function myFunction() {
16     var inpObj = document.getElementById("id1");
17     if (!inpObj.checkValidity()) {
18         document.getElementById("demo").innerHTML = inpObj.validationMessage;
19     } else {
20         document.getElementById("demo").innerHTML = "Input OK";
21     }
22 }
23 </script>
24
25 </body>
26 </html>
```

References

- https://www.w3schools.com/js/js_best_practices.asp
- https://24ways.org/2005/dont-be-eval