

CSS & JavaScript

CSS & JavaScript

- CSS Injection
- CSS Selector
- Responsive Design
- JS Introduction
- JS Injection
- JS Operator
- JS Function
- JS Method
- JS Datatype

CSS INJECTION



Three light gray circles are arranged horizontally. The first circle on the left contains the word 'INLINE'. The middle circle contains the word 'INTERNAL'. The third circle on the right contains the word 'EXTERNAL'. The circles are empty except for the text inside them.

INLINE

INTERNAL

EXTERNAL

CSS INJECTION

```
<!DOCTYPE html>
<html>

<head>
  <link rel="stylesheet" href="style.css">
  <style> .orange { color: orange; font-size: 35px}</style>
</head>

<body>
  <p style="color:red; font-size:35px">Hello Red World!</p>
  <p class="orange">Hello Orange World!</p>
  <p class="blue">Hello Blue World!</p>
</body>

</html>|
```

Hello Red World!

Hello Orange World!

Hello Blue World!

CSS Selector

Elements Selector:

- `p { color: red}`

Class Selector:

- `.blue { color: blue }`

ID Selector

- `#orange { color: orange
}`

CSS Display Property

display: inline

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum consequat scelerisque elit sit amet consequat. Aliquam erat volutpat. Aliquam venenatis gravida nisl sit amet facilisis. Nullam cursus fermentum velit sed laoreet.

display: inline-block

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum consequat scelerisque elit sit amet consequat. Aliquam erat volutpat. Aliquam venenatis gravida nisl sit amet facilisis. Nullam cursus fermentum velit sed laoreet.

display: block

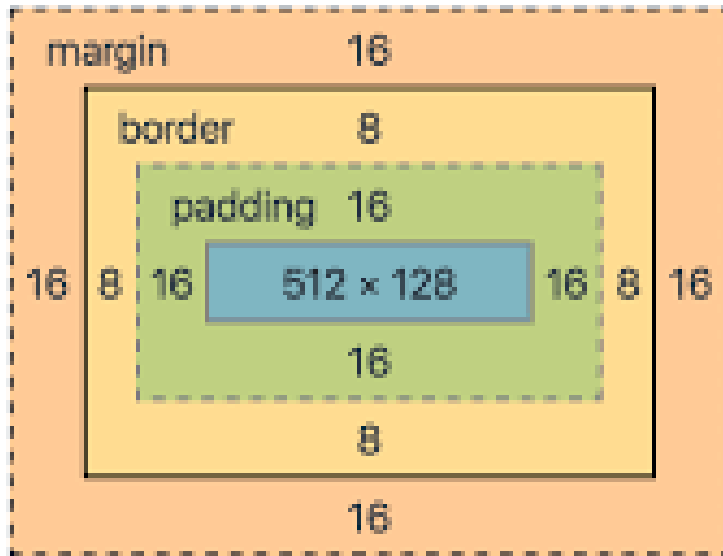
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum consequat scelerisque elit sit amet consequat. Aliquam erat volutpat.

Aliquam

venenatis

gravida nisl sit amet facilisis. Nullam cursus fermentum velit sed laoreet.

CSS Box Model



Responsive Web Design

Responsive Web design (RWD) is a Web design approach aimed at crafting sites to provide an optimal viewing experience—easy reading and navigation with a minimum of resizing, panning, and scrolling—across a wide range of devices.

- Setting The Viewport
- Responsive Images
- Frameworks: Bootstrap
- Media Query


```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

<meta> : viewport element gives the browser instructions on how to control the page's dimensions and scaling.

width=device-width : sets the width of the page to follow the screen-width of the device (which will vary depending on the device)

Initial-scale=1.0: sets the initial zoom level when the page is first loaded by the browser.

Bootstrap 4 Grid System

Bootstrap's grid system is built with flexbox and allows up to 12 columns across the page.

span 1	span 1	span 1	span 1	span 1	span 1	span 1	span 1	span 1	span 1	span 1	span 1
span 4				span 4				span 4			
span 4				span 8							
span 6						span 6					
span 12											

Structure of Bootstrap 4 Grid System

```
<!-- Control the column width, and how they should appear on different devices -->
<div class="row">
  <div class="col-*-*"></div>
  <div class="col-*-*"></div>
</div>
<div class="row">
  <div class="col-*-*"></div>
  <div class="col-*-*"></div>
  <div class="col-*-*"></div>
</div>

<!-- Or let Bootstrap automatically handle the layout -->
<div class="row">
  <div class="col"></div>
  <div class="col"></div>
  <div class="col"></div>
</div>
```

Media Query

```
@media only screen and (max-width: 600px) {  
  body {  
    background-color: lightblue;  
  }  
}
```

The @media Rule

Resize the browser window. When the width of this document is 600 pixels or less, the background-color is "lightblue", otherwise it is "yellow".

The @media Rule

Resize the browser window. When the width of this document is 600 pixels or less, the background-color is "lightblue", otherwise it is "yellow".

JavaScript

What is JS?

- JavaScript is what is called a Client-side Scripting Language. That means that it is a computer programming language that runs inside an Internet browser.

What is JS used for?

- Most of the dynamic behavior you see on a web page is thanks to JavaScript, which augments a browser's default controls and behaviors.

JS INJECTION

INLINE

EXTERNAL
(Commonly
Used)

```
<!DOCTYPE html>
<html>

  <head>
    <script>
      function Alert () {
        alert("Hello World!");
      }
    </script>
  </head>

  <body>
    <button style="margin:50px 0" onclick="Alert()">click me!</button>
  </body>

</html>
```

`<script src="script.js"><script>`

JS Assign Operators

Name	Shorthand operator	Meaning
Assignment	$x = y$	$x = y$
Addition assignment	$x += y$	$x = x + y$
Subtraction assignment	$x -= y$	$x = x - y$
Multiplication assignment	$x *= y$	$x = x * y$
Division assignment	$x /= y$	$x = x / y$
Remainder assignment	$x \% = y$	$x = x \% y$
Exponentiation assignment	$x ** = y$	$x = x ** y$
Left shift assignment	$x << = y$	$x = x << y$
Right shift assignment	$x >> = y$	$x = x >> y$
Unsigned right shift assignment	$x >>> = y$	$x = x >>> y$
Bitwise AND assignment	$x \& = y$	$x = x \& y$
Bitwise XOR assignment	$x \wedge = y$	$x = x \wedge y$
Bitwise OR assignment	$x = y$	$x = x y$

JS Comparison Operators

Operator	Description	Examples returning true
Equal (==)	Returns true if the operands are equal.	<pre>3 == var1"3" == var1 3 == '3'</pre>
Not equal (!=)	Returns true if the operands are not equal.	<pre>var1 != 4 var2 != "3"</pre>
Strict equal (===)	Returns true if the operands are equal and of the same type.	<pre>3 === var1</pre>
Strict not equal (!==)	Returns true if the operands are of the same type but not equal, or are of different type.	<pre>var1 !== "3" 3 !== '3'</pre>
Greater than (>)	Returns true if the left operand is greater than the right operand.	<pre>var2 > var1 "12" > 2</pre>
Greater than or equal (>=)	Returns true if the left operand is greater than or equal to the right operand.	<pre>var2 >= var1 var1 >= 3</pre>
Less than (<)	Returns true if the left operand is less than the right operand.	<pre>var1 < var2 "2" < 12</pre>
Less than or equal (<=)	Returns true if the left operand is less than or equal to the right operand.	<pre>var1 <= var2 var2 <= 5</pre>

JS Arithmetic Operators

Operator	Description	Example
Remainder (%)	Binary operator. Returns the integer remainder of dividing the two operands.	12 % 5 returns 2.
Increment (++)	Unary operator. Adds one to its operand. If used as a prefix operator (++x), returns the value of its operand after adding one; if used as a postfix operator (x++), returns the value of its operand before adding one.	If x is 3, then ++x sets x to 4 and returns 4, whereas x++ returns 3 and, only then, sets x to 4.
Decrement (--)	Unary operator. Subtracts one from its operand. The return value is analogous to that for the increment operator.	If x is 3, then --x sets x to 2 and returns 2, whereas x-- returns 3 and, only then, sets x to 2.
Unary negation (-)	Unary operator. Returns the negation of its operand.	If x is 3, then -x returns -3.
Unary plus (+)	Unary operator. Attempts to convert the operand to a number, if it is not already.	+ "3" returns 3. +true returns 1.
Exponentiation operator (**)	Calculates the base to the exponent power, that is, base ^{exponent}	2 ** 3 returns 8. 10 ** -1 returns 0.1.

JS Logic Operators

Operator	Usage	Description
Logical AND (&&)	expr1 && expr2	Returns expr1 if it can be converted to false; otherwise, returns expr2. Thus, when used with Boolean values, && returns true if both operands are true; otherwise, returns false.
Logical OR ()	expr1 expr2	Returns expr1 if it can be converted to true; otherwise, returns expr2. Thus, when used with Boolean values, returns true if either operand is true; if both are false, returns false.
Logical NOT (!)	!expr	Returns false if its single operand that can be converted to true; otherwise, returns true.

Console.log

```
var quick = "Fox";  
var slow = "Turtle";  
var numbers = 121;  
  
// The console.log() method is used to display data in the browser's console.  
// We can log strings, variables, and even equations.  
console.log("Teacher");  
console.log(quick);  
console.log(slow);  
console.log(numbers + 15);
```

- *console.log* is a quick expression used to print content to the debugger.
- It is a very useful tool to use during development and debugging

JS Datatypes

Primitive

- × String
- × Boolean
- × Null
- × Number
- × Undefined

Reference

- × Object

Primitive vs Reference

Primitive values uses its own storage space, changes on one does not reflected on another

Reference values are objects that are stored in the heap . Reference value stored in the variable location is a pointer to a location in memory where the object is stored

Null vs Undefined

Undefined means a variable has been declared but has not yet been assigned a value. On the other hand, *null* is an assignment value. It can be assigned to a variable as a representation of no value.

Also, undefined and null are two distinct types: undefined is a type itself (undefined) while null is an object.

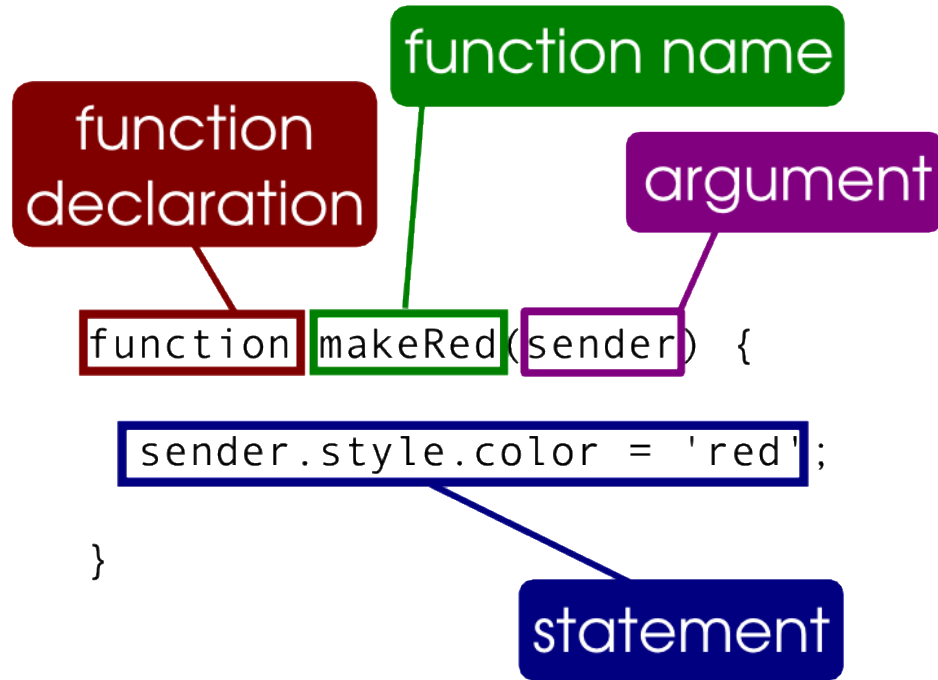
Unassigned variables are initialized by JavaScript with a default value of undefined. JavaScript never sets a value to null. That must be done programmatically.

Null vs Undefined

```
console.log("5" == 5);           // true  
console.log("5" === 5);          //false
```

```
console.log(undefined == null);   //true  
console.log(undefined === null); //false
```

JS FUNCTION



Conditional Statements

Very often when you write code, you want to perform different actions for different decisions.

You can use conditional statements in your code to do this.

In JavaScript we have the following conditional statements:

- Use ***if*** to specify a block of code to be executed, if a specified condition is true
- Use ***else*** to specify a block of code to be executed, if the same condition is false
- Use ***else if*** to specify a new condition to test, if the first condition is false
- Use ***switch*** to specify many alternative blocks of code to be executed

If/Else Statements

The *if statement* executes a statement if a specified condition is *truthy*. If the condition is *false*, another statement can be executed.

```
// If the user likes sushi (confirmSushi === true), we run the following block of code.  
if (confirmSushi) {  
  alert("You like " + sushiType + "!");  
}  
  
// If the user likes ginger tea (confirmGingerTea === true), we run the following block of code.  
else if (confirmGingerTea) {  
  alert("You like ginger tea!!");  
}  
  
// If neither of the previous condition were true, we run the following block of code.  
else {  
  document.write("You don't like sushi or ginger tea.");  
}
```

Switch Statements

expression

An expression whose result is matched against each case clause.

case valueN Optional

A case clause used to match against expression. If the expression matches the specified valueN, the statements inside the case clause are executed until either the end of the *switch* statement or a *break*.

default Optional

A default clause; if provided, this clause is executed if the value of expression doesn't match any of the case clauses.

```
var expr = 'Papayas';
switch (expr) {
  case 'Oranges':
    console.log('Oranges are $0.59 a pound.');
```

// expected output: "Oranges are \$0.59 a pound."

```
    break;
  case 'Mangoes':
  case 'Papayas':
    console.log('Mangoes and papayas are $2.79 a pound.');
```

// expected output: "Mangoes and papayas are \$2.79 a pound."

```
    break;
  default:
    console.log('Sorry, we are out of ' + expr + '.');
```

If/Else Statements

The *if statement* executes a statement if a specified condition is *truthy*. If the condition is *false*, another statement can be executed.

```
// If the user likes sushi (confirmSushi === true), we run the following block of code.  
if (confirmSushi) {  
  alert("You like " + sushiType + "!");  
}  
  
// If the user likes ginger tea (confirmGingerTea === true), we run the following block of code.  
else if (confirmGingerTea) {  
  alert("You like ginger tea!!");  
}  
  
// If neither of the previous condition were true, we run the following block of code.  
else {  
  document.write("You don't like sushi or ginger tea.");  
}
```


Loops

- There are several ways to execute a statement or block of statements repeatedly. In general, repetitive execution is called *looping*. It is typically controlled by a test of some variable, the value of which is changed each time the loop is executed. JavaScript supports many types of loops: **for** loops, **for...in** loops, **while** loops, **do...while** loops, and **switch** loops.

```
var str = "";
```

```
for (var i = 0; i < 9; i++) {  
    str = str + i;  
}
```

```
console.log(str);  
// expected output: "012345678"
```

```
var string1 = "";  
var object1 = {a: 1, b: 2, c: 3};
```

```
for (var property1 in object1) {  
    string1 = string1 + object1[property1];  
}
```

```
console.log(string1);  
// expected output: "123"
```

For Loop

For Loop: The **for statement** creates a loop that consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by a statement to be executed in the loop.

*for ([initialization]; [condition]; [final-expression])
statement*

For In Loop: The **for...in statement** iterates over all non-Symbol, enumerable properties of an object.

```
var n = 0;

while (n < 3) {
  n++;
}

console.log(n);
// expected output: 3
```

```
var result = "";
var i = 0;

do {
  i = i + 1;
  result = result + i;
} while (i < 5);

console.log(result);
// expected result: "12345"
```

While Loop

While Loop: The **while statement** creates a loop that executes a specified statement as long as the test condition evaluates to true. The condition is evaluated before executing the statement.

Do While Loop: The **do...while statement** creates a loop that executes a specified statement until the test condition evaluates to false. The condition is evaluated after executing the statement, resulting in the specified statement executing at least once.

Function Declaration vs. Expressions

```
//Function Declaration
function add(num1, num2){
    return num1 + num2;
}

//Function Expressions
var add=function (num1, num2){
    return num1 + num2;
};
```

```
//Function Declaration are Hoisted
var result = add(5,5);
function add(num1, num2){
    return num1 + num2;
}

//error!
var result = add(5,5);
var add=function (num1, num2){
    return num1 + num2;
};
```


JS Method

- When a property value in an object is a function, the property is considered to be a method.

```
var person = {  
  name: "Nicholas",  
  sayName: function() {  
    console.log(person.name);  
  }  
};
```

```
person.sayName();      // outputs "Nicholas"
```

The **this** object

- Every scope in JavaScript has a *this* object that represents the calling object for the function.
- When a function is called while attached to an object, the value of *this* is equal to that object by default.

```
var person = {  
  name: "Nicholas",  
  sayName: function() {  
    console.log(this.name);  
  }  
};
```

```
person.sayName();           // outputs "Nicholas"
```

this Example

```
function sayNameForAll() {  
    console.log(this.name);  
}
```

```
var person1 = {  
    name: "Nicholas",  
    sayName: sayNameForAll  
};
```


```
var person2 = {  
    name: "Greg",  
    sayName: sayNameForAll  
};
```

```
var name = "Michael";
```

```
person1.sayName();    // outputs "Nicholas"
```

```
person2.sayName();    // outputs "Greg"
```

```
sayNameForAll();      // outputs "Michael"
```



JS built in methods

Name	Description
toString()	Returns the string representation of the original data type
concat()	Combines the text of two strings and returns a new string.
indexOf()	Returns the index of the first occurrence of the specified value, or -1 if not found.
push()	Adds one or more elements to the end of an array and returns the new length of the array.
splice()	Adds and/or removes elements from an array.
map()	Creates a new array with the results of calling a provided function on every element in this array.
round()	Returns the value of a number rounded to the nearest integer.

Creating Objects

- Using *new* operator with a *constructor*

```
var object = new Object();
```

```
var object1=new Object();  
var object2=object1;
```



Understanding Objects

objects in JavaScript are dynamic, meaning that they can change at any point during code execution.

Defining Properties:

- Using the Object constructor and using an object literal
- When a property is first added to an object, JavaScript uses an internal method called `[[Put]]` on the object. The `[[Put]]` method creates a spot in the object to store the property
- When a new value is assigned to an existing property, a separate operation called `[[Set]]` takes place.

Add and Set Property

```
var person1 = {  
    name: "Nicholas"  
};  
  
var person2 = new Object();  
person2.name = "Nicholas";  
  
person1.age = "Redacted";  
person2.age = "Redacted";  
  
person1.name = "Greg";  
person2.name = "Michael";
```

person1	
name	"Nicholas"

[[Put]]name

person1	
name	"Nicholas"
age	"Redacted"

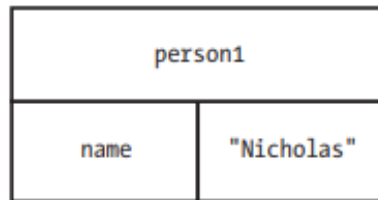
[[Put]]age

person1	
name	"Greg"
age	"Redacted"

[[Set]]name

Removing Properties

```
var person1 = {  
  name: "Nicholas"  
};  
  
console.log("name" in person1);    // true  
  
delete person1.name;               // true - not output  
console.log("name" in person1);    // false  
console.log(person1.name);         // undefined
```



`delete person1.name;`

Detecting Properties: Using *in* Operator

- The *in* operator looks for a property with a given name in a specific object and returns *true* if it finds it.

```
console.log("name" in person1);    // true
console.log("age" in person1);     // true
console.log("title" in person1);   // false
```

- The existence of a method can also be checked in the same way:

```
var person1 = {
  name: "Nicholas",
  sayName: function() {
    console.log(this.name);
  }
};

console.log("sayName" in person1); // true
```

hasOwnProperty()

- In some cases, however, you might want to check for the existence of a property only if it is an own property. The *in* operator checks for both own properties and prototype properties.
- The *toString()* method, however, is a prototype property that is present on all objects. The *in* operator returns true for *toString()*, but *hasOwnProperty()* returns false

```
var person1 = {  
  name: "Nicholas",  
  sayName: function() {  
    console.log(this.name);  
  }  
};  
  
console.log("name" in person1);           // true  
console.log(person1.hasOwnProperty("name")); // true  
  
console.log("toString" in person1);       // true  
console.log(person1.hasOwnProperty("toString")); // false
```

Enumeration

- By default, all properties that you add to an object are *enumerable*, which means that you can iterate over them using a *for-in* loop.
- *Enumerable* properties have their internal `[[Enumerable]]` attributes set to *true*.

```
var property;

for (property in object) {
  console.log("Name: " + property);
  console.log("Value: " + object[property]);
}
```

- Each time through the for-in loop, the property variable is filled with the next enumerable property on the object until all such properties have been used.

Object.keys() method

- The *Object.keys()* method returns an array of a given object's property *names*, in the same order as we get with a normal loop.

```
const object1 = {  
  a: 'somestring',  
  b: 42,  
  c: false  
};  
  
console.log(Object.keys(object1));  
// expected output: Array ["a", "b", "c"]
```

- There is a difference between the *enumerable* properties returned in a *for-in* loop and the ones returned by *Object.keys()*. The *for-in* loop also enumerates *prototype* properties, while *Object.keys()* returns only own (*instance*) properties.



Thanks!

Any questions?

You can find me at:
asel@itlize.com