CLASS and ID

id : it is unique and can be one in on element in whole page.

id can be use for scroll

anchoring for scroll to particular element or top/down.

<a href="#section-two">Section Two</a>

class : can be use for multiple div to decorate

2. NULL, Undeclared, undefined

-----------------------------------

In JavaScript, undefined means a variable has been declared but has not yet been assigned a value, such as:

 var TestVar;

 alert(TestVar); //shows undefined

 alert(typeof TestVar); //shows undefined

null is an assignment value. It can be assigned to a variable as a representation of no value:

 var TestVar = null;

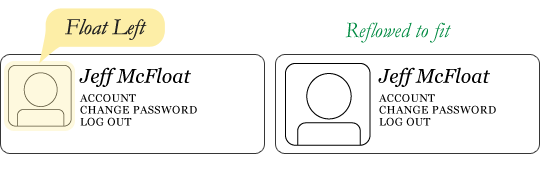
 alert(TestVar); //shows null

 alert(typeof TestVar); //shows object

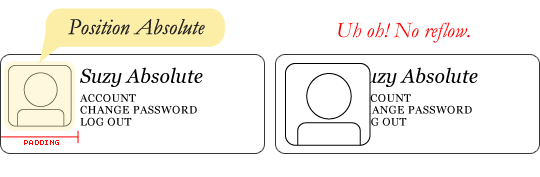
**Floats**

There are four valid values for the float property. **Left** and **Right** float elements those directions respectively. **None** (the default) ensures the element will not float and **Inherit** which will assume the float value from that elements parent element.

Floats are also helpful for layout in smaller instances. Take for example this little area of a web page. If we use float for our little avatar image, when that image changes size the text in the box will reflow to accommodate:

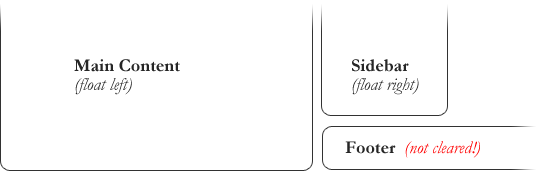


This same layout could be accomplished using relative positioning on container and absolute positioning on the avatar as well. In doing it this way, the text would be unaffected by the avatar and not be able to reflow on a size change.



**Clearing the floats**

Float's sister property is clear. An element that has the clear property set on it will not move up adjacent to the float like the float desires, but will move itself down past the float. Again an illustration probably does more good than words do.

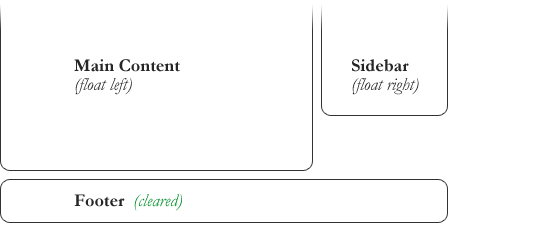


In the above example, the sidebar is floated to the right and is shorter than the main content area. The footer then is required to jump up into that available space as is required by the float. To fix this problem, the footer can be cleared to ensure it stays beneath both floated columns.

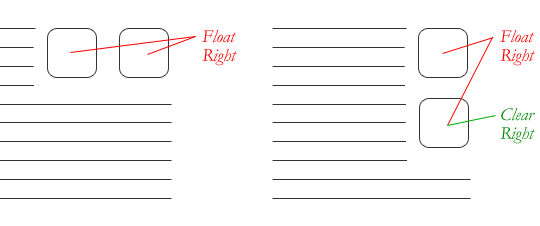
#footer {

clear: both;

}



Clear has four valid values as well. **Both** is most commonly used, which clears floats coming from either direction. **Left** and **Right** can be used to only clear the float from one direction respectively. **None** is the default, which is typically unnecessary unless removing a clear value from a cascade. **Inherit** would be the fifth, but is strangely not supported in Internet Explorer. Clearing only the left or right float, while less commonly seen in the wild, definitely has its uses.



**Absolute, Relative, Fixed Positioning**

* **Static**. This is the default for every single page element. Different elements don't have different default values for positioning, they all start out as static. Static doesn't mean much, it just means that the element will flow into the page as it normally would. The only reason you would ever set an element to position: static is to forcefully-remove some positioning that got applied to an element outside of your control. This is fairly rare, as positioning doesn't cascade.

position: relative;

An element with position: relative; is positioned relative to its normal position.

Setting the top, right, bottom, and left properties of a relatively-positioned element will cause it to be adjusted away from its normal position. Other content will not be adjusted to fit into any gap left by the element.

* An element with position: absolute; is positioned relative to the nearest positioned ancestor (instead of positioned relative to the viewport, like fixed).
* However; if an absolute positioned element has no positioned ancestors, it uses the document body, and moves along with page scrolling.
* **Note:** A "positioned" element is one whose position is anything except static.
* **Fixed**. This type of positioning is fairly rare but certainly has its uses. A fixed position element is positioned relative to the *viewport*, or the browser window itself. The viewport doesn't change when the window is scrolled, so a fixed positioned element will stay right where it is when the page is scrolled, creating an effect a bit like the old school "frames" days. Take a look at [this site](http://www.bluelounge.com/index.php) (update: dead link, sorry), where the left sidebar is fixed. This site is a perfect example for since it exhibits both good and bad traits of fixed positioning. The good is that it keeps the navigation present at all times on the page and it creates and interested effect on the page. The bad is that there are some usability concerns. On my smallish laptop, the content in the sidebar is cut off and there is no way from me to scroll down to see the rest of that content. Also if I scroll all the way down to the footer, it overlaps the footer content not allowing me to see all of that. Cool effect, can be useful, but needs to be thoroughly tested.

## Lexical scoping

Consider the following:

function init() {

var name = "Mozilla"; // name is a local variable created by init

function displayName() { // displayName() is the inner function, a closure

alert(name); // use variable declared in the parent function

}

displayName();

}

init();

init() creates a local variable name and then a function called displayName(). displayName() is an inner function that is defined inside init() and is only available within the body of that function.displayName() has no local variables of its own, however it has access to the variables of outer functions and so can use the variable name declared in the parent function.

[Run](http://jsfiddle.net/xAFs9/3/) the code and see that this works. This is an example of lexical scoping: in JavaScript, the scope of a variable is defined by its location within the source code (it is apparent lexically) and nested functions have access to variables declared in their outer scope.

## Closure

Now consider the following example:

function makeFunc() {

var name = "Mozilla";

function displayName() {

alert(name);

}

return displayName;

}

var myFunc = makeFunc();

myFunc();

If you run this code it will have exactly the same effect as the previous init() example: the string "Mozilla" will be displayed in a JavaScript alert box. What's different — and interesting — is that thedisplayName() inner function was returned from the outer function before being executed.

That the code still works may seem unintuitive. Normally, the local variables within a function only exist for the duration of that function's execution. Once makeFunc() has finished executing, it is reasonable to expect that the name variable will no longer be accessible. Since the code still works as expected, this is obviously not the case.

The solution to this puzzle is that myFunc has become a closure. A closure is a special kind of object that combines two things: a function, and the environment in which that function was created. The environment consists of any local variables that were in-scope at the time that the closure was created. In this case, myFunc is a closure that incorporates both the displayName function and the "Mozilla" string that existed when the closure was created.

Here's a slightly more interesting example — a makeAdder function:

function makeAdder(x) {

return function(y) {

return x + y;

};

}

var add5 = makeAdder(5);

var add10 = makeAdder(10);

console.log(add5(2)); // 7

console.log(add10(2)); // 12

In this example, we have defined a function makeAdder(x) which takes a single argument x and returns a new function. The function it returns takes a single argument y, and returns the sum of x and y.

In essence, makeAdder is a function factory — it creates functions which can add a specific value to their argument. In the above example we use our function factory to create two new functions — one that adds 5 to its argument, and one that adds 10.

add5 and add10 are both closures. They share the same function body definition, but store different environments. In add5's environment, x is 5. As far as add10 is concerned, x is 10.

## Practical closures

That's the theory out of the way — but are closures actually useful? Let's consider their practical implications. A closure lets you associate some data (the environment) with a function that operates on that data. This has obvious parallels to object oriented programming, where objects allow us to associate some data (the object's properties) with one or more methods.

Consequently, you can use a closure anywhere that you might normally use an object with only a single method.

Situations where you might want to do this are particularly common on the web. Much of the code we write in web JavaScript is event-based — we define some behavior, then attach it to an event that is triggered by the user (such as a click or a keypress). Our code is generally attached as a callback: a single function which is executed in response to the event.

Here's a practical example: suppose we wish to add some buttons to a page that adjust the text size. One way of doing this is to specify the font-size of the body element in pixels, then set the size of the other elements on the page (such as headers) using the relative em unit:

body {

font-family: Helvetica, Arial, sans-serif;

font-size: 12px;

}

h1 {

font-size: 1.5em;

}

h2 {

font-size: 1.2em;

}

Our interactive text size buttons can change the font-size property of the body element, and the adjustments will be picked up by other elements on the page thanks to the relative units.

Here's the JavaScript:

function makeSizer(size) {

return function() {

document.body.style.fontSize = size + 'px';

};

}

var size12 = makeSizer(12);

var size14 = makeSizer(14);

var size16 = makeSizer(16);

size12, size14, and size16 are now functions which will resize the body text to 12, 14, and 16 pixels, respectively. We can attach them to buttons (in this case links) as follows:

document.getElementById('size-12').onclick = size12;

document.getElementById('size-14').onclick = size14;

document.getElementById('size-16').onclick = size16;

<a href="#" id="size-12">12</a>

<a href="#" id="size-14">14</a>

<a href="#" id="size-16">16</a>

## What are closures and how are they used?

* This is a mouth full to try and explain, and you’ll probably need to whiteboard an example if this question comes up. The simplest definition I could find comes from Jibbering.com blog:
  + “A “closure” is an expression (typically a function) that can have free variables together with an environment that binds those variables (that “closes” the expression).”
* If you’re like most people and that definition means nothing to you, [MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Closures)  has an excellent section on closures that is chalk full of examples. Even if you’ve never recognized them as closures, you’ve probably seen or even used these patterns before.

## Can you use x === “object” to test if x is an object?

* In short, yes, but you must take into account the fact that null is considered an object in JavaScript. Even if x is null, ‘console.log(typeof x === “object”)’ will log true instead of false.
* To account for this, you must also test whether or not x is null by including the following:
  + console.log((x !== null) && (typeof x === “object”));

## What happens when you don’t declare a variable in Javascript?

* If you don’t explicitly declare a variable, you risk creating an implied global variable. This is a problem, as you will be unable to create multiple instances of that object, as each new instance will overwrite the data from the last.
* In general, global variables should be used only in very specific situations and are typically not recommended, as they can lead to a lot of odd side effects that may be difficult to track down.

## What is the difference between == and === ?

* ‘==’ evaluates equality of the value, while ‘===’ evaluates  equality of type and value.

## What datatypes are supported in Javascript?

* Undefined
* Number
* String
* Boolean
* Object
* Function
* Null

## What would  “1”+2+3  and 1+2+“3” evaluate to, respectively?

* When the first character is a string, the remaining characters will be converted into a single string, rendering 123.
* If the string is at the end, the initial characters will perform normal mathematical functions before converting into a string, resulting in 33.

## What is the  difference between window.onload and the jQuery $(document).ready() method?

* The window.onload method occurs after all the page elements have loaded(HTML, CSS, images), which can result in a delay.
* The $(document).ready() method begins to run code as soon as the Document Object Model (DOM) is loaded, which should be faster and less prone to loading errors across different browsers.

## Explain the concept of unobtrusive Javascript?

* Unobtrusive JavaScript is basically a JavaScript methodology that seeks to overcome browser inconsistencies by separating page functionality from structure. The basic premise of unobtrusive JavaScript is that page functionality should be maintained even when JavaScript is unavailable on a user’s browser.

## What is the difference between a null value and an undefined value?

* Null is used to assign an empty value to a variable and needs to be assigned manually.
* Undefined values result when you declare a variable without assigning it a value. Undefined will be the default whenever you don’t explicitly assign a value.

## Which conditional statements will JavaScript support?

* if statement
* if – else statement
* if – else if – else statement
* switch statement

## What is NaN?

* Nan is literally “Not-a-Number”. NaN usually results when either the result or one of the values in an operation is non-numeric.  Even though NaN is not a number, ‘console.log(typeof NaN === “number”);’ logs true, while NaN compared to anything else (including NaN) logs false. The only real way to test if a value is equal to NaN is with the function ‘isNaN()’.

## What types of pop-up boxes you can create in JavaScript and how can you create them?

* Alert, prompt, and confirm boxes are the three main pop-up boxes that can be created in JavaScript.
* Alert boxes are used to alert the user to important info regarding the site and require the user to click ‘OK’ in order to proceed.
* Alert boxes are created using the following syntax:
  + window.alert(“Text”);
* Prompt boxes prompt the user to input some form of data before proceeding.
* Prompt boxes are created using the following syntax:
  + window.prompt(“Prompt Text”,”Default value”);
* Confirm boxes are used when for verification, such as agreeing to a terms of service, rendering a TRUE value when the user clicks ‘OK’.
* Confirm boxes are created with the following syntax:
  + window.confirm(“Confirm text here.”);

## Which boolean operators are in JavaScript?

* ‘and’ operator: &&
* ‘or’ operator: ||
* ‘not’ operator:  !

## Why would you include ‘use strict’ at the beginning of a JavaScript source file?

* Using strict mode enforces stricter error handling when running your code. It essentially raises errors that would have otherwise failed silently. Using strict mode can help you avoid simple mistakes like creating accidental globals, undefined values, or duplicate property names. This is typically a good idea when writing JavaScript, as such errors can create a lot of frustrating side effects and be difficult to track down.

## What does a timer do and how would you implement one?

* Setting timers allows you to execute your code at predefined times or intervals.
* This can be achieved through two main methods: setInterval(); and setTimeout();
* setInterval() accepts a function and a specified number of milliseconds.
  + ex) setInterval(function(){alert(“Hello, World!”),10000) will alert the “Hello, World!” function every 10 seconds.
* setTimeout() also accepts a function, followed by milliseconds. setTimeout() will only execute the function once after the specified amount of time, and will not reoccur in intervals.

## Is there automatic type conversion in JavaScript?

* Yes, JavaScript supports automatic type conversion. In the example below, JavaScript is expecting a string. When it receives a fixnum as an operand, it’s automatically converted to a string.
  + ex)
    - 5 + “ cats” = “5 cats”

What is a potential pitfall with using typeof bar === "object" to determine if bar is an object? How can this pitfall be avoided?

Hide answer

Although typeof bar === "object" *is* a reliable way of checking if bar is an object, the surprising gotcha in JavaScript is that null is *also* considered an object!

Therefore, the following code will, to the surprise of most developers, log true (not false) to the console:

var bar = null;

console.log(typeof bar === "object"); // logs true!

As long as one is aware of this, the problem can easily be avoided by also checking if bar is null:

console.log((bar !== null) && (typeof bar === "object")); // logs false

To be entirely thorough in our answer, there are two other things worth noting:

First, the above solution will return false if bar is a function. In most cases, this is the desired behavior, but in situations where you want to also return true for functions, you could amend the above solution to be:

console.log((bar !== null) && ((typeof bar === "object") || (typeof bar === "function")));

Second, the above solution will return true if bar is an array (e.g., if var bar = [];). In most cases, this is the desired behavior, since arrays are indeed objects, but in situations where you want to also false for arrays, you could amend the above solution to be:

console.log((bar !== null) && (typeof bar === "object") && (toString.call(bar) !== "[object Array]"));

Or, if you’re using jQuery:

console.log((bar !== null) && (typeof bar === "object") && (! $.isArray(bar)));

What will the code below output to the console and why?

(function(){

var a = b = 3;

})();

console.log("a defined? " + (typeof a !== 'undefined'));

console.log("b defined? " + (typeof b !== 'undefined'));

Hide answer

Since both a and b are defined within the enclosing scope of the function, and since the line they are on begins with the varkeyword, most JavaScript developers would expect typeof a and typeof b to both be *undefined* in the above example.

However, that is *not* the case. The issue here is that most developers *incorrectly* understand the statement var a = b = 3; to be shorthand for:

var b = 3;

var a = b;

But in fact, var a = b = 3; is actually shorthand for:

b = 3;

var a = b;

As a result (if you are *not* using strict mode), the output of the code snippet would be:

a defined? false

b defined? true

But how can b be defined *outside* of the scope of the enclosing function? Well, since the statement var a = b = 3; is shorthand for the statements b = 3; and var a = b;, b ends up being a global variable (since it is not preceded by the var keyword) and is therefore still in scope even outside of the enclosing function.

Note that, in strict mode (i.e., with [use strict](http://www.w3schools.com/js/js_strict.asp)), the statement var a = b = 3; will generate a runtime error of ReferenceError: b is not defined, thereby avoiding any headfakes/bugs that might othewise result. (Yet another prime example of why you should use use strict as a matter of course in your code!)

What will the code below output to the console and why?

var myObject = {

foo: "bar",

func: function() {

var self = this;

console.log("outer func: this.foo = " + this.foo);

console.log("outer func: self.foo = " + self.foo);

(function() {

console.log("inner func: this.foo = " + this.foo);

console.log("inner func: self.foo = " + self.foo);

}());

}

};

myObject.func();

Hide answer

The above code will output the following to the console:

outer func: this.foo = bar

outer func: self.foo = bar

inner func: this.foo = undefined

inner func: self.foo = bar

In the outer function, both this and self refer to myObject and therefore both can properly reference and access foo.

In the inner function, though, this no longer refers to myObject. As a result, this.foo is undefined in the inner function, whereas the reference to the local variable self remains in scope and is accessible there. (Prior to ECMA 5, this in the inner function would refer to the global window object; whereas, as of ECMA 5, this in the inner function would be undefined.)

What is the significance of, and reason for, wrapping the entire content of a JavaScript source file in a function block?

Hide answer

This is an increasingly common practice, employed by many popular JavaScript libraries (jQuery, Node.js, etc.). This technique creates a closure around the entire contents of the file which, perhaps most importantly, creates a private namespace and thereby helps avoid potential name clashes between different JavaScript modules and libraries.

Another feature of this technique is to allow for an easily referenceable (presumably shorter) alias for a global variable. This is often used, for example, in jQuery plugins. jQuery allows you to disable the $ reference to the jQuery namespace, using jQuery.noConflict(). If this has been done, your code can still use $ employing this closure technique, as follows:

(function($) { /\* jQuery plugin code referencing $ \*/ } )(jQuery);

What is the significance, and what are the benefits, of including 'use strict' at the beginning of a JavaScript source file?

Hide answer

the short and most important answer here is that use strict is a way to voluntarily enforce stricter parsing and error handling on your JavaScript code at runtime. Code errors that would otherwise have been ignored or would have failed silently will now generate errors or throw exceptions. In general, it is a good practice.

Some of the key benefits of strict mode include:

* **Makes debugging easier.** Code errors that would otherwise have been ignored or would have failed silently will now generate errors or throw exceptions, alerting you sooner to problems in your code and directing you more quickly to their source.
* **Prevents accidental globals.** Without strict mode, assigning a value to an undeclared variable automatically creates a global variable with that name. This is one of the most common errors in JavaScript. In strict mode, attempting to do so throws an error.
* **Eliminates this coercion**. Without strict mode, a reference to a this value of null or undefined is automatically coerced to the global. This can cause many headfakes and pull-out-your-hair kind of bugs. In strict mode, referencing a a this value of null or undefined throws an error.
* **Disallows duplicate property names or parameter values.** Strict mode throws an error when it detects a duplicate named property in an object (e.g., var object = {foo: "bar", foo: "baz"};) or a duplicate named argument for a function (e.g., function foo(val1, val2, val1){}), thereby catching what is almost certainly a bug in your code that you might otherwise have wasted lots of time tracking down.
* **Makes eval() safer.** There are some differences in the way eval() behaves in strict mode and in non-strict mode. Most significantly, in strict mode, variables and functions declared inside of an eval() statement are *not* created in the containing scope (they *are* created in the containing scope in non-strict mode, which can also be a common source of problems).
* **Throws error on invalid usage of delete.** The delete operator (used to remove properties from objects) cannot be used on non-configurable properties of the object. Non-strict code will fail silently when an attempt is made to delete a non-configurable property, whereas strict mode will throw an error in such a case.

Consider the two functions below. Will they both return the same thing? Why or why not?

function foo1()

{

return {

bar: "hello"

};

}

function foo2()

{

return

{

bar: "hello"

};

}

Hide answer

Surprisingly, these two functions will *not* return the same thing. Rather:

console.log("foo1 returns:");

console.log(foo1());

console.log("foo2 returns:");

console.log(foo2());

will yield:

foo1 returns:

Object {bar: "hello"}

foo2 returns:

undefined

Not only is this surprising, but what makes this particularly gnarly is that foo2() returns undefined without any error being thrown.

The reason for this has to do with the fact that semicolons are technically optional in JavaScript (although omitting them is generally really bad form). As a result, when the line containing the return statement (with nothing else on the line) is encountered in foo2(), a semicolon is automatically inserted immediately after the return statement.

No error is thrown since the remainder of the code is perfectly valid, even though it doesn’t ever get invoked or do anything (it is simply an unused code block that defines a property bar which is equal to the string "hello").

This behavior also argues for following the convention of placing an opening curly brace at the end of a line in JavaScript, rather than on the beginning of a new line. As shown here, this becomes more than just a stylistic preference in JavaScript.

What will be the output of the following code:

for (var i = 0; i < 5; i++) {

setTimeout(function() { console.log(i); }, i \* 1000 );

}

Explain your answer. How could the use of closures help here?

Hide answer

The code sample shown will ***not*** display the values 0, 1, 2, 3, and 4 as might be expected; rather, it will display 5, 5, 5, 5, and 5.

The reason for this is that each function executed within the loop will be executed *after* the entire loop has completed and *all* will therefore reference the *last* value stored in i, which was 5.

[**Closures**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures) can be used to prevent this problem by creating a unique scope for each iteration, storing each unique value of the variable within its scope, as follows:

for (var i = 0; i < 5; i++) {

(function(x) {

setTimeout(function() { console.log(x); }, x \* 1000 );

})(i);

}

This will produce the presumably desired result of logging 0, 1, 2, 3, and 4 to the console.

**What is the difference between ViewState and SessionState?**

‘ViewState’ is specific to a page in a session.

‘SessionState’ is specific to user specific data that can be accessed across all pages in the web application.

**What are escape characters?**

Escape characters (Backslash) is used when working with special characters like single quotes, double quotes, apostrophes and ampersands. Place backslash before the characters to make it display.

Example:

|  |  |
| --- | --- |
|  | document.write "I m a "good" boy"  document.write "I m a \"good\" boy" |

**Local storage and session storage**: local storage store the in local for and do no delete until you are not deleting, session storage is deleting the data once user close the tab or browser session. Both are saving the data in local only, cookies are saving data in server and have storage less than local storage/session storage, both are supports modern browser and but is not working old browser.

## Cookies

### Pros

* Legacy support (it's been around forever)
* Persistent data
* Expiration dates

### Cons

* Each domain stores all its cookies in a single string, which can make parsing data difficult
* Data is unencrypted, which becomes an issue because...
* ... though small in size, cookies are sent with every HTTP request
* Limited size (4KB)
* SQL injection can be performed from a cookie

## Local storage

### Pros

* Support by most modern browsers
* Persistent data that is...
* ... stored directly in the browser
* Same-origin rules apply to local storage data
* Is **not** sent with every HTTP request
* ~5MB storage per domain (that's 5120KB)

### Cons

* Not supported by anything before:
  + IE 8
  + Firefox 3.5
  + Safari 4
  + Chrome 4
  + Opera 10.5
  + iOS 2.0
  + Android 2.0
* If the server needs stored client information you purposefully have to send it. [3]

## Conclusion

Based on my comparison I think that local storage is the logical successor of cookies for storing client-side data. While cookies were originally created as a work-around, local storage has been designed with a purpose (plus you have [session storage](http://dev.w3.org/html5/webstorage/#the-sessionstorage-attribute) and [database storage](http://www.webreference.com/authoring/languages/html/HTML5-Client-Side/) availabe as well.)

Local storage is easier to work with because you don't have to parse a string to get data. Instead you can make a call for the variable name and it's value is returned. The same applies with creating and deleting data as well.

For saving client-side data I believe local storage wins. As someone on the stackoverflow boards pointed out however, if your server needs information from a client cookies will be a better solution. But if you just want to save user-prefs, num vists, or other stuff like that choose local storage.

Until everyone on the Web starts using HTML5-compatible browsers cookies will still be king.

## sessionStorage Object

The sessionStorage object is equal to the localStorage object, **except** that it stores the data for only one session. The data is deleted when the user closes the specific browser tab.

The following example counts the number of times a user has clicked a button, in the current session:

**Explain what is pop()method in JavaScript?**

The pop() method is similar as the shift() method but the difference is that the Shift method works at the start of the array.  Also the pop() method take the last element off of the given array and returns it. The array on which is called is then altered.  
Example:  
var cloths = [“Shirt”, “Pant”, “TShirt”];  
cloths.pop();//Now cloth becomes Shirt,Pant

var cloths = [“Shirt”, “Pant”, “TShirt”];  
cloths.shift() // Pant,shirt

**Write the point of difference between web-garden and a web-farm?**

Both web-garden and web-farm are web hosting systems. The only difference is that web-garden is a setup that includes many processors in a single server while web-farm is a larger setup that uses more than one server.

## Pseudo Classes

:active a:active Selects the active link

:checked input:checked Selects every checked <input> element

:disabled input:disabled Selects every disabled <input> element

:empty p:empty Selects every <p> element that has no children

:enabled input:enabled Selects every enabled <input> element

:first-child p:first-child Selects every <p> elements that is the first child of its parent

:first-of-type p:first-of-type Selects every <p> element that is the first <p> element of its parent

:focus input:focus Selects the <input> element that has focus

:hover a:hover Selects links on mouse over

:in-range input:in-range Selects <input> elements with a value within a specified range

:invalid input:invalid Selects all <input> elements with an invalid value

**All CSS Pseudo Elements**

::after p::after Insert something after the content of each <p> element

::before p::before Insert something before the content of each <p> element

::first-letter p::first-letter Selects the first letter of each <p> element

::first-line p::first-line Selects the first line of each <p> element

::selection p::selection Selects the portion of an element that is selected by a user

## Reduce Activity in Loops

l = arr.length;   
for (i = 0; i < l; i++) { //Don’t add arr.length in for loop.

## 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:

Example

obj = document.getElementById("demo");  
obj.innerHTML = "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:

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

With this:

document.getElementById("demo").innerHTML = firstName + " " + lastName

## What is JSON?

* JSON stands for **J**ava**S**cript **O**bject **N**otation
* JSON is lightweight data interchange format
* JSON is language independent **\***
* JSON is "self-describing" and easy to understand

\* The JSON syntax is derived from JavaScript object notation syntax, but the JSON format is text only. Code for reading and generating JSON data can be written in any programming language.

## AngularJS Filters

AngularJS provides filters to transform data:

* currency Format a number to a currency format.
* date Format a date to a specified format.
* filter Select a subset of items from an array.
* json Format an object to a JSON string.
* limitTo Limits an array/string, into a specified number of elements/characters.
* lowercase Format a string to lower case.
* number Format a number to a string.
* orderBy Orders an array by an expression.
* uppercase Format a string to upper case.