Free Salesforce JS-Dev-101 Practice Questions

Shared by George on 13-08-2026

For More Free Questions and Preparation Resources

Check the Links on Last Page

Question 1

Question Type: MultipleChoice

Given the following code:

01 let x = null;

02 console.log(typeof x);

What is the output of line 02?

Options:

A- 'object'
B- 'undefined'
C- 'x'
D- 'null'

Answer:

A

Explanation:

This question focuses on the typeof operator and how JavaScript represents the type of the value null.

Declaration on line 1:

let x = null;

Here:

x is explicitly assigned the value null.

In JavaScript, null is a special primitive value that represents ''no value'' or ''empty value''.

Line 2:

console.log(typeof x);

The typeof operator returns a string indicating the type of the operand. For example:

typeof 1 returns 'number'

typeof 'hello' returns 'string'

typeof true returns 'boolean'

typeof undefined returns 'undefined'

typeof {} returns 'object'

typeof [] returns 'object' (arrays are objects)

typeof function() {} returns 'function'

For null, due to an early design decision in JavaScript, the result is:

typeof null === 'object'

This is a known historical quirk of the language:

Although null is a primitive and conceptually means ''no object'', typeof null returns the string 'object'.

Therefore, typeof x when x is null evaluates to 'object'.

So on line 2, the value printed is:

'object'

This matches option A.

The incorrect options are:

B . 'undefined': would be typeof x if x were undefined, not null.

C . 'x': typeof returns a string describing the type of the value, not the variable name.

D . 'null': although intuitive, JavaScript does not return 'null' for typeof null; it returns 'object' instead.

Therefore, the correct answer is:

Answe r: A

JavaScript knowledge / study guide reference concepts:

typeof operator and its return values

Primitive types in JavaScript (null, undefined, number, string, boolean, symbol, bigint)

Historical behavior: typeof null returning 'object'

Difference between null and undefined in JavaScript

Question 2

Question Type: MultipleChoice

```
function myFunction() {
a = a + b;
var b = 1;
}
myFunction();
console.log(a);
console.log(b);
```

Which statement is correct?

Options:

A- Line 02 throws a reference error, therefore line 03 is never executed.
B- Both line 02 and 03 are executed, but the values printed are undefined.
C- Both line 02 and 03 are executed, and the variables are hoisted.
D- Line 08 outputs the variable, but line 09 throws an error.

Answer:

D

Explanation:

Inside myFunction:

var b is hoisted; initially b is undefined within the function.

Expression a = a + b;:

b is undefined.

a is looked up in the global scope. In non-strict mode, reading an undeclared global gives undefined, then a = a + b becomes a = undefined + undefined NaN, and assigns global a.

After myFunction(), globally:

a exists and is NaN.

b is not declared globally; the var b was local.

So:

console.log(a); // logs NaN (executed fine)

console.log(b); // ReferenceError: b is not defined

Question 3

Question Type: MultipleChoice

Given the code below:

```
01 function Person() {
02 this.firstName = 'John';
03 }
04
05 Person.proto = {
06 job: x => 'Developer'
07 });
08
09 const myFather = new Person();
10 const result = myFather.firstName + ' ' + myFather.job();
```

What is the value of result when line 10 executes?

Options:

A- Error: myFather.job is not a function
B- undefined Developer
C- John Developer
D- John undefined

Answer:

A

Explanation:

Person.proto is being set, but JavaScript uses Person.prototype for the prototype chain, not Person.proto.

Therefore, job is not on Person.prototype, and instances of Person do not have job via prototype.

myFather is created with new Person(), so:

myFather.firstName is 'John'.

myFather.job is undefined.

Attempting to call myFather.job() results in:

TypeError: myFather.job is not a function

So option A is correct.

Question 4

Question Type: MultipleChoice

Which three actions can the code execute in the browser console?

Options:

A- Run code that is not related to the page.
B- View and change security cookies.
C- Display a report showing the performance of a page.
D- View, change, and debug the JavaScript code of the page.
E- View and change the DOM of the page.

Answer:

A, D, E

Explanation:

Comprehensive and Detailed Explanation From Exact Extract JavaScript Knowledge

The browser console (Developer Tools Console) allows:

Running code unrelated to the page

Developers can type arbitrary JavaScript not tied to the page's logic.

This is allowed A is correct.

Viewing and modifying JavaScript code execution

Using the Sources tab and console, developers can:

Inspect variables

Modify code at runtime

Set breakpoints

Therefore D is correct.

Viewing and manipulating the DOM

From the console, developers can:

Query DOM nodes using selectors

Modify element properties and attributes

Insert or remove elements

Therefore E is correct.

Why the other options are incorrect:

B (security cookies)

Many cookies have flags such as HttpOnly and Secure.

Cookies with the HttpOnly flag cannot be viewed or modified via JavaScript or console.

Therefore this is not guaranteed incorrect.

C (performance report)

The console itself does not provide a performance report.

The Performance or Lighthouse tabs provide this, not the console.

Therefore incorrect.

JavaScript Knowledge Reference (text-only)

Developer console can execute arbitrary JavaScript.

Console allows DOM inspection and manipulation.

Security cookies (HttpOnly) cannot be accessed via JavaScript.

==========

Question 5

Question Type: MultipleChoice

A developer has an irresponsible module that contains multiple functions.

What kind of export should be leveraged so that multiple functions can be used?

Options:

A- multi
B- default
C- all
D- named

Answer:

D

Question 6

Question Type: MultipleChoice

Refer to the code below:

```
01 let o = {
02 get js() {
03 let city1 = String('St. Louis');
04 let city2 = String('New York');
05
```

```
06 return {
07 firstCity: city1.toLowerCase(),
08 secondCity: city2.toLowerCase(),
09 }
10 }
11 }
```

What value can a developer expect when referencing o.js.secondCity?

Options:

A- undefined
B- An error
C- The developer missed the option --save when adding the dependency.
D- 'new york'

Answer:

D

Explanation:

Comprehensive and Detailed Explanation From Exact Extract JavaScript Knowledge

1. Getter Functions in JavaScript

In JavaScript, when an object uses the get keyword, it defines a getter method. Accessing a getter property executes the function and returns its value. Thus:

o.js

does not return the getter function; instead, it executes the function located at:

get js() { ... }

and returns the object inside the return block.

---

2. Behavior of String() and toLowerCase()

Inside the getter:

let city1 = String('St. Louis');

let city2 = String('New York');

String() creates a string value.

Then, the returned object is constructed as:

{

firstCity: city1.toLowerCase(),

secondCity: city2.toLowerCase(),

}

The method toLowerCase() is a standard JavaScript string method that returns a new string with all alphabetic characters converted to lowercase.

Therefore:

city2.toLowerCase()

returns:

'new york'

---

3. Referencing the Property

When the developer writes:

o.js.secondCity

the following happens:

The getter js runs and returns an object.

The returned object includes the property:

secondCity: 'new york'

Accessing .secondCity retrieves the lowercase string 'new york'.

Therefore, the correct value is 'new york'.

Why the Other Options Are Incorrect

A . undefined -- incorrect because the property secondCity clearly exists in the returned object.

B . An error -- incorrect because no invalid operations occur; all methods and properties are valid.

C . 'New York' -- incorrect because toLowerCase() transforms the string to lowercase.

JavaScript Knowledge Reference (Text-Based)

Getter methods using the get keyword return computed values when accessed.

JavaScript String values support the toLowerCase() method, which returns a lowercase version of the original string.

Accessing nested properties like o.js.secondCity triggers the getter, returning the constructed object.

Question 7

Question Type: MultipleChoice

Refer to the code:

01 const exec = (item, delay) =>

02 new Promise(resolve => setTimeout(() => resolve(item), delay));

03

04 async function runParallel() {

05 const [result1, result2, result3] = await Promise.all(

06 [exec('x', '100'), exec('y', '500'), exec('z', '100')]

07);

08 return `parallel is done: ${result1}${result2}${result3}`;

09 }

Which two statements correctly execute runParallel()?

Options:

A- async runParallel().then(data);
B- runParallel().then(function(data){
return data;
});
C- runParallel().done(function(data){

return data;
});
D- runParallel().then(data);

Answer:

B, D

Explanation:

Comprehensive and Detailed Explanation From Exact Extract JavaScript Knowledge

Facts about JavaScript Promises:

runParallel() is declared async, which means it always returns a Promise.

Promises are consumed using .then(), .catch(), and .finally().

There is no .done() method in native JavaScript Promises.

The async keyword cannot be placed before a function call (option A is invalid syntax).

Analysis of each option:

A . async runParallel().then(data);

Invalid syntax. async cannot prefix an expression.

B . Valid. Calls the function and attaches a .then() handler.

C . Invalid. .done() is not part of JavaScript Promise API.

D . Valid. Calls runParallel() and chains .then().

Therefore the correct answers are B and D.

JavaScript Knowledge Reference (text-only)

async functions return Promises.

Promises use .then() to retrieve resolved values.

.done() is not part of the standard Promise interface.

==========

Question 8

Question Type: MultipleChoice

Refer to the code below:

01 const server = require('server');

02

03 // Insert code here

A developer imports a library that creates a web server. The imported library uses events and callbacks to start the server.

Which code should be inserted at line 03 to set up an event and start the web server?

Options:

A- server.on('connect', (port) => { console.log('Listening on', port); });
B- server.start();
C- server((port) => { console.log('Listening on', port); });
D- server();

Answer:

C

Explanation:

Comprehensive and Detailed Explanation From Exact Extract JavaScript Knowledge:

The question specifies:

The library ''uses events and callbacks to start the server''.

We want to ''set up an event and start the web server''.

Option C:

server((port) => { console.log('Listening on', port); });

This:

Calls server(...), which (by design in such libraries) typically starts the web server.

Passes a callback function that receives port when the server is ready.

Inside the callback, it logs 'Listening on '.

This matches the ''events and callbacks'' description: the callback is invoked when the server has started listening.

Why others are incorrect:

A . server.on('connect', ...)

Assumes server is an event emitter instance with a 'connect' event, which is not given. Also, this line alone would not start the server.

B . server.start();

Assumes a start method exists; the question says the library ''uses events and callbacks to start,'' implying invocation with a callback, not a plain start() call.

D . server();

Might start the server, but does not ''set up an event'' or callback to know when or where it is listening.

Therefore, C aligns with the described usage.

Concepts: callback-based APIs, starting servers with callbacks, event/callback style vs simple method calls.

Question 9

Question Type: MultipleChoice

A developer is setting up a Node.js server and is creating a script at the root of the source code, index.js, that will start the server when executed. The developer declares a variable that needs the folder location that the code executes from.

Which global variable can be used in the script?

Options:

A- __filename
B- window.location
C- this.path
D- __dirname

Answer:

D

Explanation:

Comprehensive and Detailed Explanation From Exact Extract JavaScript Knowledge:

In Node.js:

__filename is the absolute path to the current file (including file name).

__dirname is the absolute path to the directory that contains the current file.

window.location is a browser-only global, not available in Node.js.

this.path is not a standard Node.js global for path information.

The requirement is:

''the folder location that the code executes from.''

That is exactly what __dirname provides.

So the correct choice is __dirname.

Concepts: Node.js globals, __filename, __dirname, Node vs browser environment.

_____

Question 10

Question Type: MultipleChoice

Refer to the code below:

let inArray = [[1, 2], [3, 4, 5]];

Which two statements result in the array [1, 2, 3, 4, 5]?

(With corrected typing errors: usArray inArray, ...)

Options:

A- [].concat(...inArray);
B- [].concat.apply(inArray, []);
C- [].concat::...inArray();

D- [].concat.apply({}, inArray);

Answer:

A, D

Explanation:

Comprehensive and Detailed Explanation From Exact Extract JavaScript knowledge:

We start with the array:

let inArray = [[1, 2], [3, 4, 5]];

This is an array of two inner arrays:

First element: [1, 2]

Second element: [3, 4, 5]

The desired result is to transform this into a single, flat array:

[1, 2, 3, 4, 5]

This is accomplished by using Array.prototype.concat to concatenate the inner arrays into one new array, optionally combined with the spread syntax (...) or Function.prototype.apply.

Option A: [].concat(...inArray);

Relevant concepts:

Spread syntax (...inArray) expands an iterable into separate arguments.

Array.prototype.concat joins arrays or values into a new array. When you pass arrays as arguments to concat, it flattens them one level into the result.

Step-by-step behavior:

inArray is [[1, 2], [3, 4, 5]].

...inArray expands into [1, 2] and [3, 4, 5] as separate arguments.

So [].concat(...inArray) is equivalent to:

[].concat([1, 2], [3, 4, 5]);

concat processes each argument:

For [1, 2], it adds 1 and 2 into the result array.

For [3, 4, 5], it adds 3, 4, and 5 into the result array.

The final outcome is:

[1, 2, 3, 4, 5]

Therefore, Option A correctly produces the desired array.

Option B: [].concat.apply(inArray, []);

Corrected name from usArray to inArray.

Relevant concepts:

Function.prototype.apply(fnThis, argsArray) invokes a function with a specific this value and a list of arguments passed as an array.

Here:

thisArg is inArray.

argsArray is [] (no actual arguments passed).

So the call:

[].concat.apply(inArray, []);

is equivalent to:

Array.prototype.concat.apply(inArray, []);

// i.e. inArray.concat();

Since there are no extra arguments, inArray.concat() simply returns a shallow copy of inArray:

[[1, 2], [3, 4, 5]]

This remains an array of arrays and is not flattened into [1, 2, 3, 4, 5]. Therefore, Option B does not produce the required result.

Option C: [].concat::...inArray();

Corrected name from usArray to inArray and .. to ....

This expression uses syntax that is not part of standard JavaScript:

:: (double colon) was proposed in early drafts as a bind operator but is not part of the official ECMAScript standard.

The combination concat::...inArray() is invalid in normal JavaScript engines and cannot be used as a valid way to flatten arrays.

In addition, inArray() would imply calling inArray as a function, but it is an array, which would cause a runtime error.

Hence, Option C is syntactically or semantically invalid in standard JavaScript and does not provide the required result [1, 2, 3, 4, 5].

Option D: [].concat.apply({}, inArray);

Corrected name from usArray to inArray.

Relevant concepts:

Again, Function.prototype.apply is used to call concat with a specific this value and arguments provided as an array.

concat treats its this value as an array-like object but ultimately returns a new array containing concatenated elements.

In this expression:

[].concat.apply({}, inArray);

thisArg is {} (an empty object).

argsArray is inArray, which is [[1, 2], [3, 4, 5]].

Using apply, this is interpreted as calling concat like:

Array.prototype.concat.call({}, [1, 2], [3, 4, 5]);

concat then:

Starts from the array-like this (here {} is treated as an empty base).

Takes the first argument [1, 2] and appends its elements 1 and 2 to the result.

Takes the second argument [3, 4, 5] and appends its elements 3, 4, and 5 to the result.

The resulting new array is:

[1, 2, 3, 4, 5]

Therefore, Option D also correctly produces the required array.

Final evaluation of all options:

Option A: Uses spread syntax with concat and correctly flattens one level: [1, 2, 3, 4, 5].

Option B: Equivalent to inArray.concat(), leaving it as [[1, 2], [3, 4, 5]]. Does not flatten the structure.

Option C: Uses non-standard and invalid syntax; not a valid or correct JavaScript solution.

Option D: Uses apply with concat, passing the inner arrays as arguments and flattening one level to [1, 2, 3, 4, 5].

Thus, the two correct statements are:

Reference of JavaScript knowledge documents or Study Guide (concept names only):

Array.prototype.concat (concatenating and one-level flattening behavior)

Spread syntax for arrays (...array)

Function.prototype.apply (calling a function with a given this value and arguments list)

Array flattening by one level using concat with array arguments

Distinction between nested arrays and flat arrays in JavaScript

Question 11

Question Type: MultipleChoice

Refer to the HTML below:

1. Leo
2. Tony
3. Tiger

Which JavaScript statement results in changing "Leo" to "The Lion"?

Options:

A- document.querySelectorAll('#main #Leo').innerHTML = 'The Lion';
B- document.querySelector('#main li:second-child').innerHTML = 'The Lion';
C- document.querySelectorAll('#main li,Leo').innerHTML = 'The Lion';
D- document.querySelector('#main li:nth-child(1)').innerHTML = 'The Lion';

Answer:

D

Explanation:

Comprehensive and Detailed Explanation From Exact Extract JavaScript Knowledge:

We want to select the first <li> under #main and change its content.

document.querySelector(selector) returns the first element matching the CSS selector.

In the DOM, the <li> elements under <ul> are children in order:

1st child: 'Leo'

2nd child: 'Tony'

3rd child: 'Tiger'

li:nth-child(1) is the CSS pseudo-class that selects the first <li> inside its parent.

So:

document.querySelector('#main li:nth-child(1)').innerHTML = 'The Lion';

selects the first <li> ('Leo') inside #main and sets its innerHTML to 'The Lion'.

Why others are incorrect:

A . #main #Leo

There is no element with id 'Leo'. id is not set on the <li>, so the selector matches nothing.

B . li:second-child

There is no :second-child pseudo-class. The valid pseudo-class is :nth-child(2); and that would select 'Tony', not 'Leo'.

C . '#main li,Leo'

This selector means ''all #main li and all elements named <Leo>''.

There is no <Leo> tag, and querySelectorAll returns a NodeList, which does not have a single innerHTML property.

Therefore, D is the correct statement.

Study Guide Concepts:

document.querySelector vs querySelectorAll

CSS selectors: nth-child()

DOM element innerHTML property

Question 12

Question Type: MultipleChoice

A developer implements a function that adds a few values.

```javascript
function sum(num1, num2, num3) {
if (num3 === undefined) {
num3 = 0;
}
return num1 + num2 + num3;
}
```

Which three options can the developer invoke for this function to get a return value of 10?

Select 3 answers

Options:

A- sum(5)(5);
B- sum()(10);
C- sum(5,5,)();
D- sum(10,();
E- sum()(5)(5);

Answer:

A, B

Question 13

Question Type: MultipleChoice

Given a value, which two options can a developer use to detect if the value is NaN?

Options:

A- value === Number.NaN
B- value == NaN
C- isNaN(value)
D- Object.is(value, NaN)

Answer:

C, D

Explanation:

Comprehensive and Detailed Explanation From Exact Extract JavaScript knowledge:

We already know NaN is special: it is not equal to itself.

Check each:

A . value === Number.NaN

Number.NaN is NaN.

NaN === NaN is always false.

This will never be true; cannot reliably detect NaN.

B . value == NaN

NaN == NaN is also always false.

Again, this never detects NaN.

C . isNaN(value)

Global isNaN converts its argument to a number and then checks if the result is NaN.

This can detect NaN, but it may also return true for non-number values that coerce to NaN, such as isNaN('foo').

Regardless, it is a standard way to detect if a value is ''NaN-like'' in JavaScript.

D . Object.is(value, NaN)

Object.is(NaN, NaN) returns true.

This is a strict way to detect a value that is exactly NaN (no coercion).

Therefore, among the given choices, the two viable ways to detect NaN are:

isNaN(value)

Object.is(value, NaN)

P2P exams