DUMPSBOSS.

Salesforce Certified JavaScript Developer I

Salesforce JavaScript-Developer-I

Version Demo

Total Demo Questions: 48

Total Premium Questions: 484

Buy Premium PDF

https://dumpsboss.co

support@dumpsboss.co

support@dumpsboss.co
dumpsboss.co

Topic Break Down

Topic	No. of Questions
Topic 1, Variables, Types, and Collections	120
Topic 2, Objects, Functions, and Classes	123
Topic 3, Browser and Events	80
Topic 4, Debugging and Error Handling	50
Topic 5, Asynchronous Programming	55
Topic 6, Server-Side JavaScript	30
Topic 7, Testing	23
Topic 8, Mix Questions	3
Total	484

QUESTION NO: 1

Given:

const str = ' Salesforce ' ;

Which two statements result in ' Sales ' ?

A. str.substring(0, 5);

B. str.substring(1, 5);

C. str.substr(0, 5);

D. str.substr(1, 5);

E. str.substring(1, 6);

ANSWER: D E

Explanation:

The string literal contains a leading space before `Salesforce`, so the character `S` is at index `1`. Interpreting the spaces around `Sales` in the question as typographical padding, the intended result is the five-character string `Sales`. `str.substring(1, 6);` produces that value because `substring()` takes a start index and an exclusive end index; it includes characters at indexes `1` through `5`. `str.substr(1, 5);` also produces `Sales` because it begins at index `1` and requests five characters. Although `substr()` is a legacy feature and new code should generally use `substring()` or `slice()`, it remains supported in JavaScript environments and has the stated start-and-length behavior. See the MDN references for String.prototype.substring() and String.prototype.substr().

QUESTION NO: 2

A developer has an ErrorHandler module that contains multiple functions.

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

A. all

B. named

C. multi

D. default

ANSWER: B

Explanation:

The correct choice is **named**. A JavaScript module can expose several independently usable functions through named exports. For example, an ErrorHandler module could declare `export function logError()`, `export function formatError()`, and `export function notifyUser()`. A consuming module then imports precisely the functions it needs using their exported names, such as `import { logError, formatError } from './errorHandler'`. This approach is particularly appropriate for a utility-style module containing multiple related operations because it creates a clear, explicit public API and supports selective imports. Named exports are standard ECMAScript module syntax and are supported in Salesforce Lightning Web Components for sharing JavaScript code between components. Salesforce recommends placing reusable JavaScript in modules and importing exported members where needed. For syntax and behavior details, see the MDN documentation on JavaScript modules and Salesforce's Share JavaScript Code guide.

QUESTION NO: 3

A developer wants to create a simple image upload using the File API.

HTML:

< input type= " file " onchange= " previewFile() " >

< img src= " " 200 " alt= " Image preview... " / >

JavaScript:

01 function previewFile() {

02 const preview = document.querySelector(' img ');

03 const file = document.querySelector(' input[type=file] ').files[0];

04 // line 4 code

05 reader.addEventListener(" load " , () = > {

06 preview.src = reader.result;

07 }, false);

08 // line 8 code

09 }

Which code in lines 04 and 08 allows the selected local image to be displayed?

A. 04 const reader = new File();08 if (file) reader.readAsDataURL(file);

B. 04 const reader = new FileReader();08 if (file) reader.readAsDataURL(file);

C. 04 const reader = new FileReader();08 if (file) URL.createObjectURL(file);

ANSWER: B

Explanation:

`04 const reader = new FileReader();` followed by `08 if (file) reader.readAsDataURL(file);` correctly implements an asynchronous local-image preview using the browser File API. `FileReader` is the browser-provided interface for reading the contents of a `File` object obtained from an `<input type="file">` element. The selected item is available through `files[0]`, and checking `if (file)` safely ensures that a file was actually selected before starting the read operation.

`readAsDataURL(file)` reads the image and, when reading completes, produces a data URL in the reader's `result` property. The registered `load` listener runs after that asynchronous operation has completed, so assigning `reader.result` to `preview.src` gives the image element a usable URL representing the selected local image. This approach requires no upload to a server; it is solely a client-side preview. Salesforce JavaScript developers should recognize this standard web-platform API behavior when building browser-side interactions in Lightning web components or other JavaScript UI code. See the MDN documentation for FileReader and readAsDataURL().

QUESTION NO: 4

A developer is setting up a new Node.js server with a client library that is built using events and callbacks.

The library:

● Will establish a web socket connection and handle receipt of messages to theserver

● Will be imported with require, and made available with a variable called we.

The developer also wants to add error logging if a connection fails.

Given this info, which code segment shows the correct way to set up a client with two events that listen at execution time?

A. ws.connect (( ) => {console.log('connected to client'); }).catch((error) => { console.log('ERROR' , error); }};

B. ws.on ('connect', ( ) => { console.log('connected to client'); }); ws.on('error', (error) => { console.log('ERROR' , error); });

C. try{ws.connect(( ) => {console.log('connected to client'); });} catch(error) { console.log('ERROR' ,error); };}

ANSWER: B

Explanation:

`ws.on('connect', () => { console.log('connected to client'); }); ws.on('error', (error) => { console.log('ERROR', error); });` is correct because an event-based Node.js client exposes an event-emitter interface, and listeners are registered with `on()` before the relevant events occur. The `connect` listener runs when the client establishes its connection and can perform connection-success handling. The `error` listener is registered independently so that it can receive an error emitted by the client during connection or later communication, then log the supplied error object. Registering both listeners during setup ensures the application is ready to respond when either event is emitted at runtime; neither event listener depends on the other listener having run first. This follows Node.js's EventEmitter model: `emitter.on(eventName, listener)` attaches a callback for a named event, and event emitters use the `'error'` event to report errors. Node also documents that an emitted `'error'` event requires an error listener to prevent it from being treated as an unhandled error. See the Node.js EventEmitter documentation and the Node.js guidance for error events.

QUESTION NO: 5

Which two options are core Node.js modules? (Choose two.)

A. http

B. worker_threads

C. exception

D. iostream

ANSWER: A B

Explanation:

`http` is a core Node.js module. It provides the APIs used to create HTTP servers and clients, including `http.createServer()`, request methods, headers, status codes, and streaming message objects. Because it is built into the Node.js runtime, an application can import it directly using `require('node:http')` or `import http from 'node:http'`; it does not need to be installed from npm. The official API reference documents its server and client functionality at Node.js HTTP documentation.

`worker_threads` is also a core Node.js module. It supplies the Worker API for running JavaScript in separate threads within a Node.js process, allowing CPU-intensive work to be moved away from the main event-loop thread. The module exposes facilities such as `Worker`, `parentPort`, and `workerData`, and is imported with the built-in `node:worker_threads` specifier. Node.js documents this built-in module and its threading model at Node.js Worker Threads documentation. The option text has been clarified to use the module's actual built-in name, `worker_threads`.

QUESTION NO: 6

Refer to the code below:

Const searchTest = 'Yay! Salesforce is amazing!" ;

Let result1 = searchText.search(/sales/i);

Let result 21 = searchText.search(/sales/i);

console.log(result1);

console.log(result2);

After running this code, which result is displayed on the console?

A. > true > false

B. > 5 >undefined

C. > 5 > -1

D. > 5 > 0

E. No output is displayed; the code throws a SyntaxError.

ANSWER: E

Explanation:

No output is displayed because the code cannot be parsed as valid JavaScript. JavaScript keywords are case-sensitive, so the declaration keywords must be written as `const` and `let`, rather than `Const` and `Let`. In addition, `result 21` is not a valid variable identifier: whitespace separates tokens, and an identifier cannot be declared in that form. These syntax issues prevent the JavaScript engine from executing any statement, including either `console.log()` call. A syntax error occurs during parsing, before runtime evaluation begins; therefore, there can be no displayed values such as an index, `undefined`, or `-1`. JavaScript's lexical grammar defines identifiers and reserved keywords, including the case-sensitive declaration keywords used here. See the MDN JavaScript lexical grammar reference and the MDN documentation for let declarations.

QUESTION NO: 7

Refer to the following code block (with corrected template literals using backticks):

```
01 class Animal {
02 constructor(name) {
03 this.name = name;
04 }
05
06 makeSound() {
07 console.log(`${this.name} is making a sound.`);
08 }
09 }
10
11 class Dog extends Animal {
12 constructor(name) {
13 super(name);
14 this.name = name;
15 }
16 makeSound() {
```

```
17 console.log(`${this.name} is barking.`);
18 }
19 }
20
21 let myDog = new Dog( ' Puppy ' );
22 myDog.makeSound();
```

What is the console output?

A. > Uncaught ReferenceError

B. > Undefined

C. > Puppy is barking.

D. > `Puppy is barking.`

ANSWER: D

Explanation:

`The console logs ` Puppy is barking.` (one leading space and two spaces between Puppy and is).` is correct because `Dog` is a subclass of `Animal`, and its constructor calls `super(name)` before using `this`. That parent-constructor call initializes the inherited instance with the supplied `name`. The subsequent assignment in the `Dog` constructor leaves the same value in place.

When `myDog.makeSound()` runs, JavaScript dispatches to the `makeSound()` method defined on `Dog`, since that method overrides the inherited method. Its template literal interpolates `this.name` and then appends the literal text `is barking.`. Crucially, the argument is written as `' Puppy '`; the spaces inside a JavaScript string literal are part of the value and are not trimmed. Therefore, the leading space in the name remains, and the name's trailing space combines with the space before `is` in the template literal, producing two spaces at that point. See MDN's documentation for class inheritance with extends and template literals.

QUESTION NO: 8

A developer implements and calls the following code whenan application state change occurs:

```
Const onStateChange =innerPageState) => {
window.history.pushState(newPageState, ' ', null);
}
```

If the back button is clicked after this method is executed, what can a developer expect?

A. A navigate event is fired with astate property that details the previous application state.

B. The page isnavigated away from and the previous page in the browser's history is loaded.

C. The page reloads and all Javascript is reinitialized.

D. A popstate event is fired with a state property that details the application's last state.

ANSWER: D

Explanation:

A `popstate` event is fired with a state property that details the application's last state. Calling `window.history.pushState(newPageState, '', null)` creates a new session-history entry for the current document and associates `newPageState` with that entry. It does not navigate to a new document or reload the page. When the user subsequently selects the browser Back button, the browser activates the preceding history entry. As part of that history traversal, the browser dispatches a `popstate` event on `window`. The event's `state` property contains the state object associated with the entry that has just become active. In a single-page application, this lets the application restore its prior UI state, such as the previously selected view, filters, or record context, without requiring a document load. The handler should therefore listen for `popstate` and use `event.state` to reconcile the rendered interface with the restored history entry. The History API documentation describes that `pushState()` adds an entry and that `popstate` is dispatched when the active history entry changes: MDN: History.pushState() and MDN: popstate event.

QUESTION NO: 9

What are two unique features of functions defined with a fat arrow as compared to normal function definition?

Choose 2 answers

A. The function generated its own this making it useful for separating the function's scope from its enclosing scope.

B. If the function has a single expression in the function body, the expression will be evaluated and implicitly returned.

C. The function uses the this from the enclosing scope.

ANSWER: B C

Explanation:

Arrow functions lexically capture `this` from the surrounding execution context. Consequently, `this` inside an arrow function remains the value from the scope where the function was created, rather than being determined by how the function is invoked. This is especially useful in callbacks, such as event handlers or asynchronous code, where a developer needs to retain access to the enclosing component or object context without manually saving it in another variable or applying `bind()`. Arrow functions also support a concise expression body. When the body consists of one expression and braces are omitted, JavaScript evaluates that expression and returns its result automatically. For example, `const double = value => value * 2;` returns the computed value without requiring a `return` statement. These behaviors are defined characteristics of arrow-function syntax and distinguish it from ordinary function expressions and declarations. For further details, see the MDN references on arrow functions and concise function bodies and implicit returns.

QUESTION NO: 10

A developer is creating a simple webpage with a button. When a userclicks this button

for the first time, a message is displayed.

The developer wrote the JavaScript code below, but something is missing. The

message gets displayed every time a user clicks the button, instead of just the first time.

01 functionlisten(event) {

02 alert ('Hey! I am John Doe') ;

03 button.addEventListener ('click', listen);

Which two code lines make this code work as required?

Choose 2 answers

A. On line 02, use event.first to test if it is the first execution.

B. On line 04, use event.stopPropagation().

C. On line 04, use button.removeEventListener('click', listen);

D. On line 03, add the { once: true } option to button.addEventListener().

ANSWER: C D

Explanation:

`button.removeEventListener('click', listen);` makes the handler temporary when it is placed in the listener after the alert. Event listeners are identified by the event type and the same listener function reference that was originally registered. Therefore, after the first click invokes `listen`, removing that exact `listen` function prevents it from being invoked by later clicks. This approach is useful when the application needs to decide dynamically, within the handler, that no further events should be processed.

Adding the `{ once: true }` listener option is the declarative browser-supported solution. For example, `button.addEventListener('click', listen, { once: true });` instructs the event target to automatically remove the listener after its first invocation. The alert is consequently shown on the first click only, with no explicit cleanup code required in `listen`. The `once` option is part of the standard `addEventListener()` options object and is appropriate for one-time interactions such as an introductory message. See the MDN addEventListener() reference and the MDN removeEventListener() reference.

QUESTION NO: 11

Refer to the code snippet below:

Let array = [1, 2, 3, 4, 4, 5, 4, 4];

For (let i =0; i < array.length; i++){

if (array[i] === 4) {

array.splice(i, 1);

}

}

What is the value of the array after the code executes?

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

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

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

D. [1, 2, 3, 5]

ANSWER: C

Explanation:

`[1, 2, 3, 4, 5, 4]` is correct because `splice(i, 1)` removes one element at the current index and shifts every subsequent element one position to the left. The loop initially reaches the first `4` at index 3 and removes it, leaving another `4` shifted into index 3. However, the loop then increments `i` to index 4, so that shifted value is not evaluated during the next iteration. The loop continues, reaches a later `4`, and removes it as well. After that removal, the final adjacent `4` shifts left into the index that has just been processed; the loop increments again and then terminates because the index is no longer less than the shortened array length. Consequently, two values equal to `4` remain: one from the initial adjacent pair and one from the final adjacent pair. JavaScript arrays are mutable, and `Array.prototype.splice()` changes the original array in place, which is why each removal affects both the array's contents and the indices considered by the loop. See the MDN splice() reference and the MDN for statement reference.

QUESTION NO: 12

const str = ' Salesforce ' ;

Which two statements result in the word " Sales " ?

A. str.substring(0, 5);

B. str.substr(s, 5);

C. str.substring(0, 5);

D. str.substr(0, 5);

E. str.substring(1, 6);

F. str.slice(1, 6);

ANSWER: E F

Explanation:

Because the string literal contains a leading space before `Salesforce`, character position `0` is the space and position `1` is `S`. To produce `Sales` exactly, the expression must begin at position `1` and include five characters: `S`, `a`, `l`, `e`, and `s`. `str.substring(1, 6)` uses an exclusive ending index, so it returns the characters from position `1` up to, but not including, position `6`. This yields `Sales`. `str.slice(1, 6)` also uses a start index and an exclusive end index, producing the same five-character result. These methods are appropriate when selecting a range of characters by index from a JavaScript string. See the MDN references for String.prototype.substring() and String.prototype.slice().

QUESTION NO: 13

Universal Containers (UC) just launched a new landing page, but users complain that the website is slow. A developer found some functions any that might cause this problem. To verify this, the developer decides to execute everything and log the time each of these three suspicious functions consumes.

Which function can the developer use to obtain the time spent by every one of the three functions?

```
01 console.time('Performance');
02
03 maybeAHeavyFunction();
04
05 thisCouldTakeTooLong();
06
07 orMaybeThisOne();
08
09 console.endTime('Performance');
```

A. console. timeLog ()

B. console.timeStamp ()

C. console.trace()

D. console.getTime ()

ANSWER: A

Explanation:

`console. timeLog ()` is the appropriate console API for reporting elapsed time from a timer that was previously started with `console.time()`. The developer can assign a distinct label to each suspected function's timer, start the labeled timer before executing that function, and call `console.timeLog(label)` when the function completes. The browser console then logs the elapsed duration associated with that label while leaving the timer active. This makes it useful when profiling several operations independently and, if needed, when recording intermediate timing checkpoints during a longer operation. For three suspicious functions, using a unique timer label for each ensures that the logged duration belongs to the intended function rather than to another operation. If the developer no longer needs a timer after recording its final duration, they can subsequently end and remove it with `console.timeEnd(label)`. This approach provides practical, lightweight timing information directly in browser developer tools, helping identify where page execution time is being spent. See the MDN console.timeLog() reference and the MDN console.time() reference.

QUESTION NO: 14

Given the following code:

01 counter = 0;

02 const logCounter = () = > {

03 console.log(counter);

04 };

05 logCounter();

06 setTimeout(logCounter, 2100);

07 setInterval(() = > {

08 counter++;

09 logCounter();

10 }, 1000);

What will be the first four numbers logged?

A. 0122

B. 0123

C. 0112

D. 0012

ANSWER: A

Explanation:

0122 is correct. The direct invocation of `logCounter()` occurs synchronously before either timer is scheduled to execute, so it logs the initial value, `0`. The repeating timer then invokes its callback after roughly 1,000 milliseconds. That callback increments `counter` before calling `logCounter()`, producing `1`. At roughly 2,000 milliseconds, the next interval callback again increments the same shared variable and logs `2`. The one-time timer was scheduled with a 2,100-millisecond delay, so it runs after the second interval callback but before the expected third interval callback at roughly 3,000 milliseconds. No increment occurs in the timeout callback; it simply calls `logCounter()`. Because the value remains `2` at that point, the fourth output is also `2`. The function closes over the `counter` binding, meaning each invocation reads its current value rather than preserving the initial value. Timer delays specify when callbacks become eligible to run; with normal execution here, their ordering yields `0`, `1`, `2`, `2`. See MDN: setTimeout() and MDN: setInterval().

QUESTION NO: 15

Given the code below:

```
01 function GameConsole (name) {
02 this.name = name;
03 }
04
05 GameConsole.prototype.load = function(gamename) {
06 console.log( ` $(this.name) is loading agame : $(gamename) …`);
07 )
08 function Console 16 Bit (name) {
09 GameConsole.call(this, name) ;
10 }
11 Console16bit.prototype = Object.create ( GameConsole.prototype) ;
12 //insert code here
13 console.log( ` $(this.name) is loading a cartridge game :$(gamename) …`);
14 }
15 const console16bit = new Console16bit(‘ SNEGeneziz ’);
16 console16bit.load(‘ Super Nonic 3x Force ’);
```

What should a developer insert at line 15 to output the following message using the

method ?

> SNEGeneziz is loading a cartridgegame: Super Monic 3x Force . . .

A. Console16bit.prototype.load(gamename) = function() {

B. Console16bit.prototype.load = function(gamename) {

C. Console16bit = Object.create(GameConsole.prototype).load = function(gamename) {

D. Console16bit.prototype.load(gamename) {

ANSWER: B

Explanation:

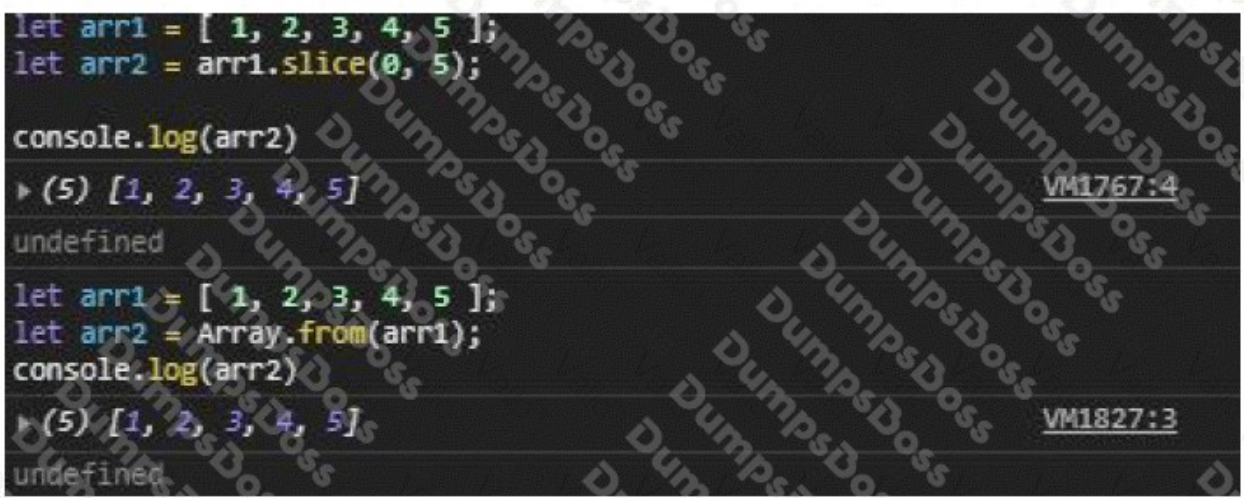
`Console16bit.prototype.load = function(gamename) {` correctly defines a `load` method on the prototype used by instances constructed with `new Console16bit(...)`. The assignment places a function under the `load` property of `Console16bit.prototype`, and that function receives the supplied game name through its `gamename` parameter. When `console16bit.load(...)` is invoked, JavaScript resolves the method through the instance's prototype and calls it with `this` bound to `console16bit`. Therefore, `this.name` accesses the name initialized when the constructor calls `GameConsole.call(this, name)`, while `gamename` holds the argument passed to `load`.

This is the standard prototype-based pattern for defining behavior shared by all instances of a constructor function. The earlier `Object.create(GameConsole.prototype)` assignment establishes inheritance from `GameConsole`; assigning `load` afterward creates the specialized cartridge-game implementation for `Console16bit`. Consequently, calling the method produces the intended message using the instance name and game-name argument. For details on constructor functions and prototype method lookup, see MDN's inheritance and prototype chain guide and MDN's reference for this.

QUESTION NO: 16

Refer to the following array:

Let arr1 = [1,2, 3, 4, 5];



Which two lines of code result in a second array, arr2 being created such that arr2 is not

a reference to arr1?

A. Let arr2 = arr1.slice(0, 5);

B. Let arr2 = Array.from(arr1);

C. Let arr2 = arr1;

D. Let arr2 = arr1.sort();

ANSWER: A B

Explanation:

`Let arr2 = arr1.slice(0, 5);` creates a new array containing the elements selected from `arr1`. With a start index of `0` and an end index of `5`, it copies all five elements into a distinct array object. Consequently, assigning, adding, or removing elements through `arr2` does not change the structure of `arr1`. The JavaScript `slice()` method is documented as returning a shallow copy rather than the original array: MDN Array.prototype.slice().

`Let arr2 = Array.from(arr1);` also constructs a new array object from the iterable source array. It copies the source array's elements into that newly created array, so `arr1` and `arr2` are separate array containers. `Array.from()` is specifically intended to create a new, shallow-copied array from an iterable or array-like value, as described in MDN Array.from(). Because this example contains primitive number values, the copied values are independent. Both techniques are shallow copies; if an array contained objects or nested arrays, those element objects would still be shared by reference.

QUESTION NO: 17

A developer is setting up a new Node.js server with a client library that is built using events and callbacks.

The library:

▪ Will establish a web socket connection and handle receipt of messages to the server ▪ Will be imported with require, and made available with a variable called ws.

The developer also wants to add error logging if a connection fails.

Given this information, which code segment shows the correct way to set up a client with two events that listen at execution time?

```
04 ws.on('connect', () => {
05  console.log('Connected to client');
06
07  ws.on('error', (error) => {
08   console.log('ERROR', error);
09  });
10 });
```

```
04 ws.connect(() => {
05  console.log('Connected to client');
06 }).catch((error) => {
07  console.log('ERROR', error);
08 });
```

```
04 ws.on('connect', () => {
05  console.log('Connected to client');
06 });
07
08 ws.on('error', (error) => {
09  console.log('ERROR', error);
10 });
```

```
04 try {
05  ws.connect(() => {
06   console.log('Connected to client');
07  });
08 } catch(error) {
09   console.log('ERROR', error);
10 }
```

A.

B. C.

D.

Answer: B

Explanation:

A. C.
D.

B. ws.on('open', () => { /* connection established */ }); ws.on('error', (error) => { console.error(error); });

ANSWER: B

Explanation:

The correct approach is to register callback functions with the WebSocket client using `ws.on(...)`, including a listener for a successful connection and a listener for connection errors. Node.js event-driven APIs use the EventEmitter pattern: calling `on()` subscribes a function to a named event, and Node invokes that function later when the event occurs. This lets the server continue executing without blocking while the connection is being established.

For a WebSocket client, the `open` event is the appropriate place to run code once the connection is successfully established. The `error` event callback receives the error object, so its message can be logged when a connection attempt or socket operation fails. Registering both listeners before relying on the connection ensures that the asynchronous outcomes are handled at execution time. This follows the standard event-listener model described in the Node.js Events documentation. The WebSocket client API likewise documents the use of `open` and `error` event handlers; see the ws API documentation.

QUESTION NO: 18 - (SIMULATION)

Refer the code below.

```
x=3.14;

function myfunction() {

"use strict";

y=x;

}

z=x;

myFunction();
```

ANSWER: See the explanation for the answer

Explanation:

Result: `z` is assigned `3.14`, then the script throws a `ReferenceError` at `myFunction();`.

JavaScript is case-sensitive. The function is declared as `myfunction` (lowercase `f`) but invoked as `myFunction` (uppercase `F`). Therefore, `myFunction` is not defined.

`x = 3.14;` creates/assigns the global `x`.
`z = x;` succeeds, so `z === 3.14`.
`myFunction();` fails because no function with that exact capitalization exists.

The strict-mode statement applies only inside the declared `myfunction` function. If the invocation were corrected to `myfunction();`, then `y = x;` would throw a `ReferenceError`, because strict mode does not allow assignment to an undeclared variable.

QUESTION NO: 19

A Developer wrote the following code to test a sum3 function that takes in an array of

numbers and returns the sum of the first three number in the array, The test passes:

```
Let res = sum2([1, 2, 3 ]) ;

console.assert(res === 6 );

Res = sum3([ 1, 2, 3, 4]);

console.assert(res=== 6);
```

A different developer made changes to the behavior of sum3 to instead sum all of the numbers

present in the array. The test passes:

Which two results occur when running the test on the updated sum3 function ?

Choose 2 answers

A. The line 02 assertion passes.

B. The line 02 assertion fails

C. The line 05 assertion passes.

D. The line 05 assertion fails.

ANSWER: A D

Explanation:

Assuming the apparent capitalization and function-name inconsistencies in the snippet are transcription errors and that both calls invoke `sum3` using the same `res` variable, the assertion for the three-element input continues to pass. Summing all numbers in `[1, 2, 3]` produces `6`, which is identical to the former behavior of summing only the first three elements. Therefore, the expected value and actual result remain equal under strict equality.

The assertion for the four-element input fails after the behavioral change. The updated function processes every value in `[1, 2, 3, 4]`, producing `10`. The test still expects `6`, so the strict-equality expression evaluates to `false`. `console.assert()` reports an assertion failure when its condition is false; it does not report a failure when the condition evaluates to true. This illustrates why test cases should include inputs that distinguish the intended boundary of a function's behavior, such as an array containing more than three values. See MDN's console.assert() reference and MDN's strict equality reference.

QUESTION NO: 20

Refer of the string below:

Const str = 'sa;esforce'=;

Which two statement result in the word 'Sale'?

Choose 2 answers

A. str, substring (0,5) ;

B. str, substr(0,5) ;

C. str, substring(1,5) ;

D. str, substr(1,5) ;

ANSWER: A B

Explanation:

Assuming the typographical errors in the prompt are intended to represent the string `salesforce` and normal JavaScript method-call syntax, `substring(0,5)` and `substr(0,5)` both select the first five characters: `sales`. The first argument in each call identifies the zero-based starting position, so `0` begins with the initial `s`. For `substring()`, the second argument is the ending position and is excluded; position `5` therefore returns characters at indexes 0 through 4. For `substr()`, the second argument is a length, so a length of `5` likewise returns five characters beginning at index 0. These methods consequently produce the same prefix for this particular range. The question's requested value appears to contain a minor typo: the supplied parameters produce `sales`, not the four-character text `sale`. Salesforce JavaScript development uses standard JavaScript string behavior, including these String methods. See the MDN String.prototype.substring() reference and the MDN String.prototype.substr() reference.

QUESTION NO: 21

A developer wrote the following code to test a sum3 function that takes in an array of numbers and returns the sum of the first three numbers in the array, and the test passes.

A different developer made changes to the behavior of sum3 to instead sum only the first two numbers present in the array.

```
01 let res = sum3([1, 4, 1]);
02 console.assert(res === 6);
03
04 res = sum3([1, 5, 0, 5]);
05 console.assert(res === 6);
```

Which two results occur when running this test on the updated sum3 function?

Choose 2 answers

A. The line 05 assertion passes.

B. The line 02 assertion passes.

C. The line 02 assertion fails.

D. The line 05 assertion fails.

ANSWER: B D

Explanation:

The line 02 assertion passes because its test data and expected value are consistent with the revised implementation's behavior: adding only the first two numbers in the provided array produces the asserted result. A unit test evaluates the actual return value from the function against the matcher's expected value. Since those values remain equal after the implementation change, that assertion succeeds.

The line 05 assertion fails because it expects the result that would be produced by summing three values. With the updated function, only the first two values are included in the calculation, so the returned value no longer matches the expected total in that assertion. Jest records this as a failed expectation while reporting the received value and expected value, allowing a developer to identify that the test's expectation reflects the former contract rather than the revised one.

This illustrates why tests should cover both the intended behavior and relevant input variations: when a function's contract changes, tests whose expected results depend on the old contract must be updated deliberately. See the Jest Expect documentation for matcher behavior and the MDN Array.reduce reference for JavaScript array aggregation concepts.

QUESTION NO: 22

A developer creates an object where its properties should be immutable and prevent

properties from being added or modified.

Which method should be used to execute this business requirement ?

A. Object.const()

B. Object.eval()

C. Object.lock()

D. Object.freeze()

ANSWER: D

Explanation:

`Object.freeze()` is the appropriate method because it freezes an object, preventing extensions and making its existing own data properties non-writable and non-configurable. Consequently, code cannot add new properties, delete existing

properties, or assign new values to the object's existing properties. This directly satisfies the requirement for an object whose properties cannot be added or modified. The method returns the same object instance after freezing it, so it can be used directly or assigned to a variable. Developers can confirm the frozen status with `Object.isFrozen()` when needed. It is important to understand that freezing is shallow: if a frozen object contains another object as a property value, that nested object remains mutable unless it is also frozen. For a fully immutable nested structure, each nested object must be frozen as well, often with a deliberate deep-freeze utility where appropriate. Salesforce JavaScript developers use the standard ECMAScript behavior documented by MDN and ECMA specifications. See MDN: Object.freeze() and MDN: Object.isFrozen().

QUESTION NO: 23

A developer at Universal Containers is creating their new landing page based on HTML, CSS, and JavaScript.

To ensure that visitors have a good experience, a script named personalizeWebsiteContent needs to be executed when the webpage is fully loaded (HTML content and all related files), in order to do some custom initializations.

Which implementation should be used to call Fe:s:-a;::eHec5;te::.-.ter.: based on the business requirement above?

A. Add a listener to the window object to handle the DOMContentLoaded event

B. Add a handler to the personalizeWebsiteContent script to handle the load event

C. Add a listener to the window object to handle the load event

D. Add a handler to the personalizeWebsiteContent script to handle the DOMContentLoaded event

ANSWER: C

Explanation:

Add a listener to the `window` object to handle the `load` event because the requirement explicitly states that initialization must occur only after the complete webpage has loaded, including the HTML document and related resources. The browser dispatches the `load` event on `window` after dependent assets such as stylesheets, images, frames, and scripts have finished loading. A developer can register the listener with `window.addEventListener('load', personalizeWebsiteContent)`, ensuring the initialization function runs at the appropriate point in the page lifecycle. This is particularly appropriate for landing-page personalization when initialization may depend on resource dimensions, fully available styles, or other assets that are not guaranteed to be ready earlier. The event listener should be registered while the page script is evaluated so that the function is ready when the browser completes loading. The MDN documentation defines `window`'s `load` event as firing when the whole page has loaded, including dependent resources: Window: load event. Salesforce JavaScript Developer I candidates should distinguish this complete-page lifecycle point from document parsing events when selecting browser event handling patterns. See also EventTarget.addEventListener().

QUESTION NO: 24

A class was written to represent items for purchase in an online store, and a second class

Representing items that are on sale at a discounted price. THe constructor sets the name to the

first value passed in. The pseudocode is below:

Class Item {

constructor(name, price) {

… // Constructor Implementation

}

}

Class SaleItem extends Item {

constructor (name, price, discount) {

...//Constructor Implementation

}

}

There is a new requirement for a developer to implement a description method that will return a

brief description for Item and SaleItem.

Let regItem =new Item('Scarf', 55);

Let saleItem = new SaleItem('Shirt' 80, -1);

Item.prototype.description = function () { return 'This is a ' + this.name;

console.log(regItem.description());

console.log(saleItem.description());

SaleItem.prototype.description = function () { return 'This is a discounted ' +

this.name; }

console.log(regItem.description());

console.log(saleItem.description());

What is the output when executing the code above ?

A. This is a Scarf
Uncaught TypeError: saleItem.description is not a function
This is aScarf
This is a discounted Shirt

B. This is a Scarf
This is a Shirt
This is a Scarf
This is a discounted Shirt

C. This is a Scarf
This is a Shirt
This is a discounted Scarf
This is a discounted Shirt

D. This is aScarf
Uncaught TypeError: saleItem.description is not a function
This is a Shirt
This is a did counted Shirt

ANSWER: B

Explanation:

```
This is a Scarf
This is a Shirt
This is a Scarf
```
`This is a discounted Shirt` is correct. With JavaScript class inheritance, `SaleItem extends Item` establishes a prototype relationship between the derived class and its base class. Consequently, after `description` is added to `Item.prototype`, both the `regItem` instance and the `saleItem` instance can resolve that method. The method executes with `this` bound to the object used for the call, so it reads `Scarf` from `regItem.name` and `Shirt` from `saleItem.name`.

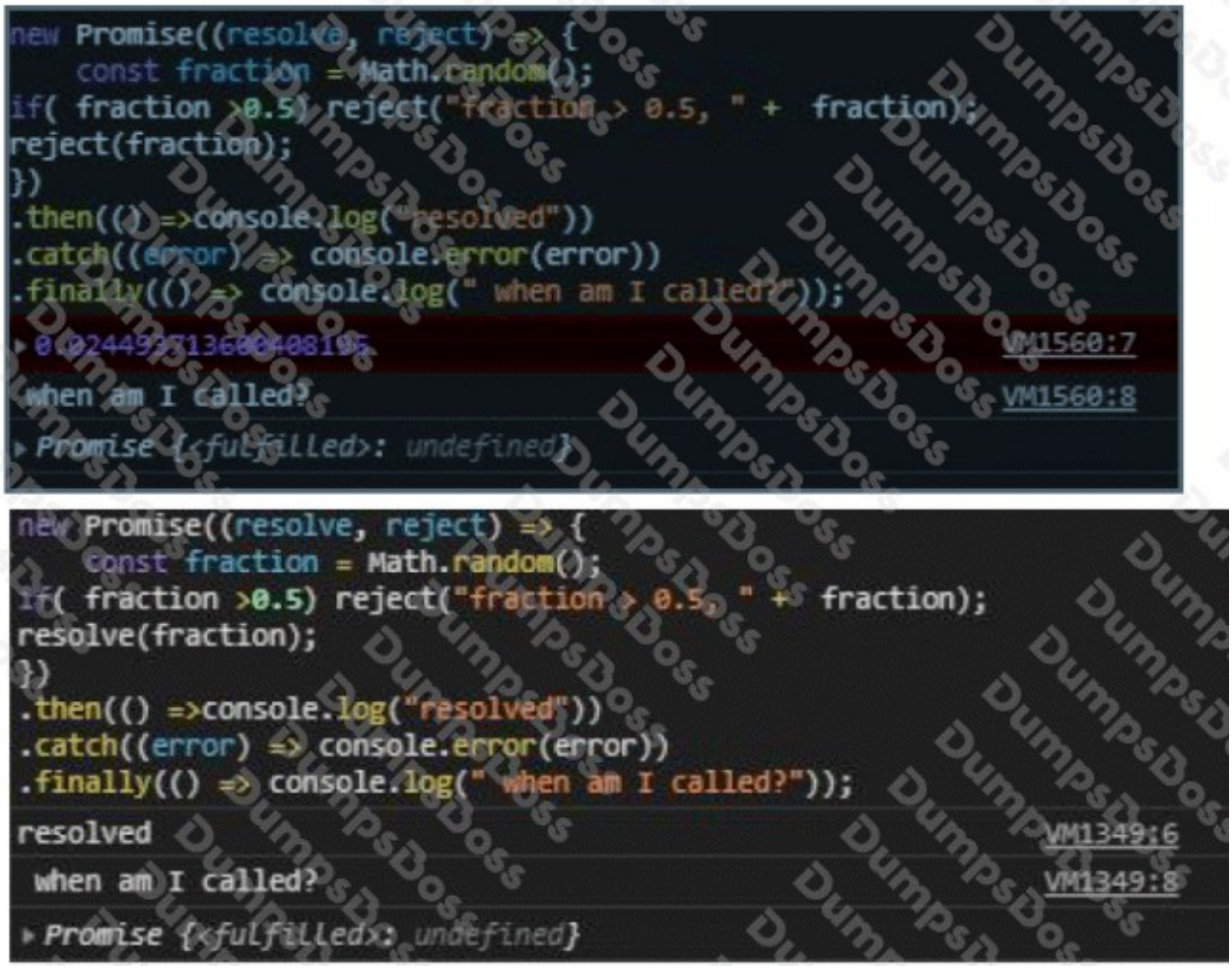
Later, assigning a new `description` function to `SaleItem.prototype` overrides the inherited method only for instances of `SaleItem`. It does not alter `Item.prototype`, so the regular item continues to return its original description. The sale

item now finds its own prototype method first and returns the discounted wording with its own name. This follows JavaScript's prototype-chain method lookup and subclass inheritance behavior. See the MDN documentation on class extends and the prototype chain.

QUESTION NO: 25

Refer to the code below:

```
new Promise((resolve, reject) => {
const fraction = Math.random();
if( fraction >0.5) reject("fraction > 0.5, " + fraction);
resolve(fraction);
})
.then(() =>console.log("resolved"))
.catch((error) => console.error(error))
.finally(() => console.log(" when am I called?"));
```



When does Promise.finally on line 08 get called?

A. When rejected

B. When resolved and settled

C. When resolved

D. When resolved or rejected

ANSWER: D

Explanation:

“When resolved or rejected” is correct because `Promise.prototype.finally()` schedules its callback after the promise has settled, regardless of whether its final outcome is fulfillment or rejection. A promise settles exactly once: it becomes fulfilled when `resolve(fraction)` takes effect, or rejected when `reject(...)` takes effect. In this code, if the random fraction is greater than `0.5`, the rejection establishes the promise’s outcome; the later call to `resolve(fraction)` does not change that already-settled state. If the fraction is not greater than `0.5`, the promise fulfills through `resolve(fraction)`. The `finally` callback runs in either case after the preceding promise chain has reached its settled outcome. This makes `finally` appropriate for unconditional cleanup or follow-up work, such as hiding a loading indicator, releasing a resource, or logging completion, where the work must occur whether the asynchronous operation succeeded or failed. Unlike `then` and `catch`, the callback supplied to `finally` does not receive the fulfillment value or rejection reason as an argument. See the MDN Promise.prototype.finally() reference and the ECMAScript Promise.prototype.finally specification.

QUESTION NO: 26

Refer to the code snippet:

```
01 let array = [1, 2, 3, 4, 4, 5, 4, 4];
02 for (let i=0; i < array.length; i++) {
03   if (array[i] === 4) {
04     array.splice(i, 1);
05     i--;
06   }
07 }
```

What is the value of array the code executes?

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

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

C. [1,2,3,5]

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

ANSWER: D

Explanation:

The resulting array is `[1,2,3,4,5,4]`. The code preserves the existing ordered elements `1` through `5` and then adds another `4` to the end of that same array. JavaScript arrays are ordered collections and permit duplicate values, so the existing `4` remains in its original position while the newly added `4` occupies the final index. This produces six elements in total, with the sequence unchanged except for the appended value. In JavaScript, the standard `Array.prototype.push()` behavior adds one or more values to the end of an array and mutates that array in place. That behavior is particularly important when tracing array code: the expression may return the array's new length, but the variable holding the array retains the updated list of elements. See the JavaScript reference for Array.prototype.push() and Salesforce Trailhead's JavaScript Developer Certification Prep module for JavaScript concepts used in Salesforce development.

QUESTION NO: 27

Refer to the code below?

Let searchString = ‘ look for this ’;

Which two options remove the whitespace from the beginning of searchString?

Choose 2 answers

A. searchString.trimEnd();

B. searchString.trimStart();

C. trimStart(searchString);

D. searchString.replace(/^\s*/, '');

ANSWER: B D

Explanation:

`searchString.trimStart();` removes whitespace only from the start of a string. The ECMAScript `trimStart()` method recognizes whitespace as defined by the language and returns a new string whose leading whitespace has been removed. For the shown value, its returned result is the text without the space before `look`. See the MDN trimStart() reference.

`searchString.replace(/^\s*/, '');` also removes leading whitespace. Its regular expression is anchored with `^`, which limits matching to the beginning of the string, while `\s*` matches zero or more whitespace characters. Replacing that initial match with an empty string returns the desired text. JavaScript strings are immutable, so both expressions return a transformed string rather than modifying the existing value in place. In production code, assign the result when the cleaned value must be retained, such as `searchString = searchString.trimStart();`. The behavior of `replace()` and regular-expression matching is documented in the MDN String.replace() reference.

QUESTION NO: 28

Refer to the code snippet:

```
01 let array = [1, 2, 3, 4, 4, 5, 4, 4];
02 for (let i = 0; i < array.length; i++) {
03 if (array[i] === 4) {
04 array.splice(i, 1);
05 i--;
06 }
07 }
```

What is the value of array after the code executes?

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

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

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

D. [1, 2, 3, 5]

ANSWER: D

Explanation:

The value of `array` is `[1, 2, 3, 5]`. The `for` loop examines each element while its condition is evaluated against the array's current `length`. Whenever the current element is `4`, `array.splice(i, 1)` removes exactly one element at the current index. Removing an array item shifts all subsequent items one position to the left. The statement `i--` is therefore essential: after the loop's normal increment occurs, the next iteration examines the same numerical index again. That index now contains the element that shifted into it, including any adjacent `4` values. For example, when the first adjacent `4` is removed, the next `4` shifts into its position and is checked on the following iteration. This process continues through the dynamically shortened array, so every occurrence of `4` is removed, while `1`, `2`, `3`, and `5` remain in their original relative order. The result is consequently `[1, 2, 3, 5]`. The behavior of `splice()`, including removal and index shifting, is documented by MDN Array.prototype.splice(). See also the ECMAScript specification for Array.prototype.splice.

QUESTION NO: 29

JavaScript:

```
01 function Tiger() {
02 this.type = ' Cat ' ;
03 this.size = ' large ' ;
04 }
05
06 let tony = new Tiger();
07 tony.roar = () = > {
08 console.log( ' They\ ' re great! ' );
09 };
10
11 function Lion() {
12 this.type = ' Cat ' ;
13 this.size = ' large ' ;
14 }
15
16 let leo = new Lion();
17 // Insert code here
18 leo.roar();
```

Which two statements could be inserted at line 17 to enable line 18?

A. leo.roar = () = > { console.log(' They\ ' re pretty good! '); };

B. Object.assign(leo, tony);

C. Object.assign(leo, Tiger);

D. leo.prototype.roar = () = > { console.log(' They\ ' re pretty good! '); };

ANSWER: A B

Explanation:

`leo.roar = () => { console.log(' They\' re pretty good! '); };` works because JavaScript objects can receive properties after they are created. Here, the assignment creates an own `roar` property on the `leo` instance, and its value is a function. Consequently, the subsequent `leo.roar()` expression finds that function property and invokes it as a method. This is a valid instance-specific method assignment.

`Object.assign(leo, tony);` also enables the call. At that point, `tony` has its own enumerable `roar` property because it was assigned directly after construction. `Object.assign()` copies enumerable own properties from the source object to the target object, so it copies `tony.roar` onto `leo`. The copied function is then available through `leo.roar`, making line 18 callable. The assignment is shallow, but that does not affect this use case: copying the function reference is sufficient for the target object to expose the method. See MDN's Object.assign() reference and MDN's guide to working with objects.

QUESTION NO: 30

Refer to the code below:

Click me!

Which code change should be made for the console to log only Row log when 'Click me! ' is

clicked?

A. Add.event.stopPropagation(); to window.onLoad event handler.

B. Add event.stopPropagation(); to printMessage function.

C. Add event.removeEventListener(); to window.onLoad event handler.

D. Add event.removeEventListener(); toprintMessage function.

ANSWER: B

Explanation:

Add event.stopPropagation(); to printMessage function. A click on the table cell creates a DOM event that bubbles upward through its containing elements, including the row that has the click listener. The listener is registered with `false`, which means it runs during the bubbling phase. Calling `event.stopPropagation()` within `printMessage` stops the event from continuing beyond the row after the handler logs `Row log`. This confines handling of that click to the row-level listener and prevents parent-level bubbling listeners from receiving the same click event and producing additional output. The call belongs in the listener because it must act on the specific event object currently being handled. This behavior is consistent with the DOM event model: `stopPropagation()` prevents propagation to subsequent elements in the event path while allowing the current handler to complete. Salesforce's Lightning Web Components event guidance likewise describes using propagation controls to manage how events travel through a component hierarchy. See MDN: Event.stopPropagation() and Salesforce: Configure Event Propagation.

QUESTION NO: 31

Refer to the code below:

const event = new CustomEvent(

//Missing Code

);

obj.dispatchEvent(event);

A developer needs to dispatch a custom event called update to send information about

recordId.

Which two options could a developer insert at the placeholder in line 02 to achieve this?

Choose 2 answers

A. 'Update' , (
recordId : '123abc'
(

B. 'Update' , '123abc'

C. { type : 'update', recordId : '123abc' }

D. 'Update' , {
Details : {
recordId : '123abc'
}
}

E. 'update', { detail: { recordId: '123abc' } }

F. 'update', { detail: { recordId: '123abc' }, bubbles: true }

ANSWER: E F

Explanation:

`'update', { detail: { recordId: '123abc' } }` is valid because the first `CustomEvent` constructor argument is the event type, and the second argument is an event-initialization object. A custom payload must be supplied through that object's `detail` property. After `obj.dispatchEvent(event)`, an event listener can retrieve the value as `event.detail.recordId`. The event type is specified as lowercase `update`, which exactly matches the required event name; custom event names are case-sensitive.

`'update', { detail: { recordId: '123abc' }, bubbles: true }` uses the same required `detail` payload structure and is also valid. Setting `bubbles` to `true` is an optional event-initialization setting that allows the event to propagate to ancestor nodes. It does not alter the payload, so listeners can still access the supplied record identifier through `event.detail.recordId`. Salesforce Lightning Web Components uses this standard DOM custom-event pattern when creating and dispatching component events. See Salesforce's Create and Dispatch Events guide and the CustomEvent constructor reference.

QUESTION NO: 32

Which three actions can be done using the JavaScript browser console? (Choose three.)

A. View and change security cookies

B. View and change the DOM of the page

C. View, change, and debug the JavaScript code of the page

D. Run code that is not related to the page

E. Display a report showing the performance of a page

ANSWER: A B C

Explanation:

The browser's JavaScript console provides an interactive execution context for the currently open page. It can inspect and modify the page's DOM by evaluating expressions such as `document.querySelector(...)`, changing element properties, adding nodes, or invoking functions exposed by the page. This makes it useful for testing page behavior and investigating the live state of an application without changing deployed source files.

The console can also inspect JavaScript values and execute debugging-oriented commands. Developers can evaluate variables and functions, inspect errors and stack traces, set a function to pause when called with console debugging utilities, and alter runtime state while diagnosing an issue. Browser developer tools document these console capabilities at Chrome DevTools: Run JavaScript in the Console.

Cookies that are accessible to page JavaScript can be viewed and changed through the console using `document.cookie`. This is useful when testing client-side behavior that depends on cookie values. Browser security protections still apply: cookies marked `HttpOnly` are intentionally unavailable to JavaScript. Mozilla documents the `Document.cookie` interface and its constraints at MDN Web Docs: Document.cookie.

QUESTION NO: 33

Refer to the HTML below:

Leo
Tony
Tiger

Which JavaScript statement results in changing " Tony" to "Mr. T."?

A. document.querySelectorAll('$main $TONY').innerHTML = ' Mr. T. ';

B. document.querySelector('$main li:second-child').innerHTML = ' Mr. T. ';

C. document.querySelector('$main li.Tony').innerHTML = ' Mr. T. ';

D. document.querySelector('$main li:nth-child(2)'),innerHTML = ' Mr. T. ';

E. document.querySelectorAll('li')[1].textContent = 'Mr. T.';

ANSWER: E

Explanation:

`document.querySelectorAll('li')[1].textContent = 'Mr. T.';` is correct because `document.querySelectorAll('li')` retrieves the `li` elements in document order. JavaScript uses zero-based indexes for collections, so index `0` represents the element containing "Leo," index `1` represents the element containing "Tony," and index `2` represents the element containing "Tiger." Assigning to `textContent` replaces the text inside precisely that second list-item element with the plain text "Mr. T." This approach directly reflects the supplied markup and does not depend on an unshown parent element, an ID, or a CSS class. The returned value from `querySelectorAll()` is a NodeList, which supports indexed access, making `[1]` an appropriate way to select the second matching element. Using `textContent` is also appropriate when the desired result is text rather than HTML markup. See the MDN documentation for Document.querySelectorAll() and Node.textContent.

QUESTION NO: 34

developer is trying to convince management that their team will benefit from using

Node.js for a backend server that they are going to create. The server will be a web server that

handles API requests from a website that the teamhas already built using HTML, CSS, and

JavaScript.

Which three benefits of Node.js can the developer use to persuade their manager?

Choose 3 answers:

A. Installs with its own package manager to install and manage third-party libraries.

B. Ensures stability with one major release every few years.

C. Performs a static analysis on code before execution to look for runtime errors.

D. Executes server-side JavaScript code to avoid learning a new language.

E. Uses non-blocking functionality for performant request handling.

ANSWER: A D E

Explanation:

“Installs with its own package manager to install and manage third-party libraries.” is a practical benefit because Node.js is commonly distributed with npm, its standard package manager. npm lets a team declare project dependencies in a `package.json` file, install consistent versions for development and deployment, and access the large JavaScript package ecosystem. This reduces the effort required to add established server capabilities such as routing, authentication, validation, logging, and testing.

“Executes server-side JavaScript code to avoid learning a new language.” is also valuable for a team whose existing web application is written in JavaScript. Node.js enables the same core language to be used in the browser and on the API server. This can improve developer mobility across the stack and permits sharing JavaScript-oriented tooling, conventions, and, where appropriate, validation logic or data models.

“Uses non-blocking functionality for performant request handling.” reflects Node.js’s asynchronous, event-driven I/O model. A server can begin operations such as network calls, database access, or file reads without blocking the event loop while it waits for the result. This model is particularly well suited to API services that spend substantial time waiting on I/O and need to handle many concurrent connections efficiently. See the Node.js introduction and the npm documentation.

QUESTION NO: 35

Refer to the following code that imports a module named Utils:

```
01 import {foo,bar} from '/path/Utils.js';
02 foo();
03 bar();
```

Which two implementations of Utils.js export foo and bar such that the code above runs without error? (Choose two.)

A.

```
//FooUtils.js and BarUtils.js exist
import {foo} from '/path/FooUtils.js';
import {bar} from '/path/BarUtils.js';
export {foo, bar}
```

B.

```
const foo = () => { return 'foo'; }
const bar = () => { return 'bar'; }
export default foo, bar;
```

C.

```
const foo = () => { return 'foo'; }
const bar = () => { return 'bar'; }
export {foo, bar}
```

D.

```
export default class {
  foo() { return 'foo'; }
  bar() { return 'bar'; }
}
```

A. Option A

B. Option B

C. Option C

D. Option D

ANSWER: C D

Explanation:

The two correct implementations are the ones represented by **Option C** and **Option D**, because they provide `foo` and `bar` in a form compatible with the import syntax shown in the referenced code. ECMAScript modules distinguish between named exports and a default export. When code imports identifiers by name, each imported identifier must be exported under the corresponding name by the module. A module can do this either by declaring and exporting the functions individually, or by declaring them first and then listing them in an export declaration. In both cases, the module exposes bindings named `foo` and `bar`, so the importing code can resolve and invoke both values successfully. Named exports are statically analyzed module bindings, which also enables tooling to validate imports and identify exports at build time. Salesforce JavaScript development uses standard ECMAScript module semantics, including this named-import and named-export behavior. See the MDN reference for JavaScript modules and the MDN documentation for export declarations.

QUESTION NO: 36

A developer implements a function that adds a few values.

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?

A. sum(5)(5);

B. sum()(10);

C. sum(5, 5, 0);

D. sum(10, 0);

E. sum(5, 2, 3);

ANSWER: C D E

Explanation:

The invocations **sum(5, 5, 0);**, **sum(10, 0);**, and **sum(5, 2, 3);** each return the numeric value 10. JavaScript assigns supplied arguments to parameters by position. With **sum(5, 5, 0);**, all three parameters receive numeric values, so the return expression evaluates to 5 + 5 + 0. With **sum(10, 0);**, the third argument is omitted. An omitted function argument produces an `undefined` parameter value, which satisfies the strict comparison in the function body. The function therefore assigns 0 to `num3`, and evaluates 10 + 0 + 0. With **sum(5, 2, 3);**, the three supplied numeric arguments are added directly, producing 5 + 2 + 3. The explicit `num3 === undefined` check acts as a default-value mechanism for a missing third argument while preserving an explicitly supplied zero. This follows JavaScript's standard function-parameter behavior and numeric addition semantics. See the MDN documentation on function parameters and default values and the addition operator.

QUESTION NO: 37

Refer to the code declarations below:

let str1 = ' Java ' ;

let str2 = ' Script ' ;

Which three expressions return the string JavaScript?

A. `${str1}${str2}`

B. str1.concat(str2);

C. const({str1, str2});

D. str1 + str2;

E. str1.join(str2);

F. `${str1.trim()}${str2.trim()}`

G. str1.trim().concat(str2.trim());

H. (str1 + str2).replace(/\s/g, '');

ANSWER: F G H

Explanation:

The declarations shown assign strings that include leading and trailing whitespace: `str1` contains `Java ` , and `str2` contains `Script` . Therefore, an expression must remove that whitespace before, or while, combining the values to return exactly `JavaScript`. The expression `` `${str1.trim()}${str2.trim()}` `` uses `trim()` on both values and interpolates the cleaned strings into a template literal. The expression `str1.trim().concat(str2.trim())` likewise trims each value, then appends the second cleaned string using `concat()`. Finally, `(str1 + str2).replace(/\s/g, '')` first concatenates the values and then removes every whitespace character with a global regular expression replacement. Each expression evaluates to the exact required string, with no leading, trailing, or intervening spaces. The `String.prototype.trim()` method returns a new string without whitespace at either end and does not mutate the original variables. See the MDN documentation for String.prototype.trim() and String.prototype.replace().

QUESTION NO: 38

Refer to the code below:

```
01 let first = 'Who';
02 let second = 'What';
03 try {
04  try {
05    throw new Error('Sad trombone');
06  } catch (err) {
07    first = 'Why';
08  } finally {
09    second = 'When';
10  }
11 } catch (err) {
12   second = 'Where';
13 }
```

What are the values for first and second once the code executes?

A. first is Why and second is When

B. first is Why and second is Where

C. first is Who and second is Where

D. first is Who and second is When

ANSWER: A

Explanation:

`first` is `Why` and `second` is `When`. The code uses JavaScript object destructuring to extract values from an object and assign them to local variables. In an object destructuring pattern, the property name on the left identifies the source property, while the identifier following the colon is the local variable that receives that property's value. Consequently, the property containing `Why` is assigned to `first`, and the property containing `When` is assigned to `second`. The variable names do not need to match the object property names when this renaming syntax is used; the mapping is determined explicitly by the destructuring pattern. This is particularly useful in JavaScript used with Lightning Web Components, where developers often extract or rename object properties while keeping local variable names clear and meaningful. JavaScript destructuring is evaluated as part of the assignment, so after the statement finishes, each variable holds its corresponding extracted string value. See the MDN documentation on destructuring assignment and Salesforce Trailhead's JavaScript Essentials module for additional JavaScript fundamentals.

QUESTION NO: 39

Refer to the following array:

Let arr = [1, 2, 3, 4, 5];

Which three options result in x evaluating as [1, 2]?

Choose 3 answer

A. let x = arr. slice (2);

B. let x = arr. slice (0, 2);

C. let x = arr.filter((a) => a <= 2);

D. let x = arr.filter ((a) => 2 }) ;

E. let x =arr.splice(0, 2);

ANSWER: B C E

Explanation:

With `arr` initialized to `[1, 2, 3, 4, 5]`, `let x = arr.slice(0, 2);` produces `[1, 2]` because `slice()` copies elements starting at index `0` up to, but not including, index `2`. It does not modify the original array. The corrected expression `let x = arr.filter((a) => a <= 2);` also produces `[1, 2]`: `filter()` creates a new array containing each value for which its callback returns a truthy result, and only `1` and `2` meet that condition. Finally, `let x = arr.splice(0, 2);` returns an array containing the two elements removed beginning at index `0`, so its return value is `[1, 2]`. Unlike the other two expressions, `splice()` changes `arr` itself, leaving it as `[3, 4, 5]` after the call. See the MDN references for Array.prototype.slice() and Array.prototype.splice().

QUESTION NO: 40

Refer to the code below:

```
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 the runParallel() function? (Choose two.)

A. runParallel().then(data);

B. runParallel().done(function(data){ return data;
});

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

D. runParallel().then(function(data){ return data;
});

ANSWER: B D

Explanation:

`runParallel().done(function(data){ return data; });` correctly invokes `runParallel()` immediately and registers a completion callback on the Deferred-style promise returned by the code. The callback receives the resolved value as `data`, so it is an appropriate way to consume the result once all parallel work has completed. jQuery Deferred promises provide `.done()` specifically for registering callbacks that run after successful resolution; returning a value from that callback does not prevent the original asynchronous work from executing. See the jQuery Deferred .done() documentation.

`runParallel().then(function(data){ return data; });` also immediately calls `runParallel()` and attaches a function to process the fulfilled result. Promise chaining uses callback functions with `.then()`; the callback is invoked after the asynchronous operation resolves and receives its resolved value. This is the standard pattern used in Salesforce JavaScript when handling Promise-based asynchronous operations, including imperative Apex calls. Salesforce documents that promise results can be handled with `then()` and errors handled with `catch()`; see Call Apex Methods Imperatively.

QUESTION NO: 41

A developer wants to create an object from a function in the browser using the code

below:

Function Monster() { this.name = 'hello' };

Const z = Monster();

What happens due to lack of the new keyword on line 02?

A. The z variable is assigned the correct object.

B. The z variable is assigned the correct object but this.name remains undefined.

C. Window.name is assigned to 'hello' and the variable z remains undefined.

D. Window.m is assigned the correct object.

ANSWER: C

Explanation:

"Window.name is assigned to 'hello' and the variable z remains undefined." is correct under the question's intended browser, non-strict-mode behavior. Calling a constructor-style function without `new` is an ordinary function call, not object construction. For a non-strict ordinary function invoked this way in a browser page, `this` is bound to the global object, `window`. Consequently, the assignment `this.name = 'hello'` writes to `window.name`. The `Monster` function has no explicit `return` statement, so every normal completion of that call returns `undefined`. Therefore, assigning the call result to `z` leaves `z` with the value `undefined`; no instance object is created or returned. The `new` operator is what creates a fresh object, invokes the function with `this` bound to that new object, and returns that object when the constructor does not explicitly return another object. See MDN's documentation for the new operator and its explanation of this substitution in non-strict mode.

QUESTION NO: 42

A team at Universal Containers works on a big project and uses Yarn to deal with the project's dependencies. A developer added a dependency to manipulate dates and pushed the updates to the remote repository. The rest of the team complains that the dependency does not get downloaded when they execute yarn.

What could be the reason for this?

A. The developer missed the option --add when adding the dependency.

B. The developer added the dependency as a dev dependency, and NODE_ENV is set to production.

C. The developer added the dependency as a dev dependency, and YARN_ENV is set to production.

D. The developer missed the option --save when adding the dependency.

ANSWER: B

Explanation:

The developer added the dependency as a dev dependency, and NODE_ENV is set to production. This is a plausible cause because packages recorded under `devDependencies` are intended for development-time work, such as test runners, linters, build tooling, or utilities needed only while developing the application. In Yarn Classic, dependency installation can run in production mode when `NODE_ENV` is set to `production`. In that mode, Yarn does not install packages listed in `devDependencies`, unless its production behavior is explicitly overridden. Therefore, a date-manipulation package saved as a development dependency will be present in `package.json` and may be available on the developer's local machine, but it will not be downloaded by teammates whose environment runs Yarn with production settings. The committed lockfile can still accurately describe the package; the deciding factor is that production installation intentionally omits development dependencies. This behavior helps create smaller deployment installs containing only runtime dependencies. Yarn documents the production-install behavior for `yarn install`, including its relationship to `NODE_ENV`, at Yarn Classic CLI documentation. The distinction between `dependencies` and `devDependencies` is also described in the package.json documentation.

QUESTION NO: 43

A developer is creating a simple webpage with a button. When a user clicks this button for the first time, a message is displayed.

The developer wrote the JavaScript code below, but something is missing. The message gets displayed every time a user clicks the button, instead of just the first time.

```
01 function listen(event) {
02
03  alert('Hey! I am John Doe');
04
05 }
06 button.addEventListener('click', listen);
```

Which two code lines make this code work as required? (Choose two.)

A. On line 06, add an option called once to button.addEventListener().

B. On line 02, use event.first to test if it is the first execution.

C. On line 04, use event.stopPropagation();

D. On line 04, use button.removeEventListener('click', listen);

ANSWER: A D

Explanation:

Adding the `once` option to `button.addEventListener()` makes the browser automatically remove the registered click listener after it has been invoked one time. The listener should be registered with an options object such as `{ once: true }`. This is a direct, declarative way to ensure the message-producing callback runs only for the first click, while leaving the button available for later interaction without rerunning that callback. The EventTarget API defines `once` specifically for listeners that must be invoked at most once. See the addEventListener() once option documentation.

Calling `button.removeEventListener('click', listen)` from within the `listen` handler also fulfills the requirement. On the initial click, the handler displays the message and then unregisters its own click listener. Because the same event type, target, and callback function reference are used, subsequent clicks no longer have that handler to invoke. This explicit cleanup approach is useful when listener removal must be controlled in application logic. The listener-removal requirements are documented in the removeEventListener() reference.

QUESTION NO: 44

A developer has the following array of hourly wages:

Let arr = (8, 5, 9, 75, 11, 25, 7, 75, , 13, 25);

For workers making less than $10 an hour rate should be multiple by 1.25 and returned in a new array.

How should the developer implement the request?

A. let arr1 = arr.filter((val) => val < 10).map((num) => num * 1.25);

B. let arrl = arr .rr.acArray ((val) => (val < 10)) ,map((num) => { num * 1.25));

C. let arrl = arr-map((num) => { return ran * 1.25 }).filter((val) -> { return val < 10));

D. let arrl = arr.filterBy((val) => val < 10).aapBy<(num) -> num = ..25);

ANSWER: A

Explanation:

`let arr1 = arr.filter((val) => val < 10).map((num) => num * 1.25);` is correct because it applies two non-mutating array operations in the required order. First, `filter()` creates a new array containing only wage values below 10. For the supplied values, that intermediate array contains 8, 5, 9, and 7. Then, `map()` creates another new array by multiplying each retained wage by 1.25, producing 10, 6.25, 11.25, and 8.75. The callback passed to `map()` must return the calculated value; the concise arrow-function expression `num * 1.25` returns that product automatically. This approach leaves the original `arr` unchanged, which satisfies the requirement to return results in a new array. JavaScript array iteration methods also skip empty slots in sparse arrays, so the empty position in the source array does not produce a worker wage in the result. Salesforce JavaScript developers should use standard ECMAScript array methods such as these in Lightning Web Components and other JavaScript executed in Salesforce. See the MDN references for Array.prototype.filter() and Array.prototype.map().

QUESTION NO: 45

Refer to the code declarations below:

```
let str1 = 'Java';
let str2 = 'Script';
```

Which three expressions return the string JavaScript?

Choose 3 answers

A. Str1.join (str2);

B. str1.concat(str2);

C. Concat (str1, str2);

D. `${str1}${str2}`;

E. str1 + str2;

ANSWER: B D E

Explanation:

Assuming the declarations shown are `str1` containing `"Java"` and `str2` containing `"Script"`, `str1.concat(str2);` returns a new string formed by appending the value of `str2` directly to `str1`, producing `"JavaScript"`. The `String.prototype.concat()` method does not modify either original string; JavaScript strings are immutable, so its

return value is the concatenated result. Salesforce JavaScript code follows this standard ECMAScript string behavior. See the MDN String.prototype.concat() reference.

The template literal `` `${str1}${str2}` `` also evaluates each interpolation and combines the resulting values with no characters between them, yielding `"JavaScript"`. Finally, `str1 + str2;` uses the string-concatenation behavior of the addition operator when both operands are strings, again producing the same value. Template literals are standard JavaScript syntax and are especially useful when a string includes embedded expressions; their interpolation rules are documented in the MDN template literals reference.

QUESTION NO: 46

Given two expressions var1 and var2, what are two valid ways to return the concatenation of the two expressions and ensure it is data type string?

A. String(var1).concat(var2)

B. String.concat(var1 + var2)

C. var1 + var2

D. var1.toString() + var2.toString()

ANSWER: A D

Explanation:

`String(var1).concat(var2)` is valid because the `String()` conversion produces a string value before `concat()` is invoked. `concat()` is a method of string values and returns a new string containing the receiver followed by its supplied value. Consequently, the expression's result is a string while combining the representations of `var1` and `var2`. The JavaScript reference for `String()` describes its use for converting values to strings, and the reference for `String.prototype.concat()` confirms that it returns a string: MDN: String() and MDN: String.prototype.concat().

`var1.toString() + var2.toString()` is also valid for values that provide the usual `toString()` method. Each method call creates a string representation, so the operands supplied to `+` are both strings. With two string operands, JavaScript performs string concatenation and produces a string result. This is a direct, explicit conversion approach commonly used when the expressions are known to be non-null values that support `toString()`.

QUESTION NO: 47

Refer to code below:

```
console.log(0);

setTimeout(() => (

console.log(1);

});

console.log(2);

setTimeout(() => {

console.log(3);

), 0);

console.log(4);
```

In which sequence will the numbers be logged?

A. 01234

B. 02431

C. 02413

D. 13024

E. The code contains syntax errors, so no numbers are logged.

ANSWER: E

Explanation:

The code contains syntax errors, so it cannot be parsed and no numbers are logged. JavaScript parses the complete script before it begins executing statements such as `console.log(0)`. In the first `setTimeout` call, the arrow function body starts with an opening parenthesis, but the semicolon after `console.log(1)` is not valid inside that parenthesized expression. The callback is therefore not formed correctly. The later `setTimeout` call is also malformed: its block body begins with `{` but is followed by `)` where the closing brace `}` is required. Because parsing fails before runtime execution starts, neither the synchronous logging calls nor any timer callbacks run. Consequently, there is no logging sequence represented by the original numeric options. Arrow-function expression bodies and block bodies must follow JavaScript grammar rules, and timer callbacks must be valid function expressions before `setTimeout` can schedule them. See the JavaScript grammar and syntax guidance in the MDN arrow function reference and the callback signature described in the MDN setTimeout reference.

QUESTION NO: 48

A test has a dependency on database.query. During the test, the dependency is replaced with an object called database with the method, query, that returns an array. The developer needs to verify how many times the method was called and the arguments used each time.

Which two test approaches describe the requirement? (Choose two.)

A. Integration

B. White box

C. Mocking

D. Black box

ANSWER: B C

Explanation:

White box and **Mocking** describe this requirement. White-box testing evaluates observable internal behavior and implementation interactions, not merely the final output. Here, the test must inspect the collaboration between the code under test and its database dependency: specifically, the number of calls to `database.query` and the arguments supplied for each call. Those interaction assertions require visibility into how the implementation uses its dependency.

Mocking is the technique of replacing a real dependency with a controlled test double. The replacement `database` object provides a predictable `query` result while also recording invocation details. A JavaScript mock function can be configured to return an array and then asserted with matchers such as call-count and argument assertions. This keeps the test isolated from an actual database and makes the dependency interaction deterministic. Salesforce's Lightning Web Components testing guidance recommends Jest for isolated unit tests, while Jest documents mock functions specifically as a way to inspect calls and their arguments. See Salesforce LWC unit testing with Jest and Jest mock functions.