



Vendor: Salesforce

Exam Code: JS-Dev-101

Exam Name: Salesforce Certified JavaScript Developer -
Multiple Choice

Version: DEMO

QUESTION 1

A developer wants to set up a secure web server with Node.js. The developer creates a directory locally called app-server, and the first file is app-server/index.js. Without using any third-party libraries, what should the developer add to index.js to create the secure web server?

- A. `const server = require('secure-server');`
- B. `const tls = require('tls');`
- C. `const http = require('http');`
- D. `const https = require('https');`

Answer: D

Explanation:

Node.js provides two built-in modules for creating servers:

`http` -> creates non-secure HTTP servers

`https` -> creates secure HTTPS servers

A secure server requires:

SSL/TLS certificates (key and cert)

The built-in Node.js module designed to handle these: `https`

Syntax for a secure server:

```
const https = require('https');  
  
const fs = require('fs');  
  
const options = {  
  key: fs.readFileSync('key.pem'),  
  cert: fs.readFileSync('cert.pem')  
};  
  
https.createServer(options, (req, res) => {  
  res.write('Secure Server');  
  res.end();  
}).listen(443);
```

QUESTION 2

Refer to the code:

```
01 const event = new CustomEvent(  
02 // Missing code  
03 );
```

```
04 obj.dispatchEvent(event);
```

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

Which two options can be inserted at line 02?

- A. { type: 'update', recordId: '123abc' }
- B. 'update', { detail: { recordId: '123abc' } }
- C. 'update', '123abc'
- D. 'update', { recordId: '123abc' }

Answer: BD

Explanation:

The correct constructor signature for CustomEvent is:

```
new CustomEvent(eventName, optionsObject)
```

Where:

eventName is a string.

optionsObject may include:

detail -> used to pass custom data

bubbles

cancelable, etc.

Example:

```
new CustomEvent('update', {  
  detail: { recordId: '123abc' }  
});
```

Now evaluate each option:

Option A

```
{ type: 'update', recordId: '123abc' }
```

Incorrect: The constructor requires (eventName, options), not a single object. type is not used this way.

Option B

```
'update', { detail: { recordId: '123abc' } }
```

Correct format. detail is the proper place for custom event data.

Option C

'update', '123abc'

Incorrect: The second argument must be an object (options), not a string.

Option D

'update', { recordId: '123abc' }

Acceptable because any extra properties on the options object are still allowed, even though best practice is to use detail.

This still creates a valid CustomEvent, and the event will dispatch successfully.

Thus the two correct answers are B and D.

QUESTION 3

Refer to the code below:

```
01 let sayHello = () => {  
02   console.log('Hello, World!');  
03 };
```

Which code executes sayHello once, two minutes from now?

- A. delay(sayHello, 120000);
- B. setTimeout(sayHello, 120000);
- C. setTimeout(sayHello(), 120000);
- D. setInterval(sayHello, 120000);

Answer: B

Explanation:

To schedule a function for later execution in JavaScript, the standard built-in method is:

setTimeout(functionReference, delayInMilliseconds)

Characteristics of setTimeout():

Executes only once.

Delay is specified in milliseconds.

Requires passing the function reference, not calling the function.

Two minutes is:

2 minutes = 120 seconds = 120,000 milliseconds

Now evaluate each option:

Option A: delay(sayHello, 120000);

Incorrect.

There is no built-in `delay()` function in JavaScript.

Option B: `setTimeout(sayHello, 120000);`

Correct.

`sayHello` is passed as a function reference, so it will execute once after 120,000 ms.

Option C: `setTimeout(sayHello(), 120000);`

Incorrect.

`sayHello()` calls the function immediately, and the return value (undefined) is passed to `setTimeout`.

This executes the function right away instead of after two minutes.

Option D: `setInterval(sayHello, 120000);`

Incorrect.

`setInterval()` repeats execution every 120,000 ms.

The question requires executing the function once, so this cannot be correct.

QUESTION 4

Refer to the code below:

```
01 let country = {  
02   get capital() {  
03     let city = Number("London");  
04  
05     return {  
06       cityString: city.toString(),  
07     }  
08   }  
09 }
```

Which value can a developer expect when referencing `country.capital.cityString`?

- A. An error
- B. NaN
- C. undefined
- D. 'London'

Answer: B

Explanation:

In the getter:

```
let city = Number("London");
```

The Number() constructor attempts to convert the string "London" into a numeric value.

"London" is not a valid numeric string.

When JavaScript attempts numeric conversion of a non-numeric string:

```
Number("London") NaN
```

Next line:

```
city.toString()
```

When city is NaN, calling .toString() yields:

```
NaN.toString() "NaN"
```

The getter returns:

```
{  
  cityString: "NaN"  
}
```

The question asks:

Which value can a developer expect?

The multiple-choice options include NaN, but not "NaN".

Given the available choices, the intended correct answer is B (NaN) because the root cause is the Number("London") conversion returning NaN.

QUESTION 5

Which statement accurately describes an aspect of promises?

- A. Arguments for the callback function passed to .then() are optional.
- B. .then() cannot be added after a catch.
- C. .then() manipulates and returns the original promise.
- D. In a .then() function, returning results is not necessary since callbacks will catch the result of a previous promise.

Answer: A

Explanation:

It is the only statement that accurately describes built-in Promise behavior:

.then() accepts optional arguments.

QUESTION 6

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.

Answer: C

Explanation:

Animal class:

Has a constructor that sets this.name.

Has a makeSound method that logs a message using this.name.

Dog class:

Extends Animal.

Its constructor calls super(name) to run the parent constructor, which sets this.name.

It overrides makeSound() to log:

```
console.log(`${this.name} is barking.`);
```

Instantiation:

```
let myDog = new Dog('Puppy');
```

```
myDog.makeSound();
```

```
new Dog('Puppy');
```

Calls Dog constructor with name = 'Puppy'.

Inside, super(name) sets this.name = 'Puppy' in the Animal constructor.

Then this.name = name; in the Dog constructor keeps it as 'Puppy'.

```
myDog.makeSound();
```

Calls the overridden makeSound() in Dog.

Logs: "Puppy is barking."

No reference errors, no undefined; the output is:

Puppy is barking.

QUESTION 7

Refer to the code below:

```
const pi = 3.1415926;
```

What is the data type of pi?

- A. Number
- B. Float
- C. Double
- D. Decimal

Answer: A

Explanation:

In JavaScript, there is only one numeric type for ordinary numbers: number.

JavaScript number is a double-precision 64-bit binary format (IEEE 754).

There is no separate float, double, or decimal type in core JavaScript.

So:

```
typeof pi === 'number';
```

Hence, the correct answer is A.

QUESTION 8

Refer to the code snippet:

```
01 function getAvailableilityMessage(item) {  
02   if (getAvailableility(item)) {  
03     var msg = "Username available";  
04   }  
05   return msg;  
06 }
```

What is the return value of msg when getAvailableilityMessage("newUserName") is executed and getAvailableility("newUserName") returns false?

- A. "newUserName"
- B. "msg is not defined"
- C. undefined
- D. "Username available"

Answer: C

Explanation:

Key details:

var has function scope, not block scope.

Declaration var msg is hoisted to the top of the function, but initialization only happens if the if condition is true.

Effectively, the function behaves like:

```
function getAvailableilityMessage(item) {  
  
  var msg; // hoisted declaration  
  
  if (getAvailableility(item)) {  
  
    msg = "Username available";  
  
  }  
}
```

```
}  
return msg;  
}
```

Now, given:

getAvailableility("newUserName") returns false.

Execution:

The if condition is false, so the body does not execute.

Therefore msg is declared but never assigned.

In JavaScript, an uninitialized variable that has been declared with var has the value undefined.

Thus:

```
return msg; // returns undefined
```

QUESTION 9

Refer to the code below:

```
01 function changeValue(param) {  
02   param = 5;  
03 }  
04 let a = 10;  
05 let b = a;  
06  
07 changeValue(b);  
08 const result = a + ' - ' + b;
```

What is the value of result when the code executes?

- A. 5-5
- B. 5-10
- C. 10-5
- D. 10-10

Answer: D

Explanation:

We must understand pass-by-value for primitives in JavaScript.

Initial values:

let a = 10;

let b = a; // b gets a copy of the value 10

So:

a is 10

b is 10 (independent copy)

Function call:

changeValue(b);

Function definition:

```
function changeValue(param) {
```

```
  param = 5;
```

```
}
```

param receives the value of b, which is 10.

Inside the function, param is a local variable.

param = 5; changes only this local copy.

It does not affect b outside the function.

After the function call:

a is still 10.

b is still 10.

Result:

```
const result = a + ' - ' + b;
```

a is 10.

b is 10.

String concatenation: '10 - 10'.

So result is:

"10 - 10"

Therefore, the correct option is D.

QUESTION 10

Universal Containers (UC) just launched a new landing page, but users complain that the website is slow. A developer found some functions 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.

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

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

- A. console.timeLog()
- B. console.trace()
- C. console.timeStamp()
- D. console.getTime()

Answer: A

Explanation:

JavaScript consoles (browsers and Node) provide timing utilities:

console.time(label) - starts a timer with the given label.

console.timeEnd(label) - stops the timer and logs the total elapsed time.

console.timeLog(label) - logs the current elapsed time without stopping the timer.

To measure the time taken by each function separately while still using a single overall timer, you can log intermediate times:

```
console.time('Performance');  
  
maybeAHeavyFunction();  
  
console.timeLog('Performance', 'after maybeAHeavyFunction');  
  
thisCouldTakeTooLong();  
  
console.timeLog('Performance', 'after thisCouldTakeTooLong');  
  
orMaybeThisOne();  
  
console.timeLog('Performance', 'after orMaybeThisOne');
```

```
console.timeEnd('Performance');
```

Each `console.timeLog('Performance')` call prints the elapsed time since `console.time('Performance')` was started, so you can infer how long each function took.

Note: `console.endTime` in the original snippet is incorrect; the real method is `console.timeEnd`.

QUESTION 11

Given the code below:

```
01 const delay = async delay => {  
02   return new Promise((resolve, reject) => {  
03     console.log(1);  
04     setTimeout(resolve, delay);  
05   });  
06 };  
  
07  
08 const callDelay = async () => {  
09   console.log(2);  
10   const yup = await delay(1000);  
11   console.log(3);  
12 };  
  
13  
14 console.log(4);  
15 callDelay();  
16 console.log(5);
```

What is logged to the console?

- A. 4 2 1 5 3
- B. 4 2 1 5 3
- C. 1 4 2 3 5
- D. 4 5 1 2 3

Answer: A

Explanation:

Execution order:

Top-level code runs synchronously:

Line 14: `console.log(4);` -> logs 4.

Line 15: `callDelay();` is called.

Inside `callDelay`:

Line 9: `console.log(2);` -> logs 2.

Line 10: `await delay(1000);`:

Calls `delay(1000)`.

Inside `delay(1000)`:

Line 3: `console.log(1);` -> logs 1.

Line 4: `setTimeout(resolve, delay);` schedules `resolve` in 1000 ms.

`delay` returns a pending Promise. `await` pauses `callDelay` here and returns control to the event loop.

Back to top-level:

Line 16: `console.log(5);` -> logs 5.

So synchronous log sequence is: 4, 2, 1, 5.

After ~1000 ms:

The `setTimeout` in `delay` resolves the Promise.

The `await` in `callDelay` resumes.

Line 11: `console.log(3);` -> logs 3.

Final log order: 4 2 1 5 3.

Both A and B show the same sequence; one must be chosen, so A is correct.

QUESTION 12

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?

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

Answer: D

Explanation:

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.

QUESTION 13

Refer to the code below:

```
flag();

function flag() {

  console.log('flag');

}

const anotherFlag = () => {

  console.log('another flag');

}

anotherFlag();
```

What is result of the code block?

- A. The console logs only 'flag'.
- B. An error is thrown.
- C. The console logs 'flag' and then an error is thrown.
- D. The console logs 'flag' and 'another flag'.

Answer: D

Explanation:

Key points:

Function declarations are hoisted (their definitions are available before the line where they appear).

Function expressions assigned to const or let are not hoisted as callable functions, but here anotherFlag is only used after it is defined.

Step-by-step:

flag(); at the top:

flag is a function declaration defined later; due to hoisting, it is available.

It logs 'flag'.

The declaration:

```
function flag() {  
  console.log('flag');  
}
```

is already in effect (hoisted before execution).

const anotherFlag = () => { console.log('another flag'); } defines anotherFlag as an arrow function.

anotherFlag(); is called after its definition; this is valid and logs 'another flag'.

No errors occur. The console output is:

flag

another flag

So option D is correct.

QUESTION 14

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     throw err;  
09   } finally {  
10     second = 'When';  
11   }  
12 } catch (err) {
```

```
13 second = 'Where';  
14 }
```

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

- A. first is Who and second is Where.
- B. first is Why and second is Where.
- C. first is Who and second is When.
- D. first is Why and second is When.

Answer: B

Explanation:

Initial values:

first = 'Who'

second = 'What'

Execution:

Inner try/catch/finally:

Line 05: throw new Error('Sad trombone');

Control goes to the inner catch.

Inner catch (lines 06?8):

```
catch (err) {
```

```
first = 'Why';
```

```
throw err;
```

```
}
```

first is set to 'Why'.

The error is rethrown.

Inner finally (lines 09?1):

```
finally {
```

```
second = 'When';
```

```
}
```

finally runs whether or not there was an error.

second becomes 'When'.

After inner finally, the rethrown error continues to propagate to the outer catch.

Outer catch (lines 12-14):

```
} catch (err) {  
  second = 'Where';  
}
```

Because the inner try rethrew, the outer catch runs.

It sets second = 'Where'.

Final values:

first was changed to 'Why' in the inner catch and never changed again.

second became 'When' in the inner finally, and then 'Where' in the outer catch.

So:

first is 'Why'

second is 'Where'

Option B is correct.

QUESTION 15

Refer to the code below:

```
const searchText = 'Yay! Salesforce is amazing!';  
  
let result1 = searchText.search(/sales/i);  
  
let result2 = searchText.search(/sales/);  
  
console.log(result1);  
  
console.log(result2);
```

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

- A. > 5 > undefined
- B. > 5 > 0
- C. > true > false
- D. > 5 > -1

Answer: D

Explanation:

String: "Yay! Salesforce is amazing!"

Index positions:

'Y' at 0, 'a' at 1, 'y' at 2, '!' at 3, space at 4, 'S' at 5, 'a' at 6, 'l' at 7, 'e' at 8, 's' at 9, etc.

Substring "Sales" starts at index 5.

String.prototype.search with a regex returns the index of the first match or -1 if there is no match.

```
searchText.search(/sales/i);
```

/sales/i is case-insensitive because of the i flag.

It matches "Sales" beginning at index 5.

So result1 is 5.

```
searchText.search(/sales/);
```

/sales/ is case-sensitive.

It requires lowercase "sales".

The text has "Sales" with uppercase S, so this does not match.

search returns -1 when there is no match.

So result2 is -1.

Console output:

First log: 5

Second log: -1

Option D matches this.

Thank You for Trying Our Product

Lead2pass Certification Exam Features:

- ★ More than **99,900** Satisfied Customers Worldwide.
- ★ Average **99.9%** Success Rate.
- ★ **Free Update** to match latest and real exam scenarios.
- ★ **Instant Download** Access! No Setup required.
- ★ Questions & Answers are downloadable in **PDF** format and **VCE** test engine format.
- ★ Multi-Platform capabilities - **Windows, Laptop, Mac, Android, iPhone, iPod, iPad**.
- ★ **100%** Guaranteed Success or **100%** Money Back Guarantee.
- ★ **Fast**, helpful support **24x7**.



View list of all certification exams: <http://www.lead2pass.com/all-products.html>



10% Discount Coupon Code: ASTR14