Version: 16.0

Question: 1

A team at Universal Containers works on a big project and uses yarn to manage 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 added the dependency as a dev dependency, and YARN_ENV is set to production.

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

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

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

Answer: C

E xpl anati on:

_____

Comprehensive and Detailed Explanation From Exact Extract JavaScript Knowledge

In JavaScript server-side development using Node.js, dependency management is typically handled through package managers such as npm or yarn. These tools categorize installed packages into:

dependencies – required for running the application in any environment

devDependencies – required only during development (testing tools, build tools, documentation generators, etc.)

When a package is installed using:

yarn add <package> --dev

it is placed under the "devDependencies" section of package.json.

_____

Behavior of Production Mode

Node.js uses the environment variable:

N ODE _ E N V= producti on

When this environment variable is set to production, both npm and Yarn follow the standard Node.js convention and skip installing devDependencies. This is done to optimize production builds and reduce deployment size. This is a known and documented behavior in Node.js package management tools.

Therefore, if:

The developer added the date-manipulation library as a dev dependency, and

Other team members execute yarn in an environment where NODE_ENV=production is set,

then Yarn will not install that dependency because devDependencies are intentionally excluded in production mode.

This explains the behavior described in the question.

_____

Why the Other Options Are Incorrect

Option A:

"YARN_ENV is set to production" is incorrect because Yarn does not use the variable YARN_ENV for dependency installation behavior. Node.js tools use NODE_ENV, not YARN_ENV.

Option B:

This is incorrect because Yarn automatically writes dependencies into package.json. Unlike older npm versions, there is no need for the --save flag.

Option D:

There is no such option as --add. The correct syntax is simply:

yarn add <package>

Missing an option that does not exist cannot be the cause.

_____

JavaScript Knowledge Reference (Text-Based, No Links)

Node.js uses the environment variable NODE_ENV to determine production or development mode.

Package managers (npm and Yarn) follow the rule that when NODE_ENV=production, only "dependencies" are installed and "devDependencies" are skipped.

Yarn automatically persists installed packages to package.json without requiring --save.

Yarn uses the command yarn add to add dependencies; there is no --add flag.

Question: 2

Refer to the code below:

```
01leto={
02 getjs(){
03    let city1 = String('St. Louis');
04    let city2 = String('New York');
05
06    return {
07     firstCity: city1.toLowerCase(),
08     secondCity: city2.toLowerCase(),
09    }
10  }
11}
```

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

A. undefined

B. An error

C. 'New York'

D. 'new york'

Answer: D

E xpl anati on:

_____

Comprehensive and Detailed Explanation From Exact Extract JavaScript Knowledge

1. Getter Functions in JavaScript

In JavaScript, when an object uses the get keyword, it defines a getter method. Accessing a getter

property executes the function and returns its value. Thus:

o.js

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

get js() { ... }

and returns the object inside the return block.

---

2. Behavior of String() and toLowerCase()

Inside the getter:

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

let city2 = String('New York');

String() creates a string value.

Then, the returned object is constructed as:

{

firstCity: city1.toLowerCase(),

secondCity: city2.toLowerCase(),

}

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

Therefore:

city2.toLowerCase()

returns:

'new york'

---

3. Referencing the Property

When the developer writes:

o. js. se condC i ty

the following happens:

The getter js runs and returns an object.

The returned object includes the property:

secondCity: 'new york'

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

Therefore, the correct value is 'new york'.

_____

Why the Other Options Are Incorrect

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

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

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

_____

JavaScript Knowledge Reference (Text-Based)

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

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

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

Question: 3

Refer to the code below:

```
01 x = 3.14;
02
03 function myFunction() {
04   'use strict';
05 y=x;
06}
07
```

08z=x;

09 myFunction();

Considering the implications of 'use strict' on line 04, which three statements describe the execution of the code?

A. 'use strict' is hoisted, so it has an effect on all lines.

B. z is equal to 3.14.

C. 'use strict' has an effect between line 04 and the end of the file.

D. 'use strict' has an effect only on line 05.

E. Line 05 throws an error.

Answer: B, D, E

E xpl anati on:

Comprehensive and Detailed Explanation From Exact Extract JavaScript knowledge:

Behavior of non-strict global code

The script does not begin with a 'use strict' directive at the top level, so the global code (outside any function) runs in non-strict (sloppy) mode.

Line 01: x = 3.14;

In non-strict mode, assigning to an undeclared identifier (no var, let, or const) creates an implicit global variable. So after line 01, x exists and equals 3.14.

Line 08: z = x;

This also runs in non-strict mode. Since x is already defined (from line 01), z is set to 3.14. Therefore, statement B is correct: z is equal to 3.14.

Scope of 'use strict' inside a function

The line:

'use strict';

inside myFunction is a directive prologue for that function, not for the entire file. This means:

Strict mode applies only within the body of myFunction, from the directive to the end of that functi on.

It does not retroactively affect code before the function or code outside of it.

Therefore:

Statement A is incorrect: 'use strict' is not “hoisted” to affect the whole file. It only affects the function body.

Statement C is incorrect: strict mode does not apply from line 04 to the end of the file; it only applies within myFunction, not to global lines like 01, 08, or 09.

Execution of myFunction in strict mode

When myFunction is called on line 09:

```
function myFunction() {
  'use strict';
  y=x;
}
```

Within this function:

Strict mode is active for its body.

In strict mode, assigning to an undeclared variable (like y here) is not allowed and results in a ReferenceError at runtime.

So line 05:

```
y=x;
```

throws a ReferenceError because y is not declared with var, let, or const.

Therefore, statement E is correct: line 05 throws an error.

Why statement D is considered correct

Statement D says:

'use strict' has an effect only on line 05.

In terms of execution in this specific code:

The only executable statement in myFunction that is affected by strict mode is the assignment on line 05.

The directive itself on line 04 is not a “normal” runtime operation; it is a directive that sets the mode.

There are no other statements inside the function that behave differently under strict mode; only the line that assigns to the undeclared variable shows a strict-mode effect.

So, describing the practical execution behavior of this snippet, strict mode manifests its effect only on line 05, making statement D correct in this context.

Summary of each option:

A: Incorrect – strict does not apply to all lines in the file.

B: Correct – z becomes 3.14 in non-strict global code.

C: Incorrect – strict does not affect code outside the function.

D: Correct – in this code, the only behavioral effect of strict mode is on line 05.

E: Correct – line 05 throws a ReferenceError due to assignment to undeclared y under strict mode.

JavaScript knowledge references (descriptive, no links):

In non-strict (sloppy) mode, assigning to an undeclared identifier creates a global variable.

'use strict' inside a function body enables strict mode for that function only.

In strict mode, assigning to an undeclared variable results in a ReferenceError.

Directive prologues ('use strict') affect only their containing function or script, not other scopes.

==========

Question: 4

Refer to the code below:

```
01 let total = 10;
02 const interval = setInterval(() => {
03 total++;
04 clearInterval(interval);
05 total++;
06 }, 0);
07 total++;
08 console.log(total);
```

Considering that JavaScript is single-threaded, what is the output of line 08 after the code executes?

A.11

B.12

C.10

D.13

Answer: A

E xpl anati on:

Comprehensive and Detailed Explanation From Exact Extract JavaScript knowledge:

Synchronous execution order

JavaScript executes code in a single thread, following a well-defined order:

All synchronous code runs first, line by line.

Asynchronous callbacks (like those scheduled with setInterval or setTimeout) are placed into the event queue and executed only after the current call stack is empty.

Let's follow the code step by step:

Line 01:

let total = 10;

total is initialized with the value 10.

Line 02:

```
const interval = setInterval(() => {
 total++;
 clearInterval(interval);
 total++;
}, 0);
```

setInterval schedules the callback function to run repeatedly after a delay of at least 0 milliseconds, but it does not run immediately. The callback is added to the timer queue and will be invoked after the current synchronous script finishes and the event loop gets to process timer callbacks.

At this point, interval holds the interval ID, but the callback has not executed yet.

Line 07:

total++;

This is still synchronous, so it runs before any scheduled callbacks.

total was 10, now it becomes 11.

Line 08:

consol e . l og (total);

At this moment, the interval callback has still not run (because the event loop has not yet processed the timer queue).

So total is 11, and console.log(total); outputs 11.

Therefore, the value printed at line 08 is 11, making option A correct.

What happens after the log (for understanding, not affecting the answer)

After the main script finishes, the event loop processes the timer callback for setInterval:

C al l back:

```
() => {
  total++;              // from 11 to 12
  clearInterval(interval); // cancels further executions
  total++;              // from 12 to 13
}
```

So eventually total becomes 13, but this happens after console.log(total) has already executed. Since the question asks specifically for the output at line 08, the asynchronous updates do not change that line's output.

Why other options are incorrect

Option B (12): This would require the callback to run before the log, which does not happen because asynchronous callbacks are queued and executed after the current stack finishes.

Option C (10): Ignores the total++ on line 07.

Option D (13): This is the final value after the callback finishes, but it occurs after the console.log line executes, not at the time line 08 runs.

JavaScript knowledge references (descriptive, no links):

JavaScript is single-threaded and uses an event loop with a call stack and task queues.

setInterval schedules callbacks to run asynchronously after a minimum delay; the callback never runs before the current synchronous code finishes.

Synchronous statements like total++ on line 07 execute before any queued interval callback.

Question: 5

Refer to the code below:

```
01 const objBook = {
02   title: 'JavaScript',
03};
04 Object.preventExtensions(objBook);
05 const newObjBook = objBook;
06 newObjBook.author = 'Robert';
```

What are the values of objBook and newObjBook respectively?

A. { title: "JavaScript" }

{ title: "JavaScript" }

B. { author: "Robert", title: "JavaScript" }

undefined

C. { author: "Robert" }

{ author: "Robert", title: "JavaScript" }

D. { author: "Robert", title: "JavaScript" }

{ author: "Robert", title: "JavaScript" }

Answer: A

E xpl anati on:

_____

Comprehensive and Detailed Explanation From Exact Extract JavaScript Knowledge

Object.preventExtensions(obj)

This built-in JavaScript method marks an object so that no new properties can be added to it.

Existing properties can still be read and updated, but adding new ones is disallowed.

const newObjBook = objBook;

Both variables reference the same object in memory. JavaScript objects are assigned by reference, not copied.

newObjBook.author = "Robert";

Because the object has been marked as non-extensible, JavaScript will not allow new properties to be added.

The behavior depends on mode:

In non-strict mode: the assignment silently fails and does nothing.

In strict mode: this would throw a TypeError.

Since nothing indicates strict mode, this is non-strict behavior, making the assignment fail silently.

Therefore, the object remains:

{ title: "JavaScript" }

Both objBook and newObjBook point to the same unchanged object.

This matches option A.

_____

JavaScript knowledge references (text-only)

Object.preventExtensions() prevents adding new properties.

Assigning an object to another variable copies the reference, not the object.

Adding a property to a non-extensible object silently fails in non-strict mode.

==========