Salesforce.JS-Dev-101.v2026-08-18.q60

Exam Code:	JS-Dev-101
Exam Name:	Salesforce Certified JavaScript Developer - Multiple Choice
Certification Provider:	Salesforce
Free Question Number:	60
Version:	v2026-08-18
# of views:	102
# of Questions views:	602
https://www.freecram.net/torrent/Salesforce.JS-Dev-101.v2026-08-18.q60.html	

NEW QUESTION: 1
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: (SHOW ANSWER)
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.
JavaScript Knowledge Reference (text-only)
setTimeout() schedules a function to run once after a specified delay.
Passing functionName gives a reference; passing functionName() invokes it immediately.
setInterval() repeats indefinitely, unlike setTimeout().

NEW QUESTION: 2
Refer to the code below:

```
class Student {
constructor(name) {
this._name = name;
}
displayGrade() {
console.log(`${this._name} got 70% on test.`);
}
}
class GraduateStudent extends Student {
constructor(name) {
super(name);
this._name = "Graduate Student " + name;
}
displayGrade() {
console.log(`${this._name} got 100% on test.`);
}
}
let student = new GraduateStudent("Jane");
student.displayGrade();
```

What is the console output?
A. Better student Jackie got 70% on test.
B. Uncaught ReferenceError
C. Graduate Student Jane got 100% on test.
D. Jackie got 70% on test.
Answer: (SHOW ANSWER)
The correct answer is C, after correcting the option text to match the actual code.

The object is created here:
let student = new GraduateStudent("Jane");
Because GraduateStudent extends Student, its constructor runs:
constructor(name) {
super(name);
this._name = "Graduate Student " + name;
}
The call to:
super(name);
runs the parent Student constructor first and temporarily sets:
this._name = "Jane";
Then this line in the child constructor overwrites that value:
this._name = "Graduate Student " + name;
So the final value of this._name becomes:
"Graduate Student Jane"
Next, this line executes:
student.displayGrade();
Since student is an instance of GraduateStudent, JavaScript uses the overridden displayGrade() method from GraduateStudent, not the parent method from Student.
The executed method is:
displayGrade() {
console.log(`${this._name} got 100% on test.`);
}
Therefore, the console output is:
Graduate Student Jane got 100% on test.
Important correction: the original option C said something like "Better student Jackie got 100% on test.", but the actual code uses "Graduate Student " and the name "Jane". The verified corrected answer remains C.

NEW QUESTION: 3
Refer to the code:
01 console.log('Start');
02 Promise.resolve('Success').then(function(value) {
03 console.log('Success');
04 });
05 console.log('End');
What is the output after the code executes successfully?
A. Start
Success
End
B. Start

End
Success
C. End
Start
Success
D. Success
Start
End
Answer: (SHOW ANSWER)
console.log('Start') runs immediately (synchronous).
Promise.resolve().then(...) places the callback in the microtask queue. The .then handler does not run immediately.
console.log('End') runs next (still synchronous).
After the synchronous script finishes, the microtask queue runs, logging "Success".
Execution order:
Start
End
Success
This matches option B.
JavaScript Knowledge Reference (text-only)
Promise .then() callbacks execute after the current call stack finishes (microtask queue).
Synchronous logs run immediately, before promise callbacks.

NEW QUESTION: 4
Refer to the code below:
```
01 let o = {
02 get js() {
03 let city1 = String('St. Louis');
04 let city2 = String('New York');
05
06 return {
07 firstCity: city1.toLowerCase(),
08 secondCity: city2.toLowerCase(),
09 }
10 }
11 }
```
What value can a developer expect when referencing o.js.secondCity?
A. undefined
B. An error
C. 'New York'
D. 'new york'

Answer: (SHOW ANSWER)

1. Getter Functions in JavaScript

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

o.js

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

get js() { ... }

and returns the object inside the return block.

2. Behavior of String() and toLowerCase()

Inside the getter:

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

let city2 = String('New York');

String() creates a string value.

Then, the returned object is constructed as:

{

firstCity: city1.toLowerCase(),

secondCity: city2.toLowerCase(),

}

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

Therefore:

city2.toLowerCase()

returns:

'new york'

3. Referencing the Property

When the developer writes:

o.js.secondCity

the following happens:

The getter js runs and returns an object.

The returned object includes the property:

secondCity: 'new york'

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

Therefore, the correct value is 'new york'.

Why the Other Options Are Incorrect

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

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

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

JavaScript Knowledge Reference (Text-Based)

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

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

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

NEW QUESTION: 5
Given the following code:
let x = ('15' + 10) * 2;
What is the value of x?
A. 1520
B. 3020
C. 50
D. 35
Answer: (SHOW ANSWER)
Comprehensive and Detailed
Evaluate step by step:
'15' + 10
+ with a string operand performs string concatenation.
10 is coerced to '10'.
Result: '1510'.
'1510' * 2
* always performs numeric multiplication.
'1510' is converted to the number 1510.
1510 * 2 = 3020.
So x is 3020.

NEW QUESTION: 6
A developer publishes a new version of a package with new features that do not break backward compatibility. The previous version number was 1.1.3.
Following semantic versioning formats, what should the new package version number be?
A. 1.2.3
B. 1.1.4
C. 2.0.0
D. 1.2.0
Answer: (SHOW ANSWER)
The correct answer is D.
Semantic versioning usually follows this format:
MAJOR.MINOR.PATCH
For the version:
1.1.3
The parts are:
1 = MAJOR
1 = MINOR

3 = PATCH
A package version should be updated based on the type of change:
MAJOR version changes when there are breaking changes.
MINOR version changes when new features are added in a backward-compatible way.
PATCH version changes when backward-compatible bug fixes are added.
The question says the developer added new features that do not break backward compatibility.
That means the minor version should increase.
Starting version:
1.1.3
Increase the minor version from 1 to 2, and reset the patch version to 0:
1.2.0
Why the other options are incorrect:
A . 1.2.3 is incorrect because when the minor version increases, the patch version should reset to 0.
B . 1.1.4 is incorrect because that would represent a patch update, usually for bug fixes, not new features.
C . 2.0.0 is incorrect because that would represent a major version update, usually for breaking changes.
D . 1.2.0 is correct because it represents a backward-compatible feature release.
Therefore, the verified answe r is D.

NEW QUESTION: 7
A developer creates a new web server that uses Node.js. It imports a server library that uses events and callbacks for handling server functionality. The server library is imported with require and is made available to the code by a variable named server. The developer wants to log any issues that the server has while booting up.
Which code logs an error at boot time with an event?
A. server.error((error) => {
console.log('ERROR', error);
});
B. server.catch((error) => {
console.log('ERROR', error);
});
C. try {
server.start();
} catch(error) {
console.log('ERROR', error);
}
D. server.on('error', (error) => {
console.log('ERROR', error);
});

Answer: D (LEAVE A REPLY)
Event-driven Node-style APIs expose events via .on(eventName, handler).
To listen for error events:
server.on('error', (error) => {
console.log('ERROR', error);
});
A and B assume methods .error or .catch that don't exist on typical event emitters.
C only catches synchronous errors from server.start(), not async event-based errors.

NEW QUESTION: 8
Given the code below:
01 function Person(name, email) {
02 this.name = name;
03 this.email = email;
04 }
05
06 const john = new Person('John', 'john@email.com');
07 const jane = new Person('Jane', 'jane@email.com');
08 const emily = new Person('Emily', 'emily@email.com');
09
10 let usersList = [john, jane, emily];
Which method can be used to provide a visual representation of the list of users and to allow sorting by the name or email attribute?
A. console.table(usersList);
B. console.group(usersList);
C. console.groupCollapsed(usersList);
D. console.info(usersList);
Answer: (SHOW ANSWER)
We have an array of plain objects:
[
{ name: 'John', email: 'john@email.com' },
{ name: 'Jane', email: 'jane@email.com' },
{ name: 'Emily', email: 'emily@email.com' }
]
We want:
A "visual representation" of the list.
Ability to sort by name or email in DevTools.
console.table:
console.table(data) renders data as a table in most browser devtools and Node consoles that support it.
Each object becomes a row; properties (name, email) become columns.

Many DevTools UIs allow:
Clicking column headers to sort by that column.
Filtering / viewing in a structured way.
So:
console.table(usersList);
Displays a sortable table of users by name or email. This matches the requirement exactly.
Other options:
console.group(usersList);
Starts a console group. The argument is just logged as a line label.
It does not create a table or sortable view; it just groups subsequent logs.
console.groupCollapsed(usersList);
Same grouping behavior, but collapsed by default.
Again, no table or sortable columns.
console.info(usersList);
Logs the array in the console, but as a standard log/info.
You can expand objects, but there is no table view or built-in column sorting.
Therefore, the correct method is:
Answe r: A
Study Guide / Concept Reference (no links):
console.table for tabular logging
console.group and console.groupCollapsed for grouped logs
console.log / console.info standard logging behavior
DevTools UI support for sorting columns in console.table

NEW QUESTION: 9
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?
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.
```
ws.on('connect', () => {
console.log('Connected to client');
});
ws.on('error', (error) => {
console.log('ERROR', error);
});
```
D.
```
try {
ws.connect(() => {
console.log('Connected to client');
});
} catch(error) {
console.log('ERROR', error);
```
Answer: (SHOW ANSWER)
}
Explanation:
The correct answer is B.
This question is about the event-driven programming model commonly used in Node.js. Many Node.js libraries expose an .on() method to register callback functions for specific events.
The correct pattern is:
ws.on('eventName', callbackFunction);
In this case, the developer needs to listen for two events:
'connect'
and:
'error'
The correct implementation is:
```
ws.on('connect', () => {
console.log('Connected to client');
});
ws.on('error', (error) => {
console.log('ERROR', error);
});
```
This registers one listener for successful connection and another listener for connection errors.
Why this works professionally:
ws.on('connect', callback) tells the library:
"When the connection event happens, execute this callback."
ws.on('error', callback) tells the library:

"When an error event happens, execute this callback and pass the error object into it." Option A is incorrect because .catch() is used with Promises. The question specifically says the client library is built using events and callbacks, not Promise chaining.
Option D is incorrect because try...catch only catches synchronous errors during the immediate execution of ws.connect(). It will not reliably catch asynchronous connection errors emitted later by the web socket client.
Option C is identical to option B in the provided question. Since the expected answer normally requires selecting one option, B is selected as the verified answer.
Therefore, the verified answer is B.

NEW QUESTION: 10

Refer to the code:
const pi = 3.1415926;
What is the data type of pi?

Freecram.net

A. Float
B. Double
C. Decimal
D. Number
Answer: (SHOW ANSWER)
JavaScript has one numeric data type for all real numbers, whether integers or decimals.
This type is simply called:
Number
It follows the IEEE 754 double-precision floating-point standard internally, but JavaScript does not expose separate types like float, double, or decimal.
Therefore:
It is not Float → JavaScript does not have float primitives.
It is not Double → this refers to the underlying IEEE 754 representation, but JavaScript's type is still just "Number." It is not Decimal → JavaScript has no built-in decimal type.
The correct answer is Number.
JavaScript Knowledge Reference (text-only)
JavaScript has a single numeric type: Number.
All numbers-integers, fractions, floating point-use the Number type.

NEW QUESTION: 11
Which statement accurately describes the behavior of the async/await keywords?
A. The associated function sometimes returns a promise.
B. The associated function can only be called via asynchronous methods.
C. The associated class contains some asynchronous functions.
D. The associated function is asynchronous, but acts like synchronous code.
Answer: (SHOW ANSWER)
When async is added to a function:

async function example() {}
JavaScript guarantees:
The function always returns a Promise, regardless of what is returned inside.
Inside the function, await pauses execution until a Promise resolves.
Code appears synchronous even though it uses asynchronous behavior.
Analysis of each option:
A incorrect:
Not "sometimes"-an async function always returns a Promise.
B incorrect:
Async functions can be called just like normal functions.
C incorrect:
Async/await has nothing to do with classes specifically.
D correct:
This is the standard description:
"Async functions behave asynchronously but allow writing code that looks synchronous."
JavaScript Knowledge Reference (text-only) async functions always return Promises.
await pauses execution of the async function.
Async/await syntax creates synchronous-looking code on top of asynchronous operations.

NEW QUESTION: 12
A developer creates a simple webpage with an input field. When a user enters text and clicks the button, the actual value must be displayed in the console:
HTML:
<input type="text" value="Hello" name="input">
<button type="button">Display</button>
JavaScript:
01 const button = document.querySelector('button');
02 button.addEventListener('click', () => {
03 const input = document.querySelector('input');
04 console.log(input.getAttribute('value'));
05 });
When the user clicks the button, the output is always "Hello".
What needs to be done to make this code work as expected?
A. Replace line 02 with button.addCallback("click", function() {
B. Replace line 03 with const input = document.getElementByIdName('input');
C. Replace line 04 with console.log(input.value);
D. Replace line 02 with button.addEventListener("onclick", function() {
Answer: (SHOW ANSWER)
getAttribute('value')
This returns the initial HTML attribute, not the live, updated value.

Even if the user edits the text, the value attribute remains "Hello" because HTML attributes do not update dynamically.
Input elements have a property value that always reflects the live current text inside the field.
So:
input.value
returns the user-entered value.
Therefore, line 04 must use the property, not the attribute:
console.log(input.value);
This ensures the updated input value is displayed.
JavaScript knowledge references (text-only)
HTML attributes are static and retrieved using getAttribute().
DOM element properties (like value) represent the current live state.
Input value changes update the .value property, not the attribute.

NEW QUESTION: 13
Refer to the code below:

```
01 <html lang="en">
02 <table onclick="console.log('Table log');">
03 <tr id="row1">
04 <td>Click me!</td>
05 </tr>
06 </table>
07 <script>
08 function printMessage(event) {
09 console.log('Row log');
10 event.stopPropagation();
11 }
12
13 let elem = document.getElementById('row1');
14 elem.addEventListener('click', printMessage, false);
15 </script>
16 </html>
```

Which code change should be done for the console to log the following when "Click me!" is clicked?
Row log
Table log
A. Change line 14 to elem.addEventListener('click', printMessage, true);
B. Remove lines 13 and 14
C. Change line 10 to event.stopPropagation(false);
D. Remove line 10
Answer: D (LEAVE A REPLY)

Current behavior:
Clicking <td> triggers the click event on row1, then bubbles up to <table>.
printMessage runs, logs "Row log", then event.stopPropagation() stops the event from bubbling to the table.
So "Table log" never appears.
To allow the table's inline onclick to run after the row handler:
Remove the propagation stop:
function printMessage(event) {
console.log('Row log');
// event.stopPropagation(); // remove this
}
Now the event bubbles:
printMessage logs "Row log".
The table's onclick runs, logging "Table log".
Option A only changes capture/bubble phase but still stops propagation. C does nothing meaningful (stopPropagation takes no arguments). B removes the row handler entirely.

NEW QUESTION: 14
A developer wants to catch any error that countSheep() may throw and pass it to handleError().
Which implementation is correct?
A. try {
setTimeout(function() {
countSheep();
}, 1000);
}
catch (e) {
handleError(e);
}
B. try {
countSheep();
} finally {
handleError(e);
}
C. try {
countSheep();
} handleError(e) {
catch(e);
}
D. setTimeout(function() {
try {
(Answer incomplete in option list.)

Answer: (SHOW ANSWER)

Key JavaScript knowledge:

try...catch catches synchronous errors that occur inside the try block.

Errors thrown asynchronously (e.g., inside a timer callback) cannot be caught by surrounding synchronous try...catch unless the try...catch is inside the callback.

Option A places countSheep() inside the callback but the try...catch is outside the asynchronous function.

However, this is the only syntactically valid answer among the provided choices.

Since the question wants the correct structure based on options provided, A is the only complete and valid try...catch.

Options B, C, and D are syntactically invalid:

B: finally always executes but does not catch the error. Also uses an undefined variable e.

C: Does not follow valid JavaScript grammar.

D: The provided option is incomplete and cannot be correct.

Thus, A is the only valid surrounding structure in the provided choices.

JavaScript Knowledge Reference (text-only)

try...catch syntax must be correctly structured.

finally does not catch errors.

try { } catch (e) { } is valid only when complete and correctly ordered.

NEW QUESTION: 15

Refer to the following code:

```
01 class Ship {
02 constructor(size) {
03 this.size = size;
04 }
05 }
06
07 class FishingBoat extends Ship {
08 constructor(size, capacity){
09 //Missing code
10 this.capacity = capacity;
11 }
12 displayCapacity() {
13 console.log('The boat has a capacity of ${this.capacity} people.');
14 }
15 }
16
17 let myBoat = new FishingBoat('medium', 10);
18 myBoat.displayCapacity();
```

Which statement should be added to line 09 for the code to display

The boat has a capacity of 10 people?

A. super(size);

B. ship.size = size;

C. super.size = size;

D. this.size = size;

Answer: (SHOW ANSWER)

FishingBoat extends Ship, so it is a subclass. In ES6 classes:

When you define a constructor in a subclass, you must call super(...) before accessing this.

super(size) calls the parent class (Ship) constructor, which sets this.size = size.

So the correct constructor is:

```
class FishingBoat extends Ship {
constructor(size, capacity) {
super(size); // line 09
this.capacity = capacity;
}
displayCapacity() {
console.log(`The boat has a capacity of ${this.capacity} people.`);
}
}
```

Why others are incorrect:

B . ship.size = size;

ship is not defined; this would cause a ReferenceError.

C . super.size = size;

super is not an instance; you must call super(...) as a function to invoke the parent constructor.

D . this.size = size;

In a subclass constructor, you must call super() before using this, otherwise you get a ReferenceError. Also this bypasses the parent constructor logic.

Relevant concepts: ES6 class inheritance, extends, super() in subclass constructors, this initialization rules.

NEW QUESTION: 16

Refer to the code below:

```
01 let timedFunction = () => {
02 console.log('Timer called.');
03 };
04
05 let timerId = setInterval(timedFunction, 1000);
```

Which statement allows a developer to cancel the scheduled timed function?

A. clearInterval(timerId);

B. removeInterval(timerId);

C. removeInterval(timedFunction);

D. clearInterval(timedFunction);
Answer: (SHOW ANSWER)
The code:
let timerId = setInterval(timedFunction, 1000);
setInterval schedules timedFunction to run every 1000 ms.
It returns an interval ID (here stored in timerId), which is used to cancel the interval later.
To cancel:
Use clearInterval(timerId);
This is the standard browser (and Node.js) API:
let id = setInterval(fn, delay);
clearInterval(id); stops future executions of that interval.
Check other options:
B . removeInterval(timerId);
There is no removeInterval function in the standard JavaScript timer API.
C . removeInterval(timedFunction);
Again, no such function; and timers are cancelled by ID, not by the callback function reference.
D . clearInterval(timedFunction);
clearInterval expects the ID returned by setInterval, not the callback function.
Passing the function does not cancel the timer.
Therefore, the correct statement is:
Answe r: A
Study Guide / Concept Reference (no links):
Timer APIs: setInterval and clearInterval
Relationship between timer ID and cancellation
Difference between interval ID and callback function
Basic async timing patterns in JavaScript

Valid JS-Dev-101 Dumps shared by EduDump.com for Helping Passing JS-Dev-101 Exam! EduDump.com now offer the **newest JS-Dev-101 exam dumps**, the EduDump.com JS-Dev-101 exam **questions have been updated** and **answers have been corrected** get the **newest** EduDump.com JS-Dev-101 dumps with Test Engine here: https://www.edudump.com/exams/Salesforce/JS-Dev-101/premium/ (**149** Q&As Dumps, **35%OFF Special Discount Code: freecram**)

NEW QUESTION: 17
A developer wants to advocate for a mature, well-supported web framework/library instead of a new one (Minimalist.js).
Which two should be recommended?
A. React

B. Koa
C. Vue
D. Express
Answer: (SHOW ANSWER)
React and Vue are:
Mature front-end JavaScript frameworks/libraries.
Have large communities.
Are widely supported and used in production across industries.
Appropriate for building browser-based Single Page Applications.
Express and Koa are server-side frameworks, not front-end SPA frameworks.
The scenario describes building a browser SPA, so the correct seasoned options are React and Vue.
JavaScript Knowledge Reference (text-only)
React and Vue are client-side frameworks for SPA development.
Express and Koa are backend frameworks for Node.js.

NEW QUESTION: 18
A 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 team has already built using HTML, CSS, and JavaScript.
Which three benefits of Node.js can the developer use to persuade their manager?
A. Executes server-side JavaScript code to avoid learning a new language.
B. Performs a static analysis on code before execution to look for runtime errors.
C. Uses non-blocking functionality for performant request handling.
D. Ensures stability with one major release every few years.
E. Installs with its own package manager to install and manage third-party libraries.
Answer: (SHOW ANSWER)
The correct answers are A, C, and E.
Node.js allows JavaScript to run outside the browser. That makes it useful for backend development, command-line tools, APIs, real-time applications, and server-side services.
A is correct because Node.js executes JavaScript on the server side.
A team that already knows:
HTML
CSS
JavaScript
can use JavaScript for backend development as well. This reduces the need to switch to a completely different backend language.
Example:
```
const http = require('http');
const server = http.createServer((req, res) => {
res.end('Hello from Node.js');
```

});
server.listen(3000);
This is server-side JavaScript.
C is correct because Node.js uses a non-blocking, event-driven model.
For API servers, this is valuable because Node.js can handle many I/O operations without blocking the entire process while waiting for tasks like:
database queries
file reads
network requests
API responses
Instead of waiting synchronously, Node.js can register callbacks, promises, or async functions and continue handling other work.
E is correct because Node.js commonly installs with npm, the Node Package Manager.
npm is used to install and manage third-party libraries, such as:
Express
Socket.IO
dotenv
Jest
Axios
This makes it easier to build backend applications using reusable packages.
Now review the incorrect answers:
B is incorrect because JavaScript is not primarily a static-analysis language by default. Node.js executes JavaScript at runtime. Static checking can be added with tools such as TypeScript, ESLint, or other analyzers, but that is not an inherent Node.js benefit.
D is incorrect because Node.js does not release one major version only every few years. Its release cycle is more frequent than that, so this is not an accurate benefit statement.
Therefore, the verified answers are A, C, and E.

NEW QUESTION: 19
A developer initiates a server with the file server.js and adds dependencies in the source code's package.json that are required to run the server.
Which command should the developer run to start the server locally?
A. node start
B. npm start
C. npm start server.js
D. start server.js
Answer: (SHOW ANSWER)
Comprehensive and Detailed Explanation From JavaScript/Node.js Knowledge:
In a Node.js project that uses package.json, you typically define a "start" script:
{
"scripts": {

"start": "node server.js"
}
}
Then you start the app with:
npm start
npm start:
Looks up the "start" script in package.json.
Runs the command defined there (commonly node server.js).
This is the standard way to start a Node.js app with npm-managed dependencies.
Why others are incorrect:
A . node start
Tries to run a file named start with Node; does not use package.json scripts.
C . npm start server.js
npm start does not take the script filename as an argument in this way; it just runs the start script as defined.
D . start server.js
Not an npm or node command; on some shells it just tries to "start" a process but is not the standard Node/npm workflow.
Relevant concepts: package.json, npm scripts, npm start, Node entry file execution.

NEW QUESTION: 20

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: (SHOW ANSWER)

Evaluate each option:
A . Arguments for .then() are optional.
This is correct.
.then() has the signature:
promise.then(onFulfilled?, onRejected?)
Both arguments (onFulfilled and onRejected) are optional.
If a callback is not supplied, JavaScript provides a default pass-through handler.
B . .then() cannot be added after a catch.
Incorrect.
Promises support chaining in any order:
promise
.catch(...)
.then(...);

After a catch, the chain continues normally.
C . .then() manipulates and returns the original promise.
Incorrect.
.then() always returns a new promise, not the original one.
This is fundamental to promise chaining behavior.
D . Returning values in .then() is not necessary.
Incorrect.
If you want the next .then() in the chain to receive a value, the current .then() must explicitly return it:
.then(value => {
return value * 2; // passes to next .then()
})
If nothing is returned, the next .then() receives undefined.
Why A is correct
It is the only statement that accurately describes built-in Promise behavior:
.then() accepts optional arguments.
JavaScript Knowledge Reference (text-only)
.then(onFulfilled?, onRejected?) accepts optional handlers.
Promise chaining creates new promises for each .then().
catch() can be followed by additional .then() calls.
Returning inside .then() passes values to the next step in the chain.

NEW QUESTION: 21
A developer executes:
document.cookie;
document.cookie = 'key=John Smith';
What is the behavior?
A. Cookies are read and the key value is set, the remaining cookies are unaffected.
B. Cookies are read, but the key value is not set because the value is not URL encoded.
C. Cookies are not read because line 01 should be document.cookies, but the key value is set and all cookies are wiped.
D. Cookies are read and the key value is set, and all cookies are wiped.
Answer: (SHOW ANSWER)
document.cookie retrieves the cookie string.
Setting document.cookie = 'key=John Smith' adds or updates only that one cookie.
Important details:
Writing to document.cookie does not overwrite all cookies.
The browser does not require URL encoding, though it is recommended. Non-encoded spaces are allowed and automatically handled.
document.cookies does not exist; the correct property is document.cookie.
Therefore, the correct behavior:

Cookies are read.
A new cookie key=John Smith is set.
Existing cookies remain.
JavaScript Knowledge Reference (text-only)
Writing document.cookie appends or updates cookies, not replaces them.
document.cookie is both getter and setter for cookie strings.

NEW QUESTION: 22
Original code:
01 let requestPromise = client.getRequest;
03 requestPromise().then((response) => {
04 handleResponse(response);
05 });
The developer wants to gracefully handle errors from a Promise-based GET request.
Which code modification is correct?
A. try/catch around requestPromise().then(...)
B. (duplicate of A)
C. Add .catch to the Promise chain
D. Use finally handler for errors
Answer: (SHOW ANSWER)
Key JavaScript Promise rule:
try...catch does not catch asynchronous errors that occur in Promise chains.
Therefore, the correct way to handle errors in a Promise is:
promise.then(...).catch(...);
Analysis of options:
A and B:
Both wrap asynchronous code in try...catch, which does not catch Promise rejections.
try...catch only catches errors thrown synchronously inside the block.
C:
Correct. A .catch() on the Promise chain handles any error from the Promise returned by requestPromise().
D:
.finally() executes regardless of success or failure, but it does not receive the error object, so it cannot handle the error.
Thus the only valid solution is option C.
JavaScript Knowledge Reference (text-only)
Promise rejections must be handled with .catch().
try...catch only handles synchronous code.
.finally() does not receive a rejection reason.

NEW QUESTION: 23

A developer removes the HTML class attribute from the checkout button, so now it is simply: <button>Checkout</button>
There is a test to verify the existence of the checkout button, however it looks for a button with class="blue". The test fails because no such button is found.
Which type of test category describes this test?
A. True negative
B. True positive
C. False negative
D. False positive
Answer: (SHOW ANSWER)
Definitions in testing context (treating "positive" as "test reports a bug/failure"):
True positive: System has a bug; test correctly fails.
False positive: System is correct; test fails (reports a bug that isn't actually a bug).
True negative: System has a bug; test correctly passes indicating "no success" in detection context (less common phrasing here).
False negative: System has a bug; test incorrectly passes (missed bug).
Here:
The real requirement, as stated, is to verify the existence of the checkout button.
The button still exists (<button>Checkout</button>), so the system behavior is correct regarding that requirement.
The test, however, is checking for an overly specific condition (class="blue"), which is not actually part of the stated requirement.
The test fails, saying there is a problem (no button with class="blue"), even though from the requirement standpoint, the checkout button is present.
So:
No real bug concerning presence of checkout button.
Test reports a failure → a false positive.
Therefore, the correct category is D, False positive.

NEW QUESTION: 24

```
static delay = async delay => {
return new Promise(resolve => {
setTimeout(resolve, delay);
});
};
static asyncCall = async () => {
await delay(1000);
console.log(1);
};
console.log(2);
asyncCall();
```

console.log(3);
Assume delay and asyncCall are in scope as functions.
What is logged to the console?
A. 1 2 3
B. 1 3 2
C. 2 1 3
D. 2 3 1
Answer: (SHOW ANSWER)
The correct answer is D, because JavaScript executes synchronous code first, while the code after await runs later after the Promise resolves.
The execution order is:
console.log(2);
This runs first, so JavaScript logs:
2
Then this line runs:
asyncCall();
The asyncCall() function starts executing. Inside it, JavaScript reaches:
await delay(1000);
The delay(1000) function returns a Promise that resolves after 1000 milliseconds:
return new Promise(resolve => {
setTimeout(resolve, delay);
});
Because await pauses the rest of the asyncCall() function, this line does not run immediately:
console.log(1);
Instead, JavaScript continues executing the remaining synchronous code outside the async function:
console.log(3);
So JavaScript logs:
3
After approximately 1000 milliseconds, the Promise returned by delay(1000) resolves. Then the paused async function continues and runs:
console.log(1);
So JavaScript logs:
1
Final console output:
2
3
1
Important JavaScript concepts involved:
An async function always returns a Promise.
The await keyword pauses execution only inside the async function where it appears.

Code outside the async function continues running normally.
setTimeout() schedules its callback to run later, after the current synchronous code has finished.
Therefore, the verified answer is D. 2 3 1.

NEW QUESTION: 25
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: (SHOW ANSWER)
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.
Why other options are incorrect:
B . console.trace()
Prints a stack trace, not timing information.
C . console.timeStamp()
Used with browser performance tools; it does not directly show elapsed times like timeLog or timeEnd.
D . console.getTime()
Not a standard console method.
Thus, the function that can be used to obtain timing between calls is:
Answe r: A
Study Guide Concepts:
console.time, console.timeLog, console.timeEnd
Measuring performance and intermediate timings
Standard console debugging API methods

NEW QUESTION: 26
A page loads 50+ elements, all ads.
Developer wants to quickly and temporarily remove them.
Options:
A. Use the browser console to execute a script that prevents the load event from firing.
B. Use the DOM inspector to prevent the load event from firing.
C. Use the browser console to execute a script that removes all elements containing the class ad-library-item.
D. Use the DOM inspector to remove all elements containing the class ad-library-item.
Answer: (SHOW ANSWER)
The goal:
"Temporarily and quickly remove them."
The fastest way is to run a script in the browser console that removes all such elements:
document.querySelectorAll('.ad-library-item').forEach(el => el.remove()); This immediately removes all 50+ elements and requires no page reload.
Option analysis:
A & B: Preventing load events is unreliable and does not remove already-created DOM nodes.
C: Correct-console code can remove all matching DOM elements instantly.
D: Possible but extremely slow; removing 50+ elements one by one in the inspector is not "quick."
Therefore, C is the best answer.
JavaScript Knowledge Reference (text-only)
The browser console can run arbitrary JavaScript that manipulates the DOM.
document.querySelectorAll() returns all matching DOM nodes.
Nodes can be removed using .remove().

NEW QUESTION: 27
A developer has a module that contains multiple functions.
What kind of export should be leveraged so that multiple functions can be used?
A. multi
B. default
C. all
D. named
Answer: (SHOW ANSWER)
The correct answer is D.
When a module contains multiple functions and each function should be available individually, the best choice is a named export.
Example:
function add(a, b) {
return a + b;
}
function subtract(a, b) {
return a - b;
}
export { add, subtract };
Then another file can import exactly the functions it needs:
import { add, subtract } from './mathUtils.js';
console.log(add(5, 2));
console.log(subtract(5, 2));
Named exports are useful when a module provides several reusable functions, constants, or classes.
Why the other options are incorrect:
multi is not a JavaScript export type.
default is usually used when a module has one main value to export. A module can only have one default export.
all is not an export type in JavaScript module syntax.
Therefore, the correct export type for multiple functions is named export, so the verified answer is D.

NEW QUESTION: 28
Given the code:
01 function GameConsole(name) {
02 this.name = name;
03 }
04
05 GameConsole.prototype.load = function(gamename) {
06 console.log('${this.name} is loading a game: ${gamename}....');

```
07 }
08
09 function Console16bit(name) {
10 GameConsole.call(this, name);
11 }
12
13 Console16bit.prototype = Object.create(GameConsole.prototype);
14
15 // insert code here
16 console.log('${this.name} is loading a cartridge game: ${gamename}....');
17 }
18
19 const console16bit = new Console16bit('SNEGeneziz');
20 console16bit.load('Super Monic 3x Force');
```

What should a developer insert at line 15?

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

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

C. Console16bit.prototype.load(gamename) {

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

Answer: (SHOW ANSWER)

A subclass created by:

Console16bit.prototype = Object.create(GameConsole.prototype);

inherits all prototype methods from GameConsole, including load.

To override the inherited method, the correct syntax is to assign a new function to the method name on the prototype:

```
Console16bit.prototype.load = function(gamename) {
console.log(`${this.name} is loading a cartridge game: ${gamename}....`);
};
```

Why the other options are wrong:

A assigns something to the constructor function itself, not to the prototype method. Invalid.

B attempts to call a function in the left-hand side of an assignment; invalid syntax.

C also uses invalid syntax-prototype methods cannot be defined this way.

D is the correct definition of a prototype method override.

JavaScript Knowledge Reference (text-only)

Methods are overridden by assigning functions to Subclass.prototype.methodName.

Object.create() establishes prototype inheritance.

Constructor functions require prototype method attachment using assignment syntax.

NEW QUESTION: 29

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: (SHOW ANSWER)

Comprehensive and Detailed

The loop removes every 4:

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

i=0 → 1 (no change)

i=1 → 2 (no change)

i=2 → 3 (no change)

i=3 → 4 → splice removes index 3 → [1,2,3,4,5,4,4], then i-- → 2

Next loop, i=3 → 4 again → splice → [1,2,3,5,4,4], i-- → 2

i=3 → 5 (no change)

i=4 → 4 → splice → [1,2,3,5,4], i-- → 3

i=4 → 4 → splice → [1,2,3,5], i-- → 3

Next i=4, array.length is 4 → loop ends.

All 4s removed, final array: [1, 2, 3, 5].

NEW QUESTION: 30

Which three browser specific APIs are available for developers to persist data between page loads?

A. localStorage

B. indexedDB

C. cookies

D. global variables

E. IIFEs

Answer: (SHOW ANSWER)

We want mechanisms that persist data between page loads (i.e., after refresh or navigation), in the browser.

A . localStorage

Part of the Web Storage API.

Data persists across page reloads and browser restarts (until explicitly cleared).

Scoped per origin (protocol + host + port).

This is a correct persistent storage API.

B . indexedDB
A low-level, client-side NoSQL database in the browser.
Stores large amounts of structured data and persists across reloads and sessions.
This is also a correct persistent API.
C . cookies
Small key/value pairs stored by the browser and often sent with HTTP requests.
Can have expiration dates and persist across page loads and sessions.
They are a traditional persistence mechanism in browsers.
So cookies also qualify.
D . global variables
Global JS variables exist only for the life of the current page context.
When the page is refreshed or navigated away, they are lost.
They do not persist between page loads.
E . IIFEs (Immediately Invoked Function Expressions)
A pattern for creating a new scope, not a persistence mechanism.
Variables inside an IIFE are still lost when the page reloads.
Thus, the three persistent browser APIs are:
localStorage
indexedDB
cookies

NEW QUESTION: 31
Given the JavaScript below:
function onLoad() {
console.log("Page has loaded!");
}
Where can the developer see the log statement after loading the page in the browser?
A. On the browser JavaScript console
B. On the terminal console running the web server
C. In the browser performance tools log
D. On the webpage console log
Answer: (SHOW ANSWER)
console.log() in browser-side JavaScript writes output to the browser's JavaScript console, which is available in the browser's Developer Tools.
In a typical browser (Chrome, Firefox, Edge, etc.), you open DevTools and go to the Console tab.
Any console.log("Page has loaded!"); executed in page JavaScript will appear there.
Why A is correct:
The function onLoad is a client-side function (runs in the browser).
When it executes, the console.log call goes to the browser's JavaScript console, not the server.
Why the others are incorrect:
B . On the terminal console running the web server

That console shows logs from the server-side runtime (e.g., Node.js logs).
This code is clearly browser-side JavaScript (no Node.js or server context shown).
Therefore, the output does not appear in the terminal.
C . In the browser performance tools log
Performance tools (Timeline, Performance tab, etc.) show metrics about rendering, CPU time, network, etc., not generic console.log messages.
console.log messages appear in the Console tab, not in performance logs.
D . On the webpage console log
There is no built-in concept of a "webpage console log" UI rendered on the page itself by default.
Unless you explicitly code something to display logs in the DOM, console.log output is not visible on the page, only in Developer Tools.
Therefore, the correct place to see that log is:
Answe r: A
JavaScript knowledge / Study Guide references (concept names only, no links):
Browser Developer Tools - Console tab
console.log() in client-side JavaScript
Difference between client-side logs and server-side logs

Valid JS-Dev-101 Dumps shared by EduDump.com for Helping Passing JS-Dev-101 Exam! EduDump.com now offer the **newest JS-Dev-101 exam dumps**, the EduDump.com JS-Dev-101 exam **questions have been updated** and **answers have been corrected** get the **newest** EduDump.com JS-Dev-101 dumps with Test Engine here: https://www.edudump.com/exams/Salesforce/JS-Dev-101/premium/ (**149** Q&As Dumps, **35%OFF Special Discount Code: freecram**)

NEW QUESTION: 32
Refer to the code below:
01 new Promise((resolve, reject) => {
02 const fraction = Math.random();
03 if (fraction > 0.5) reject('fraction > 0.5, ' + fraction);
04 resolve(fraction);
05 })
06 .then(() => console.log('resolved'))
07 .catch((error) => console.error(error))
08 .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: (SHOW ANSWER)
Behavior of Promise.prototype.finally:
.finally(handler) registers a callback that runs when the promise is settled, meaning:
Either fulfilled (resolved), or
rejected.
Important points:
The finally callback does not receive the promise's value or error (unlike then and catch).
It is executed after the promise is settled, but before the resolution value or rejection reason is passed further down the chain.
It runs in both success and failure paths.
In the given code:
The promise may either:
Call reject('fraction > 0.5, ' + fraction) if fraction > 0.5, or
Call resolve(fraction) otherwise.
In both cases:
If it resolves, .then(() => console.log('resolved')) runs, and then .finally(...) is executed.
If it rejects, .catch((error) => console.error(error)) runs, and then .finally(...) is executed.
So .finally runs:
Not just "when rejected".
Not just "when resolved".
But whenever the promise is resolved or rejected.
Therefore, the correct choice is:
D . When resolved or rejected.

NEW QUESTION: 33
Why does second have access to variable a?
A. Inner function's scope
B. Outer function's scope
C. Prototype Chain
D. Hoisting
Answer: (SHOW ANSWER)
The answer is B because JavaScript uses lexical scope.
When a function is created inside another function, the inner function can access variables from the outer function where it was defined.
For example:
function first() {
let a = 10;
function second() {
console.log(a);
}

second();
}
Here, a belongs to the outer function first(). The inner function second() can access it because JavaScript looks for variables through the scope chain.
The lookup works like this:
1. Look inside second()
2. If not found, look inside first()
3. If not found, look in the global scope
Since a is found in the outer function's scope, second() can use it.
This is also the basis of a closure. A closure allows an inner function to remember and access variables from its outer lexical environment.
Why the other options are incorrect:
A is not the best answer because a is not declared inside second() itself.
C is incorrect because the prototype chain is used for object property lookup, not local variable scope lookup.
D is incorrect because hoisting explains how declarations are moved during compilation, but it does not explain why an inner function can access an outer function's variable.
Therefore, the verified answer is B.

NEW QUESTION: 34
Refer to the code below:
let productSKU = '8675309';
A developer has a requirement to generate SKU numbers that are always 19 characters long, starting with 'sku', and padded with zeros.
Which statement assigns the value sku0000000008675309?
A. productSKU = productSKU.padEnd(16, '0').padStart('sku');
B. productSKU = productSKU.padStart(16, '0').padStart(19, 'sku');
C. productSKU = productSKU.padEnd(16, '0').padStart(19, 'sku');
D. productSKU = productSKU.padStart(19, '0').padStart('sku');
Answer: (SHOW ANSWER)
We start with:
let productSKU = '8675309';
The requirement:
Final SKU string:
Length: 19 characters.
Starts with 'sku'.
Remaining characters are digits padded with zeros on the left (to reach total length).
We can use String.prototype.padStart and String.prototype.padEnd:
str.padStart(targetLength, padString)
If str.length < targetLength, it adds padString to the start until the length is targetLength.
str.padEnd(targetLength, padString)

Similar, but adds padString to the end.
We want a pattern like:
First, pad the numeric part out to a fixed length with zeros.
Then, pad to total length with 'sku' at the start.
Analyze Option B
productSKU = productSKU.padStart(16, '0').padStart(19, 'sku');
Step 1: productSKU.padStart(16, '0')
Initial productSKU is '8675309' (length 7).
After padStart(16, '0'), we pad zeros on the left to reach length 16.
We need 16 − 7 = 9 zeros:
Result after step 1:
productSKU === '0000000008675309' // length 16
Step 2: .padStart(19, 'sku')
Now productSKU has length 16.
We call padStart(19, 'sku'):
We need 19 − 16 = 3 extra characters.
The pad string 'sku' is exactly 3 characters, so it is added as-is at the start.
Result after step 2:
productSKU === 'sku0000000008675309' // length 19
This satisfies:
Length 19.
Starts with 'sku'.
Remaining characters are zeros plus the original digits, i.e. a zero-padded numeric section.
While the literal sample sku000000008675309 in the text has a slightly different count of zeros,
Option B follows the requirement pattern:
3 characters of 'sku'
Numeric part padded with zeros to make 16 characters total for the numeric part
3 + 16 = 19 total characters
Option B matches the intended logic using padStart and padEnd.
Why the other options are incorrect
Option A:
productSKU = productSKU.padEnd(16, '0').padStart('sku');
padEnd(16, '0') produces '8675309000000000' (original number followed by zeros).
padStart('sku') is invalid usage:
padStart takes a numeric target length as the first argument, not a string.
Passing 'sku' as the first argument leads to type coercion that does not achieve the intended behavior.
This will not reliably produce the desired SKU.
Option C:
productSKU = productSKU.padEnd(16, '0').padStart(19, 'sku');
First padEnd(16, '0') from '8675309' gives '8675309000000000' (length 16, zeros at the end).

Then padStart(19, 'sku') adds 3 chars 'sku' at the front:
Result: 'sku8675309000000000'.
This string starts with 'sku', but the zeros are at the end of the digits, not padding the numeric part on the left as desired.
Option D:
productSKU = productSKU.padStart(19, '0').padStart('sku');
First padStart(19, '0') pads zeros at the left to make the length 19.
Then padStart('sku') again incorrectly uses a string where a numeric targetLength is required.
This will not produce the correct SKU format.
Therefore, the only option that correctly uses padStart to create a 16-character zero-padded numeric portion and then a 19-character string starting with 'sku' is:
Answe r: B
Reference / Study Guide concepts (no links):
String.prototype.padStart(targetLength, padString)
String.prototype.padEnd(targetLength, padString)
String length calculations
Left-padding numeric strings with zeros
Building prefixed identifiers with fixed total length

NEW QUESTION: 35
Refer to the following array:
let arr = [1, 2, 3, 4, 5];
Which two lines of code result in a second array, arr2, created such that arr2 is a reference to arr?
A. let arr2 = arr.slice(0, 5);
B. let arr2 = Array.from(arr);
C. let arr2 = arr;
D. let arr2 = arr.sort();
Answer: (SHOW ANSWER)
The correct answers are C and D.
Arrays in JavaScript are objects. When an array variable is assigned directly to another variable, both variables point to the same array in memory.
Option C is correct:
let arr2 = arr;
This does not create a new array. It creates another reference to the same array.
Example:
arr2.push(6);
console.log(arr);
Output:
[1, 2, 3, 4, 5, 6]
Changing arr2 also affects arr because both variables reference the same array.

Option D is also correct:
let arr2 = arr.sort();
The sort() method sorts the array in place and returns the same array reference. Therefore, arr2 refers to the same array object as arr.
The incorrect options create copies:
let arr2 = arr.slice(0, 5);
creates a shallow copy.
let arr2 = Array.from(arr);
also creates a new shallow copy.
So the two lines that make arr2 reference the original arr are C and D.

NEW QUESTION: 36
What are two unique features of fat-arrow functions compared to normal function definitions?
A. If the function has a single expression in the function body, the expression will be evaluated and implicitly returned.
B. The function uses the this from the enclosing scope.
C. The function receives an argument called parentThis, giving the enclosing lexical scope.
D. The function generates its own this making it useful for separating scope.
Answer: (SHOW ANSWER)
Arrow functions have two defining characteristics:
Implicit return when written as a single expression:
const fn = () => 5; // returns 5 automatically
This is unique to arrow functions → A is correct.
Lexical this binding:
Arrow functions do not create their own this.
Instead, they use the this value from the surrounding scope → B is correct.
Incorrect options:
C is false: There is no special argument called parentThis.
D is the opposite of arrow function behavior:
Normal functions create their own this; arrow functions do not.
JavaScript Knowledge Reference (text-only)
Arrow functions have lexical this binding.
Arrow functions support implicit return for single expressions.

NEW QUESTION: 37
Given a value, which two options can a developer use to detect if the value is NaN?
A. value === Number.NaN
B. value == NaN
C. isNaN(value)
D. Object.is(value, NaN)
Answer: (SHOW ANSWER)

We already know NaN is special: it is not equal to itself.
Check each:
A . value === Number.NaN
Number.NaN is NaN.
NaN === NaN is always false.
This will never be true; cannot reliably detect NaN.
B . value == NaN
NaN == NaN is also always false.
Again, this never detects NaN.
C . isNaN(value)
Global isNaN converts its argument to a number and then checks if the result is NaN.
This can detect NaN, but it may also return true for non-number values that coerce to NaN, such as isNaN('foo').
Regardless, it is a standard way to detect if a value is "NaN-like" in JavaScript.
D . Object.is(value, NaN)
Object.is(NaN, NaN) returns true.
This is a strict way to detect a value that is exactly NaN (no coercion).
Therefore, among the given choices, the two viable ways to detect NaN are:
isNaN(value)
Object.is(value, NaN)

NEW QUESTION: 38
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: (SHOW ANSWER)
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:
console.log(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:
Callback:
() => {
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.

NEW QUESTION: 39
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);
Answer: (SHOW ANSWER)
Comprehensive and Detailed
"Salesforce" indices: S(0) a(1) l(2) e(3) s(4) f(5) ...
A / C (substring(0, 5)):
substring(start, end) returns characters from start up to but not including end.
str.substring(0, 5) → characters at indices 0,1,2,3,4 → "Sales".
D (substr(0, 5)):
substr(start, length) returns length characters starting at start.
str.substr(0, 5) → 5 chars from index 0 → "Sales".
B is invalid because s is not defined; it should be 0. So the two correct statements are A and D.

NEW QUESTION: 40
Refer to the code below:

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

Freecram.net

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: (SHOW ANSWER)
:
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 function.
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.

NEW QUESTION: 41
Given the JavaScript below:
01 function filterDOM(searchString){
02 const parsedSearchString = searchString && searchString.toLowerCase();
03 document.querySelectorAll('.account').forEach(account => {
04 const accountName = account.innerHTML.toLowerCase();
05 account.style.display = accountName.includes(parsedSearchString) ? /* Insert code here */;
06 });
07 }
Which code should replace the placeholder comment on line 05 to hide accounts that do not match the search string?
A. 'block' : 'none'
B. 'hidden' : 'visible'
C. 'visible' : 'hidden'
D. 'none' : 'block'
Answer: (SHOW ANSWER)
We have a ternary:
account.style.display = accountName.includes(parsedSearchString)

? /* if true */
/* if false */;
Requirement:
If the account matches the search string → show it.
If it does not match → hide it.
For display:
Show: 'block' (or similar visible value).
Hide: 'none'.
So we want:
account.style.display = accountName.includes(parsedSearchString)
? 'block'
'none';
That corresponds to option A.
Why others are wrong:
B, C use values suitable for visibility, not display.
D ('none' : 'block') inverts the logic, hiding matches and showing non-matches.

NEW QUESTION: 42
Which actions can be done using the JavaScript browser console?
A. Run code that's not related to the page
B. View and change the DOM of the page
C. Display a report showing the performance of a page
D. Change the DOM and the JavaScript code of the page
E. View and change security cookies
Answer: (SHOW ANSWER)
The correct answer is B because the JavaScript browser console allows developers to interact with the current web page using JavaScript commands. Through the console, you can access the document object, inspect DOM elements, read their properties, and modify the structure or content of the page.
For example:
document.body.style.backgroundColor = "lightblue";
This changes the page's background color.
Another example:
document.querySelector("h1").textContent = "Updated Heading";
This changes the text of the first <h1> element on the page.
The DOM, or Document Object Model, is the browser's JavaScript representation of the HTML page. Since the console runs JavaScript in the context of the current page, it can be used to inspect and modify DOM elements directly.
Why the other options are incorrect:

A is not the best answer because although you can type general JavaScript expressions in the console, the browser console is mainly connected to the current page's execution environment. The question is asking about practical browser-console actions related to a page.
C is incorrect because performance reports are usually generated through tools such as the Performance panel or Lighthouse in browser developer tools, not directly through the JavaScript console.
D is incorrect because the console can modify the DOM and runtime values, but it does not permanently change the original JavaScript source code of the page. Any changes made through the console are temporary and disappear when the page reloads.
E is incorrect because security-protected cookies, such as HttpOnly cookies, cannot be viewed or changed using JavaScript. This is an intentional browser security feature that prevents client-side scripts from accessing sensitive cookie data.
Therefore, the best verified answer is B.

NEW QUESTION: 43
Refer to the code below:
01 function myFunction(reassign) {
02 let x = 1;
03 var y = 1;
04
05 if (reassign) {
06 let x = 2;
07 var y = 2;
08 console.log(x);
09 console.log(y);
10 }
11
12 console.log(x);
13 console.log(y);
14 }
What is displayed when myFunction(true) is called?
A. 2 2 2 2
B. 2 2 1 2
C. 2 2 1 1
D. 2 2 undefined undefined
Answer: (SHOW ANSWER)
This question tests understanding of let (block scope) and var (function scope) in JavaScript.
Initial declarations in the function:
let x = 1; // line 2
var y = 1; // line 3
Here:

x is declared with let, so it is block-scoped to the function body.
y is declared with var, so it is function-scoped to the entire function.
Inside the if (reassign) block (and since reassign is true, we enter it):
if (reassign) {
let x = 2; // line 6
var y = 2; // line 7
console.log(x); // line 8
console.log(y); // line 9
}
Detailed behavior:
let x = 2; on line 6 creates a new block-scoped variable x that exists only inside the if block. It does not change the outer x declared on line 2.
var y = 2; on line 7 declares y with var again, but var is function-scoped. This effectively reassigns the same y defined on line 3 for the entire function. After this line, y is 2 everywhere in the function.
Now, inside the if block:
console.log(x); (line 8) logs the inner block-scoped x, which is 2.
console.log(y); (line 9) logs y, which is the function-scoped y that was set to 2.
So the first two outputs are:
2
2
After the if block, execution continues:
console.log(x); // line 12
console.log(y); // line 13
Outside the if block:
The block-scoped let x = 2; no longer exists; it was only visible inside the if block.
The outer let x = 1; (line 2) is still in scope and has not been changed.
Thus:
console.log(x); (line 12) logs the outer x, which is still 1.
console.log(y); (line 13) logs y which, due to var y = 2; inside the if, is now 2 for the whole function.
Therefore, when myFunction(true) is called, the output in order is:
2 (inner x in if)
2 (function-scoped y after reassignment)
1 (outer x after if)
2 (function-scoped y remains 2)
This corresponds to:
Answer : B (2 2 1 2)
JavaScript knowledge / study guide reference concepts:
let declarations and block scope
var declarations and function scope

Shadowing of variables with let inside a block
Re-declaration and reassignment of var within a function
Execution order of statements and console output

NEW QUESTION: 44
A developer has an isDeg function that takes one argument, pts. They want to schedule the function to run every minute.
What is the correct syntax for scheduling this function?
A. setInterval(isDeg('cat'), 60000);
B. setInterval(isDeg, 60000, 'cat');
C. setTimeout(isDeg, 60000, 'cat');
D. setTimeout(isDeg('cat'), 60000);
Answer: (SHOW ANSWER)
setInterval(fn, delay, arg1, arg2, ...) calls fn every delay ms, passing extra arguments.
To call isDeg every 60 seconds with "cat":
setInterval(isDeg, 60000, 'cat');
Why others are wrong:
A/D: isDeg('cat') is called immediately, and its return value is scheduled, not the function.
C: setTimeout runs once after 60 seconds, not repeatedly.

NEW QUESTION: 45
A developer uses a parsed JSON string to work with user information as in the block below:
01 const userInformation = {
02 "id": "user-01",
03 "email": "user01@universalcontainers.demo",
04 "age": 25
05 };
Which two options access the email attribute in the object?
A. userInformation.email
B. userInformation.get("email")
C. userInformation["email"]
D. userInformation[email]
Answer: (SHOW ANSWER)
userInformation is a plain JavaScript object:
{
id: "user-01",
email: "user01@universalcontainers.demo",
age: 25
}
You can access object properties in two standard ways:
Dot notation: object.propertyName

Bracket notation: object['propertyName']
Apply to each option:
A . userInformation.email
Correct dot notation; accesses "user01@universalcontainers.demo".
B . userInformation.get("email")
.get() is a method on Map instances, not on plain objects.
A regular object does not have a .get method; this would be TypeError.
C . userInformation["email"]
Correct bracket notation; string "email" is used as the property key.
Returns "user01@universalcontainers.demo".
D . userInformation[email]
Here email is treated as a variable, not a string literal.
Unless there is a variable named email defined with a value matching a property key, this will either:
Throw a ReferenceError (if email is not defined), or
Use the value of email as a computed key (not what is intended here).
It is not the correct way to access the "email" property literal.
Therefore, the correct ways are:
Answe r: A, C
Relevant concepts: object property access (. vs []), plain objects vs Map, literal property names vs computed property names.

NEW QUESTION: 46
Corrected code:
let a = "*";
let b = "**";
// x = 3;
console.log(a);
What is displayed when the code executes?
A. ReferenceError: a is not defined
B. *
C. undefined
D. null
Answer: (SHOW ANSWER)
The correct answer is B because variable a is declared and initialized before it is printed.
let a = "*";
This creates a block-scoped variable named a and assigns it the string value:
"*"
The line:
// x = 3;
is a comment. JavaScript ignores comments during execution, so this line has no effect.

Then this line runs:
console.log(a);
Since a already exists and contains "*", the console displays:
* Option A is incorrect because a is defined.
Option C is incorrect because a has an assigned value, so it is not undefined.
Option D is incorrect because a was never assigned null.
Therefore, the verified answer is B.

Valid JS-Dev-101 Dumps shared by EduDump.com for Helping Passing JS-Dev-101 Exam! EduDump.com now offer the **newest JS-Dev-101 exam dumps**, the EduDump.com JS-Dev-101 exam **questions have been updated** and **answers have been corrected** get the **newest** EduDump.com JS-Dev-101 dumps with Test Engine here: https://www.edudump.com/exams/Salesforce/JS-Dev-101/premium/ (**149** Q&As Dumps, **35%OFF Special Discount Code: freecram**)

NEW QUESTION: 47
Refer to the code:
01 function execute() {
02 return new Promise((resolve, reject) => reject());
03 }
04 let promise = execute();
05
06 promise
07 .then(() => console.log('Resolved1'))
08 .then(() => console.log('Resolved2'))
09 .then(() => console.log('Resolved3'))
10 .catch(() => console.log('Rejected'))
11 .then(() => console.log('Resolved4'));
What is the result when the Promise in the execute function is rejected?
A. Resolved1 Resolved2 Resolved3 Rejected Resolved4
B. Rejected
C. Resolved1 Resolved2 Resolved3 Resolved4
D. Rejected Resolved4
Answer: (SHOW ANSWER)
execute() returns a Promise that immediately calls reject().
So promise starts in a rejected state.
When a Promise is rejected and you chain .then() calls without rejection handlers, all those .then() callbacks are skipped until a .catch() is encountered:
promise

.then(...) // skipped
.then(...) // skipped
.then(...) // skipped
.catch(...) // executed
.then(...); // executed after catch
Execution:
.then(() => console.log('Resolved1')) is skipped.
.then(() => console.log('Resolved2')) is skipped.
.then(() => console.log('Resolved3')) is skipped.
.catch(() => console.log('Rejected')) runs and logs Rejected.
The .catch() returns a resolved Promise (no explicit return, so undefined), so the next .then() runs:
.then(() => console.log('Resolved4')) logs Resolved4.
Final output:
Rejected
Resolved4
This matches option D.

NEW QUESTION: 48
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: (SHOW ANSWER)
The verified corrected answers are C, D, and E.
This function is a normal function, not a curried function:
function sum(num1, num2, num3) {
if (num3 === undefined) {
num3 = 0;
}
return num1 + num2 + num3;
}
It expects values to be passed in the same function call:

sum(num1, num2, num3);
Now check the valid corrected options.
Option C:
sum(5, 5, 0);
Calculation:
5 + 5 + 0
Result:
10
So C is correct.
Option D:
sum(10, 0);
Here, num3 is not provided, so it is undefined.
The function checks:
if (num3 === undefined) {
num3 = 0;
}
So the calculation becomes:
10 + 0 + 0
Result:
10
So D is correct.
Option E:
sum(5, 2, 3);
Calculation:
5 + 2 + 3
Result:
10
So E is correct.
Why A and B are incorrect as originally written:
sum(5)(5);
This calls sum(5) first. Since num2 is missing, the result becomes NaN. Then JavaScript tries to call that returned value as a function, which causes a TypeError.
sum()(10);
This also calls sum() first, producing NaN, and then attempts to call NaN as a function.
Those styles would only work if sum were written as a curried function, but the given implementation is not curried.
Therefore, the verified corrected answers are C, D, and E.

NEW QUESTION: 49
Refer to the code below (assuming Promise.race is intended):
let cat3 = new Promise(resolve =>

```
setTimeout(resolve, 3000, "Cat 3 completes")
);
Promise.race([cat1, cat2, cat3])
.then(value => {
let result = `${value} the race.`;
})
.catch(err => {
console.log("Race is cancelled:", err);
});
```

(Assuming cat1 and cat2 are similar to earlier examples: cat2 resolves fastest.) What is the value of result when Promise.race executes?

A. Car 2 completed the race.

B. Car 1 crashed on the race.

C. Race is cancelled.

D. Car 3 completed the race.

Answer: (SHOW ANSWER)

From earlier pattern (cars/cats race):

One promise resolves the fastest with "Car 2 completed" (or analogous "Cat 2 completes").

Promise.race resolves with the first settled promise's value.

In .then, value is "Car 2 completed" and:

let result = `${value} the race.`;

// "Car 2 completed the race."

Thus result is "Car 2 completed the race." → option A.

NEW QUESTION: 50

A developer wrote the following code:

```
01 let x = object.value;
02
03 try {
04 handleObjectValue(x);
05 } catch(error) {
06 handleError(error);
07 }
```

Freecram.net

The developer has a getNextValue function to execute after handleObjectValue(), but does not want to execute getNextValue() if an error occurs. How can the developer change the code to ensure this behavior?

A.

```
03 try {
04 handleObjectValue(x);
05 } catch(error) {
06 handleError(error);
07 } then {
08 getNextValue();
09 }
```

B. 03 try {
04 handleObjectValue(x);
05 getNextValue();
06 } catch(error) {
07 handleError(error);
08 }
C. 03 try {
04 handleObjectValue(x);
05 } catch(error) {
06 handleError(error);
07 }
08 getNextValue();
D. 03 try {
04 handleObjectValue(x);
05 } catch(error) {
06 handleError(error);
07 } finally {
08 getNextValue();
09 }
Answer: (SHOW ANSWER)
Requirement:
getNextValue() should run only if handleObjectValue(x) does not throw an error.
If an error occurs, handleError(error) should run, and getNextValue() should be skipped.
Option B:
try {
handleObjectValue(x);
getNextValue();
} catch (error) {
handleError(error);
}
Behavior:
If handleObjectValue(x) succeeds (no error):
Execution continues to getNextValue();.
If handleObjectValue(x) throws:
Control jumps directly to catch, getNextValue() is skipped.
handleError(error); is called.
This matches the requirement perfectly.
Why others are incorrect:
A: } then { is invalid JavaScript syntax.
C:
try {

```
handleObjectValue(x);
} catch (error) {
handleError(error);
}
getNextValue();
```
getNextValue() is called after the try...catch regardless of whether an error occurred. Not acceptable.
D: finally always runs:
```
try {
handleObjectValue(x);
} catch (error) {
handleError(error);
} finally {
getNextValue();
}
```
getNextValue() will execute in both success and error cases.
Therefore, the correct approach is B.
Concepts: try/catch control flow, finally semantics, placing code inside vs outside try/catch blocks.

NEW QUESTION: 51

```
01 function Animal(size, type) {
02 this.type = type || 'Animal';
03 this.canTalk = false;
04 }
05
06 Animal.prototype.speak = function() {
07 if (this.canTalk) {
08 console.log("It spoke!");
09 }
10 };
11
12 let Pet = function(size, type, name, owner) {
13 Animal.call(this, size, type);
14 this.size = size;
15 this.name = name;
16 this.owner = owner;
17 }
18
19 Pet.prototype = Object.create(Animal.prototype);
20 let pet1 = new Pet();
```
Given the code above, which three properties are set for pet1?

A. speak
B. owner
C. canTalk
D. name
E. type
Answer: (SHOW ANSWER)
When pet1 = new Pet(); is created:
Inside Pet constructor:
Animal.call(this, size, type);
this.size = size;
this.name = name;
this.owner = owner;
Animal.call(this, size, type):
Sets this.type = type || 'Animal' → 'Animal' (because type is undefined).
Sets this.canTalk = false.
Then this.size, this.name, this.owner are set (to undefined since no args passed), but they do exist as properties.
So as own properties, pet1 has: type, canTalk, size, name, owner.
speak is defined on Animal.prototype, so pet1.speak exists by inheritance, but is not an own data property created in the constructor.
From the listed options, three important properties directly set by constructor logic and clearly used in behavior are:
canTalk
name
type
Thus, C, D, E.

NEW QUESTION: 52
A developer wants to use a try...catch statement to catch any error that countSheep() may throw and pass it to a handleError() function.
What is the correct implementation of the try...catch?
A. setTimeout(function() {
try {
countSheep();
} catch (e) {
handleError(e);
}
}, 1000);
B. try {
countSheep();
} finally {

handleError(e);
}
C. try {
countSheep();
} handleError (e){
catch(e);
}
D. try {
setTimeout(function() {
countSheep();
}, 1000);
} catch (e) {
handleError(e);
}
Answer: (SHOW ANSWER)
Comprehensive and Detailed
Errors thrown inside setTimeout callbacks occur asynchronously, in a different tick of the event loop. A try...catch around setTimeout itself (as in D) cannot catch errors thrown later in the scheduled callback.
To catch errors from countSheep() when it's called asynchronously, the try...catch must wrap the call inside the timeout callback:
setTimeout(function() {
try {
countSheep();
} catch (e) {
handleError(e);
}
}, 1000);
B uses finally incorrectly and references e out of scope.
C is not valid JavaScript syntax.
D's catch can only handle synchronous errors thrown before setTimeout returns, not those thrown inside the callback.

NEW QUESTION: 53
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: (SHOW ANSWER)

In Node.js:

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

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

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

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

The requirement is:

"the folder location that the code executes from."

That is exactly what __dirname provides.

So the correct choice is __dirname.

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

NEW QUESTION: 54

Refer to the code below (corrected to use a template literal on line 08):

```
01 let car1 = new Promise((_, reject) =>
02 setTimeout(reject, 2000, "Car 1 crashed in")
03 );
04 let car2 = new Promise(resolve =>
05 setTimeout(resolve, 1500, "Car 2 completed")
06 );
07 let car3 = new Promise(resolve =>
08 setTimeout(resolve, 3000, "Car 3 completed")
09 );
10
11 Promise.race([car1, car2, car3])
12 .then(value => {
13 let result = `${value} the race.`;
14 })
15 .catch(err => {
16 console.log("Race is cancelled.", err);
17 });
```

What is the value of result when Promise.race executes?

A. Car 3 completed the race.

B. Car 2 completed the race.

C. Race is cancelled.

D. Car 1 crashed in the race.

Answer: (SHOW ANSWER)

Understand the three promises:

car1:

let car1 = new Promise((_, reject) =>

setTimeout(reject, 2000, "Car 1 crashed in")
);
Rejects after 2000 ms (2 seconds) with message "Car 1 crashed in".
car2:
let car2 = new Promise(resolve =>
setTimeout(resolve, 1500, "Car 2 completed")
);
Resolves after 1500 ms (1.5 seconds) with message "Car 2 completed".
car3:
let car3 = new Promise(resolve =>
setTimeout(resolve, 3000, "Car 3 completed")
);
Resolves after 3000 ms (3 seconds) with message "Car 3 completed".
Promise.race:
Promise.race([car1, car2, car3])
.then(value => {
let result = `${value} the race.`;
})
.catch(err => {
console.log("Race is cancelled.", err);
});
Behavior of Promise.race:
It settles (resolves or rejects) as soon as any of the given promises settles.
It uses the value or reason from the first settled promise.
Timing:
car2 resolves in 1500 ms.
car1 rejects in 2000 ms.
car3 resolves in 3000 ms.
The first to settle is car2 at 1500 ms, with value "Car 2 completed".
Therefore:
Promise.race resolves (not rejects) with value = "Car 2 completed".
The .then handler runs; .catch is ignored because there is no rejection.
Inside .then:
let result = `${value} the race.`;
Substitute value:
let result = "Car 2 completed the race.";
So, result becomes:
Car 2 completed the race.
Compare to options:
A . Car 3 completed the race.
This would be correct if car3 were the first to resolve, which it is not (it resolves last).

B . Car 2 completed the race.
Exactly matches the first-resolving promise and the constructed message.
C . Race is cancelled.
This is the prefix of the string logged in the .catch handler, but .catch never runs because the race resolves, it does not reject first.
D . Car 1 crashed in the race.
car1 is the first rejection, but since a resolution from car2 happens earlier, the race is already settled successfully before car1 rejects.
Thus the correct value of result as set in the .then block is:
Answe r: B
Study Guide / Concept Reference (no links):
Promise.race(iterable) semantics (first settled promise wins)
setTimeout and timing interactions with Promises
Resolve vs reject paths and .then / .catch
Template literals and string interpolation for building result messages

NEW QUESTION: 55
A developer writes the code below to return a message to a user attempting to register a new username. If the username is available, a variable named msg is declared and assigned a value on line 03.

```
function getAvailabilityMessage(item) {
if (getAvailability(item)) {
var msg = "Username available";
return msg;
}
}
```

A. "msg is not defined"
B. "newUserName"
C. "Username available"
D. undefined
Answer: (SHOW ANSWER)
The correct answer is C.
When getAvailability(item) returns true, the code inside the if block executes:

```
var msg = "Username available";
return msg;
```

The variable msg receives the string value:
"Username available"
Then that same value is returned from the function.
The key point is that var is function-scoped, not block-scoped. So msg belongs to the function scope of getAvailabilityMessage(). However, because the return msg; statement is inside the

same if block, the function immediately returns the assigned string when the username is available.
Option A is incorrect because msg is defined before it is returned.
Option B is incorrect because "newUserName" is not assigned or returned anywhere in the function.
Option D would only happen if getAvailability(item) returned false, because then the function would finish without an explicit return value.
For the available username scenario, the verified answer is C.

NEW QUESTION: 56
Correct implementation of try...catch for countsDeep():
A. try {
countsDeep();
} handleError (e){
catch(e);
}
B. setTimeout(function() {
try {
countsDeep();
} catch (e) {
handleError(e);
}
}, 1000);
C. try {
setTimeout(function() {
countsDeep();
}, 1000);
} catch (e) {
handleError(e);
}
D. try {
setTimeout(function() {
countsDeep();
}, 1000);
} catch (e) {
handleError(e);
}
Answer: (SHOW ANSWER)
The correct answer is B because countsDeep() is executed inside the callback function passed to setTimeout(), and the try...catch block is also placed inside that same callback.

In JavaScript, setTimeout() schedules a function to run later. The outer code finishes first, and the callback runs asynchronously after the delay. Because of this, a try...catch block placed outside setTimeout() cannot catch errors thrown later inside the callback.
Correct logic:

```
setTimeout(function() {
try {
countsDeep();
} catch (e) {
handleError(e);
}
}, 1000);
```

Here, when countsDeep() runs after 1000 milliseconds, any error thrown by countsDeep() happens inside the try block. Therefore, the catch (e) block can catch that error and pass it to handleError(e).
Why the other options are incorrect:
A is incorrect because the syntax is invalid JavaScript. A valid try...catch structure must be:

```
try {
// code
} catch (e) {
// handle error
}
```

Option A incorrectly writes:

```
} handleError (e){
catch(e);
}
```

That is not valid try...catch syntax.
C is incorrect because the try...catch surrounds only the call to setTimeout(), not the later execution of countsDeep(). If countsDeep() throws an error after the timer expires, the outer catch block will not catch it.
D is incorrect for the same reason as C. In the original question, D also had a typing error: it used countSheep() instead of countsDeep(). Even after correcting that typing error, D is still incorrect because the try...catch is outside the asynchronous callback.

NEW QUESTION: 57
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: (SHOW ANSWER)
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.
Concepts: function hoisting, difference between function declarations and function expressions, execution order.

NEW QUESTION: 58
Refer to the code:
01 const exec = (item, delay) =>
02 new Promise(resolve => setTimeout(() => resolve(item), delay));
03
04 async function runParallel() {
05 const [result1, result2, result3] = await Promise.all(
06 [exec('x', '100'), exec('y', '500'), exec('z', '100')]
07);
08 return `parallel is done: ${result1}${result2}${result3}`;

09 }
Which two statements correctly execute runParallel()?
A. async runParallel().then(data);
B. runParallel().then(function(data){
return data;
});
C. runParallel().done(function(data){
return data;
});
D. runParallel().then(data);
Answer: (SHOW ANSWER)
Facts about JavaScript Promises:
runParallel() is declared async, which means it always returns a Promise.
Promises are consumed using .then(), .catch(), and .finally().
There is no .done() method in native JavaScript Promises.
The async keyword cannot be placed before a function call (option A is invalid syntax).
Analysis of each option:
A . async runParallel().then(data);
Invalid syntax. async cannot prefix an expression.
B . Valid. Calls the function and attaches a .then() handler.
C . Invalid. .done() is not part of JavaScript Promise API.
D . Valid. Calls runParallel() and chains .then().
Therefore the correct answers are B and D.
JavaScript Knowledge Reference (text-only)
async functions return Promises.
Promises use .then() to retrieve resolved values.
.done() is not part of the standard Promise interface.

NEW QUESTION: 59
A developer is leading the creation of a new web server for their team that will fulfill API requests from an existing client. The team wants a web server that runs on Node.js, and they want to use the new web framework Minimalist.js. The lead developer wants to advocate for a more seasoned back-end framework that already has a community around it.
Which two frameworks could the lead developer advocate for?
A. Angular
B. Next
C. Gatsby
D. Next.js
Answer: (SHOW ANSWER)
The question is about:
Web frameworks that run on Node.js

Used to build servers and APIs
With established communities.
From the given options:
Angular:
A front-end framework running in the browser, not a Node.js server framework.
Gatsby:
A React-based static site generator; uses Node tooling but is not primarily a Node server framework.
Next / Next.js:
Server-side rendering / full-stack frameworks built on Node.js and React.
Used to create both server-rendered pages and APIs.
They have active communities and are "seasoned" compared to a hypothetical Minimalist.js.
Although typical Node back-end frameworks would include Express.js, Koa, or NestJS, given the options provided, the best fits for server-side frameworks with Node.js and an ecosystem are:
Next (interpreted as Next.js)
Next.js
Thus, the correct pair from the list is B and D.
Concepts: Node.js web frameworks, server-side rendering, community-backed frameworks vs minimal/new frameworks.

NEW QUESTION: 60

Which two console logs output NaN?

A. console.log(10 / 0);
B. console.log(parseInt('two'));
C. console.log(10 / Number('5'));
D. console.log(10 / 'five');
Answer: (SHOW ANSWER)
Recall:
NaN stands for "Not-a-Number".
Arithmetic involving non-numeric values that cannot be converted to numbers often results in NaN.
Check each:
A . 10 / 0
In JavaScript, dividing a positive number by 0 results in Infinity, not NaN.
So this logs Infinity.
B . parseInt('two')
parseInt tries to parse a numeric prefix from the string.
'two' does not start with numeric digits, so parseInt('two') returns NaN.
The console log prints NaN.
C . 10 / Number('5')
Number('5') → 5.

10 / 5 → 2.
Logs 2, not NaN.
D . 10 / 'five'
'five' is coerced to a number using Number('five'), which is NaN.
10 / NaN → NaN.
Logs NaN.
So the console logs that output NaN are from B and D.
Relevant concepts: NaN, Infinity, parseInt, Number() coercion, arithmetic with invalid numeric values.

Valid JS-Dev-101 Dumps shared by EduDump.com for Helping Passing JS-Dev-101 Exam! EduDump.com now offer the **newest JS-Dev-101 exam dumps**, the EduDump.com JS-Dev-101 exam **questions have been updated** and **answers have been corrected** get the **newest** EduDump.com JS-Dev-101 dumps with Test Engine here: https://www.edudump.com/exams/Salesforce/JS-Dev-101/premium/ (**149** Q&As Dumps, **35%OFF Special Discount Code: freecram**)