Actual4Dump

Choose the version that fits your needs	PDF Version	Desktop Test Engine	Online Test Engine
Latest and Up-to-Date exam dumps with real exam questions answers.	✓	✓	✓
Get 12-Months free updates without any extra charges.	✓	✓	✓
Experience same exam environment before appearing in the certification exam.	✗	✓	✓
100% exam passing guarante in the first attempt.	✓	✓	✓
20% discount on more than one license and 30% discount on 5+ license purchases.	✗	✓	✓
100% secure purchase on SSL.	✓	✓	✓
Completely private purchase without sharing your personal info with anyone.	✓	✓	✓

http://www.actual4dump.com

Superb Exam Dumps Materials lead you to get your certification easily - Actual4dump

Exam : **JavaScript-Developer-I-JPN**

Title : Salesforce Certified JavaScript Developer I Exam (JavaScript-Developer-I日本語版)

Vendor : Salesforce

Version : DEMO

QUESTION NO: 1
以下のコードを参照してください。
01 let total = 10;
02 const interval = setInterval(() = > {
03 total++;
04 clearInterval(interval);
05 total++;
06 }, 0);
07 total++;
08 console.log(total);
JavaScriptはシングルスレッドであることを考慮すると、コード実行後の8行目の出力はどうなりますか？
A. 11
B. 12
C. 10
D. 13
Answer: A
Explanation:
* 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.

QUESTION NO: 2
開発者は、DatePrettyPrint というモジュールを使用したいと考えています。
このモジュールは、printDate() というデフォルト関数を 1 つエクスポートします。
開発者はどのようにしてprintDate()をインポートして使用できますか？
A. import DatePrettyPrint() from ' /path/DatePrettyPrint.js ' ;
printDate();
B. import DatePrettyPrint from ' /path/DatePrettyPrint.js ' ;
DatePrettyPrint.printDate();
C. import printDate from ' /path/DatePrettyPrint.js ' ;
DatePrettyPrint.printDate();
D. import printDate from ' /path/DatePrettyPrint.js ' ;
printDate();
Answer: D
Explanation:
ES Modules follow these rules:
* Default exports are imported using:
* import anyName from ' module '
The name chosen in the import does NOT need to match the exported name.

* If a module exports:
* export default function printDate() {}
then consuming code should import the function directly:
import printDate from ' /path/DatePrettyPrint.js ' ;
* To execute it:
* printDate();
Option analysis:
* A incorrect: import DatePrettyPrint() is invalid syntax. Also calling printDate() without importing it is incorrect.
* B incorrect: A default export imported as DatePrettyPrint gives the function itself, not an object containing methods. You cannot call DatePrettyPrint.printDate().
* C incorrect: Imports printDate but then calls DatePrettyPrint.printDate(), which does not exist.
* D correct: Correct import of a default-exported function. Correct direct invocation.
JavaScript Knowledge References (text-only)
* Default export # imported without braces.
* Importing default functions gives a callable function, not an object.
* Correct syntax: import identifier from " module " .

QUESTION NO: 3
以下のコードを参照してください(8行目でテンプレートリテラルを使用するように修正済み)。

```
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 });
```

Promise.raceが実行されたとき、resultの値は何ですか？
A. 3号車はレースを完走しました。
B. 2号車はレースを完走しました。
C. レースは中止されました。
D. レース中に1号車がクラッシュしました。
Answer: B

Explanation:
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:
The answer: B
Study Guide / Concept References (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

QUESTION NO: 4
開発者は、File API
を使用してシンプルな画像アップロード機能を作成したいと考えています。
HTML:
< input type= " file " onchange= " previewFile() " >
< img src= " " height= " 200 " alt= " Image preview... " / >
JavaScript:
01 function previewFile() {
02 const preview = document.querySelector(' img ');
03 const file = document.querySelector(' input[type=file] ').files[0];
04 // line 4 code
05 reader.addEventListener(" load " , () = > {
06 preview.src = reader.result;
07 }, false);
08 // line 8 code
09 }
4行目と8行目のどのコードによって、選択したローカル画像が表示されるようになりますか？
A. 04 const reader = new File();
08 if (file) reader.readAsDataURL(file);
B. 04 const reader = new FileReader();
08 if (file) reader.readAsDataURL(file);
C. 04 const reader = new FileReader();
08 if (file) URL.createObjectURL(file);
Answer: B
Explanation:
The File API in browsers provides the FileReader object to read file contents selected from < input type= " file " > .
Important knowledge points:

* new FileReader() creates a file-reading object.
* .readAsDataURL(file) reads a file and produces a Base64 URL string.
* The " load " event fires when the file has finished reading.
* reader.result contains the data URL after reading completes.
Therefore, the correct implementation must:
* Create a FileReader instance:
* const reader = new FileReader();
* Call:
* reader.readAsDataURL(file);
* Use the load event handler to assign the image preview:
* preview.src = reader.result;
Option B is the only option that matches valid JavaScript File API usage.
Option A is incorrect because File is not a constructor for reading files.
Option C is incorrect because URL.createObjectURL(file) must be assigned directly as a URL, not used with reader.result.
JavaScript Knowledge References (text-only)
* The file-reading interface in browsers is FileReader.
* readAsDataURL() loads files as Base64 data URLs.
* The load event indicates when the reader has finished and reader.result is available.

QUESTION NO: 5
開発者が、イベントとコールバックを使用して構築されたクライアントライブラリを備えた新しいNode.jsサーバーをセットアップしています。
図書館：
* WebSocket接続を確立し、サーバーへのメッセージの受信を処理します。
* require でインポートされ、ws という変数で使用できるようになります。
*
開発者は、接続が失敗した場合にエラーログを記録する機能も追加したいと考えています。
この情報に基づいて、実行時にリッスンする2つのイベントを持つクライアントを正しく設定する方法を示しているコードセグメントはどれですか？
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: B
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 .

QUESTION NO: 6

開発者は、エンドポイントの1つで実行時エラーが繰り返し発生するため、Node.jsウェブサーバーのデバッグを行う必要がある。
開発者は、ローカルマシン上でエンドポイントをテストし、ローカルサーバーに対してリクエストを送信して動作を確認したいと考えています。ソースコードでは、server.jsファイルがサーバーを起動します。開発者は、ターミナルのみを使用してNode.jsサーバーのデバッグを行いたいと考えています。
開発者が現在のターミナルウィンドウでCLIデバッガーを開くには、どのコマンドを使用すればよいですか？
(タイプミスを修正済み：node_inspect # ノード検査、node_start_inspect # ノード開始検査)
A. node start inspect server.js
B. node -i server.js
C. node inspect server.js
D. node server.js --inspect
Answer: C

QUESTION NO: 7
開発者は以下を実行します。
ドキュメントクッキー;
document.cookie = ' key=John Smith ' ;
その行動とはどのようなものか？
A. 行 01 は document.cookies
であるべきですが、キー値が設定され、すべてのクッキーが削除されるため、クッキーは読み取られません。
B. クッキーが読み取られ、キー値が設定され、すべてのクッキーが削除されます。
C. クッキーが読み取られ、キー値が設定されます。残りのクッキーは影響を受けません。
D. Cookie は読み取られますが、値が URL
エンコードされていないため、キー値は設定されません。
Answer: C

QUESTION NO: 8
開発者がページをリロードせずにブラウザのナビゲーション履歴を更新できるようにするには、どのステートメントを使用すればよいですか？
A. window.customHistory.pushState(newStateObject, ' ' , null);
B. window.history.createState(newStateObject, ' ');
C. window.history.pushState(newStateObject, ' ' , null);
D. window.history.updateState(newStateObject, ' ');
Answer: C
Explanation:
The correct answer is C .
The browser provides the History API through:
window.history
To add a new entry to the browser's session history without refreshing the page, JavaScript uses:
window.history.pushState(state, title, url);

So the valid statement is:
window.history.pushState(newStateObject, ' ' , null);
This updates the browser history stack without forcing a full page reload. It is commonly used in single-page applications when changing views or routes dynamically.
The method accepts three arguments:
window.history.pushState(stateObject, title, url);
stateObject stores custom data associated with the history entry.
title is usually passed as an empty string because many browsers ignore it.
url optionally changes the displayed URL. Passing null means no new URL is provided.
Option A is incorrect because:
window.customHistory
is not the standard browser History API object.
Option B is incorrect because:
createState()
is not a valid History API method.
Option D is incorrect because:
updateState()
is not a valid History API method.
The correct browser API method is:
window.history.pushState()
Therefore, the verified answer is C .

QUESTION NO: 9
開発者は、ユーザーから報告されたバグを修正するよう依頼された。そのため、開発者はデバッグ用のブレークポイントを追加する。
01 function Car(maxSpeed, color) {
02 this.maxSpeed = maxSpeed;
03 this.color = color;
04 }
05 let carSpeed = document.getElementById(' carSpeed ');
06 デバッガー;
07 let fourWheels = new Car(carSpeed.value, ' red ');
コードの実行が6行目のブレークポイントで停止した場合、ブラウザのコンソールにはどのような2種類の情報が表示されますか？
A. Car オブジェクト用に作成されたインスタンスの数を表示する変数
B. window.localStorage プロパティに格納されている情報
C. carSpeed 変数と fourWheels 変数の値
D. carSpeed DOM要素に適用されるスタイル、イベントリスナー、その他の属性
Answer: B D
Explanation:
When execution hits the debugger; statement on line 06, JavaScript execution pauses at that point. Most modern browsers (for example, using DevTools) allow you to inspect the current scope , DOM, and various browser APIs in the console.
Let's look at what is available precisely at line 06.
Current code state at the breakpoint:

* Lines 01-04 define the Car constructor function.
* Line 05 executes:
* let carSpeed = document.getElementById(' carSpeed ');
So at line 06:
* carSpeed is a variable containing a reference to a DOM element (the element with id " carSpeed ").
* Line 07 has not executed yet:
* let fourWheels = new Car(carSpeed.value, ' red ');
So fourWheels has not been created and is not accessible yet (it is in the temporal dead zone for the let declaration).
Now check each option:
Option A:
" A variable displaying the number of instances created for the Car object "
* This code does not implement any mechanism to count instances of Car.
* There is no static property, global counter, or similar variable tracking the number of Car instances.
* At line 06, no instance of Car has even been created yet (new Car(...) is on line 07, which has not run).
* Therefore, there is no such variable by default in JavaScript or DevTools.
This option is not available.
Option B:
" The information stored in the window.localStorage property "
* At a breakpoint, the console is fully usable to inspect global objects.
* window.localStorage is always accessible from the console (assuming standard browser context).
* You can type:
* window.localStorage
and inspect key/value pairs stored there.
* This is independent of the current function or breakpoint line; localStorage is part of the Web Storage API on the window object.
So this information is indeed available in the console at line 06.
Option C:
" The values of the carSpeed and fourWheels variables "
* At line 06:
* carSpeed has been declared and assigned (line 05), so it is available, and its value (a DOM element) can be inspected.
* fourWheels is declared on line 07 with let and has not yet been executed.
* Variables declared with let and const are in a temporal dead zone before their declaration line completes.
* At line 06, fourWheels is not yet initialized, and attempting to access it would result in a ReferenceError.
Thus, you can inspect carSpeed but not fourWheels. The option explicitly says " the values of the carSpeed and fourWheels variables " , which is not correct at this breakpoint, because fourWheels is not available yet.
Option D:

" The style, event listeners and other attributes applied to the carSpeed DOM element "
* carSpeed holds a reference to a DOM element (document.getElementById(' carSpeed ')).
* In DevTools, you can inspect this element in several ways:
* Typing carSpeed in the console.
* Inspecting it in the Elements panel.
* From the console or Elements panel, you can view:
* Its style (inline styles and computed styles).
* Event listeners attached to it (using event listener viewer in DevTools).
* Other attributes (id, class, etc.).
All of this is accessible at the point where execution is paused.
Therefore, this information is available at the breakpoint.
Conclusion:
* B is available (global window.localStorage).
* D is available (full inspection of the carSpeed DOM element).
* A is not present in this code or environment.
* C is partially wrong (only carSpeed exists; fourWheels does not yet).
So the two correct answers are:
The answer: B, D
References of JavaScript knowledge documents or Study Guide (concept names only):
* debugger statement and pausing execution
* JavaScript execution context and scope at a breakpoint
* let declarations and temporal dead zone
* Browser DevTools console and inspection of variables
* DOM access via document.getElementById and inspection of elements
* Web Storage API: window.localStorage

QUESTION NO: 10
以下のJavaScriptコードを参照してください。
function onLoad() {
console.log("ページが読み込まれました！");
}
開発者は、ブラウザでページを読み込んだ後、ログステートメントをどこで確認できますか？
A. ブラウザのJavaScriptコンソールで
B. Webサーバーを実行しているターミナルコンソールで
C. ブラウザのパフォーマンスツールログ
D. ウェブページのコンソールログ
Answer: A
Explanation:
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:
The answer: 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

QUESTION NO: 11
型強制を考慮すると、次の式はどのような値に評価されますか？
true + ' 13 ' + NaN
A. ' true13NaN '
B. ' 113NaN '
C. 14
D. ' true13 '
Answer: A
Explanation:
Expression:
true + ' 13 ' + NaN
The + operator is left-associative, so evaluation order:
* true + ' 13 '
* When one operand is a string, + performs string concatenation.
* true is converted to string ' true ' .
* ' true ' + ' 13 ' # ' true13 ' .
* Result from step 1 with NaN:
' true13 ' + NaN
* Again, one operand is a string, so concatenation.
* NaN is converted to string ' NaN ' .
* ' true13 ' + ' NaN ' # ' true13NaN ' .

Final value: ' true13NaN ' .
So A is correct.
Why others are wrong:
* B: ' 113NaN ' would require true to coerce to 1 first and no string to be present, which is not the case because ' 13 ' forces string concatenation.
* C: 14 would require pure numeric addition, which is not the case once a string is involved.
* D: ' true13 ' ignores the final + NaN part.
Concepts: type coercion with +, boolean to string, NaN to string, left-associative evaluation.

QUESTION NO: 12
const str = ' Salesforce ' ;
どの2つの文を組み合わせると「Sales」という単語になりますか？
A. str.substring(0, 5);
B. str.substr(s, 5);
C. str.substring(0, 5);
D. str.substr(0, 5);
Answer: A D
Explanation:
Comprehensive and Detailed Explanation:
* " 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.

QUESTION NO: 13
以下のコードを参照してください。
01 let sayHello = () = > {
02 console.log(' Hello, World! ');
03 };
今から2分後にsayHelloを1回実行するコードはどれですか？
A. delay(sayHello, 120000);
B. setTimeout(sayHello, 120000);
C. setTimeout(sayHello(), 120000);
D. setInterval(sayHello, 120000);
Answer: B
Explanation:
To schedule a function for later execution in JavaScript, the standard built-in method is:
setTimeout(functionReference, delayInMilliseconds)
Characteristics of setTimeout():
* Executes only once .

* Delay is specified in milliseconds.
* Requires passing the function reference , not calling the function.
Two minutes is:
2 minutes = 120 seconds = 120,000 milliseconds
Now evaluate each option:
Option A: delay(sayHello, 120000);
Incorrect.
There is no built-in delay() function in JavaScript.
Option B: setTimeout(sayHello, 120000);
Correct.
sayHello is passed as a function reference , so it will execute once after 120,000 ms.
Option C: setTimeout(sayHello(), 120000);
Incorrect.
sayHello() calls the function immediately , and the return value (undefined) is passed to setTimeout.
This executes the function right away instead of after two minutes.
Option D: setInterval(sayHello, 120000);
Incorrect.
setInterval() repeats execution every 120,000 ms.
The question requires executing the function once , so this cannot be correct.
JavaScript Knowledge References (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().

QUESTION NO: 14
値が与えられたとき、開発者はその値がNaNかどうかを検出するために、どの2つの方法を使用できますか？
A. value === Number.NaN
B. value == NaN
C. isNaN(value)
D. Object.is(value, NaN)
Answer: C D
Explanation:
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)

QUESTION NO: 15
以下のHTMLを参照してください。
< div id= " main " >
< ul >
< li > Leo < /li >
< li > Tony < /li >
< li > Tiger < /li >
< /ul >
< /div >
JavaScriptのどの文で「Leo」が「The Lion」に変更されますか？
A. document.querySelectorAll(' #main #Leo ').innerHTML = ' The Lion ' ;
B. document.querySelector(' #main li:second-child ').innerHTML = ' The Lion ' ;
C. document.querySelectorAll(' #main li,Leo ').innerHTML = ' The Lion ' ;
D. document.querySelector(' #main li:nth-child(1) ').innerHTML = ' The Lion ' ;
Answer: D
Explanation:
We want to select the first < li > under #main and change its content.
* document.querySelector(selector) returns the first element matching the CSS selector.
* In the DOM, the < li > elements under < ul > are children in order:
* 1st child: " Leo "
* 2nd child: " Tony "
* 3rd child: " Tiger "
li:nth-child(1) is the CSS pseudo-class that selects the first < li > inside its parent.
So:
document.querySelector(' #main li:nth-child(1) ').innerHTML = ' The Lion ' ; selects the first < li > (" Leo ") inside #main and sets its innerHTML to " The Lion " .
Why others are incorrect:
* A. #main #Leo
* There is no element with id " Leo " . id is not set on the < li > , so the selector matches nothing.
* B. li:second-child
* There is no :second-child pseudo-class. The valid pseudo-class is :nth-child(2); and that would select " Tony " , not " Leo " .
* C. ' #main li,Leo '
* This selector means "all #main li and all elements named < Leo > ".

* There is no < Leo > tag, and querySelectorAll returns a NodeList, which does not have a single innerHTML property.
Therefore, D is the correct statement.
Study Guide Concepts:
* document.querySelector vs querySelectorAll
* CSS selectors: nth-child()
* DOM element innerHTML property