

IT 資格 道 場

HTML5プロフェッショナル 認定試験 レベル2 学習用テ キスト

予備知識ゼロから合格ラインへ —— 読んで理解する1冊目

無料サンプル(第1章まで)

LPI-Japan 公式「HTML5プロフェッショナル認定試験 レベル2 出題範囲」Ver2.5
(HTML 5.1 対応) 準拠
2026年10月版

ご利用にあたって

- このテキストは「IT資格道場」(www.shikaku-doujou.com)の購入者限定テキストのうち、学習用テキストの第1章までを収録した**無料サンプル**です。どなたでもご覧いただけます。
- 無料サンプルも著作物です。転載・改変しての再配布・二次販売はできません。
- AI モデルの学習データとしてのご利用(学習目的の入力・データセット化を含む)はできません。
- 製品版(学習用テキスト・頻出度別ノート・暗記用ノートの3点)は、ご購入者を識別できる透かし入りのPDFとして提供されます。
- 本PDFはスマートフォン・タブレットでの閲覧に合わせたA5版面・文字サイズで組版しています。

目次

第1章 JavaScript文法 [2.1.1]

型変換と算術演算子

typeof とプリミティブ型

parseInt と数値への変換

等価演算子 == と ===

巻き上げとスコープチェーン

this と call・apply・bind

クロージャ

strict モードと例外

URI のエスケープ関数とその他のグローバル関数

サンプルはここまでです

HTML5 プロフェッショナル認定試験 レベル2 の出題範囲 9 分野・28 小分類を、公式の番号の順に 1 章ずつ並べた学習用テキストです。範囲は JavaScript の文法から、ブラウザの API、グラフィックス、ストレージ、通信、デバイス、オフライン、セキュリティまでです。コードの出力は node と Chromium で実際に動かして確かめました。画面にだけ出るものは、動かした環境の値を示しています。

SAMPLE

第1章 JavaScript文法 [2.1.1]

試験の中心は、短いコードを読んで出力を言い当てる問題です。型変換と比較、巻き上げ、this、クロージャ、strict モードの結果を、規則から導けるようにします。

型変換と算術演算子

`+` は、片方が文字列なら連結になり、両方が数値の時だけ足し算になります。`-`、`*`、`/`、`%` は常に数値演算で、足し算の連結のような動きはしません。式は左から順に評価されるので、順序で結果が変わります。

```
console.log("7" + 2, "7" - 2, "7" * "2");
console.log(7 + "2" + 1, 1 + 2 + "3");
console.log(true + 1, null + 1, undefined + 1, [] + 1);
console.log("3" > "12", 3 > "12", "a" < "b");
console.log(10 % 4, -7 % 3, 2 ** 10, 7 / 0, -7 / 0, 0 / 0);
```

```
72 5 14
721 33
2 1 NaN 1
true false true
2 -1 1024 Infinity -Infinity NaN
```

`7 + "2" + 1` は先に `"72"` ができるので、最後も連結で `"721"` になります。`1 + 2 + "3"` は先に `3` ができて `"33"` です。数値に直すと、`true` は `1`、`false` と

`null` は `0`、`undefined` は `NaN` になります。空配列は文字列 `""` に変わるので、`[] + 1` は `"1"` で、足し算にはなりません。

比較演算子は、両方が文字列なら文字コードの順に比べます。`"3" > "12"` が `true` になるのはこのためです。片方が数値なら、もう一方を数値に直して比べます。`%` の結果の符号は左辺で決まり、右辺には従いません。`0` で割ると `Infinity` か `-Infinity` が返り、`0 / 0` は `NaN` です (3 つとも `number` 型の値)。

`&&` と `||` は、真偽値に直さず、どちらかの被演算子をそのまま返します。偽として扱われる値は次の 6 つです。

- `false`
- `0`
- `""`
- `null`
- `undefined`
- `NaN`

`??` は、左が `null` か `undefined` の時だけ右を返します。後置の `n++` は増やす前の値を返し、前置の `++n` は増やした後の値を返します。

```
console.log(0 || "x", "" || null, "a" && "b", 0 && "b", null ?? "d", 0 ?? "d");  
var n = 5;  
console.log(n++, n, ++n, n--, n);
```

```
x null b 0 d 0
5 6 7 7 6
```

typeof とプリミティブ型

プリミティブ型は、number、string、boolean、undefined、null、symbol、bigint の 7 つです。それ以外はオブジェクトで、配列、関数、日付もこちらに入ります。typeof が返すのは文字列で、型そのものではありません。

```
console.log(typeof 1, typeof "a", typeof true, typeof undefined);
console.log(typeof null, typeof {}, typeof [], typeof function () {});
console.log(typeof Symbol(), typeof 10n, typeof NaN, typeof Infinity);
console.log(typeof new String("a"), typeof String("a"), typeof
undeclaredVar);
console.log(Array.isArray([]), [] instanceof Array);
```

```
number string boolean undefined
object object object function
symbol bigint number number
object string undefined
true true
```

typeof null は歴史的な経緯で "object" を返します。配列も "object" なので、配列の判定には Array.isArray() を使います。関数だけは "function" で、NaN の typeof は "number" です。宣言していない変数に typeof を使ってもエラーにはならず、"undefined" が返ります。new String("a") はラッパー

オブジェクトなので `"object"` ですが、`new` を付けずに `String("a")` は `"string"` を返すだけで、オブジェクトは作りません。

`undefined` は、宣言だけした変数や、渡されなかった引数を持つ値です。`null` は「値が無い」ことを書き手が示すために入れる値で、勝手には入りません。

parseInt と数値への変換

`parseInt` は、先頭から数字として読める所までを整数にします。先頭の空白は読み飛ばし、符号も受け付けます。基数を省くと 10 進数として読みます (`0x` で始まる文字列だけは 16 進数)。

```
console.log(parseInt("42.9"), parseInt(" 7 "), parseInt("-0x1f"),
parseInt("0b11"));
console.log(parseInt("11", 2), parseInt("z", 36), parseInt(""),
parseInt("08"), parseInt("12", 1));
console.log(Number("12px"), Number(""), Number(" 5 "), Number(null),
Number(undefined), Number("0x10"));
console.log(isNaN("abc"), Number.isNaN("abc"), isFinite("12"),
Number.isFinite("12"));
```

```
42 7 -31 0
```

```
3 35 NaN 8 NaN
```

```
NaN 0 5 0 NaN 16
```

```
true false true false
```

`"0b11"` の `0b` は `parseInt` が認識しないので、先頭の 0 だけを読んで 0 になります。基数に使える範囲は 2 から 36 までで、1 を渡すと `NaN` です。読める数

字が1つも無ければ `NaN` で、空文字列も同じ結果になります。`parseFloat` は小数点を読みますが、16 進数は読みません。

`parseInt` を `map` にそのまま渡す書き方には注意が必要です。`map` はコールバックへ(値, 添字, 配列)の3つを渡すので、`parseInt` は添字を基数として受け取ります。`["10", "10", "10"].map(parseInt)` は、`parseInt("10", 0)`、`parseInt("10", 1)`、`parseInt("10", 2)` の結果になり、`[10, NaN, 2]` です。基数0は省いた時と同じ10進数の扱いで、基数1は範囲外なので `NaN` になります。`["1", "2", "3"].map(parseInt)` も `[1, NaN, NaN]` で、`[1, 2, 3]` にはなりません。意図どおりに変えるには `map(Number)` か `map(v => parseInt(v, 10))` と書きます。

`Number()` は文字列の全体を数値に直します。だから `"12px"` は `NaN` になり、`parseInt("12px")` の12とは一致しません。`Number("")` と `Number(null)` は0になり、`Number(undefined)` だけが `NaN` です。

グローバルの `isNaN()` と `isFinite()` は、引数をまず数値に変換します。`isNaN("abc")` が `true` になるのはそのためです。`Number.isNaN()` と `Number.isFinite()` は変換をしません。数値以外には常に `false` を返します。

等価演算子 == と ===

`===` は、型が違えば `false` で、同じ型なら値を比べます。`==` は、型が違う時に型を合わせてから比べますが、その規則は直感どおりには動きません。

```
console.log(0 == "", 0 == "0", "" == "0", false == "0");
console.log(null == 0, null >= 0, null == undefined, undefined == 0);
console.log(NaN === NaN, Object.is(NaN, NaN), [1] == 1, [1, 2] ==
"1,2");
console.log({} === {}, [] == []);
function sw(v) { switch (v) { case 1: return "one"; case "1": return
"str"; default: return "other"; } }
console.log(sw(1), sw("1"), sw(true));
```

```
true true false true
false true true false
false true true true
false false
one str other
```

`==` は、数値と文字列なら文字列を数値に直し、真偽値は先に数値に直します。`null` と `undefined` は、お互いにだけ等しく、0 や空文字列とは等しくなりません。オブジェクトと数値や文字列の比較では、オブジェクトを文字列や数値に直します。`[1] == 1` が `true` になる理由です。

`null == 0` は `false` ですが、`null >= 0` は `true` です。大小の比較は `null` を 0 に直しますが、`==` は `null` を特別に扱います。`NaN` は自分自身とも等しくありません。`Object.is(NaN, NaN)` だけが `true` を返します。オブジェクト同士は、同じ実体を指す時だけ等しいと判定されます。`switch` の `case` は `===` で比べるので、型変換は起きません。

巻き上げとスコープチェーン

`var` の宣言は、関数の先頭 (関数の外ならスクリプトの先頭) へ巻き上げられます。巻き上げられるのは宣言だけで、代入は元の位置から動きません。関数宣言は本体ごと巻き上げられるので、宣言より前の行から呼べます。関数式を `var` に代入した場合は、変数の宣言だけが巻き上げられ、値は代入の行まで `undefined` です。

```
console.log(a, typeof f, typeof g);
var a = 10;
function f() {}
var g = function () {};
var s = "global";
function outer() {
  console.log(s);
  var s = "outer";
  return s;
}
console.log(outer());
console.log(typeof h);
if (true) { var h = 1; }
console.log(typeof h);
```

```
undefined function undefined
undefined
outer
undefined
number
```

`outer()` の中の最初の `console.log(s)` は、グローバルの `s` ではなく、巻き上げられた内側の `s` を見ます。だから `undefined` になります。`var` の境は関数で、ブロックは境になりません。`if` の中の `var h` が、ブロックの外から見えるのはこのためです。

変数の名前は、現在の関数、外側の関数、グローバルの順に探されます。この連なりがスコープチェーンです。関数は書かれた場所の外側をたどります。呼び出された場所とは関係しません。

`with` 文は、指定したオブジェクトをスコープチェーンの先頭に加えます。ブロックの中では、そのオブジェクトのプロパティを名前だけで読み書きでき、`o.` を付ける必要はありません。

```
var o = { p: 1, q: 2 };
with (o) {
  p = 10;
  var r = p + q;
}
console.log(o.p, o.q, r);
```

```
10 2 12
```

`p = 10` は `o.p` への代入で、`var r` の `r` はブロックの外に残ります。strict モードでは `with` を書けません。

this と call・apply・bind

`this` の値は、関数を書いた場所ではなく、呼び出し方で決まります。

呼び出し方	this
<code>f()</code>	グローバルオブジェクト (strict モードでは <code>undefined</code>)
<code>obj.f()</code>	<code>obj</code>
<code>new F()</code>	新しく作られるオブジェクト
<code>f.call(x)</code> 、 <code>f.apply(x)</code> 、 <code>f.bind(x)</code>	<code>x</code>

`Function.call()` と `Function.apply()` は、第 1 引数を `this` にして関数をすぐ呼びます。違いは引数の渡し方だけで、`call` は 2 番目以降に 1 つずつ並べ、`apply` は配列で渡します。`bind` は呼ばずに、`this` を固定した新しい関数を返します。

```
function who(g, p) { return this.name + ":" + g + p; }
var a = { name: "A" }, b = { name: "B" };
console.log(who.call(a, "hi", "!"), who.apply(b, ["yo", "?"]));
var bound = who.bind(a, "bound");
console.log(bound("."), bound.call(b, "x"));
var o = { name: "O", m: function () { return this === o; } };
var m = o.m;
console.log(o.m(), m());
function Person(n) { this.name = n; }
Person.prototype.hello = function () { return "I am " + this.name; };
var p = new Person("K");
console.log(p.hello(), p.constructor === Person,
Object.getPrototypeOf(p) === Person.prototype);
console.log(p.hasOwnProperty("name"), p.hasOwnProperty("hello"),
"hello" in p);
for (var k in p) console.log("for-in:", k);
console.log(Object.keys(p).join());
```

```
A:hi! B:yo?
A:bound. A:boundx
true false
I am K true true
true false true
for-in: name
for-in: hello
name
```

`bind` で固定した `this` は、後から `call` で上書きできません。メソッドを変数に取り出した `m()` は、`o.` が付かない普通の呼び出しなので、`this` は `o` ではありません。

`new` で作ったオブジェクトは、コンストラクタの `prototype` を内部のプロトタイプとして持ちます。プロパティを読む時は、そのオブジェクト自身、そのプロトタイプ、さらにその先と順にたどります。この連なりをプロトタイプチェーンと呼びます（終点は `null`）。`hasOwnProperty` は自身のプロパティだけを調べますが、`in` 演算子はプロトタイプのプロパティも調べます。`for/in` は継承した列挙可能なプロパティも出し、`Object.keys()` は自身のものだけを返します。

クロージャ

クロージャは、関数が、作られたスコープの変数を覚え続ける仕組みです。外側の関数が終わった後も、内側の関数はその変数を失いません。外側の関数を呼ぶたびに変数は新しく作られ、返された関数はそれぞれ自分の変数を持ちます。

```
function counter() { var c = 0; return function () { return ++c; }; }  
var x = counter(), y = counter();  
x(); x();  
console.log(x(), y());  
var fs = [];  
for (var i = 0; i < 3; i++) { fs.push(function () { return i; }); }  
console.log(fs.map(function (f) { return f(); }).join());  
var gs = [];  
for (var j = 0; j < 3; j++) { gs.push((function (n) { return function  
( ) { return n; }; })(j)); }  
console.log(gs.map(function (f) { return f(); }).join());
```

```
3 1  
3,3,3  
0,1,2
```

`x` と `y` は別の `c` を持つので、数え方は互いに影響しません。ループで関数を作る例では、`var i` が1つしかなく、3つの関数が同じ `i` を見ます。呼び出す時点で `i` はすでに3で、どれも3を返します。ループの各回に即時関数を呼べば、その回の値が引数 `n` として別々に残ります。

strict モードと例外

スクリプトか関数の先頭に `"use strict";` と書くと、その範囲が strict モードになります。先頭以外では効きません。黙って通っていた誤りが、strict モードではエラーになります。

```
"use strict";

function t(f) { try { console.log(f()); } catch (e) {
console.log(e.name); } }

t(function () { undeclared = 1; return "ok"; });

t(function () { return this; });

t(function () { Object.freeze({ a: 1 }).a = 2; return "ok"; });

t(function () { return eval("var ev = 1; typeof ev") + typeof ev; });

t(function () { return eval("with ({}){}"); });
```

ReferenceError

undefined

TypeError

numberundefined

SyntaxError

宣言していない変数への代入は `ReferenceError` です。関数として呼んだ時の `this` は `undefined` になります。書き込めないプロパティへの代入は `TypeError` です。`eval` の中で宣言した変数は、`eval` の外に出ません。`with` 文と、`010` のような 8 進数のリテラルは `SyntaxError` になります。

// "use strict" を書かない場合

```
function t(f) { try { console.log(f()); } catch (e) {  
  console.log(e.name); } }  
  
t(function () { undeclared = 1; return "ok"; });  
  
t(function () { return this === window; });  
  
t(function () { var o = Object.freeze({ a: 1 }); o.a = 2; return o.a;  
});
```

```
ok  
true  
1
```

strict モードでなければ、未宣言の代入は成功して暗黙のグローバル変数になり、`this` はグローバルオブジェクトです。凍結したオブジェクトへの書き込みも、エラーにならず黙って無視され、エラーは出ません。

`try/catch` 文では、`try` の中で例外が起きると `catch` が実行されます。`finally` は例外の有無に関係なく、最後に必ず実行されます。`try` の中で `return` した場合も、`finally` を実行してから戻ります。

エラー	起きる場面
<code>ReferenceError</code>	存在しない変数の参照
<code>TypeError</code>	型に合わない操作
<code>RangeError</code>	値が許される範囲の外
<code>SyntaxError</code>	構文の誤り

エラー	起きる場面
URIError	URI 関数への不正な入力

```
function f() { try { return "try"; } finally { console.log("finally"); } }  
console.log(f());  
try { null.x; } catch (e) { console.log(e.name); }  
try { undefinedFn(); } catch (e) { console.log(e.name); }  
try { (1).toFixed(101); } catch (e) { console.log(e.name); }  
try { decodeURIComponent("%"); } catch (e) { console.log(e.name); }  
try { JSON.parse("{'a':1}"); } catch (e) { console.log(e.name); }
```

finally
try
TypeError
ReferenceError
RangeError
URIError
SyntaxError

`SyntaxError` は、コードを読み込む段階で起き、`catch` では拾えません。実行中に構文を解析する `JSON.parse` や `eval` だけが、`catch` で拾える `SyntaxError` を投げます。

URIのエスケープ関数とその他のグローバル関数

値を URL のクエリ文字列に埋め込む時は、`encodeURIComponent()` を使います。`&`、`=`、`/`、`?` のような区切り文字もパーセントエンコードするので、値の中の区切りが別の項目に化けることはありません。`encodeURIComponent()` は URI 全体を想定して、区切り文字を残します。デコードは `decodeURIComponent()` と `decodeURI()` が対応します。

```
var q = "a b&c=d/e?f";
console.log(encodeURIComponent(q));
console.log(encodeURI(q));
console.log(decodeURIComponent("a%20b%26c"), decodeURI("a%20b%26c"));
console.log(encodeURIComponent("日本"));
console.log(eval("1 + 2"), eval("var z = 5; z * 2"));
```

```
a%20b%26c%3Dd%2Fe%3Ff
a%20b&c=d/e?f
a b&c a b%26c
%E6%97%A5%E6%9C%AC
3 10
```

`eval()` は文字列をコードとして実行し、最後の式の値を返します。古い `escape()` と `unescape()` は非推奨なので、URL の組み立てには使いません。

組み込みのグローバルオブジェクトには、`Array`、`String`、`Number`、`Date`、`Math`、`JSON` があります。`JSON.stringify()` は、`undefined` と関数をオブジェクトのプロパティなら省き、配列の要素なら `null` にします。`JSON.parse()` は

シングルクォートの文字列を受け付けません。不正な入力には `SyntaxError` を投げます。

```
console.log(JSON.stringify({ a: 1, d: undefined, e: function () {}, f: null, h: NaN }));  
console.log(JSON.stringify([undefined, function () {}]));  
console.log(Math.max(), Math.min(), Math.max(1, "3", 2),  
Math.round(-2.5), Math.round(2.5));  
console.log([3, 20, 100].sort().join(), [3, 20, 100].sort(function (a, b) { return a - b; }).join());
```

```
{ "a": 1, "f": null, "h": null }  
[null, null]  
-Infinity Infinity 3 -2 3  
100,20,3 3,20,100
```

`Math.max()` は引数が無いと `-Infinity` を返し、文字列の `"3"` は数値に変換されます。`Math.round()` は `.5` を正の無限大の方向へ丸めるので、`-2.5` は `-2` になります。`sort()` は比較関数が無いと、要素を文字列にして並べます。数値の並べ替えに、比較関数は欠かせません。

サンプルはここまでです

続き(第2章～巻末)は、HTML5プロフェッショナル認定試験レベル2 問題集のご購入で、頻出度別ノート・暗記用ノートと合わせて3点すべてダウンロードできます。

SAMPLE

HTML5プロフェッショナル認定試験 レベル2 学習用テキスト(無料サンプル)

版

2026年10月版(LPI-Japan 公式「HTML5プロフェッショナル認定試験 レベル2 出題範囲」
Ver2.5 (HTML 5.1 対応) 準拠)

発行

IT資格道場(<https://www.shikaku-doujou.com>)

お問い合わせ

info@shikaku-doujou.com

「HTML5プロフェッショナル認定試験」は特定非営利活動法人エルピーアイジャパンの登録商標です。記載されているその他の会社名・製品名・サービス名は、各社の商標または登録商標です。

本教材は公式の出題範囲 (Ver2.5) の内容を基に作成したもので、特定非営利活動法人エルピーアイジャパンとの提携・承認関係はありません。

本文・図表・設問はすべてオリジナルの著作物です。無断での転載・複製・再配布・二次販売はできません。お気づきの点はinfo@shikaku-doujou.com までご連絡ください。