# 1) Basics, Types, Coercion (1–15)

**1) What is the result of `typeof null`?**
A) `"null"` B) `"object"` C) `"undefined"` D) `"number"`
**Answer: B**
**Explanation:** A long-standing JS quirk: `typeof null` returns `"object"`.

**2) What does `NaN === NaN` evaluate to?**
A) true B) false
**Answer: B**
**Explanation:** `NaN` is not equal to anything, even itself. Use `Number.isNaN(x)`.

**3) What's the difference between `==` and `===`?**
**Answer:** `==` coerces types; `===` requires same type and value.
**Explanation:** Prefer `===` to avoid surprise coercion.

**4) What is the output?**

```
console.log(0 == false, 0 === false);
```

**Answer:** `true false`
**Explanation:** `==` coerces (`false` → `0`), `===` doesn't.

**5) What is the output?**

```
console.log("5" + 1, "5" - 1);
```

**Answer:** `"51" 4`
**Explanation:** `+` can concatenate strings; `-` forces numeric conversion.

**6) What does `Number.isNaN("NaN")` return?**
A) true B) false
**Answer: B**
**Explanation:** `"NaN"` is a string, not the numeric `NaN`.

**7) What is `Object.is(-0, 0)`?**
A) true B) false
**Answer: B**
**Explanation:** `Object.is` distinguishes `-0` and `0` (unlike `===`).

**8) What are JS primitive types?**
**Answer:** `string, number, bigint, boolean, undefined, symbol, null`
**Explanation:** Everything else is an object (including functions).

**9) What is the output?**

```
console.log(Boolean([]), Boolean({}), Boolean(""));
```

**Answer:** `true true false`
**Explanation:** Empty array/object are truthy; empty string is falsy.

**10) What does `parseInt("08")` return (modern JS)?**
A) 8 B) 0 C) NaN
**Answer: A**
**Explanation:** Modern JS defaults to base 10 unless prefixed like `0x`. Best: `parseInt(str, 10)`.

**11) What is the output?**

```
console.log(1 + "2" + 3);
```

**Answer:** `"123"`
**Explanation:** Once string concatenation starts, `+` keeps concatenating.

**12) What's the difference between `null` and `undefined`?**
**Answer:** `undefined` = not assigned / missing; `null` = intentionally empty.
**Explanation:** Both are "no value," but different intent and behavior.

**13) What is the output?**

```
console.log(typeof (() => {}));
```

**Answer:** `"function"`
**Explanation:** Functions are callable objects; `typeof` has a special `"function"` result.

**14) What's the output?**

```
console.log([] == false);
```

**Answer:** `true`
**Explanation:** `[]` coerces to `""`, `false` coerces to `0`, `""` coerces to `0`.

**15) What is the output?**

```
console.log("2" * "3");
```

**Answer:** `6`
**Explanation:** `*` forces numeric conversion.

## 2) Scope, Hoisting, TDZ (16–30)

**16) `var`, `let`, `const` differences?**
**Answer:**

- `var`: function-scoped, hoisted (initialized as `undefined`).
- `let/const`: block-scoped, hoisted but in TDZ until declared.
- `const`: can't be reassigned (but object contents can mutate).
  **Explanation:** TDZ = Temporal Dead Zone.

**17) What happens?**

```
console.log(a);
var a = 10;
```

**Answer:** `undefined`
**Explanation:** `var a` is hoisted, value assigned later.

**18) What happens?**

```
console.log(b);
let b = 10;
```

**Answer:** ReferenceError
**Explanation:** `let` is in TDZ until the declaration line.

**19) What is the output?**

```
for (var i=0; i<3; i++) {}
console.log(i);
```

**Answer:** `3`
**Explanation:** `var` is function-scoped, not block-scoped.

**20) What is the output?**

```
for (let j=0; j<3; j++) {}
console.log(j);
```

**Answer:** ReferenceError
**Explanation:** `let` is block-scoped.

**21) What does “hoisting” mean?**
**Answer:** Declarations are processed before code runs.
**Explanation:** Variables/functions can be referenced earlier (with rules depending on type).

**22) Are function declarations hoisted?**
**Answer:** Yes, fully (name + body).
**Explanation:** You can call them before they appear in code.

**23) Are function expressions hoisted?**

```
foo();
var foo = function(){};
```

**Answer:** TypeError (`foo` is undefined at call time).
**Explanation:** `var foo` hoists as `undefined`, not the function value.

**24) What is the output?**

```
(function(){
  console.log(x);
  var x = 5;
})();
```

**Answer:** `undefined`
**Explanation:** `var x` hoisted within the IIFE function scope.

**25) What is the output?**

```
{
  const a = 1;
}
console.log(a);
```

**Answer:** ReferenceError
**Explanation:** `const` is block-scoped.

**26) Can you reassign a `const` object?**
**Answer:** You can't reassign the variable, but you can mutate properties.
**Explanation:** `const obj = {}`; `obj.x=1` works; `obj={}` doesn't.

**27) What's a closure?**
**Answer:** A function that "remembers" variables from its outer scope.
**Explanation:** It keeps access to those variables even after the outer function returns.

**28) What is the output?**

```
function make() {
  let n = 0;
  return () => ++n;
}
const inc = make();
console.log(inc(), inc(), inc());
```

**Answer:** `1 2 3`
**Explanation:** Closure keeps `n` alive.

**29) What's "global scope" in browsers?**
**Answer:** `window` (for non-module scripts).
**Explanation:** In ES modules, top-level bindings are module-scoped (not `window` properties).

**30) What's the "Temporal Dead Zone"?**
**Answer:** The time before a `let/const` is initialized where access throws.
**Explanation:** Prevents use-before-declare bugs.

## 3) Functions, `this`, Arrow Functions (31–45)

**31) What does `this` refer to in a normal function (non-strict) called as `obj.fn()`?**
**Answer:** `obj`
**Explanation:** Method call sets `this` to the receiver.

**32) What is `this` in strict mode for a plain call `fn()`?**
**Answer:** `undefined`
**Explanation:** No automatic binding to global object.

**33) Arrow functions and `this`: what's special?**
**Answer:** Arrow functions do not bind `this`; they capture `this` from surrounding scope.
**Explanation:** Great for callbacks, not great for methods needing dynamic `this`.

**34) Output?**

```
const obj = {
  x: 10,
  f: () => console.log(this.x)
};
obj.f();
```

**Answer:** Usually `undefined` (in modules: `undefined`)
**Explanation:** Arrow `this` is not `obj`; it's outer `this`.

**35) Output?**

```
const obj = {
  x: 10,
  f() { console.log(this.x); }
};
obj.f();
```

**Answer:** `10`
**Explanation:** Normal method call binds `this` to `obj`.

**36) What do `call`, `apply`, `bind` do?**
**Answer:**

- `call(thisArg, ...args)` invoke now
- `apply(thisArg, argsArray)` invoke now
- `bind(thisArg, ...args)` return new function with bound `this` (and optionally args)
  **Explanation:** `bind` doesn't run immediately.

**37) What is the output?**

```
function f(a,b){ return a+b; }
console.log(f.call(null, 2, 3));
```

**Answer:** `5`
**Explanation:** `this` not used; args passed directly.

**38) What is default parameter evaluation time?**
**Answer:** At call time.
**Explanation:** Defaults can reference earlier parameters: `function(a, b=a+1){}`.

**39) What's the difference between rest and arguments?**
**Answer:** Rest `(...args)` is a real array; `arguments` is array-like, not in arrow functions.
**Explanation:** Prefer rest.

**40) What is the output?**

```
function f(){ return arguments.length; }
console.log(f(1,2,3));
```

**Answer:** 3
**Explanation:** `arguments` counts passed args.

**41) What is "IIFE"?**
**Answer:** Immediately Invoked Function Expression.
**Explanation:** `(function(){ ... })()` used to create private scope.

**42) What is function "arity"?**
**Answer:** `fn.length` = number of declared parameters (not counting rest/default after first default).
**Explanation:** Useful but can be misleading with defaults.

**43) What is the output?**

```
function f(a,b=1,c){ }
console.log(f.length);
```

**Answer:** 1
**Explanation:** Length stops counting at first default parameter.

**44) What's the difference: named vs anonymous function expression?**
**Answer:** Named expressions help stack traces and recursion.
**Explanation:** `const f = function g(){}` — `g` is internal name.

**45) When should you avoid arrow functions?**
**Answer:** When you need dynamic `this`, `arguments`, or as constructors.
**Explanation:** Arrow functions can't be used with `new`.

# 4) Objects, Prototypes, Classes (46–60)

**46) How does prototypal inheritance work?**
**Answer:** Objects delegate property lookup to their prototype chain.
**Explanation:** If not found on object, JS checks `obj.__proto__`, etc.

**47) What does `Object.create(proto)` do?**
**Answer:** Creates a new object whose prototype is `proto`.
**Explanation:** No constructor execution; pure prototype setup.

**48) What is the output?**

```
const a = {x:1};
const b = Object.create(a);
console.log(b.x);
```

**Answer:** 1
**Explanation:** `x` is found on prototype `a`.

**49) Difference between `in` and `hasOwnProperty`?**
**Answer:** `in` checks own + prototype properties; `hasOwnProperty` checks only own.
**Explanation:** Important when iterating.

**50) What is `Object.freeze`?**
**Answer:** Prevents adding/removing/changing properties (shallow).
**Explanation:** Nested objects can still be mutated unless also frozen.

**51) What is the output?**

```
const o = Object.freeze({a:1});
o.a = 2;
console.log(o.a);
```

**Answer:** 1
**Explanation:** Assignment fails (silent in non-strict; TypeError in strict).

**52) What's the difference: `Object.seal` vs `freeze`?**
**Answer:** `seal` prevents add/remove but allows changing existing writable props; `freeze` prevents changes too.
**Explanation:** Both are shallow.

**53) What are property descriptors?**
**Answer:** Metadata like `writable`, `enumerable`, `configurable`, `value`, `get/set`.
**Explanation:** Controlled via `Object.defineProperty`.

**54) What does `new` do (high-level)?**
**Answer:**

1. Creates new object
2. Sets prototype to constructor's `prototype`
3. Binds `this` inside constructor
4. Returns object (unless constructor returns another object)
   **Explanation:** Core of constructor behavior.

**55) What's a class in JS (really)?**
**Answer:** Syntactic sugar over prototypes.
**Explanation:** Methods go on `ClassName.prototype`.

**56) What is the output?**

```
class A { m(){ return 1; } }
const a = new A();
console.log(a.m());
```

**Answer:** 1
**Explanation:** Standard class method.

**57) What is `static` in classes?**
**Answer:** A method/property on the class itself, not instances.
**Explanation:** `A.staticMethod()` not `a.staticMethod()`.

**58) What's the difference: `__proto__` vs `prototype`?**
**Answer:**

- `obj.__proto__` (or `Object.getPrototypeOf(obj)`) = object's prototype
- `Fn.prototype` = prototype used for instances created via `new Fn()`
  **Explanation:** People confuse these a lot.

**59) What does `instanceof` check?**
**Answer:** Whether a constructor's `prototype` is in the object's prototype chain.
**Explanation:** Can be fooled if prototypes change or across realms/iframes.

**60) How do you copy objects safely?**
**Answer:** Depends:

- Shallow: `{...obj}` / `Object.assign`

- Deep: `structuredClone(obj)` (best) or careful custom recursion
  **Explanation:** JSON stringify loses functions, `undefined`, symbols, dates, maps, etc.

# 5) Arrays, Iteration, Functional Methods (61–70)

**61) Difference between `map` and `forEach`?**
**Answer:** `map` returns a new array of transformed values; `forEach` returns `undefined`.
**Explanation:** Use `map` for transformation, `forEach` for side effects.

**62) What does `filter` do?**
**Answer:** Returns new array of items where callback returns truthy.
**Explanation:** Doesn't mutate original.

**63) What does `reduce` do?**
**Answer:** Reduces array to single value via accumulator.
**Explanation:** Great for sums, grouping, building objects.

**64) Output?**

```
console.log([1,2,3].reduce((a,b)=>a+b, 0));
```

**Answer:** `6`
**Explanation:** Accumulator starts at `0`.

**65) Difference between `slice` and `splice`?**
**Answer:** `slice` returns a copy portion (non-mutating). `splice` mutates (remove/insert).
**Explanation:** Interview classic.

**66) Output?**

```
const a=[1,2,3];
a.splice(1,1,9);
console.log(a);
```

**Answer:** `[1,9,3]`
**Explanation:** Removed index 1, inserted 9.

**67) What does `Array.from` do?**
**Answer:** Creates array from iterable/array-like, optionally mapping.
**Explanation:** Useful for NodeLists: `Array.from(document.querySelectorAll(...))`.

**68) What is `for...of` used for?**
**Answer:** Iterating values of iterables (arrays, strings, maps, sets).
**Explanation:** Unlike `for...in` which iterates keys (and can include inherited keys).

**69) Output?**

```
const arr=[10,20];
for (const i in arr) console.log(i);
```

**Answer:** `0` then `1`
**Explanation:** `for...in` iterates indices (as strings).

**70) What's time complexity of `push` vs `unshift` on arrays?**
**Answer:** `push` is usually O(1) amortized; `unshift` is O(n).
**Explanation:** `unshift` shifts all indexes.

# 6) Async JS, Event Loop, Promises (71–90)

**71) What is the event loop?**
**Answer:** JS runs single-threaded; the event loop schedules tasks/microtasks to run when call stack is empty.
**Explanation:** Explains why async callbacks run later.

**72) Difference: microtask vs macrotask?**
**Answer:** Microtasks (Promise callbacks, queueMicrotask) run before the next macrotask (setTimeout, I/O, UI events).
**Explanation:** Microtasks drain first.

**73) Output?**

```
console.log("A");
setTimeout(()=>console.log("B"),0);
Promise.resolve().then(()=>console.log("C"));
console.log("D");
```

**Answer:** `A D C B`
**Explanation:** Promise then = microtask; setTimeout = macrotask.

**74) What does `async` do to a function?**
**Answer:** Makes it return a Promise automatically.
**Explanation:** Return value becomes resolved value; thrown error becomes rejection.

**75) What does `await` do?**
**Answer:** Pauses async function until Promise settles; resumes with resolved value or throws on rejection.
**Explanation:** Only valid inside `async` functions (or top-level in modules depending environment).

**76) How do you handle errors with async/await?**
**Answer:** `try/catch` around `await`, or handle `.catch` on returned Promise.
**Explanation:** Rejections become thrown exceptions.

**77) What is `Promise.all` behavior?**
**Answer:** Resolves when all resolve; rejects immediately on first rejection.
**Explanation:** Result order matches input order.

**78) `Promise.allSettled` vs `Promise.all`?**
**Answer:** `allSettled` waits for all and returns statuses; never rejects because of input rejections.
**Explanation:** Good when you want results even if some fail.

**79) What does `Promise.race` do?**
**Answer:** Settles with the first Promise to settle (resolve or reject).
**Explanation:** Useful for timeouts.

**80) What does `Promise.any` do?**
**Answer:** Resolves with first fulfilled Promise; rejects only if all reject (AggregateError).
**Explanation:** “Any success wins.”

**81) Why might `await` in a loop be slow?**
**Answer:** It runs sequentially.
**Explanation:** Use `Promise.all` for parallel if safe.

**82) Convert callback to Promise (concept)?**
**Answer:** Wrap in `new Promise((res, rej)=>...)` and resolve/reject inside callback.
**Explanation:** “Promisify” pattern.

**83) What is a “thenable”?**
**Answer:** Any object with a `.then` method treated like a Promise in resolution.
**Explanation:** Promises “assimilate” thenables.

**84) Output?**

```
(async () => {
  return 5;
})().then(console.log);
```

**Answer:** `5`
**Explanation:** Async returns resolved Promise with value 5.

**85) Output?**

```
(async () => {
  throw new Error("x");
})().catch(e => console.log("caught"));
```

**Answer:** `caught`
**Explanation:** Throw inside async = rejected Promise.

**86) What's the difference between `.then(fn)` and `.then(() => fn())`?**
**Answer:** If `fn` needs arguments or correct binding, you might need wrapper.
**Explanation:** Passing `fn` directly passes resolved value as argument.

**87) What is "async starvation" risk?**
**Answer:** Too many microtasks can delay rendering/macrotasks.
**Explanation:** Microtasks drain fully before next macrotask.

**88) What is `AbortController` used for?**
**Answer:** Cancelling fetch/async operations via signals.
**Explanation:** Prevents wasted work and race conditions.

**89) What is a race condition in async JS?**
**Answer:** Outcome depends on timing/order of async operations.
**Explanation:** Fix with cancellation, sequencing, or single source of truth.

**90) What does `fetch` return?**
**Answer:** A Promise that resolves to a Response (even for HTTP errors like 404).
**Explanation:** You must check `response.ok` yourself.

# 7) DOM, Browser, Modules, Misc (91–100)

**91) What is event delegation?**
**Answer:** Attach one listener to a parent and handle events from children via bubbling.
**Explanation:** Efficient for many dynamic elements.

**92) `event.target` vs `event.currentTarget`?**
**Answer:** `target` = actual clicked element; `currentTarget` = element with the listener.
**Explanation:** Crucial for delegation.

**93) What is the difference between `localStorage` and `sessionStorage`?**
**Answer:** `localStorage` persists; `sessionStorage` lasts per tab session.
**Explanation:** Both store strings, synchronous APIs.

**94) What is CORS (high level)?**
**Answer:** Browser security policy controlling cross-origin requests.
**Explanation:** Server must allow via headers; client can't "disable" it safely.

**95) What is an ES module?**
**Answer:** JS file using `import/export`, with its own scope and strict mode by default.
**Explanation:** In browsers: `<script type="module">`.

**96) Difference between default and named export?**
**Answer:** Default: `export default ...` imported without braces; named uses braces.
**Explanation:** `import x from` vs `import {x} from`.

**97) What is tree shaking?**
**Answer:** Bundler removes unused exports from final build (when code is statically analyzable).
**Explanation:** Helps reduce bundle size.

**98) What is the difference between `defer` and `async` on script tags?**
**Answer:**

- `defer`: downloads in parallel, executes in order after HTML parse
- `async`: downloads in parallel, executes ASAP (order not guaranteed)
  **Explanation:** `defer` is safer for dependencies.

**99) What is a memory leak in JS?**
**Answer:** Objects not released because references remain (e.g., unremoved event listeners, global caches).
**Explanation:** GC can't free referenced objects.

**100) What is "strict mode" and why use it?**
**Answer:** `"use strict"` enforces safer rules (no accidental globals, stricter `this`, etc.).
**Explanation:** ES modules are strict by default.