

Junior Developer Interview Questions and Answers

Junior (0-2 years) | All technologies

Interview questions and answers for junior software developers with 0-2 years of experience, covering core language features, databases, frameworks and everyday debugging.

39 questions Read online: <https://www.freedevtemplates.com/interview-questions/level/junior>

JavaScript

1. What is a closure in JavaScript and why is it useful?

Level: junior JavaScript

A closure is a function that keeps access to the variables of the scope where it was created, even after that outer function has finished running.

Every function in JavaScript carries a reference to the lexical environment it was defined in. When an inner function is returned or passed elsewhere, that environment stays alive for as long as the function can be called, so the inner function can still read and update the outer variables.

Closures give you private state without classes, power patterns such as memoization, debounce and once-only functions, and make callbacks and React hooks work. The trade-off is memory: anything captured by a closure cannot be garbage collected while the closure is reachable.

Key points

A closure = a function plus its surrounding scope

Used for private state, factories, callbacks and memoization

Captured variables stay in memory while the closure is reachable

```
function createCounter() {  
  let count = 0; // private, not reachable from outside  
  return function () {  
    count += 1;  
    return count;  
  };  
}  
  
const next = createCounter();  
next(); // 1  
next(); // 2
```

2. How does the this keyword work in JavaScript?

Level: junior JavaScript

this is decided by how a function is called, not where it is written: as a method it is the object before the dot, with new it is the new object, and arrow functions inherit it from the enclosing scope.

For a normal function there are four rules, applied in this order: new Foo() makes this the newly created object; call, apply or bind set it explicitly; obj.method() makes this the object before the dot; otherwise it is undefined in strict mode (or the global object in sloppy mode).

Arrow functions do not have their own this. They use the this of the scope they were defined in, which is why they are handy for callbacks inside methods. The classic bug is passing obj.method as a callback: the call site changes, so this is lost.

Key points

this depends on the call site

call, apply and bind set this explicitly

Arrow functions capture this lexically and cannot be rebound

```
const user = {
  name: "Asha",
  regular() { return this.name; },
  arrow: () => typeof this, // "this" is NOT user here
};

user.regular(); // "Asha"
const detached = user.regular;
detached(); // undefined (or TypeError) - lost context
detached.call({ name: "Ravi" }) // "Ravi"
user.regular.bind(user)(); // "Asha"
```

3. What is the difference between promises and async/await in JavaScript?

Level: junior JavaScript

async/await is syntax built on top of promises that lets you write asynchronous code that reads like synchronous code; under the hood every async function returns a promise.

A promise represents a value that will be available later and is in one of three states: pending, fulfilled or rejected. You consume it with `.then()` and `.catch()`. `async/await` lets you pause an async function with `await` until a promise settles, and handle errors with ordinary `try/catch`.

They are equivalent in power, but `await` makes sequential logic and error handling much easier to read. A common mistake is awaiting independent calls one after another. Use `Promise.all` to run them in parallel.

Key points

async functions always return a promise

await only pauses the async function, not the whole thread

Use `Promise.all` for independent work

```
// promises
fetch("/api/user").then(r => r.json()).then(showUser).catch(handleError);

// async/await
async function load() {
  try {
    const res = await fetch("/api/user");
    showUser(await res.json());
  } catch (err) {
    handleError(err);
  }
}

// run independent calls in parallel
const [user, posts] = await Promise.all([getUser(), getPosts()]);
```

4. What is event delegation in JavaScript?

Level: junior JavaScript

Event delegation means attaching one listener to a parent element and using event bubbling to handle events from its children, instead of adding a listener to each child.

When you click a child element, the event bubbles up through its ancestors. A single listener on the parent can inspect `event.target` (or use `event.target.closest(selector)`) to find which child was clicked and react accordingly.

This uses less memory for long lists and also works for elements added later, because the listener lives on the

parent. It does not work for events that do not bubble, such as focus and blur (use `focusin` and `focusout` instead).

```
document.querySelector("#todo-list").addEventListener("click", (event) => {
  const item = event.target.closest("li[data-id]");
  if (!item) return;
  removeTodo(item.dataset.id);
});
```

React

5. What are React hooks and what are the rules of hooks?

Level: junior React

Hooks are functions such as `useState` and `useEffect` that let function components use state, lifecycle behaviour and other React features; they must be called at the top level of a component or custom hook, never inside loops, conditions or nested functions.

Before hooks, state and lifecycle methods required class components. Hooks let you do the same in function components and, importantly, let you extract and reuse stateful logic as custom hooks without wrapper components or render props.

React tracks hooks by the order in which they are called during each render. If a hook is skipped in one render because of an if statement, every hook after it would read the wrong state. That is why the two rules exist: only call hooks at the top level, and only call them from React function components or other custom hooks (whose names start with `use`). The `eslint-plugin-react-hooks` package enforces both.

Key points

Call hooks at the top level only

Call hooks only from components or custom hooks

React relies on call order to match state to hooks

```
function useWindowWidth() { // custom hook
  const [width, setWidth] = useState(window.innerWidth);
  useEffect(() => {
    const onResize = () => setWidth(window.innerWidth);
    window.addEventListener("resize", onResize);
    return () => window.removeEventListener("resize", onResize);
  }, []);
  return width;
}
```

6. How does `useEffect` work in React and what is the dependency array?

Level: junior React

`useEffect` runs a function after the component renders; the dependency array tells React when to re-run it: with no array it runs after every render, with an empty array only after the first, and with values only when those values change.

Effects are for synchronising your component with something outside React, such as subscriptions, timers, network requests or the document title. The function you return from an effect is the cleanup, and React runs it before the next run of the effect and when the component unmounts.

Every value from the component that the effect uses should be listed in the dependency array, otherwise the effect works with stale values. In development with Strict Mode, React deliberately mounts, unmounts and re-mounts components once to help you find missing cleanup. Do not use effects for things you can calculate during render or handle in an event handler.

Key points

No array: every render. `[]`: once. `[a, b]`: when a or b change

Return a function to clean up

Do not use effects for derived data or user-triggered actions

```
useEffect(() => {
  const controller = new AbortController();
  fetch("/api/users/" + userId, { signal: controller.signal })
    .then(r => r.json())
    .then(setUser)
    .catch(err => { if (err.name !== "AbortError") setError(err); });

  return () => controller.abort(); // cleanup when userId changes or unmounts
}, [userId]);
```

7. Why do we need keys in React lists?

Level: junior React

Keys give each list item a stable identity so React can match items between renders and correctly add, remove or reorder them without recreating or mixing up their state.

When React diffs two lists it needs to know which new item corresponds to which old one. Without keys it falls back to matching by position, so inserting an item at the top makes every item look changed, and local state such as an input's text can end up attached to the wrong row.

A key must be unique among siblings and stable across renders. Use an ID from your data. Using the array index is acceptable only for static lists that are never reordered, filtered or inserted into. Never use `Math.random()`, since a new key on every render makes React remount every item.

```
{todos.map(todo => (
  <TodoItem key={todo.id} todo={todo} /> // stable ID, not the index
))}
```

8. What is the difference between controlled and uncontrolled components in React?

Level: junior React

A controlled component keeps the input's value in React state and updates it through `onChange`, while an uncontrolled component lets the DOM hold the value and you read it with a `ref` when needed.

Controlled inputs make React the single source of truth: the value comes from state and every keystroke calls a handler that updates state. This makes validation, formatting and conditional disabling easy. The cost is a re-render on each keystroke.

Uncontrolled inputs use `defaultValue` and `ref`, so the browser manages the value and you read it on submit. They are simpler for basic forms and file inputs (which are always uncontrolled). Form libraries often use uncontrolled inputs under the hood to avoid re-rendering the whole form while typing.

```
// controlled
const [email, setEmail] = useState("");
<input value={email} onChange={e => setEmail(e.target.value)} />

// uncontrolled
const ref = useRef(null);
<input defaultValue="" ref={ref} />
const onSubmit = () => console.log(ref.current.value);
```

9. What is prop drilling and how do you avoid it in React?

Level: junior React

Prop drilling is passing props through several layers of components that do not use them just to reach a deeply nested child; you avoid it with composition, the Context API or a state management library.

First try composition: pass the child component itself, or pass children, so intermediate components never need to know about the data. This is often enough and keeps components easy to test.

If many distant components need the same value, such as the current user or theme, use the Context API. Context re-renders every consumer when its value changes, so keep contexts small and focused, and split frequently changing values from stable ones. For complex client state, a store such as Redux Toolkit or Zustand lets components subscribe to just the slice they need.

```
const ThemeContext = createContext("light");

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Layout /> { /* no theme prop needed */}
    </ThemeContext.Provider>
  );
}

function Button() {
  const theme = useContext(ThemeContext);
  return <button className={theme}>Save</button>;
}
```

HTML & CSS

10. What is the difference between relative, absolute, fixed and sticky positioning in CSS?

Level: junior HTML & CSS

Relative offsets an element from its normal place; absolute positions it against the nearest positioned ancestor; fixed is against the viewport; sticky sits in normal flow until a scroll threshold, then sticks.

position: relative with top/left moves the element but the original space is kept. position: absolute takes it out of flow; the containing block is the nearest ancestor with position other than static (or the initial containing block). That is why a modal overlay needs position: relative on its parent.

fixed stays put when the page scrolls, which is how sticky headers used to be built. sticky is a hybrid: the element scrolls with its container until top: 0 (or similar) is reached, then it behaves like fixed within that container. sticky fails if any ancestor has overflow: hidden.

Key points

absolute is relative to the nearest positioned ancestor

fixed is relative to the viewport

sticky needs a threshold (top/left) and no overflow-hidden ancestors

11. What is the difference between Flexbox and CSS Grid?

Level: junior HTML & CSS

Flexbox is one-dimensional (a row or a column). CSS Grid is two-dimensional (rows and columns at once). Use Flexbox for alignment along a line; use Grid when the layout is a real 2D structure.

A nav bar, a cluster of buttons, or a card header with a title on the left and actions on the right is a Flexbox problem. A page with a sidebar, a main column and a footer, or a photo gallery with consistent tracks, is a Grid problem. They combine well: a Grid page whose cards use Flexbox internally.

Grid lets you name areas (grid-template-areas) and place items on explicit lines. Flexbox shines at distributing leftover space (justify-content, flex-grow) and wrapping. Neither replaces the other; interviews often ask you to

pick one and say why.

```
.page {
display: grid;
grid-template-columns: 240px 1fr;
grid-template-rows: auto 1fr auto;
min-height: 100vh;
}

.toolbar {
display: flex;
justify-content: space-between;
align-items: center;
gap: 1rem;
}
```

12. How does CSS specificity work?

Level: junior HTML & CSS

Specificity is the score the browser uses to decide which rule wins: inline styles beat IDs, IDs beat classes/attributes/pseudo-classes, and those beat elements. When scores tie, the last rule in the source wins.

Count a tuple (inline, IDs, classes, elements). `#nav .item` is (0, 1, 1, 0). `div.card p` is (0, 0, 1, 2). `!important` jumps the queue and should be a last resort; too many of them mean the specificity war is already lost.

A more maintainable approach is low-specificity utilities or a method such as BEM, plus CSS layers (`@layer`) so you can control order without stacking IDs. Attribute selectors and pseudo-classes (`.btn:hover`) count as classes. The universal selector `*` and the `:where()` pseudo-class add zero specificity.

13. What is the difference between em, rem, %, vh and vw in CSS?

Level: junior HTML & CSS

em is relative to the parent font size, rem is relative to the root font size, % is relative to the parent's relevant size, and vh/vw are 1% of the viewport height or width.

rem is the default choice for type and spacing because it stays predictable when components nest. em compounds: a 1.2em padding inside a 1.2em heading is larger than you think. % on width is the parent's width; % on height only works if the parent has an explicit height.

100vh is the full viewport height, which on mobile can include the area hidden by the browser chrome and cause jumpiness. Newer units such as svh, lvh and dvh address that. For fluid type, `clamp(1rem, 2vw, 2rem)` is a common pattern.

14. What is the difference between cookies, localStorage and sessionStorage?

Level: junior HTML & CSS

Cookies are sent to the server with every matching request and can expire; localStorage persists until cleared and stays in the browser; sessionStorage lasts for the current tab only.

Use cookies for authentication when the server must see the value (prefer `HttpOnly`, `Secure` and `SameSite`). Use `localStorage` for non-sensitive UI preferences that should survive a restart. Use `sessionStorage` for a wizard or draft that should die when the tab closes.

Both storage APIs are readable by JavaScript, so they are the wrong place for access tokens if you have XSS. Cookies set without `HttpOnly` have the same problem. All three are origin-scoped. `localStorage` has a typical quota of about 5 MB; cookies should stay tiny because they travel on every request.

TypeScript

15. What is the difference between any, unknown and never in TypeScript?

Level: junior TypeScript

any turns off checking. unknown is a type-safe any: you must narrow it before use. never means a value cannot exist, used for exhaustive checks and functions that always throw.

A value typed any can be passed anywhere and you can read any property on it, so a typo compiles. unknown is what you give to untrusted input (JSON.parse, a network payload) until you prove its shape with typeof, in, or a type guard.

never is the empty set. A function with return type never does not return (throw or infinite loop). In a switch, assigning the leftover value to const _exhaustive: never = value makes the compiler fail when a new union member is added and not handled.

```
function handle(x: unknown) {
  if (typeof x === "string") return x.toUpperCase();
  throw new Error("expected string");
}

function fail(message: string): never {
  throw new Error(message);
}
```

16. What are generics in TypeScript?

Level: junior TypeScript

Generics are type parameters: they let a function, class or type work with many types while keeping the relationship between inputs and outputs, instead of falling back to any.

identity<T>(value: T): T returns the same type it received. Without T you would write any and lose that link. Constraints (T extends { id: string }) describe the minimum shape. Several parameters (K, V) model maps and API wrappers.

Common interview follow-ups: default parameters (T = string), keyof T for safe property access, and why Array<T> and T[] are the same. Prefer inference (wrap(user)) over forcing wrap<User>(user) at every call site.

```
function first<T>(items: T[]): T | undefined {
  return items[0];
}

first(["a", "b"]); // string | undefined
first<number>([1, 2, 3]); // number | undefined
```

17. What are union and intersection types in TypeScript?

Level: junior TypeScript

A union (A | B) is a value that is one of the members. An intersection (A & B) is a value that must satisfy both types at once.

Unions model "this or that": a result that is { ok: true, data: T } | { ok: false, error: string }. Narrow with a discriminant field (ok) before you read data. Intersections model "this and that": User & { role: "admin" } has every User field plus role.

A union of objects is not an object with optional fields. A | B only lets you access properties common to both until you narrow. That is the usual bug when someone writes user.role on a User | Admin without checking.

18. What are utility types in TypeScript like Partial, Pick and Omit?

Level: junior TypeScript

Utility types are built-in helpers that transform an existing type: Partial makes every field optional, Required the opposite, Pick keeps a set of keys, Omit removes them, and Record builds a key/value map.

Partial<User> is useful for a PATCH body. Pick<User, "id" | "name"> is a list-row type. Omit<User, "password"> is what you send to the client. Record<Id, User> is a dictionary. ReturnType<typeof fn> and Parameters<typeof fn> pull types from functions so you do not repeat them.

You can write the same things with mapped types; the utilities are the readable version. Interviews sometimes ask you to implement Partial yourself: { [K in keyof T]?: T[K] }.

```
type User = { id: string; name: string; password: string };
type PublicUser = Omit<User, "password">;
type UserPatch = Partial<Pick<User, "name" | "password">>;
```

SQL

19. How do you find the second highest salary in SQL?

Level: junior SQL

Use a subquery that takes the MAX salary below the overall MAX, or use DENSE_RANK() and pick rank 2, which also handles ties and generalises to the Nth highest.

The subquery version works on every database. The window function version is the one interviewers like best, because changing 2 to N gives you the Nth highest salary, and DENSE_RANK correctly treats equal salaries as the same rank. Use DISTINCT so tied rows return one value.

Another common answer is ORDER BY salary DESC LIMIT 1 OFFSET 1 (MySQL, PostgreSQL) or OFFSET 1 ROWS FETCH NEXT 1 ROWS ONLY (SQL Server), but you need SELECT DISTINCT salary first or ties break it. If there is no second salary, wrap the query so it returns NULL rather than an empty result when the question asks for that.

Key points

Subquery with MAX is the simplest answer

DENSE_RANK handles ties and any N

ROW_NUMBER would be wrong when salaries tie

```
-- works everywhere
SELECT MAX(salary) AS second_highest
FROM employees
WHERE salary < (SELECT MAX(salary) FROM employees);

-- Nth highest with window function (here N = 2)
SELECT DISTINCT salary
FROM (
  SELECT salary, DENSE_RANK() OVER (ORDER BY salary DESC) AS rnk
  FROM employees
) ranked
WHERE rnk = 2;
```

20. How do you find and delete duplicate rows in SQL?

Level: junior SQL

Find duplicates with GROUP BY and HAVING COUNT(*) > 1, then delete them by numbering rows within each duplicate group with ROW_NUMBER() and removing every row after the first.

First decide what makes two rows duplicates (for example the same email) and which row you want to keep

(usually the oldest, so the lowest id). Always run the matching SELECT before the DELETE, and take a backup or work inside a transaction.

ROW_NUMBER() OVER (PARTITION BY email ORDER BY id) numbers rows inside each group of identical emails, so any row with a number above 1 is a duplicate. In MySQL older than 8.0 window functions are unavailable, and you cannot select from the same table in a DELETE subquery directly, so you wrap the subquery in a derived table or self-join instead. Afterwards, add a unique constraint so the duplicates cannot come back.

```
-- 1. find
SELECT email, COUNT(*) AS copies
FROM users
GROUP BY email
HAVING COUNT(*) > 1;

-- 2. delete, keeping the lowest id (PostgreSQL / SQL Server / MySQL 8+)
WITH ranked AS (
SELECT id, ROW_NUMBER() OVER (PARTITION BY email ORDER BY id) AS rn
FROM users
)
DELETE FROM users WHERE id IN (SELECT id FROM ranked WHERE rn > 1);

-- 3. prevent it happening again
ALTER TABLE users ADD CONSTRAINT uq_users_email UNIQUE (email);
```

21. What is the difference between UNION and UNION ALL in SQL?

Level: junior SQL

UNION combines two result sets and removes duplicate rows, while UNION ALL combines them and keeps every row, which makes it faster.

UNION has to sort or hash the combined result to find duplicates, which costs time and memory. If you know the two queries cannot return the same row, or you want the duplicates, use UNION ALL. Both require the same number of columns with compatible types, and the column names come from the first query.

```
SELECT email FROM newsletter_subscribers
UNION -- one row per distinct email
SELECT email FROM customers;

SELECT email FROM newsletter_subscribers
UNION ALL -- keeps duplicates, faster
SELECT email FROM customers;
```

22. What is database normalization and what are the normal forms?

Level: junior SQL

Normalization is organising tables to reduce redundancy and update anomalies: 1NF means atomic values, 2NF removes partial dependencies on part of a composite key, and 3NF removes dependencies between non-key columns.

In 1NF every column holds a single value (no comma-separated lists) and each row is unique. In 2NF the table is in 1NF and every non-key column depends on the whole primary key, which matters for composite keys. In 3NF the table is in 2NF and non-key columns depend only on the key, not on other non-key columns (a city and zip_code pair stored with an order is an example of what to move out).

Normalisation avoids storing the same fact in many places, so updates and deletes cannot leave inconsistent data. The trade-off is more joins. Reporting databases and read-heavy systems often denormalise on purpose, accepting some redundancy for faster reads.

Key points

1NF: atomic values. 2NF: no partial dependency. 3NF: no transitive dependency
Goal: remove insert, update and delete anomalies
Denormalise deliberately for read performance

C#

23. What is the difference between const and readonly in C#?

Level: junior C#

A const is a compile-time constant that must be assigned where it is declared, while a readonly field can be assigned at declaration or in the constructor, so its value can be decided at runtime.

const values are baked into the calling code at compile time and are implicitly static. They can only be primitive types, strings or null. If you change a const in a library, every project using it must be recompiled to see the new value.

readonly fields are normal fields that cannot be reassigned after the constructor finishes. They can hold any type, can differ per instance, and are read at runtime. Note that readonly makes the reference unchangeable, not the object it points to. For compile-time values that will never change (such as the number of days in a week) use const; otherwise prefer readonly.

```
public class Config
{
    public const int MaxRetries = 3; // compile-time
    public readonly DateTime CreatedAt; // runtime, per instance

    public Config() { CreatedAt = DateTime.UtcNow; }
}
```

24. What is the difference between ref and out parameters in C#?

Level: junior C#

Both pass an argument by reference, but ref requires the variable to be initialised before the call, while out requires the method to assign a value before it returns.

Use ref when the method both reads and modifies the caller's variable. Use out when the method only produces a value, such as the bool TryParse(string, out int result) pattern, which lets a method return a success flag and a value together. A third modifier, in, passes by reference as read-only, useful to avoid copying large structs.

In modern C# you can often return a tuple instead ((bool ok, int value)), which is clearer than out parameters.

```
if (int.TryParse(input, out int number))
{
    Console.WriteLine(number * 2);
}

static void AddTen(ref int x) => x += 10;
int n = 5;
AddTen(ref n); // n is now 15
```

25. What is the difference between an abstract class and an interface in C#?

Level: junior C#

An abstract class can hold state, constructors and partial implementation but a class can inherit only one, while an interface defines a contract that a class can implement many of.

Use an abstract class when related classes share fields, constructor logic or common behaviour, and you want to

force subclasses to fill in the rest. A class can extend only one base class, so this is a strong coupling.

Use an interface to describe a capability (IDisposable, IComparable, IEmailSender) that unrelated types can implement, and to make code testable and replaceable through dependency injection. Since C# 8 interfaces may include default method implementations, but they still cannot hold instance fields. A common guideline: start with an interface for the contract, and add an abstract base class only if there is real shared code.

Key points

Abstract class: shared state and code, single inheritance

Interface: contract, multiple implementation

Interfaces make dependency injection and testing easier

```
public interface INotifier { void Send(string message); }

public abstract class NotifierBase : INotifier
{
    protected readonly ILogger Logger;
    protected NotifierBase(ILogger logger) => Logger = logger;
    public abstract void Send(string message);
}
```

26. What is boxing and unboxing in C# and why does it affect performance?

Level: junior C#

Boxing converts a value type to object by allocating it on the heap, and unboxing converts it back; both add allocations and casts, so avoid them in hot code paths.

Assigning an int to an object variable boxes it: the runtime allocates a new object on the heap and copies the value in. Casting back with (int)obj is unboxing, which checks the type and copies the value out. Boxing in a loop creates garbage that the collector has to clean.

Generics were introduced to avoid this: List<int> stores ints directly, whereas the old ArrayList boxed every item. Watch for hidden boxing when a struct is used as an interface type, or when you pass a value type to a method taking object, such as older string.Format overloads.

```
int i = 42;
object boxed = i; // boxing: heap allocation
int j = (int)boxed; // unboxing: type check + copy

var list = new List<int>(); // no boxing
list.Add(i);
```

27. What is LINQ in C# and what is deferred execution?

Level: junior C#

LINQ (Language Integrated Query) lets you query collections, databases and XML with a consistent syntax, and deferred execution means a query does not run when you define it but when you enumerate the results.

Methods such as Where, Select and OrderBy only build up a description of the query. The work happens when you iterate with foreach or call an operator that forces evaluation, such as ToList(), ToArray(), Count() or First().

This is efficient, but it has traps: if the source changes between defining and enumerating the query, you see the new data, and enumerating the same query twice runs it twice (a database query would hit the database twice). Call ToList() when you want a snapshot or need to reuse the results.

```
var numbers = new List<int> { 1, 2, 3 };
var evens = numbers.Where(n => n % 2 == 0); // nothing runs yet

numbers.Add(4);
Console.WriteLine(string.Join(", ", evens)); // 2,4 (runs now)

var snapshot = numbers.Where(n => n > 1).ToList(); // runs immediately
```

Java

28. What is the difference between an abstract class and an interface in Java?

Level: junior Java

An abstract class can have instance fields, constructors and a mix of implemented and abstract methods but supports only single inheritance, while an interface defines a contract that a class can implement multiple times and, since Java 8, can include default methods.

Choose an abstract class when subclasses share state and base behaviour (for example a BaseRepository with a connection field). Choose an interface when you want to define a capability that unrelated classes can have, such as Comparable or Runnable, and when you want callers to depend on an abstraction.

Interface fields are implicitly public static final. Since Java 8 an interface can have default and static methods, and since Java 9 private methods, which lets library authors add methods without breaking existing implementations. A class can implement many interfaces but extend only one class.

```
interface Payment {
    void pay(double amount);
    default String currency() { return "USD"; }
}

abstract class BasePayment implements Payment {
    protected final String accountId;
    BasePayment(String accountId) { this.accountId = accountId; }
}
```

29. What is the difference between ArrayList and LinkedList in Java?

Level: junior Java

ArrayList is backed by a resizable array with $O(1)$ index access and $O(n)$ insertion in the middle, while LinkedList is a doubly linked list with $O(n)$ index access and $O(1)$ insertion once you hold the node; in practice ArrayList is faster for most workloads.

ArrayList stores elements contiguously, which is cache-friendly, so iteration and `get(i)` are very fast. Adding at the end is amortised $O(1)$; inserting or removing in the middle shifts the following elements.

LinkedList stores each element in a node with previous and next pointers, which costs extra memory and scatters nodes around the heap. Finding the element at position `i` requires walking the list. It only wins when you repeatedly add or remove at the ends or through an iterator. For queue and deque behaviour, ArrayDeque is usually a better choice than LinkedList.

Key points

ArrayList: $O(1)$ get, amortised $O(1)$ add at end

LinkedList: $O(n)$ get, extra memory per element

Default to ArrayList; use ArrayDeque for queues

30. What is the difference between checked and unchecked exceptions in Java?

Level: junior Java

Checked exceptions (subclasses of `Exception` other than `RuntimeException`) must be caught or declared with `throws`, while unchecked exceptions (`RuntimeException` and `Error`) do not have to be handled and usually signal programming bugs.

Checked exceptions such as `IOException` and `SQLException` represent failures the caller can reasonably be expected to handle or recover from, so the compiler forces you to deal with them. Unchecked exceptions such as `NullPointerException`, `IllegalArgumentException` and `IndexOutOfBoundsException` indicate bugs, and fixing the code is better than catching them. `Error` (for example `OutOfMemoryError`) signals serious JVM problems you should not catch.

Good practice: never swallow an exception silently, catch the most specific type, add context when you wrap it, and keep the original as the cause (new `ServiceException("msg", e)`). Many modern frameworks, including Spring, prefer unchecked exceptions because checked ones clutter method signatures.

31. What are the main features of Java 8: lambdas, streams and Optional?

Level: junior Java

Java 8 introduced lambda expressions, functional interfaces, the Stream API for declarative collection processing, `Optional` to represent missing values, default methods in interfaces, method references and the `java.time` date and time API.

A lambda is a short way to write an instance of a functional interface (an interface with one abstract method), such as `Comparator` or `Runnable`. The Stream API lets you describe a pipeline: a source, zero or more intermediate operations such as `filter` and `map` that are lazy, and a terminal operation such as `collect` that triggers the work.

Use `Optional` as a return type to make absence explicit instead of returning `null`, but not for fields or method parameters. Streams do not modify the source collection, and parallel streams should be used with care: they only help for large, CPU-bound, stateless operations.

Key points

Lambdas implement functional interfaces

Stream intermediate operations are lazy until a terminal operation runs

`Optional` is for return values, not fields or parameters

```
List<String> names = users.stream()
    .filter(u -> u.isActive())
    .map(User::getName)
    .sorted()
    .collect(Collectors.toList());

Optional<User> user = repository.findById(id);
String email = user.map(User::getEmail).orElse("unknown");
```

Python

32. What are `*args` and `**kwargs` in Python?

Level: junior Python

`*args` collects extra positional arguments into a tuple and `**kwargs` collects extra keyword arguments into a dictionary, which lets a function accept a variable number of arguments.

The names `args` and `kwargs` are conventions; the stars are what matter. They are useful for wrappers and decorators that must pass any arguments through to another function, and for APIs that accept optional settings.

The same stars work at the call site to unpack: `print(*numbers)` spreads a list into separate arguments and

`func(**options)` spreads a dictionary into keyword arguments. The order in a signature is: normal parameters, `*args`, keyword-only parameters, `**kwargs`.

```
def report(title, *args, **kwargs):
    print(title, args, kwargs)

report("Sales", 1, 2, region="EU", year=2026)
# Sales (1, 2) {'region': 'EU', 'year': 2026}

nums = [3, 4]
opts = {"sep": "-"}
print(*nums, **opts) # 3-4
```

33. Why should you not use a mutable default argument in Python?

Level: junior Python

Default argument values are evaluated once when the function is defined, so a default list or dictionary is shared between every call that does not pass its own, causing data to leak between calls.

This is one of the most common Python gotchas. The fix is to use `None` as the default and create a new object inside the function.

```
def add_item(item, items=[]): # bug: one shared list
    items.append(item)
    return items

add_item("a") # ['a']
add_item("b") # ['a', 'b'] <- surprise

def add_item(item, items=None): # correct
    if items is None:
        items = []
    items.append(item)
    return items
```

34. What is the difference between a shallow copy and a deep copy in Python?

Level: junior Python

A shallow copy creates a new container but reuses the same nested objects, while a deep copy recursively copies everything so the new object is fully independent.

With a shallow copy (`copy.copy()`, `list.copy()`, slicing `items[:]`), changing a nested list in the copy also changes it in the original, because both point to the same inner object. `copy.deepcopy()` duplicates the nested objects as well, handling circular references, but it is slower and uses more memory.

Plain assignment `b = a` does not copy at all: both names refer to the same object.

```
import copy

original = [[1, 2], [3, 4]]
shallow = copy.copy(original)
deep = copy.deepcopy(original)

original[0].append(99)
print(shallow[0]) # [1, 2, 99] - shared inner list
print(deep[0]) # [1, 2] - independent
```

35. What is the difference between @staticmethod and @classmethod in Python?

Level: junior Python

A classmethod receives the class as its first argument (cls) and can create or configure instances, while a staticmethod receives no automatic argument and is just a function placed inside the class namespace.

Use @classmethod for alternative constructors such as Date.from_string(...), and because it uses cls, it works correctly for subclasses. Use @staticmethod for a utility function that is related to the class but needs neither the instance nor the class. An ordinary method receives self, the instance.

```
class User:
    def __init__(self, name):
        self.name = name

    @classmethod
    def from_email(cls, email):
        return cls(email.split("@")[0])

    @staticmethod
    def is_valid_email(email):
        return "@" in email
```

Node.js

36. What is the difference between require and import in Node.js?

Level: junior Node.js

require is the CommonJS module system, which loads modules synchronously at runtime, while import is the ES module standard, which is statically analysed, asynchronous and supports top-level await and tree shaking.

CommonJS uses require() and module.exports; it is the default in .js files unless your package.json says otherwise. ES modules use import and export, enabled by "type": "module" in package.json or by using .mjs files. Imports are hoisted and resolved before the code runs, which allows tools to remove unused exports.

Differences to remember: ES modules require file extensions in relative imports, have no __dirname (use import.meta.url), and can only be loaded from CommonJS with dynamic import(). Dynamic import() works in both systems.

```
// CommonJS
const express = require("express");
module.exports = { createApp };

// ES modules
import express from "express";
export function createApp() {}
```

37. What is middleware in Express and how does it work?

Level: junior Node.js

Middleware are functions that receive the request, response and a next callback; they run in the order they are registered and can modify the request, end the response or pass control on with next().

Each middleware can do work such as logging, parsing JSON bodies, checking authentication or setting headers, and then either sends a response or calls next() to continue to the next function. Order matters: a body parser must be registered before the routes that read req.body.

Error-handling middleware has four parameters (err, req, res, next) and is registered last. In Express 4, an error

thrown inside an async handler is not caught automatically; you must catch it and call `next(err)` or use a small wrapper (Express 5 handles rejected promises for you).

```
app.use(express.json());

function requireAuth(req, res, next) {
  if (!req.headers.authorization) return res.status(401).json({ error: "Unauthorized" });
  next();
}

app.get("/profile", requireAuth, (req, res) => res.json({ ok: true }));

app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).json({ error: "Something went wrong" });
});
```

System Design

38. What is the difference between horizontal and vertical scaling?

Level: junior System Design

Vertical scaling means giving one machine more CPU, memory or disk, while horizontal scaling means adding more machines and spreading the load across them.

Vertical scaling is the simplest option and needs no code changes, but it has a hard ceiling, usually costs more per unit of capacity at the top end, and leaves a single point of failure. Horizontal scaling has almost no ceiling and improves availability, but the application must be designed for it: stateless servers, shared sessions and caches, and a load balancer in front.

Databases are the hard part. Stateless web servers scale out easily; a relational database is usually scaled up first, then read replicas are added, and finally the data is partitioned (sharded) when one primary can no longer handle the writes.

Key points

Vertical: bigger machine. Horizontal: more machines

Horizontal scaling needs stateless services

Databases are scaled up, then replicated, then sharded

39. What is a load balancer and what are the common load balancing algorithms?

Level: junior System Design

A load balancer distributes incoming requests across several servers to improve capacity and availability, using algorithms such as round robin, least connections or consistent hashing.

Round robin sends each request to the next server in turn and works when requests cost about the same. Least connections sends the request to the server with the fewest active connections, which is better when request times vary. IP hash or consistent hashing sends the same client or key to the same server, which helps with caches and sticky sessions.

A load balancer also runs health checks and stops sending traffic to failed servers. Layer 4 balancers route by IP and port and are very fast, while layer 7 balancers understand HTTP and can route by path, host or header and terminate TLS. The load balancer itself must not be a single point of failure, so it is deployed redundantly.

These answers are written for real interview conversations. Prefer explaining a trade-off out loud over reciting a definition. Full explanations and related questions: <https://www.fredevtemplates.com/interview-questions/level/junior>