

FLASHCARDS

Java

263 tarjetas de repaso rápido

frente = pregunta, dorso = respuesta · numeradas · 14 subtemas



ESTRELLA = pregunta clave de examen

Java · 263

Toca su nivel abajo a la izquierda; el mazo viene ordenado de lo BÁSICO a lo AVANZADO.

CÓMO USARLO

1 · IMPRIMIR

A4 a doble cara, volteando por el BORDE LARGO, escala 100% (sin "ajustar a página"). Corta por las líneas discontinuas: cada pregunta queda con su respuesta al dorso, verificable por la numeración.

2 · CELULAR

Abre flashcards.html en el navegador (sin internet): toca para voltear, marca «La sé» o «Repasar» y el progreso queda guardado. Incluye búsqueda, filtros por subtema, modo aleatorio y lista completa.

3 · ANKI / QUIZLET

Importa cards.csv (Frente ; Reverso ; Categoría ; Subtema) para estudiar con repetición espaciada.

MUESTRA A ESCALA REDUCIDA

FRENTE (página impar)

REVERSO (página par, espejado)

JAVA

J005

¿Java pasa parámetros por valor o por referencia?

JAVA

J005

RESPUESTA

SIEMPRE por valor.
Para objetos, el valor copiado es la referencia: el método puede mutar el objeto apuntado, pero reasignar el parámetro (param = otro) NO afecta al llamador.
Primitivos: copia directa, sin efecto externo.

Al imprimir a doble cara, cada respuesta queda exactamente detrás de su pregunta; la numeración lo verifica.

Contenido: 263 tarjetas + chuleta exprés al dorso de esta portada · 68 páginas · generado 12/09/2026

CHULETA EXPRÉS

Lo básico que siempre se pregunta: HTTP, seguridad, testing, datos, Docker/K8s y GCP.

- Verbos HTTP: GET leer · POST crear · PUT reemplaza · PATCH parcial · DELETE borra. Idempotentes: GET/PUT/DELETE. Seguro: GET.
- Códigos HTTP: 200 OK · 201 Created (+Location) · 204 sin body · 400 validación · 401 no autenticado · 403 sin permiso · 404 no existe · 409 conflicto · 429 rate limit · 500 bug · 503 caído.
- ACID: Atomicidad, Consistencia, Aislamiento, Durabilidad. CAP: ante partición (P) eliges consistencia (C) o disponibilidad (A).
- Escalera REST (Richardson): 0 RPC/POX · 1 recursos · 2 verbos + códigos HTTP · 3 HATEOAS.
- JWT: header.payload.signature; firma HS256 (simétrica) o RS256 (asimétrica). Claims clave: sub, iss, aud, exp, iat. Nunca guardar datos sensibles: es legible (base64).
- OAuth2: Authorization Code + PKCE para apps/SPA · Client Credentials para servicio-a-servicio · Refresh Token para renovar sin re-login.
- SQL rápido: índices B-tree aceleran WHERE/JOIN · EXPLAIN para ver el plan · N+1 se resuelve con JOIN FETCH / batch · paginar con keyset, no solo OFFSET.
- Concurrencia en BD: optimista = @Version + retry (mejor para poca contención) · pesimista = SELECT FOR UPDATE (contención alta).
- Pool de conexiones: HikariCP (default en Boot). Ajustar maximumPoolSize (10-20 suele bastar) y connectionTimeout.
- Caché HTTP: Cache-Control (max-age, no-cache) + ETag/If-None-Match → 304 Not Modified. Caché de app: Spring @Cacheable.
- JUnit 5: @Test, @BeforeEach, AssertJ assertThat. Mockito: @Mock + @InjectMocks. Tipos de test Spring: @WebMvcTest (capa web), @DataJpaTest (repositorios), @SpringBootTest (integración completa).
- TDD: rojo (test que falla) → verde (código mínimo) → refactor. Pirámide: muchos unit, algunos integration, pocos E2E.
- SOLID: S una razón de cambio por clase · O abierto a extensión, cerrado a modificación · L subtipos sustituibles · I interfaces pequeñas · D depender de abstracciones.
- GoF que salen siempre: Singleton, Factory Method, Builder, Strategy, Adapter, Observer, Decorator, Proxy (Spring AOP usa proxies dinámicos).
- Maven: coordenadas groupId:artifactId:version · scopes: compile, provided, runtime, test · BOM para alinear versiones · mvnw wrapper para versionar Maven.
- Docker: imagen inmutable por capas con cache · multi-stage build para imagen liviana · no correr como root · ENTRYPOINT vs CMD · healthcheck.
- Kubernetes: Pod (unidad mínima) · Deployment (réplicas + rolling update) · Service (IP estable/DNS) · Ingress (rutas HTTP) · HPA (autoescalado) · ConfigMap/Secret (config).
- Logging: niveles TRACE < DEBUG < INFO < WARN < ERROR. Estructurado (JSON) y con correlation ID. Nunca datos personales ni secretos en logs.
- CORS: el navegador lanza preflight OPTIONS; el servidor responde con Access-Control-Allow-Origin/Methods/Headers. Es un mecanismo del navegador, no del servidor.
- Git/entrega: feature branches + Pull Request · squash merge · Conventional Commits (feat/fix/chore) · SemVer MAYOR.MINOR.PATCH (mayor = rompe compatibilidad).
- Rate limiting: 429 Too Many Requests + header Retry-After. Algoritmos: token bucket, sliding window. Se aplica en gateway.
- Async en Spring: @Async + @EnableAsync (executor), @EventListener para eventos en proceso, CompletableFuture para composición.
- Programación reactiva (WebFlux): Mono (0..1) y Flux (0..N), backpressure, no bloquea hilos; útil con mucha I/O concurrente. MVC por hilos sigue siendo válido para CRUD normal.
- Microservicios: contrato primero (OpenAPI), consumer-driven contract tests (Pact), CI/CD por servicio, feature flags para desacoplar deploy de release.
- En la nube: 12-factor (config en entorno, procesos stateless, logs como stream), probes de salud, escalado horizontal primero, idempotencia en todo endpoint que escribe.
- K8s probes: liveness reinicia el pod, readiness lo saca del balanceo, startup da margen a arranques lentos. Requests/limits de CPU-memoria definen la calidad de servicio.
- JOINs SQL: INNER (solo coincidencias) · LEFT (toda la izquierda + coincidencias) · RIGHT · FULL. Subconsultas correlacionadas fila a fila = sospechar de N+1.
- HTTP/2: multiplexa streams en una conexión, comprime headers (HPACK) y permite prioridades; gRPC exige HTTP/2. HTTP/3 va sobre QUIC/UDP.
- Complejidad: HashMap get/put O(1) medio · búsqueda binaria O(log n) · ordenar O(n log n) · recorrer O(n). ArrayList contains O(n).
- Colecciones thread-safe: ConcurrentHashMap (no bloquea el mapa entero) · CopyOnWriteArrayList (muchas lecturas) · BlockingQueue (productor/consumidor).
- Bean Validation: @Valid en el parámetro + @NotNull/@Size/@Email/@Pattern en el DTO; si falla, MethodArgumentNotValidException → 400 con detalles.
- Clientes REST en Spring: RestClient (síncrono, moderno) · WebClient (reactivo) · OpenFeign (declarativo por interfaz). Configura siempre timeout y retry.
- Docker Compose levanta el stack local (app + BD + cola) con docker compose up; perfiles para servicios opcionales. En prod: Kubernetes o Cloud Run.
- GCP rápido: proyecto = unidad de facturación/IAM · región (multi-zona) vs zona · Cloud Run = contenedor serverless · Pub/Sub = at-least-once · IAM = least privilege · ADC para autenticar apps sin claves.

JAVA★

Sintaxis y tipos

J001

final / finally / finalize

BASICO

JAVA

Sintaxis y tipos

J002

¿Cuántos tipos de datos primitivos hay y cuáles son?

BASICO

JAVA

Sintaxis y tipos

J003

Primitivos vs wrappers (Integer, Double...)

BASICO

JAVA

Sintaxis y tipos

J004

¿Qué es el autoboxing y su trampa clásica?

BASICO

JAVA

Sintaxis y tipos

J005

¿Java pasa parámetros por valor o por referencia?

BASICO

JAVA

Sintaxis y tipos

J006

Overflow aritmético en int: ¿qué pasa?

BASICO

JAVA

Sintaxis y tipos

J007

Casting entre tipos numéricos: implícito vs explícito

BASICO

JAVA

Sintaxis y tipos

J008

¿Qué es var (Java 10+) y qué NO es?

BASICO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

Sintaxis y tipos

J002

RESPUESTA

8 primitivos: byte (8 bits), short (16), int (32), long (64), float (32), double (64), char (16, Unicode) y boolean.
Los demás tipos son objetos (clases, arrays, interfaces).

BASICO

JAVA



Sintaxis y tipos

J001

RESPUESTA

final: no se puede cambiar — variable (constante), método (no sobrescribible), clase (no heredable).
finally: bloque tras try/catch que casi siempre se ejecuta (limpieza).
finalize(): lo llama el GC antes de liberar el objeto; deprecado — no confíes en él.

BASICO

JAVA

Sintaxis y tipos

J004

RESPUESTA

Conversión automática primitiva $\leftrightarrow$ wrapper.
Trampa 1: un wrapper null al desempaquetar lanza NullPointerException.
Trampa 2: en loops grandes genera objetos (perf).
Trampa 3: comparaciones mixtas: `int == Integer` desempaqueta (NPE si null).

BASICO

JAVA

Sintaxis y tipos

J003

RESPUESTA

Primitivos: valor puro, rápido, no null, no van en colecciones.
Wrappers: objetos inmutables, permiten null, necesarios en genéricos/colecciones.
Autoboxing convierte automáticamente en ambas direcciones (con costo de rendimiento en loops).

BASICO

JAVA

Sintaxis y tipos

J006

RESPUESTA

Los enteros hacen wrap-around silencioso (sin excepción):
`Integer.MAX_VALUE + 1` $\rightarrow$ `Integer.MIN_VALUE`.
Soluciones: usar long, `Math.addExact/subtractExact` (lanzan `ArithmeticException`), o `BigInteger`.
Clásico bug: promedio $(a+b)/2 \rightarrow$ mejor `a + (b-a)/2`.

BASICO

JAVA

Sintaxis y tipos

J005

RESPUESTA

SIEMPRE por valor.
Para objetos, el valor copiado es la referencia: el método puede mutar el objeto apuntado, pero reasignar el parámetro (param = otro) NO afecta al llamador.
Primitivos: copia directa, sin efecto externo.

BASICO

JAVA

Sintaxis y tipos

J008

RESPUESTA

Inferencia de tipo local: `var lista = new ArrayList<String>()`; el tipo se fija en compilación (sigue siendo estático).
NO es tipo dinámico, NO aplica a campos, parámetros ni retornos, NO con null solo, NO sin inicializador.
Úsalo cuando el tipo es evidente a la derecha.

BASICO

JAVA

Sintaxis y tipos

J007

RESPUESTA

Implícito (widening): sin pérdida,
`byte` $\rightarrow$ `short` $\rightarrow$ `int` $\rightarrow$ `long` $\rightarrow$ `float` $\rightarrow$ `double` (`int` $\rightarrow$ `float` puede perder precisión en números grandes).
Explícito (narrowing): obligatorio y trunca: `int x=(int)3.99` $\rightarrow$ 3;
`byte b=(byte)130` $\rightarrow$ -126 (wrap).
`char` participa como numérico (0..65535).

BASICO

JAVA

Sintaxis y tipos

J009

Diferencia entre ++i e i++

BASICO

JAVA

Sintaxis y tipos

J010

¿Qué hace el operador ternario y su trampa con autoboxing?

BASICO

JAVA

Sintaxis y tipos

J011

switch clásico: ¿qué pasa sin break?

BASICO

JAVA

Sintaxis y tipos

J012

for vs while vs do-while vs for-each

BASICO

JAVA

Sintaxis y tipos

J013

Arrays en Java: características clave

BASICO

JAVA

Sintaxis y tipos

J014

¿Qué significa que un literal sea 1_000_000 o 0b1010?

BASICO

JAVA

Sintaxis y tipos

J015

¿Diferencia entre variable local, de instancia y estática?

BASICO

JAVA

Sintaxis y tipos

J016

¿Qué es instanceof y el pattern matching (Java 16+)?

BASICO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA
RESPUESTA

Sintaxis y tipos

J010

condicion ? valorSi : valorNo.

Trampa: si un brazo es primitivo y otro wrapper, el resultado se unifica (puede desempaquetar y lanzar NPE con null):

Integer x = cond ? 1 : null; → ok

Integer y = cond ? 1 : objetoIntegerNull; → NPE al evaluar.

BASICO

JAVA
RESPUESTA

Sintaxis y tipos

J009

++i (pre-incremento): incrementa y devuelve el valor nuevo.

i++ (post-incremento): devuelve el valor viejo y luego incrementa.

En x = i++ se asigna el valor anterior. En expresiones complejas mezclarlos dificulta leer el código.

BASICO

JAVA
RESPUESTA

Sintaxis y tipos

J012

for: núcleo conocido de iteraciones.

while: repite mientras la condición sea true (puede no ejecutarse nunca).

do-while: ejecuta al menos una vez.

for-each (for tipo x : colección): recorre iterables sin índice; no permite modificar la estructura ni saltar elementos fácilmente.

BASICO

JAVA
RESPUESTA

Sintaxis y tipos

J011

Cae al siguiente caso (fall-through) y ejecuta todo lo demás hasta un break o el final.

El switch de expresión (Java 14+) con flecha (case X ->) NO cae: cada rama es independiente y requiere exhaustividad con enums/sealed.

BASICO

JAVA
RESPUESTA

Sintaxis y tipos

J014

Subrayados en literales numéricos solo separan dígitos (legibilidad), desde Java 7.

Prefijos: 0x hexadecimal, 0b binario, 0 octal (¡trampa!), sufijos L long, f float, d double.

'\u0041' es escape Unicode para 'A'.

BASICO

JAVA
RESPUESTA

Sintaxis y tipos

J013

Tamaño fijo, longitud con .length (sin paréntesis), índices 0..n-1, covariantes (String[] es Object[] → ArrayStoreException posible), inicialización: int[] a = {1,2,3}; Arrays.copyOf, Arrays.sort, Arrays.asList (vista de tamaño fijo). Fuera de rango → ArrayIndexOutOfBoundsException.

BASICO

JAVA
RESPUESTA

Sintaxis y tipos

J016

instanceof verifica el tipo en runtime (null → siempre false).

Pattern matching: if (obj instanceof String s) { usar s } — hace el cast y declara la variable en una línea, sin repetir código.

BASICO

JAVA
RESPUESTA

Sintaxis y tipos

J015

Local: dentro de un método, vive durante la llamada, DEBE inicializarse antes de usarse (no hay default).

De instancia: por objeto, default según tipo (0, false, null).

Estática: por clase, compartida por todas las instancias, se inicializa al cargar la clase.

BASICO

JAVA

Sintaxis y tipos

J017

¿Cómo funciona el paso de String en un switch?

BASICO

JAVA

Sintaxis y tipos

J018

¿Qué es NaN y su trampa en comparaciones?

BASICO

JAVA

Sintaxis y tipos

J019

División entera vs decimal

BASICO

JAVA

Sintaxis y tipos

J020

¿Qué son los wrappers inmutables y cuáles tienen métodos útiles?

BASICO

JAVA

Sintaxis y tipos

J021

char vs String: diferencias clave

BASICO

JAVA

Sintaxis y tipos

J022

¿Qué es el alcance (scope) de una variable?

BASICO

JAVA

Sintaxis y tipos

J023

¿Cuándo usar long vs int vs double para dinero/tiempo?

BASICO

JAVA

Sintaxis y tipos

J024

Método main: firma válida y sobrecarga

BASICO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

Sintaxis y tipos

J018

RESPUESTA

Double.NaN es 'no es un número' (0/0.0).
NaN == NaN es FALSE; usar Double.isNaN() para detectarlo.
NaN en ordenamientos/comparaciones puede romper contratos de Comparator (Double.compare sí lo maneja).

BASICO

JAVA

Sintaxis y tipos

J017

RESPUESTA

Desde Java 7. Compara usando equals (case-sensitive), no ==.
Null en el switch → NullPointerException (verificar antes).
Con enum, los case van SIN calificar el nombre (case LUNES, no case Dia.LUNES).

BASICO

JAVA

Sintaxis y tipos

J020

RESPUESTA

Integer, Long, etc. son inmutables: las operaciones crean nuevos objetos.
Útiles: Integer.parseInt/valueOf, MAX_VALUE/MIN_VALUE, compare, Integer.parseInt(s, radix), Integer.toBinaryString, Character.isDigit/isLetter, Double.parseDouble.

BASICO

JAVA

Sintaxis y tipos

J019

RESPUESTA

int/int trunca: $7/2 \rightarrow 3$ (no redondea, hacia cero).
% es el residuo: $7\%2 \rightarrow 1$ (funciona con negativos con signo del dividendo).
Para decimal: que un operando sea double ($7/2.0 \rightarrow 3.5$).
División entera por cero → ArithmeticException; con doubles → Infinity o NaN.

BASICO

JAVA

Sintaxis y tipos

J022

RESPUESTA

Bloque: entre { } donde se declara (locales).
Campo: toda la clase (instancia/estática).
Parámetro: todo el método.
Una variable local puede ocultar un campo (this.x para desambiguar).
Los bloques internos NO pueden redeclarar una local del bloque externo.

BASICO

JAVA

Sintaxis y tipos

J021

RESPUESTA

char es un primitivo UTF-16 de 16 bits con comillas simples ('A');
String es objeto inmutable con comillas dobles.
char + int hace aritmética ('A'+1 → 66); char + String concatena.
Emojis y algunos Unicode ocupan 2 chars (surrogate pairs) → cuidado con length().

BASICO

JAVA

Sintaxis y tipos

J024

RESPUESTA

public static void main(String[] args) — también varargs String... args.
JVM solo invoca esa firma exacta; puedes SOBRECARGAR main pero la otra no se ejecuta sola.
args nunca es null (puede ser vacío), no incluye el nombre del programa.

BASICO

JAVA

Sintaxis y tipos

J023

RESPUESTA

Dinero: NUNCA double → BigDecimal o long de centavos.
Tiempo: epoch millis o nanos no caben en int (2038/overflow) → long.
Contadores pequeños y índices: int.
Identificadores grandes (IDs de DB): long.

BASICO

JAVA

Sintaxis y tipos

J025

varargs: sintaxis y reglas

BASICO

JAVA

Sintaxis y tipos

J026

¿Cómo se compila y ejecuta un programa Java? ¿Qué es el classpath?

BASICO

JAVA

Sintaxis y tipos

J027

Paquetes e imports: convención y propósito

BASICO

JAVA

Sintaxis y tipos

J028

Convenciones de nombres en Java (JavaBeans)

BASICO

JAVA

Sintaxis y tipos

J029

¿Qué es el Integer cache y cuándo sorprende?

INTERMEDIO

JAVA

Sintaxis y tipos

J030

BigDecimal: ¿cuándo y por qué?

INTERMEDIO

JAVA

Sintaxis y tipos

J031

¿Qué es 'effectively final' y por qué importa en lambdas?

INTERMEDIO

JAVA

Sintaxis y tipos

J032

Orden de inicialización de una clase

INTERMEDIO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

Sintaxis y tipos

J026

RESPUESTA

javac Hola.java → Hola.class (bytecode). java Hola → la JVM lo ejecuta.
 Varias clases/jars: java -cp "lib/*:clases" MiApp (classpath = dónde busca clases).
 Con Maven/Gradle esto lo automatiza el build; en un jar ejecutable: java -jar app.jar.
 -classpath mal puesto → ClassNotFoundException.

BASICO

JAVA

Sintaxis y tipos

J025

RESPUESTA

void sumar(int... nums) — acepta 0 o más ints (llega como array).
 Reglas: solo UNO por método y debe ser el ÚLTIMO parámetro.
 Sobrecarga con varargs es ambigua con facilidad: sumar(int...) y sumar(int, int...) → llamar sumar(5) no compila.
 Es azúcar sobre String[] args.

BASICO

JAVA

Sintaxis y tipos

J028

RESPUESTA

Clases/Interfaces/interfaces funcionales: PascalCase (MiServicio).
 Métodos y variables: camelCase (obtenerUsuario).
 Constantes: UPPER_SNAKE (MAX_INTENTOS).
 Paquetes: todo.minuscula.
 Getters/setters: getNombre/setNombre (isActive para boolean) — los frameworks (Jackson, Spring) confían en estas convenciones.

BASICO

JAVA

Sintaxis y tipos

J027

RESPUESTA

Paquete = namespace + organización: package com.empresa.proyecto.servicio;
 Convención: dominio invertido en minúsculas (com.google.x).
 import com.empresa.util.Fechas; o import estático: import static java.lang.Math.max;
 Sin import → nombre completo (java.util.List) o error de compilación.
 Los paquetes también definen el acceso package-private.

BASICO

JAVA

Sintaxis y tipos

J030

RESPUESTA

Para dinero y precisión decimal exacta: double/float usan binario (0.1+0.2 != 0.3).
 Crear SIEMPRE con String: new BigDecimal("0.1"), nunca new BigDecimal(0.1) (arrastra el error binario).
 Comparar con compareTo, no equals (equals considera escala: 1.0 != 1.00).
 Especificar scale y RoundingMode en divide().

INTERMEDIO

JAVA

Sintaxis y tipos

J029

RESPUESTA

Integer.valueOf() cachea de -128 a 127: dos autoboxed en ese rango son == el mismo objeto; fuera de él, objetos distintos.
 Por eso Integer a=127, b=127 → a==b es true, pero con 128 es false.
 Regla: en wrappers SIEMPRE usar equals(), nunca ==.

INTERMEDIO

JAVA

Sintaxis y tipos

J032

RESPUESTA

1) static: campos y bloques estáticos, en orden de aparición (una vez, al cargar).
 2) Por cada instancia: campos de instancia y bloques de instancia en orden, luego el constructor (que llama implícitamente a super() primero).
 Clásico de examen: el constructor de la clase padre corre ANTES que el cuerpo del hijo, pero los campos con inicializador del padre antes de su constructor.

INTERMEDIO

JAVA

Sintaxis y tipos

J031

RESPUESTA

Una variable local que no cambia después de inicializarse.
 Las lambdas y clases anónimas solo pueden capturar variables locales effectively final: se copian por valor al crear la lambda.
 Solución para acumular: usar un contenedor mutable (array de 1, AtomicInteger) o mejor refactorizar con reduce/collect.

INTERMEDIO

JAVA★

POO

J033

== vs equals() vs hashCode()

BASICO

JAVA★

POO

J034

¿Interfaz vs clase abstracta?

BASICO

JAVA★

POO

J035

Overloading vs Overriding

BASICO

JAVA★

POO

J036

¿Qué hace static?

BASICO

JAVA

POO

J037

¿Cuáles son los 4 pilares de la POO y qué significa cada uno?

BASICO

JAVA

POO

J038

Composición vs herencia: ¿cuándo cada una?

BASICO

JAVA

POO

J039

Constructores: reglas esenciales

BASICO

JAVA

POO

J040

this y super: usos

BASICO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA ★
RESPUESTA

POO

J034

Interfaz: contrato puro; métodos abstract + default/static (Java 8+); una clase implementa varias.
 Clase abstracta: no instanciable; tiene estado (campos), constructores y código común; herencia simple.
 Interfaz para capacidad ('puede...'); abstracta para compartir estado.

BASICO

JAVA ★
RESPUESTA

POO

J033

== compara referencias (mismo objeto en memoria).
 equals() compara estado; se sobrescribe junto a hashCode.
 Contrato: si a.equals(b), entonces a.hashCode() == b.hashCode().
 Si no cumples el contrato, HashMap/HashSet se rompen.

BASICO

JAVA ★
RESPUESTA

POO

J036

El miembro pertenece a la CLASE, no a cada instancia: compartido por todos los objetos y usable sin instanciar.
 Clase anidada static: no guarda referencia al exterior.
 Bloque static: inicialización una sola vez al cargar la clase.

BASICO

JAVA ★
RESPUESTA

POO

J035

Overloading (sobrecarga): mismo nombre, distinta firma (parámetros); se resuelve en COMPILACIÓN.
 Overriding (sobrescritura): redefinir un método heredado con la misma firma; se resuelve en EJECUCIÓN según el tipo real del objeto (dynamic dispatch → polimorfismo).

BASICO

JAVA
RESPUESTA

POO

J038

Regla: favorece composición.
 Herencia solo con relación is-a REAL y jerarquías estables (LSP garantizado); reutiliza y sustituye tipos.
 Composición (has-a): delega, flexible, fácil de testear, evita acoplarse a detalles del padre.
 Herencia rompe encapsulación: los cambios del padre afectan a los hijos.

BASICO

JAVA
RESPUESTA

POO

J037

Encapsulación: ocultar estado interno (campos private + métodos de acceso) y exponer comportamiento.
 Abstracción: modelar lo esencial, exponiendo QUÉ hace, no CÓMO (interfaces/abstract).
 Herencia: reutilización is-a de código y tipo (extends).
 Polimorfismo: un mismo mensaje, comportamientos según el tipo real (overriding + dynamic dispatch).

BASICO

JAVA
RESPUESTA

POO

J040

this: referencia al objeto actual — desambiguar campos (this.x = x), llamar otro constructor (this(...)), pasar el objeto actual.
 super: acceder al miembro del padre oculto (super.metodo()) o llamar su constructor (super(...)).
 En métodos static NO existe ni this ni super.

BASICO

JAVA
RESPUESTA

POO

J039

Mismo nombre que la clase, sin tipo de retorno.
 Si no defines ninguno, hay uno default sin args (desaparece al definir uno con args).
 this(...) llama a otro constructor (debe ser la primera línea);
 super(...) al del padre (también primera línea, implícito si no se escribe).
 Constructores no se heredan ni se sobrescriben (solo se sobrecargan).

BASICO

JAVA

POO

J041

Modificadores de acceso en Java

BASICO

JAVA

POO

J042

¿Puede una clase abstracta no tener métodos abstractos? ¿Y una interfaz tener estado?

BASICO

JAVA

POO

J043

Métodos default y static en interfaces (Java 8+)

BASICO

JAVA

POO

J044

¿Qué es una clase inmutable y cómo se escribe una?

BASICO

JAVA

POO

J045

Patrón Builder: ¿cuándo vale la pena?

BASICO

JAVA

POO

J046

Clases anidadas: static nested vs inner vs local vs anónima

BASICO

JAVA

POO

J047

¿Qué métodos define Object y cuáles conviene sobrescribir?

BASICO

JAVA★

POO

J048

Contrato correcto de equals()

BASICO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

POO

J042

RESPUESTA

Sí: una clase abstracta puede no tener métodos abstractos (solo evita instanciarse).
Una interfaz NO puede tener estado (campos de instancia); sus 'campos' son public static final constantes.
Ambas pueden tener métodos default/static (interfaz) o concretos (abstracta).

BASICO

JAVA

POO

J041

RESPUESTA

private: solo la clase.
(package-private, sin modificador): solo el mismo paquete.
protected: paquete + subclases (aunque estén en otro paquete).
public: todos.
Top-level: solo public o package-private.
Clases deben ser lo más cerradas posible.

BASICO

JAVA

POO

J044

RESPUESTA

Su estado no cambia tras construirse.
Receta: clase final, campos final private, sin setters, constructor que copia defensivamente colecciones/arrays mutables, getters que devuelven copias de lo mutable, no exponer this durante construcción.
Beneficios: thread-safe gratis, cacheable, seguro como clave de HashMap.

BASICO

JAVA

POO

J043

RESPUESTA

default: implementación en la interfaz, para evolución de APIs y mixins; la clase puede sobrescribirlo; en conflicto de diamante la clase DEBE resolverlo: InterfazA.super.metodo().
static: método de la interfaz (Interfaz.metodo()), no se hereda.
private (Java 9+): helpers internos compartidos entre defaults.

BASICO

JAVA

POO

J046

RESPUESTA

static nested: no necesita instancia externa (no guarda referencia al exterior).
inner (no static): requiere instancia externa, accede a sus campos (Outer.this); puede causar memory leaks (actividad Android).
local: dentro de un método.
anónima: sin nombre, se implementa al vuelo (hoy en día suele reemplazarse por lambdas si es una interfaz funcional).

BASICO

JAVA

POO

J045

RESPUESTA

Cuando un constructor tendría muchos parámetros (especialmente opcionales del mismo tipo → errores de orden) o pasos opcionales.
new Pizza.Builder("m").queso(true).aceitunas().build();
Legible, inmutable al final, validación centralizada en build().
Alternativa moderna: records + builder propio o withers.

BASICO

JAVA



POO

J048

RESPUESTA

Reflexivo: x.equals(x) true.
Simétrico: x.equals(y) → y.equals(x).
Transitivo: x=y, y=z → x=z.
Consistente: mismo resultado en llamadas repetidas.
x.equals(null) → false (nunca NPE).
Receta: instanceof o getClass, comparar campos relevantes, usar Objects.equals para null-safe.

BASICO

JAVA

POO

J047

RESPUESTA

equals, hashCode, toString (casi siempre), clone (raro, mejor copias), getClass, notify/notifyAll/wait (final, concurrencia), finalize (deprecado).
Regla de oro: equals y hashCode SIEMPRE juntos y consistentes.

BASICO

JAVA★

POO

J049

Contrato de hashCode() y qué pasa si lo rompes

BASICO

JAVA

POO

J050

toString(): buenas prácticas

BASICO

JAVA

POO

J051

Enums: capacidades que casi nadie usa

BASICO

JAVA

POO

J052

Clases sealed (Java 17+): ¿para qué sirven?

BASICO

JAVA

POO

J053

Records vs clases normales: ¿cuándo cada uno?

BASICO

JAVA

POO

J054

Comparable vs Comparator

BASICO

JAVA

POO

J055

Upcasting y downcasting

BASICO

JAVA

POO

J056

¿Cuándo marcar una clase/método final?

BASICO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

POO

J050

RESPUESTA

Generado por IDE/records para depurar rápido.
 Incluir campos clave; NO incluir secretos ni datos personales (va a logs).
 getClass().getSimpleName() + "(a=" + a + ")".
 Evitar llamar métodos que disparan lazy loading (StackOverflow/exception en JPA).

BASICO

JAVA



POO

J049

RESPUESTA

Consistente entre llamadas; iguales por equals → mismo hashCode (obligatorio); desiguales → IDEALMENTE distinto (colisión permitida).
 Si lo rompes: HashMap/HashSet almacenan en buckets distintos y no encuentran objetos 'iguales' → contains() false con un objeto que debería estar.
 hashCode debe usar los MISMOS campos que equals.

BASICO

JAVA

POO

J052

RESPUESTA

Restringen QUIÉN puede heredar/implementar: public sealed interface Forma permits Circulo, Cuadrado.
 Los permits deben ser final, sealed o non-sealed.
 Habilita switch con pattern matching exhaustivo (el compilador sabe todas las subclases).
 Modela jerarquías cerradas de dominio con seguridad.

BASICO

JAVA

POO

J051

RESPUESTA

Constructor private, campos y métodos, puede implementar interfaces, abstract methods por constante (case por valor), valueOf/values(), ordinal (¡evita persistirlo!), compareTo por orden de declaración, EnumSet/EnumMap (rápidos).
 Enum con constructor: enum Planeta { TIERRA(5.97){...} } con campos y comportamiento por constante.

BASICO

JAVA

POO

J054

RESPUESTA

Comparable (compareTo): orden NATURAL, lo implementa la propia clase (String, Integer).
 Comparator: órdenes EXTERNOS y múltiples: Comparator.comparing(User::getNombre).thenComparing(User::getEdad).reversed().
 Contrato: sgn(compare(x,y)) == -sgn(compare(y,x)), consistente con equals idealmente; romperlo rompe TreeSet/TreeMap.

BASICO

JAVA

POO

J053

RESPUESTA

Record: dato immutable transparente — genera constructor canónico, accesorios sin 'get', equals/hashCode/toString; NO tiene setters, no hereda.
 Clase: cuando necesitas mutabilidad, identidad con estado variable, herencia, o validación compleja post-construcción.
 Records perfectos para DTOs, valores, resultados de cálculo y claves compuestas de Map.

BASICO

JAVA

POO

J056

RESPUESTA

Clase final: no heredable (String, Integer) — inmutabilidad y seguridad, diseños cerrados.
 Método final: no sobrescribible — proteger invariantes.
 Costo: pierde flexibilidad/testing con mocks de herencia. Diseña final por defecto en clases de valor; abierto solo donde hay extensión real pensada.

BASICO

JAVA

POO

J055

RESPUESTA

Upcasting (hijo→padre): implícito y seguro, pierde acceso a miembros del hijo.
 Downcasting (padre→hijo): cast explícito, lanza ClassCastException si el tipo real no coincide; verificar con instanceof antes.
 Downcasts frecuentes = diseño gritando por polimorfismo o genéricos.

BASICO

JAVA

POO

J057

Herencia de constructores y llamada implícita a super()

BASICO

JAVA

POO

J058

¿Qué es una interfaz funcional?

BASICO

JAVA

POO

J059

Diferencia entre abstracción e encapsulación (pregunta trampa)

BASICO

JAVA

POO

J060

¿Qué es el acoplamiento y la cohesión?
¿Cómo se relacionan con buen diseño?

BASICO

JAVA

POO

J061

Sobrecarga y resolución: ¿qué método se elige?

BASICO

JAVA

POO

J062

¿Se puede sobrecargar/sobrescribir main o un constructor?

BASICO

JAVA

POO

J063

¿Qué es y para qué sirve una clase de utilidades (utility class)?

BASICO

JAVA

POO

J064

¿Qué es el dynamic dispatch (enlace dinámico)?

INTERMEDIO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

POO

J058

RESPUESTA

Interfaz con EXACTAMENTE un método abstracto (SAM); puede tener defaults/statics.

Anotación @FunctionalInterface (opcional, la valida el compilador).

Destino de lambdas y referencias de método.

Core en java.util.function: Function<T,R>, Predicate<T>, Consumer<T>, Supplier<T>, BiFunction, UnaryOperator.

BASICO

JAVA

POO

J057

RESPUESTA

Los constructores NO se heredan.

Todo constructor empieza (implícita o explícitamente) con super() — si el padre NO tiene constructor sin args, el hijo DEBE llamar super(args) explícitamente o no compila.

El bloque del padre corre antes que el del hijo.

BASICO

JAVA

POO

J060

RESPUESTA

Cohesión: una clase con UNA responsabilidad clara (alta cohesión = bien).

Acoplamiento: cuánto depende una clase de otras (bajo acoplamiento = bien).

Objetivo: alta cohesión + bajo acoplamiento → fácil de cambiar, testear y reusar.

Herramientas: interfaces, inyección de dependencias, módulos.

BASICO

JAVA

POO

J059

RESPUESTA

Abstracción: decidir QUÉ expone el modelo (el contrato, el 'qué') — nivel de diseño.

Encapsulación: OCULTAR el estado y proteger invariantes (private + métodos) — mecanismo del lenguaje.

Se puede encapsular sin abstraer bien (getters de todo = encapsulación rota).

BASICO

JAVA

POO

J062

RESPUESTA

main: se puede SOBRECARGAR (otras firmas) pero la JVM solo ejecuta public static void main(String[]).

Constructores: solo SOBRECARGA (entre ellos con this(...)); no existe overriding de constructores porque no se heredan.

Un método static NO se sobrescribe (se oculta, hiding).

BASICO

JAVA

POO

J061

RESPUESTA

El compilador elige el método por el tipo ESTÁTICO de los argumentos, con la conversión más específica posible (primitivo exacto > widening > autoboxing > varargs).

Trampa: sum(5) con sum(int) y sum(long): elige int.

Con null ambiguo entre sobrecargas → error de compilación (hay que hacer cast).

BASICO

JAVA

POO

J064

RESPUESTA

En runtime se invoca el método del TIPO REAL del objeto, no del tipo de la referencia:

Animal a = new Perro(); a.hablar(); → ejecuta Perro.hablar().

Solo aplica a métodos de instancia (no static, no campos, no private/final).

Los CAMPOS no son polimórficos: se resuelven por el tipo de la referencia (field hiding).

INTERMEDIO

JAVA

POO

J063

RESPUESTA

Clase final con constructor private (no instanciable) y solo métodos static: Collections, Arrays, StringUtils.

Útil para funciones puras sin estado.

Crítica: dificulta mocking y polimorfismo; en código testeable se prefieren componentes inyectados con interfaz.

BASICO

JAVA

POO

J065

¿Qué es el patrón Método Fábrica estático (effective Java item 1) y por qué preferirlo a constructores?

INTERMEDIO

JAVA

POO

J066

Field hiding vs overriding: diferencia sutil pero de examen

INTERMEDIO

JAVA

POO

J067

clone() y por qué casi siempre se evita

INTERMEDIO

JAVA

POO

J068

¿Qué es la covarianza de arrays y su riesgo?

AVANZADO

JAVA★

String y texto

J069

¿Por qué String es immutable?

BASICO

JAVA

String y texto

J070

Métodos esenciales de String que hay que saberse

BASICO

JAVA

String y texto

J071

StringBuilder vs StringBuffer vs String.concat

BASICO

JAVA

String y texto

J072

split() y su trampa con espacios finales y caracteres especiales

BASICO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

POO

J066

RESPUESTA

Campos y métodos static se resuelven por el TIPO DE LA REFERENCIA (hiding, en compilación).
Métodos de instancia se resuelven por el TIPO REAL del objeto (overriding, en runtime).
Padre p = new Hijo(); p.x da el campo del Padre; p.metodo() ejecuta el del Hijo.

INTERMEDIO

JAVA

POO

J065

RESPUESTA

Métodos static que devuelven instancias: List.of(), Integer.valueOf(), Optional.of().
Ventajas: tienen NOMBRE, pueden cachear, devolver subtipos, no requieren crear objeto nuevo.
Convenciones: of, ofNullable, valueOf, from, newInstance.

INTERMEDIO

JAVA

POO

J068

RESPUESTA

String[] es subtipo de Object[] (arrays covariantes en runtime).
Riesgo: Object[] arr = new String[1]; arr[0] = 42 compila pero lanza ArrayStoreException en runtime.
Las colecciones NO son covariantes (List<String> no es List<Object>) — por eso existen los wildcards.

AVANZADO

JAVA

POO

J067

RESPUESTA

Object.clone es protected, requiere implementar Cloneable (que no tiene métodos) y hacer super.clone(); con campos mutables hay que hacer copia profunda a mano.
Alternativas: constructor de copia (new Punto(Punto otro)), método estático copiaDe(), o record (immutable no necesita clonar).
Effective Java: prefiere factories de copia.

INTERMEDIO

JAVA

String y texto

J070

RESPUESTA

length, charAt, substring(begin[,end]) — end exclusivo, indexOf/lastIndexOf, contains, startsWith/endsWith, replace, split, trim/strip (strip es Unicode, Java 11), toUpper/LowerCase, isEmpty/.isBlank (sin contenido o solo espacios), equals/equalIgnoreCase, compareTo, chars(), repeat, join, format.

BASICO

JAVA



String y texto

J069

RESPUESTA

Su contenido no cambia tras crearse (los métodos devuelven nuevos Strings).
Beneficios:

- String pool reutiliza instancias
- Thread-safe
- hashCode cacheado
- Seguro como clave en HashMap y para rutas/URLs

BASICO

JAVA

String y texto

J072

RESPUESTA

split usa regex: split(".") NO funciona (punto = 'cualquier carácter') → escapar split("\\.").
Los strings vacíos del FINAL se descartan por defecto: "a,b,.".split(",") → [a,b].
Usar split(",", -1) para conservarlos (útil en CSV).

BASICO

JAVA

String y texto

J071

RESPUESTA

StringBuilder: mutable, NO sincronizado (el normal desde Java 5).
StringBuffer: igual pero sincronizado (legado).
concat/+ para pocas uniones.
Capacidad inicial 16; crece duplicando+2. new StringBuilder(tamañoEstimado) evita reallocs.

BASICO

JAVA

String y texto

J073

String.format y text blocks

BASICO

JAVA

String y texto

J074

trim() vs strip() vs isBlank()

BASICO

JAVA

String y texto

J075

Comparar Strings correctamente en todos los casos

BASICO

JAVA

String y texto

J076

String.join y StringJoiner

BASICO

JAVA

String y texto

J077

¿Cómo iterar los caracteres de un String moderno?

BASICO

JAVA

String y texto

J078

Conversión String ↔ número y sus excepciones

BASICO

JAVA

String y texto

J079

¿Qué es un text block y cómo maneja la indentación?

BASICO

JAVA

String y texto

J080

¿Qué es el String pool y cómo funciona?

INTERMEDIO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

String y texto

J074

RESPUESTA

trim() elimina caracteres <= espacio (ASCII, legado).
strip() (Java 11) usa Character.isWhitespace — Unicode correcto.
isBlank() true si vacío o solo whitespace.
isEmpty() solo true con length 0.

BASICO

JAVA

String y texto

J073

RESPUESTA

format("%s tiene %d años", nombre, edad): %s string, %d entero, %f decimal (%.2f 2 decimales), %n salto, %% literal.
Text blocks (Java 15+): """ ... """ multilinea; quita indentación común (incidental), respeta saltos; útiles para JSON/SQL legible.

BASICO

JAVA

String y texto

J076

RESPUESTA

String.join(", ", "a", "b") o String.join(delim, lista).
StringJoiner permite prefijo/sufijo: new StringJoiner(", ", "[", "]").
Equivalente en streams: Collectors.joining(", ", "[", "]").

BASICO

JAVA

String y texto

J075

RESPUESTA

SIEMPRE a.equals(b), no ==.
Ignorando mayúsculas: equalsIgnoreCase o mejor compareTo case-normalized.
Null-safe: Objects.equals(a,b) o "constante".equals(variable).
Orden natural: compareTo (0 = iguales) — cuidado con null (NPE).

BASICO

JAVA

String y texto

J078

RESPUESTA

Integer.parseInt("42") → int; valueOf("42") → Integer.
Double.parseDouble, Long.parseLong.
Formato inválido → NumberFormatException (checked? NO, unchecked).
Convertir número a String: String.valueOf(x) o Integer.toString(x).
Base distinta: parseInt("ff", 16) → 255.

BASICO

JAVA

String y texto

J077

RESPUESTA

chars() devuelve IntStream: "abc".chars().filter(c -> c != 'a')...
Loop clásico con charAt(i) para índices.
OJO: codePoints() para Unicode real (emojis ocupan 2 chars; chars() parte los surrogate pairs).

BASICO

JAVA

String y texto

J080

RESPUESTA

Área del heap donde la JVM guarda Strings únicos. Los literales ("hola") se internan automáticamente: dos literales iguales son el MISMO objeto.
new String("hola") crea objeto FUERA del pool (por eso == da false); .intern() lo mete/busca en el pool.
Ahorra memoria y permite comparación rápida.

INTERMEDIO

JAVA

String y texto

J079

RESPUESTA

""" ... """ (Java 15+). El compilador calcula la indentación MÍNIMA común (incluida la línea de cierre) y la quita.
Secuencias: \s preserva espacio al final de línea, \ evita el salto de línea (concatenar), \n explícito si se quiere.
No interpola variables: usar .formatted() o format.

BASICO

JAVA

String y texto

J081

¿Por qué concatenar String en un loop es un problema?

INTERMEDIO

JAVA

String y texto

J082

Regex en Java: Pattern y Matcher

INTERMEDIO

JAVA

String y texto

J083

¿Cuánta memoria ocupa un String y por qué importa?

AVANZADO

JAVA★

Colecciones

J084

ArrayList vs LinkedList

BASICO

JAVA★

Colecciones

J085

List vs Set vs Map

BASICO

JAVA

Colecciones

J086

Jerarquía del Collection Framework (mapa mental)

BASICO

JAVA

Colecciones

J087

HashSet: cómo decide si un elemento 'ya está'

BASICO

JAVA

Colecciones

J088

LinkedHashMap y LinkedHashSet: el punto medio perfecto

BASICO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

String y texto

J082

RESPUESTA

Pattern p = Pattern.compile("regex"); Matcher m = p.matcher(texto); m.find() itera coincidencias, m.group() la extrae, m.matches() exige coincidencia total.
Grupos con paréntesis: m.group(1).
Backslashes dobles en Java: "\\d+".
Para validar simple: cadena.matches("regex").
Compilar el Pattern UNA vez si se reutiliza (thread-safe).

INTERMEDIO

JAVA

String y texto

J081

RESPUESTA

Cada + crea un String nuevo y copia todo el contenido → $O(n^2)$ en el loop.
Solución: StringBuilder en un hilo (append), StringBuffer solo si varios hilos comparten el builder (raro, está sincronizado).
El compilador optimiza concatenaciones EN UNA MISMA expresión, no en loops.

INTERMEDIO

JAVA ★

Colecciones

J084

RESPUESTA

ArrayList: array dinámico; get por índice $O(1)$; insertar/borrar en medio $O(n)$ (desplaza elementos).
LinkedList: doblemente enlazada; add/remove en extremos $O(1)$; get $O(n)$.
En la práctica gana ArrayList casi siempre (localidad de caché).

BASICO

JAVA

String y texto

J083

RESPUESTA

Objeto String ≈ 24B cabecera + array de bytes. Desde Java 9 es byte[] con codificación compacta (Latin-1 si basta, UTF-16 si no) — menos memoria que el char[] antiguo.
Duplicados en colecciones grandes → intern() o mapas de deduplicación.
Substrings ya NO comparten array (copia) desde Java 7u6.

AVANZADO

JAVA

Colecciones

J086

RESPUESTA

Iterable → Collection → {List, Set, Queue}. Map aparte (no es Collection).
List: ArrayList, LinkedList, Vector(legacy), CopyOnWriteArrayList.
Set: HashSet, LinkedHashSet, TreeSet, EnumSet.
Queue/Deque: ArrayDeque, PriorityQueue, LinkedList.
Map: HashMap, LinkedHashMap, TreeMap, ConcurrentHashMap, Hashtable(legacy), EnumMap.

BASICO

JAVA ★

Colecciones

J085

RESPUESTA

List: ordenada, con índices, permite duplicados (ArrayList, LinkedList).
Set: SIN duplicados (HashSet, TreeSet ordena por comparación).
Map: pares clave→valor, claves únicas; no hereda de Collection (HashMap, TreeMap).

BASICO

JAVA

Colecciones

J088

RESPUESTA

Hash $O(1)$ + ORDEN DE INSERCIÓN predecible (lista doble enlazada interna).
Access-order mode: new LinkedHashMap(16, 0.75f, true) reordena por acceso → base perfecta para una caché LRU (sobrescribir removeEldestEntry).

BASICO

JAVA

Colecciones

J087

RESPUESTA

1) calcula hashCode() del objeto para elegir bucket, 2) dentro del bucket compara con equals().
Si sobrescribes equals sin hashCode (o el objeto MUTA su hash después de insertar) → contains() falla.
Por eso: objetos clave inmutables o no mutar campos usados en hash tras insertar.

BASICO

JAVA

Colecciones

J089

ArrayDeque vs Stack vs LinkedList para pilas y colas

BASICO

JAVA

Colecciones

J090

PriorityQueue: comportamiento y trampas

BASICO

JAVA

Colecciones

J091

¿Cómo eliminar elementos de una lista mientras la recorres?

BASICO

JAVA★

Colecciones

J092

Comparable vs Comparator aplicado a ordenar colecciones

BASICO

JAVA

Colecciones

J093

List.of() vs Arrays.asList() vs new ArrayList<>()

BASICO

JAVA

Colecciones

J094

Colecciones no modificables vs inmutables

BASICO

JAVA★

Colecciones

J095

Métodos de Map modernos que quitan muchos ifs

BASICO

JAVA

Colecciones

J096

Iterator: contrato y uso correcto

BASICO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

Colecciones

J090

RESPUESTA

Heap binario: el elemento de MAYOR prioridad sale primero (el menor con orden natural).
 poll() $O(\log n)$, peek $O(1)$; NO está ordenada internamente (iterar no da orden).
 No admite null; con objetos necesita Comparable/Comparator.
 Uso: top-K, schedulers, Dijkstra.

BASICO

JAVA

Colecciones

J089

RESPUESTA

ArrayDeque: la opción recomendada — array circular, $O(1)$ amortizado en ambos extremos, menos GC.
 java.util.Stack hereda de Vector (sincronizado, legado, diseño malo) → evitar.
 LinkedList también implementa Deque pero con overhead de nodos y peor localidad de caché.
 Métodos: push/pop/peek, offer/poll/element, addFirst/last.

BASICO

JAVA ★

Colecciones

J092

RESPUESTA

lista.sort(comparator) o Collections.sort(lista).
 Comparator.comparing(User::getApellido).thenComparing(User::getEdad).reversed().
 nullsFirst/nullsLast para nulls.
 Comparar descendente: comparing(X).reversed() o .reversed() al final de la cadena completa (ojo: reversed() revierte TODO lo acumulado).

BASICO

JAVA

Colecciones

J091

RESPUESTA

1) Iterator: it.remove() — el clásico correcto.
 2) removeIf(predicado) — Java 8+, el más limpio (interno usa iterator).
 3) Recolectar en nueva lista (si no quieres mutar).
 MAL: for-each con lista.remove(x) → ConcurrentModificationException.

BASICO

JAVA

Colecciones

J094

RESPUESTA

Collections.unmodifiableList(vista): solo prohíbe modificar LA VISTA; si la lista de fondo cambia, la vista la refleja.
 List.of()/Set.of()/Map.of() (Java 9+): inmutables de verdad, copian los datos.
 Para exponer estado interno: copiar o devolver inmutable.

BASICO

JAVA

Colecciones

J093

RESPUESTA

List.of() (Java 9+): inmutable de verdad — add/remove/set lanzan UnsupportedOperationException; no permite nulls.
 Arrays.asList(): vista de tamaño FIJO sobre el array — set() sí funciona, add/remove no; refleja cambios del array.
 new ArrayList<>(): mutable completa.
 Pasar una de las fijas a código que muta → excepción en runtime.

BASICO

JAVA

Colecciones

J096

RESPUESTA

hasNext(), next(), remove() (único borrado seguro durante iteración).
 Solo se avanza hacia adelante, una pasada.
 ListIterator además: hasPrevious/previous, add, set, índices — bidireccional en List.
 Lanzan ConcurrentModificationException si la colección cambia por fuera.

BASICO

JAVA ★

Colecciones

J095

RESPUESTA

getOrDefault, putIfAbsent, computeIfAbsent (patrón memoización: map.computeIfAbsent(k, x -> new ArrayList<>()).add(v)), computeIfPresent, merge (contador: map.merge(k, 1, Integer::sum)), forEach(k,v), getOrDefault.
 put devuelve el valor ANTERIOR (null si no había).

BASICO

JAVA

Colecciones

J097

¿Por qué Map no extiende de Collection?

BASICO

JAVA

Colecciones

J098

entrySet vs keySet para recorrer un Map

BASICO

JAVA★

Colecciones

J099

¿Cómo funciona HashMap internamente?

INTERMEDIO

JAVA

Colecciones

J100

TreeSet/TreeMap: qué garantizan y qué cuestan

INTERMEDIO

JAVA

Colecciones

J101

ArrayList: cómo crece internamente

INTERMEDIO

JAVA

Colecciones

J102

fail-fast vs fail-safe en iteradores

INTERMEDIO

JAVA

Colecciones

J103

Complejidades que hay que memorizar

INTERMEDIO

JAVA★

Colecciones

J104

ConcurrentHashMap: qué lo hace diferente de un HashMap sincronizado

INTERMEDIO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

Colecciones

J098

RESPUESTA

Si necesitas clave Y valor: `entrySet()` — una sola búsqueda.
`keySet()` + `get(k)` hace DOBLE hash por entrada (innecesario).
`forEach((k,v) -> ...)` es el moderno.
`values()` cuando solo importan los valores.

BASICO

JAVA

Colecciones

J097

RESPUESTA

Porque es un mapeo clave→valor, no una colección de elementos: sus 'colecciones' derivadas son `keySet()`, `values()` y `entrySet()` (esas SÍ son `Collection/View`).
 Diseño: `Map` es otra abstracción distinta; se iteran sus entradas con `entrySet()`.

BASICO

JAVA

Colecciones

J100

RESPUESTA

Mantienen elementos ORDENADOS según `Comparable/Comparator` (árbol rojo-negro).
 Costo: `add/remove/contains` $O(\log n)$ vs $O(1)$ de `HashSet/HashMap`.
 Extras navegables: `first/last`, `floor/ceiling`, `headSet/tailSet`, `descendingSet`.
 No admiten null (con orden natural) y rompen el contrato si el comparador es inconsistente con equals.

INTERMEDIO

JAVA



Colecciones

J099

RESPUESTA

Array de buckets; índice = `hash(key) & (n-1)`.
 Colisiones: lista enlazada; desde Java 8 se balancea como árbol rojo-negro si el bucket llega a 8.
 Capacidad inicial 16, factor de carga 0.75 → `resize` duplica.
 Claves únicas; una clave null permitida.

INTERMEDIO

JAVA

Colecciones

J102

RESPUESTA

Fail-fast (`ArrayList`, `HashMap`): detectan modificación estructural durante la iteración (contador `modCount`) → `ConcurrentModificationException`. No es para hilos, es detección de bugs.
 Fail-safe: `CopyOnWriteArrayList` (itera sobre snapshot) y `ConcurrentHashMap` (iterador débilmente consistente) no lanzan.
 Eliminar mientras iteras: usar `iterator.remove()`, no `lista.remove()`.

INTERMEDIO

JAVA

Colecciones

J101

RESPUESTA

Array interno; al llenarse crece ~50% (nueva capacidad = `vieja*1.5`) y copia elementos.
`ensureCapacity()/constructor` con capacidad evita `reallocs` cuando sabes el tamaño.
`remove/contains` $O(n)$; `trimToSize()` libera sobrante.

INTERMEDIO

JAVA



Colecciones

J104

RESPUESTA

Segmentación/CAS: permite lecturas concurrentes sin lock y escrituras con locks finos por nodo — mucho mejor throughput.
 NO permite claves/valores null.
 Operaciones compuestas atómicas: `putIfAbsent`, `compute`, `merge`.
 Iterador débilmente consistente (no lanza CME).
`Collections.synchronizedMap` bloquea TODO el mapa → cuello de botella.

INTERMEDIO

JAVA

Colecciones

J103

RESPUESTA

`ArrayList`: `get` $O(1)$, `add` final amortizado $O(1)$, `add/remove` medio $O(n)$, `contains` $O(n)$.
`HashMap/HashSet`: $O(1)$ medio, $O(\log n)$ peor (treeify).
`TreeMap/TreeSet`: $O(\log n)$.
`LinkedList`: `add` extremos $O(1)$, `get i` $O(n)$.
`ArrayDeque`: extremos $O(1)$.
`PriorityQueue`: `offer/poll` $O(\log n)$.
`Binary search`: $O(\log n)$ (requiere orden).

INTERMEDIO

JAVA

Colecciones

J105

¿Cuándo CopyOnWriteArrayList?

INTERMEDIO

JAVA

Colecciones

J106

Colecciones y null: quién lo permite y quién no

INTERMEDIO

JAVA

Colecciones

J107

EnumSet y EnumMap: por qué los menciona todo entrevistador

INTERMEDIO

JAVA

Colecciones

J108

Collections utility: métodos que salvan vidas

INTERMEDIO

JAVA

Colecciones

J109

Capacidad inicial y factor de carga: ¿por qué afinarlos?

INTERMEDIO

JAVA

Colecciones

J110

¿List.contains con objetos propios devuelve false? Los 3 sospechosos

INTERMEDIO

JAVA

Colecciones

J111

remove(int index) vs remove(Object) en List: la trampa clásica

INTERMEDIO

JAVA

Colecciones

J112

¿Cómo funciona el resize de HashMap y su costo?

AVANZADO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

Colecciones

J106

RESPUESTA

Permiten null: ArrayList, LinkedList, HashMap (una clave null), HashSet, LinkedHashMap.
 NO permiten: TreeMap/TreeSet (con orden natural, lanza NPE al comparar), ConcurrentHashMap, Hashtable, Queue concurrentes, List.of/Set.of/Map.of, PriorityQueue.
 Diseño moderno: evitar nulls (Optional, valores default).

INTERMEDIO

JAVA

Colecciones

J105

RESPUESTA

Cuando hay MUCHÍSIMAS lecturas y casi cero escrituras (listeners, configs).
 Cada escritura copia el array completo → costosa, pero los iteradores nunca ven CME ni cambios (snapshot).
 Mal para listas que mutan seguido.

INTERMEDIO

JAVA

Colecciones

J108

RESPUESTA

sort, shuffle, reverse, swap, binarySearch (requiere orden), min/max con comparator, frequency, disjoint, unmodifiableX, synchronizedX, emptyList/emptyMap, singletonList, nCopies, addAll, rotate.
 Muchos tienen equivalente moderno en la propia interfaz (list.sort, list.forEach).

INTERMEDIO

JAVA

Colecciones

J107

RESPUESTA

Implementaciones especiales para enums basadas en arrays de bits/array plano: muchísimo más rápidas y compactas que HashSet/HashMap de enums.
 EnumSet.of(Dia.LUN, Dia.VIE), EnumMap<key, Valor>.
 Orden = orden de declaración del enum.

INTERMEDIO

JAVA

Colecciones

J110

RESPUESTA

1) equals() no sobrescrito (compara referencias).
 2) equals sobrescrito pero hashCode no (en HashSet/HashMap).
 3) El objeto MUTÓ un campo que participa en equals/hashCode después de insertarlo.
 Fix: equals+hashCode consistentes + objetos clave inmutables.

INTERMEDIO

JAVA

Colecciones

J109

RESPUESTA

HashMap default: capacidad 16, factor 0.75 → resize a las 12 inserciones (duplica y rehash).
 Si sabes que guardarás 1M entradas: new HashMap<>(1_500_000) evita ~17 resizes.
 Capacidad deseada = esperado / 0.75 (+1 para hash seguro).
 ArrayList: igual idea con constructor.

INTERMEDIO

JAVA

Colecciones

J112

RESPUESTA

Al superar capacidad*loadFactor: crea array del doble y RE-DISTRIBUYE todas las entradas (rehash). Costo O(n) puntual, amortizado O(1) por inserción.
 Java 8 conserva el orden relativo dentro de buckets (split por bit alto), antes invertía listas.
 Con muchas inserciones previstas, presizar el mapa evita trabajo.

AVANZADO

JAVA

Colecciones

J111

RESPUESTA

List<Integer> nums: remove(1) elimina el ÍNDICE 1;
 remove(Integer.valueOf(1)) elimina el OBJETO 1.
 El compilador elige por el tipo estático del argumento.
 Fuente infinita de bugs con listas de números.

INTERMEDIO

JAVA

Colecciones

J113

¿Qué son las vistas (views) de colecciones?

AVANZADO

JAVA

Genéricos

J114

¿Para qué existen los genéricos y qué problema arreglan?

BASICO

JAVA

Genéricos

J115

¿Por qué no hay List<int>?

BASICO

JAVA

Genéricos

J116

Métodos genéricos: sintaxis y cuándo

BASICO

JAVA

Genéricos

J117

Raw types y por qué el compilador los marca

BASICO

JAVA

Genéricos

J118

Herencia con genéricos: ¿List<Dog> es subtipo de List<Animal>?

BASICO

JAVA

Genéricos

J119

Genéricos, static y arrays: las tres prohibiciones

BASICO

JAVA

Genéricos

J120

Bounded types: <T extends Comparable<T>>

INTERMEDIO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

Genéricos

J114

RESPUESTA

Tipado seguro en compilación para colecciones/algoritmos reutilizables: `List<String>` garantiza que solo hay Strings (antes: raw types y casts a mansalva con `ClassCastException` en runtime). Eliminan casts explícitos y documentan el contrato del tipo.

BASICO

JAVA

Colecciones

J113

RESPUESTA

Objetos que ENVUELVEN otro sin copiar: `keySet()`, `subList()`, `unmodifiableList()`, `Arrays.asList()`. Cambios en el respaldo se reflejan en la vista (y a veces al revés: `subList` escribe en la original). Peligro: mutar el respaldo invalida la vista (`subList` → `ConcurrentModificationException`).

AVANZADO

JAVA

Genéricos

J116

RESPUESTA

`static <T> T primero(List<T> lista)` — el `<T>` va ANTES del tipo de retorno.
Se infiere del contexto: `String s = primero(nombres)`.
Útil cuando el método es genérico aunque la clase no, o para vincular tipos de argumento y retorno entre sí.

BASICO

JAVA

Genéricos

J115

RESPUESTA

Los genéricos solo aceptan tipos de referencia (type erasure + Object interno).
`List<int>` no existe → `List<Integer>` con autoboxing (overhead de memoria/GC).
Alternativas de rendimiento: `IntStream`, librerías (fastutil, Eclipse Collections) o arrays primitivos.

BASICO

JAVA

Genéricos

J118

RESPUESTA

NO (invarianza) — aunque `Dog` extends `Animal`. Si lo fuera, podrías meter un `Cat` en una `List<Dog>` vía la referencia `List<Animal>` y romper todo.
Lo correcto: `List<? extends Animal>` para leer.
`G = G<D>` solo si el tipo parametrizado coincide EXACTO: `ArrayList<Dog>` sí es `Collection<Dog>`.

BASICO

JAVA

Genéricos

J117

RESPUESTA

`List` sin parámetro = raw type: vuelve al mundo pre-Java 5 (acepta cualquier cosa, casts inseguros).
NO es lo mismo que `List<Object>` (esta es tipada y segura).
Usar raw types genera unchecked warnings y bugs silenciosos; solo aparecen por interoperabilidad legada.

BASICO

JAVA

Genéricos

J120

RESPUESTA

Limita `T` a un tipo o subtipos: `<T extends Number>` permite usar métodos de `Number` sobre `T`.
Múltiples bounds: `<T extends Comparable<T> & Serializable>` (clase primero, interfaces después).
En métodos: `static <T extends Comparable<T>> T max(List<T> lista)`.

INTERMEDIO

JAVA

Genéricos

J119

RESPUESTA

- 1) Campos static NO pueden usar `T` de la clase (`T` existe por instancia).
- 2) No se pueden crear arrays genéricos: `new T[10]`, `new List<String>[10]` (erasure no puede chequear; workaround: `(T[]) new Object[10]` con `@SuppressWarnings`).
- 3) No instanceof con tipos parametrizados (solo `List<?>` o clase cruda).

BASICO

JAVA

Genéricos

J121

@SuppressWarnings("unchecked"): cuándo es legítimo

INTERMEDIO

JAVA

Genéricos

J122

¿Qué es el type erasure y sus consecuencias?

AVANZADO

JAVA

Genéricos

J123

Wildcard ? extends vs ? super (regla PECS)

AVANZADO

JAVA

Genéricos

J124

¿Qué es el wildcard capture y el helper idiom?

AVANZADO

JAVA★

Excepciones

J125

Checked vs unchecked exceptions

BASICO

JAVA★

Excepciones

J126

¿Qué hace try-with-resources?

BASICO

JAVA★

Excepciones

J127

Jerarquía completa de Throwable

BASICO

JAVA

Excepciones

J128

throw vs throws vs Throwable

BASICO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

Genéricos

J122

RESPUESTA

El compilador BORRA los parámetros de tipo: `List<String>` y `List<Integer>` son la misma clase en runtime (`List`).
 Consecuencias: NO hay `new T()`, `T.class`, ni `instanceof List<String>`, ni arrays genéricos `new T[]`; los genéricos no participan en sobrecarga (`f(String)` vs `f(Integer)` ok, pero `f(List<String>)` vs `f(List<Integer>)` NO).
 Los tipos acotados se reemplazan por su bound y se insertan casts automáticos.

AVANZADO

JAVA

Genéricos

J121

RESPUESTA

Solo cuando TÚ garantizas la seguridad de tipo que el compilador no puede probar (casts de erasure, APIs legadas).
 Anotar la variable/método MÍNIMO posible, con comentario de por qué es seguro.
 Si se suprime a nivel de clase, se esconden bugs reales.

INTERMEDIO

JAVA

Genéricos

J124

RESPUESTA

El compilador trata cada uso de `?` como un tipo desconocido ÚNICO que no puede nombrarse: no puedes `lista.set(0, lista.get(0))` con `List<?>`.
 Truco: helper genérico privado `<T> void swapHelper(List<T> l){...}` y llamarlo desde el método con wildcard — el compilador captura el wildcard como `T` real.

AVANZADO

JAVA

Genéricos

J123

RESPUESTA

PECS: Producer Extends, Consumer Super.
`? extends T`: productor — LEES elementos como `T`, NO puedes agregar (solo null): `List<? extends Number>` acepta `List<Integer>`.
`? super T`: consumidor — ESCRIBES `T`, lo que lees es `Object`.
 Si solo lees: `extends`; si solo escribes: `super`; si ambos: tipo exacto.

AVANZADO

JAVA ★

Excepciones

J126

RESPUESTA

Cierra automáticamente cualquier recurso que implemente `AutoCloseable` al salir del bloque, incluso si hay excepción.
 Elimina el finally con `close()`.
 Si fallan el bloque y el `close()`, la del `close()` viaja como 'suppressed' adjunta a la principal.

BASICO

JAVA ★

Excepciones

J125

RESPUESTA

Checked (heredan de `Exception`): el compilador obliga a capturarlas o declararlas con `throws` — `IOException`, `SQLException`.
 Unchecked (heredan de `RuntimeException`): no obliga — `NullPointerException`, `IllegalArgumentException`.
 Error: fallos fatales de JVM (`OutOfMemoryError`).

BASICO

JAVA

Excepciones

J128

RESPUESTA

`throw`: sentencia que LANZA una instancia (`throw new IllegalArgumentException("...")`).
`throws`: en la firma, DECLARA que el método puede lanzar ciertas checked (puede listar unchecked pero es ruido).
`Throwable`: la superclase (solo se puede lanzar `Throwable` y subclases).

BASICO

JAVA ★

Excepciones

J127

RESPUESTA

`Throwable` → { `Error`, `Exception` }.
 Error: JVM irrecuperable (`OutOfMemoryError`, `StackOverflowError`) — no capturar.
`Exception` → `RuntimeException` (unchecked: `NPE`, `IllegalArgumentException`, `IndexOutOfBoundsException`) y las demás (checked: `IOException`, `SQLException`, `InterruptedException`).
 Custom: extender `Exception` (checked) o `RuntimeException` según semántica.

BASICO

JAVA

Excepciones

J129

¿Cuándo crear una excepción custom y con qué firma?

INTERMEDIO

JAVA

Excepciones

J130

Orden de catch y multi-catch

INTERMEDIO

JAVA

Excepciones

J131

¿Cuándo NO se ejecuta el finally?

INTERMEDIO

JAVA

Excepciones

J132

Anti-patrones clásicos de manejo de excepciones

INTERMEDIO

JAVA

Excepciones

J133

InterruptedException: qué significa y cómo tratarla

INTERMEDIO

JAVA★

Excepciones

J134

try-with-resources: reglas completas

INTERMEDIO

JAVA

Excepciones

J135

Rethrow y pérdida de stack trace

INTERMEDIO

JAVA

Excepciones

J136

assert: qué es y por qué no se usa en producción

INTERMEDIO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

Excepciones

J130

RESPUESTA

Los catch van de MÁS específico a MÁS general; catch de Exception antes que IOException no compila (bloque inalcanzable).

Multi-catch (Java 7+): catch (IOException | SQLException e) — e es implícitamente final; no pueden ser padre/hijo entre sí.

INTERMEDIO

JAVA

Excepciones

J129

RESPUESTA

Cuando el error tiene semántica de dominio (SaldoInsuficienteException).

Extender RuntimeException para no infectar firmas (regla moderna) o Exception si el llamador DEBE manejarla.

Constructor estándar: (String mensaje), (String mensaje, Throwable causa) — SIEMPRE preservar la causa para no perder el stack original.

INTERMEDIO

JAVA

Excepciones

J132

RESPUESTA

1) catch (Exception e) {} — tragarse el error (logging y rethrow o manejar).

2) catch(Exception e){ throw new RuntimeException(e.getMessage()) } — perder la causa (usar new RuntimeException(msg, e)).

3) Usar excepciones para control de flujo (costosas: llenan stack trace).

4) catch Throwable/Error.

5) Mensajes sin contexto ('error').

INTERMEDIO

JAVA

Excepciones

J131

RESPUESTA

Casi nunca: ni con return ni con excepción se salta.

Solo se pierde si: System.exit(), crash de la JVM, hilo daemon interrumpido al cerrar la JVM, o loop infinito dentro del try.

Trampa legada: un return dentro del finally SILENCIOSAMENTE descarta la excepción o el return del try — prohibido.

INTERMEDIO

JAVA ★

Excepciones

J134

RESPUESTA

Recursos: objetos AutoCloseable, separados por ';' (se cierran en orden INVERSO).

La variable es effectively final; se pueden declarar fuera y usar 'final var' (Java 9+).

Funciona también con try(recurso) sin catch.

Patrón: try (var conn = ...; var st = conn.createStatement()) {...}

INTERMEDIO

JAVA

Excepciones

J133

RESPUESTA

Se lanza en sleep/wait/join cuando el hilo es interrumpido (interrupt()).

Es CHECKED: fuerza a decidir.

Si no la manejas: restaurar la interrupción

(Thread.currentThread().interrupt()) y salir, para no 'tragarte' la señal de cancelación.

Nunca catch vacío.

INTERMEDIO

JAVA

Excepciones

J136

RESPUESTA

assert cond : "mensaje" — verificación de invariantes de desarrollo.

Se evalúa SOLO si la JVM corre con -ea (enableassertions); por defecto están DESACTIVADOS → en producción no protegen nada.

Para validación real:

IllegalArgumentException/Objects.requireNonNull (siempre activos).

INTERMEDIO

JAVA

Excepciones

J135

RESPUESTA

throw e; re-lanza la MISMA excepción conservando stack original.

throw new MiExcepcion(msg) sin causa: pierdes la historia → SIEMPRE wrap con causa: new MiExcepcion("contexto", e).

fillInStackTrace()/StackWalker para casos especiales.

INTERMEDIO

JAVA

Excepciones

J137

Objects utility para validar argumentos temprano

INTERMEDIO

JAVA

Excepciones

J138

Excepciones suprimidas (suppressed)

AVANZADO

JAVA

Excepciones

J139

¿Lanzar excepciones es caro? ¿Por qué?

AVANZADO

JAVA

Streams y lambdas

J140

Sintaxis de lambda: todas las variantes

BASICO

JAVA

Streams y lambdas

J141

Las 4 interfaces funcionales core y para qué sirven

BASICO

JAVA

Streams y lambdas

J142

Method references: las 4 formas

BASICO

JAVA

Streams y lambdas

J143

¿Qué es un Stream y en qué se diferencia de una colección?

BASICO

JAVA

Streams y lambdas

J144

Operaciones intermedias vs terminales y la pereza (lazy)

BASICO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

Excepciones

J138

RESPUESTA

En try-with-resources, si el try lanza X y el close() del recurso lanza Y: X se propaga y Y queda adjuntada.
 Recuperarlas: ex.getSuppressed() → Throwable[].
 No aplica al finally clásico (allí la excepción del finally REEMPLAZA a la original — se pierde).

AVANZADO

JAVA

Excepciones

J137

RESPUESTA

Objects.requireNonNull(x, "no puede ser null") → NPE con mensaje claro.
 Objects.requireNonNullElse(x, default).
 requireNonNull dentro de constructores/setters = fail-fast en el punto correcto (no un NPE críptico 10 líneas después).

INTERMEDIO

JAVA

Streams y lambdas

J140

RESPUESTA

(int x) -> x*1, (x) -> x*1, x -> x*1, () -> 42, (a, b) -> { int s = a+b; return s; }.
 Tipo inferido del contexto (interfaz funcional destino).
 Una expresión = return implícito; con {} se necesita return explícito.
 Variables capturadas: effectively final.

BASICO

JAVA

Excepciones

J139

RESPUESTA

Sí: se construye el stack trace (recorre la pila) en el constructor, aunque nadie lea el stack.
 En hot paths de validación frecuente (parsing masivo) pesa.
 Alternativa: resultados tipo Optional/Either o validación previa.
 Para errores REALES, el costo es irrelevante frente a la claridad.

AVANZADO

JAVA

Streams y lambdas

J142

RESPUESTA

1) Estático: Math::abs
 2) Instancia concreta: System.out::println
 3) Instancia del primer parámetro: String::toUpperCase (equivalente a s.toUpperCase())
 4) Constructor: ArrayList::new
 Pasa cualquier firma compatible con la interfaz funcional.

BASICO

JAVA

Streams y lambdas

J141

RESPUESTA

Function<T,R>: T→R (map). Predicate<T>: T→boolean (filter/test).
 Consumer<T>: T→void (efecto, forEach/accept). Supplier<T>: ()→T (produce, get).
 Variantes: BiFunction, BiPredicate, UnaryOperator (T→T), BinaryOperator (T,T→T), especializadas IntPredicate, ToIntFunction (evitan boxing).

BASICO

JAVA

Streams y lambdas

J144

RESPUESTA

Intermedias (devuelven Stream, perezosas): filter, map, flatMap, sorted, distinct, limit, skip, peek, takeWhile/dropWhile.
 Terminales (disparan el trabajo): collect, forEach, reduce, count, anyMatch/allMatch/noneMatch, findFirst/findAny, toArray, min/max.
 Sin terminal, NADA se ejecuta (fuente clásica de 'mi stream no hace nada').

BASICO

JAVA

Streams y lambdas

J143

RESPUESTA

Secuencia de elementos que NO almacena datos: procesa una fuente (colección, array, generador) de forma declarativa, perezosa y (opcionalmente) paralela.
 Colección = estructura de datos; Stream = pipeline de operaciones.
 Un stream se consume UNA vez; reutilizarlo lanza IllegalStateException.

BASICO

JAVA

Streams y lambdas

J145

map vs flatMap: la diferencia que siempre preguntan

BASICO

JAVA

Streams y lambdas

J146

reduce: las 3 formas y sus riesgos

BASICO

JAVA

Streams y lambdas

J147

Collectors que hay que dominar

BASICO

JAVA

Streams y lambdas

J148

groupBy: los niveles que salen en exámenes

BASICO

JAVA

Streams y lambdas

J149

Streams de primitivas: IntStream y amigos

BASICO

JAVA

Streams y lambdas

J150

Short-circuit: anyMatch, findFirst y streams infinitos

BASICO

JAVA

Streams y lambdas

J151

Optional en streams y su uso correcto

BASICO

JAVA

Streams y lambdas

J152

sorted(), distinct() y sus costos ocultos

BASICO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA
RESPUESTA

Streams y lambdas

J146

1) Optional<T> reduce(accumBinaryOp)
 2) T reduce(identity, accum)
 3) <U> U reduce(identity, accum, combiner) (paralelo).
 Riesgos: identity NO neutro rompe paralelismo (usar 0 para suma, no -1); accum asociativo para paralelo.
 Para colecciones mejor Collectors.

BASICO

JAVA
RESPUESTA

Streams y lambdas

J145

map: 1 elemento → 1 elemento transformado (Stream<R>).
 flatMap: 1 elemento → MUCHOS elementos aplanados: lista de listas → lista plana (Stream<R> desde Streams internos).
 Ejemplo: pedidos.stream().flatMap(p -> p.getItems().stream()) = todos los ítems.

BASICO

JAVA
RESPUESTA

Streams y lambdas

J148

1 nivel: groupingBy(Persona::getCiudad) → Map<Ciudad, List<Persona>>.
 Con downstream: groupingBy(Persona::getCiudad, counting()) → Map<Ciudad, Long>.
 Con mapping: groupingBy(c, mapping(Persona::getNombre, toList())).
 Anidado: groupingBy(a, groupingBy(b, counting())) → conteo por dos dimensiones.
 partitioningBy es el caso booleano → Map<Boolean, List>.

BASICO

JAVA
RESPUESTA

Streams y lambdas

J147

toList/toSet/toCollection, joining(delimitador), counting, summingInt/averagingInt, summarizingInt (stats), groupingBy (y con downstream), partitioningBy (boolean), toMap(keyFn, valFn[, mergeFn, mapSupplier]), mapping, filtering, collectingAndThen, teeing (dos recolectores + merge).
 toMap SIN mergeFn lanza IllegalStateException con claves duplicadas.

BASICO

JAVA
RESPUESTA

Streams y lambdas

J150

anyMatch/allMatch/noneMatch, findFirst/findAny, limit — cortan el procesamiento cuando ya tienen respuesta.
 Permite streams infinitos: Stream.iterate(1, i -> i*2).limit(10), Stream.generate(supplier).
 Java 9+: iterate con predicado de parada, Stream.ofNullable().forEach NO corta (usar takeWhile).

BASICO

JAVA
RESPUESTA

Streams y lambdas

J149

IntStream/LongStream/DoubleStream evitan boxing.
 Crear: IntStream.range(0,n) / rangeClosed, lista.stream().mapToInt(String::length).
 Extras: sum(), average(), summaryStatistics(), boxed() para volver a Stream<Integer>.
 map vs mapToInt: mapToObj, mapToDouble para cruzar.

BASICO

JAVA
RESPUESTA

Streams y lambdas

J152

sorted: estado con todo el stream en memoria + ordena O(n log n) — no es perezoso de verdad.
 distinct: mantiene un HashSet interno (necesita equals/hashCode).
 sorted después de un filter reduce datos; Comparator.comparing natural.
 En paralelo sorted es aún más caro.

BASICO

JAVA
RESPUESTA

Streams y lambdas

J151

findFirst/findAny/min/max/reduce devuelven Optional.
 Encadenar: stream.findFirst().map(String::toUpperCase).orElse("-").
 orElse vs orElseGet: orElse EVALÚA SIEMPRE el argumento (cuidado con llamadas costosas); orElseGet es perezoso.
 orElseThrow(supplier) para error explícito.

BASICO

JAVA

Streams y lambdas

J153

¿Cuándo NO usar streams?

BASICO

JAVA

Streams y lambdas

J154

Lambda vs clase anónima: diferencias finas

BASICO

JAVA

Streams y lambdas

J155

Comparador moderno con lambdas

BASICO

JAVA★

Streams y lambdas

J156

¿Qué son los Streams de Java 8?

INTERMEDIO

JAVA★

Streams y lambdas

J157

¿Para qué sirve Optional?

INTERMEDIO

JAVA

Streams y lambdas

J158

peek(), parallelStream() y sus trampas

AVANZADO

JAVA

Streams y lambdas

J159

Collectors.teeing y collectingAndThen (nivel pro)

AVANZADO

JAVA★

Concurrencia

J160

volatile vs synchronized

INTERMEDIO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

Streams y lambdas

J154

RESPUESTA

Lambda: solo interfaces funcionales, this se refiere a la clase ENVOLVENTE (no a la anónima), no crea una clase nueva por uso (invokeDynamic).

Anónima: sirve para clases/abstractas, tiene su propio this, puede tener estado/campos.

Para implementar interfaces funcionales simples, lambda siempre.

BASICO

JAVA

Streams y lambdas

J153

RESPUESTA

Lógica con múltiples efectos/branches complejos (mejor loop claro).

Mutación de estado externo dentro del pipeline (violación + bugs en paralelo).

Solo un par de operaciones triviales donde el loop es igual de claro.

Rendimiento crítico con elementos primitivos y acceso indexado (arrays + loops).

Legibilidad primero.

BASICO

JAVA ★

Streams y lambdas

J156

RESPUESTA

Pipeline funcional sobre colecciones:

fuelle → intermedias PEREZOSAS (filter, map, sorted) →

TERMINAL (collect, forEach, reduce) que dispara la evaluación.

No mutan la fuente.

parallelStream() para procesar en paralelo (cuidado con orden y threads).

INTERMEDIO

JAVA

Streams y lambdas

J155

RESPUESTA

Comparator.comparing(P::getNombre) /
.comparingInt(P::getEdad), thenComparing, reversed(),
nullsFirst/nullsLast(Comparator.naturalOrder()).

En streams: sorted(Comparator.comparing(...)).

max/min de stream aceptan comparator directamente.

BASICO

JAVA

Streams y lambdas

J158

RESPUESTA

peek: pensado para DEBUG (ver elementos que pasan); NO garantiza ejecución salvo terminal, y abusar de él para mutar es frágil.

parallelStream: usa ForkJoinPool.commonPool — SOLO para fuentes fácilmente dividibles y operaciones SIN estado, asociativas, sin efectos secundarios; NO para I/O; puede degradar (compartido en toda la JVM).

findAny/forEach en paralelo no preservan orden; usar
forEachOrdered.

AVANZADO

JAVA ★

Streams y lambdas

J157

RESPUESTA

Contenedor que expresa 'puede no haber valor' en el tipo de retorno; evita null y NPE.

Métodos: map, filter, orElse / orElseGet, orElseThrow, ifPresent.

Antipatrón: Optional como campo o parámetro.

Nunca .get() sin comprobar.

INTERMEDIO

JAVA ★

Concurrencia

J160

RESPUESTA

volatile: garantiza VISIBILIDAD (las escrituras de un hilo se ven al instante) pero NO atomicidad — i++ sigue roto.

synchronized: exclusión mutua + visibilidad mediante lock de monitor; bloquea un método o bloque.

INTERMEDIO

JAVA

Streams y lambdas

J159

RESPUESTA

teeing(f1, f2, merge): combina dos recolecciones sobre el mismo stream — ej. máx y mín juntos, o porcentaje: teeing(counting(), summingInt(...), (n, s) -> 100.0*s/n).

collectingAndThen(collector, finisher): transforma el resultado final — ej. toList → List.copyOf, o toSet → TreeSet::new.

AVANZADO

JAVA★

Concurrencia

J161

¿Qué es ExecutorService?

INTERMEDIO

JAVA★

Concurrencia

J162

¿Qué añade CompletableFuture sobre Future?

INTERMEDIO

JAVA

Concurrencia

J163

Ciclo de vida de un hilo en Java

INTERMEDIO

JAVA

Concurrencia

J164

Race condition: qué es y el ejemplo de siempre

INTERMEDIO

JAVA

Concurrencia

J165

synchronized: método vs bloque y sobre QUÉ se bloquea

INTERMEDIO

JAVA

Concurrencia

J166

ReentrantLock vs synchronized: cuándo el extra vale la pena

INTERMEDIO

JAVA

Concurrencia

J167

AtomicInteger y CAS: cómo funcionan sin locks

INTERMEDIO

JAVA

Concurrencia

J168

CountDownLatch vs CyclicBarrier vs Semaphore

INTERMEDIO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA ★
RESPUESTA

Concurrencia

J162

Composición NO bloqueante:

- thenApply / thenCompose / thenCombine encadenan etapas
- callbacks sin .get() bloqueante
- errores con exceptionally / handle
- combina tareas paralelas

Es la base de la programación asíncrona en Java moderno.

INTERMEDIO

JAVA ★
RESPUESTA

Concurrencia

J161

Pool de hilos que ejecuta tareas sin crear hilos a mano.

Envías Callable/Runnable y recibes Future con el resultado.

Fábricas: newFixedThreadPool, newCachedThreadPool, newScheduledThreadPool.

Siempre cerrarlo: shutdown() / shutdownNow().

INTERMEDIO

JAVA
RESPUESTA

Concurrencia

J164

El resultado depende del entrelazado de hilos. Clásico:

contador++ no es atómico (leer-modificar-escribir: 3 pasos) → dos hilos pueden leer el mismo valor y perder una suma.

Fixes: synchronized, AtomicInteger (CAS), LongAdder para contadores de alta contención.

No confundir con deadlock.

INTERMEDIO

JAVA
RESPUESTA

Concurrencia

J163

NEW → (start) RUNNABLE → (corriendo / listo) → BLOCKED (esperando lock) → WAITING (wait/join sin timeout) →

TIMED_WAITING (sleep/wait con timeout) → TERMINATED.

No se puede reiniciar un hilo terminado

(IllegalThreadStateException): usa run() en lugar de start() NO crea hilo.

INTERMEDIO

JAVA
RESPUESTA

Concurrencia

J166

ReentrantLock: tryLock(timeout) (no bloquear indefinidamente), lockInterruptibly(), fairness (cola FIFO), múltiples Condition.

Costo: debes hacer unlock() en finally SIEMPRE.

Regla: synchronized por defecto; ReentrantLock solo para esas necesidades avanzadas.

INTERMEDIO

JAVA
RESPUESTA

Concurrencia

J165

Método de instancia: lock del objeto (this). Método static: lock del objeto Class (Clase.class).

Bloque: synchronized(otroObjeto) { } — lock específico, mínimo alcance recomendado.

Reentrante: el mismo hilo puede tomar el lock de nuevo.

Todo lo compartido entre hilos debe protegerse con el MISMO lock.

INTERMEDIO

JAVA
RESPUESTA

Concurrencia

J168

CountDownLatch: cuenta regresiva de UNA VEZ (await espera a que count llegue a 0) — 'espera a que N tareas terminen'; no reutilizable.

CyclicBarrier: N hilos se ESPERAN MUTUAMENTE en un punto; reutilizable; acción barrier opcional.

Semaphore(N): permisos — limita acceso concurrente a un recurso (pool de conexiones); acquire/release.

INTERMEDIO

JAVA
RESPUESTA

Concurrencia

J167

Compare-And-Swap: operación atómica de hardware — 'si el valor actual es X, ponme Y' — reintentando en loop si falló.

AtomicInteger.incrementAndGet() es lock-free y muy rápido.

Limitación: operaciones compuestas complejas (compareAndSet encadenado, acumulación con lógica → synchronized o LongAdder).

Otros: AtomicReference, AtomicBoolean, AtomicLong.

INTERMEDIO

JAVA★

Concurrencia

J169

BlockingQueue: el patrón productor-consumidor en 3 líneas

INTERMEDIO

JAVA

Concurrencia

J170

ScheduledExecutorService: tareas periódicas correctas

INTERMEDIO

JAVA

Concurrencia

J171

¿Qué es la contención (contention) y cómo se reduce?

INTERMEDIO

JAVA

Concurrencia

J172

@Async en Spring vs CompletableFuture manual

INTERMEDIO

JAVA

Concurrencia

J173

Crear hilos: Runnable vs Thread vs Callable

AVANZADO

JAVA★

Concurrencia

J174

wait/notify/notifyAll: contrato exacto

AVANZADO

JAVA

Concurrencia

J175

Deadlock: las 4 condiciones y cómo prevenirlo

AVANZADO

JAVA

Concurrencia

J176

LongAdder vs AtomicLong

AVANZADO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA
RESPUESTA

Concurrencia

J170

`scheduleAtFixedRate(tarea, ini, periodo)`: intenta ejecutar cada 'periodo' (fijo desde el inicio; si la tarea tarda más, se solapan las intenciones).

`scheduleWithFixedDelay(tarea, ini, delay)`: espera 'delay' ENTRE fin e inicio (nunca solapa).

Si la tarea LANZA excepción, las siguientes se CANCELAN → `try/catch` interno obligatorio.

INTERMEDIO

JAVA ★
RESPUESTA

Concurrencia

J169

`ArrayBlockingQueue<Integer> cola = new ArrayBlockingQueue<>(100);`
 Productor: `cola.put(item)` — bloquea si llena. Consumidor: `cola.take()` — bloquea si vacía.
 Desacopla velocidades, acota memoria (bounded), sin wait/notify manual.
 Variantes: `LinkedBlockingQueue`, `PriorityBlockingQueue`, `SynchronousQueue` (entrega directa).

INTERMEDIO

JAVA
RESPUESTA

Concurrencia

J172

`@Async`: Spring envuelve el método con un proxy y lo ejecuta en un executor configurado (`TaskExecutor`) — necesidad: `@EnableAsync` y NO self-invocation (mismo límite que `@Transactional`).

`CompletableFuture`: control fino de composición y executor por llamada.

Ambos devuelven `Future/CompletableFuture` si se quiere el resultado. Nunca anotar métodos `private`.

INTERMEDIO

JAVA
RESPUESTA

Concurrencia

J171

Tiempo que los hilos gastan esperando locks/recursos compartidos.

Reducir: scopes más cortos de `synchronized`, locks finos/por-striping (`ConcurrentHashMap`), estructuras lock-free (CAS), datos por hilo (`ThreadLocal`), inmutabilidad, `LongAdder`, particionar el estado.

Medir: profilers de contención, JFR.

INTERMEDIO

JAVA ★
RESPUESTA

Concurrencia

J174

Se llaman DENTRO de `synchronized` sobre el MISMO objeto monitor, sino `IllegalMonitorStateException`.

`wait()`: libera el lock y espera; `notify()`: despierta UNO; `notifyAll()`: despierta todos (el más seguro — evita hilos dormidos para siempre).

Siempre en un `while(!condicion) wait()` — para despertares espurios.

Hoy se prefiere `java.util.concurrent` (`BlockingQueue`, `CountDownLatch`).

AVANZADO

JAVA
RESPUESTA

Concurrencia

J173

`Runnable`: tarea sin resultado, `new Thread(runnable).start()`.

`Thread`: heredar (acopla tarea con ejecución — evitar).

`Callable<V>`: devuelve resultado y lanza excepciones checked — se usa con `ExecutorService/Future`.

Recomendado: nunca crear hilos 'a mano'; usar pools (executors).

AVANZADO

JAVA
RESPUESTA

Concurrencia

J176

`LongAdder`: mantiene celdas múltiples que se suman al leer (`sum()`) — muchísimo mejor bajo ALTA CONTENCIÓN de escritura; lecturas ligeramente más caras.

`AtomicLong`: todos los hilos CAS sobre un solo valor → colisiones.

Contadores de métricas → `LongAdder`.

AVANZADO

JAVA
RESPUESTA

Concurrencia

J175

Exclusión mutua, retención y espera, no expulsión, espera circular.

Prevención: adquirir locks en ORDEN GLOBAL consistente, usar `tryLock` con timeout (`ReentrantLock`), evitar locks anidados, granularidad fina.

Detección: dump de hilos (jstack) muestra 'Found one Java-level deadlock'.

AVANZADO

JAVA

Concurrencia

J177

ExecutorService: ciclo de vida correcto

AVANZADO

JAVA★

Concurrencia

J178

¿Cómo dimensionar un pool de hilos?

AVANZADO

JAVA

Concurrencia

J179

Future: qué ofrece y qué le falta

AVANZADO

JAVA

Concurrencia

J180

CompletableFuture: recetario de métodos

AVANZADO

JAVA

Concurrencia

J181

ThreadLocal: qué es y su fuga de memoria clásica

AVANZADO

JAVA

Concurrencia

J182

interrupt(): qué hace y qué no

AVANZADO

JAVA

Concurrencia

J183

Hilos daemon vs usuario

AVANZADO

JAVA

Concurrencia

J184

Colecciones thread-safe: mapa rápido de decisión

AVANZADO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA ★
RESPUESTA

Concurrencia

J178

CPU-bound: núcleos + 1
(Runtime.getRuntime().availableProcessors()).
I/O-bound: más hilos porque pasan bloqueados: núcleos * (1 + tiempoEspera/tiempoCPU) o empíricamente decenas.
En servicios web: cada pool por dependencia (bulkhead) con colas ACOTADAS (LinkedBlockingQueue sin límite = OOM).
Medir y ajustar.

AVANZADO

JAVA
RESPUESTA

Concurrencia

J177

Crear: Executors.newFixedThreadPool(n), newCachedThreadPool (cuidado: hilos ilimitados), newSingleThreadExecutor, newScheduledThreadPool.
Ejecutar: submit (Callable → Future), execute (Runnable).
Cerrar: shutdown() (termina tareas pendientes, no acepta nuevas) → awaitTermination → shutdownNow() (interrumpe) en finally.
No cerrar = hilos no-daemon impiden salir de la JVM.

AVANZADO

JAVA
RESPUESTA

Concurrencia

J180

supplyAsync(tarea, executor) / runAsync.
Transformar: thenApply (sync), thenApplyAsync.
Encadenar: thenCompose (evita CF<CF<T>>), thenCombine (une dos independientes).
Consumir: thenAccept/thenRun.
Errores: exceptionally, handle, whenComplete.
Todos los *Async aceptan executor propio (¡sino commonPool!).

AVANZADO

JAVA
RESPUESTA

Concurrencia

J179

get() (bloquea, con timeout opcional),
cancel(mayInterruptIfRunning), isDone/isCancelled.
Falta: callbacks no bloqueantes, composición, manejo de errores elegante → por eso CompletableFuture.
Trampa: get() sin timeout puede colgar para siempre.

AVANZADO

JAVA
RESPUESTA

Concurrencia

J182

NO detiene el hilo: pone un flag y hace que sleep/wait/join/bloqueos interrumpibles lancen InterruptedException.
El hilo decide cooperar: revisar
Thread.currentThread().isInterrupted() en loops largos.
Limpiar el flag al capturar la excepción si vas a continuar:
Thread.currentThread().interrupt().

AVANZADO

JAVA
RESPUESTA

Concurrencia

J181

Variable con valor POR HILO (cada hilo ve su copia): user context, transacciones, SimpleDateFormat legado.
Uso: ThreadLocal.withInitial() -> new ...).
Fuga: en pools de hilos los hilos viven para siempre → si no haces remove() en el finally, el valor queda anclado para siempre.
Siempre try { } finally { threadLocal.remove(); }.

AVANZADO

JAVA
RESPUESTA

Concurrencia

J184

Map concurrente → ConcurrentHashMap (o ConcurrentSkipListMap si ordenado).
Lista concurrente → CopyOnWriteArrayList (muchas lecturas) o listas sincronizadas pequeñas.
Cola productor/consumidor → ArrayBlockingQueue (acotada) / LinkedBlockingQueue / ConcurrentLinkedQueue (lock-free).
Set concurrente → ConcurrentHashMap.newKeySet().
NUNCA Vector/Hashtable para código nuevo.

AVANZADO

JAVA
RESPUESTA

Concurrencia

J183

Daemon: hilo de servicio (GC, monitores) — la JVM sale SIN esperarlos; setDaemon(true) ANTES de start().
Usuario: la JVM espera a que terminen.
Trampa: tareas críticas en daemon pueden cortarse a la mitad al cerrar la app.

AVANZADO

JAVA

Concurrencia

J185

¿Por qué 'double-checked locking' era difícil y cómo se arregla?

AVANZADO

JAVA

Concurrencia

J186

Happens-before: la idea central del Java Memory Model

AVANZADO

JAVA

Concurrencia

J187

¿Por qué volatile NO hace atómico i++?

AVANZADO

JAVA

Concurrencia

J188

Singleton thread-safe: las 4 formas canónicas

AVANZADO

JAVA

Concurrencia

J189

ForkJoinPool y work-stealing: para qué existe

AVANZADO

JAVA

Concurrencia

J190

Virtual threads (Java 21): qué cambian de verdad

AVANZADO

JAVA

Concurrencia

J191

¿Qué es el 'pinning' de virtual threads?

AVANZADO

JAVA

Concurrencia

J192

Inmutabilidad como estrategia de concurrencia

AVANZADO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

Concurrencia

J186

RESPUESTA

Garantía de VISIBILIDAD: si la acción A happens-before B, todo lo que A escribió es visible para B.

Construyen happens-before: release/acquire del mismo lock (unlock→lock), escritura/lectura volatile, start()/join() de un hilo, métodos de concurrencia del paquete j.u.c.

Sin una de estas relaciones, los cambios pueden no verse NUNCA (reordenamiento + cachés de CPU).

AVANZADO

JAVA

Concurrencia

J185

RESPUESTA

Singleton lazy: sin volatile, un hilo puede ver un objeto a medio construir (reordenamiento de instrucciones).

Fix: campo private static volatile Instancia instancia.

Alternativas mejores: holder idiom (clase interna estática — lazy y thread-safe por classloading) o enum singleton.

AVANZADO

JAVA

Concurrencia

J188

RESPUESTA

- 1) Enum singleton (SimpleSingleton.INSTANCE) — la más segura.
- 2) Holder idiom: clase interna estática con la instancia (lazy por classloading, sin locks).
- 3) Double-checked locking con volatile.
- 4) Campo static final inicializado al cargar (eager).

En Spring, el contenedor ya maneja singletons — casi no se escribe a mano.

AVANZADO

JAVA

Concurrencia

J187

RESPUESTA

volatile garantiza visibilidad y orden, no atomicidad compuesta: i++ sigue siendo leer+sumar+escribir (3 pasos que pueden entrelazarse).

Fix: AtomicInteger.incrementAndGet(), o synchronized.

volatile perfecto para: flags de estado (running = false), publicación de referencias inmutables.

AVANZADO

JAVA

Concurrencia

J190

RESPUESTA

Hilos ligeros gestionados por la JVM (no 1:1 con hilos del SO): millones posibles.

Para I/O-bound: Thread.ofVirtual().start() o Executors.newVirtualThreadPerTaskExecutor(); el hilo se desmonta al bloquearse en I/O → concurrencia masiva con código SÍNCRONO simple (no callbacks).

NO acelera CPU-bound. Evitar pooling de virtual threads (uno por tarea) y sincronización prolongada (pinning).

AVANZADO

JAVA

Concurrencia

J189

RESPUESTA

Pool que DIVIDE tareas recursivamente (fork/join) y los hilos ROBAN trabajo de las colas de otros (work-stealing) → buen balanceo para divide & conquer (procesar árboles, parallel streams lo usan).

RecursiveTask<V>/RecursiveAction; invoke()/compute().

Es el commonPool por defecto de los parallel streams.

AVANZADO

JAVA

Concurrencia

J192

RESPUESTA

Objetos inmutables NO necesitan locks: compartirlos es gratis y seguro (publicación segura vía volatile/final).

Diseño: state en records inmutables, cambios = nueva instancia (copy-on-write mentalidad).

Es la base de ConcurrentHashMap, CopyOnWrite*, akka/actores y del estilo funcional.

AVANZADO

JAVA

Concurrencia

J191

RESPUESTA

Un virtual thread montado sobre un carrier no puede desmontarse si ejecuta synchronized prolongado o native (JNI) bloqueante → bloquea su hilo del SO (¡el recurso escaso que querías ahorrar!).

Java 21+ y 24 han mitigado mucho el caso synchronized.

Workaround histórico: ReentrantLock en lugar de synchronized en hot paths de I/O.

AVANZADO

JAVA★

JVM y rendimiento

J193

¿Diferencia entre JDK, JRE y JVM?

BASICO

JAVA

JVM y rendimiento

J194

¿Qué hace la JVM al ejecutar tu .class?

BASICO

JAVA★

JVM y rendimiento

J195

Estructura de memoria de la JVM

BASICO

JAVA★

JVM y rendimiento

J196

¿Cómo funciona el Garbage Collector?

INTERMEDIO

JAVA★

JVM y rendimiento

J197

Stack vs Heap

INTERMEDIO

JAVA

JVM y rendimiento

J198

G1 GC: cómo funciona en 5 puntos

AVANZADO

JAVA

JVM y rendimiento

J199

ZGC / Shenandoah vs G1: ¿cuándo low-latency?

AVANZADO

JAVA★

JVM y rendimiento

J200

OutOfMemoryError: los tipos y qué significan

AVANZADO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

JVM y rendimiento

J194

RESPUESTA

- 1) ClassLoader carga el bytecode (loading, linking —verify/prepare/resolve—, initializing).
 - 2) El intérprete ejecuta bytecode línea a línea.
 - 3) El compilador JIT (C1 cliente, C2 optimizador) detecta hotspots y compila a código máquina nativo con optimizaciones (inlining, escape analysis, loop unrolling).
 - 4) El GC gestiona el heap.
- Resultado: arranque lento, luego muy rápido.

BASICO

JAVA



JVM y rendimiento

J193

RESPUESTA

JVM: ejecuta el bytecode (.class) y da independencia de plataforma.
 JRE = JVM + librerías para EJECUTAR programas.
 JDK = JRE + herramientas para DESARROLLAR (javac, javadoc, jdb).
 Desarrollas con JDK; ejecutas con JRE.

BASICO

JAVA



JVM y rendimiento

J196

RESPUESTA

Rastrea objetos alcanzables desde las raíces (hilos, static); lo no alcanzable se elimina del heap.
 Heap generacional: Young (Eden + Survivor, GCs menores) y Old (GC mayor).
 System.gc() solo es una sugerencia: no garantiza nada.

INTERMEDIO

JAVA



JVM y rendimiento

J195

RESPUESTA

Heap: objetos — Young (Eden + 2 Survivor) y Old/Tenured.
 Metaspace (Java 8+): metadatos de clases (reemplaza PermGen), en memoria nativa.
 Stacks: uno por hilo (frames, variables locales).
 Code cache: JIT nativo.
 Además: direct buffers, thread stacks.
 Flags: -Xms/-Xmx (heap), -XX:MaxMetaspaceSize, -Xss (stack).

BASICO

JAVA

JVM y rendimiento

J198

RESPUESTA

- GC por defecto desde Java 9.
- 1) Divide el heap en regiones (~2048) iguales.
 - 2) Marca concurrente para saber vivos.
 - 3) Jóvenes: pausas cortas (copy entre regiones).
 - 4) Mixtas: recoge regiones viejas con más basura primero.
 - 5) Objetivo de pausa configurable: -XX:MaxGCPauseMillis=200.
- Compacta (sin fragmentación como CMS).

AVANZADO

JAVA



JVM y rendimiento

J197

RESPUESTA

Stack: por hilo, guarda frames de métodos y variables locales/primitivas (LIFO, muy rápido).
 Heap: compartido, guarda objetos y arrays; lo gestiona el GC.
 OutOfMemoryError → heap. StackOverflowError → recursión sin fin.

INTERMEDIO

JAVA



JVM y rendimiento

J200

RESPUESTA

Java heap space: heap lleno de objetos vivos (fuga o -Xmx chico).
 GC overhead limit exceeded: >98% de tiempo en GC recuperando <2%.
 Metaspace: demasiadas clases (generación dinámica, fuga de classloaders).
 Unable to create native thread: límite de hilos del SO.
 Direct buffer memory: NIO buffers fuera del heap.
 Diagnóstico: heap dump
 (-XX:+HeapDumpOnOutOfMemoryError) + Eclipse MAT.

AVANZADO

JAVA

RESPUESTA

JVM y rendimiento

J199

ZGC/Shenandoah: pausas < 1-10 ms INDEPENDIENTES del tamaño del heap (TB de heap), casi todo concurrente.
 Costo: algo más de CPU total (concurrente) y footprint.
 Para: APIs con SLO de latencia estricta, heaps enormes.
 G1: balance por defecto excelente.
 Activar: -XX:+UseZGC.

AVANZADO

JAVA

JVM y rendimiento

J201

Memory leak en Java (¡con GC!): las fuentes típicas

AVANZADO

JAVA

JVM y rendimiento

J202

Referencias fuertes, suaves, débiles y fantasmas

AVANZADO

JAVA

JVM y rendimiento

J203

Classloading: delegación padre-hijo

AVANZADO

JAVA

JVM y rendimiento

J204

¿Qué mide un profiler y cuáles hay en el kit JVM?

AVANZADO

JAVA

JVM y rendimiento

J205

Flame graphs: cómo leerlos en 30 segundos

AVANZADO

JAVA

JVM y rendimiento

J206

Autoboxing y rendimiento: el caso típico

AVANZADO

JAVA

JVM y rendimiento

J207

¿Qué es JMH y por qué System.nanoTime() no basta para benchmarks?

AVANZADO

JAVA

JVM y rendimiento

J208

Escape analysis y optimizaciones del JIT

AVANZADO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

JVM y rendimiento

J202

RESPUESTA

Strong: nunca se recolecta mientras sea alcanzable.
SoftReference: se limpia ANTES de OOM — cachés de memoria sensibles.
WeakReference: se limpia en el próximo GC si solo hay referencias débiles — WeakHashMap (metadatos asociados a objetos).
PhantomReference: notificación post-mortem (cleanup de nativo; reemplazado por Cleaner).

AVANZADO

JAVA

JVM y rendimiento

J201

RESPUESTA

Colecciones estáticas que solo crecen (cachés sin eviction), ThreadLocal sin remove en pools, listeners/handlers no desregistrados, classloaders con despliegues calientes (tomcat), objetos anclados por campos estáticos, subcadenas/arrays grandes retenidos.
Síntoma: heap sube escalonadamente y nunca baja; Full GCs frecuentes.
Herramienta: heap dump + dominators tree (MAT).

AVANZADO

JAVA

JVM y rendimiento

J204

RESPUESTA

CPU por método (hotspots), memoria/retención, hilos, I/O.
Herramientas: JFR (Java Flight Recorder) + JMC — bajo overhead, producción; jconsole/visualvm (local); async-profiler (flame graphs).
Metodología: medir ANTES de optimizar; regla 90/10 (pocos métodos concentran el tiempo).

AVANZADO

JAVA

JVM y rendimiento

J203

RESPUESTA

Jerarquía: Bootstrap → Platform → Application (→ custom).
Delegación: el loader pide AL PADRE primero; si no lo encuentra, carga él — evita duplicar clases core y asegura que java.lang.String sea 'el' String.
Carga perezosa: una clase se inicializa al PRIMER uso activo (new, acceso static).
Aplicaciones web contenedores usan loaders por app (aislamiento).

AVANZADO

JAVA

JVM y rendimiento

J206

RESPUESTA

En loops intensivos: List<Long> suma via boxed (cada operación crea objetos) vs long[] o IntStream.range().sum() (primitivos).
long total = 0; for (long x : array) total += x — gana por goleada.
Medir con JMH si es crítico; microoptimizar sin datos es adivinar.

AVANZADO

JAVA

JVM y rendimiento

J205

RESPUESTA

Cada caja = un método en ejecución; ancho = proporción de tiempo mostrado.
X axis: stack apilado (abajo = llamador); el ANCHO total es lo que importa.
Platachas anchas planas = hotspot claro.
En modo diff (antes/después) las mejoras se ven por colores.

AVANZADO

JAVA

JVM y rendimiento

J208

RESPUESTA

El JIT analiza si un objeto ESCAPA del método: si no, puede eliminar la asignación (scalar replacement), hacer inlining de métodos calientes y unrolling de loops.
Inlining es la reina: habilita todas las demás — métodos GRANDES o final/mega-mórficos (2 tipos reales máx) se inlinean mejor.
Consecuencia: abstracciones 'gratis' si JIT las elimina; megamorfismo mata el inlining.

AVANZADO

JAVA

JVM y rendimiento

J207

RESPUESTA

JMH (OpenJDK Microbenchmark Harness): maneja warmup de JIT, dead-code elimination, aislamiento de forks, distribución estadística.
nanoTime a mano mide el código optimizado-away o sin calentar (primeras ejecuciones lentas).
Uso: anotaciones @Benchmark, @State, fork, warmup — resultados en ops/time con percentiles.

AVANZADO

JAVA

JVM y rendimiento

J209

Flags JVM que todo backend debería conocer

AVANZADO

JAVA

JVM y rendimiento

J210

String deduplication y compact strings

AVANZADO

JAVA

JVM y rendimiento

J211

Cachés de CPU y localidad de datos: por qué ArrayList le gana a LinkedList

AVANZADO

JAVA★

Java moderno

J212

¿Qué es un record (Java 14+)?

BASICO

JAVA

Java moderno

J213

switch de expresión (Java 14+): qué aporta

BASICO

JAVA

Java moderno

J214

HttpClient (Java 11+): uso básico

BASICO

JAVA

Java moderno

J215

Nuevos métodos de colección (Java 8-11) que debes usar

BASICO

JAVA

Java moderno

J216

Unnamed variables y patterns (⌵) — Java 21/22

BASICO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA
RESPUESTA

JVM y rendimiento

J210

Compact strings (Java 9+): byte[] Latin-1 cuando basta → ~50% menos memoria en textos occidentales.
 String deduplication (G1, -XX:+UseStringDeduplication): GC comparte arrays de chars idénticos entre Strings (para apps con millones de Strings duplicados).
 Complementarias a intern().

AVANZADO

JAVA
RESPUESTA

JVM y rendimiento

J209

-Xms/-Xmx (heap inicial/máx, iguales en prod),
 -XX:MaxMetaspaceSize, -XX:+HeapDumpOnOutOfMemoryError
 -XX:HeapDumpPath, -XX:MaxGCPauseMillis (G1), -Xlog:gc* (logs GC), -XX:+UseStringDeduplication (G1), -agentlib:jwp (debug),
 -Dsystem.properties.
 Prod: habilitar JFR:
 -XX:StartFlightRecording=duration=60s,filename=...

AVANZADO

JAVA ★
RESPUESTA

Java moderno

J212

Clase inmutable de datos en una línea:
 record Point(int x, int y) {}
 Genera constructor, accesores (x(), y()), equals, hashCode y toString.
 No puede extender otras clases. Ideal para DTOs y valores.

BÁSICO

JAVA
RESPUESTA

JVM y rendimiento

J211

La CPU trae líneas de caché (64B) contiguas: recorrer un array es prefetch-friendly.
 LinkedList: cada nodo es un objeto suelto en el heap → cache misses en cada salto.
 Regla práctica: ArrayList casi siempre gana incluso en inserciones medias por la localidad.
 Datos contiguos = rendimiento (primitivos > objetos).

AVANZADO

JAVA
RESPUESTA

Java moderno

J214

HttpClient client = HttpClient.newHttpClient();
 HttpRequest req = HttpRequest.newBuilder(URI.create(url)).header("Accept", "application/json").GET().build();
 HttpResponse<String> resp = client.send(req, BodyHandlers.ofString()); — también sendAsync(CompletableFuture).
 Soporta HTTP/2, timeouts (connectTimeout, request timeout), y reemplaza HttpURLConnection.

BÁSICO

JAVA
RESPUESTA

Java moderno

J213

Devuelve VALOR: int dias = switch(mes) { case ENE, MAR -> 31; default -> 30; }.
 Flecha -> sin fall-through; caso múltiple con comas; yield para bloques: case X -> { ... yield resultado; }.
 Exhaustividad: con enums/sealed, cubrir todos los casos reemplaza el default.
 Menos bugs de break olvidado.

BÁSICO

JAVA
RESPUESTA

Java moderno

J216

Usa '_' cuando no necesitas la variable: catch (Exception _), for (var _ : lista), pattern Punto(int _, int y).
 Expressa intención: 'esto aquí no importa' y evita nombres throw-away (ignored, unused).
 Además unnamed classes y el main compacto (Java 21+ preview → 25 final) para scripts.

BÁSICO

JAVA
RESPUESTA

Java moderno

J215

forEach/forEachRemaining, removeIf, replaceAll, computeIfAbsent/compute/merge (Map), getOrDefault, putIfAbsent, List.sort, Iterator.removeRemaining? — y en Java 21: SequencedCollection (getFirst/getLast/reversed()) para List/Deque.
 List.of/Set.of/Map.of + ofEntries y Map.entry (Java 9).

BÁSICO

JAVA

Java moderno

J217

Anotaciones: @Retention, @Target y cómo se leen en runtime

BASICO

JAVA

Java moderno

J218

Reflection: usos legítimos y costos

BASICO

JAVA

Java moderno

J219

JShell y scripts Java: qué permiten hoy

BASICO

JAVA

Java moderno

J220

¿Qué cambió con las versiones LTS y el ritmo semestral?

BASICO

JAVA

Java moderno

J221

Variedades de Optional (Java 9-11): or, ifPresentOrElse, stream

INTERMEDIO

JAVA

Java moderno

J222

Pattern matching para switch (Java 21): ejemplo con sealed

AVANZADO

JAVA

Java moderno

J223

Record patterns (Java 21): deconstrucción

AVANZADO

JAVA

Java moderno

J224

¿Qué es JPMS (Project Jigsaw, Java 9) y cuándo importa?

AVANZADO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

Java moderno

J218

RESPUESTA

Instanciar/invocar en runtime por nombre: `Class.forName()`, `getDeclaredConstructor().newInstance()`, `method.invoke(obj, args)` — `setAccessible(true)` para privados.
Usos: frameworks (DI, ORMs, serializers), testing de privados (mejor no).
Costos: lento vs llamada directa, rompe encapsulación, choca con modules/native-image. Cachear Method/Constructor si se usa mucho.

BASICO

JAVA

Java moderno

J217

RESPUESTA

@Retention: SOURCE (solo compilador), CLASS (default, no runtime), RUNTIME (reflexión las ve — Spring/JPA usan RUNTIME).
@Target: dónde aplica (METHOD, FIELD, TYPE, PARAMETER).
Leerlas: `clase.getMethod("x").getAnnotation(MiAnon.class)`.
Es el mecanismo que usan los frameworks para configurar sin XML.

BASICO

JAVA

Java moderno

J220

RESPUESTA

Ritmo: feature release cada 6 meses; LTS cada 2 años: 8 (2014), 11 (2018), 17 (2021), 21 (2023), 25 (2025).
21 trae: virtual threads (final), pattern matching switch, record patterns, sequenced collections.
Empresas: saltan de LTS a LTS; features 'preview' se estabilizan en 1-3 releases.

BASICO

JAVA

Java moderno

J219

RESPUESTA

JShell (Java 9+): REPL — probar APIs, calcular, explorar sin proyecto.
Scripts: `java script.java` ejecuta un .java directo (single-file source), con clases locales/unnamed y main compacto para utilidades rápidas.
En contenedores: `jshell -c` o imagen con JDK (no JRE slim).

BASICO

JAVA

Java moderno

J222

RESPUESTA

sealed interface Forma permits Circulo, Cuadrado.
`String nombre = switch(forma) { case Circulo c -> "círculo r=" + c.radio(); case Cuadrado c when c.lado() > 10 -> "cuadrado grande"; default -> "otro"; }`.
Guardas con when; el compilador verifica exhaustividad — agregar un subtipo rompe la compilación de todos los switch, que es BUENO.

AVANZADO

JAVA

Java moderno

J221

RESPUESTA

`or(supplier)`: Optional alternativo perezoso.
`ifPresentOrElse(consumer, runnable)`: hace una cosa si hay valor, otra si no (adiós if/else con `isPresent`).
`stream()`: convierte `Optional<Stream>` flat en un stream — ideal en `flatMap`: `repositorio.findById(id).stream()...`
`isEmpty()` complementa `isPresent`.

INTERMEDIO

JAVA

Java moderno

J224

RESPUESTA

Sistema de módulos: `module-info.java` declara `requires` y `exports` — encapsulación REAL (paquetes no exportados no son accesibles ni por reflexión por defecto).
Importa para: JDK modularizado (jlink para runtimes mínimos), bibliotecas fuertemente encapsuladas.
La mayoría de apps de negocio NO usan módulos propios, pero conviven con el JDK modular.

AVANZADO

JAVA

Java moderno

J223

RESPUESTA

`record Punto(int x, int y) { }`
`if (obj instanceof Punto(int x, int y)) { usar x, y }` — extrae componentes directamente.
Anidados: `case Linea(Punto(int x1,_), Punto p2)` — '_' es unnamed variable (Java 21+).
Elimina getters intermedios en lógica de descomposición.

AVANZADO

JAVA

Java moderno

J225

Proxy dinámico: cómo funciona Spring AOP por dentro

AVANZADO

JAVA

Java moderno

J226

Virtual threads + Structured Concurrency: el nuevo modelo

AVANZADO

JAVA

I/O y serialización

J227

java.io vs java.nio: la diferencia en una frase

BASICO

JAVA

I/O y serialización

J228

Files y Path: el recetario de operaciones de archivo

BASICO

JAVA

I/O y serialización

J229

Encoding: por qué especificar SIEMPRE el charset

BASICO

JAVA

I/O y serialización

J230

Lectura de archivos grandes sin reventar memoria

BASICO

JAVA

I/O y serialización

J231

transient y serialVersionUID

BASICO

JAVA

I/O y serialización

J232

Scanner vs BufferedReader para leer entrada

BASICO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA
RESPUESTA

Java moderno

J226

StructuredTaskScope (preview): lanza subtarefas virtuales y garantiza que NINGUNA quede huérfana — scope.join() espera; shutdownOnFailure o shutdownOnSuccess para cancelar hermanos.
Código concurrente con la claridad del secuencial: try (var scope = new StructuredTaskScope.ShutdownOnFailure()) { ... scope.join().throwIfFailed(); }

AVANZADO

JAVA
RESPUESTA

Java moderno

J225

Proxy.newProxyInstance(loader, interfaces, handler) crea en runtime una clase que implementa las interfaces y delega al InvocationHandler.
Spring: si tu bean implementa interfaces usa proxy JDK (por interfaz); si no, CGLIB (subclase generada).
Por eso: métodos final/static/private no se interceptan y self-invocation no pasa por el proxy.

AVANZADO

JAVA
RESPUESTA

I/O y serialización

J228

Path p = Path.of("dir", "archivo.txt"); Files.readAllLines(p), readString(p), write(p, bytes), exists, createDirectories, copy/move (REPLACE_EXISTING), delete, walk (stream recursivo), list, size, getLastModifiedTime.
Buffered: Files.newBufferedReader/Writer (charset explícito!).

BASICO

JAVA
RESPUESTA

I/O y serialización

J227

I/O clásico: streams bloqueantes orientados a byte/char.
NIO: channels + buffers + selectores (I/O no bloqueante) y un FileSystem moderno (Path/Files).
Para apps comunes: Files/Paths (NIO.2) para operaciones de archivo, streams para contenido.

BASICO

JAVA
RESPUESTA

I/O y serialización

J230

NUNCA readAllLines/readAllBytes para GBs.
Usar: Files.lines(p) (Stream perezoso, cerrar con try-with-resources), BufferedReader.lines(), o streaming por líneas.
Escribir con BufferedWriter y flush por lotes.
Para binario: InputStream.transferTo(out) copia en buffer.

BASICO

JAVA
RESPUESTA

I/O y serialización

J229

DefaultCharset depende del SO/locale (Windows-1252, UTF-8...) → acentos rotos entre entornos ('Ä©').
SIEMPRE: new InputStreamReader(in, StandardCharsets.UTF_8), Files.readString(p, UTF_8), getBytes(UTF_8).
Java 18+: UTF-8 es el default de la plataforma (JEP 400), pero explícito sigue siendo mejor.

BASICO

JAVA
RESPUESTA

I/O y serialización

J232

Scanner: parseo cómodo (nextInt, nextLine) pero lento y con trampa nextInt + nextLine (deja el \n — consumirlo).
BufferedReader.readLine: rápido, devuelve String, parseas con Integer.parseInt.
En challenges/entrada masiva: BufferedReader + StringTokenizer (split manual rápido).

BASICO

JAVA
RESPUESTA

I/O y serialización

J231

transient: campo NO se serializa (contraseñas, conexiones, cachés derivadas) — al deserializar queda null/0.
serialVersionUID: huella de versión de la clase; si cambia entre serializar y deserializar → InvalidClassException. Fijarlo explícito para controlar compatibilidad.
writeObject/readObject custom para control fino.

BASICO

JAVA

I/O y serialización

J233

Temp files, cleaning y Files.walk

BASICO

JAVA

I/O y serialización

J234

Serializable: cómo funciona y sus peligros

INTERMEDIO

JAVA

Fechas y hora

J235

java.time: las clases que hay que conocer (y qué reemplazan)

BASICO

JAVA

Fechas y hora

J236

¿Todas inmutables y thread-safe? ¿Y por qué importa?

BASICO

JAVA

Fechas y hora

J237

Duration vs Period vs Instant

BASICO

JAVA

Fechas y hora

J238

DateTimeFormatter: parseo y formato seguro

BASICO

JAVA

Fechas y hora

J239

Zonas horarias: la regla de oro al guardar fechas

BASICO

JAVA

Fechas y hora

J240

Operaciones frecuentes con LocalDate

BASICO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

I/O y serialización

J234

RESPUESTA

implements Serializable → ObjectOutputStream serializa el GRAFO completo de objetos (transitivamente).

Peligros: deserialización de datos no confiables = vulnerabilidad grave (gadget chains); versionado frágil (serialVersionUID); expone representación interna.

Alternativas modernas: JSON (Jackson), Protobuf/Avro.

Si lo usas: serialVersionUID fijo + transient para no serializar campos.

INTERMEDIO

JAVA

I/O y serialización

J233

RESPUESTA

Files.createTempFile(prefix, suffix) / createTempDirectory — en %TEMP%.

Borrar: Files.delete en finally, o deleteOnExit() (solo crece la lista; no para largos).

Files.walk(ruta) → Stream<Path> recursivo (cerrarlo); filter por extensiones; Files.walkFileTree para control fino (visitor).

BASICO

JAVA

Fechas y hora

J236

RESPUESTA

Sí: java.time es inmutable — cada operación devuelve NUEVA instancia (fecha.plusDays(1) NO muta).

Comparte instancias entre hilos sin riesgo (Date/Calendar no).

Trampa clásica: fecha = fecha.plusDays(1); sin reasignar se 'pierde' el cambio.

BASICO

JAVA

Fechas y hora

J235

RESPUESTA

LocalDate (solo fecha), LocalTime, LocalDateTime, ZonedDateTime (con zona), OffsetDateTime (con offset), Instant (punto en la línea UTC, timestamps), Duration (tiempo/horas), Period (días/meses/años), DateTimeFormatter, ZonedDateTime.

Reemplazan a Date/Calendar (mutables, meses 0-based, zona confusa — legado).

BASICO

JAVA

Fechas y hora

J238

RESPUESTA

DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm") o constantes ISO (ISO_LOCAL_DATE).

Formato: fecha.format(fmt). Parseo: LocalDate.parse("2024-01-31") (ISO) o LocalDate.parse(s, fmt).

IMMUTABLE y thread-safe (SimpleDateFormat NO lo era — bug clásico).

Locales: ofPattern(..., Locale.forLanguageTag("es")).

BASICO

JAVA

Fechas y hora

J237

RESPUESTA

Duration: cantidad de TIEMPO en escala de reloj (horas, minutos, segundos, nanos) — Duration.between(ini, fin).

Period: cantidad en CALENDARIO (años, meses, días) — Period.between(f1, f2).

Instant: punto exacto en UTC (epoch seconds+nanos) — para timestamps y machine time.

ZonedDateTime para mostrar a humanos con zona.

BASICO

JAVA

Fechas y hora

J240

RESPUESTA

plusDays/plusMonths/plusYears/plus(Period), minusX, withDayOfMonth, atStartOfDay(), atTime(h,m), toLocalDate/toLocalDateTime, isBefore/isAfter/isEqual, until(otro, ChronoUnit.DAYS).

Primer/último del mes: with(TemporalAdjusters.firstDayOfMonth()), lastDayOfMonth.

lengthOfMonth(), isLeapYear().

BASICO

JAVA

Fechas y hora

J239

RESPUESTA

Guardar y transmitir en UTC (Instant) o timestamps; convertir a zona SOLO al mostrar.

ZonedDateTime.now(ZonedDateTime.of("America/Lima")); conversión: zdt.withZoneSameInstant(otraZona).

Nunca hacer aritmética de fechas con Strings ni asumir la zona del servidor.

OffsetDateTime vs ZonedDateTime: offset fijo vs reglas de zona (DST).

BASICO

JAVA

Fechas y hora

J241

LocalDate vs LocalDateTime vs Instant:
¿cuál guardar para un evento?

BASICO

JAVA

Fechas y hora

J242

¿Cómo migrar código legado con Date y Calendar?

BASICO

JAVA

Fechas y hora

J243

Comparar y medir intervalos con java.time

BASICO

JAVA

Fechas y hora

J244

Reloj inyectable (Clock) para código testeable

BASICO

JAVA★

Snippets: ¿qué imprime?

J245

Integer a=127,b=127; Integer c=128,d=128;
a==b ? c==d ?

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J246

String s="hola"; String t="ho"+"la"; s==t ?
new String("hola")==s ?

INTERMEDIO

JAVA★

Snippets: ¿qué imprime?

J247

List<Integer> l = new
ArrayList<>(List.of(1,2,3)); l.remove(1);
¿qué queda?

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J248

try { return "try"; } finally {
System.out.println("fin"); } — ¿orden de
salida?

INTERMEDIO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

Fechas y hora

J242

RESPUESTA

Convertir: `date.toInstant()` (Java 8+), `Instant`→`Date`:
`Date.from(instant)`.
Calendar: `cal.toInstant()`.
Nuevas APIs no aceptan `Date`: convertir en la frontera del sistema legado y usar `java.time` por dentro.
Evitar mezclar en el mismo dominio.

BASICO

JAVA

Fechas y hora

J241

RESPUESTA

`Instant` (o timestamp UTC en DB): eventos de sistema, auditoría, `createdAt` — sin ambigüedad.
`LocalDate`: fechas de calendario puras (cumpleaños, vencimientos sin hora).
`LocalDateTime`: hora local sin zona (horarios de tienda) — solo si TODOS comparten zona.
Regla: si cruza zonas o sistemas → `Instant/ZonedDateTime`.

BASICO

JAVA

Fechas y hora

J244

RESPUESTA

`Clock.systemDefaultZone()` en producción; `Clock.fixed(instant, zone)` en tests.
Instanciar servicios con `Clock` y usar `Instant.now(clock)` / `LocalDate.now(clock)` → pruebas deterministas de 'hoy', vencimientos y expiraciones sin sleeps.

BASICO

JAVA

Fechas y hora

J243

RESPUESTA

`isBefore/isAfter/isEqual` entre objetos del mismo tipo.
`Duration.between(t1, t2).toMillis()` / `toMinutes()`.
`ChronoUnit.DAYS.between(f1, f2)`, `MONTHS.between`.
between exige tipos compatibles (`LocalDate` con `LocalDate`, `Instant` con `Instant`).

BASICO

JAVA

Snippets: ¿qué imprime?

J246

RESPUESTA

`true` y `false`.
"ho"+"la" es constante plegada en COMPILACIÓN → misma referencia del pool que `s`.
`new String("hola")` crea un objeto nuevo en el heap → `== false` (equals sería `true`).
`s.intern() == new String("hola").intern()` → `true`.

INTERMEDIO

JAVA



Snippets: ¿qué imprime?

J245

RESPUESTA

`true` y `false`.
`Integer.valueOf` cachea [-128, 127] → `a` y `b` son el MISMO objeto.
128 crea objetos distintos → `==` compara referencias y da `false`.
Lección: wrappers SIEMPRE `equals()`.

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J248

RESPUESTA

Imprime 'fin' y LUEGO devuelve "try": el `finally` corre ANTES de que el `return` del `try` complete la salida del método.
Si el `finally` hiciera `return "finally"`, ese valor GANA y se silencian excepciones pendientes (mala práctica).

INTERMEDIO

JAVA



Snippets: ¿qué imprime?

J247

RESPUESTA

[1, 3] — `remove(1)` elimina el ÍNDICE 1 (el 2), no el valor 1.
Para eliminar el valor: `remove(Integer.valueOf(1))`.
El overload elegido depende del tipo ESTÁTICO del argumento:
`int` → índice; `Integer` → objeto.

INTERMEDIO

JAVA ★

Snippets: ¿qué imprime?

J249

```
System.out.println(0.1+0.2 == 0.3); y  
System.out.println(1/2);
```

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J250

```
Integer x = null; int y = (condicion) ? x :  
0; — ¿qué pasa si condicion es true?
```

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J251

```
char c = 'A'; System.out.println(c + 1);  
System.out.println("" + c + 1);
```

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J252

```
Map<Integer,String> m = new  
HashMap<>(); m.put(1,"a");  
m.put(null,"b"); ¿lanza algo? ¿y en  
TreeMap?
```

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J253

```
static int x=10; static { x=20; } ¿en qué  
orden se ejecuta con new Clase()?
```

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J254

```
double d = Double.NaN; d == d — ¿true o  
false?
```

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J255

```
String s = "a"; s.concat("b");  
System.out.println(s);
```

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J256

```
List.of("a").add("b") — ¿qué pasa?
```

INTERMEDIO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA

Snippets: ¿qué imprime?

J250

RESPUESTA

NullPointerException.

El ternario con un brazo int y otro Integer fuerza UNBOXING de ambos → desempaqueta el null.

Fix: `int y = (cond && x != null) ? x : 0;` o evitar mezclar primitivo y wrapper en ternarios.

INTERMEDIO

JAVA



Snippets: ¿qué imprime?

J249

RESPUESTA

false y 0.

$0.1 + 0.2 = 0.30000000000000004$ en binario IEEE754 → comparar doubles con epsilon o BigDecimal.

1/2 es división ENTERA (trunca a 0); 1.0/2 sería 0.5.

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J252

RESPUESTA

HashMap: perfecto, admite UNA clave null (y valores null) — no lanza nada.

`TreeMap.put(null,...)` → NullPointerException (comparar null con orden natural rompe el árbol).

`ConcurrentHashMap` → NullPointerException también.

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J251

RESPUESTA

66 y A1.

`char + int` = aritmética (promoción a int → 65+1).

`String + char` inicia concatenación → "A1" (todo lo que sigue se concatena).

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J254

RESPUESTA

false.

NaN es lo único distinto de sí mismo en IEEE754.

Detectarlo: `Double.isNaN(d)`. Y NaN en un HashSet puede duplicarse (equals falsificado por comparación de bits:

`Double.valueOf(NaN).equals(NaN)` sí es true — inconsistencia deliberada).

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J253

RESPUESTA

En orden de aparición del archivo: primero `x=10` (inicializador de campo), luego el bloque `static x=20`.

Todo el bloque estático corre UNA vez al cargar la clase, ANTES de cualquier instancia y antes de main.

Reordenar líneas cambia el resultado final → pregunta clásica de examen.

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J256

RESPUESTA

UnsupportedOperationException EN RUNTIME.

`List.of` crea colecciones INMUTABLES; cualquier mutación lanza excepción.

Si necesitas mutable: `new ArrayList<>(List.of("a"))` — constructor copia.

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J255

RESPUESTA

"a" — `String` es INMUTABLE: `concat` devuelve un NUEVO `String` que se descarta al no asignarlo.

Lo mismo con `toUpperCase`, `replace`, `trim`...

Solo si haces `s = s.concat("b")` cambia la variable (a un objeto nuevo).

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J257

HashSet<Point> set: agrego p (x=1), luego p.x=2, set.contains(p nuevo con x=2) → ?

INTERMEDIO

JAVA

Snippets: ¿qué imprime?

J258

System.out.println(10 + 20 + "" + 10 + 20);

INTERMEDIO

JAVA★

JDBC y localización

J259

¿Qué es JDBC?

BASICO

JAVA

JDBC y localización

J260

Localización (i18n): Locale, ResourceBundle y NumberFormat

BASICO

JAVA

JDBC y localización

J261

JDBC: flujo básico de una consulta (código)

INTERMEDIO

JAVA★

JDBC y localización

J262

PreparedStatement vs Statement: ¿por qué SIEMPRE PreparedStatement?

INTERMEDIO

JAVA

JDBC y localización

J263

Transacciones en JDBC plano

INTERMEDIO

REVERSO — cada respuesta queda al dorso de su pregunta (misma numeración)

JAVA
RESPUESTA

Snippets: ¿qué imprime?

J258

`"302010"`.De izquierda a derecha: $10+20$ (aritmética) = 30; al tocar el String "" todo lo demás se CONCATENA: $30 + "10" + "20" \rightarrow 302010$.

Evaluó de expresión: la aritmética solo ocurre mientras no haya String en la cadena.

INTERMEDIO

JAVA
RESPUESTA

Snippets: ¿qué imprime?

J257

false (o comportamiento inconsistente): el objeto `MUTÓ` su hashCode DESPUÉS de insertarse — quedó en un bucket que ya no le corresponde.

El set puede no encontrar ni eliminar el elemento (remove falla silenciosamente).

Lección: campos que participan en hash/equals deben ser inmutables.

INTERMEDIO

JAVA
RESPUESTA

JDBC y localización

J260

Locale: identifica idioma+país (new Locale("es", "PE"), Locale.getDefault()).

ResourceBundle: archivos messages_es.properties, messages_en.properties → bundle.getString("saludo") según el Locale.

NumberFormat/DateFormat: formatean números, monedas y fechas según el locale (1.234,56 vs 1,234.56).

OCP lo pregunta poco, pero aparece en apps multi-idioma.

BASICO

JAVA ★
RESPUESTA

JDBC y localización

J259

Java Database Connectivity: la API estándar de Java para hablar con bases de datos relacionales.

Estructura: DriverManager/DataSource → Connection → Statement → ResultSet.

Cada motor (Postgres, MySQL...) provee su DRIVER (jar) que implementa la API.

Es la base sobre la que trabajan JPA/Hibernate y JdbcTemplate.

BASICO

JAVA ★
RESPUESTA

JDBC y localización

J262

PreparedStatement: la consulta se PRE-COMPIILA y los parámetros van aparte (?) → immune a SQL Injection y más rápida si se reutiliza.

Statement con concatenación: 'WHERE nombre = ' + input + ' ' → input='; DROP TABLE users; -- = desastre.

Regla de seguridad: jamás concatenar input del usuario en SQL.

INTERMEDIO

JAVA
RESPUESTA

JDBC y localización

J261

```
try (Connection cn = DriverManager.getConnection(url, user, pass);
    PreparedStatement ps = cn.prepareStatement("SELECT nombre FROM users WHERE id = ?")) {
    ps.setLong(1, id);
    try (ResultSet rs = ps.executeQuery()) {
        while (rs.next()) System.out.println(rs.getString("nombre"));
    }
}
```

Recursos AutoCloseable → try-with-resources obligatorio.

INTERMEDIO

JAVA
RESPUESTA

JDBC y localización

J263

Por defecto está en autocommit (cada sentencia es su tx).

Control manual:

```
cn.setAutoCommit(false);
try { ...updates...; cn.commit(); }
catch (Exception e) { cn.rollback(); }
```

Es lo que @Transactional hace por ti debajo (con DataSourceTransactionManager).

INTERMEDIO