JAVA

Análisis para Expertos

Alejandro G Vera

El Compendio Definitivo de la Entrevista Técnica

100 Preguntas y Análisis de Expertos

Un recurso exhaustivo para evaluar la comprensión fundamental, aplicación práctica y perspicacia arquitectónica en el ecosistema Java.

El Compendio Definitivo de la Entrevista Técnica de Java: 100 Preguntas y Análisis de Expertos

Introducción

Este documento sirve como un recurso exhaustivo y de nivel experto para llevar a cabo y prepararse para entrevistas técnicas de Java. Su objetivo es ir más allá de la memorización de conceptos y evaluar la comprensión fundamental, la aplicación práctica y la perspicacia arquitectónica de un candidato. Las 100 preguntas aquí presentes han sido seleccionadas para cubrir la amplitud y profundidad del ecosistema de Java, desde la mecánica del lenguaje principal hasta los marcos de trabajo empresariales. Están diseñadas para diferenciar a los candidatos al poner a prueba su conocimiento sobre las compensaciones, los mecanismos internos y los principios de diseño que definen a un ingeniero de software de Java verdaderamente competente.
El compendio está organizado en siete secciones temáticas, cada una construida sobre la anterior, para proporcionar una progresión lógica desde los conceptos fundamentales hasta temas especializados y avanzados. Cada respuesta está diseñada para ofrecer no solo una definición, sino también los principios subyacentes, las implicaciones prácticas y ejemplos de código relevantes. Este enfoque estructurado tiene como objetivo evaluar la comprensión holística de un candidato, reflejando la amplia gama de temas que se encuentran en el desarrollo de software moderno.

Sección I: Java Core y Principios de Programación Orientada a Objetos

Esta sección establece una base de comprensión del lenguaje Java y el paradigma de la Programación Orientada a Objetos (POO). Pone a prueba el dominio del candidato sobre los conceptos fundamentales que sustentan todo el desarrollo en Java.

1. ¿Cuál es la diferencia entre JDK, JRE y JVM?

Esta es una pregunta fundamental para evaluar si el candidato comprende las herramientas básicas del ecosistema de Java. La respuesta debe diferenciar claramente los tres componentes en función de su propósito y contenido.

- **JVM (Java Virtual Machine):** Es una especificación que define una máquina abstracta. Su función principal es proporcionar un entorno de ejecución independiente de la plataforma que convierte el bytecode de Java en código máquina nativo. La JVM es la pieza clave que permite el principio de "Escribir una vez, ejecutar en cualquier lugar" (*Write Once, Run Anywhere*). Cada sistema operativo (Windows, macOS, Linux) tiene su propia implementación de la JVM, pero todas son capaces de ejecutar el mismo bytecode. La JVM también es responsable de la gestión de la memoria, incluida la

recolección de basura (*Garbage Collection*), y de la seguridad.

- **JRE (Java Runtime Environment):** Es la implementación de la JVM. Es un paquete de software que contiene todo lo necesario para ejecutar aplicaciones Java, pero no para desarrollarlas. Incluye la JVM, las bibliotecas de clases principales de Java (como rt.jar) y otros archivos de soporte. Un usuario final que solo necesita ejecutar una aplicación Java solo necesita instalar el JRE.
- **JDK (Java Development Kit):** Es el kit de desarrollo completo para programadores de Java. Incluye todo lo que contiene el JRE, además de herramientas de desarrollo esenciales como el compilador (javac), el depurador (jdb), el generador de documentación (javadoc) y otras utilidades. Por lo tanto, para desarrollar, compilar y ejecutar aplicaciones Java, se necesita el JDK.

En resumen, la relación jerárquica es: JDK \supset JRE \supset JVM. Un desarrollador utiliza el JDK, mientras que un entorno de producción solo necesita el JRE para ejecutar la aplicación.

2. ¿Cómo logra Java la independencia de la plataforma?

La independencia de la plataforma es una de las características más importantes de Java. Una respuesta completa debe ir más allá de simplemente mencionar "bytecode" y explicar el papel de la JVM como una capa de abstracción.
El proceso funciona de la siguiente manera:

1. **Compilación a Bytecode:** Cuando un desarrollador escribe código fuente de Java (un archivo .java), el compilador de Java (javac), que forma parte del JDK, no lo compila a código máquina nativo específico de un sistema operativo. En su lugar, lo compila en un formato intermedio llamado **bytecode** (un archivo .class).
2. **Ejecución por la JVM:** Este archivo de bytecode es universal y no está ligado a ninguna plataforma de hardware o software. Para ejecutar este bytecode, una máquina debe tener instalado un Java Runtime Environment (JRE), que incluye una Java Virtual Machine (JVM).
3. **Interpretación y Compilación JIT:** La JVM actúa como un intérprete que lee el bytecode y lo traduce a instrucciones de máquina nativas para el sistema operativo y la arquitectura de CPU subyacentes. Para mejorar el rendimiento, la JVM utiliza un compilador **Just-In-Time (JIT)**. El JIT identifica las partes del código que se ejecutan con más frecuencia (puntos calientes o *hotspots*) y las compila a código nativo en tiempo de ejecución. Este código nativo compilado se almacena en caché y se reutiliza en llamadas posteriores, lo que acelera significativamente la ejecución.

Por lo tanto, la independencia de la plataforma se logra porque el código compilado (bytecode) es portable, mientras que la dependencia de la plataforma se aísla dentro de la implementación de la JVM. Esto permite que el mismo archivo .jar se ejecute sin cambios en Windows, macOS o Linux, siempre que cada sistema tenga su JRE correspondiente.

3. ¿Cuál es la diferencia entre tipos de datos primitivos y tipos de referencia?

Esta pregunta explora las diferencias fundamentales en cómo Java gestiona los datos, lo cual tiene implicaciones directas en la gestión de la memoria, el rendimiento y el comportamiento del paso de parámetros.

Característica	Tipos Primitivos	Tipos de Referencia
Definición	Son los tipos de datos básicos e indivisibles del lenguaje. Hay ocho: byte, short, int, long, float, double, char, boolean.	Son objetos creados a partir de clases (como String, ArrayList), interfaces, arrays o enums. Hacen referencia a una ubicación en la memoria.
Almacenamiento	Almacenan su valor directamente en la memoria **Stack** (si son variables locales) o como parte del objeto en el **Heap** (si son variables de instancia).	La variable almacena una **referencia** (dirección de memoria) en el Stack, que apunta a la ubicación del objeto real en la memoria **Heap**.
Valor por Defecto	Tienen valores por defecto numéricos (0, 0.0), false para boolean, y '\u0000' para char si son variables de instancia.	El valor por defecto es siempre null, lo que indica que la referencia no apunta a ningún objeto.
Paso a Métodos	Se pasan **por valor**. Se crea una copia del valor de la variable y se pasa al método. Cualquier cambio dentro del método no afecta a la variable original.	También se pasan **por valor**, pero el valor que se copia es la **referencia** al objeto. Esto significa que el método recibe una copia de la dirección de memoria. Aunque no se puede cambiar a qué objeto apunta la referencia original, sí se puede modificar el estado del objeto al que apunta.
Uso de Memoria	Ocupan una cantidad fija y generalmente pequeña de memoria.	Ocupan más memoria debido a la sobrecarga del objeto (cabeceras, etc.) además de los datos que contiene.

4. ¿Por qué la clase String es inmutable en Java y cómo funciona el String Pool?

La inmutabilidad de String es una decisión de diseño fundamental en Java con importantes beneficios en seguridad, concurrencia y rendimiento.
Inmutabilidad de String: Un objeto String es inmutable porque, una vez creado, su estado (la secuencia de caracteres que representa) no puede ser modificado. Cualquier operación que parezca modificar un String (como concat(), substring() o replace()) en realidad crea y devuelve un nuevo objeto String con el resultado de la operación, dejando el original intacto. Esto se logra declarando la clase String como final (para que no pueda ser extendida) y su array de caracteres interno (char) como private y final.
Las razones clave para esta inmutabilidad son:

1. **Seguridad:** Los String se utilizan ampliamente como parámetros en operaciones críticas (nombres de archivo, conexiones de red, credenciales). Si fueran mutables, un método malicioso podría cambiar el valor del String después de una comprobación de seguridad, llevando a vulnerabilidades.
2. **Concurrencia (Thread Safety):** Al ser inmutables, los objetos String son intrínsecamente

seguros para subprocesos (*thread-safe*). Pueden ser compartidos entre múltiples hilos sin necesidad de sincronización, ya que ningún hilo puede alterar su estado.

3. **Rendimiento y Caching:** La inmutabilidad permite que el hashCode() de un String se calcule una sola vez y se almacene en caché. Esto hace que los String sean claves muy eficientes en colecciones basadas en hash como HashMap.

String Pool (Piscina de Cadenas): El String Pool es un área especial de almacenamiento en la memoria Heap donde Java guarda los literales de cadena. Su propósito es optimizar el uso de la memoria y mejorar el rendimiento al evitar la creación de objetos String duplicados.

- **Cómo funciona:**
    - Cuando se declara un literal de cadena, como String s1 = "Java";, la JVM primero busca en el String Pool si ya existe una cadena con el valor "Java".
    - Si la encuentra, la variable s1 simplemente recibe la referencia a ese objeto existente en el pool.
    - Si no la encuentra, se crea un nuevo objeto String en el pool con el valor "Java", y s1 recibe la referencia a este nuevo objeto.
    - Cuando se crea un String usando el operador new, como String s2 = new String("Java");, se fuerza la creación de un **nuevo** objeto en el Heap, fuera del pool, incluso si "Java" ya existe en el pool. Sin embargo, el literal "Java" utilizado en el constructor también reside en el pool. El método intern() se puede usar para obtener explícitamente la referencia del pool para un String.

5. Explique la diferencia entre final, finally y finalize.

Esta es una pregunta clásica para comprobar la precisión y el conocimiento detallado del lenguaje por parte de un candidato, ya que los nombres son similares pero sus funciones son completamente diferentes.

- **final:** Es un **modificador (palabra clave)** que se puede aplicar a variables, métodos y clases para restringir su modificación.
    - **Variable final:** Una vez asignado un valor, no se puede cambiar. Actúa como una constante.
    - **Método final:** No puede ser sobrescrito (*overridden*) por una subclase. Se utiliza para garantizar que el comportamiento de un método no sea alterado.
    - **Clase final:** No puede ser extendida (heredada). La clase String es un ejemplo de una clase final.
- **finally:** Es un **bloque de código** utilizado en el manejo de excepciones (try-catch-finally). El código dentro del bloque finally se ejecuta **siempre**, independientemente de si se lanza o no una excepción en el bloque try, e incluso si hay una declaración return en el bloque try o catch. Su propósito principal es garantizar la liberación de recursos (como cerrar archivos, conexiones de red o bases de datos) para evitar fugas de recursos. La única situación en la que un bloque finally no se ejecuta es si la JVM se detiene abruptamente (por ejemplo, con System.exit()) o si ocurre un error irrecuperable.
- **finalize():** Es un **método** de la clase Object. La JVM lo llama en un objeto justo antes de que el recolector de basura lo reclame. Su intención original era permitir la limpieza de recursos no gestionados por Java (como manejadores de archivos nativos). Sin embargo, su uso está **fuertemente desaconsejado** y ha sido **deprecado** a partir de Java 9. Las razones son que no hay garantía de cuándo (o si) se ejecutará, y puede introducir problemas de rendimiento y concurrencia. La alternativa moderna y recomendada para la gestión de recursos es el uso de la interfaz AutoCloseable y el bloque try-with-resources.

6. ¿Cuáles son los cuatro pilares de la Programación Orientada a Objetos (POO)? Proporcione un ejemplo práctico para cada uno.

Los cuatro pilares de la POO son los principios fundamentales que guían el diseño de software modular, flexible y mantenible. Son Encapsulación, Herencia, Polimorfismo y Abstracción.

1. **Encapsulación:**
    - **Concepto:** Es el mecanismo de agrupar los datos (atributos) y los métodos que operan sobre esos datos dentro de una única unidad, una clase. También implica ocultar el estado interno de un objeto del mundo exterior y solo permitir el acceso a través de métodos públicos (getters y setters). Esto protege los datos de modificaciones no autorizadas y reduce la complejidad del sistema.
    - **Ejemplo Práctico:** Una clase CuentaBancaria que encapsula un atributo saldo. El saldo es private, por lo que no puede ser modificado directamente. Solo se puede acceder a él a través de métodos públicos como depositar(double monto) y retirar(double monto), que contienen la lógica para validar las operaciones (p. ej., no permitir un retiro que deje el saldo negativo).

```
public class CuentaBancaria {
    private double saldo; // Ocultación de datos

    public CuentaBancaria(double saldoInicial) {
        this.saldo = saldoInicial;
    }

    public void depositar(double monto) {
        if (monto > 0) {
            this.saldo += monto;
        }
    }

    public double getSaldo() { // Acceso controlado
        return this.saldo;
    }
}
```

2. **Herencia:**
    - **Concepto:** Es un mecanismo por el cual una clase (subclase o clase hija) puede heredar atributos y métodos de otra clase (superclase o clase padre). Promueve la reutilización de código y establece una relación "es un" (*is-a*) entre las clases.
    - **Ejemplo Práctico:** Una clase base Vehiculo con atributos como marca y velocidad, y un método acelerar(). Las clases Coche y Bicicleta pueden heredar de Vehiculo, reutilizando sus propiedades y métodos, y añadiendo los suyos propios (p. ej., numeroDePuertas para Coche).

```
public class Vehiculo {
    protected String marca;
    public void acelerar() {
        System.out.println("El vehículo está acelerando.");
```

```
    }
}

public class Coche extends Vehiculo { // Coche "es un" Vehiculo
    private int numeroDePuertas;
}
```

3. **Polimorfismo:**
    - **Concepto:** Significa "muchas formas". Permite que objetos de diferentes clases respondan al mismo mensaje (llamada de método) de maneras específicas para cada clase. Se manifiesta principalmente a través de la **sobrescritura de métodos** (*method overriding*), donde una subclase proporciona una implementación específica de un método que ya está definido en su superclase.
    - **Ejemplo Práctico:** Siguiendo con el ejemplo anterior, tanto Coche como Bicicleta pueden sobrescribir un método moverse(). Un objeto de tipo Vehiculo puede referirse a una instancia de Coche o Bicicleta, y al llamar a moverse(), se ejecutará la implementación correcta en tiempo de ejecución.

```
public abstract class Animal {
    public abstract void hacerSonido();
}

public class Perro extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("Guau");
    }
}

public class Gato extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("Miau");
    }
}

// Uso polimórfico
Animal miMascota = new Perro();
miMascota.hacerSonido(); // Imprime "Guau"
miMascota = new Gato();
miMascota.hacerSonido(); // Imprime "Miau"
```

4. **Abstracción:**
    - **Concepto:** Es el proceso de ocultar los detalles complejos de implementación y mostrar solo la funcionalidad esencial al usuario. Se centra en el "qué" hace un objeto en lugar del "cómo" lo hace. En Java, la abstracción se logra mediante **clases abstractas** e **interfaces**.
    - **Ejemplo Práctico:** Una interfaz ControlRemoto puede definir métodos como encender(), apagar() y cambiarCanal(). Cualquier clase que implemente esta

interfaz (como ControlRemotoTV o ControlRemotoAireAcondicionado) debe proporcionar la implementación específica de estos métodos, pero el usuario solo necesita conocer la interfaz ControlRemoto para interactuar con ellos, sin preocuparse por los detalles internos de cada dispositivo.

7. ¿Cuál es la diferencia entre sobrecarga de métodos (overloading) y sobrescritura de métodos (overriding)?

Esta pregunta evalúa la comprensión del candidato sobre el polimorfismo en Java, que se divide en polimorfismo en tiempo de compilación (estático) y en tiempo de ejecución (dinámico).

Característica	Sobrecarga de Métodos (Overloading)	Sobrescritura de Métodos (Overriding)
Propósito	Aumentar la legibilidad del programa permitiendo que múltiples métodos con el mismo nombre realicen tareas similares pero con diferentes tipos o número de argumentos.	Proporcionar una implementación específica de un método en una subclase que ya está definido en su superclase.
Relación	Ocurre dentro de la misma clase o en una relación de herencia.	Ocurre solo en una relación de herencia entre una superclase y una subclase.
Firma del Método	Los métodos deben tener el **mismo nombre** pero una **lista de parámetros diferente** (diferente número de parámetros, tipo de parámetros o ambos).	Los métodos deben tener el **mismo nombre, misma lista de parámetros y mismo tipo de retorno** (o un subtipo, conocido como tipo de retorno covariante a partir de Java 5).
Tipo de Polimorfismo	**Polimorfismo en tiempo de compilación (Estático)**. El compilador decide qué método llamar en tiempo de compilación basándose en los argumentos de la llamada.	**Polimorfismo en tiempo de ejecución (Dinámico)**. La JVM decide qué método llamar en tiempo de ejecución basándose en el tipo real del objeto.
Palabra clave @Override	No se puede usar.	Es opcional pero muy recomendable. Ayuda al compilador a verificar que el método realmente está sobrescribiendo un método de la superclase.
Métodos static, final, private	Los métodos estáticos pueden ser sobrecargados.	Los métodos estáticos, finales y privados no pueden ser sobrescritos. Los métodos estáticos se ocultan (*hiding*), no se sobrescriben.

Ejemplo de Sobrecarga:

```
class Calculadora {
    int sumar(int a, int b) { return a + b; }
```

```
    double sumar(double a, double b) { return a + b; }
}
```

Ejemplo de Sobrescritura:

```
class Animal {
    void comer() { System.out.println("El animal come"); }
}

class Perro extends Animal {
    @Override
    void comer() { System.out.println("El perro come croquetas"); }
}
```

8. ¿Cuándo usaría una clase abstracta en lugar de una interfaz?

La elección entre una clase abstracta y una interfaz es una decisión de diseño importante que depende de la relación entre las clases y la funcionalidad deseada.

Debería usar una Clase Abstracta cuando:

1. **Quiere compartir código entre clases estrechamente relacionadas.** Las clases abstractas permiten definir métodos concretos (con implementación) y métodos protected. Esto es útil cuando varias subclases comparten un comportamiento común. Por ejemplo, una clase abstracta Animal podría tener un método concreto respirar() que es igual para todos los animales.
2. **Necesita declarar campos no estáticos y no finales.** Las clases abstractas pueden tener variables de instancia (private, protected) que definen el estado de un objeto. Las interfaces solo pueden tener constantes (public static final).
3. **Espera que las clases que extienden su clase base tengan muchos métodos o campos en común.** Si las clases están fuertemente acopladas y forman parte de una jerarquía clara (relación "es un"), una clase abstracta es una buena opción.
4. **Quiere controlar la evolución de la API.** Si agrega un nuevo método a una clase abstracta, puede proporcionar una implementación predeterminada y las clases existentes no se romperán.

Debería usar una Interfaz cuando:

1. **Espera que clases no relacionadas implementen la interfaz.** Por ejemplo, las interfaces Comparable o Serializable pueden ser implementadas por clases muy diferentes (como String, File, etc.) que no tienen una relación jerárquica entre sí. Las interfaces definen un "contrato" de comportamiento (relación "puede hacer").
2. **Quiere especificar el comportamiento de un tipo de dato, pero no le preocupa quién lo implementa.** Las interfaces son ideales para definir un contrato que múltiples clases pueden cumplir.
3. **Quiere aprovechar la herencia múltiple de tipos.** Una clase en Java puede implementar múltiples interfaces, pero solo puede extender una clase. Esto hace que las interfaces sean la única forma de lograr la herencia múltiple de comportamiento.
4. **Quiere desacoplar componentes.** Programar contra interfaces en lugar de implementaciones concretas es un principio de diseño clave que promueve un bajo acoplamiento y facilita la prueba y el mantenimiento.

A partir de **Java 8**, las interfaces pueden tener **métodos default y static**, lo que ha reducido la

brecha con las clases abstractas. Sin embargo, la distinción fundamental sigue siendo la misma: las clases abstractas son para compartir estado y comportamiento común en una jerarquía de clases relacionadas, mientras que las interfaces son para definir un contrato de comportamiento que puede ser implementado por cualquier clase.

9. Explique el principio de "composición sobre herencia" y por qué se considera una buena práctica.

Este principio de diseño de software sugiere que las clases deben lograr el comportamiento polimórfico y la reutilización de código mediante la composición (conteniendo instancias de otras clases) en lugar de la herencia (heredando de una clase base). Es una de las pautas de diseño más importantes en la programación orientada a objetos.
La herencia crea una relación **"es un"** (*is-a*), mientras que la composición crea una relación **"tiene un"** (*has-a*). Aunque la herencia es poderosa, a menudo se abusa de ella, lo que lleva a varios problemas:

1. **Acoplamiento Fuerte:** La herencia crea una de las formas más fuertes de acoplamiento en la POO. La subclase está íntimamente ligada a la implementación de su superclase. Un cambio en la superclase puede romper inesperadamente la funcionalidad de la subclase, un problema conocido como el **problema de la clase base frágil** (*Fragile Base Class Problem*).
2. **Ruptura de la Encapsulación:** La herencia puede romper la encapsulación, ya que la subclase puede necesitar conocer los detalles de la implementación de la superclase para funcionar correctamente, especialmente si accede a miembros protected.
3. **Flexibilidad Limitada en Tiempo de Ejecución:** La relación de herencia se establece en tiempo de compilación y no se puede cambiar en tiempo de ejecución. La composición, por otro lado, permite cambiar el comportamiento de un objeto en tiempo de ejecución al cambiar el objeto compuesto.
4. **Explosión de Clases:** Si se necesita una combinación de diferentes comportamientos, la herencia puede llevar a una explosión de subclases para cubrir todas las combinaciones posibles.

Ventajas de la Composición:

1. **Flexibilidad y Desacoplamiento:** La composición conduce a un diseño más flexible y desacoplado. Las clases se centran en una única responsabilidad y colaboran a través de interfaces bien definidas. El comportamiento de un objeto se puede cambiar en tiempo de ejecución al proporcionar un componente diferente.
2. **Mejor Encapsulación:** La clase contenedora no necesita exponer sus detalles internos. Interactúa con el objeto contenido a través de su interfaz pública, manteniendo una fuerte encapsulación.
3. **Facilidad de Prueba:** Los sistemas basados en composición son más fáciles de probar. Dado que las dependencias se basan en interfaces, se pueden inyectar fácilmente implementaciones simuladas (*mocks*) para las pruebas unitarias.

¿Cuándo usar la herencia? La herencia sigue siendo apropiada cuando la relación entre las clases es genuinamente una relación "es un" y la subclase es un subtipo verdadero de la superclase (cumple con el Principio de Sustitución de Liskov). Es decir, la subclase puede ser utilizada en cualquier lugar donde se espere la superclase sin alterar la corrección del programa.
En resumen, se debe favorecer la composición porque ofrece más flexibilidad, un acoplamiento

más débil y un diseño más robusto y mantenible. La herencia debe usarse con moderación y solo cuando la relación de subtipado es clara y estable.

10. ¿Qué es un constructor y se puede heredar?

Un **constructor** es un método especial dentro de una clase que se utiliza para inicializar los objetos recién creados. Se llama automáticamente cuando se crea una instancia de una clase utilizando el operador new.
Las características clave de un constructor son:

- **Nombre:** El nombre del constructor debe ser exactamente el mismo que el nombre de la clase.
- **Sin Tipo de Retorno:** Un constructor no tiene un tipo de retorno explícito, ni siquiera void. Su propósito implícito es devolver una instancia de la clase.
- **Inicialización:** Su función principal es establecer el estado inicial de un objeto, asignando valores a sus variables de instancia.
- **Sobrecarga:** Una clase puede tener múltiples constructores, siempre que sus listas de parámetros sean diferentes (constructor sobrecargado). Esto permite crear objetos de diferentes maneras.
- **Constructor por Defecto:** Si una clase no define explícitamente ningún constructor, el compilador de Java proporciona un constructor por defecto sin argumentos que no hace nada.

¿Se puede heredar un constructor? No, los constructores no se heredan. Las subclases no heredan los constructores de su superclase. Esta es una regla fundamental en Java y otras lenguas orientadas a objetos.
Sin embargo, una subclase **puede y debe llamar** a un constructor de su superclase. Esta llamada se realiza implícita o explícitamente desde el constructor de la subclase.

- **Llamada Implícita:** Si el constructor de la subclase no llama explícitamente a un constructor de la superclase, el compilador inserta automáticamente una llamada a super(); como la primera declaración. Esto invoca al constructor sin argumentos de la superclase. Si la superclase no tiene un constructor sin argumentos, esto resultará en un error de compilación.
- **Llamada Explícita:** Se puede llamar explícitamente a un constructor de la superclase usando la palabra clave super(...) con los argumentos apropiados. Esta debe ser la **primera declaración** en el constructor de la subclase.

Este mecanismo asegura que un objeto se construya correctamente desde la base de su jerarquía de herencia hacia abajo.

11. ¿Cuál es la diferencia entre == y el método equals()?

Esta es una pregunta fundamental que pone a prueba la comprensión del candidato sobre la igualdad de objetos frente a la igualdad de referencias.

- **Operador ==:**
    - Para **tipos primitivos** (int, char, boolean, etc.), el operador == compara los **valores literales**. Devuelve true si los valores son idénticos.
    - Para **tipos de referencia** (objetos), el operador == compara las **direcciones de memoria** de las referencias. Devuelve true solo si las dos referencias apuntan exactamente al **mismo objeto** en el heap.
- **Método equals():**

    - El método equals() está definido en la clase java.lang.Object y, por lo tanto, está disponible para todos los objetos.
    - La implementación por defecto en la clase Object se comporta exactamente como el operador ==; es decir, compara las referencias de los objetos.
    - Sin embargo, muchas clases en la biblioteca de Java (como String, Integer, Date y las clases de colección) **sobrescriben** el método equals() para implementar la **igualdad lógica** o de contenido. En este caso, equals() compara los **valores o el estado** de los objetos, no sus referencias. Por ejemplo, new String("test").equals(new String("test")) devuelve true porque el contenido de las cadenas es el mismo, aunque sean dos objetos diferentes en el heap.

Contrato de equals() y hashCode(): Cuando se sobrescribe el método equals(), es imperativo sobrescribir también el método hashCode(). El contrato establece que si dos objetos son iguales según equals(), deben tener el mismo hashCode(). No cumplir con este contrato romperá el funcionamiento de las colecciones basadas en hash como HashMap y HashSet.

Tabla de Resumen:

Comparación	Operador ==	Método equals()
Tipos Primitivos	Compara valores.	No aplicable (no es un método).
Tipos de Referencia	Compara direcciones de memoria (igualdad de referencia).	Por defecto, compara referencias. Sobrescrito, compara el contenido o estado del objeto (igualdad lógica).
Ejemplo String	new String("a") == new String("a") es false.	new String("a").equals(new String("a")) es true.

12. ¿Puede ser estático el método main? ¿Por qué?

Sí, el método main en Java **debe** ser declarado como public static void main(String args). La razón de cada una de estas palabras clave es fundamental para entender cómo la JVM inicia una aplicación.

- **public:** El método main es el punto de entrada de la aplicación. Debe ser public para que la Java Virtual Machine (JVM), que es un programa externo, pueda llamarlo desde fuera de la clase. Si fuera private, protected o tuviera el modificador de acceso por defecto, la JVM no tendría los permisos necesarios para invocarlo.
- **static:** La palabra clave static es crucial. Permite que la JVM llame al método main **sin necesidad de crear una instancia (un objeto) de la clase que lo contiene**. Cuando la JVM inicia, carga la clase en memoria pero no crea ningún objeto de ella. Si main no fuera estático, la JVM tendría que crear un objeto de la clase primero para poder llamar al método, lo que crearía un problema de "el huevo y la gallina": ¿cómo se llamaría al constructor para crear el objeto si el programa aún no ha comenzado a ejecutarse? Hacer main estático resuelve este problema al asociarlo con la clase misma, no con una instancia.
- **void:** El método main no devuelve ningún valor al llamador (la JVM). Su propósito es iniciar la ejecución del programa. Si necesitara devolver un código de salida, se usaría System.exit(int status).
- **String args:** Este es un array de String que permite pasar argumentos a la aplicación desde la línea de comandos.

Si el método main no se declara como static, el programa compilará sin errores, pero la JVM no

podrá encontrarlo como el punto de entrada y lanzará un error en tiempo de ejecución (NoSuchMethodError: main) o simplemente no se ejecutará, indicando que no se encontró el método principal.

13. ¿Qué es una clase anónima?

Una clase anónima es una clase interna que no tiene nombre. Se declara y se instancia en una sola expresión. Se utiliza típicamente cuando se necesita crear una instancia de un objeto con una pequeña modificación o implementación de un método, sin la necesidad de crear una subclase separada y con nombre.
Las clases anónimas son útiles para:

1. **Implementar interfaces:** Especialmente interfaces con un solo método (conocidas como interfaces funcionales antes de Java 8), como Runnable, ActionListener o Comparator.
2. **Extender clases:** Para sobrescribir métodos de una clase existente de forma concisa.

Características:

- **Sin nombre:** No tienen un nombre de clase, lo que las hace "anónimas".
- **Expresión única:** Se definen y se instancian en el mismo lugar.
- **Acceso a variables:** Pueden acceder a los miembros de su clase contenedora. También pueden acceder a las variables locales del ámbito en el que se declaran, siempre que estas sean final o efectivamente final (no modificadas después de la inicialización).
- **Limitaciones:** No pueden tener constructores explícitos (ya que no tienen nombre), pero pueden tener un bloque de inicialización de instancia. No pueden ser static.

Ejemplo (antes de Java 8):

```
// Implementando la interfaz Runnable con una clase anónima
Thread t = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Ejecutando en un nuevo hilo.");
    }
});
t.start();
```

Con la introducción de las **expresiones lambda en Java 8**, el uso de clases anónimas para implementar interfaces funcionales se ha vuelto menos común, ya que las lambdas ofrecen una sintaxis mucho más concisa y clara. El ejemplo anterior se puede reescribir con una lambda como:

```
Thread t = new Thread(() -> System.out.println("Ejecutando en un nuevo
hilo."));
t.start();
```

Sin embargo, las clases anónimas siguen siendo necesarias si se necesita extender una clase o implementar una interfaz con más de un método abstracto.

14. ¿Qué es una clase interna (Inner Class) y cuáles son sus tipos?

Una clase interna (o anidada) es una clase definida dentro de otra clase. Agrupar clases de esta manera es una forma lógica de organizar el código, ya que permite que una clase sea un miembro de otra. Esto aumenta la encapsulación y conduce a un código más legible y

mantenible.
Existen cuatro tipos de clases anidadas en Java:

1. **Clase Interna Miembro (Member Inner Class):**
   - Es una clase no estática definida a nivel de miembro de otra clase.
   - Cada instancia de una clase interna miembro está asociada a una instancia de la clase externa.
   - Puede acceder a todos los miembros (incluidos los privados) de la clase externa.
   - No puede tener miembros estáticos (a menos que sean constantes static final).

```
class Externa {
    private int datoExterno = 10;
    class Interna {
        void mostrar() {
            System.out.println("Dato externo: " + datoExterno);
        }
    }
}
// Para instanciarla: Externa.Interna interna = new Externa().new
Interna();
```

2. **Clase Anidada Estática (Static Nested Class):**
   - Es una clase declarada como static dentro de otra clase.
   - No está asociada a una instancia de la clase externa, por lo que no puede acceder a los miembros de instancia de la clase externa directamente. Solo puede acceder a los miembros estáticos.
   - Se comporta más como una clase de nivel superior que ha sido anidada por conveniencia de empaquetado.

```
class Externa {
    static int datoEstatico = 20;
    static class AnidadaEstatica {
        void mostrar() {
            System.out.println("Dato estático: " + datoEstatico);
        }
    }
}
// Para instanciarla: Externa.AnidadaEstatica anidada = new
Externa.AnidadaEstatica();
```

3. **Clase Interna Local (Local Inner Class):**
   - Es una clase definida dentro de un bloque de código, típicamente dentro de un método.
   - Su alcance está limitado al bloque en el que se define.
   - Puede acceder a los miembros de la clase externa y a las variables locales del método (siempre que sean final o efectivamente final).

```
class Externa {
    void miMetodo() {
        class InternaLocal {
            void saludar() {
                System.out.println("Hola desde la clase interna
```

```
local.");
            }
        }
        InternaLocal local = new InternaLocal();
        local.saludar();
    }
}
```

4. **Clase Interna Anónima (Anonymous Inner Class):**
    - Como se describió en la pregunta anterior, es una clase interna sin nombre que se declara e instancia en una sola expresión. Se utiliza para implementaciones o extensiones de un solo uso.

15. ¿Qué son las clases Wrapper y por qué son necesarias?

Las **clases Wrapper** (envoltorio) en Java son clases que encapsulan los ocho tipos de datos primitivos en objetos. Cada tipo primitivo tiene una clase wrapper correspondiente en el paquete java.lang: Byte, Short, Integer, Long, Float, Double, Character y Boolean.
Son necesarias por varias razones fundamentales:

1. **Uso en Colecciones:** El Java Collections Framework (como ArrayList, HashMap, etc.) solo puede almacenar objetos, no tipos primitivos. Las clases wrapper permiten almacenar valores primitivos en estas colecciones. Por ejemplo, no se puede hacer ArrayList<int>, pero sí ArrayList<Integer>.
2. **Soporte de Nulidad:** Los tipos primitivos no pueden tener un valor null. Deben tener un valor por defecto (p. ej., 0 para int). Las clases wrapper, al ser objetos, pueden ser null, lo que es útil para representar la ausencia de un valor en ciertos escenarios, como en las bases de datos.
3. **Provisión de Métodos de Utilidad:** Las clases wrapper proporcionan una variedad de métodos de utilidad para conversiones y operaciones. Por ejemplo, la clase Integer tiene métodos como parseInt(String s) para convertir una cadena a un entero, toString() para convertir un entero a cadena, y constantes como Integer.MAX_VALUE.
4. **Necesidad en APIs que trabajan con Objetos:** Muchas APIs en Java, especialmente en áreas como la reflexión (Reflection) o la serialización, están diseñadas para trabajar con objetos. Las clases wrapper permiten que los valores primitivos se utilicen en estos contextos.

Autoboxing y Unboxing: A partir de Java 5, el proceso de conversión entre tipos primitivos y sus clases wrapper se automatizó a través del **autoboxing** y **unboxing**.

- **Autoboxing:** Es la conversión automática que el compilador de Java realiza de un tipo primitivo a su correspondiente clase wrapper.
  ```
  Integer i = 100; // Autoboxing de int a Integer
  ```

- **Unboxing:** Es el proceso inverso, la conversión automática de un objeto wrapper a su valor primitivo.
  ```
  int j = i; // Unboxing de Integer a int
  ```

Esta característica simplifica enormemente el código, haciendo que el uso de clases wrapper sea casi transparente.

16. ¿Qué es la inmutabilidad y cómo se crea una clase inmutable en Java?

La **inmutabilidad** es la propiedad de un objeto cuyo estado no puede ser modificado después de su creación. La clase String es el ejemplo más conocido de una clase inmutable en Java.
Beneficios de la Inmutabilidad:

- **Seguridad para Subprocesos (Thread Safety):** Los objetos inmutables son intrínsecamente seguros para ser compartidos entre múltiples hilos sin necesidad de sincronización, ya que su estado no puede cambiar.
- **Simplicidad:** El razonamiento sobre el código es más sencillo, ya que no hay que preocuparse de que el estado de un objeto cambie inesperadamente.
- **Uso como Claves en Mapas:** Son excelentes candidatos para ser claves en HashMap o HashSet, ya que su hashCode() se puede calcular una vez y almacenar en caché, y no cambiará durante la vida del objeto.
- **Seguridad:** Ayuda a prevenir la corrupción de datos al evitar que el estado del objeto sea modificado por código no autorizado.

Pasos para crear una clase inmutable:

1. **Declarar la clase como final:** Esto evita que otras clases la extiendan y modifiquen su comportamiento.
2. **Hacer todos los campos private y final:** private evita el acceso directo desde fuera de la clase. final asegura que los campos solo se puedan asignar una vez, generalmente en el constructor.
3. **No proporcionar métodos "setter":** No debe haber métodos que modifiquen el estado de los campos.
4. **Inicializar todos los campos en el constructor:** Todos los campos final deben ser inicializados en el constructor.
5. **Realizar copias defensivas para campos mutables:** Si la clase contiene referencias a objetos mutables (como Date o ArrayList), se deben tomar precauciones especiales:
    - En el constructor, no asigne directamente el objeto mutable pasado como argumento. En su lugar, cree una **copia profunda** (*deep copy*) de él.
    - En los métodos "getter" que devuelven un campo mutable, no devuelva la referencia al objeto interno. En su lugar, devuelva una **copia profunda** de él.

Ejemplo de una clase inmutable:

```
import java.util.Date;

public final class PeriodoInmutable {

    private final Date inicio;
    private final Date fin;

    public PeriodoInmutable(Date inicio, Date fin) {
        // Copia defensiva en el constructor
        this.inicio = new Date(inicio.getTime());
        this.fin = new Date(fin.getTime());

        // Comprobación de validez
        if (this.inicio.compareTo(this.fin) > 0) {
```

```
            throw new IllegalArgumentException(inicio + " después de "
+ fin);
        }
    }

    public Date getInicio() {
        // Devolver copia defensiva para evitar la modificación
externa
        return new Date(inicio.getTime());
    }

    public Date getFin() {
        // Devolver copia defensiva
        return new Date(fin.getTime());
    }
}
```

17. ¿Qué es el polimorfismo en tiempo de ejecución y cómo se logra en Java?

El **polimorfismo en tiempo de ejecución**, también conocido como **polimorfismo dinámico** o **despacho dinámico de métodos** (*dynamic method dispatch*), es la capacidad de la JVM de decidir en tiempo de ejecución qué implementación de un método sobrescrito se debe llamar, basándose en el tipo real del objeto en lugar del tipo de la variable de referencia.
Es uno de los conceptos más poderosos de la POO, ya que permite escribir código flexible y extensible que puede trabajar con objetos de múltiples subtipos de una manera genérica.
Se logra en Java a través de la sobrescritura de métodos (*method overriding*):

1. **Herencia:** Debe existir una relación de herencia, con una superclase y al menos una subclase.
2. **Sobrescritura de Métodos:** La subclase debe proporcionar su propia implementación de un método que ya está presente en la superclase. La firma del método (nombre y parámetros) debe ser idéntica, y el tipo de retorno debe ser el mismo o un subtipo (covariante).
3. **Referencia de Superclase a Objeto de Subclase:** Una variable de referencia del tipo de la superclase se utiliza para apuntar a un objeto de la subclase.

Cómo funciona (Despacho Dinámico de Métodos): Cuando se llama a un método sobrescrito a través de una referencia de superclase, Java no decide qué método ejecutar en tiempo de compilación. En su lugar, espera hasta el tiempo de ejecución. En ese momento, la JVM verifica el tipo real del objeto al que apunta la referencia y llama a la versión del método que pertenece a esa clase específica.
Ejemplo:

```
class Forma {
    void dibujar() {
        System.out.println("Dibujando una forma genérica");
    }
}
```

```
class Circulo extends Forma {
    @Override
    void dibujar() {
        System.out.println("Dibujando un círculo");
    }
}

class Cuadrado extends Forma {
    @Override
    void dibujar() {
        System.out.println("Dibujando un cuadrado");
    }
}

public class TestPolimorfismo {
    public static void main(String args) {
        Forma miForma;

        miForma = new Forma();
        miForma.dibujar(); // Llama al método de Forma

        miForma = new Circulo();
        miForma.dibujar(); // Llama al método de Circulo en tiempo de
ejecución

        miForma = new Cuadrado();
        miForma.dibujar(); // Llama al método de Cuadrado en tiempo de
ejecución
    }
}
```

En este ejemplo, aunque la variable miForma es de tipo Forma, la llamada a miForma.dibujar() invoca la implementación de Circulo o Cuadrado dependiendo del objeto que se le asignó en tiempo de ejecución. Esto es el despacho dinámico de métodos en acción.

18. ¿Cuál es la diferencia entre una copia superficial (shallow copy) y una copia profunda (deep copy)?

Esta pregunta evalúa la comprensión del candidato sobre la clonación de objetos y el manejo de referencias, lo cual es crucial cuando se trabaja con objetos complejos que contienen otros objetos.
Copia Superficial (Shallow Copy): Una copia superficial de un objeto crea un nuevo objeto, pero no crea copias de los objetos a los que se hace referencia dentro del objeto original. En su lugar, copia las referencias a esos objetos.

- **Comportamiento:**
    - Crea un nuevo objeto en el heap.

    - Copia el valor de los campos de tipo primitivo del objeto original al nuevo objeto.
    - Para los campos que son tipos de referencia, copia la **dirección de memoria** (la referencia), no el objeto en sí.
- **Consecuencia:** Tanto el objeto original como su copia superficial apuntarán a los **mismos objetos internos**. Si se modifica el estado de uno de estos objetos internos a través de la copia, el cambio también será visible en el objeto original, y viceversa.
- **Implementación:** El método clone() por defecto de la clase Object realiza una copia superficial.

Copia Profunda (Deep Copy): Una copia profunda de un objeto crea un nuevo objeto y, recursivamente, crea copias de todos los objetos a los que se hace referencia dentro del objeto original.

- **Comportamiento:**
    - Crea un nuevo objeto en el heap.
    - Copia el valor de los campos de tipo primitivo.
    - Para los campos que son tipos de referencia, crea **nuevas copias** de los objetos referenciados y luego asigna las referencias de estos nuevos objetos a los campos de la copia.
- **Consecuencia:** El objeto original y su copia profunda son **completamente independientes**. Los cambios realizados en los objetos internos de la copia no afectarán al objeto original.
- **Implementación:** No hay un mecanismo automático en Java para la copia profunda. Debe ser implementada manualmente, a menudo sobrescribiendo el método clone() para que también clone los objetos internos, o utilizando un constructor de copia, o mediante serialización/deserialización.

Ejemplo Visual: Imaginemos un objeto Universidad que contiene una lista de objetos Estudiante.

- **Copia Superficial:** Se crea una nueva Universidad, pero su lista de estudiantes es la misma referencia que la de la universidad original. Si se agrega un estudiante a la lista de la copia, también aparecerá en la lista de la original.
- **Copia Profunda:** Se crea una nueva Universidad, se crea una nueva lista, y cada Estudiante de la lista original se clona y se agrega a la nueva lista. Las dos universidades son totalmente independientes.

19. ¿Qué son los especificadores de acceso en Java?

Los especificadores de acceso (o modificadores de acceso) son palabras clave en Java que definen el nivel de visibilidad y accesibilidad de clases, métodos y variables. Son un pilar fundamental de la encapsulación, ya que permiten controlar qué partes del código pueden acceder a otras.

Hay cuatro especificadores de acceso en Java:

1. **public:**
    - **Visibilidad:** La más alta. Un miembro public es accesible desde cualquier otra clase, en cualquier paquete.
    - **Uso:** Se utiliza para las APIs públicas de una clase, es decir, los métodos y campos que se pretende que sean utilizados por código externo.
2. **protected:**
    - **Visibilidad:** Un miembro protected es accesible dentro de su propio paquete y por subclases en otros paquetes.

    - **Uso:** Se utiliza para permitir que las subclases accedan y modifiquen el comportamiento de la superclase, sin exponer esos miembros al resto del mundo. Es una herramienta clave para el diseño de herencia.
3. **default (o Package-Private):**
    - **Visibilidad:** Este es el nivel de acceso por defecto si no se especifica ningún modificador. Un miembro con acceso por defecto solo es visible para las clases dentro del **mismo paquete**. No es accesible desde fuera del paquete, ni siquiera por subclases.
    - **Uso:** Útil para ocultar detalles de implementación a nivel de paquete, permitiendo que las clases de un mismo paquete colaboren estrechamente.
4. **private:**
    - **Visibilidad:** La más restrictiva. Un miembro private solo es accesible dentro de la misma clase en la que se declara.
    - **Uso:** Es la base de la encapsulación y la ocultación de datos. Se utiliza para los detalles de implementación internos de una clase que no deben ser expuestos.

Regla de Oro: Siempre se debe usar el especificador de acceso más restrictivo posible que permita que el código funcione. Esto maximiza la encapsulación y reduce el acoplamiento entre componentes.

20. ¿Qué significa "programar para una interfaz, no para una implementación"?

Este es un principio de diseño fundamental en la programación orientada a objetos, popularizado por el libro "Design Patterns: Elements of Reusable Object-Oriented Software" (el libro de la "Banda de los Cuatro" o GoF).
Significa que el código debe depender de abstracciones (interfaces o clases abstractas) en lugar de depender de implementaciones concretas. En la práctica, esto se traduce en declarar variables y parámetros de métodos con el tipo de la interfaz, en lugar del tipo de la clase concreta que la implementa.
Ejemplo:
Mal (programar para la implementación):

```
ArrayList<String> miLista = new ArrayList<>();
miLista.add("Hola");
// Este código está fuertemente acoplado a ArrayList.
// Si mañana decidimos que LinkedList es más eficiente,
// tendríamos que cambiar el tipo de la variable en todas partes.
```

Bien (programar para la interfaz):

```
List<String> miLista = new ArrayList<>(); // o new LinkedList<>();
miLista.add("Hola");
// Este código depende de la interfaz List.
// Podemos cambiar la implementación (de ArrayList a LinkedList)
// sin cambiar el resto del código que usa miLista,
// siempre que solo use métodos definidos en la interfaz List.
```

Beneficios de este principio:

1. **Bajo Acoplamiento:** El código cliente no está acoplado a los detalles de una

implementación específica. Esto hace que el sistema sea más flexible y fácil de mantener.
2. **Flexibilidad y Extensibilidad:** Es fácil cambiar una implementación por otra (como cambiar ArrayList por LinkedList) sin afectar al código cliente. También es fácil introducir nuevas implementaciones de la interfaz en el futuro.
3. **Facilidad de Prueba:** Al depender de interfaces, se pueden crear fácilmente objetos simulados (*mocks* o *stubs*) para las pruebas unitarias. Esto permite probar los componentes de forma aislada.
4. **Promueve el Polimorfismo:** Fomenta el uso del polimorfismo, ya que el código puede operar sobre cualquier objeto que implemente la interfaz, sin conocer su tipo concreto.

Este principio es la base de muchos patrones de diseño y marcos de trabajo, como el de Inyección de Dependencias (utilizado masivamente en Spring), donde los componentes se "cablean" juntos a través de sus interfaces.

Sección II: El Java Collections Framework

Esta sección se sumerge en una de las APIs más utilizadas y probadas del JDK. Las preguntas aquí evalúan el conocimiento del candidato sobre estructuras de datos, sus características de rendimiento y los contratos que gobiernan su comportamiento.

21. Describa la jerarquía principal del Java Collections Framework.

El Java Collections Framework (JCF) es una arquitectura unificada para representar y manipular colecciones. Su jerarquía se basa en un conjunto de interfaces clave que definen los diferentes tipos de colecciones.
La jerarquía principal se puede resumir de la siguiente manera:

1. **Iterable<T>:** Es la interfaz raíz de toda la jerarquía. Su único método, iterator(), garantiza que cualquier colección que la implemente pueda ser recorrida. La instrucción for-each de Java funciona con cualquier objeto que implemente Iterable.
2. **Collection<E>:** Es la interfaz principal del framework, que extiende a Iterable. Representa un grupo de objetos, conocidos como sus elementos. Define los comportamientos básicos comunes a la mayoría de las colecciones, como add(), remove(), contains(), size() e isEmpty(). No tiene implementaciones directas, pero es extendida por otras interfaces más específicas.
3. **List<E>:** Extiende a Collection. Es una **colección ordenada** (también conocida como secuencia) que permite **elementos duplicados**. Los elementos se pueden acceder por su índice entero (posición en la lista).
    - **Implementaciones clave:** ArrayList, LinkedList, Vector, Stack.
4. **Set<E>:** Extiende a Collection. Es una colección que **no permite elementos duplicados**. Modela la abstracción matemática de un conjunto. La mayoría de las implementaciones no garantizan un orden específico de los elementos.
    - **Implementaciones clave:** HashSet, LinkedHashSet, TreeSet.
5. **Queue<E>:** Extiende a Collection. Está diseñada para mantener elementos antes de su procesamiento, típicamente en un orden **FIFO (First-In, First-Out)**. Proporciona operaciones adicionales para inserción, extracción e inspección.
    - **Implementaciones clave:** LinkedList, PriorityQueue, ArrayDeque.
6. **Map<K, V>:** Esta interfaz **no extiende Collection**, ya que representa una estructura de

datos diferente: un objeto que mapea claves a valores. Cada clave puede mapear a, como máximo, un valor, y no puede haber claves duplicadas.
    - **Implementaciones clave:** HashMap, Hashtable, LinkedHashMap, TreeMap.

Relaciones importantes:

- List, Set y Queue son los tres subtipos principales de Collection.
- Map es una interfaz de alto nivel separada, pero se considera parte del framework. Proporciona vistas de colección de sus claves, valores o entradas a través de los métodos keySet(), values() y entrySet().

Esta estructura jerárquica permite a los programadores trabajar con colecciones de manera abstracta (programando para la interfaz) y cambiar las implementaciones concretas según las necesidades de rendimiento y comportamiento sin alterar el código cliente.

22. ¿Cuál es la diferencia entre Collection y Collections?

Esta es una pregunta de precisión para verificar si el candidato distingue entre la interfaz fundamental y la clase de utilidad.

- **Collection:**
    - **Tipo:** Es una **interfaz** (java.util.Collection).
    - **Propósito:** Es la **interfaz raíz** de la jerarquía de colecciones (excluyendo los mapas). Define el contrato básico para todas las colecciones de tipo List, Set y Queue. Especifica métodos fundamentales como add(), remove(), contains(), size(), etc..
    - **Uso:** Se utiliza para declarar variables y parámetros de métodos de forma polimórfica, permitiendo que el código trabaje con cualquier tipo de colección que implemente esta interfaz.
- **Collections:**
    - **Tipo:** Es una **clase de utilidad final** (java.util.Collections).
    - **Propósito:** Proporciona **métodos estáticos de utilidad** para operar sobre o devolver colecciones. No se puede instanciar (su constructor es privado).
    - **Uso:** Ofrece una amplia gama de funcionalidades, entre las que se incluyen:
        - **Algoritmos:** sort(), binarySearch(), reverse(), shuffle(), max(), min().
        - **Wrappers:** Métodos para hacer que una colección sea de solo lectura (unmodifiableList(), unmodifiableSet(), etc.) o segura para subprocesos (synchronizedList(), synchronizedSet(), etc.).
        - **Constructores de conveniencia:** emptyList(), singleton(), etc.

En resumen, Collection es el **plano** (la interfaz) que define lo que es una colección, mientras que Collections es la **caja de herramientas** (la clase de utilidad) que proporciona herramientas para trabajar con esas colecciones.

23. Explique la diferencia entre iteradores fail-fast y fail-safe.

Esta pregunta evalúa el conocimiento sobre la concurrencia en el contexto de las colecciones y la gestión de modificaciones inesperadas durante la iteración.

Iteradores Fail-Fast:

- **Comportamiento:** Un iterador fail-fast lanza una excepción java.util.ConcurrentModificationException si la colección subyacente es **modificada estructuralmente** (se añaden o eliminan elementos) en cualquier momento después de la creación del iterador, por cualquier medio que no sea el propio método remove() del

iterador.

- **Mecanismo:** La mayoría de las implementaciones de colecciones no concurrentes (ArrayList, HashMap, HashSet) utilizan un contador de modificaciones interno (a menudo llamado modCount). Cada vez que la colección se modifica estructuralmente, este contador se incrementa. Cuando se crea un iterador, este guarda el valor actual del modCount. En cada operación del iterador (next(), hasNext()), se compara el modCount actual de la colección con el valor guardado. Si son diferentes, se lanza la excepción.
- **Propósito:** No garantizan que la excepción se lance en todas las circunstancias, pero hacen un "mejor esfuerzo" para detectar problemas de concurrencia. Su objetivo es fallar rápidamente y de forma limpia en lugar de arriesgarse a un comportamiento no determinista y a datos corruptos.
- **Ejemplos:** Los iteradores devueltos por ArrayList, HashMap, HashSet, Vector.

Iteradores Fail-Safe (o más precisamente, débilmente consistentes):

- **Comportamiento:** Un iterador fail-safe no lanza ConcurrentModificationException. Permiten modificaciones concurrentes en la colección subyacente mientras se itera sobre ella.
- **Mecanismo:** Típicamente, funcionan operando sobre un **clon o una copia** de la colección original. La iteración se realiza sobre esta copia, por lo que las modificaciones en la colección original no afectan al iterador.
- **Propósito:** Proporcionar una iteración segura en entornos concurrentes. Sin embargo, tienen una desventaja: el iterador podría no reflejar el estado más actual de la colección. Es decir, no verán las modificaciones realizadas después de que se creó el iterador.
- **Ejemplos:** Los iteradores devueltos por las clases del paquete java.util.concurrent, como ConcurrentHashMap y CopyOnWriteArrayList.

Característica	Fail-Fast	Fail-Safe
Excepción	Lanza ConcurrentModificationException ante modificaciones concurrentes.	No lanza excepciones por modificaciones concurrentes.
Fuente de Iteración	Itera sobre la colección original.	Itera sobre una copia o clon de la colección.
Visibilidad de Cambios	La iteración se detiene.	No refleja los cambios realizados después de su creación.
Uso de Memoria	Bajo.	Alto (debido a la copia).
Clases Típicas	ArrayList, HashMap	ConcurrentHashMap, CopyOnWriteArrayList

24. Compare ArrayList y LinkedList en términos de estructura interna, rendimiento y casos de uso.

Esta es una pregunta clásica que evalúa la comprensión de las estructuras de datos subyacentes y sus implicaciones en el rendimiento.

ArrayList

- **Estructura Interna:** Se basa en un **array dinámico**. Almacena los elementos en un array contiguo en la memoria. Si el array se llena, se crea un nuevo array más grande (generalmente un 50% más grande) y los elementos del array antiguo se copian al nuevo.

- **Rendimiento:**
    - **Acceso a elementos (get(index)):** Muy rápido, con un tiempo de complejidad de **O(1)**. Esto se debe a que puede calcular directamente la ubicación del elemento en la memoria a partir del índice.
    - **Añadir/Eliminar al final (add(element)):** Generalmente rápido, **O(1) amortizado**. Es O(1) si hay espacio en el array, pero puede ser O(n) si se necesita redimensionar el array.
    - **Añadir/Eliminar en el medio o al principio (add(index, element), remove(index)):** Lento, con un tiempo de complejidad de **O(n)**. Requiere desplazar todos los elementos posteriores para hacer espacio o llenar el hueco.
- **Uso de Memoria:** Más eficiente en términos de memoria, ya que solo almacena los elementos y una pequeña sobrecarga para la capacidad del array.
- **Caso de Uso Ideal:** Cuando la operación principal es el **acceso aleatorio y la iteración** sobre la lista, y las inserciones/eliminaciones son poco frecuentes o se realizan principalmente al final de la lista.

LinkedList

- **Estructura Interna:** Se basa en una **lista doblemente enlazada**. Cada elemento (o nodo) contiene el dato, una referencia al nodo anterior y una referencia al nodo siguiente.
- **Rendimiento:**
    - **Acceso a elementos (get(index)):** Lento, con un tiempo de complejidad de **O(n)**. Para encontrar un elemento en una posición específica, debe recorrer la lista desde el principio (o el final, lo que esté más cerca) hasta llegar al índice deseado.
    - **Añadir/Eliminar al principio o al final (addFirst(), addLast(), removeFirst(), removeLast()):** Muy rápido, **O(1)**. Solo implica actualizar las referencias de los nodos adyacentes.
    - **Añadir/Eliminar en el medio (add(index, element), remove(index)):** Lento en la búsqueda (**O(n)** para encontrar el nodo), pero rápido en la inserción/eliminación una vez encontrado el nodo (**O(1)**).
- **Uso de Memoria:** Menos eficiente en términos de memoria, ya que cada nodo tiene una sobrecarga adicional para almacenar las referencias al nodo anterior y al siguiente.
- **Caso de Uso Ideal:** Cuando la operación principal son las **inserciones y eliminaciones frecuentes** en cualquier parte de la lista, especialmente al principio y al final. Actúa eficientemente como una cola (Queue) o pila (Stack).

25. Explique cómo funciona HashMap internamente y el contrato entre hashCode() y equals().

Esta es una pregunta profunda que separa a los candidatos que simplemente usan HashMap de los que entienden su funcionamiento interno, lo cual es crucial para el rendimiento y la corrección.

Funcionamiento Interno de HashMap: HashMap utiliza una estructura de datos interna llamada **tabla hash**, que es básicamente un array de "cubetas" o "buckets". Cada cubeta es, en su forma más simple, una lista enlazada (o un árbol a partir de Java 8).

El proceso de put(key, value) funciona así:

1. **Calcular el hashCode:** Primero, se llama al método hashCode() de la clave (key). Este método devuelve un valor entero.
2. **Calcular el índice de la cubeta:** Este hashCode se somete a una función de hash

interna para calcular un índice dentro del array de cubetas. Esto determina en qué cubeta se almacenará el par clave-valor. La fórmula suele ser algo como index = hashCode & (n - 1), donde n es la capacidad del array.

3. **Manejo de Colisiones:**
    - **Si la cubeta está vacía:** Se crea un nuevo nodo (Entry o Node) que contiene la clave, el valor, el hash y una referencia nula al siguiente, y se coloca en esa cubeta.
    - **Si la cubeta no está vacía (colisión):** Esto significa que otra clave ya ha generado el mismo índice. El nuevo nodo se añade a la estructura de esa cubeta.
        - Se recorre la lista enlazada (o el árbol) en la cubeta. Para cada nodo existente, se compara la clave del nuevo nodo con la clave del nodo existente usando el método equals().
        - Si se encuentra una clave igual (equals() devuelve true), se reemplaza el valor antiguo por el nuevo y se devuelve el valor antiguo.
        - Si no se encuentra ninguna clave igual después de recorrer toda la estructura, el nuevo nodo se añade al final de la lista enlazada.
4. **Treeification (a partir de Java 8):** Si el número de nodos en una cubeta supera un cierto umbral (TREEIFY_THRESHOLD, por defecto 8), la lista enlazada se convierte en un **árbol rojo-negro** para mejorar el rendimiento de búsqueda en caso de muchas colisiones, pasando de O(n) a O(\log n).

El proceso de get(key) sigue una lógica similar: calcula el hash, encuentra la cubeta y luego busca la clave correcta dentro de esa cubeta usando equals().

El Contrato entre hashCode() y equals(): Este contrato es **esencial** para el correcto funcionamiento de todas las colecciones basadas en hash.

1. **Consistencia de equals():** Durante la vida de un objeto, si los campos utilizados en la comparación equals() no se modifican, equals() debe devolver consistentemente el mismo resultado.
2. **Consistencia de hashCode():** Durante la vida de un objeto, hashCode() debe devolver consistentemente el mismo valor entero, siempre que no se modifique la información utilizada en las comparaciones equals().
3. **La Regla Fundamental: Si dos objetos son iguales según el método equals(), entonces deben tener el mismo valor hashCode()**.
4. **La Inversa no es Obligatoria:** Si dos objetos tienen el mismo valor hashCode(), no están obligados a ser iguales según equals(). Esto es una colisión y es manejada por la estructura de datos.

Consecuencias de Romper el Contrato:

- Si se rompe la regla 3 (objetos iguales con hashCode diferentes), el HashMap se romperá. Al intentar recuperar un objeto (get(key)), se podría buscar en una cubeta incorrecta (basada en el hashCode de la clave de búsqueda) y no encontrar el objeto, aunque esté presente en el mapa.

26. Compare HashMap, Hashtable y ConcurrentHashMap.

Esta pregunta evalúa el conocimiento del candidato sobre las diferentes implementaciones de mapas, especialmente en el contexto de la concurrencia.

Característica	HashMap	Hashtable	ConcurrentHashMap
Seguridad para Hilos	No es seguro para hilos	Es seguro para hilos	Es seguro para hilos

Característica	HashMap	Hashtable	ConcurrentHashMap
	(no sincronizado).	(sincronizado).	(sincronizado).
Mecanismo de Sincronización	Ninguno. Se debe sincronizar externamente si se usa en un entorno multihilo (p. ej., Collections.synchronizedMap(new HashMap<>())).	Sincroniza **cada método público**. Esto significa que solo un hilo puede acceder al mapa a la vez, lo que crea un cuello de botella de rendimiento.	Utiliza un mecanismo de bloqueo más sofisticado y granular. En versiones antiguas, usaba **bloqueo por segmentos** (*segment locking*). En Java 8+, usa **bloqueo a nivel de nodo** (*node-level locking*), lo que permite una alta concurrencia.
Manejo de Nulos	Permite **una clave nula** y **múltiples valores nulos**.	**No permite** ni claves ni valores nulos. Lanza NullPointerException.	**No permite** ni claves ni valores nulos. Lanza NullPointerException.
Iterador	El iterador es **fail-fast**. Lanza ConcurrentModificationException si el mapa se modifica durante la iteración.	El iterador (a través de Enumeration) no es fail-fast. Los iteradores de sus vistas de colección son fail-fast.	El iterador es **fail-safe** (o débilmente consistente). No lanza ConcurrentModificationException y refleja el estado del mapa en un momento dado, pero no necesariamente las actualizaciones posteriores.
Rendimiento	Muy alto en entornos de un solo hilo.	Bajo en entornos de alta concurrencia debido al bloqueo de todo el mapa en cada operación.	Muy alto en entornos de alta concurrencia, ya que permite lecturas concurrentes y escrituras concurrentes en diferentes partes del mapa.
Herencia y Legado	Introducido en Java 1.2. Hereda de AbstractMap.	Parte del framework original de Java (legado). Hereda de Dictionary.	Introducido en Java 1.5. Hereda de AbstractMap.

Conclusión:

- Use HashMap para aplicaciones de un solo hilo.
- Evite Hashtable en nuevo código; es una clase legada con un rendimiento de concurrencia pobre.
- Use ConcurrentHashMap para aplicaciones multihilo de alto rendimiento. Es la opción estándar para mapas concurrentes.

27. ¿Cuál es la diferencia entre HashSet y TreeSet?

Esta pregunta compara dos de las implementaciones de Set más comunes, centrándose en el

orden, el rendimiento y los requisitos de los elementos.

- **HashSet:**
  - **Orden: No garantiza ningún orden** de los elementos. El orden puede cambiar con el tiempo.
  - **Implementación Interna:** Utiliza un HashMap internamente para almacenar los elementos. Los elementos del Set son las claves del HashMap, y se utiliza un objeto PRESENT ficticio y compartido como valor.
  - **Rendimiento:** Ofrece un rendimiento de tiempo **constante amortizado (O(1))** para las operaciones básicas (add, remove, contains, size), asumiendo que la función de hash dispersa los elementos adecuadamente entre las cubetas.
  - **Manejo de Nulos:** Permite **un** elemento nulo.
  - **Requisitos del Elemento:** Los objetos almacenados en un HashSet deben implementar correctamente los métodos hashCode() y equals().
- **TreeSet:**
  - **Orden:** Almacena los elementos en un **orden de clasificación**. Por defecto, utiliza el **orden natural** de los elementos (la clase del elemento debe implementar la interfaz Comparable). Alternativamente, se puede proporcionar un Comparator en el momento de la creación para especificar un orden personalizado.
  - **Implementación Interna:** Utiliza un TreeMap internamente, que a su vez se basa en un **árbol rojo-negro** (una estructura de árbol de búsqueda binaria autoequilibrada).
  - **Rendimiento:** El rendimiento para las operaciones básicas es de **O(\log n)**, ya que cada operación implica navegar por el árbol.
  - **Manejo de Nulos: No permite** elementos nulos. Intentar añadir un null lanzará una NullPointerException, ya que no se puede comparar.
  - **Requisitos del Elemento:** Los objetos almacenados deben ser mutuamente comparables, es decir, deben implementar la interfaz Comparable o se debe proporcionar un Comparator.

Cuándo usar cuál:

- Use HashSet cuando necesite un Set y no le importe el orden, y el rendimiento sea la principal prioridad.
- Use TreeSet cuando necesite un Set que mantenga los elementos ordenados. También proporciona métodos de navegación útiles como first(), last(), headSet(), tailSet().

28. Explique las interfaces Comparable y Comparator. ¿Cuándo usaría cada una?

Comparable y Comparator son dos interfaces en Java utilizadas para ordenar objetos. La elección entre ellas depende de si se está definiendo un orden "natural" para una clase o si se necesitan múltiples formas de ordenar o si no se puede modificar la clase original.

Comparable<T>:

- **Propósito:** Se utiliza para definir el **orden natural** de los objetos de una clase.
- **Paquete:** java.lang.
- **Método:** Contiene un solo método: int compareTo(T o).
  - Devuelve un entero negativo si this objeto es menor que o.
  - Devuelve cero si this objeto es igual a o.
  - Devuelve un entero positivo si this objeto es mayor que o.

- **Implementación:** La clase cuyos objetos se van a ordenar **debe implementar** esta interfaz. Esto significa que la lógica de ordenación es parte de la propia clase del objeto.
- **Uso:** Se utiliza cuando solo hay una forma lógica y obvia de ordenar los objetos de una clase. Por ejemplo, la clase String implementa Comparable para ordenar alfabéticamente.
- **Ejemplo:**

```
public class Estudiante implements Comparable<Estudiante> {
    private int id;
    //... otros campos y métodos
    @Override
    public int compareTo(Estudiante otro) {
        return Integer.compare(this.id, otro.id); // Orden natural
por ID
    }
}
// Collections.sort(listaDeEstudiantes); usará este orden natural.
```

Comparator<T>:

- **Propósito:** Se utiliza para definir un **orden personalizado o múltiple** para los objetos de una clase.
- **Paquete:** java.util.
- **Método:** Contiene un método principal: int compare(T o1, T o2).
    - La lógica de retorno es la misma que compareTo, pero compara dos objetos (o1 y o2) pasados como argumentos.
- **Implementación:** Se crea una **clase separada** que implementa la interfaz Comparator. La lógica de ordenación está desacoplada de la clase del objeto.
- **Uso:**
    - Cuando se necesita ordenar objetos de múltiples maneras (p. ej., ordenar Estudiante por nombre, luego por nota).
    - Cuando se quiere ordenar objetos de una clase que **no se puede modificar** (p. ej., una clase de una biblioteca de terceros que no implementa Comparable).
    - Cuando se quiere usar expresiones lambda para una ordenación concisa (a partir de Java 8).
- **Ejemplo:**

```
public class ComparadorEstudiantePorNombre implements
Comparator<Estudiante> {
    @Override
    public int compare(Estudiante e1, Estudiante e2) {
        return e1.getNombre().compareTo(e2.getNombre());
    }
}
// Collections.sort(listaDeEstudiantes, new
ComparadorEstudiantePorNombre());
// Con Java 8:
//
listaDeEstudiantes.sort(Comparator.comparing(Estudiante::getNombre
));
```

Resumen:

- Use Comparable para el orden natural y único de una clase que usted controla.
- Use Comparator para definir órdenes alternativos, para ordenar clases que no controla, o para una sintaxis más limpia con lambdas.

29. ¿Qué colecciones proporcionan acceso aleatorio a sus elementos?

El acceso aleatorio (random access) se refiere a la capacidad de acceder a cualquier elemento de una colección en tiempo constante (O(1)) o casi constante, independientemente de su posición. Esta capacidad está formalizada en Java por la interfaz marcadora java.util.RandomAccess.
Una interfaz marcadora no tiene métodos; su propósito es simplemente "marcar" una clase para indicar que posee una cierta propiedad. En este caso, RandomAccess indica que la implementación de List tiene un acceso aleatorio rápido.
Las clases de colección que implementan RandomAccess y, por lo tanto, proporcionan un acceso aleatorio eficiente son aquellas que se basan en arrays. Las principales son:

1. **ArrayList:** Es la implementación canónica de una lista de acceso aleatorio. Dado que utiliza un array internamente, puede calcular la dirección de memoria de cualquier elemento i directamente, lo que resulta en un rendimiento de O(1) para el método get(i).
2. **Vector:** Al igual que ArrayList, Vector también se basa en un array dinámico y, por lo tanto, implementa RandomAccess y ofrece un acceso de O(1).
3. **Stack:** Como Stack extiende Vector, también hereda esta propiedad.

Por el contrario, las colecciones que se basan en estructuras de datos enlazadas, como LinkedList, **no** implementan RandomAccess. Para acceder a un elemento en una LinkedList, se debe recorrer la lista desde el principio o el final, lo que resulta en un rendimiento de O(n).
El framework de Collections utiliza esta interfaz marcadora para optimizar algoritmos. Por ejemplo, un bucle for tradicional con un índice es más rápido para una lista que implementa RandomAccess, mientras que un Iterator es más rápido para una que no lo hace.

30. ¿Qué es una BlockingQueue y para qué se utiliza principalmente?

Una BlockingQueue es una interfaz del paquete java.util.concurrent que representa una cola segura para hilos con una característica adicional: puede **bloquear** las operaciones de inserción y extracción bajo ciertas condiciones.
Comportamiento de Bloqueo:

1. **Bloqueo en la Inserción:** Si un hilo intenta añadir un elemento a una BlockingQueue que está llena (ha alcanzado su capacidad máxima), el hilo se bloqueará y esperará hasta que otro hilo extraiga un elemento, liberando espacio. El método put(element) exhibe este comportamiento.
2. **Bloqueo en la Extracción:** Si un hilo intenta extraer un elemento de una BlockingQueue que está vacía, el hilo se bloqueará y esperará hasta que otro hilo añada un elemento. El método take() exhibe este comportamiento.

Características Principales:

- **Seguridad para Hilos:** Todas las implementaciones de BlockingQueue son seguras para hilos. Todas las operaciones de encolado y desencolado son atómicas y utilizan bloqueos internos para gestionar la concurrencia.

- **No Acepta Nulos:** No se pueden insertar elementos null. Intentarlo lanzará una NullPointerException.
- **Capacidad:** Pueden ser de capacidad limitada (acotada) o ilimitada (no acotada). Una cola no acotada nunca se bloqueará en la inserción.

Uso Principal: El Patrón Productor-Consumidor El caso de uso más común y canónico para BlockingQueue es la implementación del **patrón productor-consumidor**. En este patrón:

- Uno o más hilos **productores** crean elementos (datos, tareas, etc.) y los añaden a la cola.
- Uno o más hilos **consumidores** extraen elementos de la cola y los procesan.

La BlockingQueue actúa como el búfer compartido y sincronizado entre los productores y los consumidores. Simplifica enormemente la implementación de este patrón porque maneja automáticamente toda la sincronización y la comunicación entre hilos:

- Los productores no necesitan preocuparse por si la cola está llena; el método put() se bloqueará si es necesario.
- Los consumidores no necesitan preocuparse por si la cola está vacía; el método take() se bloqueará si es necesario.

Esto elimina la necesidad de escribir código manual de wait(), notify() y synchronized, que es complejo y propenso a errores.

Implementaciones Comunes:

- ArrayBlockingQueue: Una cola de bloqueo acotada respaldada por un array.
- LinkedBlockingQueue: Una cola de bloqueo opcionalmente acotada respaldada por nodos enlazados.
- PriorityBlockingQueue: Una cola de bloqueo no acotada que ordena los elementos según su orden natural o un Comparator.
- SynchronousQueue: Una cola de bloqueo con capacidad cero. Cada operación de inserción debe esperar a una operación de extracción correspondiente, y viceversa.

31. ¿Por qué la interfaz Map no extiende la interfaz Collection?

Esta es una pregunta de diseño de API que evalúa la comprensión de las abstracciones fundamentales del framework. Aunque los Map se agrupan conceptualmente con las colecciones, su estructura fundamental es diferente.

Las razones principales son:

1. **Diferencia de Abstracción Fundamental:**
    - La interfaz Collection representa un **grupo de elementos individuales**. Su método add(E element) toma un solo elemento.
    - La interfaz Map representa un conjunto de **pares clave-valor**. Su método put(K key, V value) requiere dos objetos: una clave y un valor.
    - Forzar a Map a extender Collection crearía una incompatibilidad semántica. El método add de Collection no tendría sentido en el contexto de un Map. ¿Qué se añadiría? ¿Una clave? ¿Un valor? ¿Un par? La única opción lógica sería un par (como Map.Entry), pero esto haría que la interfaz Collection fuera menos general y más engorrosa para List y Set.
2. **Contratos Incompatibles:**
    - El contrato de Collection se centra en la pertenencia de elementos individuales (contains(Object o)).
    - El contrato de Map se centra en la búsqueda por clave (containsKey(Object key), containsValue(Object value), get(Object key)). Estos métodos no tienen un

equivalente directo en la interfaz Collection.

3. **Vistas de Colección como Solución:** En lugar de una relación de herencia, el framework proporciona una solución más elegante: los Map pueden ser **vistos como colecciones** a través de tres métodos:
    - **keySet():** Devuelve un Set de todas las claves en el mapa.
    - **values():** Devuelve una Collection de todos los valores en el mapa.
    - **entrySet():** Devuelve un Set de los pares clave-valor (como objetos Map.Entry).

Estas vistas proporcionan un puente entre el mundo de los Map y el de las Collection, permitiendo que los algoritmos y las operaciones de Collection (como la iteración) se apliquen a los mapas de una manera clara y bien definida, sin corromper las abstracciones de ninguna de las dos interfaces.

32. ¿Qué sucede si se añade un objeto a una colección mientras se itera sobre ella?

La respuesta a esta pregunta depende del tipo de iterador que se esté utilizando: fail-fast o fail-safe.

Usando un iterador Fail-Fast (el caso común):

- **Comportamiento:** Si se modifica estructuralmente una colección (añadiendo, eliminando elementos) directamente a través de la colección en lugar de a través del método remove() del propio iterador, el iterador lanzará una java.util.ConcurrentModificationException.
- **Ejemplo:**

```
List<String> lista = new ArrayList<>(Arrays.asList("a", "b",
"c"));
for (String s : lista) { // El for-each usa un iterador
internamente
    if (s.equals("b")) {
        lista.add("d"); // Esto lanzará
ConcurrentModificationException
    }
}
```

- **Razón:** Las colecciones no concurrentes como ArrayList y HashMap no están diseñadas para ser modificadas por un hilo mientras otro hilo las está iterando. El iterador fail-fast está diseñado para detectar esta situación y fallar inmediatamente para evitar un comportamiento no determinista o la corrupción de datos.

Excepción: Usando el método remove() del iterador: El único método seguro para modificar una colección durante la iteración con un iterador fail-fast es llamar al método remove() del propio iterador.

```
List<String> lista = new ArrayList<>(Arrays.asList("a", "b", "c"));
Iterator<String> it = lista.iterator();
while (it.hasNext()) {
    String s = it.next();
    if (s.equals("b")) {
        it.remove(); // Esto es seguro y funciona
    }
```

```
}
// La lista ahora es ["a", "c"]
```

Nota: El iterador no tiene un método add(). La interfaz ListIterator sí tiene un método add(), que es seguro de usar.
Usando un iterador Fail-Safe:

- **Comportamiento:** Si se utiliza una colección del paquete java.util.concurrent, como CopyOnWriteArrayList o ConcurrentHashMap, sus iteradores son fail-safe. No lanzarán ConcurrentModificationException.
- **Mecanismo:** Estos iteradores operan sobre una instantánea o copia de la colección en el momento en que se creó el iterador.
- **Consecuencia:** La iteración continuará sin errores, pero el iterador **no reflejará** las modificaciones realizadas en la colección después de su creación. Es decir, si se añade un elemento a un CopyOnWriteArrayList mientras se itera sobre él, el iterador no "verá" ese nuevo elemento.

33. ¿Cómo se puede crear una colección sincronizada a partir de una colección dada?

Java proporciona una forma sencilla de crear un *wrapper* (envoltorio) seguro para hilos alrededor de una colección no sincronizada existente utilizando la clase de utilidad java.util.Collections.
Existen métodos de fábrica estáticos para los principales tipos de colecciones:

- **Para una List:** Collections.synchronizedList(List<T> list)
  ```
  List<String> listaNoSincronizada = new ArrayList<>();
  List<String> listaSincronizada =
  Collections.synchronizedList(listaNoSincronizada);
  ```

- **Para un Set:** Collections.synchronizedSet(Set<T> set)
  ```
  Set<String> setNoSincronizado = new HashSet<>();
  Set<String> setSincronizado =
  Collections.synchronizedSet(setNoSincronizado);
  ```

- **Para un Map:** Collections.synchronizedMap(Map<K, V> map)
  ```
  Map<String, Integer> mapNoSincronizado = new HashMap<>();
  Map<String, Integer> mapSincronizado =
  Collections.synchronizedMap(mapNoSincronizado);
  ```

Cómo funcionan: Estos métodos devuelven una nueva instancia de una clase interna privada (como SynchronizedList) que envuelve la colección original. Cada método público de este *wrapper* está declarado como synchronized, lo que significa que cada llamada a un método (como add(), get(), remove()) adquiere un bloqueo en el objeto *wrapper*, garantizando que solo un hilo pueda ejecutar un método a la vez.
Advertencia Importante sobre la Iteración: Aunque los métodos individuales son seguros para hilos, las operaciones compuestas, como la iteración, **no lo son**. Si un hilo está iterando sobre la colección sincronizada, otro hilo podría modificarla entre las llamadas a hasNext() y next(), lo que podría causar una ConcurrentModificationException.

Para iterar de forma segura sobre una colección sincronizada, se debe sincronizar manualmente en el bloque del iterador:

```
List<String> listaSincronizada = Collections.synchronizedList(new
ArrayList<>());
//... añadir elementos desde múltiples hilos

// Iteración segura
synchronized (listaSincronizada) {
    Iterator<String> it = listaSincronizada.iterator();
    while (it.hasNext()) {
        String s = it.next();
        //... procesar s
    }
}
```

Alternativa Moderna: En el desarrollo moderno, para la concurrencia se prefieren las clases del paquete java.util.concurrent (como ConcurrentHashMap, CopyOnWriteArrayList) sobre estos *wrappers* sincronizados. Ofrecen una concurrencia y un rendimiento superiores debido a sus algoritmos de bloqueo más sofisticados.

34. ¿Cuál es el propósito de la interfaz RandomAccess?

RandomAccess es una **interfaz marcadora** (*marker interface*) en el Java Collections Framework. No declara ningún método. Su único propósito es "marcar" o indicar que una implementación de List soporta un **acceso aleatorio rápido y eficiente**.
¿Qué significa "acceso aleatorio rápido"? Significa que el tiempo para acceder a un elemento a través de su índice (list.get(index)) es de tiempo constante, **O(1)**. Esto es característico de las listas basadas en arrays, como ArrayList, donde la posición de memoria de cualquier elemento se puede calcular directamente.
Por el contrario, las listas que no implementan RandomAccess, como LinkedList, tienen un acceso secuencial. Para obtener el elemento en el índice n, deben recorrer n elementos desde el principio, lo que resulta en un rendimiento de O(n).
¿Para qué se utiliza? El principal uso de esta interfaz es permitir que los algoritmos genéricos cambien su comportamiento para proporcionar un buen rendimiento. Por ejemplo, el método Collections.sort() puede comprobar si una lista es una instancia de RandomAccess.

- Si lo es, puede usar un bucle for basado en índices para acceder y mover elementos, lo cual es muy eficiente para ArrayList.
- Si no lo es (como en el caso de LinkedList), usará un Iterator para recorrer la lista, lo cual es mucho más eficiente para LinkedList que hacer llamadas repetidas a get(index).

Clases que la implementan:

- ArrayList
- Vector
- Stack

En resumen, RandomAccess es una pista para los algoritmos que les dice: "Es seguro y eficiente usar acceso basado en índices con esta lista".

35. ¿Se puede añadir un elemento null a un TreeSet o HashSet?

La respuesta difiere entre TreeSet y HashSet debido a sus diferentes mecanismos internos.

- **HashSet:**
  - **Sí, permite un único elemento null**.
  - HashSet utiliza un HashMap internamente. Cuando se añade un elemento, se utiliza como clave en el HashMap. HashMap permite una clave null. Por lo tanto, HashSet también puede almacenar un valor null. Si se intenta añadir null más de una vez, las inserciones posteriores se ignorarán, ya que los Set no permiten duplicados.
- **TreeSet:**
  - **No, no permite elementos null**.
  - TreeSet mantiene sus elementos en un orden de clasificación. Para ello, debe comparar los elementos entre sí. Esta comparación se realiza utilizando el método compareTo() (de la interfaz Comparable) o compare() (de un Comparator).
  - Si se intenta añadir un elemento null a un TreeSet, se lanzará una NullPointerException en tiempo de ejecución. Esto ocurre porque al intentar comparar el null con otros elementos existentes para determinar su posición en el árbol, la llamada a compareTo() o compare() sobre un objeto null falla.

En resumen:

- HashSet: Sí, un null.
- TreeSet: No, nunca.

36. ¿Cuál es el tamaño por defecto y el factor de carga de un HashMap? ¿Qué ocurre cuando se supera el factor de carga?

Esta pregunta evalúa el conocimiento sobre los parámetros de ajuste de rendimiento de HashMap.

- **Tamaño por Defecto (Capacidad Inicial):** La capacidad inicial por defecto de un HashMap, si no se especifica en el constructor, es **16**. Esta es la cantidad de "cubetas" o *buckets* que el HashMap crea inicialmente en su array interno.
- **Factor de Carga por Defecto (Load Factor):** El factor de carga por defecto es **0.75**. El factor de carga es una medida de cuán lleno puede estar el HashMap antes de que su capacidad se incremente automáticamente.

¿Qué ocurre cuando se supera el factor de carga? Cuando el número de entradas en el HashMap excede el producto de la capacidad actual y el factor de carga, el HashMap se **redimensiona** (*rehashes*).
El umbral para la redimensión se calcula como: umbral = capacidad * factor_de_carga. Por defecto, esto es 16 * 0.75 = 12.
Cuando se añade el 13º elemento, ocurre lo siguiente:

1. **Creación de un Nuevo Array:** Se crea un nuevo array de cubetas, típicamente con el **doble de la capacidad** del anterior (de 16 a 32, de 32 a 64, y así sucesivamente).
2. **Rehashing:** Se recorren **todas** las entradas existentes en el HashMap antiguo. Para cada entrada, se recalcula su índice en el nuevo array (que ahora es más grande) y se coloca en la nueva cubeta correspondiente.
3. **Impacto en el Rendimiento:** El proceso de *rehashing* es una operación costosa (O(n), donde n es el número de entradas), ya que implica la creación de una nueva estructura de datos y la reinserción de todos los elementos.

Importancia de la Configuración: Elegir una capacidad inicial adecuada es importante para el

rendimiento.

- Si se sabe de antemano que se almacenarán muchos elementos, es una buena práctica crear el HashMap con una capacidad inicial suficientemente grande para evitar múltiples operaciones de *rehashing*.

```
// Si se esperan 1000 elementos, se puede inicializar así para evitar rehashing.
// Capacidad = 1000 / 0.75 = 1334. Se elige la siguiente potencia de 2, que es 2048.
Map<String, String> mapa = new HashMap<>(2048);
```

37. ¿Cuál es la diferencia entre Array y ArrayList? ¿Cuándo usaría uno sobre el otro?

Array y ArrayList son dos de las formas más comunes de almacenar colecciones de elementos en Java, pero tienen diferencias fundamentales en cuanto a tamaño, funcionalidad y tipo de datos.

Característica	Array	ArrayList
Tipo	Es una estructura de datos fundamental del lenguaje.	Es una clase que forma parte del Java Collections Framework (java.util.ArrayList).
Tamaño	**Tamaño fijo.** La longitud de un array se establece en el momento de su creación y no se puede cambiar.	**Tamaño dinámico.** Puede crecer y encogerse automáticamente según sea necesario.
Tipo de Datos	Puede contener tanto **tipos primitivos** (int, char, etc.) como **objetos**.	Solo puede contener **objetos**. Para los tipos primitivos, utiliza las clases wrapper correspondientes (p. ej., Integer para int) a través del autoboxing.
Rendimiento	Generalmente más rápido, especialmente para tipos primitivos, debido a la ausencia de la sobrecarga de los objetos wrapper y los métodos de la colección.	Ligeramente más lento debido a la sobrecarga de la clase ArrayList y el boxing/unboxing de primitivos.
Funcionalidad	Proporciona una funcionalidad básica (acceso por índice, propiedad length). No tiene métodos incorporados para manipulación (como añadir, eliminar, buscar).	Ofrece una rica API con muchos métodos de utilidad para añadir, eliminar, buscar, ordenar y manipular elementos (add, remove, contains, sort, etc.).
Declaración	int miArray = new int;	ArrayList<Integer> miLista = new ArrayList<>();

¿Cuándo usar Array?

- Cuando el **tamaño de la colección es fijo y conocido** de antemano.

- Cuando se necesita el **máximo rendimiento** y se trabaja con tipos primitivos.
- En programación de bajo nivel o cuando se interactúa con APIs que requieren arrays (p. ej., main(String args)).
- Para arrays multidimensionales, donde la sintaxis de los arrays es más natural.

¿Cuándo usar ArrayList?

- Cuando el **tamaño de la colección es desconocido o variable**.
- Cuando se necesita la **flexibilidad** de los métodos del Collections Framework (añadir, eliminar, buscar, etc.).
- En la mayoría de los casos de desarrollo de aplicaciones de alto nivel, donde la legibilidad y la funcionalidad son más importantes que las micro-optimizaciones de rendimiento.

En general, para la mayoría de las aplicaciones de negocio, ArrayList es la opción preferida debido a su flexibilidad y facilidad de uso.

38. ¿Cómo se puede hacer que un ArrayList sea de solo lectura?

Se puede hacer que un ArrayList (o cualquier List) sea de solo lectura utilizando el método Collections.unmodifiableList() de la clase de utilidad java.util.Collections.

Cómo funciona:

1. Se crea y se puebla un ArrayList normal y modificable.
2. Se pasa esta lista al método Collections.unmodifiableList().
3. Este método devuelve una **vista de solo lectura** de la lista original. No crea una copia de los datos, sino un *wrapper* (envoltorio) que no permite operaciones de modificación.

Comportamiento:

- La lista devuelta es inmutable en el sentido de que no se pueden añadir, eliminar o modificar sus elementos a través de sus propios métodos.
- Cualquier intento de llamar a un método de modificación en la vista de solo lectura (como add(), remove(), set(), clear()) lanzará una java.lang.UnsupportedOperationException.

Ejemplo:

```
import java.util.*;

public class ListaSoloLectura {
    public static void main(String args) {
        List<String> listaModificable = new ArrayList<>();
        listaModificable.add("Java");
        listaModificable.add("Python");

        // Crear una vista de solo lectura
        List<String> listaSoloLectura =
Collections.unmodifiableList(listaModificable);

        // Intentar modificar la vista de solo lectura
        try {
            listaSoloLectura.add("C++"); // Esto lanzará
UnsupportedOperationException
        } catch (UnsupportedOperationException e) {
            System.out.println("Error: No se puede modificar una lista
de solo lectura.");
```

```
            e.printStackTrace();
        }

        // Leer de la lista de solo lectura es posible
        System.out.println("Contenido de la lista de solo lectura: " +
listaSoloLectura);

        // ¡Cuidado! Los cambios en la lista original se reflejan en
la vista
        listaModificable.add("Go");
        System.out.println("Contenido de la lista de solo lectura
después de modificar la original: " + listaSoloLectura);
    }
}
```

Punto Clave: Es crucial entender que la vista devuelta no es una copia independiente. Si la lista original subyacente se modifica, esos cambios serán visibles en la vista de solo lectura. Por lo tanto, para una verdadera inmutabilidad, se debe asegurar que no existan más referencias a la lista original modificable.

39. ¿Qué es una PriorityQueue y en qué se diferencia de una cola normal?

Una PriorityQueue es una implementación especializada de la interfaz Queue que ordena los elementos según su **prioridad**, en lugar del orden de inserción FIFO (First-In, First-Out) de una cola normal.

Diferencias Clave con una Cola Normal (como LinkedList):

- **Orden de los Elementos:**
  - **Cola Normal (LinkedList):** Sigue estrictamente el orden FIFO. El primer elemento añadido es el primer elemento en ser eliminado.
  - **PriorityQueue:** No sigue el orden FIFO. El elemento que se extrae (poll(), peek()) es siempre el que tiene la **mayor prioridad**. Por defecto, la mayor prioridad corresponde al elemento "menor" según su orden natural (el valor numérico más bajo, el primer carácter alfabético, etc.).
- **Ordenación:**
  - La prioridad se determina de dos maneras:
    1. **Orden Natural:** Si los objetos implementan la interfaz Comparable, se utiliza su orden natural.
    2. **Comparator:** Se puede proporcionar un Comparator personalizado al constructor de la PriorityQueue para definir un orden de prioridad específico.
- **Implementación Interna:**
  - Se basa en una estructura de datos llamada **montículo binario** (*binary heap*). Esta estructura garantiza que la operación de añadir (add/offer) y la de eliminar el elemento de mayor prioridad (poll/remove) tengan un rendimiento de **O(\log n)**. La operación de inspeccionar el elemento de mayor prioridad (peek/element) es de **O(1)**.
- **Iteración:** La iteración sobre una PriorityQueue **no garantiza** que los elementos se

devuelvan en orden de prioridad. La única garantía es que peek() y poll() devolverán el elemento de mayor prioridad.

Caso de Uso: Se utiliza en algoritmos donde se necesita procesar repetidamente el elemento de mayor prioridad de una colección. Ejemplos comunes incluyen:

- **Algoritmo de Dijkstra** para encontrar el camino más corto en un grafo.
- **Algoritmo de Prim** para encontrar el árbol de expansión mínima.
- En sistemas operativos, para la planificación de tareas, donde las tareas de mayor prioridad se ejecutan primero.
- Cualquier escenario que requiera un sistema de "lo más importante primero".

40. ¿Qué son las vistas de colección (Collection Views) proporcionadas por la interfaz Map?

Dado que la interfaz Map no extiende Collection, proporciona tres métodos que devuelven **vistas de colección** de sus contenidos. Estas vistas son la forma principal de iterar sobre un Map y permiten que los Map sean tratados como colecciones en ciertos contextos.

Una "vista" no es una copia de los datos. Es una representación respaldada por el Map original, lo que significa que los cambios en el Map se reflejan en la vista, y (en la mayoría de los casos) los cambios en la vista (como eliminar un elemento) se reflejan en el Map.

Las tres vistas de colección son:

1. **keySet():**
    - **Devuelve:** Un Set<K> que contiene todas las **claves** del mapa.
    - **Comportamiento:** Como es un Set, no contiene claves duplicadas (lo cual es inherente a un Map).
    - **Modificación:** Se puede eliminar una entrada del mapa original llamando a remove() en el Set de claves. Las operaciones de adición (add(), addAll()) no están soportadas.
2. **values():**
    - **Devuelve:** Una Collection<V> que contiene todos los **valores** del mapa.
    - **Comportamiento:** Esta colección puede contener duplicados si múltiples claves mapean al mismo valor.
    - **Modificación:** Al igual que con keySet(), se puede eliminar una entrada del mapa original eliminando su valor de esta colección (esto eliminará la primera entrada que coincida con ese valor). Las operaciones de adición no están soportadas.
3. **entrySet():**
    - **Devuelve:** Un Set<Map.Entry<K, V>> que contiene los **pares clave-valor** del mapa. Cada elemento del Set es un objeto Map.Entry.
    - **Comportamiento:** Esta es la vista más potente y, a menudo, la más eficiente para iterar, ya que proporciona acceso tanto a la clave como al valor en cada paso de la iteración, sin necesidad de hacer una búsqueda adicional (get(key)).
    - **Modificación:** Se pueden eliminar entradas del mapa a través del iterador de este Set. Además, se puede cambiar el valor asociado a una clave llamando al método setValue() en un objeto Map.Entry.

Ejemplo de Iteración (forma preferida):

```
Map<String, Integer> mapa = new HashMap<>();
mapa.put("Uno", 1);
mapa.put("Dos", 2);
```

```
// Iterar usando entrySet es más eficiente que iterar sobre keySet y
hacer get()
for (Map.Entry<String, Integer> entrada : mapa.entrySet()) {
    String clave = entrada.getKey();
    Integer valor = entrada.getValue();
    System.out.println("Clave: " + clave + ", Valor: " + valor);
}
```

Sección III: Concurrencia y Multihilo Avanzados

Esta sección pone a prueba uno de los aspectos más desafiantes de Java. Evalúa la capacidad del candidato para escribir código concurrente seguro y eficiente.

41. ¿Cuál es la diferencia entre un Proceso y un Hilo (Thread)?

Esta es una pregunta fundamental sobre los conceptos básicos de la concurrencia en los sistemas operativos, que es esencial para entender el multihilo en Java.

- **Proceso:**
  - **Definición:** Un proceso es una **instancia de un programa en ejecución**. Es un entorno de ejecución autónomo y aislado.
  - **Memoria:** Cada proceso tiene su **propio espacio de memoria** en el heap, que no se comparte con otros procesos. Esto proporciona aislamiento y seguridad: un fallo en un proceso generalmente no afecta a otros.
  - **Recursos:** Los procesos son "pesados" (*heavyweight*). Crear un proceso es costoso en términos de tiempo y recursos del sistema operativo, ya que requiere la asignación de un nuevo espacio de memoria.
  - **Comunicación:** La comunicación entre procesos (IPC - Inter-Process Communication) es compleja y lenta, ya que requiere mecanismos del sistema operativo como tuberías (*pipes*), sockets o memoria compartida.
  - **Ejemplo:** Ejecutar dos aplicaciones diferentes, como un navegador web y un editor de texto, crea dos procesos separados.
- **Hilo (Thread):**
  - **Definición:** Un hilo es la **unidad de ejecución más pequeña** dentro de un proceso. Un proceso puede tener uno o más hilos.
  - **Memoria:** Los hilos dentro del mismo proceso **comparten el mismo espacio de memoria del heap**. Sin embargo, cada hilo tiene su **propia pila de llamadas** (*call stack*) y su propio contador de programa (PC Register). La pila se utiliza para las variables locales y las llamadas a métodos de ese hilo.
  - **Recursos:** Los hilos son "ligeros" (*lightweight*). Crearlos es mucho más rápido y consume menos recursos que crear un proceso.
  - **Comunicación:** La comunicación entre hilos es mucho más simple y rápida, ya que pueden acceder directamente a los datos compartidos en el heap. Sin embargo, este acceso compartido es la principal fuente de problemas de concurrencia (como condiciones de carrera y deadlocks) y requiere sincronización.
  - **Ejemplo:** En un procesador de textos, un hilo puede estar manejando la entrada

del usuario, otro puede estar revisando la ortografía en segundo plano, y un tercero puede estar guardando el documento automáticamente.

Resumen de la Diferencia: Un proceso es como una casa, con sus propias paredes, recursos y dirección. Los hilos son como las personas que viven en esa casa: comparten los recursos comunes (la cocina, el salón), pero cada uno tiene su propia habitación privada (la pila).

42. ¿Cuáles son las dos formas de crear un hilo en Java? ¿Cuál es la preferida y por qué?

Existen dos formas principales de crear un hilo en Java :

1. **Extendiendo la clase java.lang.Thread:**
    - Se crea una nueva clase que hereda de Thread.
    - Se sobrescribe el método run(), que contiene la lógica que ejecutará el hilo.
    - Se crea una instancia de esta nueva clase y se llama a su método start() para iniciar la ejecución del hilo.

```
class MiHilo extends Thread {
    public void run() {
        System.out.println("Hilo extendiendo Thread en
ejecución.");
    }
}
// Para usarlo: new MiHilo().start();
```

2. **Implementando la interfaz java.lang.Runnable:**
    - Se crea una clase que implementa la interfaz Runnable.
    - Se implementa el único método de la interfaz, run().
    - Se crea una instancia de esta clase, se pasa como argumento al constructor de la clase Thread, y luego se llama al método start() del objeto Thread.

```
class MiRunnable implements Runnable {
    public void run() {
        System.out.println("Hilo implementando Runnable en
ejecución.");
    }
}
// Para usarlo: new Thread(new MiRunnable()).start();
```

¿Cuál es la preferida y por qué?

La forma **preferida es implementar la interfaz Runnable**. Las razones se basan en buenos principios de diseño de software:

1. **Favorece la Composición sobre la Herencia:** Implementar Runnable separa la tarea a realizar (la lógica en run()) de la maquinaria de ejecución del hilo (Thread). La clase que contiene la tarea no necesita ser un Thread; simplemente es una tarea que puede ser ejecutada por un Thread. Esto es más flexible y sigue el principio de "composición sobre herencia".
2. **Java no soporta Herencia Múltiple:** Si su clase ya extiende otra clase, no puede extender también la clase Thread. Sin embargo, puede implementar cualquier número de interfaces. Implementar Runnable no le impide extender otra clase.

3. **Mejor Reutilización y Diseño:** La misma instancia de Runnable puede ser pasada a múltiples objetos Thread si es necesario, o puede ser enviada a un ExecutorService (un pool de hilos), lo que la hace más reutilizable y compatible con el framework de concurrencia moderno de Java.

En resumen, aunque extender Thread es más simple para casos triviales, implementar Runnable conduce a un diseño más limpio, flexible y robusto.

43. Describa el ciclo de vida de un hilo en Java.

El ciclo de vida de un hilo describe los diferentes estados por los que puede pasar un hilo desde su creación hasta su terminación. Comprender estos estados es crucial para la depuración y el análisis de aplicaciones multihilo.
Los estados de un hilo en Java, definidos en la enumeración Thread.State, son:

1. **NEW (Nuevo):**
    - Un hilo está en este estado después de que se ha creado una instancia de la clase Thread (p. ej., Thread t = new Thread();), pero antes de que se llame al método start().
    - En este punto, el hilo no está vivo y no ha sido asignado a ningún recurso del sistema.
2. **RUNNABLE (Ejecutable):**
    - Un hilo entra en este estado después de que se llama a su método start().
    - Un hilo en estado RUNNABLE está listo para ser ejecutado y está esperando que el planificador de hilos (*thread scheduler*) del sistema operativo le asigne tiempo de CPU.
    - Este estado agrupa tanto "listo para ejecutar" como "en ejecución". Desde la perspectiva de la JVM, un hilo que está actualmente en la CPU se considera RUNNABLE.
3. **BLOCKED (Bloqueado):**
    - Un hilo entra en este estado cuando intenta adquirir un bloqueo de monitor (p. ej., al entrar en un bloque o método synchronized) que actualmente está en posesión de otro hilo.
    - El hilo permanecerá en estado BLOCKED hasta que el bloqueo sea liberado por el otro hilo.
4. **WAITING (En Espera):**
    - Un hilo entra en este estado cuando está esperando indefinidamente a que otro hilo realice una acción particular.
    - Esto ocurre cuando se llama a uno de los siguientes métodos sin un tiempo de espera:
        - Object.wait()
        - Thread.join()
        - LockSupport.park()
    - Un hilo en WAITING solo puede volver a RUNNABLE si es "despertado" explícitamente por otro hilo (p. ej., con Object.notify() o Object.notifyAll()).
5. **TIMED_WAITING (En Espera con Tiempo):**
    - Este estado es similar a WAITING, pero el hilo solo esperará durante un intervalo de tiempo especificado.
    - Entra en este estado al llamar a métodos con un tiempo de espera:
        - Thread.sleep(long millis)

        - Object.wait(long timeout)
        - Thread.join(long millis)
        - LockSupport.parkNanos(long nanos)
        - LockSupport.parkUntil(long deadline)
    - El hilo volverá a RUNNABLE si el tiempo de espera expira o si es notificado por otro hilo.
6. **TERMINATED (Terminado):**
    - Un hilo entra en este estado cuando ha completado la ejecución de su método run() o cuando termina debido a una excepción no capturada.
    - Una vez que un hilo está en estado TERMINATED, no puede ser reiniciado.

44. ¿Cuál es la diferencia entre llamar al método start() y al método run() de un hilo?

Esta es una pregunta crítica que revela si un candidato entiende la esencia de cómo se inicia un nuevo hilo de ejecución en Java.

- **start():**
    - **Acción:** El método start() es el que **crea un nuevo hilo de ejecución**.
    - **Proceso:** Cuando se llama a t.start(), ocurren varias cosas:
        1. La JVM asigna los recursos necesarios para un nuevo hilo.
        2. Registra el nuevo hilo con el planificador de hilos del sistema operativo.
        3. El nuevo hilo entra en el estado RUNNABLE.
        4. Cuando el planificador de hilos le da tiempo de CPU, el nuevo hilo comienza a ejecutar la lógica definida en su método run().
    - **Resultado:** La llamada a start() es asíncrona y regresa inmediatamente. El hilo llamador (p. ej., el hilo main) continúa su ejecución en paralelo con el nuevo hilo.
    - **Restricción:** Solo se puede llamar a start() una vez en un objeto Thread. Llamarlo una segunda vez lanzará una IllegalThreadStateException.
- **run():**
    - **Acción:** El método run() simplemente contiene la **lógica o el código que el hilo ejecutará**.
    - **Proceso:** Si se llama directamente a t.run(), **no se crea un nuevo hilo**. En su lugar, el método run() se ejecuta como una llamada a un método normal, **dentro del hilo actual** (el hilo que hizo la llamada).
    - **Resultado:** La llamada a run() es síncrona. El hilo llamador se bloqueará hasta que el método run() termine su ejecución. No hay concurrencia ni paralelismo.
    - **Restricción:** Se puede llamar al método run() tantas veces como se desee, como cualquier otro método.

Analogía:

- Llamar a start() es como decirle a un corredor: "¡Prepárate, listo, ya! ¡Corre tu carrera!". El corredor (el nuevo hilo) empieza a correr por su cuenta, y tú (el hilo principal) puedes seguir haciendo otras cosas.
- Llamar a run() es como tomar el plan de entrenamiento del corredor y ejecutarlo tú mismo. No hay un segundo corredor; solo tú estás haciendo el trabajo.

En resumen: Para lograr el multihilo, siempre se debe llamar a start(). Llamar a run() directamente anula el propósito de usar hilos.

45. ¿Cuál es la diferencia entre un método synchronized y un bloque synchronized? ¿Cuál es preferible?

Ambos son mecanismos en Java para lograr la exclusión mutua y garantizar que solo un hilo a la vez pueda ejecutar una sección crítica de código. Sin embargo, difieren en su alcance y flexibilidad.

Método synchronized:

- **Sintaxis:** Se aplica la palabra clave synchronized a la declaración completa de un método.

```
public synchronized void miMetodo() {
    // Código crítico
}
```

- **Bloqueo:**
    - Para un **método de instancia**, adquiere un bloqueo en el objeto this (la instancia actual de la clase). Esto significa que ningún otro hilo puede ejecutar **ningún otro método synchronized** en el **mismo objeto** hasta que el primer hilo libere el bloqueo.
    - Para un **método estático**, adquiere un bloqueo en el objeto Class de la clase (p. ej., MiClase.class).
- **Alcance:** El bloqueo se mantiene durante toda la ejecución del método.

Bloque synchronized:

- **Sintaxis:** Se utiliza la palabra clave synchronized para proteger solo una parte del código dentro de un método. Requiere que se especifique un objeto en el que se adquirirá el bloqueo.

```
public void miMetodo() {
    // Código no crítico
    synchronized (this) { // o synchronized(otroObjeto)
        // Código crítico
    }
    // Más código no crítico
}
```

- **Bloqueo:** Adquiere un bloqueo en el objeto especificado entre paréntesis. Puede ser this, o cualquier otro objeto. Esto permite un control de bloqueo mucho más granular.
- **Alcance:** El bloqueo solo se mantiene mientras se ejecuta el código dentro del bloque.

¿Cuál es preferible?

En general, **los bloques synchronized son preferibles** a los métodos synchronized. Las razones son:

1. **Principio de Mínimo Privilegio (o Mínimo Bloqueo):** Solo se debe bloquear la sección de código que es absolutamente crítica y que accede a los recursos compartidos. Los métodos a menudo contienen código no crítico que no necesita ser sincronizado. Sincronizar todo el método puede bloquear a otros hilos innecesariamente, reduciendo la concurrencia y el rendimiento.
2. **Flexibilidad en el Objeto de Bloqueo:** Los bloques synchronized permiten elegir en qué objeto se va a bloquear. Esto es muy útil para evitar el bloqueo de todo el objeto this. Se pueden usar objetos de bloqueo privados y dedicados (private final Object lock = new

Object();) para diferentes secciones críticas, lo que permite que diferentes hilos ejecuten diferentes secciones críticas en el mismo objeto simultáneamente.
3. **Evitar Deadlocks:** Al usar objetos de bloqueo más granulares, se puede diseñar un sistema que sea menos propenso a los deadlocks, ya que no se está bloqueando el objeto this de forma generalizada.

Un método synchronized es esencialmente un atajo sintáctico para un bloque synchronized(this) que abarca todo el cuerpo del método. Es conveniente para métodos pequeños y simples, pero para un control de concurrencia más robusto y de mayor rendimiento, los bloques synchronized son la mejor opción.

46. ¿Qué hace la palabra clave volatile y cuándo se debe usar?

La palabra clave volatile es un mecanismo de sincronización más ligero que synchronized. Su propósito principal es garantizar la **visibilidad** de las escrituras de una variable entre diferentes hilos.

El Problema de Visibilidad: En un entorno multihilo, para optimizar el rendimiento, cada hilo puede tener su propia copia en caché local (en los registros de la CPU o en la caché de la CPU) de las variables compartidas. Cuando un hilo modifica una variable, puede que solo la modifique en su caché local. Otros hilos, que leen desde la memoria principal o desde sus propias cachés, podrían no ver este cambio inmediatamente, lo que lleva a datos obsoletos e inconsistencias.

¿Qué hace volatile? Cuando una variable se declara como volatile:

1. **Garantiza la Visibilidad:** Cualquier escritura en una variable volatile se vacía inmediatamente a la memoria principal. Cualquier lectura de una variable volatile se realiza siempre directamente desde la memoria principal, no desde la caché de un hilo. Esto asegura que cualquier hilo que lea la variable vea siempre el valor más reciente escrito por cualquier otro hilo.
2. **Establece una Relación "Happens-Before":** Una escritura en una variable volatile *sucede antes* (*happens-before*) que cualquier lectura posterior de esa misma variable. Esto no solo garantiza la visibilidad del valor de la variable volatile en sí, sino también la de todas las demás variables que fueron modificadas por el mismo hilo *antes* de escribir en la variable volatile.
3. **Previene la Reordenación de Instrucciones:** El compilador y la CPU no pueden reordenar las lecturas y escrituras de la variable volatile con respecto a otras lecturas y escrituras de memoria.

volatile vs. synchronized:

- volatile solo garantiza la visibilidad y el orden. **No garantiza la atomicidad** de operaciones compuestas (como i++, que es una lectura, una modificación y una escritura).
- synchronized garantiza tanto la **visibilidad** como la **atomicidad** (exclusión mutua).

¿Cuándo usar volatile? Se debe usar volatile solo cuando se cumplen **todas** las siguientes condiciones:

1. Las escrituras en la variable no dependen de su valor actual (p. ej., no es un contador como i++). Asignar un nuevo valor (variable = nuevoValor) es un buen caso.
2. La variable no participa en invariantes con otras variables de estado.
3. El acceso a la variable no requiere bloqueo por ninguna otra razón.

Un caso de uso canónico es una **bandera de estado** simple que es escrita por un hilo y leída por otros para controlar la terminación de un bucle.

```
public class Worker implements Runnable {
    private volatile boolean running = true;

    public void stop() {
        running = false;
    }

    public void run() {
        while (running) {
            // Hacer trabajo...
        }
        System.out.println("El trabajador se ha detenido.");
    }
}
```

Aquí, un hilo puede llamar a stop() para cambiar el valor de running, y volatile garantiza que el hilo que ejecuta el bucle while verá este cambio y saldrá del bucle.

47. Explique wait(), notify() y notifyAll(). ¿Por qué deben ser llamados desde un bloque o método synchronized?

wait(), notify() y notifyAll() son métodos de la clase java.lang.Object que forman la base de la comunicación entre hilos en Java. Permiten que los hilos coordinen sus acciones basándose en el estado de un objeto compartido.

- **wait():**
  - Cuando un hilo llama a objeto.wait(), hace dos cosas:
    1. **Libera el bloqueo del monitor** que posee sobre objeto.
    2. Entra en el estado WAITING (o TIMED_WAITING si se usa wait(timeout)).
  - El hilo permanecerá en espera hasta que otro hilo llame a notify() o notifyAll() en el **mismo objeto**.
- **notify():**
  - Despierta a **un único hilo** que está esperando en el monitor de ese objeto. Si hay varios hilos esperando, la JVM elige uno arbitrariamente para despertar.
  - El hilo despertado no se ejecuta inmediatamente. Entra en el estado BLOCKED y debe competir para volver a adquirir el bloqueo del monitor una vez que el hilo que llamó a notify() lo libere.
- **notifyAll():**
  - Despierta a **todos los hilos** que están esperando en el monitor de ese objeto.
  - Todos los hilos despertados entran en el estado BLOCKED y compiten por el bloqueo. Solo uno lo conseguirá, y los demás seguirán bloqueados.

¿Por qué deben ser llamados desde un contexto synchronized? Esta es la parte crucial de la pregunta. La razón está ligada al concepto de **monitor de objeto** y a la prevención de condiciones de carrera.

1. **Requisito de Propiedad del Monitor:** Para llamar a wait(), notify() o notifyAll() en un objeto, un hilo **debe ser el propietario del monitor de ese objeto**. La única forma de que un hilo se convierta en propietario del monitor de un objeto en Java es entrando en un bloque o método synchronized protegido por ese objeto. Si se intenta llamar a estos

métodos fuera de un contexto sincronizado, se lanzará una java.lang.IllegalMonitorStateException.

2. **Prevención de "Señales Perdidas" (Lost Wake-up Problem):** La sincronización es necesaria para evitar una condición de carrera conocida como el problema de la "señal perdida". Consideremos este escenario sin sincronización:
    - Hilo A (Consumidor) comprueba si una condición es false.
    - La condición es false, por lo que Hilo A decide que va a esperar.
    - **¡Aquí ocurre la condición de carrera!** Justo antes de que Hilo A llame a wait(), el planificador de hilos lo suspende.
    - Hilo B (Productor) se ejecuta, cambia la condición a true y llama a notify(). Como Hilo A aún no está esperando, la notificación se pierde.
    - Hilo A se reanuda y llama a wait(), entrando en un estado de espera para siempre, porque la notificación que lo habría despertado ya ocurrió y se perdió.

El bloque synchronized resuelve esto. El Hilo A adquiere el bloqueo, comprueba la condición y, si es false, llama a wait(). La llamada a wait() libera el bloqueo y pone al hilo a dormir **atómicaente**. El Hilo B no puede adquirir el bloqueo para cambiar la condición y llamar a notify() hasta que el Hilo A haya liberado el bloqueo (ya sea saliendo del bloque o llamando a wait()). Esto garantiza que la notificación no se pueda perder.
Por esta razón, el uso canónico de wait() es siempre dentro de un bucle while que comprueba la condición, y todo el bloque está sincronizado.

```
synchronized (lock) {
    while (!condicion) {
        lock.wait();
    }
    // Proceder, la condición ahora es verdadera
}
```

48. ¿Qué es un deadlock (interbloqueo)? ¿Cómo se puede prevenir y detectar?

Un **deadlock** es una situación en la que dos o más hilos se bloquean permanentemente, cada uno esperando que el otro libere un recurso que él mismo necesita para continuar.
Condiciones para un Deadlock (Condiciones de Coffman): Un deadlock solo puede ocurrir si las siguientes cuatro condiciones se cumplen simultáneamente:

1. **Exclusión Mutua:** Al menos un recurso debe ser no compartible, es decir, solo un hilo puede usarlo a la vez.
2. **Retención y Espera (Hold and Wait):** Un hilo que ya posee al menos un recurso está esperando para adquirir recursos adicionales que están en posesión de otros hilos.
3. **No Apropiación (No Preemption):** Un recurso no puede ser arrebatado a la fuerza de un hilo; solo puede ser liberado voluntariamente por el hilo que lo posee.
4. **Espera Circular (Circular Wait):** Existe una cadena de dos o más hilos, donde cada hilo está esperando un recurso que posee el siguiente hilo en la cadena (p. ej., Hilo A espera el recurso de Hilo B, y Hilo B espera el recurso de Hilo A).

Prevención de Deadlocks: La estrategia para prevenir deadlocks es romper al menos una de las cuatro condiciones:

1. **Romper la Espera Circular:** Esta es la técnica de prevención más común y práctica.

Consiste en establecer un **orden global para la adquisición de bloqueos**. Todos los hilos deben adquirir los bloqueos siempre en el mismo orden predefinido. Si el Hilo A y el Hilo B necesitan los bloqueos X e Y, ambos deben intentar adquirir X primero y luego Y. Esto evita que A tenga X y espere por Y, mientras B tiene Y y espera por X.

2. **Romper la Retención y Espera:** Un hilo debe adquirir todos los bloqueos que necesita a la vez, o no adquirir ninguno. Si no puede obtener todos los bloqueos, debe liberar los que ya tiene y volver a intentarlo más tarde. Esto puede ser ineficiente y llevar a la inanición (*starvation*).
3. **Romper la No Apropiación:** Utilizar mecanismos de bloqueo que soporten tiempos de espera, como Lock.tryLock(timeout). Si un hilo no puede adquirir un bloqueo dentro de un tiempo determinado, libera los bloqueos que ya tiene y lo intenta de nuevo.
4. **Romper la Exclusión Mutua:** Esto generalmente no es posible, ya que la mayoría de los recursos compartidos requieren acceso exclusivo. Sin embargo, se pueden usar estructuras de datos no bloqueantes y algoritmos sin bloqueo (utilizando operaciones atómicas como CAS) para evitar la necesidad de bloqueos.

**Detección de Deadlocks

Fuentes citadas

1. 50+ Java Interview Questions You Need to Prepare For (With Answers & Tips!) - Utho, https://utho.com/blog/java-programming-interview-questions/ 2. Core Java Interview Questions and Answers (2025) - InterviewBit, https://www.interviewbit.com/java-interview-questions/ 3. Java Interview Questions and Answers - GeeksforGeeks, https://www.geeksforgeeks.org/java/java-interview-questions/ 4. Top JVM Interview Questions for Software Engineering Interviews ..., https://medium.com/double-pointer/top-jvm-interview-questions-for-software-engineering-interviews-0f48e21253c7 5. Top 30 JVM(Java Virtual Machine ... - JavaMadeSoEasy.com (JMSE), https://www.javamadesoeasy.com/2017/03/top-30-jvmjava-virtual-machine.html 6. 56 Java Interview Questions And Answers For All Levels - DataCamp, https://www.datacamp.com/blog/java-interview-questions 7. Java Data Type Interview Questions - CodeWithCurious, https://codewithcurious.com/interview/java-data-type-interview-questions/ 8. 35 Java Exception Handling Interview Questions and Answers, https://interviewkickstart.com/blogs/interview-questions/java-exception-handling-interview-questions 9. 52 Important OOP Interview Questions in 2025 - GitHub, https://github.com/Devinterview-io/oop-interview-questions 10. 40+ OOPs Interview Questions and Answers (2025) - InterviewBit, https://www.interviewbit.com/oops-interview-questions/ 11. 30 OOPs Interview Questions and Answers [2025 Updated ..., https://www.geeksforgeeks.org/oops-interview-questions/ 12. aershov24/design-patterns-interview-questions: Design ... - GitHub, https://github.com/aershov24/design-patterns-interview-questions 13. What are some real world Design Pattern interview questions? : r/learnjava - Reddit, https://www.reddit.com/r/learnjava/comments/1jtpu47/what_are_some_real_world_design_pattern_interview/ 14. 52 OOPs Interview Questions and Answers (2025) - Simplilearn.com, https://www.simplilearn.com/tutorials/java-tutorial/oops-interview-questions 15. 55 Java 8 Interview Questions and Answers (2025) - Simplilearn.com, https://www.simplilearn.com/java-8-interview-questions-and-answers-article 16. 40 Java Collections Interview Questions and Answers | DigitalOcean, https://www.digitalocean.com/community/tutorials/java-collections-interview-questions-and-answ

ers 17. Java Collections Interview Questions and Answers - GeeksforGeeks, https://www.geeksforgeeks.org/java/java-collections-interview-questions/ 18. Top Java Collection Interview Questions and Answers for 2025, https://www.simplilearn.com/tutorials/java-tutorial/java-collection-interview-questions 19. Top Java Collections Interview Questions (2025) - InterviewBit, https://www.interviewbit.com/java-collections-interview-questions/ 20. Crack the Top 50 Java Data Structure Interview Questions - Educative.io, https://www.educative.io/blog/top-50-java-data-structure-interview-questions 21. Java Multithreading Concurrency Interview Questions and Answers ..., https://www.digitalocean.com/community/tutorials/java-multithreading-concurrency-interview-questions-answers 22. 50 Interview Questions on Multithreading with Answers - DEV ..., https://dev.to/krishna7852/50-interview-questions-on-multithreading-with-answers-2f8n 23. Java Multithreading Interview Questions and Answers ..., https://www.geeksforgeeks.org/java/java-multithreading-interview-questions-and-answers/