

PROMETEO

Unidad 3: Estructuras de Control

```
1 // Condicionales
2 if (edad > 18) {
3     console.log("Eres mayor de edad");
4 } else {
5     console.log("Eres menor de edad");
6 }
7
8 if (edad > 18) {
9     console.log("Eres mayor de edad");
10 } else {
11     console.log("Eres menor de edad");
12 }
13
14 if (edad > 18) {
15     console.log("Eres mayor de edad");
16 }
```

Las estructuras condicionales son el mecanismo fundamental que permite a tu programa tomar decisiones inteligentes basadas en condiciones específicas.

Sesión 6: Condicionales (if, if-else, switch).

Las estructuras condicionales son el mecanismo fundamental que permite a tu programa tomar decisiones inteligentes basadas en condiciones específicas. Son la base de toda lógica empresarial, desde la validación más simple de un formulario hasta los algoritmos más complejos de inteligencia artificial que determinan qué contenido mostrar a cada usuario.

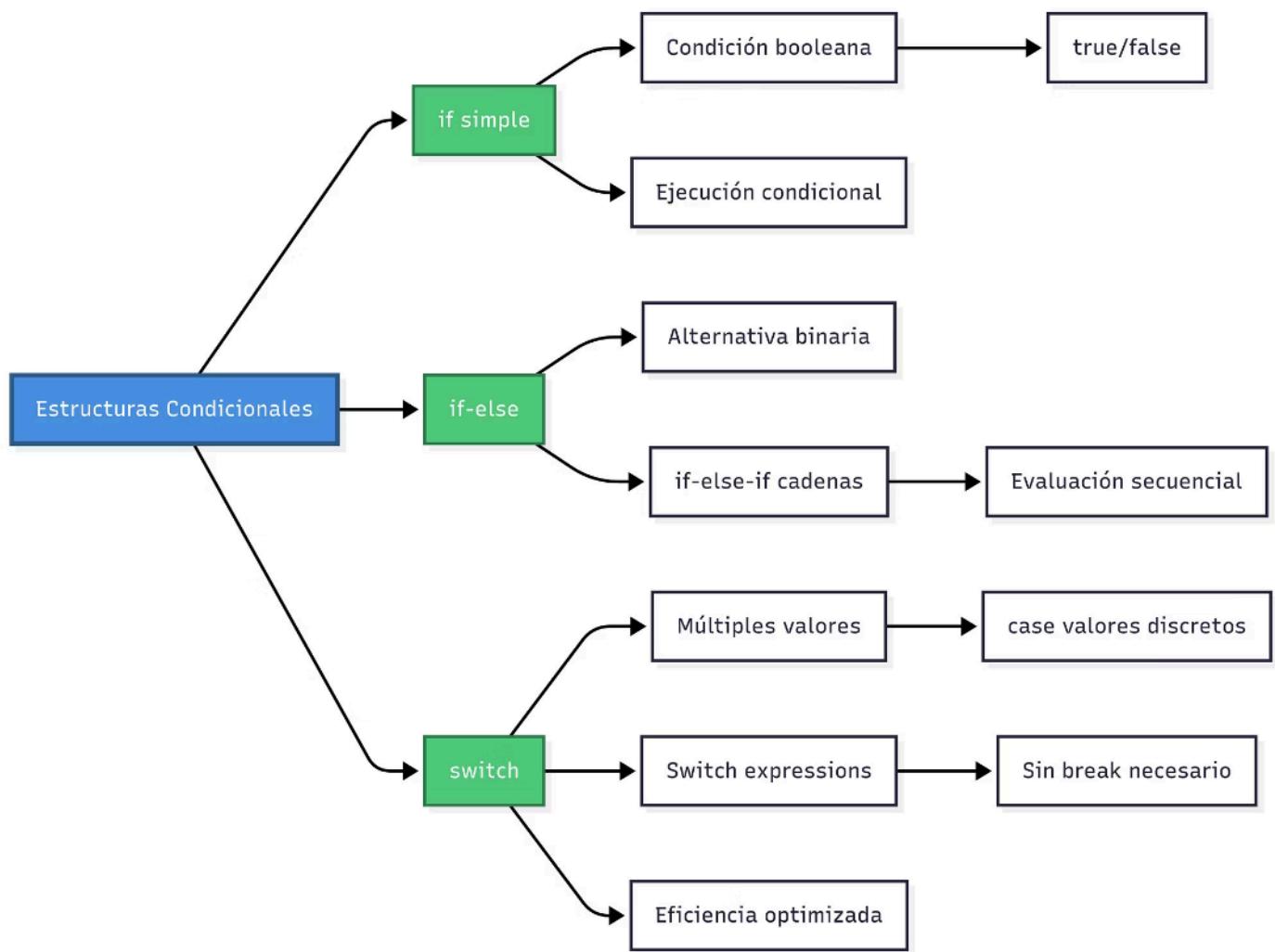
La estructura if evalúa una expresión booleana y ejecuta un bloque de código solo si el resultado es verdadero. Su simplicidad es engañosa: esta estructura básica es la base de sistemas que procesan millones de transacciones diarias. if-else añade una alternativa para cuando la condición es falsa, permitiendo que tu programa responda apropiadamente a ambos escenarios posibles.

Las cadenas if-else-if permiten evaluar múltiples condiciones mutuamente excluyentes de forma ordenada y legible. **Cada condición se evalúa secuencialmente hasta encontrar la primera verdadera**, ejecutando su bloque correspondiente e ignorando las restantes. Esta estructura es ideal para categorizar datos, determinar rangos de valores, o implementar lógica de negocio con múltiples casos.

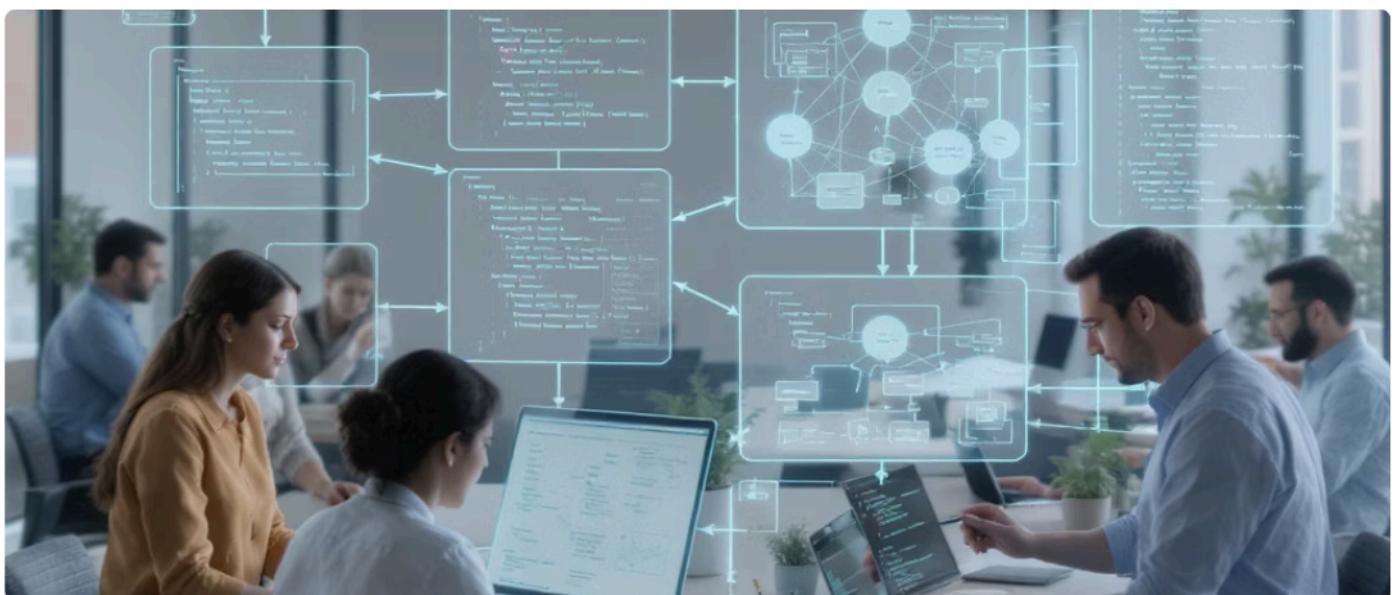
El switch es la estructura óptima cuando comparas una variable contra múltiples valores específicos y discretos. Es más legible y eficiente que cadenas largas de if-else cuando tienes muchas opciones. Desde Java 14, las switch expressions eliminan la necesidad de break statements y permiten asignaciones directas, haciendo el código más conciso y menos propenso a errores.



Esquema visual



El diagrama muestra cómo las estructuras condicionales proporcionan diferentes niveles de complejidad y optimización. `if` para decisiones simples, `if-else` para alternativas, cadenas `if-else-if` para múltiples condiciones relacionadas, y `switch` para múltiples valores discretos de una misma variable.





Caso de Estudio: Sistema de Descuentos de Amazon

Amazon procesa más de 5 mil millones de búsquedas y 300 millones de transacciones diarias, utilizando estructuras condicionales complejas para calcular descuentos dinámicos que maximizan conversión y rentabilidad. Su algoritmo evalúa múltiples factores simultáneamente: tipo de cliente, categoría de producto, temporada, inventario disponible, y comportamiento de compra histórico.

El sistema utiliza cadenas if-else-if sofisticadas para segmentar clientes: if (isPrime && purchaseAmount > 50) aplica envío gratuito, else if (purchaseAmount > 25 && isFirstPurchase) ofrece descuento de bienvenida, else if (isReturningCustomer && daysSinceLastPurchase > 30) activa ofertas de reactivación. Cada condición está cuidadosamente ordenada por prioridad comercial.

Para categorías de productos utilizan switch expressions modernas:

```
return switch(category) { case "ELECTRONICS" ->
    calculateTechDiscount(); case "BOOKS" -> applyEducationalDiscount();
    case "FASHION" -> seasonalFashionDiscount(); default ->
    standardDiscount(); }. Esta estructura permite añadir nuevas categorías fácilmente sin modificar lógica existente.
```

El impacto es significativo: este sistema de decisiones condicionales genera más de **40% de los ingresos de Amazon** a través de recomendaciones y ofertas personalizadas, procesando decisiones en menos de 50 milisegundos para mantener la experiencia de usuario fluida.

Herramientas y Consejos

Claridad en Condiciones

Utiliza paréntesis para clarificar condiciones complejas: if ((age >= 18 && hasLicense) || isEmergency) es más legible que confiar en la precedencia de operadores. La claridad siempre supera la brevedad en código profesional.

Switch Expressions

Prefiere switch expressions en Java 14+ para evitar olvidos de break: String result = switch(day) { case "MONDAY" -> "Inicio de semana"; case "FRIDAY" -> "Fin de semana cerca"; default -> "Día normal"; }. Son más seguras y concisas que switch tradicionales.

Evitar Anidamiento

Evita if-else profundamente anidados usando return temprano o extrayendo métodos auxiliares. Código con más de 3 niveles de anidamiento se vuelve difícil de entender y mantener.

Mitos y Realidades

 **Mito:** "switch es siempre más rápido que if-else"

→ **FALSO.** Para pocas condiciones (2-3), if-else puede ser más rápido debido a optimizaciones del compilador. switch es más eficiente con muchas opciones (5+) porque el compilador puede generar jump tables, pero la diferencia es mínima en aplicaciones típicas. La elección debe basarse en legibilidad y mantenibilidad.

 **Mito:** "Más condiciones hacen el programa más inteligente"

→ **FALSO.** Condiciones excesivas crean complejidad innecesaria y bugs potenciales. El principio KISS (Keep It Simple, Stupid) aplica especialmente a lógica condicional. Amazon optimiza constantemente sus algoritmos eliminando condiciones redundantes, no añadiendo más.

Resumen final para el examen

Condicionales: if (simple), if-else (alternativa), if-else-if (múltiples condiciones), switch (valores discretos). Elegir estructura según contexto y legibilidad. Switch expressions más seguras que tradicionales. Amazon exemplifica uso profesional en descuentos dinámicos. Evitar anidamiento excesivo para mantenibilidad.

Sesión 7: Bucles while, do-while, for

Los bucles son estructuras de control que permiten ejecutar código repetidamente mientras se cumpla una condición específica. Son fundamentales para automatizar tareas que serían imposibles o impracticables realizar manualmente, desde procesar millones de registros en una base de datos hasta generar interfaces de usuario dinámicas que se adaptan a diferentes cantidades de contenido.

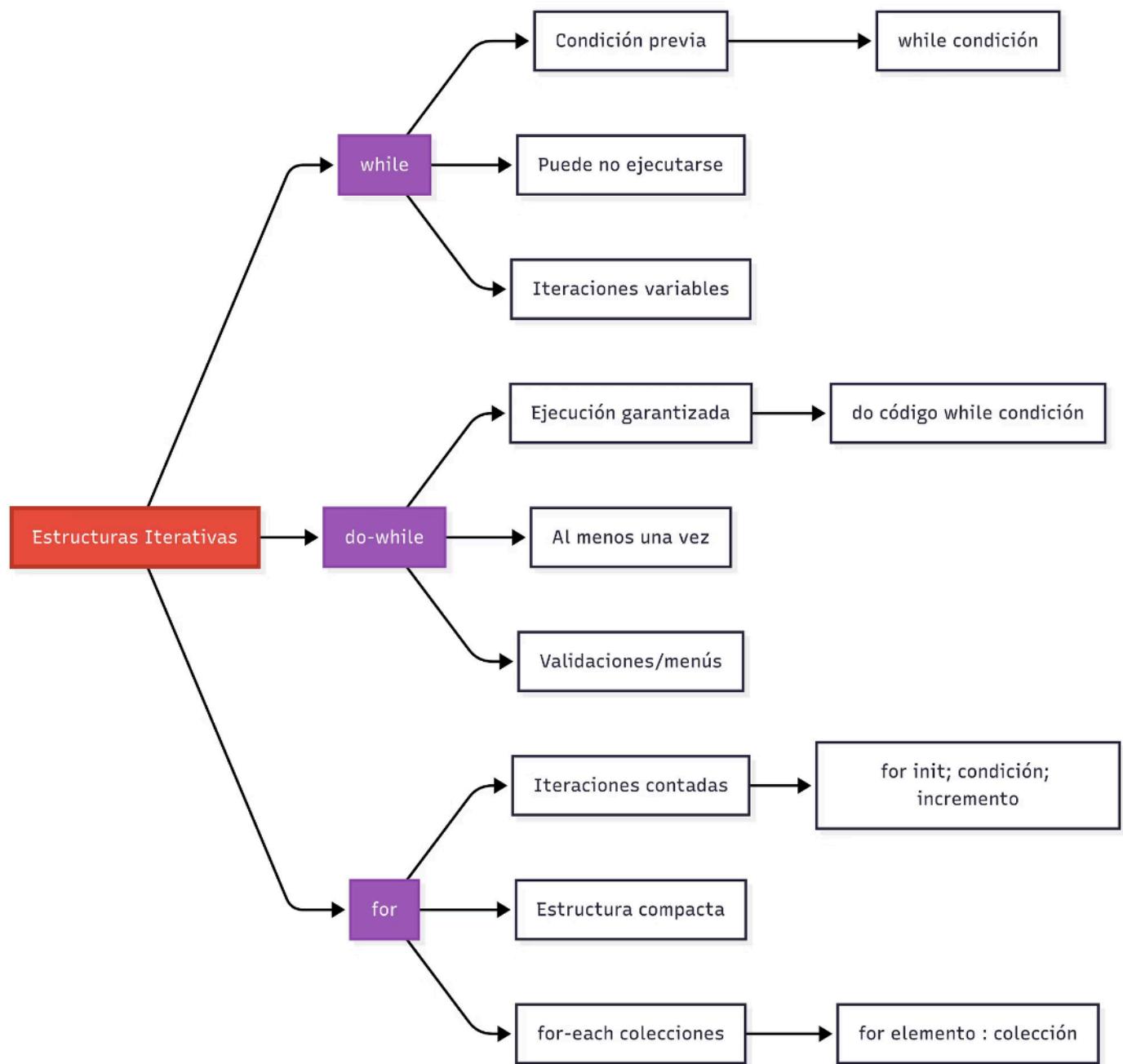
El bucle while evalúa la condición antes de cada iteración, lo que significa que si la condición es falsa desde el inicio, el código nunca se ejecuta. Esta característica lo hace ideal para situaciones donde no conoces exactamente cuántas iteraciones necesitas: leer un archivo hasta el final, procesar datos hasta que se agoten, o esperar una respuesta de red con timeout.

El bucle do-while garantiza que el código se ejecute al menos una vez antes de evaluar la condición. Esta garantía es crucial para menús de usuario, validaciones de entrada, o cualquier situación donde necesitas al menos un intento antes de decidir si continuar. La diferencia con while puede parecer sutil, pero es fundamental en muchos algoritmos.

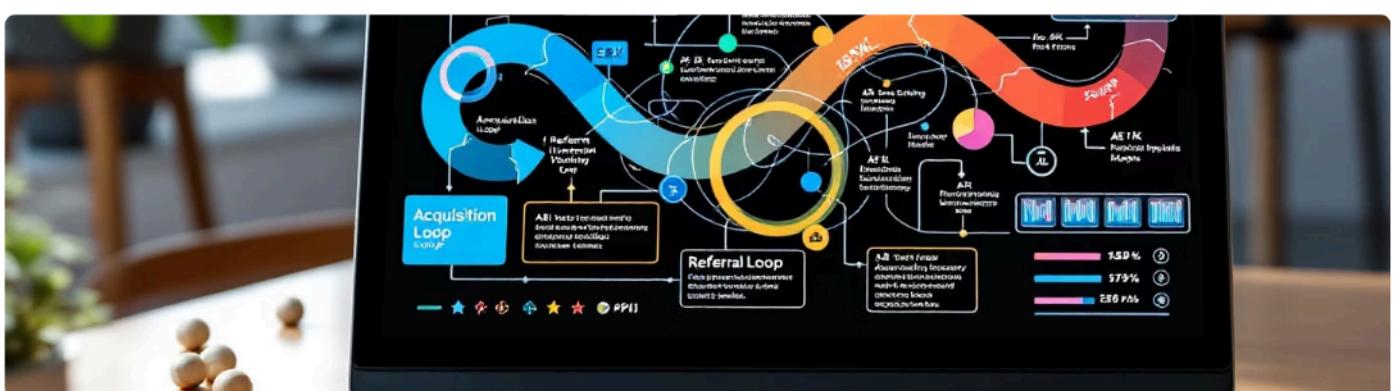
El bucle for es la elección perfecta cuando conoces el número exacto de iteraciones o trabajas con rangos específicos. Su estructura compacta (inicialización; condición; incremento) hace el código más legible y menos propenso a errores como bucles infinitos. El for-each, introducido en Java 5, simplifica enormemente la iteración sobre colecciones.



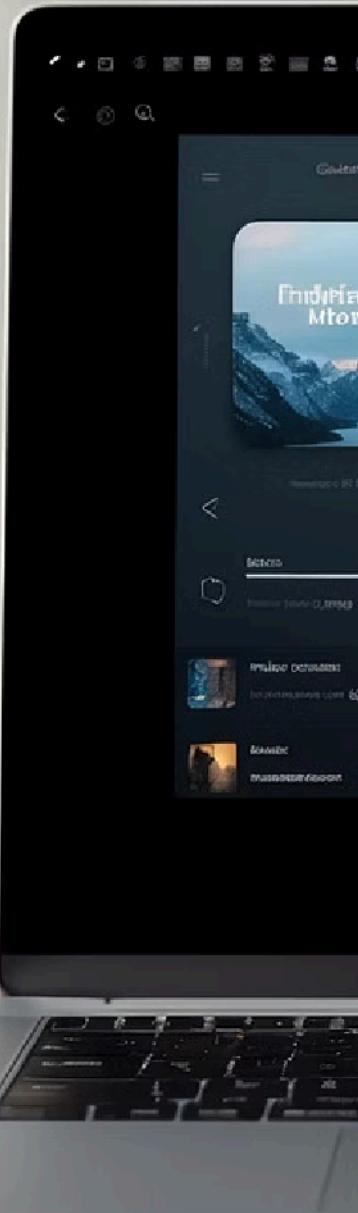
Esquema Visual de Bucles



Este diagrama ilustra las características distintivas de cada tipo de bucle y sus casos de uso óptimos. La elección correcta depende de si conoces el número de iteraciones, si necesitas garantizar al menos una ejecución, y si trabajas con colecciones.



Caso de Estudio: Procesamiento de Datos de Spotify



Spotify procesa más de **5 mil millones de streams diarios** utilizando diferentes tipos de bucles según la naturaleza específica de cada tarea de procesamiento. Para analizar playlists utilizan bucles for porque conocen exactamente el número de canciones a procesar. Para streams en tiempo real utilizan while porque no saben cuándo terminará la sesión del usuario.

Su sistema de recomendaciones utiliza do-while para garantizar que cada usuario reciba al menos una recomendación, incluso si sus datos de escucha son limitados o atípicos. El algoritmo itera buscando canciones similares hasta encontrar suficientes opciones o alcanzar un máximo de intentos predefinido para evitar bucles infinitos.

Los bucles procesan cantidades masivas de información: 2.9 mil millones de horas de música mensualmente, análisis de comportamiento de 500+ millones de usuarios, y generación de más de 4 mil millones de recomendaciones diarias. Un bucle infinito accidental podría colapsar servidores y afectar la experiencia de millones de usuarios simultáneamente.

Por esta razón, implementan múltiples mecanismos de seguridad: timeouts automáticos, contadores máximos de iteraciones, monitoreo en tiempo real de bucles que exceden umbrales esperados, y circuit breakers que detienen procesamiento si detectan patrones anómalos.

Herramientas y Consejos para Bucles



Seguridad en Bucles

Incluye siempre una forma clara de salir del bucle para evitar bucles infinitos. Usa contadores de seguridad: int maxAttempts = 1000; while(condition && attempts < maxAttempts) como red de seguridad en bucles while.



For-each para Colecciones

Prefiere for-each para recorrer colecciones: for(String song : playlist) es más legible, seguro, y eficiente que índices manuales. Elimina errores comunes como IndexOutOfBoundsException.



Control de Flujo

Utiliza break para salir anticipadamente del bucle y continue para saltar a la siguiente iteración sin ejecutar el resto del código. Estas instrucciones proporcionan control fino sobre el flujo de ejecución.

Mitos y Realidades

X Mito: "for es siempre más eficiente que while"

→ **FALSO.** El compilador de Java optimiza ambos tipos de bucle de forma similar cuando la lógica es equivalente. La elección debe basarse en legibilidad y propósito: for para iteraciones contadas, while para condiciones dinámicas. La diferencia de rendimiento es negligible en aplicaciones típicas.

X Mito: "Los bucles infinitos siempre son errores de programación"

→ **FALSO.** En aplicaciones servidor, servicios en tiempo real, y sistemas operativos, bucles infinitos controlados son normales y necesarios para mantener servicios activos continuamente. La clave está en tener mecanismos de control y salida apropiados.

Resumen final para el examen

Bucles: while (condición previa), do-while (al menos una ejecución), for (iteraciones contadas). Elegir según conocimiento de iteraciones y garantías necesarias. for-each para colecciones. Spotify ejemplifica uso masivo con 2.9 mil millones horas procesadas. Incluir salidas de seguridad siempre.

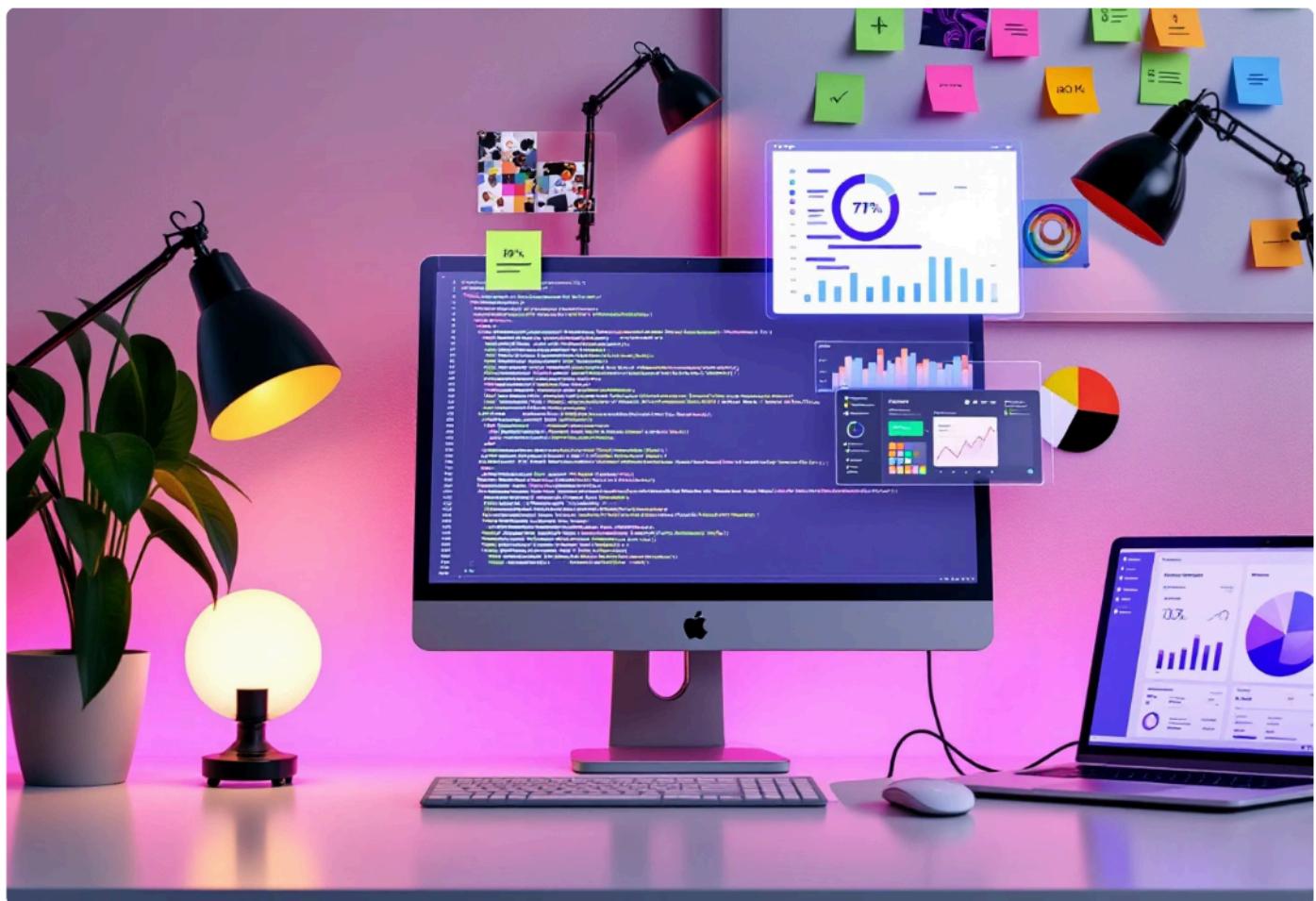
Sesión 8: Anidamiento de estructuras y buenas prácticas

El anidamiento de estructuras de control significa colocar bucles dentro de condicionales, condicionales dentro de bucles, o cualquier combinación de estas estructuras para resolver problemas complejos. Aunque es una herramienta poderosa, el anidamiento mal gestionado puede crear código incomprensible y propenso a errores, por lo que requiere disciplina y técnicas específicas para mantener la calidad.

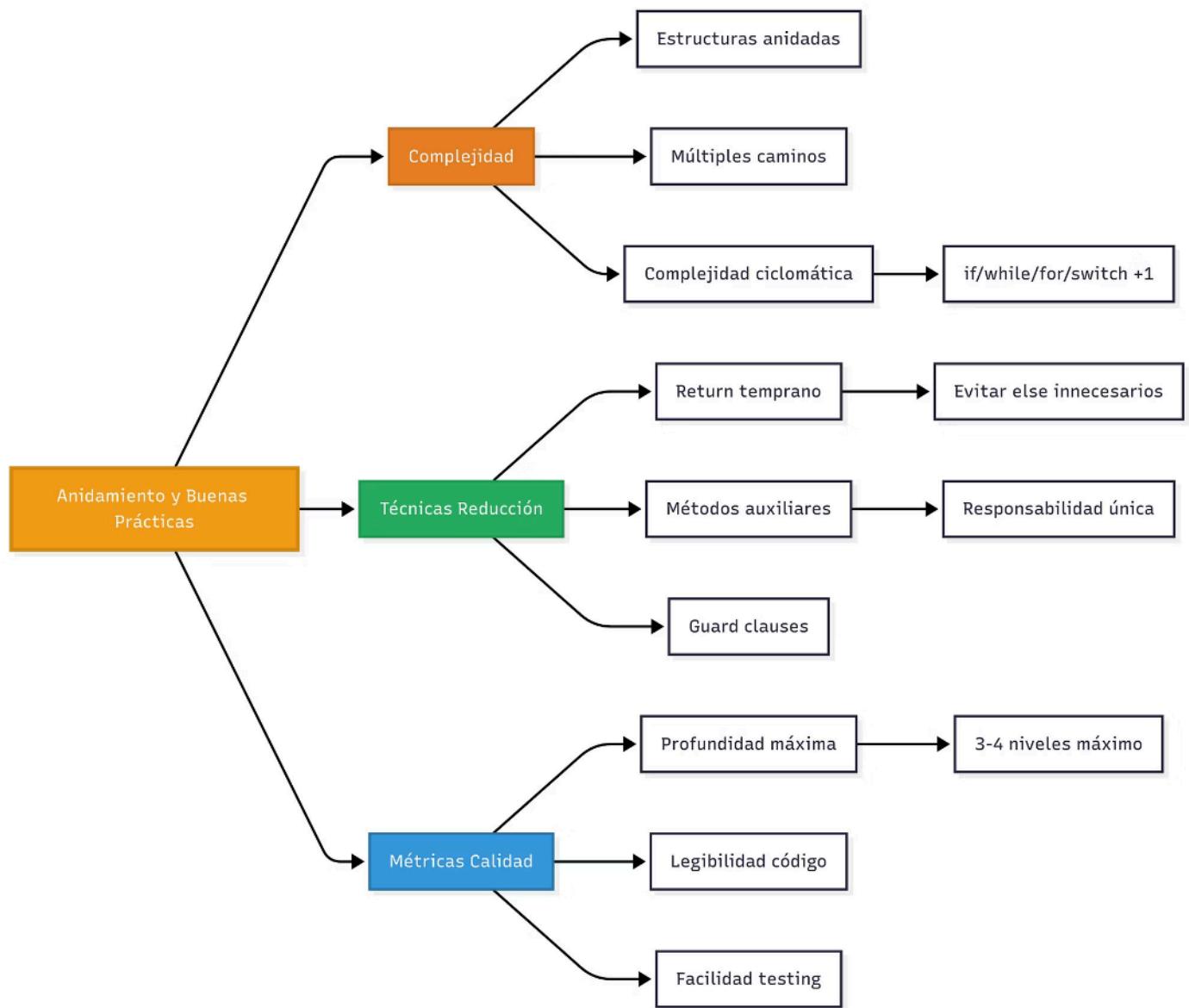
La **complejidad ciclomática** es una métrica que mide cuántos caminos diferentes puede tomar tu código durante la ejecución. Cada if, while, for, switch case, y operador lógico (`&&`, `||`) añade un punto de complejidad. Mantener esta métrica baja (idealmente menos de 10) hace el código más fácil de entender, testear, y mantener a largo plazo.

Las buenas prácticas para manejar anidamiento incluyen limitar la profundidad máxima (no más de 3-4 niveles), usar nombres descriptivos para variables de control, extraer lógica compleja a métodos separados con responsabilidades específicas, y aplicar el principio de "return temprano" para evitar else innecesarios que aumentan la indentación.

Técnicas avanzadas para reducir complejidad incluyen el uso de métodos auxiliares que encapsulan lógica específica, operadores ternarios para condiciones simples, y patrones como guard clauses que validan condiciones al inicio del método y retornan temprano si no se cumplen, evitando anidamiento profundo en el resto del código.



Esquema visual



El diagrama muestra cómo la complejidad del código se puede controlar mediante técnicas específicas y métricas objetivas. El objetivo es mantener funcionalidad completa mientras se preserva la legibilidad y mantenibilidad.





Caso de Estudio: Algoritmo de Búsqueda de Google

Google procesa más de **8.5 mil millones de búsquedas diarias** utilizando algoritmos con estructuras anidadas extremadamente complejas. Su algoritmo PageRank original combinaba bucles for para recorrer miles de millones de páginas web, condicionales if para evaluar relevancia según múltiples criterios, y bucles while para iterar hasta que los valores de ranking convergieran.

El algoritmo original tenía anidamiento profundo que dificultaba optimizaciones y parallelización. Google lo refactorizó sistemáticamente extrayendo funciones especializadas: calculateRelevance(), updatePageRank(), checkConvergence(), filterSpamSites(). Cada función tiene una responsabilidad única y complejidad controlada, típicamente menos de 7 puntos ciclomáticos.

Esta refactorización permitió paralelización masiva en miles de servidores, testing independiente de cada componente, y optimizaciones específicas por función. El resultado: tiempo de respuesta promedio de **0.2 segundos** para búsquedas que analizan trillones de páginas web, procesando consultas en más de 150 idiomas simultáneamente.

La lección clave es que la complejidad del problema no requiere código complejo. Los algoritmos más sofisticados del mundo se construyen combinando funciones simples y bien definidas, no con estructuras anidadas profundamente.

Herramientas y Consejos Finales

Regla de 3 Niveles

Aplica la regla "no más de 3 niveles de anidamiento" como guía estricta. Si necesitas más profundidad, es señal clara de que debes extraer lógica a métodos separados con nombres descriptivos que expliquen su propósito específico.

Métricas Automáticas

Configura tu IDE para mostrar warnings cuando la complejidad ciclomática supere 10. IntelliJ IDEA, Eclipse, y VSCode tienen plugins que calculan esta métrica automáticamente y resaltan métodos problemáticos.

Principio Fail Fast

Implementa el principio "fail fast": valida condiciones de error al inicio del método y usa return temprano para evitar anidamiento innecesario. Esto hace el código más legible y eficiente.

Mitos y Realidades

X Mito: "Más anidamiento significa código más sofisticado y potente"

→ **FALSO.** Código profesional prioriza simplicidad y legibilidad sobre complejidad aparente. Anidamiento excesivo indica necesidad de refactoring, no sofisticación. Los mejores programadores escriben código que parece simple pero resuelve problemas complejos elegantemente.

X Mito: "Extraer métodos hace el programa más lento por las llamadas adicionales"

→ **FALSO.** Los compiladores modernos optimizan llamadas a métodos pequeños automáticamente (inlining). El beneficio en mantenibilidad, testing, y legibilidad supera ampliamente cualquier overhead teórico, que en la práctica es inexistente.

Resumen final para el examen

Anidamiento: estructuras dentro de estructuras, requiere control de complejidad.

Complejidad ciclomática: métrica de caminos código, mantener <10. Técnicas: return temprano, métodos auxiliares, guard clauses. Google ejemplifica refactoring exitoso para rendimiento masivo. Simplicidad supera complejidad aparente.