

PROMETEO

Unidad 2.

Ampliación en

Estructuras

Básicas

En Java conviven dos formas de representar los datos básicos: tipos primitivos (int, double, char, boolean, etc.) y clases envoltorio (wrappers) (Integer, Double, Character, Boolean, ...).

Sesión 5: Tipos primitivos vs objetos. Autoboxing y unboxing

Fundamentos: elegir entre valor "puro" o valor "con contexto"

En Java conviven dos formas de representar los datos básicos: tipos primitivos (int, double, char, boolean, etc.) y clases envoltorio (wrappers) (Integer, Double, Character, Boolean, ...). La diferencia esencial es cómo se almacenan y para qué se usan:

Los primitivos guardan el valor directamente (en registros/stack según el caso). Son más rápidos y consumen menos memoria. Son ideales para cálculos intensivos y estructuras de datos muy usadas en tiempo crítico (por ejemplo, sumar millones de números en un bucle).

Los wrappers son objetos que encapsulan un valor primitivo y añaden métodos y comportamiento (por ejemplo, Integer.parseInt, comparaciones, conversión a String). Se almacenan en el heap, pueden ser nulos y se integran con el ecosistema de objetos de Java (colecciones genéricas, APIs que esperan referencias, etc.).

Para reducir la fricción entre ambos mundos, Java introdujo:

Autoboxing: conversión automática de primitivo → objeto.

```
Integer n = 5; // autoboxing (int -> Integer)
```

Unboxing: conversión automática de objeto → primitivo.

```
int x = n; // unboxing (Integer -> int)
```

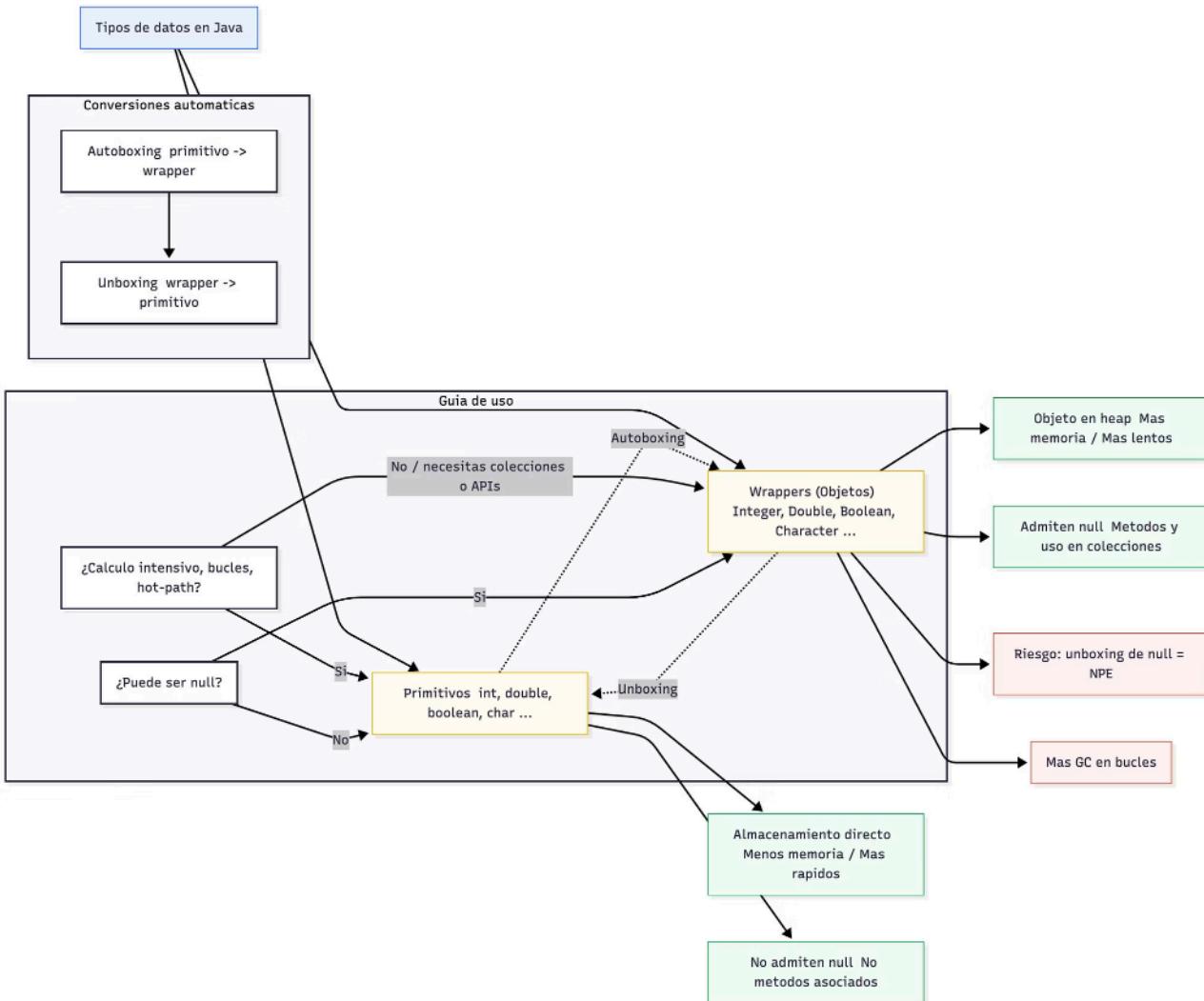
Estas conversiones aumentan la legibilidad (menos "ruido" de casting), pero no son gratis. Crear objetos (autoboxing) y extraer su valor (unboxing) implica asignaciones en memoria, posibles comprobaciones y más GC. En bucles o rutas críticas, ese coste se acumula. Además, el unboxing de null lanza NullPointerException, un riesgo habitual al trabajar con Integer, Double, etc.

Por eso, saber cuándo usar primitivos y cuándo wrappers no es un detalle estético: es clave para el rendimiento, la seguridad y la claridad del código.

En síntesis:

- **Primitivo** = rendimiento y bajo coste.
- **Wrapper** = interoperabilidad con colecciones/APIs, posibilidad de null, métodos utilitarios.
- **Autoboxing/unboxing** = comodidad... con coste. Úsalos conscientemente, no en caliente (bucles con millones de iteraciones) y evita unboxing de posibles null si no lo controlas.

Esquema Visual: mapa de decisión y flujo de autoboxing/unboxing



Cómo leer el diagrama

- Bloque B (primitivos): máximo rendimiento; no null; menos flexible.
- Bloque C (wrappers): flexibles, colecciones y APIs, pueden ser null, pero más coste.
- Bloque D: flechas discontinuas muestran autoboxing/unboxing automáticos.
- Bloque E: matriz de decisión práctica. Si la ruta es crítica en tiempo o memoria → primitivo. Si necesitas null, métodos de clase, o colecciones genéricas (List) → wrapper.
- Alertas: null + unboxing = NullPointerException; más objetos = más trabajo para el GC y potencial caída de rendimiento en bucles.



Caso de Estudio: Microoptimizaciones con grandes efectos (LinkedIn)

Contexto. Plataformas con altísimo tráfico (por ejemplo, LinkedIn) ejecutan cálculos masivos para ranking y recomendación: contadores, estadísticas, agregaciones sobre millones de eventos. En esos escenarios, una diferencia "pequeña" por operación se multiplica por miles de millones.

Estrategia. En módulos "hot-path" (bucles intensivos y pipelines de datos), el equipo prioriza primitivos (int, double) en lugar de Integer/Double para:

- Evitar autoboxing implícito dentro de bucles (creación de objetos).
- Reducir presión sobre el GC por objetos temporales.
- Eliminar el riesgo de unboxing sobre null (NPE) en datos intermedios.

Comparativa típica (simplificada):

```
// Variante menos eficiente (autoboxing en cada iteración)
Integer total = 0;
for (int i = 0; i < N; i++) {
    total += i; // autoboxing/unboxing repetidos}
```

```
// Variante eficiente (primitivo)
int sum = 0;
for (int i = 0; i < N; i++) {
    sum += i; // sin objetos temporales}
```

Resultado. En tareas de alto volumen:

- La versión primitiva reduce asignaciones y pausas de GC, disminuyendo latencia y uso de memoria.
- Las colecciones de trabajo (por ejemplo, tablas hash internas) se mantienen con representaciones compactas y conversiones controladas en los bordes (solo cuando una API exige objetos).

Lección. Un pequeño ajuste de tipos en bucles críticos puede producir grandes mejoras de tiempo de respuesta y footprint de memoria. La clave está en usar wrappers solo cuando aportan valor (colecciones, null, APIs) y mantener primitivos en el corazón del cómputo.

Herramientas y Consejos (aplicación inmediata)

Perfilado antes de afirmar

Mide, no adivines. Usa VisualVM, IntelliJ Profiler o JConsole para comparar:

- Tiempo de ejecución de int vs Integer en bucles largos.
- Cantidad de asignaciones y pausas de GC.
- Hotspots de métodos con autoboxing accidental.

Evita autoboxing en rutas críticas

- Escribe los acumuladores y contadores como primitivos.
- Si una API exige List, separa: cálculo con int, conversión a Integer solo al final.
- Revisa warnings del IDE: muchos detectan autoboxing implícito.

Cuida el null en wrappers

Antes de unboxing, comprueba null:

```
Integer maybe = getValue();
int safe = (maybe != null) ? maybe : 0;
```

En DTOs/entradas externas, valida y normaliza (evita NPE en unboxing).

Colecciones: entiende por qué necesitan wrappers

Las colecciones genéricas (List, Map, Set) trabajan con referencias. Por eso List no existe. Usa List cuando realmente necesitas una colección; si solo procesas secuencias numéricas grandes, valora arrays primitivos (int[]) o estructuras especializadas (por ejemplo, bibliotecas con primitive collections).

Evita autoboxing en expresiones mixtas

No mezcles Integer y int en la misma operación si no es necesario; fuerza el lado primitivo. Por ejemplo:

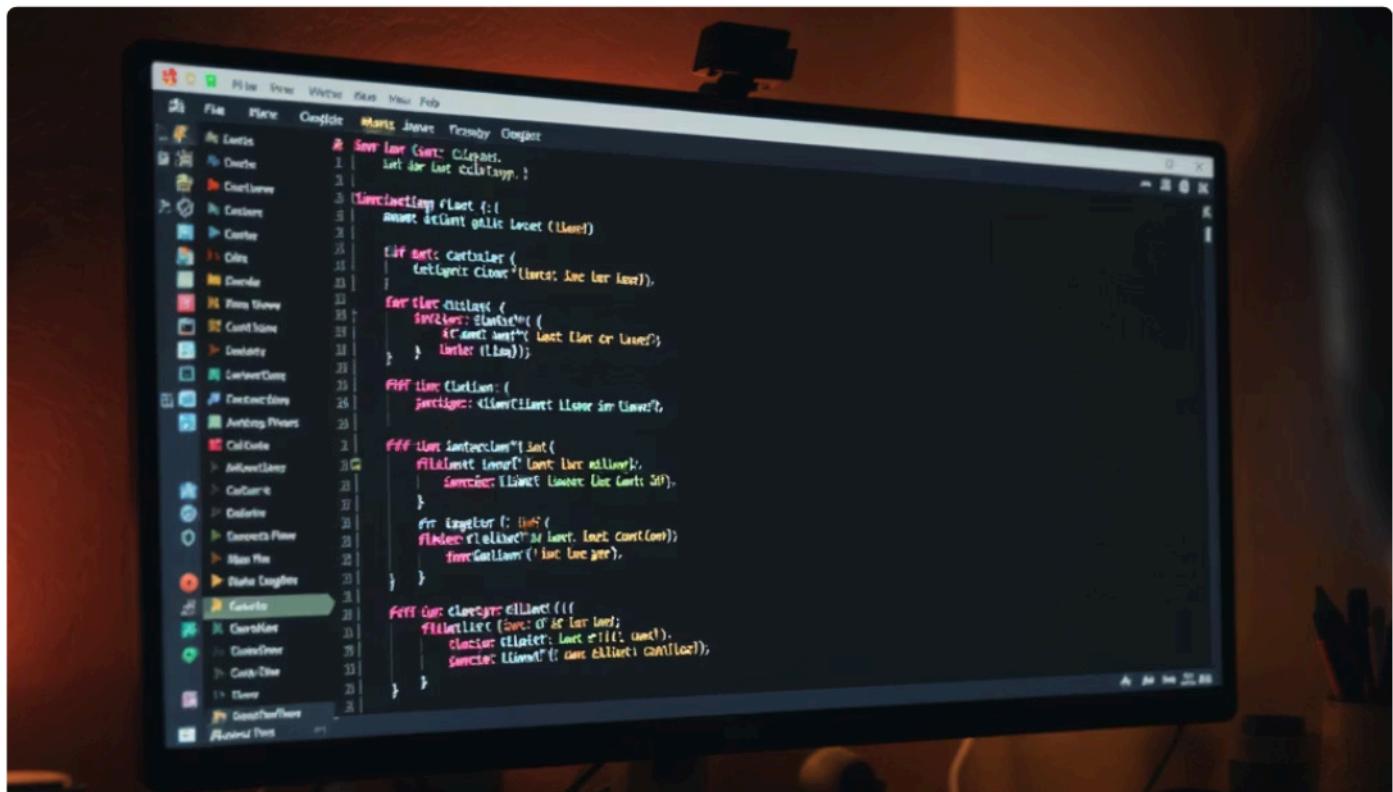
```
int base = 10;  
Integer inc = 2;  
int r = base + inc.intValue(); // explícito, evita autobox/unbox implícito
```

Activa inspecciones del IDE y reglas estáticas

- IntelliJ / SonarLint detectan autoboxing innecesario, unboxing peligroso y comparaciones == entre wrappers.
- Usa comparaciones seguras: Objects.equals(a, b) para Integer (evita sorpresas por caching e interning).

Comparaciones y constante de tiempo

- Evita == con Integer para comparar valores numéricos; usa equals.
- == en wrappers compara referencia, no valor (aunque a veces "parezca" funcionar por el Integer cache entre -128 y 127; no dependas de ello).



Mitos y Realidades

✗ Mito: "Los objetos son siempre mejores que los primitivos."

→ **FALSO.** Los objetos aportan flexibilidad (métodos, null, colecciones), pero pagan con más memoria, más GC y riesgo de NullPointerException al hacer unboxing. En cálculo intensivo, los primitivos son la opción eficiente.

✗ Mito: "El autoboxing es gratis y siempre conviene usarlo por legibilidad."

→ **FALSO.** El autoboxing crea objetos y el unboxing puede lanzar NPE. En rutas calientes (bucles, agregaciones) degrada rendimiento. Úsalo con criterio: legibilidad sí, pero no a costa de latencia y memoria.

☐ Resumen Final (para examen)

- **Primitivos:** valor directo, más rápidos y ligeros, sin null.
- **Wrappers:** objetos con métodos, admiten null, necesarios en colecciones y algunas APIs.
- **Autoboxing:** primitivo → objeto (cómodo pero con coste).
- **Unboxing:** objeto → primitivo (cuidado con null).
- Elige primitivos en cálculos intensivos; wrappers cuando necesites referencias/colecciones o null.



Sesión 6: Gestión de memoria en Java – Stack, Heap y Garbage Collector

Fundamentos: Cómo Java administra la memoria automáticamente

Comprender la gestión de memoria es fundamental para escribir programas eficientes y estables en Java. Aunque el lenguaje simplifica la vida del desarrollador al encargarse de muchas tareas internas, entender qué ocurre "bajo el capó" marca la diferencia entre un código funcional y un código realmente optimizado.

A diferencia de lenguajes como C o C++, en los que el programador debe reservar y liberar memoria manualmente, Java se ejecuta dentro de la JVM (Java Virtual Machine). Esta máquina virtual abstrae el hardware y administra la memoria de forma automática, pero eso no significa que podamos olvidarnos de ella: malas prácticas o desconocimiento del funcionamiento interno pueden generar fugas, lentitud o bloqueos.

Las dos grandes áreas de memoria: Stack y Heap

Stack (pila)

- Almacena variables locales, parámetros de métodos y referencias temporales.
- Se gestiona automáticamente con el flujo de ejecución: cuando un método termina, su espacio en el stack se libera.
- Es extremadamente rápido, ya que la memoria se gestiona en bloques (LIFO: Last In, First Out).
- Los tipos primitivos se guardan directamente en el stack (por ejemplo, int x = 5;).
- Las referencias a objetos también se almacenan aquí, aunque los objetos propiamente dichos viven en el heap.

Heap (montículo)

- Es la región de memoria donde se guardan todos los objetos y sus datos asociados.
- Tiene una vida más larga y se gestiona dinámicamente.
- Permite la creación de objetos en tiempo de ejecución mediante new.
- Su tamaño puede ajustarse con parámetros de la JVM (-Xms, -Xmx).
- Es más flexible, pero más lento de acceder y susceptible a fragmentación si se abusa de la creación de objetos.

Ejemplo conceptual:

```
int a = 10; // a se almacena en el stack  
String s = new String("Hi"); // referencia en stack, objeto "Hi" en heap
```

El Garbage Collector (GC): el limpiador invisible

El Garbage Collector es un proceso de la JVM que busca y elimina objetos del heap que ya no tienen referencias activas. En otras palabras, si ningún hilo o variable apunta a un objeto, ese espacio puede reciclarse. El GC trabaja en segundo plano, liberando memoria sin intervención del programador.

Sin embargo, esta "magia" tiene costes:

- Interrumpe momentáneamente los hilos para hacer su limpieza (Stop the World).
- Su eficiencia depende del patrón de creación de objetos.
- En aplicaciones con miles de instancias temporales (por ejemplo, microservicios de alto tráfico), una mala gestión puede provocar pausas frecuentes o consumo excesivo.

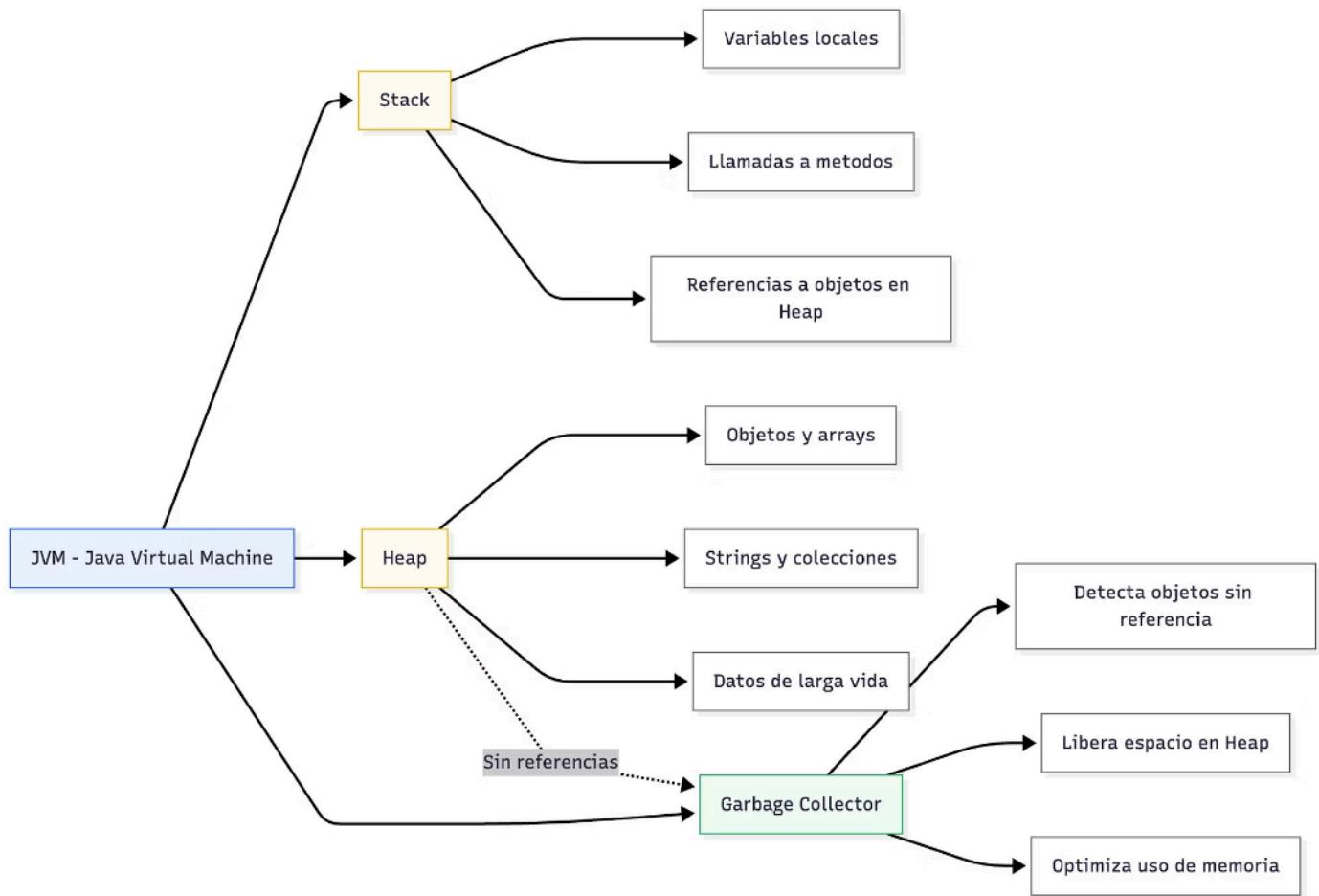
En resumen:

- **Stack** = rápido y efímero.
- **Heap** = dinámico y persistente.
- **Garbage Collector** = mecanismo automático que mantiene el heap limpio, pero no exime de responsabilidad.

La clave profesional está en diseñar código que ayude al GC, no que dependa completamente de él.



Esquema Visual: Arquitectura de memoria en la JVM



Explicación del esquema:

- La JVM coordina las tres zonas principales: stack, heap y garbage collector.
- El stack guarda variables locales y las referencias a objetos.
- El heap contiene los objetos propiamente dichos, accesibles mientras exista alguna referencia válida.
- El garbage collector monitoriza el heap constantemente, elimina lo que ya no se usa y compacta el espacio.
- Flecha discontinua: cuando un objeto del heap pierde su referencia en el stack, el GC lo marca como "candidato a eliminación".

Este modelo asegura equilibrio entre rendimiento, seguridad y automatización —una de las razones por las que Java sigue siendo el estándar de la industria.



Caso de Estudio: Instagram y la optimización del heap

Contexto

Instagram, como muchas aplicaciones masivas, procesa millones de imágenes y metadatos cada minuto. Su backend en Java maneja operaciones intensivas en memoria (filtros, metadatos, cachés). El equipo de ingeniería detectó que el Garbage Collector se activaba con demasiada frecuencia, causando microcortes y latencias elevadas en las respuestas del servidor.

Estrategia

El análisis de heap (mediante VisualVM y Eclipse Memory Analyzer) reveló que se generaban miles de objetos temporales en operaciones repetitivas —por ejemplo, cadenas (String) y colecciones intermedias creadas dentro de bucles.

Para optimizar:

- Reducieron la creación de objetos temporales reutilizando estructuras (StringBuilder, List.clear() en lugar de new ArrayList<> en cada iteración).
- Aplicaron pooling de objetos: mantuvieron instancias reutilizables para evitar asignaciones y liberaciones constantes.
- Ajustaron parámetros de la JVM:
 - -Xms2G -Xmx4G → definen tamaño inicial y máximo del heap.
 - -XX:+UseG1GC → seleccionaron el Garbage Collector G1, optimizado para baja latencia.

Resultado

20%

Reducción tiempo respuesta

Tiempo de respuesta promedio mejorado

30%

Menos consumo memoria

Disminución del consumo total de memoria

Menos pausas del GC (de varias por segundo a menos de una cada 10 segundos).

Conclusión

Optimizar la memoria no significa eliminar objetos, sino gestionar su ciclo de vida con inteligencia. El GC es eficiente, pero su trabajo depende directamente de las decisiones de diseño del programador.

Herramientas y Consejos Profesionales

Consejos prácticos

Evita crear objetos dentro de bucles críticos

Cada new en una iteración genera presión sobre el heap. Si es posible, declara y reutiliza fuera del bucle.

Usa StringBuilder o StringBuffer para concatenar cadenas

Las concatenaciones con + crean nuevas instancias de String (inmutables). StringBuilder reduce esa sobrecarga drásticamente.

Libera referencias explícitamente

Cuando un objeto deja de ser necesario, asigna su referencia a null para que el GC pueda eliminarlo antes.

```
dataCache = null; // facilita recolección temprana
```

Prefiere tipos primitivos cuando sea posible

Los wrappers (Integer, Double, etc.) generan más objetos y, por tanto, más carga al heap.

Usa objetos inmutables con criterio

Son más seguros y evitan fugas, pero pueden generar más instancias si se abusan en cálculos masivos.

Analiza patrones de memoria

Usa herramientas de perfilado (ver abajo) para detectar qué objetos dominan el heap o qué referencias impiden al GC liberar memoria.

Herramientas profesionales

Herramienta	Función principal	Uso recomendado
VisualVM	Analiza uso de CPU y memoria, detecta fugas.	Ideal para perfiles iniciales y comparar versiones.
Eclipse Memory Analyzer (MAT)	Examina heap dumps para identificar objetos retenidos.	Para diagnósticos detallados en entornos de producción.
JConsole	Monitoriza en tiempo real el heap y los hilos activos.	Perfecta para ver comportamiento del GC mientras corre el programa.
YourKit / IntelliJ Profiler	Perfilado avanzado, identifica métodos que generan más objetos.	En aplicaciones grandes o microservicios.

- ☐ **Consejo profesional:** Configura la JVM con el parámetro -XX:+PrintGCDetails para observar en consola cada ciclo del Garbage Collector. Entender cuándo y por qué se activa es una forma directa de mejorar el rendimiento.

Mitos y Realidades

 **Mito 1:** "El Garbage Collector elimina todos los problemas de memoria."

→ **FALSO.** El GC solo limpia objetos sin referencia. Si una estructura mantiene una referencia activa aunque ya no se use (por ejemplo, listas que nunca se vacían), el GC no podrá eliminarla, generando fugas de memoria lógicas (logical leaks).

 **Mito 2:** "Como Java gestiona la memoria, no necesito preocuparme por optimizarla."

→ **FALSO.** Java automatiza, pero no adivina. Un diseño ineficiente (crear objetos en exceso, mantener colecciones enormes, abusar de strings) satura el heap y ralentiza el GC. La memoria se comporta como un recurso físico limitado, no como un espacio infinito.

 **Realidad**

La gestión inteligente consiste en colaborar con la JVM, no en ignorarla. El mejor código no es el que crea menos objetos, sino el que crea los necesarios en el momento correcto y los libera a tiempo.

Resumen Final (para examen)

- **Stack:** almacena variables locales y llamadas a métodos. Rápido, se libera automáticamente.
- **Heap:** almacena objetos dinámicos. Es más grande y más lento.
- **Garbage Collector:** elimina objetos sin referencias, liberando memoria en el heap.
- **Buenas prácticas:**
 - Evitar objetos innecesarios en bucles.
 - Usar StringBuilder para concatenar.
 - Monitorear memoria con herramientas.
 - Liberar referencias cuando no se necesiten.
- **Optimización real:** reducir la carga del GC mediante diseño eficiente.

Sesión 8: Conversiones implícitas y explícitas seguras

Fundamentos: cómo Java transforma tipos de forma automática o controlada

En el corazón del lenguaje Java está la seguridad de tipos: cada variable y operación está vinculada a un tipo de dato definido, lo que evita gran parte de los errores en tiempo de ejecución. Sin embargo, en muchos programas es necesario convertir valores de un tipo a otro. Por ejemplo, pasar de int a double, o de double a int. Este proceso se llama casting o conversión de tipos.

Java distingue claramente entre conversiones implícitas (widening) y conversiones explícitas (narrowing). Entender la diferencia entre ambas es fundamental para evitar pérdidas de precisión, truncamientos o comportamientos inesperados.

Conversiones implícitas (automáticas o widening)

Las realiza automáticamente el compilador cuando el nuevo tipo puede contener sin riesgo al anterior. Se denominan widening conversions porque amplían la capacidad de representación.

Ejemplo:

```
int a = 10;  
double b = a; // conversión implícita de int → double
```

En este caso, no hay pérdida de información: un double puede representar cualquier valor entero.

Regla general: byte → short → int → long → float → double

El flujo va de tipos más pequeños a más grandes (en capacidad o precisión).

Conversiones explícitas (manuales o narrowing)

Ocurren cuando el programador obliga al compilador a convertir un valor a un tipo más pequeño o incompatible. Se requiere especificar la conversión con paréntesis.

Ejemplo:

```
double x = 3.9;  
int y = (int) x; // conversión explícita: pierde los decimales
```

Aquí, Java trunca (no redondea) el valor, generando una posible pérdida de datos.

Por qué importa la conversión segura

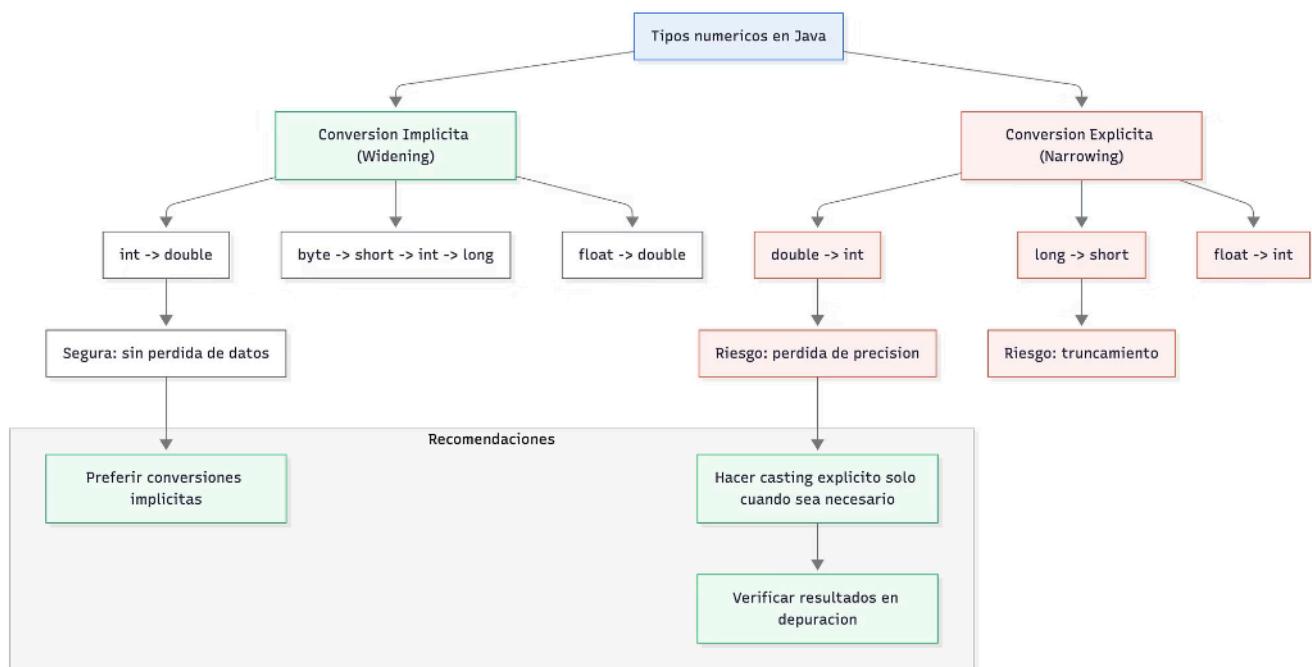
Las conversiones parecen inocentes, pero en sistemas reales —especialmente en aplicaciones financieras, científicas o de IoT— pueden provocar errores graves:

- Redondeos inesperados.
- Desbordamientos numéricos.
- Pérdida de precisión en acumulaciones o promedios.
- Errores de tipo en librerías que esperan datos concretos (float vs double).

Un ejemplo clásico de error real fue el "bug del Ariane 5", en el que una conversión incorrecta entre tipos de número provocó el fallo de un cohete en 1996. Aunque aquel sistema no era Java, el principio es el mismo: una conversión mal gestionada puede ser catastrófica.

Conclusión: en Java, las conversiones son potentes, pero debes usarlas con criterio. Si el compilador no lo hace automáticamente, pregúntate por qué antes de forzar el casting.

Esquema Visual: mapa de conversiones en Java



Explicación del diagrama:

- Las conversiones implícitas (widening) se realizan automáticamente y son seguras: no pierden datos.
- Las explícitas (narrowing) requieren un casting manual porque pueden perder información o truncar valores.
- El bloque inferior resume la regla de oro profesional: "prefiere siempre el camino seguro, y cuando tengas que forzar, valida los resultados".

Inicio

Buscar

Tu biblioteca

Crear playlist

Canciones que te g...

Pop

¿QUÉ QUIERES ESCUCHAR? ¡DALE AL PLAY!

PATROCINADO

Escúchalo en Spotify

Caso de Estudio: Spotify y la precisión del sonido

Contexto

Spotify procesa millones de flujos de audio cada segundo. Cada canción se traduce internamente en secuencias numéricas de amplitud, muestreo y volumen. Estos valores deben ser precisos, ya que pequeñas variaciones se traducen en distorsiones o pérdidas de calidad.

Estrategia

El equipo de ingeniería detectó que algunas rutinas de procesamiento usaban conversiones implícitas de float a int al calcular promedios y umbrales. En audio, pasar de un número decimal a entero implica truncar los decimales, lo que altera la amplitud real del sonido.

Solución aplicada:

- Uniformizaron los tipos de datos: todo el procesamiento intermedio se realiza en double para conservar precisión.
- Conversiones controladas: las transformaciones a int se realizan solo en el paso final de exportación, mediante cast explícito controlado y redondeo:

```
int sample = (int) Math.round(amplitude * 32767);
```

- Automatización de validaciones: añadieron verificaciones automáticas en los tests unitarios para detectar conversiones arriesgadas (float → int, double → float).

Resultado

- Eliminación de errores de truncamiento en cálculos intermedios.
- Mejora de la fidelidad del sonido hasta un **2% en precisión RMS**.
- Código más legible y mantenible al tener reglas claras de conversión.

- Conclusión profesional:** en entornos que manejan datos numéricos, las conversiones deben planificarse. No basta con que el código compile; debe mantener la exactitud de la información.

Herramientas y Consejos Profesionales

Consejos prácticos



Prefiere conversiones implícitas

Si Java puede hacer la conversión automáticamente, confía en ella. No forces un casting innecesario, ya que puede ocultar errores.



Usa castings explícitos solo cuando entiendas las consecuencias

Cuando conviertas de double a int, asume que los decimales se pierden. Si necesitas redondear, hazlo con Math.round() o Math.floor().



Elige tipos grandes para operaciones acumulativas

Si estás sumando muchos int, usa long o double para evitar desbordamientos.

```
long total = 0;  
for (int i : valores) total += i;
```



Valida conversiones en tiempo de ejecución

Usa logs o condiciones para detectar posibles pérdidas:

```
double d = 1e10;  
int i = (int) d;  
if (i != d) System.out.println("Atención:  
pérdida de precisión");
```



Evita conversiones encadenadas

Convertir float → int → byte multiplica el riesgo. Hazlo en un solo paso controlado o reestructura la lógica.



Cuidado con los literales

Recuerda que los literales numéricos sin sufijo son int o double por defecto. Si necesitas long, añade L:

```
long x = 10000000000L; // no 'int', sino  
'long'
```

Herramientas útiles

Herramienta	Función principal	Uso recomendado
SonarLint / SonarQube	Analiza código estático; detecta castings inseguros y pérdidas de precisión.	En proyectos profesionales y CI/CD.
IntelliJ Code Analyzer	Señala conversiones innecesarias o duplicadas.	Durante desarrollo en IDE.
Online Java Compiler (JDoodle / Repl.it)	Permite probar castings y conversiones rápidamente.	Para práctica o prototipado.
FindBugs / SpotBugs	Analiza bytecode y detecta errores ocultos relacionados con tipos.	Ideal para auditorías de código.

- ☐ **Tip avanzado:** activa el warning level alto en tu IDE para que señale castings innecesarios o riesgosos; muchas veces el propio compilador ya detecta situaciones de narrowing potencialmente inseguras.



Mitos y Realidades

 Mito 1: "Si el programa compila, la conversión es segura."

→ **FALSO.** Compilar solo garantiza que la conversión es sintácticamente válida, no que sea lógicamente correcta. Por ejemplo, (int) 3.9 compila, pero trunca el decimal. La seguridad real depende de entender qué se pierde o altera.

 Mito 2: "Todas las conversiones numéricas se comportan igual."

→ **FALSO.** Las conversiones entre float, double y int pueden comportarse distinto según el rango de valores. Por ejemplo, convertir 1e20 (un número muy grande) a int producirá overflow. Cada tipo tiene límites físicos de representación (Integer.MAX_VALUE, Double.MAX_VALUE) que deben conocerse.

 Realidad

El compilador es tu primera línea de defensa, pero la responsabilidad final es del desarrollador. Entender el modelo de datos y sus límites te permite evitar errores sutiles y mejorar la precisión del software.

Resumen Final (para examen)

- **Conversiones implícitas (widening):** automáticas, seguras. Ejemplo: int → double.
- **Conversiones explícitas (narrowing):** manuales con (tipo), pueden perder precisión. Ejemplo: double → int.
- **Regla general:** solo forzar casting si es estrictamente necesario.
- **Pérdidas típicas:** truncamiento, redondeo, overflow.
- **Herramientas recomendadas:** SonarLint, IntelliJ Analyzer, VisualVM.
- **Principio clave:** comprueba, valida y documenta cada conversión relevante.