
**UNIVERSIDAD NACIONAL POLITECNICA**

**DIRECCIÓN DE CIENCIAS BÁSICAS Y TECNOLOGÍA**

**CARRERA DE INGENIERÍA EN SISTEMAS DE INFORMACIÓN**

**DESARROLLO DE APLICACIONES III**

---

# Conceptos de datos enumerados **(enum)**, formato de datos, números aleatorios y recursividad en el lenguaje de programación **JAVA**

---

**Elaborado por:**

Br. Walter Alexei Amador Jirón.

Mtro. Lesbia Elena Valerio Lacayo.

Managua-Nicaragua

Octubre, 2025

## Introducción

La programación en Java requiere el dominio de conceptos fundamentales que permiten desarrollar aplicaciones eficientes, mantenibles y escalables. En este documento estaremos hablando sobre los datos enumerados (enum), formato de datos, números aleatorios y recursividad ya que estos modulos son importantes a la hora de desarrollar aplicaciones y entender los conceptos del lenguaje.

# I) Datos Enumerados (enum)

Un **dato enumerado (enum)** en Java es un tipo de dato especial que permite definir un conjunto **limitado, constante y predefinido** de valores bajo un mismo tipo. Su finalidad es representar de forma semántica un grupo fijo de elementos relacionados, garantizando que una variable solo pueda adoptar uno de esos valores válidos.

El uso de enum proporciona **seguridad de tipo** (type-safety), lo que significa que el compilador puede detectar errores en tiempo de compilación si se intenta asignar un valor no permitido. Este enfoque evita el uso de literales sin contexto (como cadenas o números arbitrarios) que pueden provocar inconsistencias o errores lógicos en el programa.

**Características principales**

1. Conjunto finito de valores: los enumerados se utilizan cuando se conocen todos los posibles valores en tiempo de compilación. Esto evita el uso de constantes dispersas en el código y mejora la organización.
2. Definición de constantes simbólicas: cada elemento del enum es una constante, generalmente representada con letras mayúsculas, siguiendo las convenciones de nomenclatura de Java para constantes.
3. Tipo seguro y fuertemente tipado: a diferencia de las constantes definidas con static final, los enumerados no se tratan como simples valores literales, sino como **instancias únicas** de un tipo de dato definido.

4. Herencia implícita: todos los enumerados heredan de la clase base java.lang.Enum, lo que les otorga propiedades y métodos comunes. No pueden

heredar de otras clases, pero sí pueden implementar interfaces, lo que amplía su funcionalidad.

5. Estructura de clase interna: aunque se perciben como una lista de valores, internamente los enum son **clases completas**. Esto significa que pueden incluir:
    - *Atributos (campos):* para asociar información a cada valor.
    - *Constructores:* que inicializan los atributos (siempre privados o con acceso restringido).
    - *Métodos:* tanto comunes como específicos para cada valor enumerado.
6. Inmutabilidad: los valores definidos dentro de un enum son constantes y, por lo tanto, no pueden modificarse durante la ejecución del programa.

## Función y utilidad

El propósito principal de los enumerados es representar conceptos que poseen un conjunto fijo de opciones dentro del dominio del problema.
Ejemplos comunes incluyen:

- ✓ Días de la semana.
- ✓ Estados de un proceso.
- ✓ Direcciones cardinales.
- ✓ Niveles de prioridad.
- ✓ Tipos de usuario o roles en un sistema.

De este modo, el uso de enum mejora la claridad semántica del código, ya que los valores adquieren significado propio, en lugar de ser meras etiquetas numéricas o textuales.

## Ventajas conceptuales

1. Claridad semántica: cada valor enumerado representa una entidad con nombre y propósito específico.
2. Mantenimiento y escalabilidad: cualquier modificación o ampliación del conjunto de valores se realiza en un único lugar.
3. Compatibilidad con estructuras de control: los enum pueden integrarse directamente en sentencias condicionales como switch, facilitando la toma de decisiones según el valor de la enumeración.

4. Compatibilidad con colecciones especializadas: Java proporciona estructuras optimizadas como EnumSet y EnumMap para trabajar con enumeraciones de manera más eficiente.
5. Evita valores inválidos: se garantiza que una variable de tipo enum solo contendrá un valor válido del conjunto definido.

## Métodos predefinidos

Dado que todos los enumerados heredan de java.lang.Enum, disponen de una serie de métodos útiles integrados, entre los que destacan:

- values(): devuelve un arreglo con todos los valores definidos en el enum.
- valueOf(String name): permite obtener una constante a partir de su nombre textual.
- ordinal(): devuelve la posición que ocupa cada valor dentro de la declaración del enum.

Estos métodos facilitan la manipulación y recorrido de los valores definidos, lo que resulta útil en procesos de comparación o validación.

## Restricciones

- Los enum no pueden ser instanciados directamente por el programador, ya que sus instancias se crean automáticamente por el sistema en el momento de la inicialización de la clase.
- No pueden heredar de otras clases, aunque sí pueden implementar interfaces.
- Los constructores de los enum siempre son privados o con acceso restringido, para evitar que se creen nuevas instancias fuera de la definición del tipo enumerado.

## Buenas prácticas

A) Utilizar nombres descriptivos y en mayúsculas para las constantes.
B) Evitar que los enumerados representen conjuntos que puedan cambiar dinámicamente (por ejemplo, registros de una base de datos).

C) Mantener la lógica interna del enum coherente con su propósito: su función principal es representar un conjunto de valores fijos, no reemplazar clases con comportamientos complejos.
D) Documentar adecuadamente cada constante si su función no es evidente a simple vista.

*Ejemplo:*

# II) Formato de Datos

El **formato de datos** en Java se refiere al proceso de **convertir valores internos (numéricos, cadenas, fechas, objetos, etc.) en representaciones de texto estructuradas y legibles**, y viceversa en algunos casos. El formateo es crucial para presentar información al usuario, generar informes, logs, tablas alineadas, exportar datos y para la interoperabilidad entre sistemas que esperan textos con un formato determinado.

En Java existen dos enfoques principales para formateo textual: (1) **formatos basados en especificadores de formato estilo printf** (con clases/funciones que siguen la sintaxis de Formatter), y (2) **formatos basados en patrones** (clases como DecimalFormat o DateTimeFormatter que usan patrones específicos para números y fechas). Ambos enfoques se complementan y sirven fines distintos.

**Componentes fundamentales del formateo estilo printf**

La familia de formateo estilo printf (incluye Formatter, System.out.printf, String.format) trabaja con **especificadores** que indican cómo transformar un argumento en texto. Cada especificador tiene una sintaxis compuesta por varias partes que controlan el aspecto final:

- **Inicio:** el carácter % marca el inicio del especificador.
- **Flags (banderas):** modificadores opcionales que alteran alineación, signo, relleno, agrupamiento, etc. Son ajustes de presentación (por ejemplo: alineación izquierda, forzar signo positivo, rellenar con ceros).
- **Ancho de campo (width):** tamaño mínimo del campo. Si el valor es más corto, se añade relleno; si es mayor, el campo se expande.
- **Precisión (.precision):** controla aspectos como número de decimales para flotantes o número máximo de caracteres para cadenas.
- **Conversión (conversion):** especifica el tipo de representación (enteros, punto flotante, cadena, caracteres, booleanos, fechas, etc.).

Este esquema permite una gran flexibilidad: alineación, padding, control de decimales, notación científica, uso de argumentos por índice, y opciones de localización.

**Tipos de conversiones y su propósito conceptual**

Las conversiones delimitan **cómo se interpreta** el valor pasado y **cómo debe renderizarse**. Existen conversiones generales para objetos y cadenas, conversiones numéricas (enteros y flotantes), conversiones para caracteres y booleanos, y conversiones especiales para fechas y tiempos. Estos grupos permiten adaptar la salida al tipo de dato subyacente y al contexto semántico del valor.

**Flags y su significado semántico**

Las banderas no alteran el valor, sólo su presentación. Entre sus funcionalidades conceptuales están:

- **Alineación:** izquierda o derecha dentro del ancho de campo.
- **Signo:** mostrar siempre el signo, mostrar espacios para valores positivos, o usar paréntesis para negativos (en mecanismos específicos).
- **Relleno:** elegir carácter de relleno (por ejemplo, cero) para campos numéricos.
- **Agrupamiento:** insertar separadores de miles según convenciones locales.
- **Formateo alternativo:** cambiar forma de salida en ciertas conversiones (por ejemplo, prefijos para bases numéricas).

**Ancho, precisión y efectos en la representación**

- **Ancho (width):** controla la presentación tabular y la alineación visual. Es fundamental en salidas de consola, reportes y cuando se requiere formato columnar.
- **Precisión:** para números de punto flotante controla la cantidad de dígitos fraccionarios; para cadenas limita la longitud mostrada. La precisión es crítica para el control de redondeo y la compactación de salidas.

Estos parámetros, en combinación con las banderas, permiten un control fino sobre la estética y la exactitud numérica de la salida.

**Localización (Locale) y su importancia**

El formato tiene que considerar **reglas culturales y lingüísticas**: separadores decimales, separadores de miles, orden fecha/mes/año, símbolos de moneda, formato de signos, y nombres de mes/día. Java permite especificar un Locale al formatear, para que la presentación respete convenciones regionales. La localización es esencial cuando el software se entrega a usuarios en múltiples países o cuando se genera información destinada a cumplir normas regionales de presentación.

**Patrones de formateo (DecimalFormat, DateTimeFormatter) — enfoque basado en patrones**

Para números y fechas existe un enfoque alternativo en el que se definen **patrones** (cadena descriptiva que indica estructura: dígitos significativos, agrupamiento, símbolo de moneda, formato de era, zona horaria, etc.). Este enfoque es más expresivo para casos complejos de formato (por ejemplo, mostrar moneda con símbolo local, usar formatos ISO o formatos amigables para usuarios) y suele proporcionar más control semántico que los especificadores individuales.

### Formateo de fechas y tiempos

El formateo de fechas y tiempos merece tratamiento propio: existen formatos basados en patrones (patrones de año, mes, día, hora, minuto, segundo, zona horaria, etc.) y estándares (por ejemplo, representaciones ISO). Desde Java 8, las clases del paquete de fecha/tiempo moderno proporcionan herramientas seguras y manejables para formateo localizado y thread-safe, a diferencia de las APIs legacy.

### Rounding (redondeo), precisión numérica y BigDecimal

Cuando se formatean números, la forma en que se maneja el redondeo y la precisión es crítica, sobre todo en dominios financieros. El uso de tipos numéricos de alta precisión y la especificación explícita del modo de redondeo (por ejemplo, redondeo al par más próximo o hacia arriba) evita errores de representación y acumulación. En Java, las clases de formato permiten controlar la precisión mostrada, pero la precisión aritmética debe gestionarse a nivel del tipo numérico (por ejemplo, BigDecimal) antes de formatear.

### Parsing vs Formato: bidireccionalidad

Algunas clases de formateo también admiten **parseo**, es decir, convertir texto formateado de vuelta a un objeto (números, fechas). El parseo requiere conocer exactamente el patrón o el Locale empleado y suele ser más frágil ante entradas inesperadas; por eso, la robustez del parseo suele implicar validaciones y manejo de excepciones.

### Excepciones y validación

El uso incorrecto de especificadores o tipos incompatibles provoca excepciones de formato en tiempo de ejecución. Por ello, es importante validar las plantillas de formato y los tipos de argumentos, y manejar las excepciones relacionadas para evitar fallos inesperados en la presentación.

### Rendimiento y consideraciones de concurrencia

- **Rendimiento:** la creación repetida de objetos de formateo puede ser costosa en entornos de alta demanda. Se suele recomendar reutilizar instancias de formateadores cuando sea seguro hacerlo.
- **Thread-safety:** algunas APIs de formateo no son seguras para uso concurrente; otras, especialmente las introducidas en versiones modernas, ofrecen garantías de inmutabilidad y son seguras para threads. Es imprescindible conocer la semántica de la clase de formateo que se emplea para evitar condiciones de carrera.

**Buenas prácticas conceptuales**

- Elegir el mecanismo de formateo adecuado según el dominio: printf-style para salidas tabulares o cortas; patrones y formateadores especializados para números financieros y fechas.
- Siempre considerar Locale al formatear información destinada a usuarios finales.
- Controlar explícitamente la precisión y el modo de redondeo antes de formatear valores monetarios.
- Mantener las plantillas de formato centralizadas y documentadas para facilitar mantenimiento y traducción.
- Preferir APIs modernas e inmutables (cuando existan) para evitar problemas de concurrencia.
- Separar la lógica de negocio del formateo: calcular y preparar datos antes de aplicar cualquier representación textual.

*Ejemplo:*

# III) Números Aleatorios

Los **números aleatorios** en Java representan valores generados por un algoritmo que intenta imitar la imprevisibilidad de los fenómenos naturales. Aunque se les denomina "aleatorios", en la mayoría de los casos son el resultado de un proceso **determinístico** controlado por un algoritmo matemático, lo que los clasifica como números pseudoaleatorios (PRNG, Pseudo-Random Number Generator).

En términos computacionales, la verdadera aleatoriedad es difícil de lograr, ya que los sistemas digitales funcionan bajo reglas lógicas predecibles. Por ello, Java como la mayoría de los lenguajes implementa generadores pseudoaleatorios que, partiendo de una **semilla (seed)** inicial, producen una secuencia de números que cumplen con las propiedades estadísticas de aleatoriedad, aunque sean reproducibles.

**Naturaleza de los números pseudoaleatorios**

Un generador pseudoaleatorio utiliza una fórmula matemática para transformar una semilla inicial en una secuencia de valores aparentemente impredecibles. Esta secuencia es determinística, lo que significa que, si se conoce la semilla y el algoritmo, se puede reproducir exactamente la misma serie de números.

A pesar de ello, los PRNG son útiles en gran variedad de contextos donde la imprevisibilidad absoluta no es un requisito, tales como simulaciones, pruebas estadísticas, juegos o generación de datos de prueba.

**Clases principales en Java**

1. *java.util.Random*
   - ✓ Esta clase forma parte de la biblioteca estándar de Java y se emplea para generar números pseudoaleatorios basados en un algoritmo determinístico. Utiliza una semilla de 48 bits, la cual se transforma mediante una función lineal congruencial (Linear Congruential Generator, LCG).
   - ✓ El comportamiento de Random es repetible: al proporcionar la misma semilla, se obtiene exactamente la misma secuencia de valores. Esta característica es útil en pruebas, depuración o entornos donde se requiere control sobre los resultados. Sin embargo, su predictibilidad la hace inadecuada para aplicaciones que manejan datos sensibles o requieren confidencialidad.

2. *java.security.SecureRandom*

- Esta clase fue introducida para resolver las limitaciones de seguridad de Random. A diferencia de un generador determinístico, SecureRandom obtiene su entropía (semilla o fuente de aleatoriedad) de **fuentes físicas o del sistema operativo**, como movimientos del ratón, tiempos de respuesta del hardware o ruido del sistema.
- El resultado es un conjunto de números **criptográficamente seguros**, imposibles de predecir incluso si se conoce parte de la secuencia o del estado interno del generador. Su diseño cumple con estándares de seguridad como **FIPS 140-2**, utilizados en sistemas criptográficos y financieros.
- Por su naturaleza no determinística, SecureRandom es más lento, pero **idóneo para operaciones críticas**, como generación de contraseñas, llaves de cifrado, tokens de sesión o identificadores únicos no predecibles.

**Diferencias conceptuales entre Random y SecureRandom**

| Aspecto | Random | SecureRandom |
|---|---|---|
| **Tipo de generador** | Pseudoaleatorio (determinístico) | Criptográficamente seguro (no determinístico) |
| **Tamaño de semilla** | 48 bits | Hasta 128 bits o más |
| **Reproducibilidad** | Reproducible si se conoce la semilla | No reproducible |
| **Uso recomendado** | Simulaciones, juegos, pruebas | Seguridad, autenticación, criptografía |
| **Velocidad** | Más rápido | Más lento, pero más seguro |
| **Fuente de entropía** | Algoritmo matemático fijo | Entropía del sistema o hardware |

*Tabla1: Comparación entre Random y SecureRandom*

**Semilla y determinismo**

La **semilla (seed)** es el valor inicial a partir del cual el generador produce la secuencia de números. Su elección influye directamente en la aleatoriedad y reproducibilidad del proceso. En los generadores pseudoaleatorios, cambiar la semilla implica generar una secuencia distinta, pero si se reutiliza la misma semilla, la secuencia será idéntica.

En el contexto de seguridad, esto constituye una vulnerabilidad, ya que un atacante podría inferir o replicar la secuencia si logra determinar la semilla. Por esta razón, en aplicaciones criptográficas se evita el uso de semillas predecibles y se recurre a mecanismos que garantizan la entropía real del sistema.

### Aleatoriedad criptográfica

En el ámbito de la seguridad informática, la aleatoriedad cumple un papel fundamental. Muchos sistemas de cifrado, autenticación y firma digital dependen de valores impredecibles para mantener la integridad de las comunicaciones.

Por ello, **SecureRandom** está diseñado para generar números que no solo parecen aleatorios, sino que "resisten ataques estadísticos y predictivos". Su algoritmo puede utilizar fuentes de entropía reales y mezclar diferentes entradas para crear un estado interno altamente complejo e irreproducible.

De este modo, SecureRandom se considera un **CSPRNG** (Cryptographically Secure Pseudo-Random Number Generator), cumpliendo con los criterios de resistencia ante análisis inverso y predictibilidad.

### Propiedades estadísticas deseables

Un buen generador de números aleatorios debe cumplir ciertas propiedades estadísticas para considerarse fiable:

1. Uniformidad: los números generados deben distribuirse de manera uniforme dentro del rango especificado.
2. Independencia: los valores consecutivos no deben mostrar correlación.
3. Repetición mínima: las secuencias deben ser lo suficientemente largas antes de repetirse.
4. Imprevisibilidad: en el contexto de seguridad, no debe ser posible inferir el siguiente valor con base en los anteriores.
5. Alta entropía: las semillas deben tener alta variabilidad y provenir de fuentes impredecibles.

### Consideraciones de rendimiento y precisión

El uso de generadores más complejos, como SecureRandom, conlleva un **costo en rendimiento** debido a la necesidad de recopilar entropía del sistema y realizar cálculos adicionales. Por tanto, la elección del generador depende del contexto: en tareas que priorizan velocidad sobre seguridad (como simulaciones o juegos), Random es suficiente; pero en contextos donde la **confidencialidad o autenticidad** es crítica, la única opción viable es SecureRandom.

Además, Java garantiza que los generadores mantengan **alta precisión estadística** incluso en aplicaciones concurrentes, aunque se recomienda crear instancias separadas por hilo o sincronizar el acceso en entornos multihilo.

**Fuentes de entropía del sistema**

En sistemas modernos, la entropía que alimenta a los generadores seguros proviene de diversos eventos y dispositivos físicos: tiempos de ejecución variables, latencia de red, ruido térmico, movimientos del ratón, entre otros. Estas fuentes son inherentemente impredecibles, lo que asegura que la secuencia resultante no pueda ser replicada ni estimada.

**Buenas prácticas conceptuales**

- ✓ Usar Random solo en contextos no críticos, como juegos, algoritmos de simulación o tareas estadísticas simples.
- ✓ Emplear SecureRandom en todo proceso que involucre datos confidenciales o autenticación.
- ✓ Evitar el uso de semillas fijas en sistemas que deban producir secuencias únicas o impredecibles.
- ✓ Considerar la localización y el hardware, ya que algunos entornos pueden limitar la disponibilidad de fuentes de entropía.
- ✓ Revisar las propiedades estadísticas de los generadores cuando se utilizan para simulaciones científicas o análisis probabilísticos.
- ✓ Controlar la concurrencia, creando instancias separadas por hilo para evitar bloqueos o resultados inconsistentes.

*Ejemplo:*

# IV) Recursividad

La **recursividad** es una técnica fundamental en la programación que consiste en la capacidad de un método para llamarse a sí mismo directa o indirectamente con el fin de resolver un problema a través de su división en subproblemas más simples. En Java, la recursividad se implementa mediante la invocación de un método dentro de su propio cuerpo, lo que permite expresar soluciones de manera más natural, especialmente en problemas que presentan una estructura jerárquica o repetitiva.

Desde un punto de vista conceptual, la recursividad se basa en el principio de **autorreferencia,** un problema se resuelve definiéndose en función de sí mismo, pero en una versión más pequeña o reducida. Este proceso continúa hasta alcanzar una **condición base**, que representa el caso más simple y que no requiere más llamadas recursivas. A partir de ese punto, las llamadas pendientes comienzan a resolverse en orden inverso, reconstruyendo la solución global.

En Java, cada llamada recursiva ocupa un espacio en la **pila de llamadas (call stack)**, que almacena las variables locales, parámetros y dirección de retorno. Esto significa que la recursión implica un consumo de memoria proporcional al número de invocaciones activas. Si la recursión no posee una condición de parada válida o si el número de llamadas es excesivo, se produce un error de tipo **StackOverflowError**, lo que interrumpe la ejecución del programa.

## Estructura de un método recursivo

Un método recursivo en Java se compone de dos elementos esenciales:

1. **Caso base:** define el punto en el que la recursión se detiene. Sin esta condición, el método se llamaría indefinidamente a sí mismo, causando un desbordamiento de pila.
2. **Llamada recursiva:** representa el paso donde el método se invoca nuevamente con un problema reducido o simplificado, acercándose progresivamente al caso base.

Estos dos componentes garantizan que el proceso recursivo sea **correcto, finito y convergente** hacia la solución.

## Casos de uso comunes de la recursividad

La recursividad es especialmente útil en problemas donde los datos o las operaciones tienen una **estructura jerárquica o repetitiva**, y donde la solución depende de la resolución de subproblemas similares. Algunos casos de uso frecuentes incluyen:

- Cálculos matemáticos: como el factorial de un número, la serie de Fibonacci o las potencias.
- Estructuras de datos jerárquicas: recorrido de árboles binarios, directorios, o grafos.
- Algoritmos de búsqueda y ordenamiento: como mergesort, quicksort, y búsqueda binaria.
- Procesamiento de estructuras anidadas: análisis de expresiones aritméticas, generación de combinaciones y permutaciones.
- Resolución de problemas combinatorios: laberintos, backtracking o problemas de optimización como el de las N-reinas.

## Ventajas de la recursividad

1. **Claridad y legibilidad del código:** las soluciones recursivas suelen ser más fáciles de entender, ya que reflejan la definición matemática o lógica del problema.
2. **Simplificación de algoritmos complejos:** permite expresar soluciones que serían complicadas o extensas si se implementaran de forma iterativa.
3. **Ideal para estructuras jerárquicas:** se adapta naturalmente a problemas como el recorrido de árboles o el análisis de estructuras anidadas.
4. **Reducción de código repetitivo:** evita la necesidad de bucles anidados y estructuras de control complejas.

## Desventajas de la recursividad

1. **Mayor consumo de memoria:** cada llamada recursiva genera un nuevo marco en la pila, lo que puede saturar la memoria del sistema en casos de recursión profunda.

2. **Menor eficiencia en algunos casos:** las llamadas repetidas pueden generar cálculos redundantes si no se emplean técnicas de optimización (como la *memoización*).
3. **Riesgo de StackOverflowError:** si no existe un caso base correctamente definido o si la recursión es demasiado profunda.
4. **Dificultad para depurar:** seguir el flujo de ejecución de una recursión puede ser más complejo que en un algoritmo iterativo.

## Buenas prácticas en el uso de recursividad

1. **Definir claramente el caso base:** siempre debe existir una condición que detenga el proceso recursivo.
2. **Asegurar la reducción del problema:** cada llamada recursiva debe acercarse al caso base, evitando ciclos infinitos.
3. **Evitar llamadas innecesarias:** optimizar la lógica para minimizar el número de llamadas recursivas.
4. **Usar recursión con propósito:** solo aplicar la técnica cuando la naturaleza del problema lo justifique.
5. **Implementar memoización o almacenamiento intermedio:** en problemas donde se repiten los mismos cálculos (como Fibonacci), almacenar resultados intermedios mejora la eficiencia.
6. **Controlar la profundidad de recursión:** evitar estructuras o valores de entrada que generen un número excesivo de llamadas.
7. **Evaluar alternativas iterativas:** si el algoritmo puede expresarse con bucles sin perder claridad ni funcionalidad, la iteración suele ser más eficiente en términos de rendimiento.

## Consideraciones de diseño

En Java, la recursividad puede clasificarse en dos tipos principales:

1. **Recursividad directa:** cuando un método se invoca a sí mismo dentro de su propio cuerpo.
2. **Recursividad indirecta:** cuando un método A llama a otro método B, y este último vuelve a invocar al primero.

Asimismo, según la posición de la llamada, puede ser:

1. **Recursión de cola (tail recursion):** la llamada recursiva es la última instrucción del método. Aunque Java no optimiza este tipo de recursión automáticamente, en teoría permitiría reducir el consumo de memoria.
2. **Recursión no terminal:** se ejecutan operaciones adicionales después de la llamada recursiva, por lo que cada nivel debe conservar su contexto.

*Ejemplo:*

## V) Referencias

Baeldung. (2025). Java String.format(). https://www.baeldung.com/string-format

GeeksforGeeks. (2017). Random vs secure random numbers in Java. https://www.geeksforgeeks.org/java/random-vs-secure-random-numbers-java/

GeeksforGeeks. (2019). Recursion in Java. https://www.geeksforgeeks.org/java/recursion-in-java/

Oracle. (2006). Formatting numeric print output. Oracle Java Tutorials. https://docs.oracle.com/javase/tutorial/java/data/numberformat.html

Oracle. (2024). Enum types. Oracle Java Tutorials. https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

Oracle. (2025). SecureRandom (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html

Schildt, H. (2019). Java: The complete reference (11th ed.). McGraw-Hill Education.

The Server Side. (2025). Five examples of recursion in Java. https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/examples-Java-recursion-recursive-methods