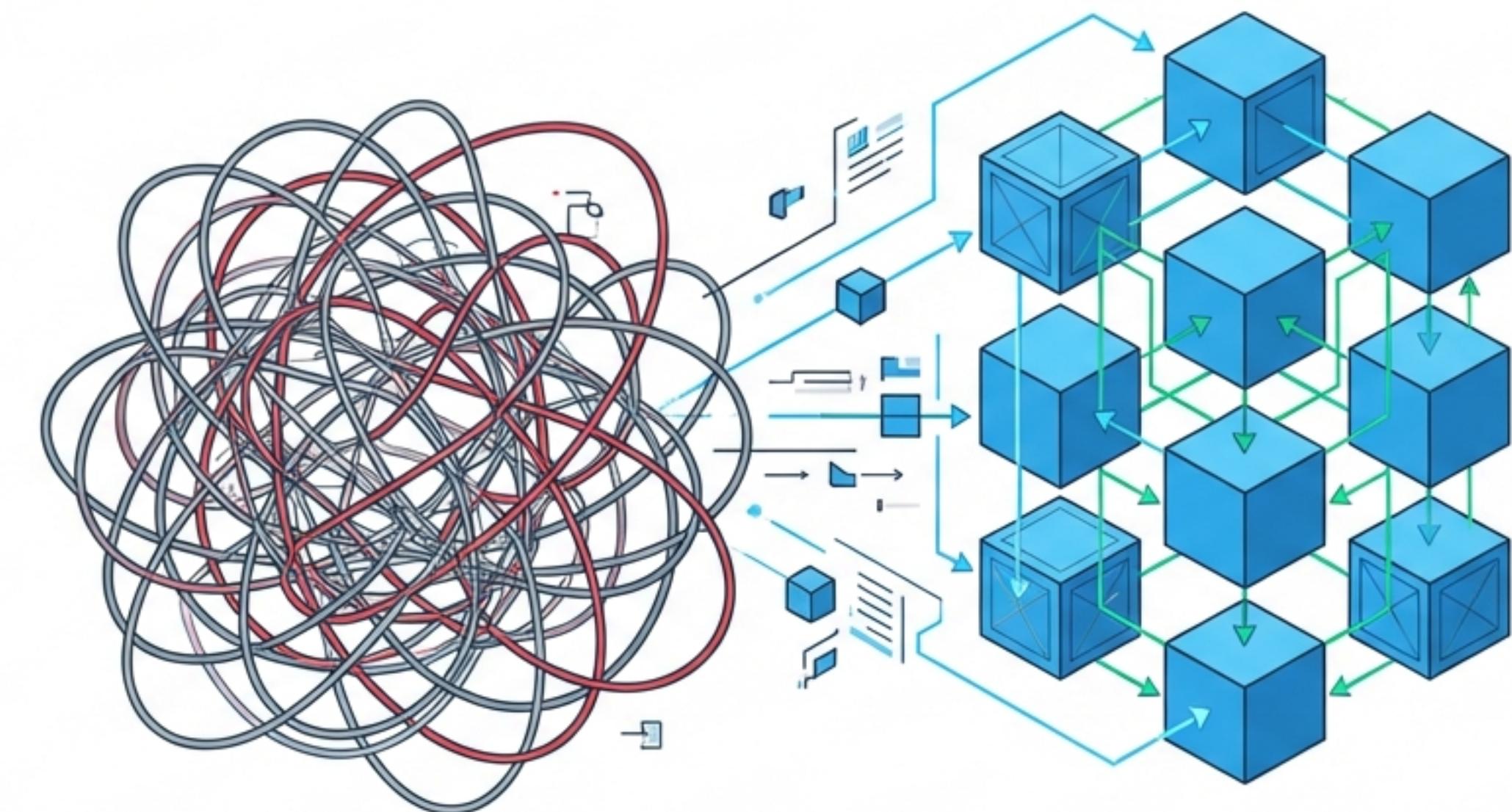


Divide y Vencerás: Dominando la Programación Modular

Estrategias para código eficiente,
mantenible y escalable.



Basado en el Tema 4: Programación Modular

La Trampa del Código Monolítico

```
function main() {
    const note = new Note();
    note.setTitle('NoteBrain');
    note.setBody('This is a note');
    note.setCategory('Work');

    const notes = [
        note,
        new Note({ title: 'Meeting', body: 'Meeting with team' })
    ];
    note.setNotes(notes);
}

main();

```

Programa Monolítico

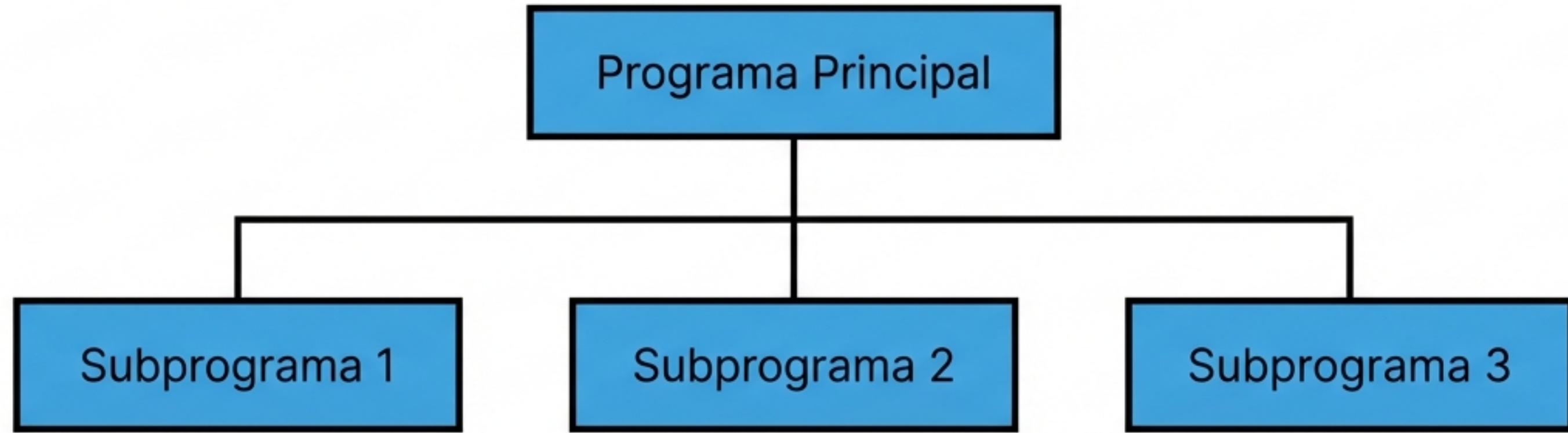
⚠ **Duplicación de código:** Repetir los mismos fragmentos dificulta la actualización.

⚠ **Escasa reutilización:** Imposible aprovechar funcionalidades en otros proyectos.

⚠ **Errores costosos:** Un fallo afecta a todo el sistema y es difícil de rastrear.

Un programa monolítico es difícil de depurar y costoso de mantener.

La Solución: Diseño Descendente (Top-Down)



Metodología

Divide el software en módulos independientes (Cajas Negras).

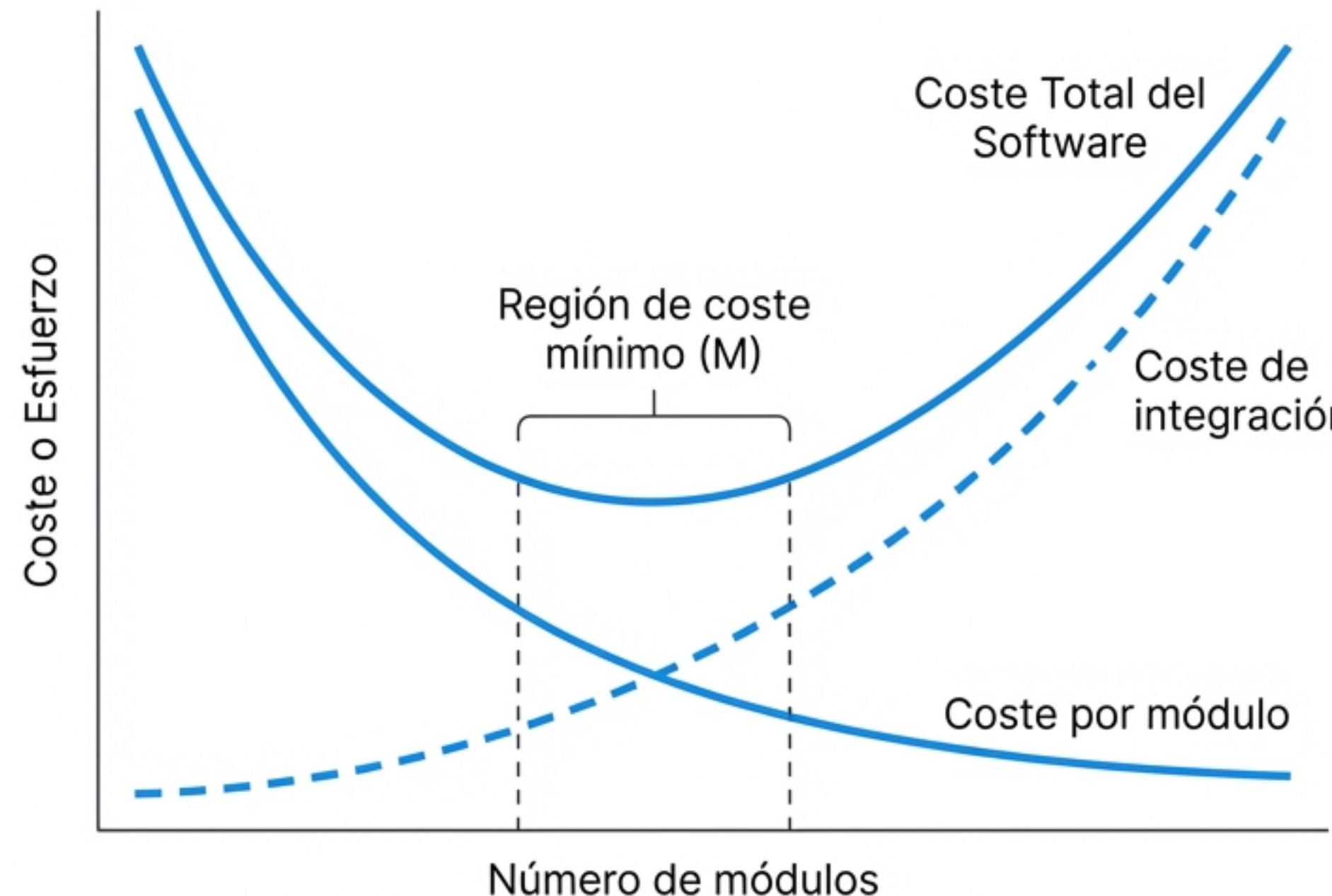
Entrada/Salida

Cada módulo recibe datos, procesa una tarea y devuelve un resultado.

Independencia

Desarrollo y pruebas por separado para cada componente.

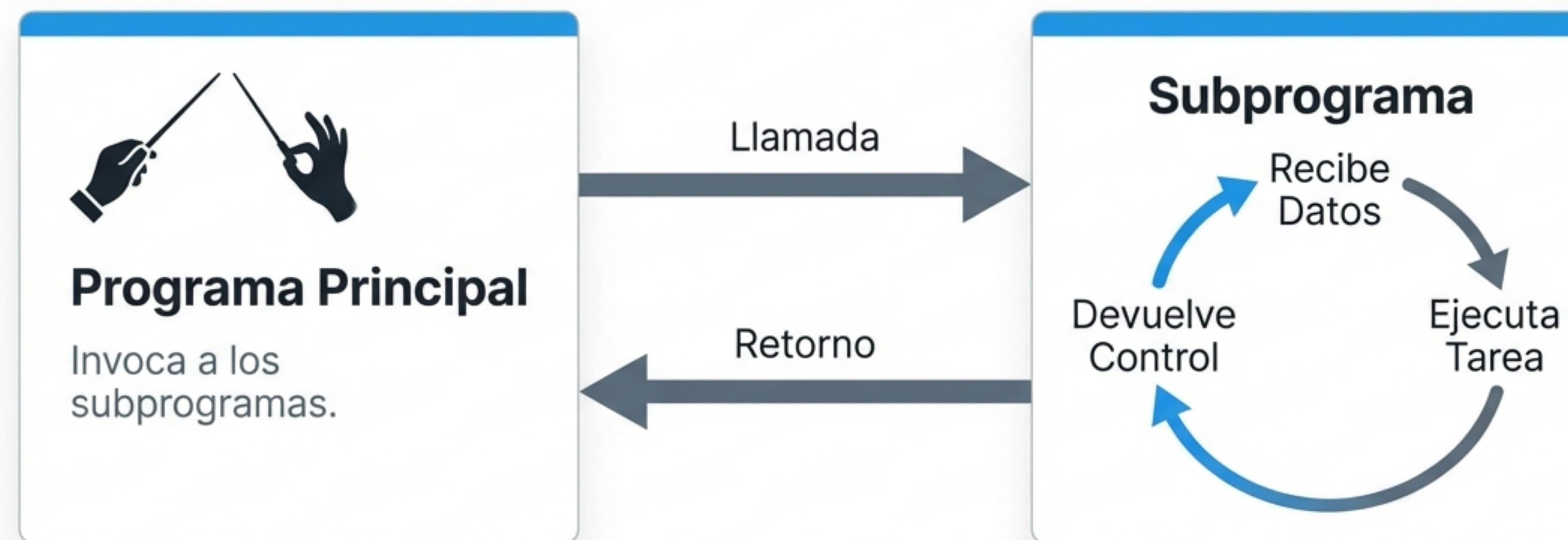
El Coste de la Modularización



Dividir el código reduce el coste por módulo, pero aumenta el coste de integración. El objetivo es encontrar el equilibrio.

Región de coste mínimo: El equilibrio entre tamaño y cantidad de módulos.

Arquitectura: Programa Principal y Subprogramas



Ubicación: Interno (mismo archivo) o Externo (bibliotecas enlazadas).

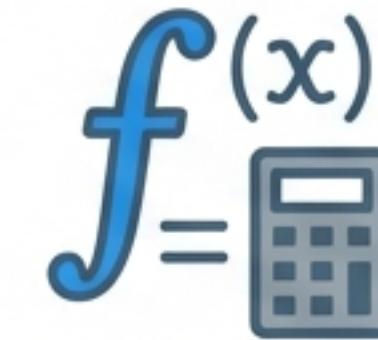
Los Bloques de Construcción: Procedimientos vs. Funciones



Procedimientos

Orientados a la Acción

- Realizan transformaciones (ej. mostrar datos, ordenar).
- No devuelven un valor asociado al nombre (`void`).
- Modifican el estado mediante parámetros.



Funciones

Orientadas al Cálculo

- Equivalente matemático: $Y = f(X)$.
- Devuelven un resultado explícito (`return`).
- Se definen especificando el tipo de dato.

Caso Práctico: 3, 2, 1... ¡Despegue!

Monolítico (Repetitivo)

```
for (int i = 3; i > 0; i--) { println(i); }  
println('¡Despegue!');  
  
for (int i = 3; i > 0; i--) { println(i); }  
println('¡Despegue!');  
  
for (int i = 3; i > 0; i--) { println(i); }  
println('¡Despegue!');  
  
// ... repetido 3 veces ...
```

Código Duplicado

Código Duplicado

Código Duplicado

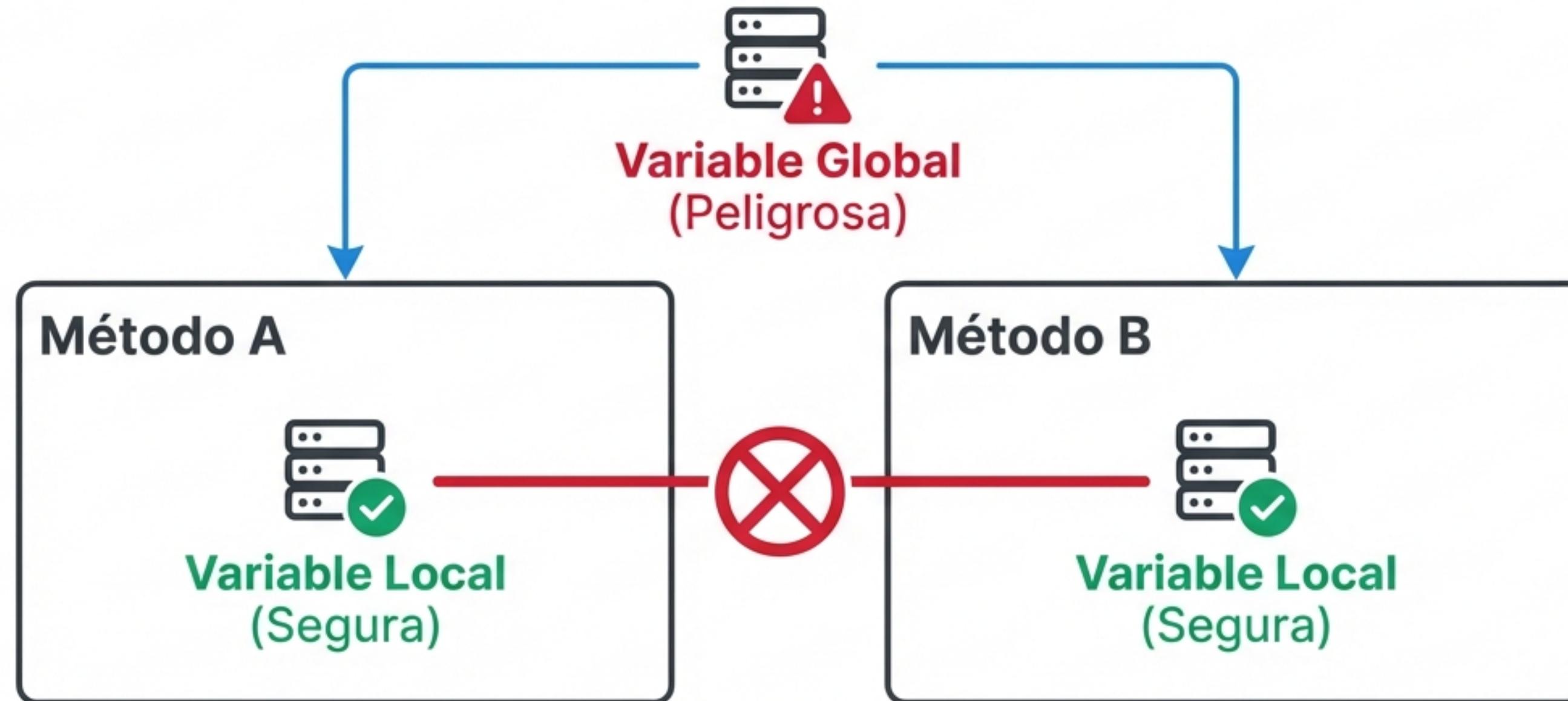
Modular (Reutilizable)

```
despegue();  
despegue();  
despegue();
```

```
void despegue() {  
    for (int i = 3; i > 0; i--) { println(i); }  
    println('¡Despegue!');  
}
```

Las Reglas de Visibilidad (Scope)

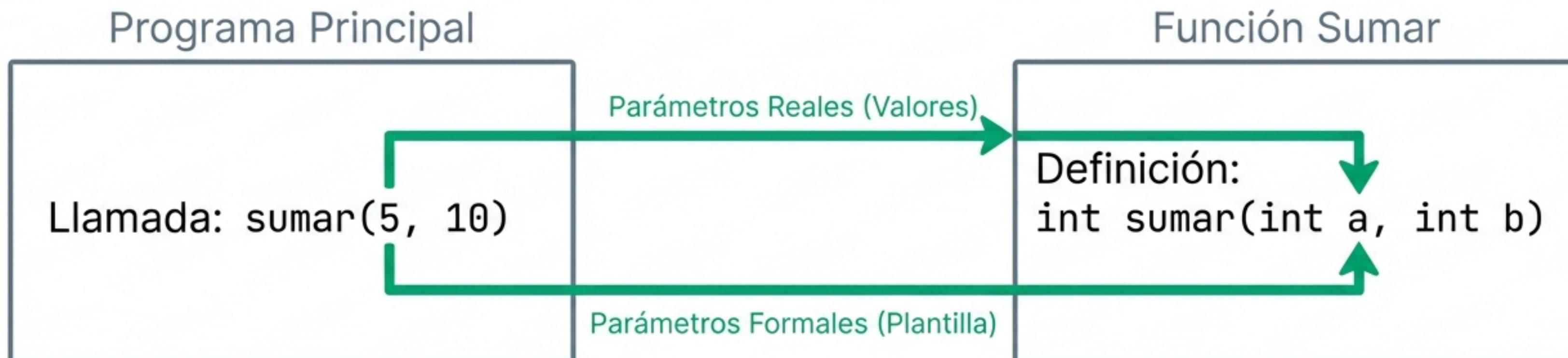
Ámbito Global / Clase



Variables Globales: Efectos secundarios y riesgo ✓ de errores.

Variables Locales: Se crean y destruyen automáticamente, sin interferencias.

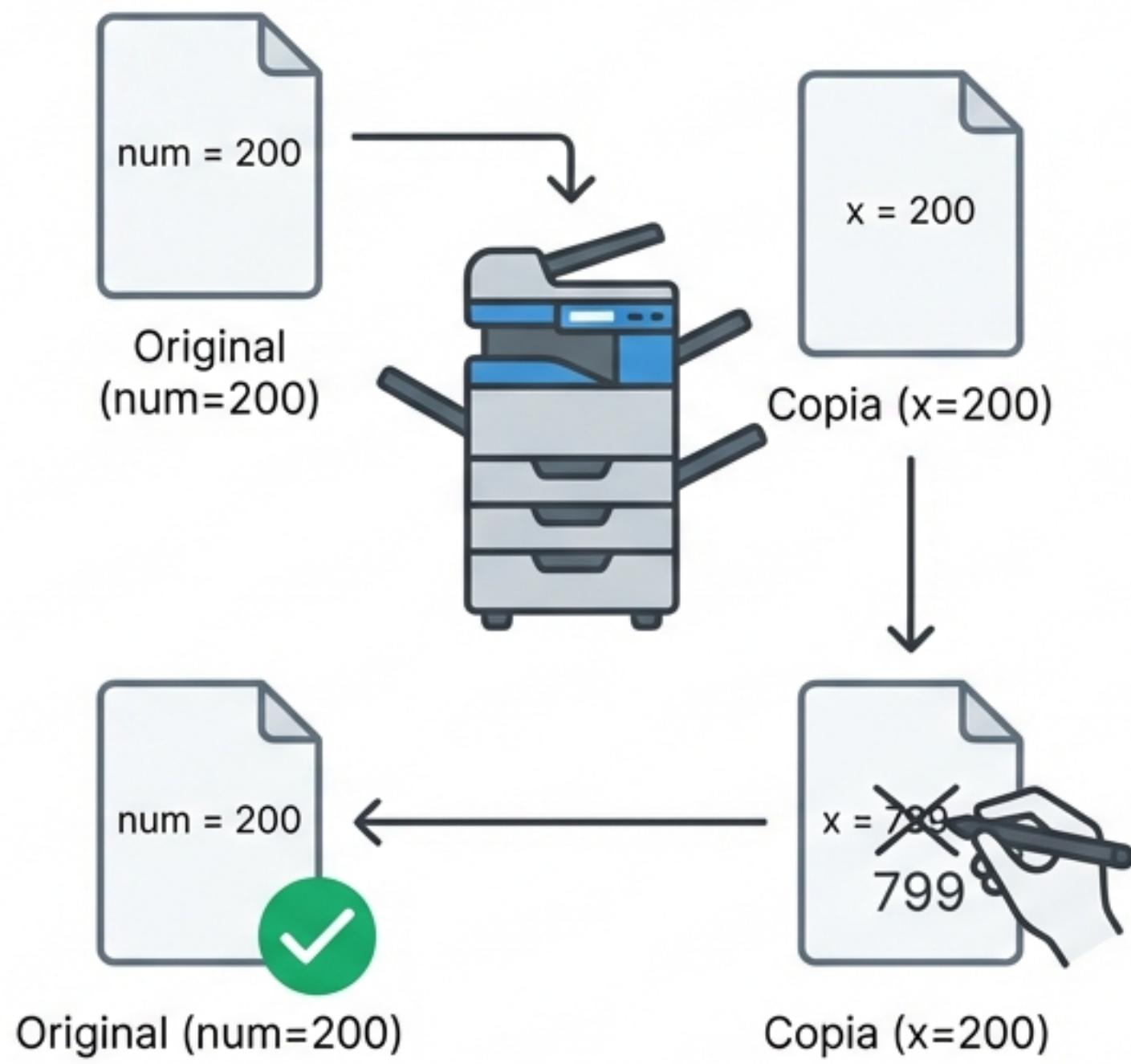
Conectando Módulos: El Paso de Parámetros



Regla de Oro: Deben coincidir en **ORDEN** y **TIPO** de dato.

Paso de Parámetros por Valor (La Copia)

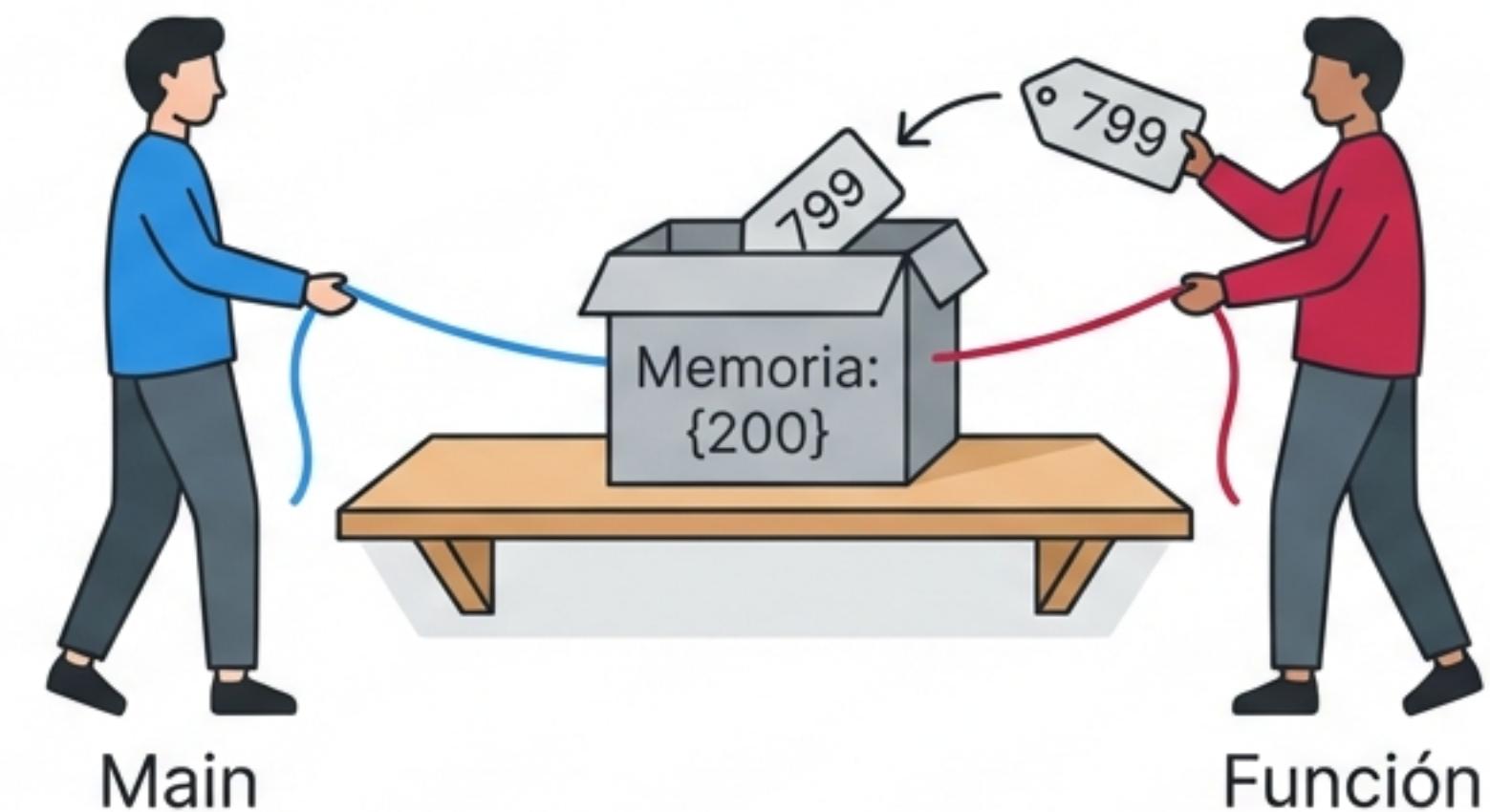
```
1 int num = 200;  
2 test(num);  
3 // num sigue siendo 200  
4  
5 void test(int x) {  
6     x = 799;  
7 }
```



La función trabaja sobre una fotocopia.
El original no cambia.

Paso de Parámetros por Referencia (El Enlace)

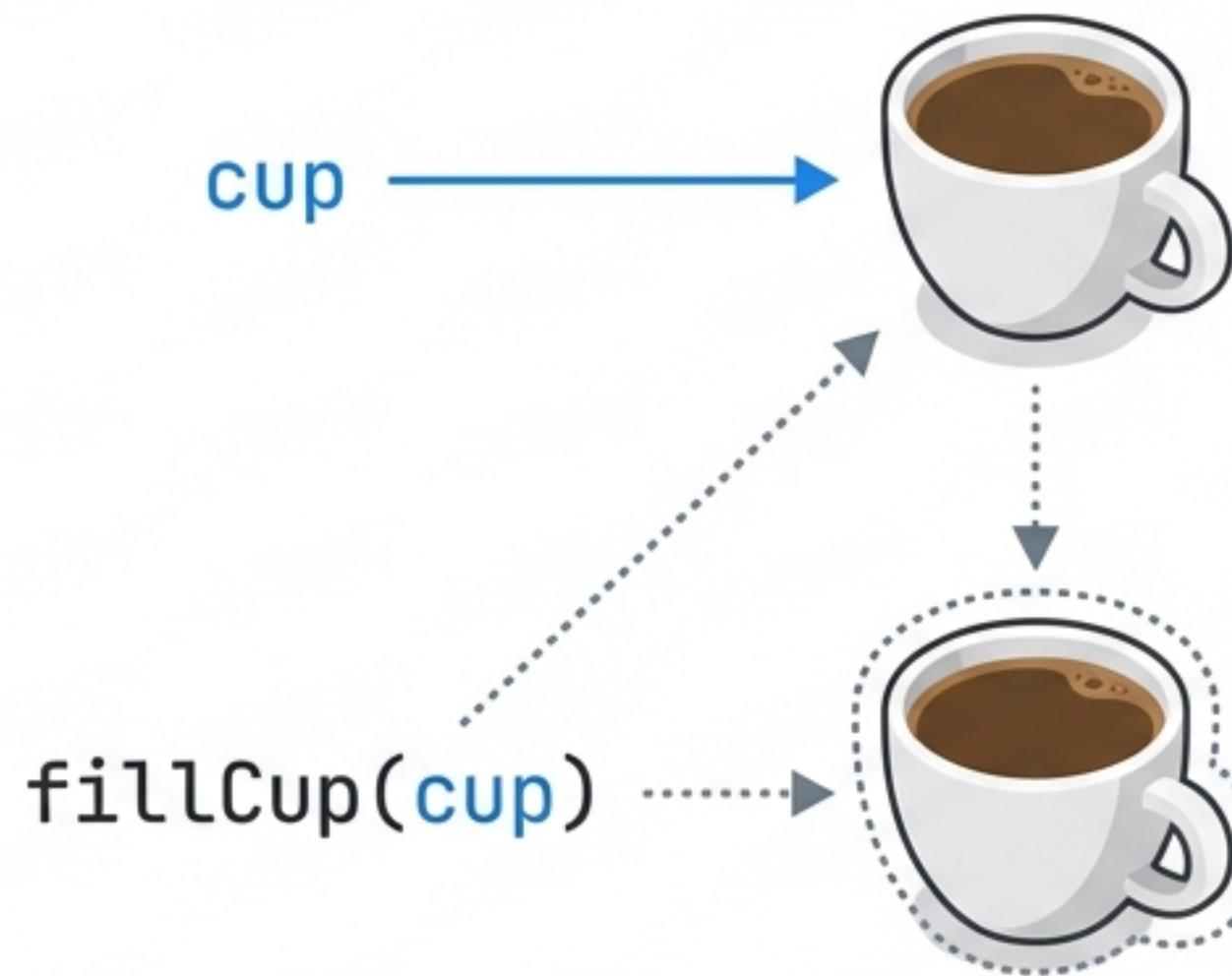
```
1 int[] lista = {200};  
2 test(lista);  
3 // lista[0] ahora es 799  
4  
5 void test(int[] x) {  
6     x[0] = 799;  
7 }
```



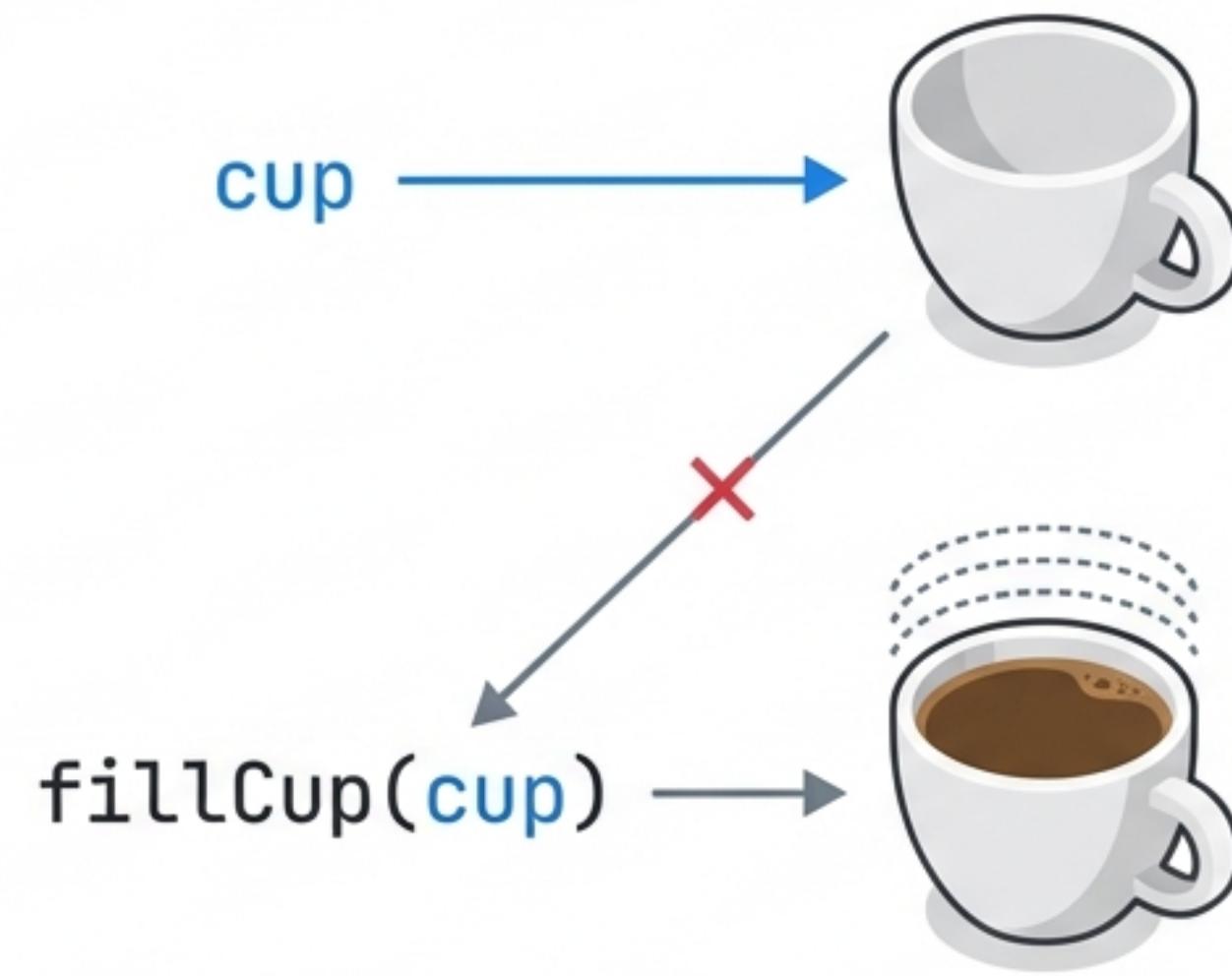
Ambos comparten un enlace al mismo objeto. Los cambios son globales.

Visualizando la Diferencia

Por Referencia



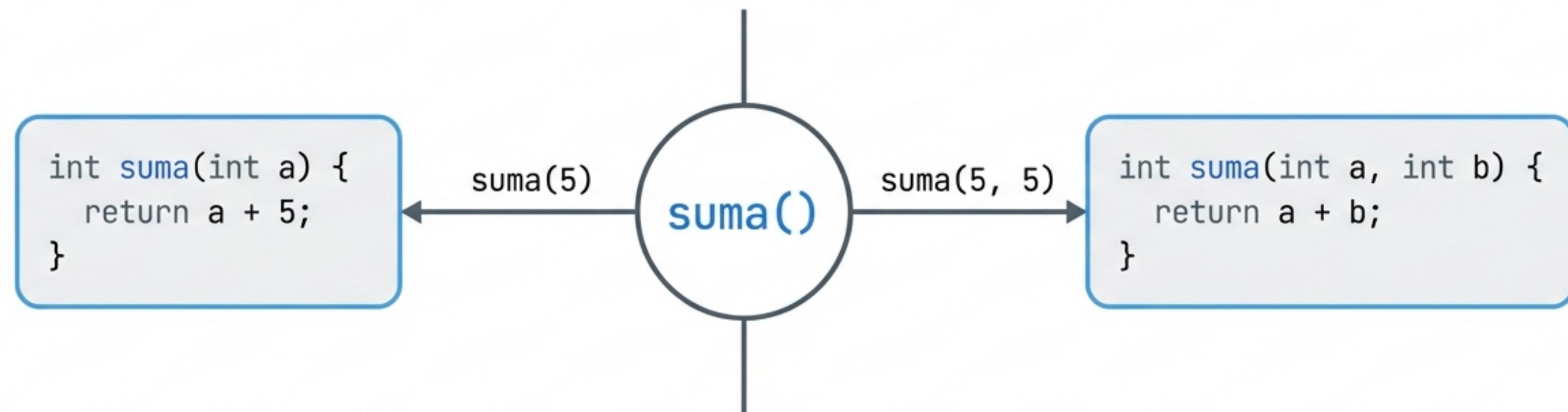
Por Valor



Objetos y Arrays pasan la referencia. Tipos primitivos pasan una copia.

Flexibilidad: Sobrecarga (Overloading)

Mismo nombre, distinta implementación.



El compilador decide qué función usar
según los parámetros recibidos.

No Reinventes la Rueda: Librerías

Los desarrolladores profesionales utilizan conjuntos de funciones predefinidas (API) para ahorrar tiempo y evitar errores.



Matemáticas

Math.sqrt()
Math.pow()
Math.round()



Cadenas de Texto

Manipulación
Búsqueda
Formato



Aleatorios

Generación de números
Probabilidad

Resumen y Mejores Prácticas

-  **Modularidad:** Divide problemas grandes en pequeños.
-  **Visibilidad:** Prefiere variables locales sobre globales.
-  **Parámetros:** Distingue entre paso por Valor (copia) y Referencia (enlace).
-  **Reutilización:** Si escribes lo mismo dos veces, crea una función.

```
public static void eco(String mensaje, int n) {  
    // Función que imprime el mensaje 'n' veces  
    for (int i = 0; i < n; i++) {  
        System.out.println(mensaje);  
    }  
}
```

Un código limpio es un código modular.