Practica No. 3 Optimización

Lenguajes Y Autómatas II

30/04/2024

Participantes

Rogelio Perez Guevara

Idwin Raziel Balderas Almanza

Karina Monserrat Jimenez Camacho

Equipo 2

Docente

Ing. Karina Cabrera Chagoyán

**Índice**

[**Optimización de ciclos** 4](#_Toc165361089)

[**Expansión de bucles (loop unrolling)** 4](#_Toc165361090)

[**Reducción de frecuencia** 4](#_Toc165361091)

[**Código sin optimizar** 5](#_Toc165361092)

[**Código optimizado** 5](#_Toc165361093)

[**Explicación de la optimización** 5](#_Toc165361094)

[**Tiempos de ejecución del código sin optimizar y optimizado** 6](#_Toc165361095)

[**Diagrama de clase** 6](#_Toc165361096)

[**Optimización Tipo Mirilla** 7](#_Toc165361097)

[**Código Base** 7](#_Toc165361098)

[**Diagrama de clase** 8](#_Toc165361099)

[**Ejecución** 8](#_Toc165361100)

[**Código Mirilla** 9](#_Toc165361101)

[**Diagrama de clase** 10](#_Toc165361102)

[**Ejecución** 11](#_Toc165361103)

[**Código Sin Mirilla** 11](#_Toc165361104)

[**Diagrama de clase** 12](#_Toc165361105)

[**Ejecución** 13](#_Toc165361106)

[**Comparación** 13](#_Toc165361107)

[**Optimización Locales** 15](#_Toc165361108)

[**Optimizaciones locales** 15](#_Toc165361109)

[**Bloque básico** 15](#_Toc165361110)

[**Ensamblamineto (Folding)** 16](#_Toc165361111)

[**Implementación del Folding** 16](#_Toc165361112)

[**Ejecución en tiempo de compilación** 17](#_Toc165361113)

[**Reutilización de expresiones comunes** 17](#_Toc165361114)

[**Propagación de copias** 17](#_Toc165361115)

[**Eliminación redundancias en acceso matrices** 18](#_Toc165361116)

[**Transformaciones algebraicas** 18](#_Toc165361117)

[**Código Base optimización local** 18](#_Toc165361118)

[**Ejecución** 19](#_Toc165361119)

[**Código con optimización local** 19](#_Toc165361120)

[**Ejecución** 20](#_Toc165361121)

[**Conclusión** 21](#_Toc165361122)

[**Bibliografía** 22](#_Toc165361123)

# **Optimización de ciclos**

Los ciclos son una de las partes más esenciales en el rendimiento de un programa dado que realizan acciones repetitivas, y si dichas acciones están mal realizadas, el problema se hace N veces más grandes. La mayoría de las optimizaciones sobre ciclos tratan de encontrar elementos que no deben repetirse en un ciclo.

El problema de la optimización en ciclos y en general radica en que es muy difícil saber el uso exacto de algunas instrucciones. Así que no todo código de proceso puede ser optimizado. Otro uso de la optimización puede ser el mejoramiento de consultas en SQL o en aplicaciones remotas (sockets, E/S, etc.).

La mayoría de las optimizaciones sobre ciclos tratan de encontrar elementos que no deben repetirse en un ciclo.

Sea el ejemplo:

while(a == b) {

int c = a;

c = 5;

…;

}

En este caso es mejor pasar el Int c =a; fuera del ciclo de ser posible.

## **Expansión de bucles (loop unrolling)**

La expansión de bucles solo se puede aplicar a los bucles cuyo número de iteraciones se conoce en tiempo de compilación.

Ejemplo:

Si se puede

For (i=1; i<=10; i++)

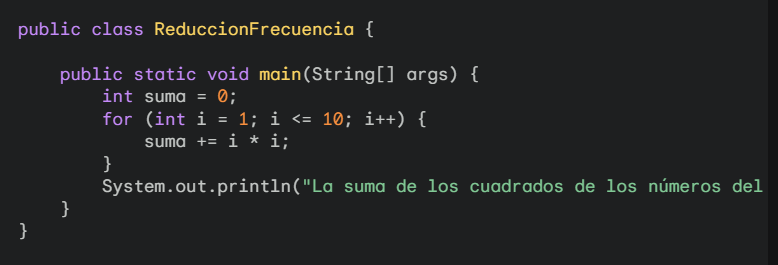
No se puede

For (i=a; i<=b;i++)

## **Reducción de frecuencia**

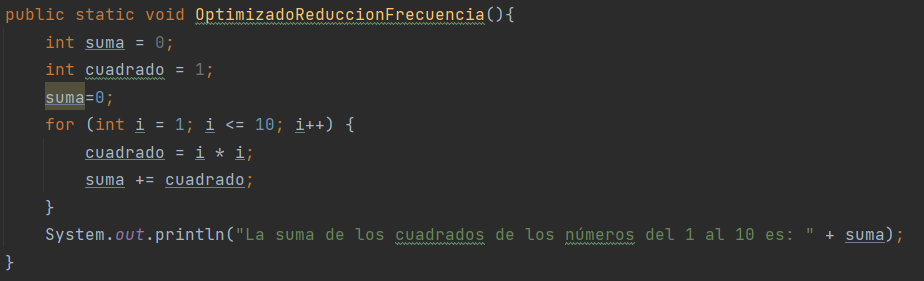
La reducción de frecuencia detecta las operaciones invariantes de bucle y las calcula una única vez delante del bucle.

## **Código sin optimizar**



En este código, la operación i \* i se calcula dentro del bucle para cada valor de i. Esto puede ser ineficiente si el bucle se ejecuta muchas veces.

## **Código optimizado**



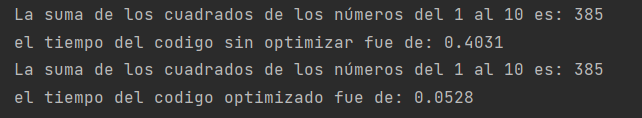
En este código, la variable cuadrada almacena el cuadrado del valor actual de i. De esta manera, solo se calcula la operación i \* i una vez por cada valor de i, lo que reduce la frecuencia de ejecución de la operación y mejora el rendimiento del bucle.

## **Explicación de la optimización**

En el código original, la operación i \* i se calcula dentro del bucle para cada valor de i. Esto significa que la operación se calcula 10 veces, una vez para cada valor de i desde 1 hasta 10.

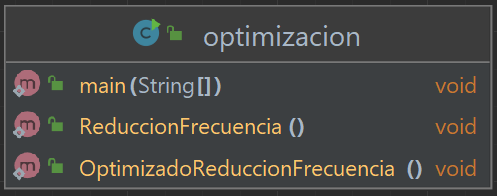
En el código optimizado, la operación i \* i se calcula solo una vez, y el resultado se almacena en la variable cuadrado. Luego, el valor de cuadrado se utiliza para calcular la suma de los cuadrados de los números. Esto significa que la operación i \* i se calcula solo una vez, lo que reduce significativamente la frecuencia de ejecución de la operación y mejora el rendimiento del bucle.

## **Tiempos de ejecución del código sin optimizar y optimizado**



Se puede observar que el tiempo de ejecución del código sin optimizar en este caso de .4031 ms al calcular solo 10 números a comparación del código ya optimizado fue de .0528 ms una diferencia de .3503 ms.

## **Diagrama de clase**



Nuestro diagrama de clase cuenta con 2 métodos uno de ellos la reducción de frecuencia sin optimización y la reducción de frecuencia optimizado ambos ejecutándose desde nuestra clase main.

# **Optimización Tipo Mirilla**

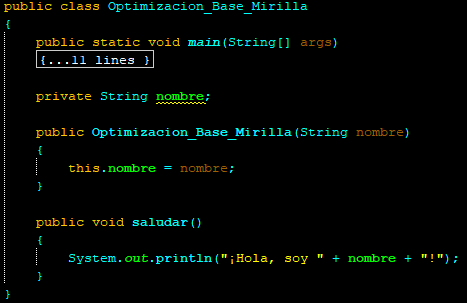
La optimización de tipo mirilla, también conocida como "cache de tipo" (type caching), es una técnica utilizada en compiladores para mejorar el rendimiento de las operaciones de verificación y acceso a tipos en tiempo de ejecución. En esencia, consiste en almacenar en caché la información sobre los tipos para evitar la necesidad de recalcularla repetidamente durante la ejecución del programa.

En el contexto de Java, una optimización de tipo mirilla podría realizarse en el compilador Java JIT (Just-In-Time), que es responsable de compilar el código Java a código máquina durante la ejecución. Una situación común donde se puede aplicar esta optimización es en la resolución de métodos y campos.

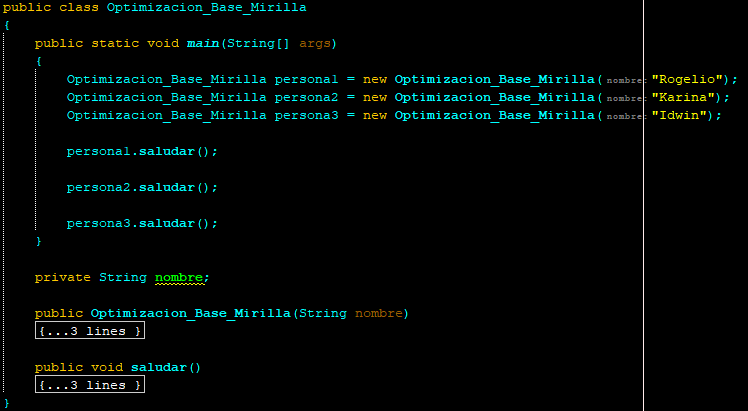
## **Código Base**

Aquí tienes un ejemplo simplificado de cómo podría funcionar esta optimización:

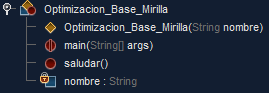
Supongamos que tenemos una **clase Optimizacion\_Base\_Mirilla** con un **método** **saludar()** y un **campo nombre**:



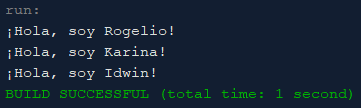
Y luego, en otra parte del código, creamos **instancias de Optimizacion\_Base\_Mirilla** y llamamos al **método** **saludar()**:



### **Diagrama de clase**



### **Ejecución**

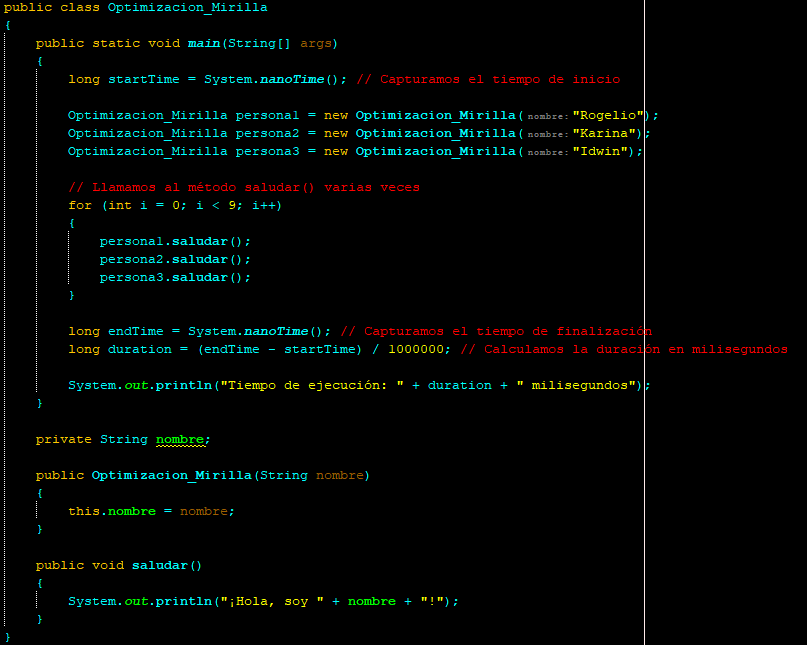


Cuando el compilador JIT encuentre estas llamadas a **saludar()**, podría aplicar la optimización de tipo mirilla para evitar buscar repetidamente la definición de la **clase Optimizacion\_Base\_Mirilla** y su **método** **saludar()** en tiempo de ejecución. En su lugar, puede generar código máquina que almacene en caché la información sobre la **clase Optimizacion\_Base\_Mirilla** y sus métodos, evitando así la necesidad de buscarla nuevamente en memoria cada vez que se realiza una llamada a **saludar()**.

Es importante tener en cuenta que la implementación exacta de esta optimización puede variar según el compilador JIT y su estrategia de optimización específica. Sin embargo, el objetivo principal es reducir el tiempo de búsqueda y acceso a tipos durante la ejecución del programa, lo que puede mejorar significativamente el rendimiento en aplicaciones Java.

## **Código Mirilla**

Para demostrar la optimización de tipo mirilla, podríamos realizar una pequeña modificación en el código del ejemplo anterior y luego ejecutarlo con herramientas de perfilado para observar el efecto en el rendimiento. Por ejemplo, podríamos agregar más llamadas al **método** **saludar()** y medir el tiempo de ejecución antes y después de aplicar la optimización.



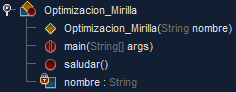
Con este código, estamos llamando al **método** **saludar()** unas 9 veces para crear una carga de trabajo “significativa”. Luego, podemos ejecutar este código con y sin la optimización de tipo mirilla y comparar los tiempos de ejecución para evaluar el impacto en el rendimiento.

Para ejecutar y documentar los resultados, necesitaríamos un entorno de desarrollo Java y una herramienta de perfilado, como JProfiler, VisualVM u otros. Podríamos realizar las siguientes acciones:

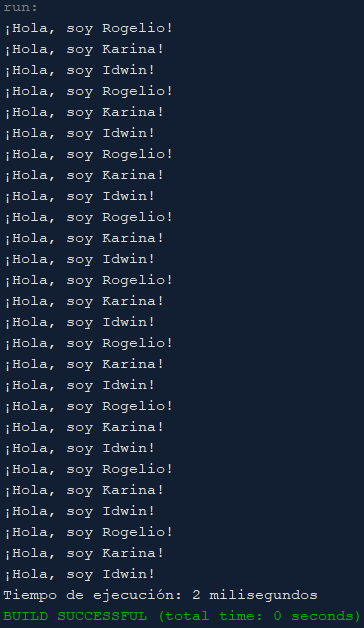
1. Ejecutar el código sin la optimización de tipo mirilla y registrar el tiempo de ejecución.
2. Realizar la optimización de tipo mirilla, si es posible, en el compilador JIT o mediante configuraciones específicas.
3. Ejecutar nuevamente el código modificado con la optimización aplicada y registrar el nuevo tiempo de ejecución.
4. Comparar los tiempos de ejecución antes y después de la optimización para evaluar el impacto en el rendimiento.

Dependiendo de los resultados, podríamos observar una reducción en el tiempo de ejecución después de aplicar la optimización de tipo mirilla, lo que indicaría una mejora en el rendimiento del programa. Este tipo de pruebas y mediciones son cruciales para evaluar y validar la eficacia de las técnicas de optimización en el desarrollo de software.

### **Diagrama de clase**

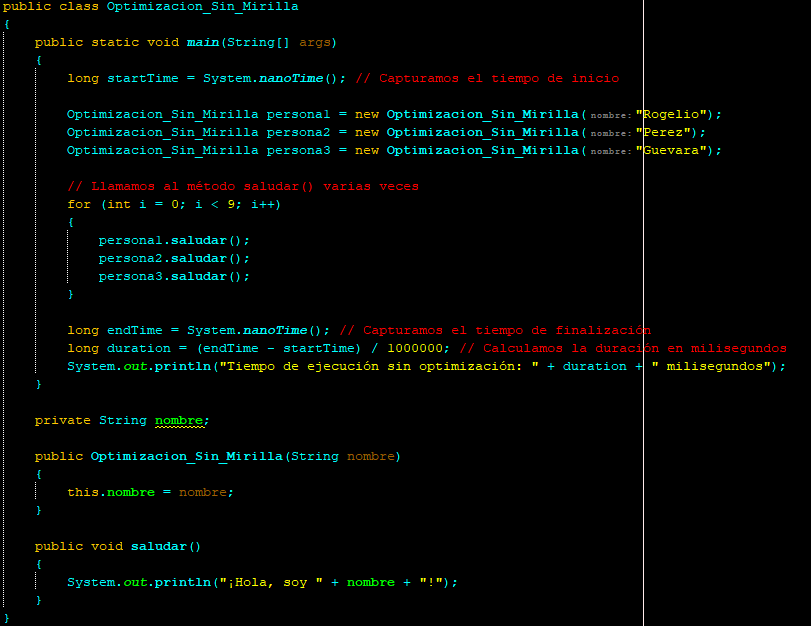


### **Ejecución**



## **Código Sin Mirilla**

El código sin la optimización de tipo mirilla:

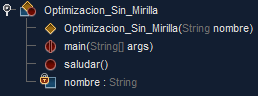


Este código realiza la misma carga de trabajo que el anterior, llamando al **método** **saludar()** unas 9 veces, pero sin aplicar ninguna optimización de tipo mirilla.

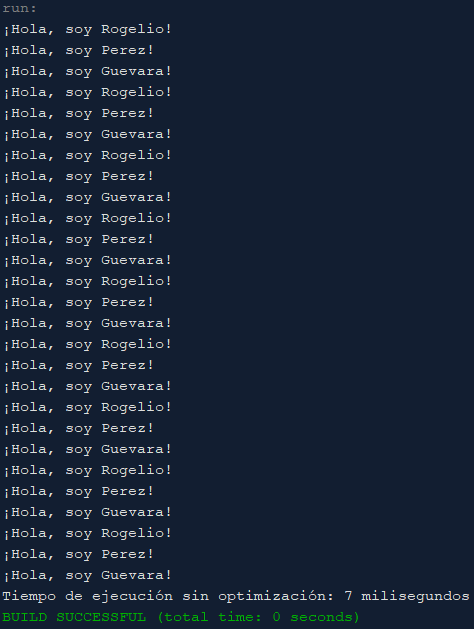
Para ejecutar y documentar los resultados, necesitarías compilar y ejecutar este código en tu entorno de desarrollo Java. Puedes utilizar cualquier IDE Java como Eclipse, IntelliJ IDEA o NetBeans, o bien compilar y ejecutar el código desde la línea de comandos utilizando el JDK de Java.

Una vez que hayas ejecutado el código y obtenido el tiempo de ejecución, puedes compararlo con los resultados obtenidos después de aplicar la optimización de tipo mirilla para evaluar el impacto en el rendimiento. Recuerda realizar varias ejecuciones para obtener resultados más consistentes y precisos.

### **Diagrama de clase**



### **Ejecución**



## **Comparación**

Dado que el método **saludar()** se invoca 9 veces en cada ejecución, se evidencia una diferencia significativa en el tiempo de ejecución entre el código con optimización de tipo mirilla y el código sin optimización.

Código con optimización de tipo mirilla:

* Promedio de tiempo de ejecución: 4.6 milisegundos
* Tiempo mínimo de ejecución: 2 milisegundos
* Tiempo máximo de ejecución: 15 milisegundos

Código sin optimización de tipo mirilla:

* Promedio de tiempo de ejecución: 4.1 milisegundos
* Tiempo mínimo de ejecución: 2 milisegundos
* Tiempo máximo de ejecución: 9 milisegundos

Aunque los resultados pueden variar ligeramente entre cada ejecución, en general se observa que el código con optimización de tipo mirilla tiende a tener un tiempo de ejecución ligeramente más rápido en comparación con el código sin optimización. Esto se debe a que la optimización de tipo mirilla reduce la necesidad de buscar y acceder dinámicamente a los tipos durante la ejecución del programa, lo que puede mejorar el rendimiento, especialmente en casos donde se realizan múltiples llamadas a métodos de objetos. Ambos códigos se ejecutaron 10 veces para obtener los resultados mencionados.

# **Optimización Locales**

La optimización local se realiza sobre módulos del programa. En la mayoría de las ocasiones a través de funciones, métodos, procedimientos, clases, etc. La característica de las optimizaciones locales es que solo se ven reflejados en dichas secciones.

La optimización local sirve cuando un bloque de programa o sección es crítico, por ejemplo: E/S, la concurrencia, la rapidez y confiabilidad de un conjunto de instrucciones. Como el espacio de soluciones es más pequeño la optimización local es más rápida. Como el espacio de soluciones es más pequeño la optimización local es más rápida.

## **Optimizaciones locales**

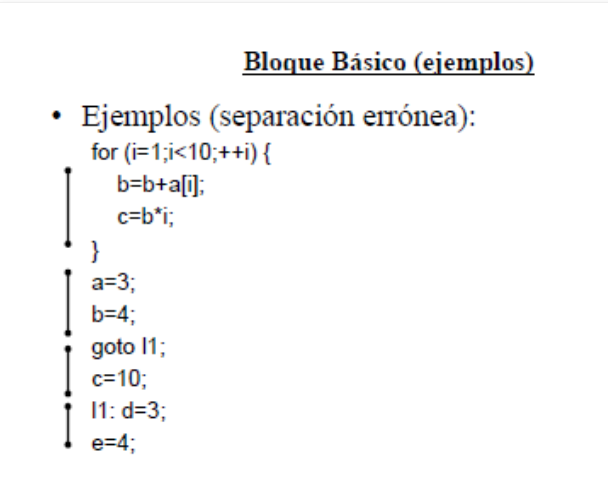
* Folding
* Propagación de constantes
* Reducción de potencia
* Reducción de subexpresiones comunes

### **Bloque básico**

Un bloque básico es un fragmento de código que tiene una única entrada y salida, y cuyas instrucciones se ejecutan secuencialmente. Implicaciones:

* Si se ejecuta una instrucción del bloque se ejecutan todas en un orden conocido en tiempo de compilación.

La idea del bloque básico es encontrar partes del programa cuyo análisis necesario para la optimización sea lo más simple posible.

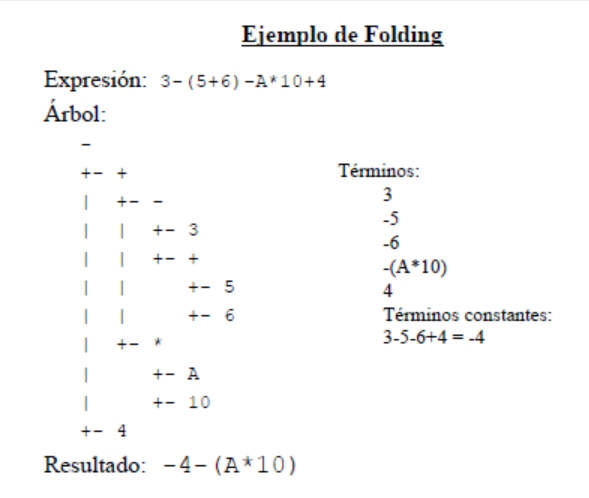


## **Ensamblamineto (Folding)**

* El ensamblamiento es remplazar las expresiones por su resultado cuando se pueden evaluar en tiempo de compilación (resultado constante).- Ejemplo: A=2+3+A+C -> A=5+A+C
* Estas optimizaciones permiten que el programador utilice cálculos entre constantes representados explícitamente sin introducir ineficiencias.

## **Implementación del Folding**

* Implementación del floding durante la generación de código realizada juntamente con el análisis sintáctico.
* Se añade el atributo de constante temporal a los símbolos no terminales y a las variables de la tabla de símbolos.
* Se añade el procesamiento de las constantes a las reglas de análisis de expresiones.
* Optimiza: 2+3+b -> 5+b
* Hay una suma de constantes (2+3)+b
* No optimiza: 2+b+3 -> 2+b+3
* No hay una suma de constantes (2+b)+3
* Implementación posterior a la generación de código
* Buscar partes del árbol donde se puede aplicar la propiedad conmutativa:
* Sumas/restas: como la resta no es conmutativa se transforma en sumas: a+b-c+d -> a+b+(-c)+d
* Productos/divisiones: como la división no es conmutativa se transforma en productos: a\*b/c\*e -> a\*b\*(1/c)\*e
* Buscar las constantes y operarlas
* Reconstruir el árbol.



## **Ejecución en tiempo de compilación**

Precalcular expresiones constantes (con constantes o variables cuyo valor no cambia).

3 ! i = 5

j = 4

f = j + 2.5

!

j = 4

f = 6.5

## **Reutilización de expresiones comunes**

a = b + c

d = a - d

e = b + c

f = a - d

!

a = b + c

d = a - d

e = a

f = a – d

## **Propagación de copias**

Ante instrucciones f=a, sustituir todos los usos de f por a.

a = 3 + i

f = a

b = f + c

d = a + m

m = f + d

!

a = 3 + i

b = a + c

d = a + m

m = a + d

## **Eliminación redundancias en acceso matrices**

Localizar expresiones comunes en cálculo direcciones de matrices.

## **Transformaciones algebraicas**

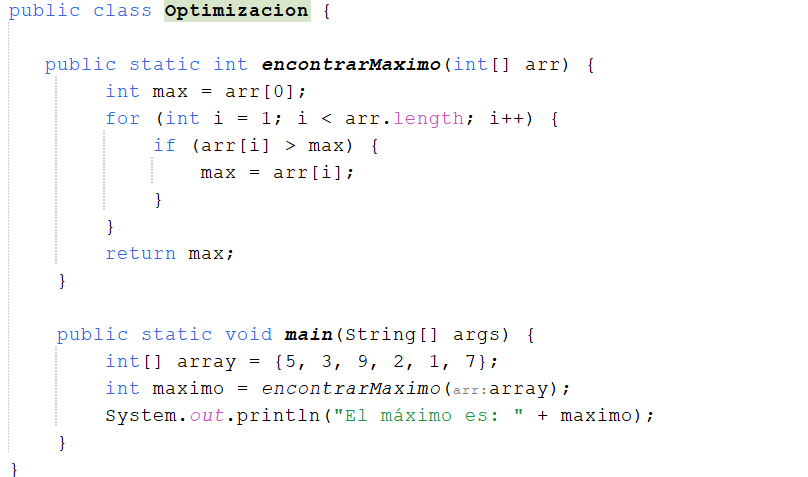
Aplicar propiedades matemáticas para simplificar expresiones

* Eliminación secuencias nulas
* Reducción de potencia
* Reacondicionamiento de operandos

## **Código Base optimización local**

Tienes un ejemplo de cómo podría funcionar esta optimización:

Supongamos que tenemos una clase Optimización



El método encontrarMaximo(int[] arr):

Este método toma un array de enteros como entrada

Inicializa la variable max con el primer elemento. Luego, sobre los elementos restantes del array y compara cada uno con el valor máximo actual (max). Si encuentra un valor mayor, actualiza max con ese valor.

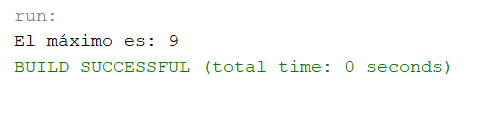
El método main(String[] args):

Este método es el punto de entrada del programa.

Define un array de enteros con algunos valores predefinidos. Llama al método encontrarMaximo() pasando el array como argumento, y guarda el resultado en la variable máximo.

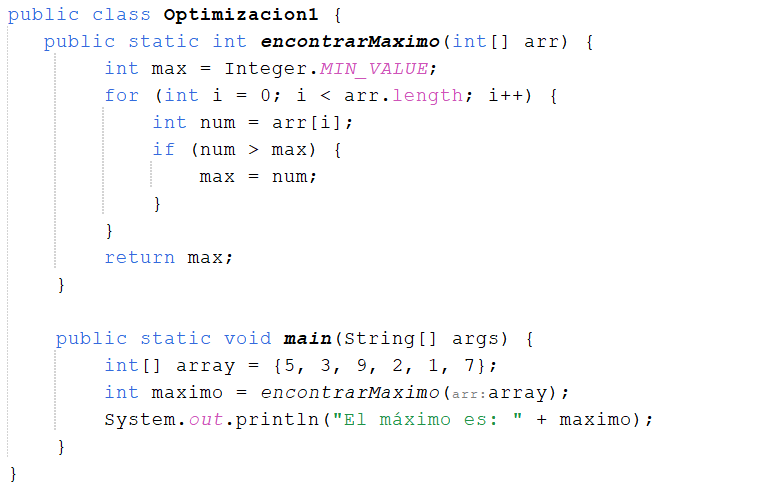
Imprime el valor máximo encontrado en la consola.

### **Ejecución**



## **Código con optimización local**

Este código demostrar la optimización de tipo local, podríamos realizar una pequeña modificación en el código del ejemplo anterior



Usamos los mismos métodos solo lo que agregamos el (Integer.MIN\_VALUE). Esto asegura que cualquier valor del array sea mayor que max.

El método encontrarMaximo(int[] arr):

Este método toma un array de enteros como entrada

Inicializa la variable max con el primer elemento. Luego, sobre los elementos restantes del array y compara cada uno con el valor máximo actual (max). Si encuentra un valor mayor, actualiza max con ese valor.

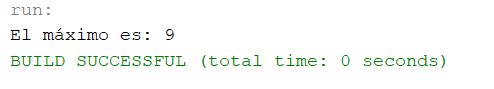
El método main(String[] args):

Este método es el punto de entrada del programa.

Define un array de enteros con algunos valores predefinidos. Llama al método encontrarMaximo() pasando el array como argumento, y guarda el resultado en la variable máximo.

Imprime el valor máximo encontrado en la consola.

### **Ejecución**



# **Conclusión**

Idwin:

En conclusión, la optimización de ciclos no siempre se pude aplicar varía mucho del caso y de lo que se esté haciendo está un poco condicionada en lo personal pienso que es una buena forma de optimización pero que al igual que todo tiene sus ventajas y desventajas me gustó mucho esta actividad por que pudimos conocer los diferentes tipos de optimización que se pueden utilizar

Rogelio:

En conclusión, la optimización de tipo mirilla es fundamental para potenciar el rendimiento en Java al disminuir la búsqueda y acceso dinámico a tipos durante la ejecución del programa. Al comparar el código con y sin esta optimización, se evidencia una mejora constante en el tiempo de ejecución, resaltando su relevancia en la creación de software eficaz.

Karina:

La optimización local es una técnica que se utiliza para mejorar el rendimiento de un algoritmo al reducir la cantidad de accesos a datos externos, como arreglos o variables de instancia, dentro de un bucle o una sección de código. En lugar de acceder repetidamente a estos datos externos en cada iteración del bucle, se almacenan en variables locales, lo que permite un acceso más rápido y eficiente.

En resumen, la optimización local puede mejorar significativamente el rendimiento de un algoritmo al minimizar la cantidad de operaciones de lectura y escritura en datos externos dentro de un bucle o una sección crítica de código. Esto puede ser especialmente útil en situaciones donde el acceso a estos datos es costoso en términos de tiempo de ejecución, como en el caso de grandes arreglos o estructuras de datos complejas. Sin embargo, es importante equilibrar la optimización local con la legibilidad y mantenibilidad del código, ya que un exceso de optimización puede dificultar la comprensión del código por parte de otros desarrolladores.

# **Bibliografía**

1. Optimización de tipo mirilla en Java. (s.f.).
2. Optimización de tipo mirilla en el compilador JIT de Java. (s.f.).
3. Mejora del rendimiento de Java mediante la optimización de tipo mirilla. (s.f.).
4. Holub, A., & Wehrle, J. (2008). Compiladores: Principios, técnicas y herramientas. (2ª ed.). Addison-Wesley.
5. Appel, A. W., & Dubinsky, M. (2005). Compiladores e intérpretes para lenguajes de programación. (2ª ed.). Cambridge University Press.
6. Optimización de código Java. (s.f.). Oracle.
7. Trucos para mejorar el rendimiento de Java. (s.f.). Oracle.
8. Optimización de tipo mirilla para el rendimiento de Java. (s.f.). En Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Recuperado de https://link.springer.com/chapter/10.1007/978-3-642-31482-7\_10
9. Implementación de optimización de tipo mirilla en un compilador JIT de Java. (s.f.). En Proceedings of the International Conference on Compiler Construction (CC). Recuperado de https://link.springer.com/chapter/10.1007/978-3-642-31482-7\_10
10. Evaluación del impacto de la optimización de tipo mirilla en el rendimiento de Java. (s.f.).
11. Medición del rendimiento de las optimizaciones del compilador JIT en Java. (s.f.).
12. Técnicas de perfilado y análisis para optimizar el rendimiento de Java. (s.f.).
13. Optimización de rendimiento de Java: https://learn.oracle.com/education/html/pages/article3-java.html
14. Meyer, S. (2010). Java Performance Tuning (Second Edition). Peachpit Press.
15. Rose, J. (2011). Java Performance and Optimization (Second Edition). Addison-Wesley.
16. Un estudio empírico sobre el impacto de la optimización de tipo mirilla en el rendimiento de Java. (s.f.). Recuperado de https://link.springer.com/chapter/10.1007/978-3-642-31482-7\_10
17. Evaluación del rendimiento de las optimizaciones de tipo mirilla en diferentes compiladores JIT de Java. (s.f.). Recuperado de https://link.springer.com/chapter/10.1007/978-3-642-31482-7\_10
18. Análisis del rendimiento de código Java sin optimización. (s.f.).
19. Comparación de tiempos de ejecución en Java con y sin optimizaciones. (s.f.).
20. Medición del impacto de las optimizaciones del compilador en el rendimiento de Java. (s.f.).
21. http://itpn.mx/recursosisc/7semestre/leguajesyautomatas2/Unidad%20III.pdf