



## **Primer Estudio De Caso: Gestión de Llamadas a Rutinas.**

Estudiante:

Minor Andrés Mora Vargas

Escuela de ingeniería del Software, U CENFOTEC

Estructuras de Datos, SOFT-10

Prof. Romario Salas Cerdas

San José, 19 de octubre de 2025

## Tabla de Contenido

<b>Introducción .....</b>	<b>3</b>
<b>Desarrollo .....</b>	<b>4</b>
<b>Seguridad, orden y eficiencia.....</b>	<b>4</b>
<b>Ejemplo práctico .....</b>	<b>5</b>
<b>Aplicación del estudio de caso .....</b>	<b>5</b>
<b>Descripción de un ejemplo concreto en software de sistema .....</b>	<b>6</b>
<b>Ejemplos visuales:.....</b>	<b>7</b>
1.    Stack Trace real en Java:.....	7
2.    Representación gráfica del Call Stack: .....	8
<b>Representación visual del comportamiento de la pila.....</b>	<b>8</b>
1.    Estado inicial de la pila vacía: .....	8
2.    Primera llamada apilada:.....	9
3.    Llamada anidada: .....	9
4.    Finalización de la llamada interna:.....	9
5.    Finalización de “println” y pila vacía: .....	10
<b>Conclusión .....</b>	<b>11</b>
<b>Referencias .....</b>	<b>12</b>

## Introducción

La pila es una de las estructuras de datos más importantes dentro de la programación, ya que representa la forma en que un programa organiza y controla sus operaciones. Su principio de “último en entrar, primero en salir” se aplica perfectamente cuando las funciones se van llamando unas a otras dentro del código. Cada vez que se ejecuta una función, esta se apila, y al terminar, se desapila, permitiendo que el programa retome su flujo sin perder el orden.

En este trabajo se desarrolla la idea de un gestor de llamadas a funciones, que utiliza una pila para manejar el proceso de ejecución y retorno de cada rutina. El objetivo es comprender cómo se organiza internamente esa secuencia de llamadas y cómo la pila ayuda a mantener todo bajo control. Más que un ejercicio técnico, se busca entender la lógica detrás de su funcionamiento: cómo algo tan simple como apilar y desapilar puede sostener el orden completo de un programa y garantizar que cada función retorne correctamente a su punto de origen.

## **Desarrollo**

### **Importancia de la pila en el gestor de llamadas**

La pila cumple un papel fundamental en la ejecución de los programas, ya que permite controlar el orden en que se ejecutan las funciones y cómo se devuelven los resultados. Su principio de LIFO refleja exactamente lo que ocurre cuando una función invoca a otra.

Por ejemplo, si la función `main` llama a `CalcularPromedio`, y esta a su vez llama a `SumarNotas`, la última en ejecutarse será `SumarNotas`, pero también será la primera en finalizar y salir de la pila. Este mecanismo mantiene la secuencia ordenada y evita errores en el flujo del programa.

Cuando se ejecuta una nueva función, se crea un espacio temporal en la pila donde se guardan sus variables locales, los parámetros que recibe y la dirección de retorno, que indica a qué punto del programa se debe volver al terminar. Cuando la función finaliza, ese espacio se elimina y el control regresa a la función anterior. De esta forma, la pila actúa como un organizador automático que administra las llamadas activas y garantiza que cada ejecución ocurra en el orden correcto.

### **Seguridad, orden y eficiencia**

Una de las razones por las que la pila es tan útil es porque ofrece seguridad en el control del flujo. Gracias a ella, el programa puede detectar errores como llamadas incorrectas, retornos incompatibles o incluso desbordamientos de pila (Stack Overflow) cuando una función se llama a sí misma de forma infinita.

Además, las operaciones principales de la pila (`push` y `pop`) tienen un costo constante  $O(1)$ , lo que la hace extremadamente eficiente: puede manejar miles de llamadas en muy poco tiempo sin consumir demasiados recursos.

En cuanto al orden, la pila garantiza que el programa se ejecute de manera estructurada. No importa cuántas funciones estén anidadas, siempre se completan de adentro hacia afuera, lo que evita confusiones y pérdida de datos. Esto resulta esencial en lenguajes que permiten recursividad, ya que cada llamada se almacena con sus propios valores sin interferir con las demás.

### Ejemplo práctico

Un ejemplo claro del uso de la pila se da en un programa con funciones anidadas como:

```
main() → println() → sqrt(20)
```

Primero se apila main, luego println, y dentro de esta se apila sqrt. Cuando sqrt termina su ejecución, se desapila y devuelve el resultado a println, que lo imprime y se desapila también. Finalmente, solo queda main, que concluye el programa.

Este proceso de apilar y desapilar ocurre constantemente en cualquier aplicación, desde las más simples hasta los sistemas operativos. Incluso los errores comunes como el Stack Overflow provienen de este mismo concepto: cuando la pila se llena por exceso de llamadas recursivas sin final, el sistema deja de tener espacio para nuevas funciones y se detiene por seguridad. Esto demuestra que, aunque la pila parezca una estructura simple, tiene una gran importancia en la estabilidad y control del software.

### Aplicación del estudio de caso

En este estudio se implementa una simulación del comportamiento real de la pila en la gestión de llamadas a funciones. Para ello se crearon las clases Nodo, Pila, Llamada y GestorLlamadas.

Cada vez que el usuario llama una función desde el menú, el sistema ejecuta un PUSH, agregando un nodo con la información de esa rutina.

Cuando se finaliza una función, se realiza un POP, removiendo el nodo del tope y mostrando cómo el control regresa a la función anterior.

Este enfoque permite visualizar en consola el comportamiento interno de la pila, tal como ocurre en un programa real. Además, demuestra cómo la pila contribuye a mantener la lógica del programa bien estructurada, evitando errores de flujo o pérdida de información. Aunque en el código Java no se maneja la memoria real de las funciones, la simulación ayuda a comprender el mismo principio que utiliza cualquier lenguaje para ejecutar sus llamadas y retornos de manera segura y ordenada.

### **Descripción de un ejemplo concreto en software de sistema**

En los sistemas operativos modernos, el control de las funciones y procesos depende directamente de una pila interna. Un ejemplo muy claro se encuentra en Linux, dentro del manejo de los procesos y subprocessos.

Cada vez que un programa se ejecuta, el sistema crea un stack de ejecución exclusivo para ese proceso. En él se almacenan los datos necesarios para seguir la secuencia de llamadas, como las direcciones de retorno, los parámetros de las funciones y las variables locales. Si una función A llama a una función B, el sistema apila la información de B sobre A, y cuando B termina, se desapila para continuar con A.

Este mismo mecanismo también se observa en Java, dentro de la JVM (Java Virtual Machine). La JVM utiliza un Java Stack para controlar la ejecución de los métodos. Por ejemplo, si un programa ejecuta `main()`, y dentro de él se llama a “ProcesarDatos”, y luego a “CalcularPromedio”, el orden de la pila será:

```
Main → ProcesarDatos → CalcularPromedio.
```

Cuando “CalcularPromedio” finaliza, se desapila y el control regresa a “ProcesarDatos”, y luego finalmente a `main`.

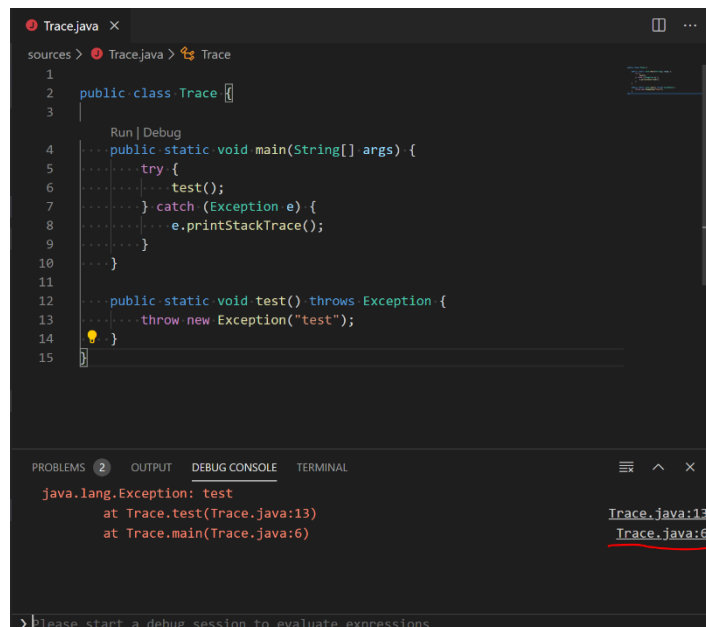
Es importante señalar que, si ocurre un error dentro de una función, el sistema muestra el contenido de la pila en el mensaje de error, conocido como Stack Trace. Este registro indica exactamente en qué punto falló la ejecución, ayudando al programador a depurar el código.

En resumen, tanto en Linux como en Java, la pila cumple el mismo papel que en el gestor de llamadas desarrollado: mantener el orden de ejecución y permitir que cada función retorne correctamente a su origen, garantizando un flujo estable y seguro dentro del software.

## Ejemplos visuales:

### 1. Stack Trace real en Java:

La imagen muestra la traza de pila generada por la JVM al producirse una excepción. En ella se observa cómo las llamadas activas “main” y “test” se registran en orden descendente, reflejando la estructura de la pila de ejecución del programa.



```
Trace.java X
sources > Trace.java Trace
1
2 public class Trace {
3
4     Run | Debug
5     public static void main(String[] args) {
6         try {
7             test();
8         } catch (Exception e) {
9             e.printStackTrace();
10        }
11
12    public static void test() throws Exception {
13        throw new Exception("test");
14    }
15 }

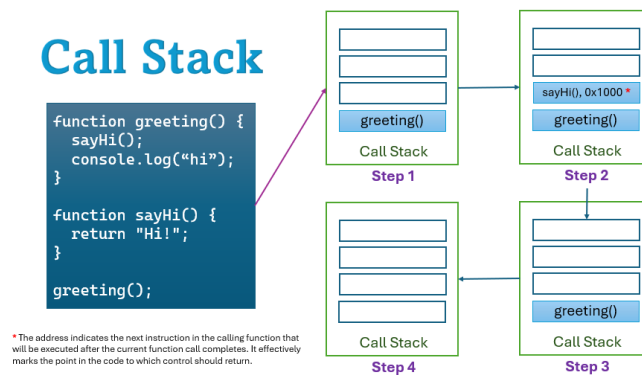
PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
java.lang.Exception: test
    at Trace.test(Trace.java:13)
    at Trace.main(Trace.java:6)
Trace.java:13
Trace.java:6

> please start a debug session to evaluate expressions
```

## 2. Representación gráfica del Call Stack:

El diagrama ilustra cómo las funciones se apilan y desapilan durante la ejecución. Cada vez que se llama una función, se apila en la parte superior; al finalizar, se desapila para devolver el control a la anterior.

Este flujo visualiza el mismo principio LIFO que se simuló en el gestor de llamadas desarrollado en este estudio.



## Representación visual del comportamiento de la pila

Para evidenciar el comportamiento del programa, muestro capturas del menú principal y de las operaciones realizadas con la pila. Cada imagen representa un momento clave en la ejecución del sistema.

### 1. Estado inicial de la pila vacía:

Cuando iniciamos el programa, la pila se encuentra vacía. Desde el menú se pueden realizar operaciones que simulan el proceso real de apilar y desapilar llamadas a funciones.

```
===== GESTOR DE LLAMADAS A RUTINAS =====
1. Llamar a una función
2. Finalizar última llamada
3. Mostrar pila actual
0. Salir
Seleccione una opción:
```



## 2. Primera llamada apilada:

La función “println” se apila, quedando en el tope de la pila. Representa una llamada activa que aún no ha finalizado.

```
Nombre de la función: println
Tipo de retorno: void
Parámetros: double
PUSH ? Llamada a println(double) -> void

--- Estado actual de la pila ---
println(double) -> void
-----
```

## 3. Llamada anidada:

Ahora la función “segundaprueba” se apila encima de “println”, indicando que está siendo ejecutada dentro de ella. Este estado demuestra el comportamiento de la pila.

```
Nombre de la función: segundaprueba
Tipo de retorno: double
Parámetros: double
PUSH ? Llamada a segundaprueba(double) -> double

--- Estado actual de la pila ---
segundaprueba(double) -> double
println(double) -> void
-----
```

## 4. Finalización de la llamada interna:

Ahora, la función que estaba en el tope ha terminado su ejecución y se desapila, retornando el control a la función anterior. El estado actual de la pila muestra que “println” permanece activa.

```
===== GESTOR DE LLAMADAS A RUTINAS =====
1. Llamar a una función
2. Finalizar última llamada
3. Mostrar pila actual
0. Salir
Seleccione una opción: 2
POP ? Retorna de segundaprueba(double) -> double

--- Estado actual de la pila ---
println(double) -> void
-----
```

### 5. Finalización de “println” y pila vacía:

Por último, se representa el último paso del proceso, la función “println” que era la primera llamada, ha finalizado y se ha desapilado. El mensaje confirma que ya no quedan llamadas activas y que el flujo del programa terminó correctamente.

```
===== GESTOR DE LLAMADAS A RUTINAS =====  
1. Llamar a una función  
2. Finalizar última llamada  
3. Mostrar pila actual  
0. Salir  
Seleccione una opción: 2  
POP ? Retorna de println(double) -> void  
La pila está vacía.
```

## Conclusión

El desarrollo de este estudio permitió comprender de manera práctica cómo una estructura tan sencilla como la pila puede sostener todo el orden de ejecución dentro de un programa. A través del gestor de llamadas a rutinas se pudo visualizar lo que normalmente sucede de forma invisible en el interior de los lenguajes de programación: cómo las funciones se apilan al ser llamadas y se desapilan al finalizar, siguiendo el principio de “último en entrar, primero en salir”.

Más allá de la parte técnica, el ejercicio ayudó a conectar la teoría con la realidad del software moderno. Tanto los sistemas operativos como los entornos de ejecución, como la JVM en Java, dependen de la pila para mantener el control y la estabilidad del flujo de ejecución. Esto demuestra que, incluso en programas complejos, las bases de la programación siguen apoyándose en estructuras de datos simples, pero bien diseñadas.

Para nosotros como estudiantes, entender el funcionamiento de la pila no solo mejora la capacidad de programar con lógica y orden, sino que también permite analizar y depurar el código con mayor claridad. Este conocimiento se vuelve fundamental para cualquier desarrollador o aspirante que busque escribir programas más seguros, eficientes y fáciles de mantener.

## Referencias

GeeksforGeeks. (2025, 10 de septiembre). *Introduction to Stack Data Structure*.

Recuperado el 17 de octubre de 2025, de

<https://www.geeksforgeeks.org/dsa/introduction-to-stack-data-structure-and-algorithm-tutorials/>

MDN Web Docs. (s.f.). *Pila de llamadas (Call Stack)*. Mozilla Foundation. Recuperado el 17 de octubre de 2025, de

[https://developer.mozilla.org/es/docs/Glossary/Call\\_stack](https://developer.mozilla.org/es/docs/Glossary/Call_stack)

Wikipedia. (2025, febrero 14). *Stack trace*. En *Wikipedia, La enciclopedia libre*.

Recuperado el 17 de octubre de 2025, de [https://es.wikipedia.org/wiki/Stack\\_trace](https://es.wikipedia.org/wiki/Stack_trace)

Wikipedia. (2025, marzo 2). *Pila de llamadas*. En *Wikipedia, La enciclopedia libre*.

Recuperado el 17 de octubre de 2025, de

[https://es.wikipedia.org/wiki/Pila\\_de\\_llamadas](https://es.wikipedia.org/wiki/Pila_de_llamadas)

WSCubeTech. (s.f.). *Stack Data Structure: Examples, Uses, Implementation, More*.

Recuperado el 17 de octubre de 2025, de

<https://www.wscubetech.com/resources/dsa/stack-data-structure>