

# Tipos de Datos Abstractos (TDA)

---

## ¿Qué es un Tipo de Dato Abstracto (TDA)?

Un Tipo de Dato Abstracto (TDA) es:

- **Un tipo de dato creado por el programador**  
No viene dado por el lenguaje. El programador decide cómo se llama y qué representa.
- **No pertenece a los tipos básicos del lenguaje**  
No es `int`, `double`, `char`, etc.  
Se construye a partir de ellos.
- **Modela entidades del mundo real o conceptual**  
Representa cosas como un punto, una fecha, una cuenta, una pila, etc.
- **Combina dos aspectos inseparables:**
  - **Información (datos):** lo que el TDA guarda
  - **Comportamiento (operaciones):** lo que se puede hacer con eso que guarda

---

### ◆ Significado de “abstracto”

La palabra **abstracto** indica que:

- **Importa qué hace el TDA**
- **No importa cómo está implementado internamente**

Esto significa que:

- El usuario del TDA solo necesita saber **qué operaciones existen**
- No necesita saber **cómo están guardados los datos ni cómo funcionan por dentro**

---

## Comparación con tipos primitivos

Tipo primitivo	TDA
<code>int</code> guarda un solo valor numérico	<code>Punto</code> guarda varias coordenadas
No tiene comportamiento propio	Tiene operaciones asociadas
Forma fija	Forma definida por el programador

Un tipo primitivo:

- Es simple

- Tiene una función muy limitada

Un TDA:

- Agrupa varios datos
  - Tiene reglas de uso
  - Tiene sentido por lo que permite hacer
- 

## Definición formal de un TDA

**TDA = Representación + Operaciones**

- **Representación:** los datos que forman el TDA
- **Operaciones:** las acciones posibles sobre esos datos

Ambas partes son necesarias.

Sin datos no hay información.

Sin operaciones no hay uso.

---

### ◆ Representación

La **representación** describe:

- **Qué datos contiene el TDA**  
Es decir, qué información guarda.
  - **Cómo se almacenan internamente**  
Cómo están organizados dentro del programa.
- 

### Características de la representación

- **Es privada**  
Solo el propio TDA puede acceder directamente a sus datos.
  - **No debe ser accesible directamente**  
El usuario no puede modificar los datos sin pasar por operaciones.
  - **Puede usar múltiples tipos primitivos**  
Puede combinar enteros, textos, etc.
- 

### En Java

- La representación se implementa mediante **atributos privados**
- Estos atributos forman el “estado” del TDA

---

## Ejemplo conceptual

```
Punto = { x, y, z, dimensión }
```

Esto indica:

- Qué datos existen
  - No cómo se usan todavía
- 

## ◆ Operaciones

Las **operaciones** definen:

- **Qué se puede hacer con los datos**
- **Cómo interactúa el usuario con el TDA**

Son la única forma correcta de usar la representación.

---

## Características de las operaciones

- **Son públicas**  
Están disponibles para el usuario.
  - **Manipulan la representación**  
Acceden y modifican los datos internos.
  - **Mantienen la coherencia del TDA**  
Evitan estados incorrectos o inconsistentes.
- 

## En Java

- Se implementan mediante **métodos públicos**
  - El usuario solo puede actuar a través de ellos
- 

## Ejemplos de operaciones

- Obtener coordenadas
- Modificar valores
- Desplazar el punto
- Consultar la dimensión

Cada operación tiene un propósito claro y controlado.

---

## ◆ Separación clave: “qué” vs “cómo”

Este es un principio central del TDA.

Usuario del TDA	Implementador del TDA
Usa las operaciones	Define la lógica
No ve los atributos	Decide la estructura
No sabe cómo funciona	Garantiza consistencia

El usuario:

- Confía en que el TDA funciona correctamente
- No necesita conocer los detalles internos

El implementador:

- Controla cómo se guardan los datos
- Asegura que las operaciones sean correctas

👉 Esto permite **cambiar la implementación sin afectar al usuario**, siempre que las operaciones sigan siendo las mismas.

---

La **POO** es el medio práctico para implementar un TDA.

En Java:

TDA  $\Leftrightarrow$  Clase

Esto significa que:

- Cada TDA se representa como una clase
  - La clase materializa la idea abstracta del TDA
-

## **Concepto TDA      Java**

Tipo abstracto    Clase

Representación    Atributos

Operaciones    Métodos

Interfaz            Métodos públicos

Implementación    Código privado

---

## **Aclaración**

- No toda clase es un TDA
- Pero todo TDA en Java se implementa como una clase

Una clase puede existir sin modelar un TDA completo.

---

## **◆ Interfaz vs Implementación**

---

### **◆ Interfaz pública**

La **interfaz pública** es:

- Lo que el usuario ve
- Lo que el usuario puede usar
- La forma correcta de interactuar con el TDA

Incluye:

- **Métodos públicos**
  - **Constructores**
- 

### **Ejemplo de uso desde afuera**

- Crear un punto
- Obtener X
- Mover el punto

El usuario solo conoce estas acciones.

---

## Implementación privada

La **implementación privada** es:

- Lo que el usuario no ve
- Lo que define el funcionamiento interno
- Donde se toman las decisiones técnicas

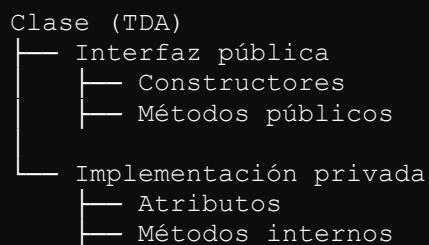
Incluye:

- **Atributos**
- **Métodos auxiliares**
- **Detalles de almacenamiento**

Todo esto está oculto.

---

## Estructura conceptual completa



→ El usuario **confía en el comportamiento**, no en la forma interna.

---

## Especificación de un TDA

Antes de programar, un TDA debe estar **claramente definido**.

Siempre consta de **dos partes obligatorias**.

---

### Descripción del conjunto de datos

Define:

- Qué elementos forman el TDA
- Qué información almacena

En POO:

- Corresponde a los **atributos**
- 

## Ejemplo

```
Punto = { x: entero, y: entero, z: entero, dimensión: texto }
```

---

## Descripción de las operaciones

Define:

- Qué acciones están permitidas
- Sobre qué datos actúan
- Qué resultados producen

En POO:

- Corresponde a los **métodos**
- 

## Ejemplo

```
getX() → entero  
mover(dx, dy)  
mover(dx, dy, dz)
```

Esto describe **cómo se puede usar el TDA**.

---

“ Un TDA no se define por sus variables,  
sino por las operaciones que permite ”

Las variables existen para que las operaciones tengan sentido.

---

Las estructuras clásicas se entienden primero como **TDA**.

Estructura	Operaciones
Pila	insertar, eliminar, tope
Cola	encolar, desencolar
Lista	insertar, borrar, recorrer
Árbol	agregar, buscar, recorrer

- 👉 Primero se define **qué operaciones existen**,
- 👉 luego se decide **cómo se implementan internamente**.

## Ejemplo de TDA: Punto

Representa un punto en el plano 2D o 3D

---

## Paso 1: Implementación del TDA (Clase Punto)

```
public class Punto {  
  
    // ===== REPRESENTACIÓN (datos) =====  
    private int x;  
    private int y;  
    private int z;  
    private String dimension;  
  
    // ===== CONSTRUCTORES =====  
  
    // Punto 2D  
    public Punto(int x, int y) {  
        this.x = x;  
        this.y = y;  
        this.z = 0;  
        this.dimension = "2D";  
    }  
  
    // Punto 3D  
    public Punto(int x, int y, int z) {  
        this.x = x;  
        this.y = y;  
        this.z = z;  
        this.dimension = "3D";  
    }  
  
    // Punto vacío (origen)  
    public Punto() {  
        this.x = 0;  
        this.y = 0;  
        this.z = 0;  
        this.dimension = "2D";  
    }  
  
    // ===== OPERACIONES (métodos) =====  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    public int getZ() {  
        return z;  
    }  
  
    public String getDimension() {  
        return dimension;  
    }  
}
```

```

public void setX(int x) {
    this.x = x;
}

public void setY(int y) {
    this.y = y;
}

public void setZ(int z) {
    this.z = z;
    this.dimension = "3D";
}

// Desplazar el punto
public void mover(int dx, int dy) {
    this.x += dx;
    this.y += dy;
}

public void mover(int dx, int dy, int dz) {
    this.x += dx;
    this.y += dy;
    this.z += dz;
    this.dimension = "3D";
}
}

```

**Este es un TDA completo:**

- Datos encapsulados
  - Operaciones bien definidas
  - El usuario no accede directamente a los atributos
- 

## Paso 2: Uso del TDA desde una clase principal

```

public class Main {
    public static void main(String[] args) {

        // Crear un punto 3D
        Punto p = new Punto(6, 4, 8);

        System.out.println("Dimensión: " + p.getDimension());
        System.out.println("X: " + p.getX());
        System.out.println("Y: " + p.getY());
        System.out.println("Z: " + p.getZ());

        // Mover el punto
        p.mover(2, -1, 3);

        System.out.println("Después de mover:");
        System.out.println("X: " + p.getX());
        System.out.println("Y: " + p.getY());
        System.out.println("Z: " + p.getZ());
    }
}

```

---

Teoría	Código
Conjunto de datos	x, y, z, dimension
Operaciones	get, set, mover()
Encapsulamiento	private
Interfaz pública	métodos public
Implementación	lógica interna
TDA definido por el programador	class Punto

---

“ Un TDA no es solo datos,  
es **datos + reglas de uso** ”

En Java:

- **Clase = TDA**
- **Objeto = instancia del TDA**
- **Métodos = forma correcta de usar los datos**

## Uso de los Tipos de Datos Abstractos (TDA)

---

Los **Tipos de Datos Abstractos** se utilizan cuando:

- Se necesita trabajar con **conjuntos de datos**
- Los datos **no son simples**
- Se requieren **operaciones específicas** sobre esos datos
- El tamaño o la forma de los datos **puede cambiar**

Por eso los TDA son la base de las **estructuras de datos**.

---

## Uso de TDA en Conjuntos

¿Qué es un **conjunto** en informática?

Un **conjunto** es una **colección de elementos** donde:

- No importa el orden
- No se repiten elementos
- Importa **qué elementos pertenecen y cuáles no**

Ejemplos:

Un conjunto de alumnos aprobados, un conjunto de figuras geométricas, un conjunto de números primos.

Aplicaciones típicas:

- Unión
- Intersección
- Diferencia

Aunque los elementos puedan ser parecidos (por ejemplo, figuras geométricas), la diferencia está en sus **propiedades internas** (lados, forma, etc.).

.

---

#### ◆ **¿Por qué los elementos “parecidos” no son iguales?**

Aunque los elementos puedan parecer similares, internamente **no lo son**.

Ejemplo con figuras geométricas:

- Un **cuadrado** tiene:
  - 4 lados iguales
  - 4 ángulos rectos
- Un **rectángulo** tiene:
  - 4 lados
  - Lados opuestos iguales
  - 4 ángulos rectos

Desde afuera pueden verse parecidos, pero **sus propiedades internas son distintas**.

El conjunto no solo guarda “objetos”, guarda **objetos con estructura interna**.

---

#### **¿Por qué se usa un TDA para representar conjuntos?**

Se utiliza un **Tipo de Dato Abstracto** porque:

##### **1 Hay un conjunto de datos**

- Elementos del mismo tipo lógico
- Con reglas claras (sin repetidos, sin orden)

##### **2 Hay operaciones bien definidas**

- Unión
- Intersección
- Diferencia
- Pertenencia (¿está o no está?)

Estas operaciones forman parte del **contrato del TDA**.

---

## Ventaja clave del TDA en conjuntos

El usuario del conjunto:

- **No necesita saber cómo se guardan los elementos**
- Solo necesita saber **qué operaciones puede hacer**

Por ejemplo:

- El conjunto puede implementarse con:
  - Un arreglo
  - Una lista
  - Un árbol
- Pero las operaciones siguen siendo las mismas

**Eso es abstracción.**

---

- ✓ Un conjunto es más que una lista de cosas
- ✓ Lo importante son las **operaciones entre conjuntos**
- ✓ El TDA permite definir:
  - Qué es un conjunto
  - Qué se puede hacer con él sin preocuparse por cómo está implementado

## ◆ **Listas Enlazadas (TDA dinámico)**

Una **lista enlazada** es un TDA que:

- Almacena una secuencia de elementos
- Permite **agregar y eliminar** elementos dinámicamente
- No tiene tamaño fijo

Comparación conceptual:

Vector	Lista enlazada
Tamaño fijo	Tamaño dinámico
Memoria reservada	Memoria en ejecución
Difícil de crecer	Fácil de modificar

---

## Usos típicos

- Procesadores de texto
  - Bases de datos
  - Manejo dinámico de información
- 

## ◆ Colas (Queue)

Una **cola** es un TDA con una regla clara:

### **FIFO – First In, First Out**

Características:

- El primero que entra es el primero que sale
- Se atiende por orden de llegada

Ejemplos reales:

- Filas de atención
  - Cajeros
  - Casetas de peaje
  - Simulaciones de turnos
- 

## ◆ Pilas (Stack)

Una **pila** es un TDA con la regla opuesta:

### **LIFO – Last In, First Out**

Características:

- El último elemento agregado es el primero en salir
- Solo se accede al elemento superior

Usos típicos:

- Evaluación de expresiones aritméticas
  - Deshacer acciones
  - Manejo de llamadas internas
- 

## ◆ Árboles

Un **árbol** es un TDA que:

- Organiza los datos de forma jerárquica
- Tiene un orden lógico
- Facilita la búsqueda y el ordenamiento

Aplicaciones:

- Búsqueda en bases de datos
  - Organización de información
  - Estructuras ordenadas
- 

## ◆ Grafos

Un **grafo** es un TDA que:

- Representa conexiones entre elementos
- Permite analizar caminos posibles

Aplicaciones:

- Transporte
  - Rutas más cortas
  - Optimización de costos
  - Resolución de problemas matemáticos
- 

## ◆ Otros usos de los TDA

Los TDA también se usan para:

- Números complejos
  - Modelos matemáticos
  - Entidades abstractas que no existen como tipo primitivo
-

**Cada estructura (listas, pilas, colas, árboles, grafos) es un TDA**  
Primero se define **qué operaciones permite**  
Luego se decide **cómo se implementa**

## ◆ TDA: Pila

---

### ¿Qué es una Pila?

Una **pila** es un **Tipo de Dato Abstracto (TDA)** que representa:

- Una **colección de elementos**
- Donde el **acceso a los datos está estrictamente controlado**

No se puede acceder libremente a cualquier elemento.  
Solo existe **una única zona de acceso**.

---

### Regla fundamental de la Pila

**LIFO**

**Last In, First Out**

(El último elemento en entrar es el primero en salir)

Esto significa que:

- El orden de salida depende del orden de entrada
  - El elemento más reciente tiene prioridad
- 

### Ejemplos

- **Pila de platos**

El último plato que se coloca arriba es el primero que se retira.

- **Pila de libros**

Para sacar un libro del medio, primero hay que quitar los de arriba.

- **Historial de deshacer (Undo)**

La última acción realizada es la primera que se deshace.

---

En programación, una pila se usa cuando:

- Importa el orden temporal de las acciones
- Se necesita revertir operaciones
- El acceso arbitrario a los datos no está permitido

Ejemplos típicos:

- Evaluación de expresiones
  - Manejo de llamadas a funciones
  - Análisis de paréntesis
- 

## Característica esencial de la Pila

En una pila:

- **Solo se puede agregar un elemento en un extremo**
- **Solo se puede quitar un elemento del mismo extremo**

Ese extremo único se denomina:

### Tope de la pila

---

El usuario **no puede**:

- Acceder a elementos intermedios
- Eliminar elementos del fondo
- Modificar el orden interno

La pila impone reglas de acceso muy estrictas.

---

Visualmente, una pila se puede pensar así:

```
|   e3   | ← tope
|   e2   |
|   e1   |
-----
```

- $e_3$  es el último elemento agregado
  - $e_1$  es el primero que entró
  - Solo  $e_3$  puede ser accedido directamente
- 

## Ejemplo en Java (uso)

Supongamos una pila de enteros:

```
Pila p = new Pila(5);  
p.apilar(10);  
p.apilar(20);  
p.apilar(30);
```

Estado interno conceptual:

```
| 30 | ← tope  
| 20 |  
| 10 |  
-----
```

Si se ejecuta:

```
p.desapilar();
```

Resultado:

- Se elimina el 30
  - El nuevo tope es 20
- 

## Qué NO permite una Pila (ejemplo)

Esto **no** es válido conceptualmente:

```
p.obtenerElemento(1); // ✗ No permitido  
p.eliminarFondo(); // ✗ No permitido
```

Estas operaciones romperían el modelo del TDA Pila.

---

Se usa una pila como **TDA** porque:

- Existe un conjunto de datos
  - Existen operaciones bien definidas
  - El usuario conoce **qué puede hacer**
  - El usuario **no conoce cómo se guarda**
- 

“ Una pila no es un arreglo,  
es una regla de acceso a los datos ”

# Operaciones típicas del TDA Pila

Una pila se define por sus **operaciones**, no por cómo se guarda.

Operaciones fundamentales:

- **apilar** (push)  
→ Agrega un elemento en el tope
  - **desapilar** (pop)  
→ Quita el elemento del tope
  - **tope** (peek)  
→ Devuelve el elemento del tope sin quitarlo
  - **estaVacia**  
→ Indica si la pila no tiene elementos
  - **estaLlena**  
→ Indica si la pila alcanzó su capacidad  
(esto es relevante en implementación estática)
- 

## Especificación del TDA Pila

La **especificación** de un TDA define:

- **Qué es la pila**
- **Qué operaciones permite**
- **Qué condiciones deben cumplirse**

La especificación **no habla de código**,  
solo describe el **comportamiento esperado**.

---

## Descripción del conjunto de datos

Esta parte define **qué datos contiene la pila**.

Define:

- **Qué tipo de elementos almacena**
- **Cuántos elementos puede contener como máximo**

Ejemplo:

```
Pila = { e1, e2, e3, ..., en }
```

Donde:

- $e_1$  es el primer elemento que entró
  - $e_n$  es el último elemento agregado (tope)
- 

## Condiciones del conjunto de datos

En una pila siempre se cumple que:

- **Todos los elementos son del mismo tipo lógico**  
(por ejemplo: todos enteros, todos objetos `String`, etc.)
  - **Existe un orden interno**  
(desde la base hasta el tope)
  - **El acceso está restringido al tope**  
No se puede acceder directamente a otros elementos
- 

## Ejemplo en Java

Si la pila es de enteros:

```
Pila<Integer> pila;
```

Los datos almacenados podrían ser:

```
Pila = { 5, 12, 8 }
```

Donde:

- 5 es la base
- 8 es el tope

El usuario **solo puede interactuar con el 8.**

---

## Descripción de las operaciones

Esta parte define **qué acciones están permitidas** sobre la pila.

Cada operación indica:

- Qué hace
- En qué condiciones se puede usar

---

### ◆ apilar(x)

- Inserta el elemento  $x$  en el **tope de la pila**
- El nuevo elemento pasa a ser el último

Condición:

- La pila **no debe estar llena**

Ejemplo en Java:

```
pila.apilar(20);
```

Antes:

```
{ 5, 12 }
```

Después:

```
{ 5, 12, 20 }
```

---

### ◆ desapilar()

- Elimina y devuelve el elemento del **tope**
- Reduce la cantidad de elementos

Condición:

- La pila **no debe estar vacía**

Ejemplo:

```
int x = pila.desapilar();
```

Si la pila era:

```
{ 5, 12, 20 }
```

Resultado:

- $x = 20$
  - La pila queda { 5, 12 }
- 

### ◆ tope()

- Devuelve el elemento del tope
- **No modifica** la pila

Ejemplo:

```
int x = pila.tope();
```

La pila:

```
{ 5, 12, 20 }
```

Resultado:

- $x = 20$
  - La pila **no cambia**
- 

#### ◆ **estaVacia()**

- Devuelve `true` si no hay elementos
- Devuelve `false` en caso contrario

Ejemplo:

```
if (pila.estaVacia()) {  
    System.out.println("Pila vacía");  
}
```

---

#### ◆ **estaLlena()**

- Devuelve `true` si alcanzó su capacidad máxima
- Solo tiene sentido en implementación estática

Ejemplo:

```
if (pila.estaLlena()) {  
    System.out.println("Pila llena");  
}
```

---

☞ Estas operaciones **definen completamente el TDA Pila**.

---

## Separación clave: qué vs cómo

### Usuario del TDA Implementador del TDA

Usa `apilar()` Decide cómo guardar

Usa `desapilar()` Controla límites

No ve estructura Mantiene coherencia

El usuario confía en el **comportamiento**,  
no en la **estructura interna**.

---

## Implementación estática de una Pila

La implementación estática es **una forma concreta** de realizar el TDA.

Características:

- Capacidad fija
- Tamaño máximo definido previamente
- No cambia durante la ejecución

La **especificación del TDA no cambia**,  
solo cambia **cómo se implementa**.

---

## Representación en la implementación estática de una Pila

En una **implementación estática** del TDA Pila, la representación interna se basa en **dos elementos fundamentales**:

- Un **arreglo de tamaño fijo**
- Una **variable que indica el tope**

Estos dos elementos son suficientes para modelar completamente el comportamiento de una pila.

---

## El arreglo de tamaño fijo

El arreglo:

- Es el espacio donde se almacenan los elementos
- Tiene una **capacidad máxima definida al crearse**
- No puede crecer ni reducirse durante la ejecución

Ejemplo en Java:

```
int[] datos = new int[5];
```

Esto significa:

- La pila puede almacenar **como máximo 5 elementos**
- Las posiciones válidas del arreglo son 0 a 4
- Al inicio, el arreglo existe, pero **no contiene elementos útiles**

Representación conceptual inicial:

```
datos = [ _ , _ , _ , _ , _ ]
```

Los guiones bajos (\_) indican posiciones vacías o sin uso lógico.

---

## La variable **tope**

La variable **tope**:

- Indica **dónde está el último elemento agregado**
- Marca el límite entre datos válidos y espacio libre
- Controla el acceso a la pila

Ejemplo en Java:

```
int tope = -1;
```

¿Por qué -1?

- Porque no hay ningún elemento en el arreglo
- No existe ninguna posición válida ocupada
- Es una forma clara de indicar **pila vacía**

Representación inicial completa:

```
datos = [ _ , _ , _ , _ , _ ]
tope = -1
```

---

## Significado conjunto de **datos** y **tope**

Ambos elementos trabajan **siempre juntos**:

- **datos** guarda los valores
- **tope** indica **hasta dónde mirar**

Ejemplo:

Si **tope = 2**, entonces:

```
datos = [ 10 , 25 , 8 , _ , _ ]
          0      1      2
tope = 2
```

Significa:

- Hay **3 elementos en la pila**
  - El elemento del tope es 8
  - Las posiciones 3 y 4 no forman parte de la pila
- 

## Operaciones y representación

Cada operación del TDA Pila se traduce en **acciones concretas** sobre `datos` y `tope`.

---

### ◆ Operación apilar(x)

Conceptualmente:

- Agrega un elemento al **tope de la pila**

Internamente:

- Incrementa `tope`
- Guarda el valor en `datos[tope]`

Ejemplo en Java:

```
datos[++tope] = x;
```

Ejemplo paso a paso:

Estado inicial:

```
datos = [ _ , _ , _ , _ , _ ]
tope = -1
```

Se ejecuta:

```
apilar(15);
```

Estado resultante:

```
datos = [ 15 , _ , _ , _ , _ ]
tope = 0
```

---

### Operación desapilar()

Conceptualmente:

- Quita el elemento del **tope**

Internamente:

- Obtiene `datos[tope]`
- Decrementa `tope`

Ejemplo en Java:

```
int valor = datos[tope--];
```

Ejemplo:

Estado inicial:

```
datos = [ 15 , 30 , 50 , _ , _ ]
tope   = 2
```

Se ejecuta:

```
int x = desapilar();
```

Resultado:

- `x = 50`

Estado final:

```
datos = [ 15 , 30 , 50 , _ , _ ]
tope   = 1
```

El valor sigue en el arreglo, pero **ya no pertenece a la pila**.

---

## Operación `estaVacia()`

Conceptualmente:

- Verifica si la pila no tiene elementos

Condición interna:

```
tope == -1
```

Ejemplo:

```
if (tope == -1) {
    System.out.println("La pila está vacía");
}
```

---

## Operación `estaLlena()`

Conceptualmente:

- Verifica si no se pueden agregar más elementos

Condición interna:

```
tope == datos.length - 1
```

Ejemplo:

```
if (tope == datos.length - 1) {  
    System.out.println("La pila está llena");  
}
```

---

**El usuario nunca ve estas reglas internas.**

Solo utiliza los métodos públicos del TDA.

---

## Ventajas de la implementación estática

- Estructura simple y clara
- Operaciones muy rápidas
- Fácil de implementar y depurar
- Bajo consumo de recursos

Es ideal para:

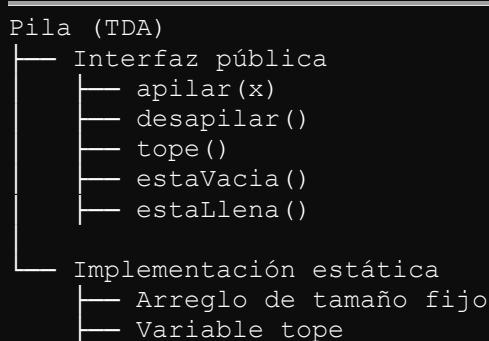
- Ejercicios iniciales
  - Sistemas con límites conocidos
  - Casos donde el tamaño máximo es predecible
- 

## Limitaciones de la implementación estática

- El tamaño no puede cambiar
- Riesgo de desbordamiento si se excede la capacidad
- Puede desperdiciar memoria si se usa poco

Estas limitaciones corresponden al **cómo**,  
no al **qué** del TDA Pila.

---



- El **comportamiento externo** es estable, aunque la **implementación interna** cambie.

## Clase PilaEstatica (implementación TDA)

```
public class PilaEstatica {  
  
    // Arreglo de tamaño fijo que almacena los elementos de la pila  
    private int[] datos;  
  
    // Índice del último elemento insertado  
    // tope = -1 indica pila vacía  
    private int tope;  
  
    // Constructor: crea una pila vacía con capacidad fija  
    public PilaEstatica(int capacidad) {  
        datos = new int[capacidad];  
        tope = -1;  
    }  
  
    // Verifica si la pila no contiene elementos  
    public boolean estaVacia() {  
        return tope == -1;  
    }  
  
    // Verifica si la pila alcanzó su capacidad máxima  
    public boolean estaLlena() {  
        return tope == datos.length - 1;  
    }  
  
    // Inserta un elemento en el tope de la pila  
    public void apilar(int x) {  
        if (estaLlena()) {  
            System.out.println("Error: pila llena");  
            return;  
        }  
        datos[++tope] = x;  
    }  
  
    // Elimina y devuelve el elemento del tope  
    public int desapilar() {  
        if (estaVacia()) {  
            System.out.println("Error: pila vacía");  
            return -1;  
        }  
        return datos[tope--];  
    }  
}
```

```

// Devuelve el elemento del tope sin eliminarlo
public int tope() {
    if (estaVacia()) {
        System.out.println("Pila vacía");
        return -1;
    }
    return datos[tope];
}

```

- El usuario **nunca ve** el arreglo ni el índice tope
  - Solo interactúa mediante apilar, desapilar, tope, etc.
  - Esto garantiza el **comportamiento LIFO** del TDA Pila
- 

## Uso del TDA Pila (desde otra clase)

El usuario **solo interactúa con la interfaz.**

```

public class Main {
    public static void main(String[] args) {

        PilaEstatica pila = new PilaEstatica(5);

        pila.apilar(10);
        pila.apilar(20);
        pila.apilar(30);

        System.out.println(pila.tope());      // 30
        System.out.println(pila.desapilar()); // 30
        System.out.println(pila.tope());      // 20
    }
}

```

El usuario:

- No sabe cómo se guarda
  - No accede al arreglo
  - Confía en el comportamiento
- 

## ¿Por qué esto es un TDA y no solo una clase?

Porque:

- Los atributos son **privados**
- Las operaciones están **controladas**
- El acceso a los datos está **restringido**
- El comportamiento es estable

**Una clase cualquiera puede no ser un TDA,  
pero esta clase sí lo es.**

---

## **Si cambiamos la implementación...**

Podríamos usar:

- Lista enlazada
- Arreglo dinámico
- Otra estrategia

**La interfaz no cambia  
El TDA sigue siendo el mismo**

## **TDA: Cola**

### **Especificación e implementación estática**

---

La regla fundamental que la define es:

#### **FIFO (First In, First Out)**

El primer elemento en entrar es el primero en salir.

El comportamiento es similar a una fila de espera: las personas se agregan al final y son atendidas por el frente.

En una cola:

- Los elementos se insertan por un extremo (final).
- Los elementos se eliminan por el extremo opuesto (frente).

El usuario no puede acceder a elementos intermedios ni alterar el orden.

---

## **2. Definición formal del TDA Cola**

Un TDA Cola se define por:

**Cola = Representación + Operaciones**

La definición no depende del código ni del lenguaje, sino del comportamiento que se espera.

---

### 3. Especificación del TDA Cola

#### 3.1 Descripción del conjunto de datos

La cola almacena una secuencia de elementos del mismo tipo lógico.

Formalmente puede representarse como:

$$\text{Cola} = \{ e_1, e_2, e_3, \dots, e_n \}$$

Donde:

- $e_1$  es el primer elemento en entrar (frente).
  - $e_n$  es el último elemento agregado (final).
  - El orden interno es significativo.
  - El acceso está restringido al frente y al final.
- 

#### 3.2 Descripción de las operaciones

La especificación define las operaciones permitidas sobre la cola:

- `encolar(x)`  
Inserta el elemento  $x$  al final de la cola.  
Requiere que la cola no esté llena.
- `desencolar()`  
Elimina y devuelve el elemento del frente.  
Requiere que la cola no esté vacía.
- `frente()`  
Devuelve el elemento del frente sin eliminarlo.
- `estaVacia()`  
Devuelve verdadero si la cola no contiene elementos.
- `estaLlena()`  
Devuelve verdadero si la cola alcanzó su capacidad máxima.

Estas operaciones definen completamente el comportamiento del TDA Cola.

---

### 6. Representación en una implementación estática de Cola

En Java, una **cola estática** se implementa utilizando estructuras simples, ya que su tamaño máximo se define al momento de crearla y no cambia durante la ejecución del programa.

Para representar una cola estática se utilizan:

- Un arreglo de tamaño fijo, que almacena los elementos.
- Dos índices enteros, que indican las posiciones relevantes dentro del arreglo.
- Una variable adicional que controla cuántos elementos hay realmente en la cola.

Representación conceptual:

```
int[] datos = new int[5];
int frente = 0;
int fin = -1;
int cantidad = 0;
```

Significado de cada componente:

- El arreglo `datos` es el espacio físico donde se guardan los elementos de la cola.
- La variable `frente` indica la posición del elemento que será el próximo en salir.
- La variable `fin` indica la posición del último elemento que fue insertado.
- La variable `cantidad` indica cuántos elementos hay actualmente en la cola.

Cuando la cola está vacía:

- `cantidad` vale 0.
  - `frente` apunta a una posición válida, pero no hay elementos.
  - `fin` vale -1, indicando que aún no se ha insertado ningún elemento.
- 

## 7. Relación entre las operaciones y la representación interna

Cada operación del TDA Cola se implementa manipulando estas variables internas, aunque el usuario del TDA nunca interactúa directamente con ellas.

### **encolar(x)**

- Avanza el índice `fin`.
- Almacena el valor `x` en `datos[fin]`.
- Incrementa la variable `cantidad`.

Esto representa lógicamente que un nuevo elemento se agrega al final de la cola.

### **desencolar()**

- Obtiene el valor almacenado en `datos[frente]`.
- Avanza el índice `frente`.
- Decrementa la variable `cantidad`.

De esta forma, el elemento que estaba primero es el que se elimina, respetando el orden FIFO.

### estaVacia()

- Se cumple cuando `cantidad == 0`.
- No depende de los índices, sino del número real de elementos.

### estaLlena()

- Se cumple cuando `cantidad == datos.length`.
- Indica que no hay más espacio disponible en el arreglo.

Estas reglas internas garantizan el comportamiento correcto de la cola, aunque permanecen ocultas para el usuario.

---

## 8. Implementación estática del TDA Cola en Java

La siguiente clase implementa el TDA Cola respetando exactamente la especificación definida:

```
public class ColaEstatica {  
  
    private int[] datos;  
    private int frente;  
    private int fin;  
    private int cantidad;  
  
    public ColaEstatica(int capacidad) {  
        datos = new int[capacidad];  
        frente = 0;  
        fin = -1;  
        cantidad = 0;  
    }  
  
    public boolean estaVacia() {  
        return cantidad == 0;  
    }  
  
    public boolean estaLlena() {  
        return cantidad == datos.length;  
    }  
  
    public void encolar(int x) {  
        if (estaLlena()) {  
            System.out.println("Error: cola llena");  
            return;  
        }  
        fin++;  
        datos[fin] = x;  
        cantidad++;  
    }  
  
    public int desencolar() {
```

```

        if (estaVacia()) {
            System.out.println("Error: cola vacía");
            return -1;
        }
        int valor = datos[frente];
        frente++;
        cantidad--;
        return valor;
    }

    public int frente() {
        if (estaVacia()) {
            System.out.println("Cola vacía");
            return -1;
        }
        return datos[frente];
    }
}

```

Esta implementación es **estática**, ya que el tamaño del arreglo se define una sola vez y no se modifica.

---

## 9. Uso del TDA Cola

El siguiente ejemplo muestra cómo un usuario utiliza la cola sin conocer su estructura interna:

```

public class Main {
    public static void main(String[] args) {

        ColaEstatica cola = new ColaEstatica(5);

        cola.encolar(10);
        cola.encolar(20);
        cola.encolar(30);

        System.out.println(cola.frente());           // 10
        System.out.println(cola.desencolar());        // 10
        System.out.println(cola.frente());           // 20
    }
}

```

Desde el punto de vista del usuario:

- Solo existen las operaciones `encolar`, `desencolar` y `frente`.
  - No se accede al arreglo ni a los índices.
  - El comportamiento FIFO se mantiene siempre.
- 

Una **cola** no se define por el arreglo ni por los índices que usa, sino por la regla que impone:

**el primero en entrar es el primero en salir,**  
independientemente de cómo se implemente internamente.