Programación II – ALGORTIRMOS Y ESTRUCTURAS DE DATOS II

**Docente:** Esp. Lic. PEREZ, Nicolás Ignacio

[nicoperez@uade.edu.ar](mailto:nicoperez@uade.edu.ar)

Repositorio de la materia: <https://github.com/NicolasPerezUNLaSMN/PROG_II_UADE_JAVA>

# Facultad de Ingeniería y Ciencias Exactas

Universidad Argentina de la Empresa



#### Precondiciones, Postcondiciones y Axiomas en las Interfaces

En el paradigma de Tipos de Datos Abstractos (TDA) en Java, las interfaces suelen definir contratos mediante métodos que establecen reglas de uso basadas en **precondiciones, postcondiciones y axiomas**:

* **Precondiciones**: Son condiciones que deben cumplirse antes de ejecutar un método. Garantizan que la operación es válida.
* **Postcondiciones**: Son condiciones que deben cumplirse después de ejecutar el método, asegurando que la operación produjo un resultado esperado.
* **Axiomas**: Son reglas lógicas que describen el comportamiento del TDA y cómo sus operaciones deben interactuar entre sí. Por ejemplo, si stack.push(x) es seguido de stack.pop(), el resultado debe ser x.

**Por ejemplo en nuestra interface Persona sencilla, quedaría algo así:**

#### Uso de @Override

@Override es una anotación en Java utilizada para indicar que un método está sobrescribiendo una versión de un método en una superclase o interfaz.

* **Cuándo usarlo:** Siempre que se sobrescriba un método en una subclase o se implemente un método de una interfaz. Ayuda a evitar errores y mejora la legibilidad.
* **Cuándo no usarlo:** Si el método de la subclase no coincide exactamente en la firma con el de la superclase (nombre, parámetros y tipo de retorno), @Override generará un error de compilación. Tampoco es necesario si el método es completamente nuevo en la subclase.

#### Clases Anidadas y Relaciones entre Objetos

Si tenemos dos clases como Equipo y Persona, donde un Equipo tiene un capitán (una única instancia de Persona) y un conjunto de jugadores (un array de Persona), existen dos tipos de relaciones:

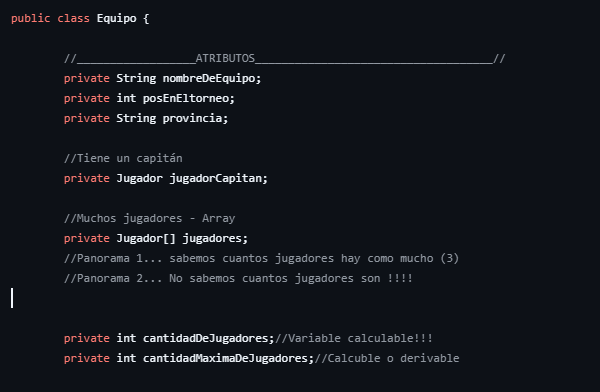
* **Composición:** El capitán es un solo objeto Persona, estrechamente ligado a Equipo. Si el equipo deja de existir, su capitán también puede desaparecer.
* **Agregación:** El array de jugadores representa una relación más flexible, ya que los jugadores pueden existir independientemente de un equipo específico.

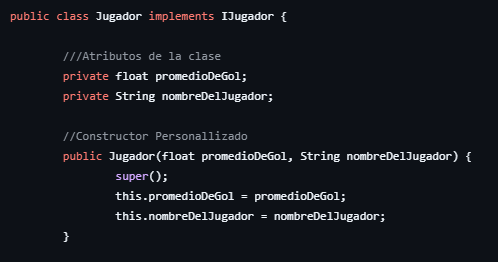
Este tipo de modelado es útil para representar jerarquías y relaciones entre entidades en un sistema.

#### 4. Relaciones entre Clases en Base a un Objeto y un Array

En el ejemplo presentado, la clase Equipo tiene dos relaciones con la clase Jugador:

* **Relación unitaria (Composición):** Equipo tiene un atributo jugadorCapitan de tipo Jugador. Esto significa que cada equipo posee exactamente un capitán. Esta relación es de **composición**, ya que el capitán pertenece exclusivamente a un equipo y depende directamente de su existencia.
* **Relación con un array (Agregación):** Equipo también posee un array de Jugador, lo que representa una relación de **agregación**. Aquí, los jugadores forman parte del equipo, pero pueden existir fuera de él, lo que indica una relación más flexible. Existen dos escenarios:
  + **Tamaño fijo:** Si el número de jugadores está predefinido (ejemplo: jugadores = new Jugador[3]), tenemos una estructura **estática**.







Ejemplo completo en:

<https://github.com/NicolasPerezUNLaSMN/PROG_II_UADE_JAVA/tree/master/Clase%20-%20II/Project1>

En las próximas clases justamente veremos otras estructuras de datos que serán útiles para anidar clases, las mismas son las siguientes (que por suerte java ya las tiene implementadas):

#### **Estructuras Estáticas**

1. **Arrays (int[], String[], etc.)** – Conjunto de elementos del mismo tipo, con tamaño fijo.
2. **Colas (Queue<T> con ArrayDeque o LinkedList)** – FIFO (First In, First Out).
3. **Colas con prioridad (PriorityQueue<T>)** – FIFO pero con prioridad en la extracción.
4. **Pilas (Stack<T>)** – LIFO (Last In, First Out).
5. **Conjuntos (Set<T> con HashSet, TreeSet)** – Estructura sin duplicados.
6. **Diccionarios (Map<K, V> con HashMap, TreeMap)** – Almacena pares clave-valor.

#### **Estructuras Dinámicas**

1. **Listas (List<T> con ArrayList, LinkedList)** – Permiten tamaño variable.
2. **Colas (Queue<T> con LinkedList, PriorityQueue)** – Implementación dinámica.
3. **Colas con prioridad (PriorityQueue<T>)** – Se ajusta dinámicamente según prioridad.
4. **Pilas (Deque<T> con ArrayDeque, LinkedList)** – Pueden crecer o reducirse.
5. **Conjuntos (Set<T> con HashSet, TreeSet)** – Implementaciones sin límite predefinido.
6. **Diccionarios (Map<K, V> con HashMap, TreeMap)** – Pueden expandirse dinámicamente.

**Cola, implementación estática:**

Ejemplo completo en:

<https://github.com/NicolasPerezUNLaSMN/PROG_II_UADE_JAVA/tree/master/Clase%20-%20III/Clase3EstructurasEstaticasI>

1. **Atributos de la Clase**

* MAX\_SIZE: Tamaño máximo de la cola (definido como constante).
* elementos: Array donde se almacenan los elementos.
* inicio: Índice del primer elemento en la cola.
* fin: Índice donde se agregará el próximo elemento.
* cantidad: Cantidad actual de elementos en la cola.

1. **Constructor e Inicialización**

* El constructor llama a inicializarCola() para asignar memoria y valores iniciales.
* inicio y fin comienzan en 0, indicando que la cola está vacía.

1. **Encolar (Agregar un Elemento)**

* Se verifica si la cola está llena (colaLlena()).
* Se inserta el elemento en elementos[fin].
* fin avanza usando **aritmética modular** ((fin + 1) % MAX\_SIZE), lo que permite un comportamiento circular.
* Se incrementa cantidad.

1. **Desacolar o Desencolar (Eliminar un Elemento)**

* Se verifica si la cola está vacía (colaVacia()).
* Se obtiene el valor en elementos[inicio].
* Se avanza inicio con aritmética modular.
* Se reduce cantidad.

1. **Métodos Auxiliares**

* colaVacia(): Retorna true si no hay elementos (cantidad == 0).
* colaLlena(): Retorna true si la cola está llena (cantidad == MAX\_SIZE).

1. **Obtener el Primer Elemento**

* Devuelve el elemento en inicio, sin eliminarlo.
* Lanza una excepción si la cola está vacía.

1. **Ejercicio de la clase**

Seguir trabajando con el TDA, Persona y Vehículo, hacer que el Vehículo además de tener un conductor tenga 2 Pasajeros.