ANALISTA DE SISTEMAS

TALLER DE PROGRAMACIÓN 1

Profesor Marco Cupo

UNIDAD 5 - PILAS, COLAS Y LISTAS ORDENADAS

ORT אורט ARGENTINA
Educando para la vida

"

Un **Tipo de dato abstracto (TDA)** es una estructura que integra un conjunto de datos (elementos) con un grupo de operaciones permitidas específicas que determinan la forma en la cual esos elementos son manipulados.

ORT ARGENTINA
Educando para la vida

- Cada TDA provee de una interfaz a través de la cual podemos realizar las operaciones permitidas, abstrayéndose de cómo estas están implementadas.
- Esto quiere decir que un mismo TDA puede ser implementado utilizando distintas estructuras de datos internas, pero proveyendo siempre la misma funcionalidad a través de su interfaz (API).

ORT ARGENTINA
Educando para la vida

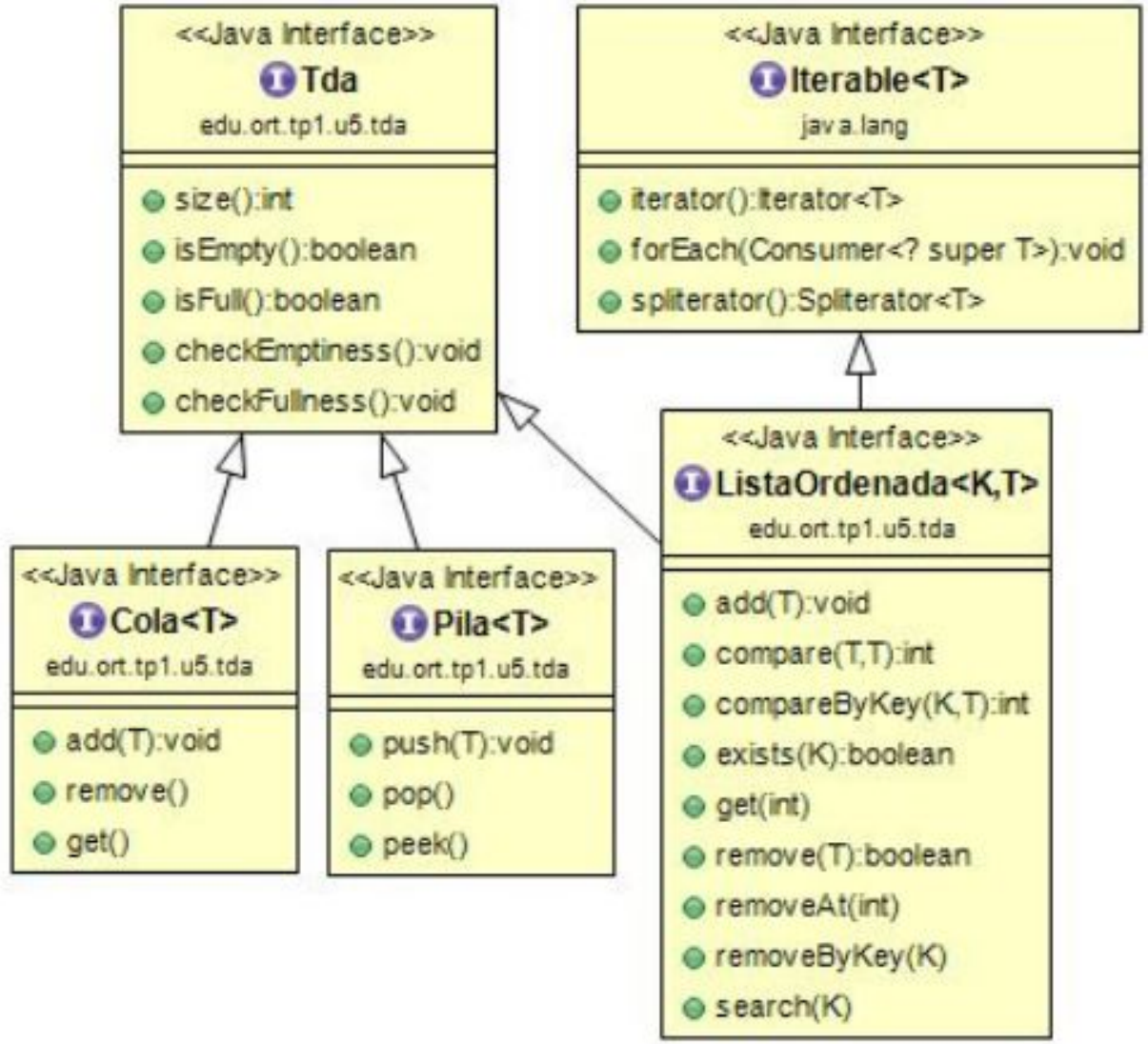
DEFINICIÓN

- Las **Pilas** son estructuras **LIFO (Last In, First Out)**, donde el último elemento que se agrega es el primero que se sacará.
- Las **Colas** son estructuras **FIFO (First In, First Out)**, donde los elementos se sacan en el mismo orden que fueron agregados.
- Un detalle que tienen en común ambas estructuras es que **no permiten acceder a ningún otro elemento contenido en el TDA, no es posible buscar ni elegir otro elemento que no sea el primero que está disponible para su extracción**.

DEFINICIÓN

- La **ListaOrdenada** (que extiende Lista) contiene una colección de objetos; son estructuras de acceso secuencial y aleatorio. No tienen un orden preestablecido para acceder a sus elementos. Mientras que la Lista permite manejarlos tal como un ArrayList (eligiendo el orden según la necesidad momentánea) la **ListaOrdenada** los mantiene ordenados según un criterio determinado por una clave y en orden ascendente o descendente.
- Más allá de mantener sus elementos ordenados o no, una clasificación muy básica de las listas puede darse respecto a la cantidad y dirección de las relaciones dadas entre sus elementos contenidos: hay **listas unidireccionales**, donde sólo se puede ir de un elemento al siguiente; **bidireccionales**, donde se puede ir tanto al siguiente como al anterior; entre varios otros tipos que no están en el alcance de nuestra materia.

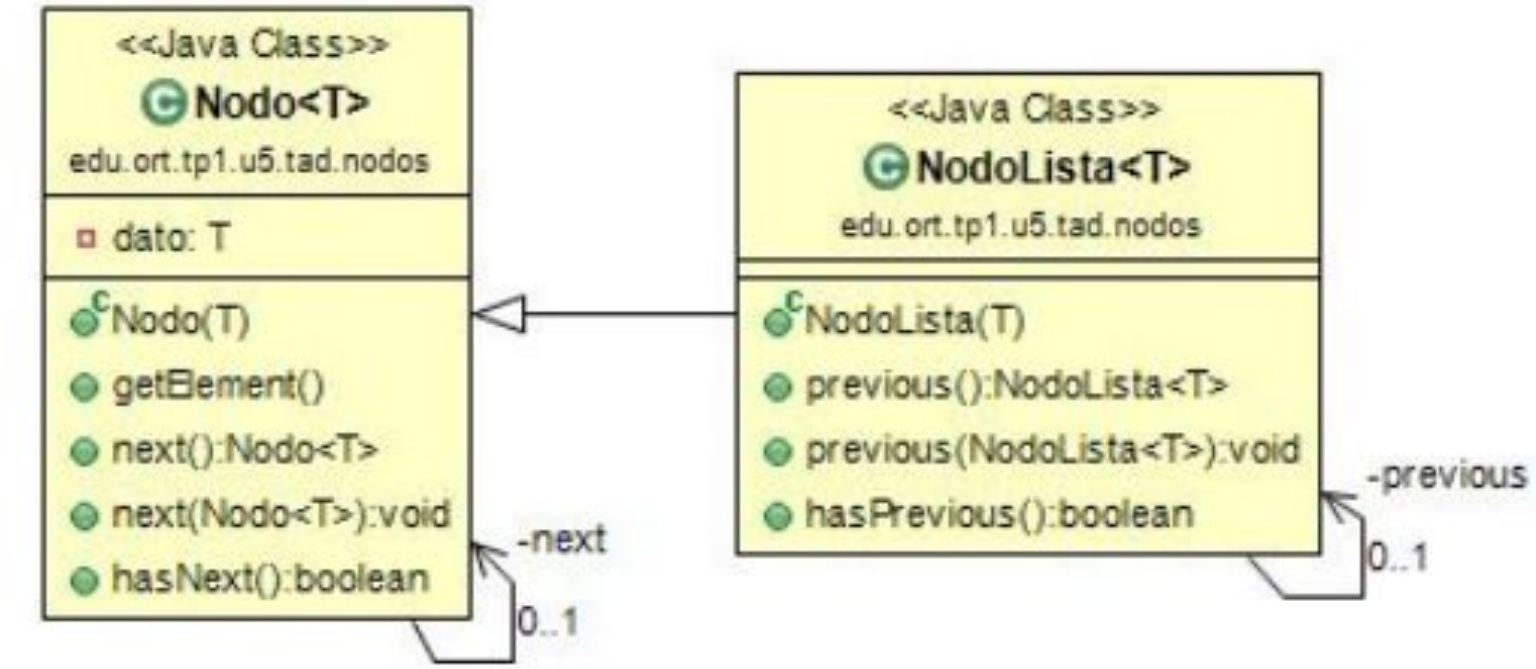
DIAGRAMA DE CLASES DE LAS INTERFACES QUE VAMOS A USAR



- Si bien estas estructuras se pueden implementar de muchas formas distintas, hemos elegido para desarrollarlas una implementación utilizando lo que se conoce como **Nodos Enlazados**.

- Un **nodo** enlazado en sí no es nada complejo: es un elemento (para nosotros un **objeto**) que tiene dos partes:
  - Una está destinada a guardar el dato (que siempre será un objeto)
  - Y al menos una referencia hacia otro elemento.

- A nivel de clases, un **nodo** puede verse como se ve a continuación. Veremos que en el diagrama de clases se muestra tanto un nodo simplemente enlazado (**Nodo**) como otro doblemente enlazado (**NodoLista**). Ambos usan **Generics** como forma de conocer qué tipo de elemento van a contener.



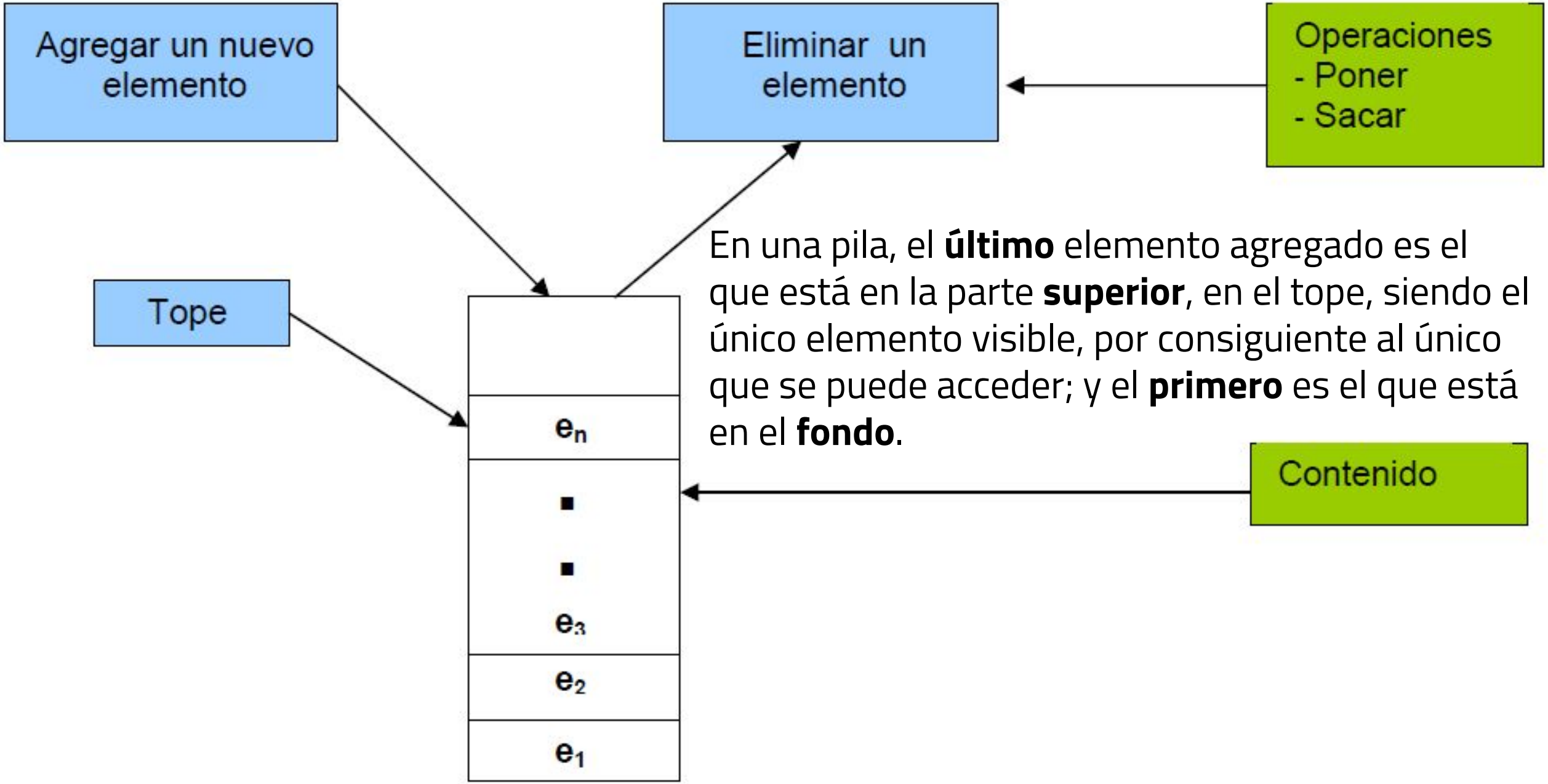
"

Una **pila** es un conjunto de elementos homogéneos (todos del mismo tipo) que solamente puede crecer o decrecer por la parte superior (llamada cabeza) de la pila, o sea, sólo por uno de sus extremos.

- Entonces podríamos decir que los elementos que se encuentran en una pila, **están ubicados de acuerdo a cuánto tiempo llevan en ella**.
- Además el orden de los elementos está relacionado con el **momento de llegada de los mismos**.

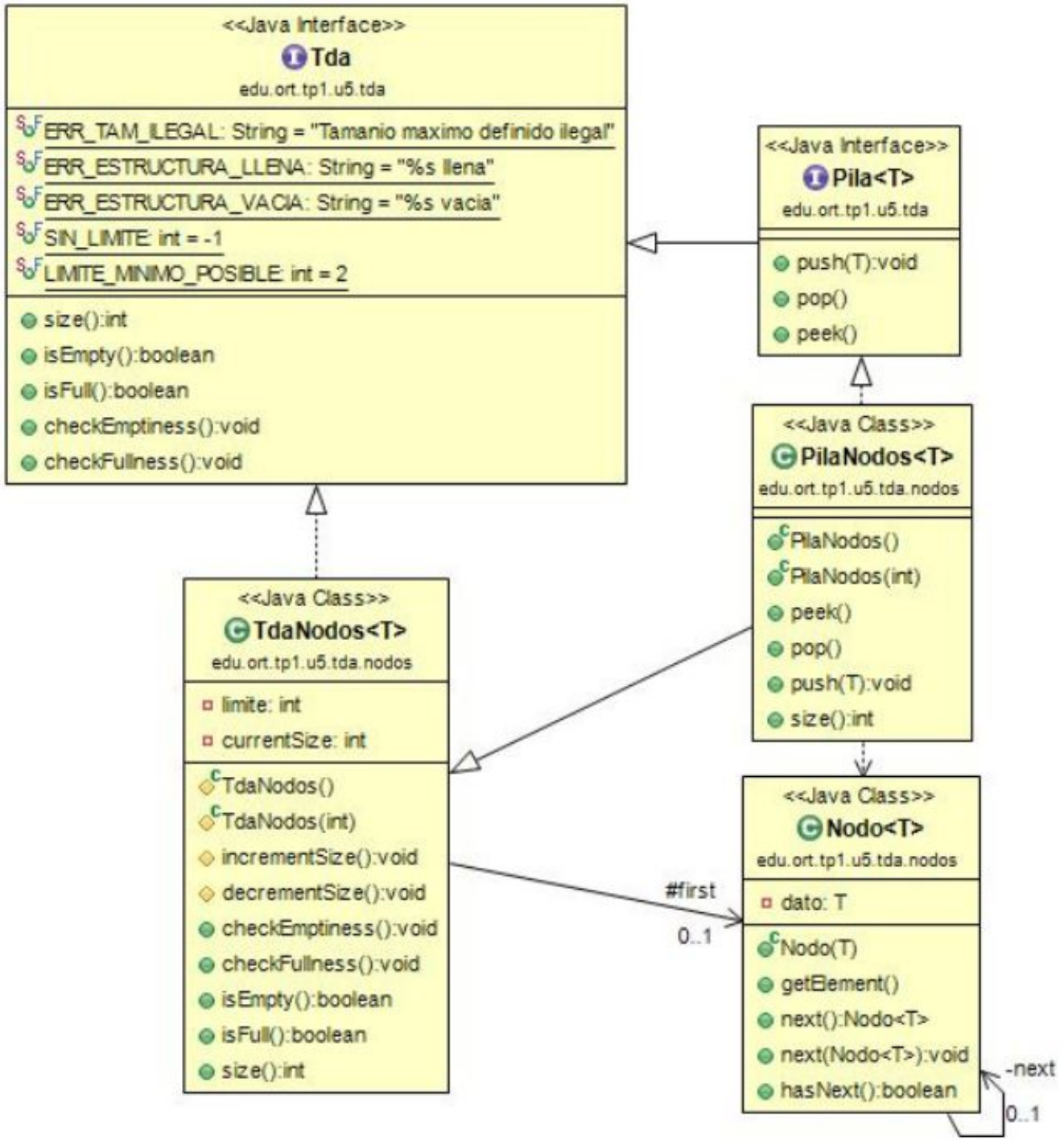
ORT ARGENTINA
Educando para la vida

DEFINICIÓN - PILAS



En una pila, el **último** elemento agregado es el que está en la parte **superior**, en el tope, siendo el único elemento visible, por consiguiente al único que se puede acceder; y el **primero** es el que está en el **fondo**.

IMPLEMENTACIÓN PARA PILA



PILAS - OPERACIONES

Constructor()	• Setea en null el valor de **first**. • Setea el **limite** para isFull() en SIN_LIMITE. • Setea **currentSize** en 0.
Constructor(int)	• Setea en null a **first**. • Setea el **limite** para isFull() con el valor del parámetro. • Setea **currentSize** en 0.
void push(TipoElemento)	Si la pila estuviese llena lanzará una excepción. Crea un nodo donde pone el elemento recibido y lo engancha delante de los previos (si los hay). Actualiza **first**. Además incrementa en 1 el valor de **currentSize**.
TipoElemento pop()	Si la pila estuviese vacía lanzará una excepción. Extrae el primer nodo de la cadena de nodos actualizando **first**, decrementa en 1 el valor de **currentSize** y devuelve el elemento guardado en el nodo extraído (el nodo en sí se descarta).
TipoElemento peek()	Si la pila estuviese vacía lanzará una excepción. Devuelve el elemento del nodo referenciado por **first**, sin extraerlo.
boolean isEmpty()	Devuelve verdadero cuando **currentSize** vale 0.
boolean isFull()	Si **limite** vale -1 (porque se usó el constructor por defecto) devolverá siempre false. Si no, su valor dependerá de la comparación entre **limite** y **currentSize**.

PILAS - EJEMPLOS - OBJETOS PRIMITIVOS

```
public static void main(String[] args) {

    //Declaro una variable de tipo Interfaz Pila que va a contener clases de tipo String
    Pila<String> pila;
    //Instancio un clase de tipo PilaNodos que implementa la interfaz Pila y por ende es una operación segura
    //Upcasting
    pila = new PilaNodos<>();

    System.out.println("Apilamos algunos elementos...");
    pila.push("Pilas");
    pila.push("en");
    pila.push("Java");
    System.out.println("Elemento en la cima: " + pila.peek());//Devuelve el elemento que esta en la cima pero no lo retira
    //System.out.println("Tamaño de la pila: " + pila.size());//Arroja una excepcion, el tamaño de la pila no es visible
    System.out.println("Desapilamos todos elementos...");
    System.out.println(pila.pop());
    System.out.println(pila.pop());
    System.out.println(pila.pop());
    //System.out.println(pila.pop());//Arroja una excepcion
    if(pila.isEmpty())//Metodo para validar si la pila esta vacia
        System.out.println("La pila esta vacia");
}
```


ORT ARGENTINA
Educando para la vida

PILAS - EJEMPLOS - OBJETOS

```
public static void main(String[] args) {

    //Declaro una variable de tipo Interfaz Pila que va a contener clases de tipo String
    Pila<Auto> pila;
    //Instancio un clase de tipo PilaNodos que implementa la interfaz Pila y por ende es una operación segura
    //Upcasting
    pila = new PilaNodos<>();

    System.out.println("Apilamos algunos elementos...");
    Auto auto1 = new Auto("ABCD123", "Fiesta", "Ford");
    Auto auto2 = new Auto("RTY443", "Corolla", "Toyota");
    pila.push(auto1);
    pila.push(auto2);
    System.out.println("Elemento en la cima: " + pila.peek());//Devuelve el elemento que esta en la cima pero no lo retira
    System.out.println("Desapilamos todos elementos...");
    System.out.println(pila.pop());
    System.out.println(pila.pop());
    //System.out.println(s.pop());//Arroja una excepcion
    if(pila.isEmpty())//Metodo para validar si la pila esta vacia
        System.out.println("La pila esta vacia");
}
```

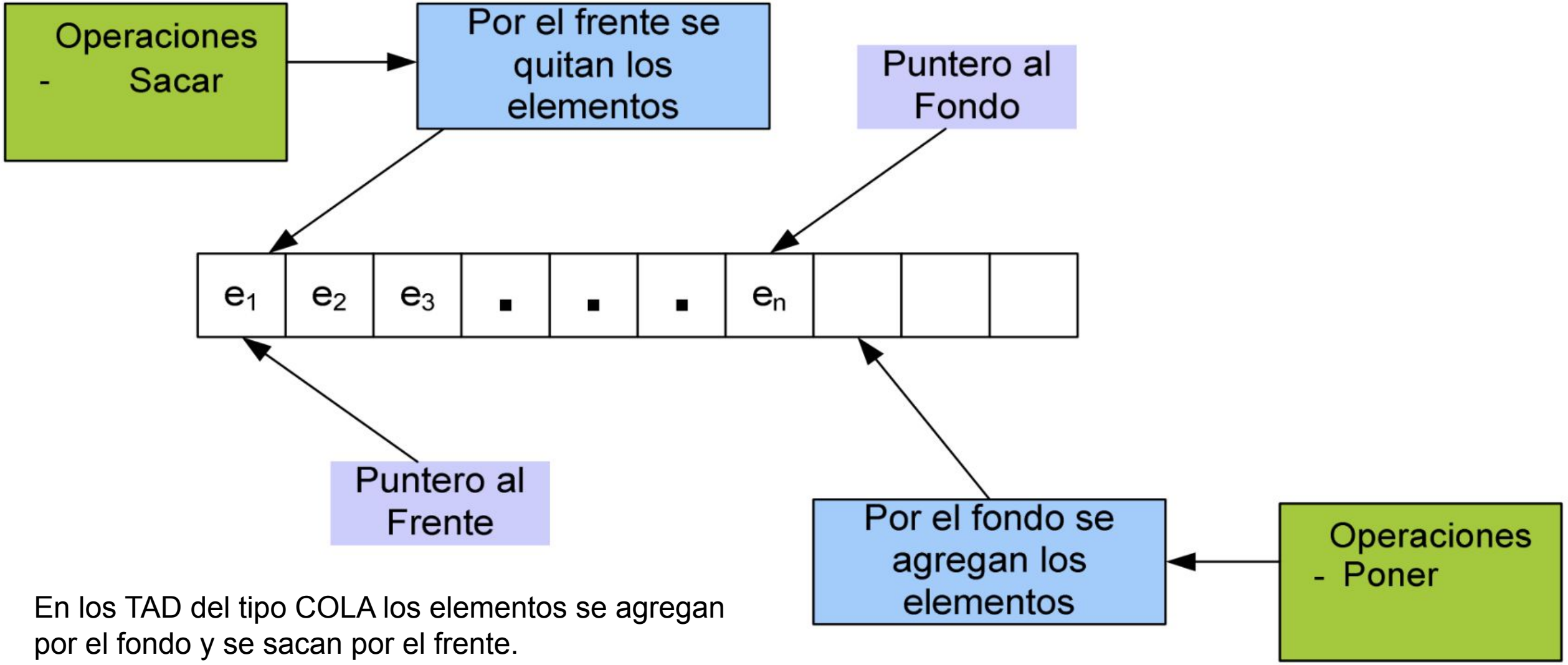

ORT ARGENTINA
Educando para la vida

"

Una **cola** es un grupo de elementos homogéneos (todos del mismo tipo), en donde los nuevos elementos se agregan por un extremo (el final) y se sacan por otro extremo (el frente).

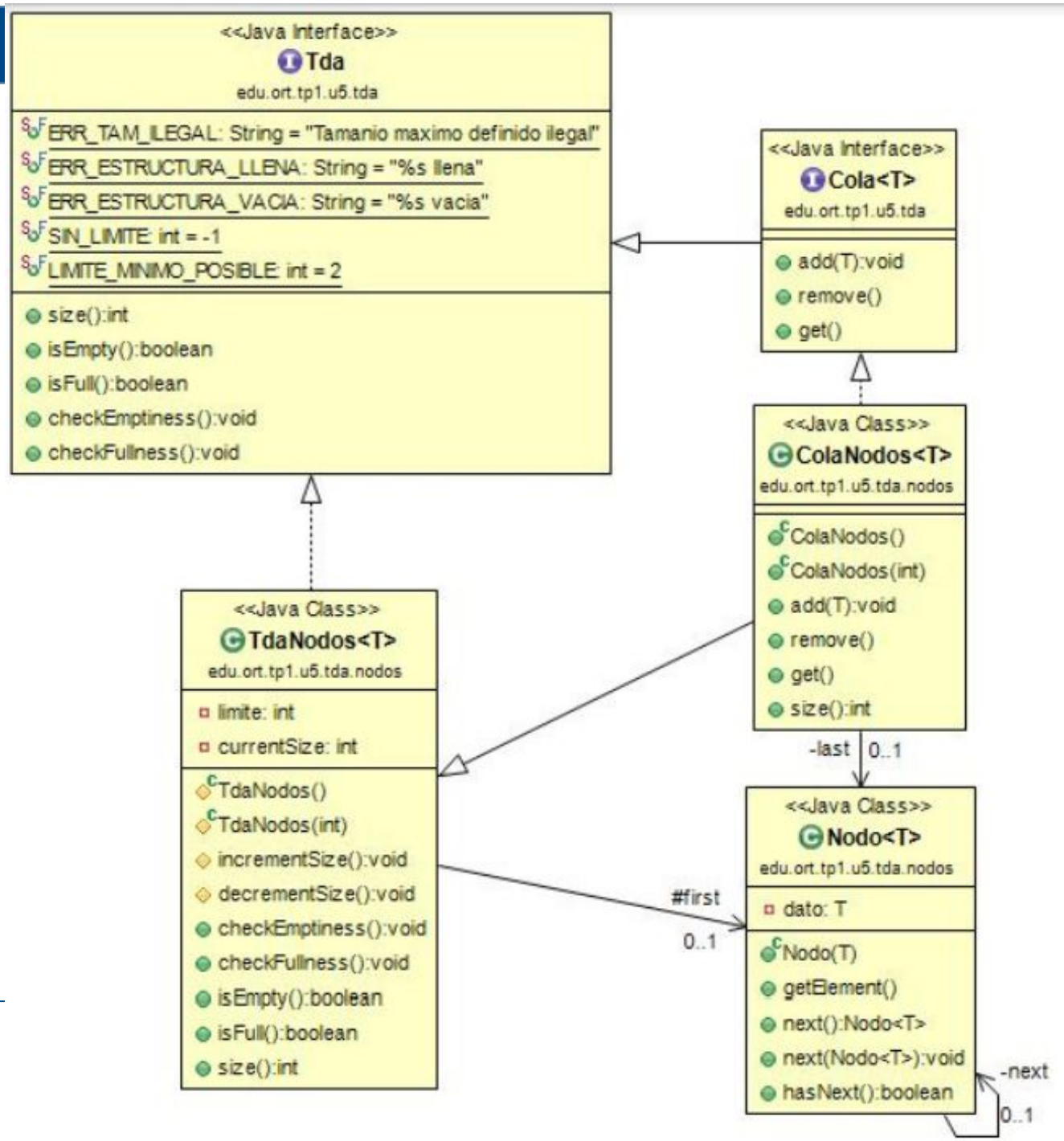
ORT אורט ARGENTINA
Educando para la vida

DEFINICIÓN - COLAS



En los TAD del tipo COLA los elementos se agregan por el fondo y se sacan por el frente.

IMPLEMENTACIÓN PARA COLA



COLAS - OPERACIONES

Constructor()	• Setea en null a **first** y **last**. • Setea el **limite** para isFull() en SIN_LIMITE. • Setea **currentSize** en 0.
Constructor(int)	• Setea en nul a **first** y **last**. • Setea el **limite** para isFull() con el valor del parámetro. • Setea **currentSize** en 0.
void add(T)	Si la cola estuviese llena lanzará una excepción. Crea un nodo donde pone el elemento recibido y lo engancha al final de los previos (si los hay) o como primer elemento. Actualiza **first** (si el el único elemento) y **last**. Además incrementa en 1 el valor de **currentSize**.
T remove()	Si la cola estuviese vacía lanzará una excepción. Extrae el primer nodo de la cadena de nodos actualizando **first**, decrementa en 1 el valor de **currentSize** y devuelve el elemento guardado en el nodo extraído (el nodo en sí se descarta).
T get()	Si la cola estuviese vacía lanzará una excepción. Devuelve el elemento del nodo referenciado por **first**, sin extraerlo
boolean isEmpty()	Devuelve true cuando **currentSize** vale 0.
boolean isFull()	Si **limite** vale SIN_LIMITE (-1) porque se usó el constructor por defecto devolverá siempre falso. Si no, su valor dependerá de la comparación entre **limite** y **currentSize**.

COLAS - EJEMPLOS - OBJETOS PRIMITIVOS

```
public static void main(String[] args) {

    //Declaro una variable de tipo Interfaz Cola que va a contener clases de tipo String
    Cola<String> cola;
    //Instancio un clase de tipo ColaNodos que implementa la interfaz Cola y por ende es una operación segura
    //Upcasting
    cola = new ColaNodos<>();
    System.out.println("Encolamos algunos elementos...");
    cola.add("Colas");
    cola.add("en");
    System.out.println("Elemento primero en la cola: " + cola.get());//Devuelve el elemento que esta primero pero no lo retira
    //System.out.println("Tamaño de la cola: " + cola.size());//Arroja una excepcion, el tamaño de la cola no es visible
    System.out.println("Desencolamos todos elementos...");
    System.out.println(cola.remove());
    System.out.println(cola.remove());
    //System.out.println(cola.remove());//Arroja una excepcion
    System.out.println("Tamaño de la cola: " + cola.size());//Devuelve la cantidad de elementos que posee la cola
    if(cola.isEmpty())//Metodo para validar si la cola esta vacia
        System.out.println("La cola esta vacia");
}
```


ORT ARGENTINA
Educando para la vida

COLAS - EJEMPLOS - OBJETOS

```
public static void main(String[] args) {

    //Declaro una variable de tipo Interfaz Cola que va a contener clases de tipo String
    Cola<Persona> cola;
    //Instancio un clase de tipo ColaNodos que implementa la interfaz Cola y por ende es una operación segura
    //Upcasting
    cola = new ColaNodos<>();
    System.out.println("Encolamos algunos elementos...");
    Persona personal = new Persona("Juan", "Perez");
    Persona persona2 = new Persona("Daniela", "Gonzalez");
    cola.add(personal);
    cola.add(persona2);
    System.out.println("Elemento primero en la cola: " + cola.get());//Devuelve el elemento que esta primero pero no lo retira
    System.out.println("Desencolamos todos elementos...");
    System.out.println(cola.remove());
    System.out.println(cola.remove());
    //System.out.println(s.remove());//Arroja una excepcion
    if(cola.isEmpty())//Metodo para validar si la cola esta vacia
        System.out.println("La cola esta vacia");
}
```


ORT ARGENTINA
Educando para la vida

COLAS - EJEMPLOS - COMO RECORRER UNA COLA

```
Cola<Persona> cola;
cola = new ColaNodos<>();
System.out.println("Encolamos algunos elementos...");
Persona personal = new Persona("Juan", "Perez");
Persona persona2 = new Persona("Daniela", "Gonzalez");
cola.add(personal);
cola.add(persona2);
System.out.println("Todas las personas de la cola");
//Agrego el centinela
Persona centinela = new Persona("*","*");
cola.add(centinela);
Persona persona;
persona = cola.remove();
//Lo uso para recorrer toda la cola y saber donde termina
while (!persona.getNombre().equals("*")) {
    System.out.println(persona.toString());
    cola.add(persona);
    persona=cola.remove();
}
System.out.println("Desencolamos todos elementos...");
System.out.println(cola.remove());
System.out.println(cola.remove());
```

Agrego un elemento a la cola como “centinela” que va a marcar el final de la misma, luego voy sacando elemento por elemento y lo vuelvo a introducir hasta llegar al centinela, ahi se que termine de recorrer toda la cola

"

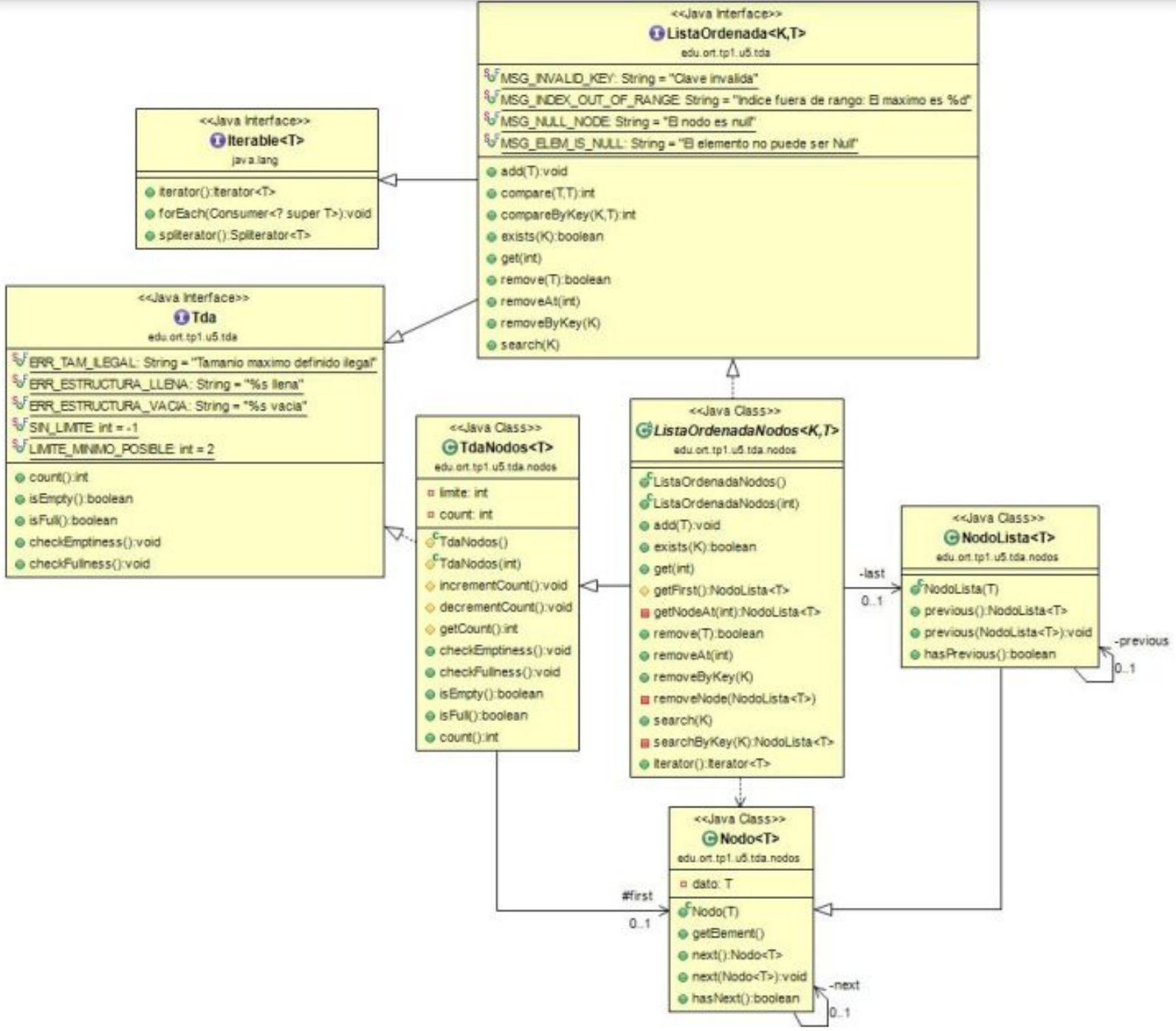
Una **lista** es una colección homogénea de elementos con una relación lineal entre ellos. Es decir, cada elemento de la lista (excepto el primero) tiene un único elemento predecesor y cada elemento (excepto el último) tiene un elemento sucesor.

ORT ARGENTINA
Educando para la vida

DEFINICIÓN - LISTAS ORDENADAS

- A diferencia de las estructuras anteriores, las listas ordenadas no pueden adivinar cuál será el orden que tendrán
- En principio las listas ordenadas intentarán acomodar sus elementos ordenándolos de menor a mayor o de mayor a menor (es indistinto)

IMPLEMENTACIÓN PARA LISTAS



LISTAS - OPERACIONES

Constructor()	• Setea en null el valor de **first** y **last**. • Setea el **limite** para isFull() en SIN_LIMITE. • Setea **currentSize** en 0.
Constructor(int)	• Setea en null a **first** y **last**. • Setea el **limite** para isFull() con el valor del parámetro. • Setea **currentSize** en 0.
void add(T)	Si la lista estuviese llena lanzará una excepción. Crea un nodo donde pone el elemento recibido y lo inserta donde corresponda. Dependiendo de la posición que ocupe el nodo al ser ubicado en la lista puede llegar a actualizar **first** y/o **last**. Además incrementa en 1 el valor de **currentSize**. Nota: La posición del nodo quedará determinada por lo que se haya implementado en el método compare(...) y los valores utilizados como clave
boolean exists(K key)	Indica si hay algún elemento en la lista cuya clave coincida con el valor recibido.
T get(int pos)	Si la lista estuviese vacía o si la posición no existiese lanzará una excepción. Devuelve el elemento del nodo ubicado en pos, pero sin extraerlo.
boolean isEmpty()	boolean isEmpty() Devuelve verdadero cuando **currentSize** vale 0.
boolean isFull()	Si **limite** vale SIN_LIMITE (-1, porque se usó el constructor por defecto) devolverá siempre false. Si no, su valor dependerá de la comparación entre **limite** y **currentSize**

LISTAS - OPERACIONES

boolean remove(T elem)	Si existiese dentro de la lista, remueve el nodo del elemento recibido por parámetro. Devuelve true cuando la operación fue exitosa.
T removeAt(int pos)	Si la lista estuviese vacía lanzará una excepción. Extrae el elemento del nodo pos de la cadena de nodos (de ser necesario actualiza **first** y/o **last**), decrementa en 1 el valor de **currentSize** y devuelve el elemento que estaba guardado en el nodo extraído (el nodo en sí se descarta).
T removeByKey(K key)	Si la lista estuviese vacía lanzará una excepción. Extrae el nodo del elemento cuya clave coincida con el valor recibido por parámetro, decrementa en 1 el valor de **currentSize** y devuelve el elemento que estaba guardado en el nodo extraído (el nodo en sí se descarta). De no encontrar el elemento devolverá null
T search(K key)	Si la lista estuviese vacía lanzará una excepción. Obtiene el elemento cuya clave coincida con el valor recibido por parámetro, pero no lo extrae. De no encontrar el elemento devolverá null.
Iterator iterator()	Devuelve un iterador para recorrer la lista. Este iterador puede ser pedido explícitamente o, como comúnmente sucede, se obtiene de manera implícita al recorrer la lista con un for-each.

ORT ARGENTINA
Educando para la vida

LISTAS- EJEMPLOS - COMO CREAR LISTAS ORDENADAS

- A diferencia de Pilas y Colas, la clase que usaremos para implementar Listas ordenadas (**ListaOrdenadaNodos**) es abstracta, ya que la responsabilidad de definir el método de ordenamiento de los nodos va a depender de cada implementación particular (no va a ser lo mismo ordenar por nombre que por precio por ejemplo)
- Por lo tanto siempre debemos declarar una nueva clase que extenderá a **ListaOrdenadaNodos**, donde debemos sobreescribir los métodos **compare** y **compareByKey**. Ejemplo:

```
public class EmpleadosPorLegajo extends ListaOrdenadaNodos<Integer, Empleado> {

    @Override
    public int compare(Empleado empleado1, Empleado empleado2) {
        return empleado1.getLegajo() - empleado2.getLegajo();
    }

    @Override
    public int compareByKey(Integer clave, Empleado empleado) {
        return clave - empleado.getLegajo();
    }
}
```

- Ambos métodos de comparación, en cualquier implementación que hagamos, **deben devolver un número entero**:
  - Si devuelve 0 (cero) se asume que las claves de ambos elementos comparados (o la clave y el elemento comparado en el segundo método) son **iguales** o **equivalentes** (digamos que valen lo mismo).
  - Si devuelve un número positivo significa que **el segundo es mayor que el primero**.
  - Si devuelve un número negativo indica que **el segundo es menor que el primero** (digamos que están al revés).

¿Cómo comparar claves no numéricas?

- Podemos escribir estas rutinas como queramos, siempre y cuando devolvamos un número entero. En el siguiente ejemplo vamos a aprovechar una característica de los Strings (que implementan la interfaz Comparable) para ordenar la lista ya no por el número de legajo, sino por el nombre (asumamos que es el nombre completo)

```
public class EmpleadosPorNombre extends ListaOrdenadaNodos<String, Empleado> {

    @Override
    public int compare(Empleado dato1, Empleado dato2) {
        return dato1.getNombre().compareTo(dato2.getNombre());
    }

    @Override
    public int compareByKey(String clave, Empleado empleado) {
        return clave.compareTo(empleado.getNombre());
    }
}
```

LISTAS- EJEMPLOS - OBJETOS

```
public class TestListaOrdenada {

    // Declaro las dos listas con las que voy a querer mantener ordenados los
    // empleados.
    private static EmpleadosPorLegajo listaPorLegajo;
    private static EmpleadosPorNombre listaPorNombre;

    public static void main(String[] args) {
        // Inicializo ambas clases
        listaPorLegajo = new EmpleadosPorLegajo();
        listaPorNombre = new EmpleadosPorNombre();

        // Agrego los empleados a ambas listas
        agregarEmpleado(5, "Sebastian");
        agregarEmpleado(3, "Mariano");
        agregarEmpleado(7, "Nir");
        agregarEmpleado(1, "Gaby");
        agregarEmpleado(2, "Dany");
        agregarEmpleado(4, "Carlos");
        agregarEmpleado(8, "Sebastian");

        // listo los empleados ordenados de dos formas distintas (por legajo y por nombre)
        listar(listaPorLegajo);
        listar(listaPorNombre);

        // Elimino a uno de los empleados de una de las listas
        System.out.println("Eliminando a Dany de la lista por legajo...");
        Empleado e = listaPorLegajo.removeByKey(new Integer(2));
```

LISTAS- EJEMPLOS - OBJETOS

```
    // muestro el empleado removido
    System.out.println("Se removio a " + e);

    System.out.println("Sin embargo, el empleado 'sobrevive' en la lista por nombre");
    listar(listaPorLegajo);
    listar(listaPorNombre);

    // lo removemos de la segunda lista y volvemos a mostrar la lista por nombre
    System.out.println("Eliminando a Dany de la lista por nombre...");
    e = listaPorNombre.removeByKey("Dany");

    // muestro el empleado removido
    System.out.println("Se removio a " + e);

    listar(listaPorNombre);
}

private static void agregarEmpleado(int legajo, String nombre) {
    System.out.println("Agrego " + legajo + " - " + nombre + " en ambas listas");
    Empleado e = new Empleado(legajo, nombre);
    listaPorLegajo.add(e);
    listaPorNombre.add(e);
}

private static void listar(ListaOrdenadaNodos<?, Empleado> lista) {
    System.out.println(String.join(" ", lista.getClass().getSimpleName().split("(?=\\p{Upper})")));
    for (Empleado empleado : lista) {
        System.out.println(empleado);
    }
}
```