Pilas, Colas y Listas Ordenadas en Java

Un Tipo de dato abstracto (TDA) es una estructura que integra un conjunto de datos (elementos) con un grupo de operaciones específicas que determinan el comportamiento de la estructura y la forma en la cual manipula los elementos contenidos.

Cada TDA provee de una interfaz a través de la cual podemos realizar las operaciones permitidas, abstrayéndose de cómo éstas 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). Bajo POO, estas estructuras aprovechan el Polimorfismo, y en el caso específico de las implementaciones que presentamos, agregamos Generics para aumentar la flexibilidad de las estructuras y evitar o minimizar redefiniciones de clases innecesarias.

Tipos de TDA

Hay muchas TADs distintas, las que se separan por su comportamiento general.

- **Pila** sabe apilar elementos sin importarle su tipo: para manipularlos usa un único punto de acceso, tanto para cargar nuevos elementos como para extraerlos (el último que entra es el primero que sale).
- **Cola** sabe encolar elementos, también sin importarle su tipo: los carga siempre al final y siempre saca el primero que recibió (el primero que entra es el primero que sale).
- **ListaOrdenada** (que extiende **Lista**) contiene una colección de objetos. 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 (veremos esto en detalle cuando tratemos esta estructura). 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. Hay muchísimas más variaciones que exceden lo que aquí queremos tratar.

Algunos otros TDAs, que son conocidos pero sobre los que no vamos a trabajar son:

- **Árbol**: con varias estructuras distintas dentro de esta subfamilia de TDAs, los árboles tienen la cualidad excepcional respecto a las otras estructuras de no ser secuenciales. Cada *nodo* tiene un elemento/dato y referencias para acceder a sus nodos hijos, los que forman cada un nuevo árbol (cada nodo es *raíz* de su subárbol). Así se genera una estructura arbolada que por cada *nivel* se abre en más *ramas*, pues estas referencias deben cumplir dos condiciones: no puede haber dos claves idénticas y ningún nodo puede referenciar a un nodo previo. Esto hace que la búsqueda de los elementos contenidos no sea secuencial y así se utilicen menos *visitas* para llegar a un elemento (o descartar su existencia).

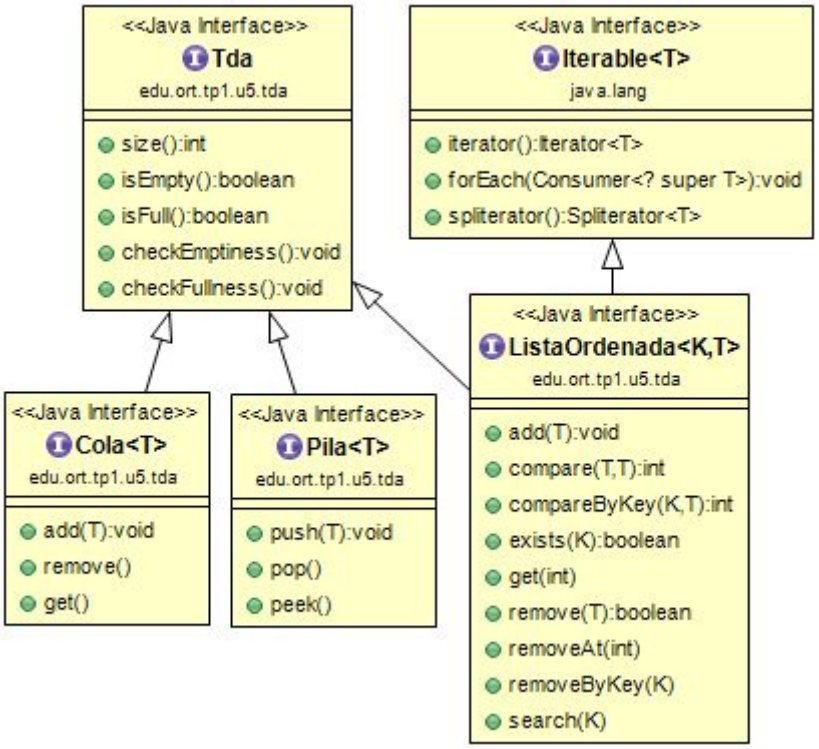
- **Mapa** mantiene una suerte de *diccionario* donde hay una clave que es irrepetible y un elemento relacionado. Pueden existir distintas implementaciones (incluso aprovechando algunas de las estructuras anteriores) pues, ordenada o no, la estructura se caracteriza por no dar lugar a dos elementos con la misma clave.
- **Grafo**: quizá la más compleja de estas estructuras, un grafo permite representar cualquier tipo de relaciones donde cada elemento puede apuntar a uno o muchos más para formar una red relacional. Aunque hay muchas clasificaciones para grafos, la más conocida es la que los agrupa en *direccionados* e *indireccionados*.
- **Urna** o **Bolsa** contiene una colección de objetos que se sacarán de a uno al azar. Aunque **Urna** no suele incluirse entre los TDAs "formales" es una estructura con muchos usos que merece ser conocida: se utiliza, por ejemplo, para sacar los números de una lotería o bingo, pues su mecanismo permite que aún trabajando "al azar" nunca se obtenga en mismo resultado (es la que se implementó para explicar ***Generics***).

En esta materia exploramos sólo algunas de estas estructuras. Como primeros ejemplos veremos aquí implementaciones de Pilas y Colas usando Nodos, aunque es posible realizar otras implementaciones, como por ejemplo con ArrayLists y Arrays. Dado que cada una de estas estructuras puede implementarse de varias formas distintas y usando diferentes técnicas que incluso pueden usar otros TDAs como punto de partida, como regla general y bajo la POO, todos los TDAs se declaran usando interfaces.

Respecto a estas implementaciones, y como la funcionalidad intrínseca de estas estructuras no tiene en cuenta el tipo de elemento contenido (en cierta forma podemos decir que a cada una de estas estructuras *no le importa* el tipo de información que guarda) las implementaciones que presentamos hacen uso de *Generics* debido a que el *comportamiento* de ambas estructuras se basa en el orden por el cual se puede acceder, agregar y quitar un elemento:

- 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.
- Las **Listas** son estructuras de acceso secuencial y aleatorio. No tienen un orden preestablecido para acceder a sus elementos (nosotros ya conocemos las operaciones que se pueden realizar sobre un ArrayList, por ejemplo). En este caso vamos a utilizar un tipo especial de listas, las **Listas Ordenadas**, donde sus elementos quedarán ordenados según un dato especial asociado a cada elemento al cual denominamos ***clave***.

A continuación puede verse el diagrama de relaciones entre las interfaces que declaran la funcionalidad de los TDAs que vamos a usar.

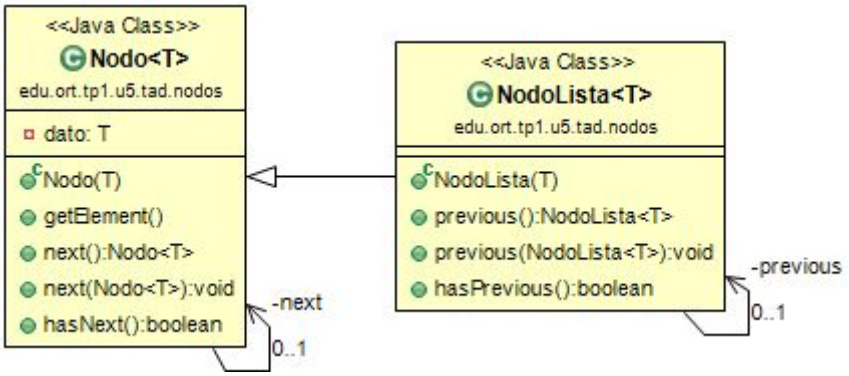


Un detalle que tienen en común entre ***Pila*** y ***Cola*** es que ambas estructuras **no permiten** acceder a ningún otro elemento contenido en el TDA que no sea el primero. **Es imposible** buscar o acceder a otro elemento que no sea este. Pero ***ListaOrdenada*** no sólo permite hacerlo, sino que también extiende la interfaz ***Iterable*** y por lo tanto podrá ser recorrida completa y secuencialmente usando *for-each*.

Antes de continuar

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..También verás 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.



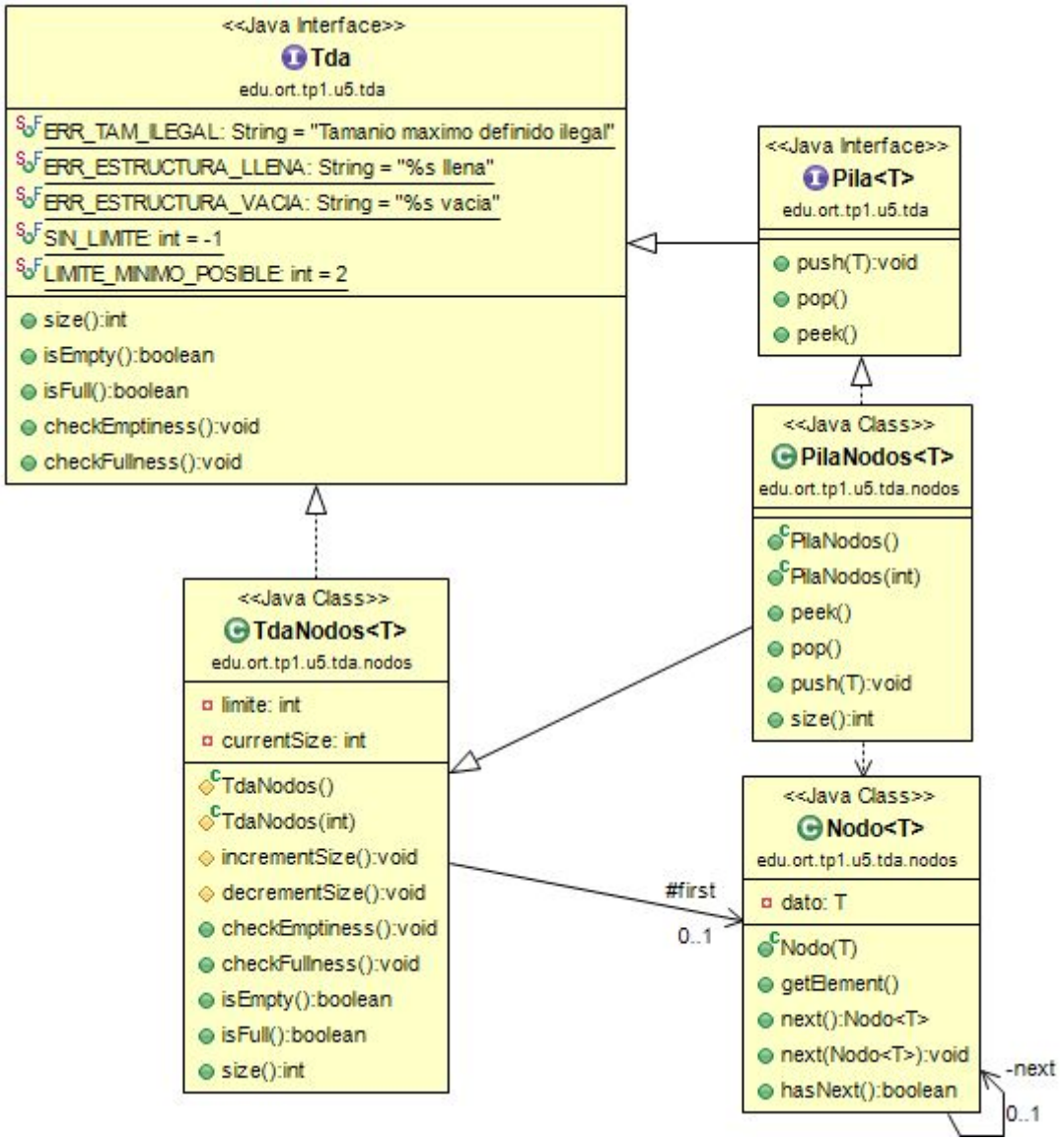
Una comparación quizás burda pero efectiva es la de los vagones de un tren: cada vagón se engancha al siguiente, y el tipo de vagón está determinado por el tipo de contenido que transportará: pasajeros, carga, etc. En el caso de NodoLista podemos imaginar que así como hay un gancho al siguiente vagón, cada vagón tiene también un gancho para enlazarse con el anterior a él.

Si te interesa comprender cómo funciona cada una de las estructuras que implementamos con nodos, adelante. Vas a encontrar aquí ejemplos prácticos de Herencia, Polimorfismo, Clases Abstractas, Generics, Agregación y todo lo que fuiste conociendo a lo largo del cuatrimestre; posiblemente también encuentres cosas que te pueden servir en otro momento para solucionar algún problema de implementación (esté o no relacionado con las TDAs).

Y si no te interesa, podés saltearte completamente cómo están implementadas estas estructuras pues lo que nos interesa es que aprendas a usarlas y combinarlas. No obstante, revisar los fuentes de su implementación puede darte pistas de cómo solucionar algunos problemas que se pueden presentar en tu vida como programador.

Implementación para Pila

La implementación de **Pila<*TipoElemento*>** es extremadamente similar, aunque los nombres para los métodos *add()*, *remove()* y *get()* suelen encontrarse como *push()*, *pop()* y *peek()*, respectivamente, y por eso éstos son los nombres que elegimos para diferenciar claramente la Pila de la Cola:



Puede verse que en la implementación de **Pila** no necesita guardar ningún dato por su cuenta (***first*** es atributo común de la clase **TdaNodos**). Por lo tanto no tiene atributos propios, *conoce* a los nodos (relación de dependencia) y solamente aporta *funcionalidad específica* a **TdaNodos** extendiéndola con sus propios métodos para la manipulación de elementos (*push(...)*, *peek()* y *pop()*) que harán que *funcione* como una pila. Además, por implementación **tampoco** dejará usar el método *size()* original.

¿Qué hace cada método?

La tabla que aparece a continuación muestra qué hace la clase para cumplir con la interfaz (se explica la funcionalidad sin importar si se hace en la propia clase o en la clase base)

Constructor()	• Setea en null el valor de ***first***. • Setea el ***limite*** para isFull() en SIN_LIMITE. • Setea ***currentSize*** en 0.

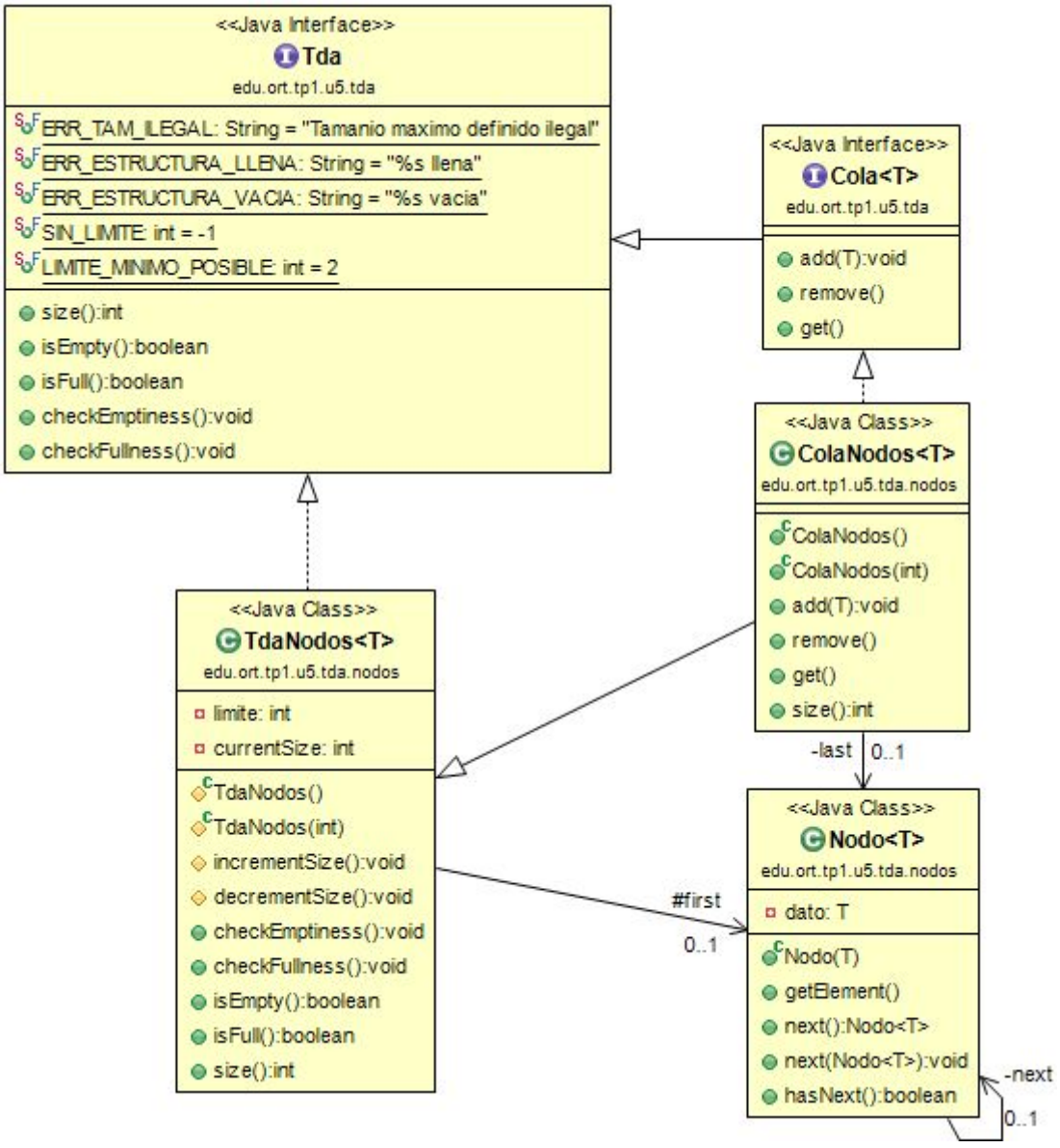
Constructor(int)	• Setea en null a ***first***. • Setea el ***limite*** para isFull() con el valor del parámetro. • Setea ***currentSize*** en 0.
void push(TipoElemento)	Si la pila estuviese llena lanzará una excepción. Crea un nodo donde pone el elemento recibido y lo *engancha* delante de los previos (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***.

Implementación para Cola

La primera implementación que mostramos es la del TDA **Cola<*TipoElemento*>.**

Al igual que se hace con ArrayList, *TipoElemento* indica qué clase de elemento podrá contener (puede ser tanto una *clase* como una *interfaz*). Presentamos la implementación con ***Nodos***.

Al igual que las estructuras y clases nativas del lenguaje, esta implementación aprovechará el mecanismo de *excepciones* para informar cualquier error o mensaje que se produzca, como por ejemplo si se intenta extraer un elemento de la estructura cuando ésta esté vacía. Pueden verse, en la declaración de la interfaz, la declaración de varias constantes de texto que serán utilizadas para lanzar las excepciones cuando sea necesario. Para no obligar a escribir estructuras try-catch todo el tiempo estas excepciones serán de la familia de *RuntimeException*, (lo que posiblemente no sea una buena práctica) pero serán la única manera de exponer un mensaje desde el interior de las clases (nunca harán por su cuenta un *System.out.println()* ni nada similar). Por implementación **tampoco** dejará usar el método *size()*, el cual al intentar usarlo lanzará una excepción.



¿Qué hace cada método?

La tabla que aparece a continuación muestra qué hace la clase para cumplir con la interfaz (se explica la funcionalidad sin importar si se hace en la propia clase o en la clase base):

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***.

TIPS: Transferencia de valores entre constructores

El constructor por defecto de cada clase debe hacer prácticamente lo mismo que el constructor parametrizado pues la única diferencia es cuál es el valor se asigna al límite que determina si la estructura está llena o no).

Por eso es una pena redefinir el constructor de la cada clase dos veces, con todo idéntico menos esa única diferencia.

Entonces, ¿por qué no reutilizar todo el constructor parametrizado llamándolo desde el constructor por defecto?

El secreto es llamar al constructor parametrizado utilizando la referencia especial ***this***.

Mostramos acá la implementación que usamos (**ColaNodos**) que usa ***super()*** o ***super****(limite)* para invocar al constructor de la clase base:

```
public ColaNodos() {
	super();
}

public ColaNodos(int limite) {
	super(limite);
	this.last = null;
}
```

Pero allí…

```
protected TadNodos() {
	this(SIN_LIMITE);
}

protected TadNodos(int limite) {
	if (limite != SIN_LIMITE && limite < LIMITE_MINIMO_POSIBLE) {
		throw new IllegalArgumentException(ERR_TAM_ILEGAL);
	}
	this.limite = limite;
	this.first = null;
	this.currentSize = 0;
}
```

Como puede verse en el código, el constructor por defecto simplemente invoca al constructor parametrizado usando ***this*** con el parámetro SIN_LIMITE. De esta manera reaprovecha no sólo la definición del resto de los atributos de la clase, sino también el control del parámetro recibido para asegurar que su valor no sea ilógico e incompatible con lo que se pretende de la estructura.

A diferencia de la implementación de **Pila**, la **Cola** si necesita guardar una referencia por su cuenta, que es la referencia al último elemento. Por supuesto, Cola aporta la lógica necesaria para que la estructura se comporte como tal.

Agregar y quitar elementos en la implementación con nodos de Pila y Cola

Un detalle a tener en cuenta en cuanto a estas implementaciones es que no debemos confundir el ***nodo*** con el ***elemento*** o ***dato***.

El ***dato*** o ***elemento*** es lo que se agrega, mira o quita; el ***nodo*** es su *contenedor* circunstancial y quien lo guarda en su interior. Por otra parte, los nodos ***nunca*** **son accesibles a través de la interfaz del TDA** y **están encapsulados**.

Por lo tanto tampoco podemos acceder a la cadena de nodos. Para quien usa el TDA, cualquier implementación es completamente *transparente* e indistinguible de otra.

Observemos con detenimiento las siguientes implementaciones de los métodos de intercambio de Cola (*add(...)*, *remove()* y *get()*):

Esta es la implementación de ***add(..)***. Puede verse al principio el control de cola llena (implementado de manera global para todas los TDAs en **TadNodos**). Después de la creación del nodo usando el operador ***new*** hace la inserción selectiva del nuevo nodo dependiendo del estado de la Cola (si está vacía o no). Siempre pondrá el elemento como elemento final (usará el atributo *last*) y siempre incrementará en uno la cantidad de elementos.

```
@Override
public void add(T elemento) {
	checkFullness();
	Nodo<T> nuevoNodo = new Nodo<>(elemento);
	if (isEmpty()) {
		first = nuevoNodo;
	} else {
		last.next(nuevoNodo);
	}
	last = nuevoNodo;
	incrementSize();
}
```

El método ***remove()*** hace lo opuesto. Chequeará que la cola no esté vacía y tomará el primer nodo, siempre referenciado por ***first***. Al actualizar este atributo asignándole *cabecera.**getNext()*** hace que el nodo que se esté eliminando ya no sea tenido en cuenta por la estructura (quedó "desenganchado"). Sin embargo se mantiene la referencia a este con la variable auxiliar *aux*; eso se aprovechará en la última línea, después de restar uno a cantidad, para devolver el elemento usando *aux.**getElement()***.

```
@Override
public T remove() {
	checkEmptiness();
	Nodo<T> aux = first;
	first = first.next();
	decrementSize();
	return aux.getElement();
}
```

Nos queda ver la implementación de ***get()***, que no es más que una simplificación del método anterior ya que no debe actualizar nada, sólo devolver el dato o elemento del nodo señalado por ***first***.

```
@Override
public T get() {
	checkEmptiness();
	return first.getElement();
}
```

Uso de Pilas y Colas

Para utilizar las estructuras haremos exactamente lo mismo que hacemos con ArrayList: las declararemos indicando el tipo de elemento contenido dentro de los paréntesis agudos: Por ejemplo podríamos hacer:

```
Pila<Caja> pilaDeCajas = new PilaNodos<>();
Cola<Persona> filaDePasajeros = new ColaNodos<>();
```

Para agregar una caja en la pilaDeCajas haremos:

```
pilaDeCajas.push(new Caja());
```

Y para agregar una persona a la fila…

```
filaDePasajeros.add(new Persona());
```

Para ver los elementos disponibles haremos:

```
System.out.println("Caja visible: " + pilaDeCajas.peek());
System.out.println("Primera persona en la fila: " + filaDePasajeros.get());
```

Cabe aclarar que ninguna de estas dos operaciones modifica el estado de las estructuras (no agrega ni quita ningún elemento, simplemente muestran el único que se puede *ver* en cada caso). En cambio…

```
System.out.println("Caja visible: " + pilaDeCajas.pop());
System.out.println("Primera persona en la fila: " + filaDePasajeros.remove());
```

… quitan el primer elemento disponible en cada estructura. En este caso dejarán ambas estructuras vacías pues las dos tenían un único elemento cada una. Ahora el método *isEmpty()* de ambas devolverá *true*.

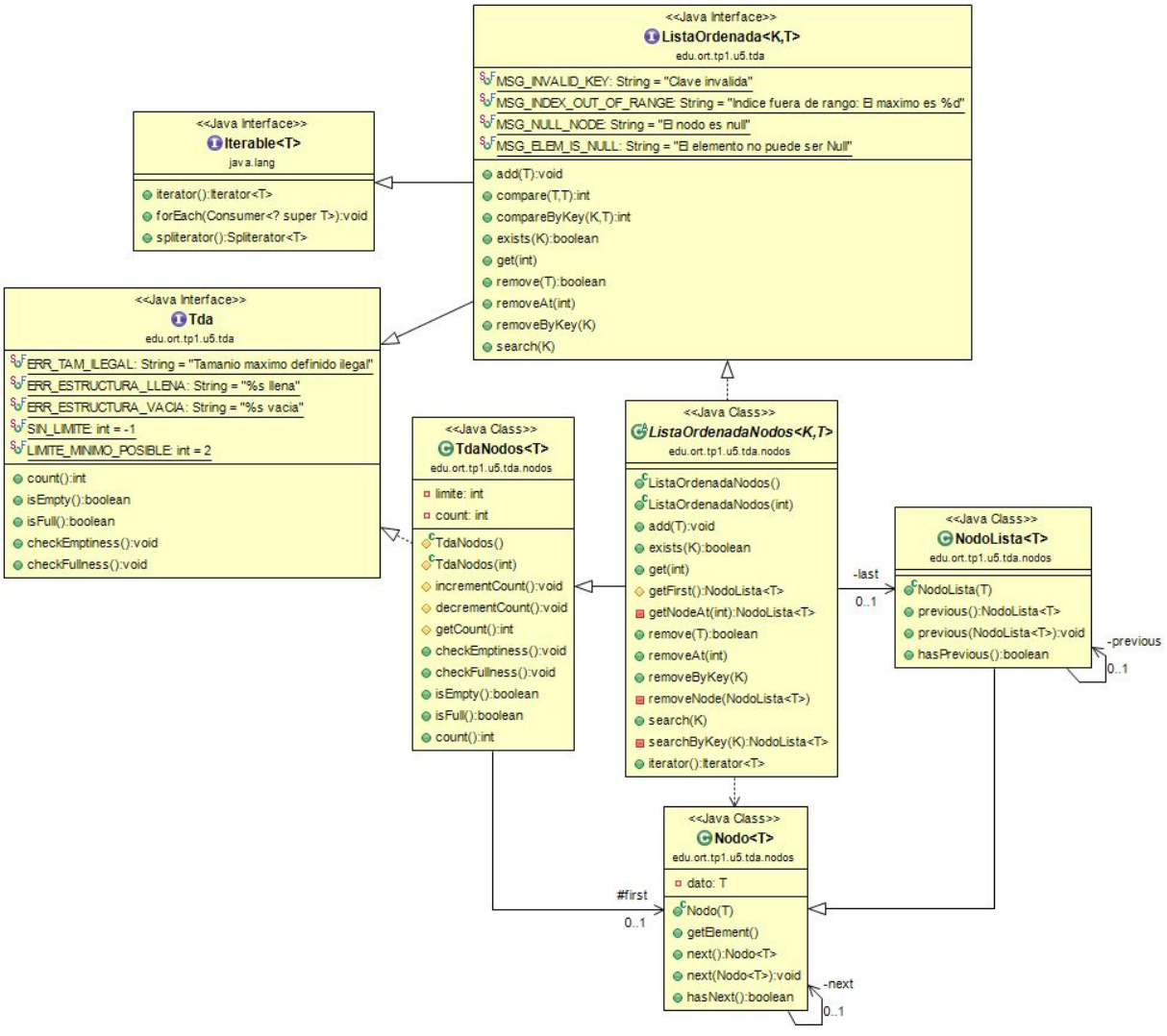
Como ven, una vez implementadas las estructuras su uso es tan simple como el de un ArrayList.

Queda ahora ver la Lista Ordenada.

Implementación de Listas Ordenadas

A diferencia de las estructuras anteriores, las listas ordenadas no pueden adivinar cuál será el orden que tendrán sus elementos pues esto depende del dato o los datos que usemos para definir su clave.

En principio las listas ordenadas intentarán acomodar sus elementos ordenándolos de menor a mayor o de mayor a menor (es indistinto). Sea como sea, la clase no puede saber de antemano. la regla por la cual los elementos podrán ser comparados. Por lo tanto las listas ordenadas se implementan como una clase abstracta. Las listas, entonces, deberán ser nuevas clases derivadas de **ListaOrdenada<K, T>**. Estas serán las que implementen los dos métodos que en la clase **ListaOrdenadaNodos<K, T>** no figuran (sólo declarados en la interfaz) y que son necesarios para que la lista "aprenda" a ordenar los elementos: **compare(elemento1, elemento2)** y **compareKey(clave, elemento)**. La primera es utilizada durante la inserción de los nuevos elementos; la segunda, para buscar un elemento a partir de su clave.



¿Qué hace cada método?

La tabla que aparece a continuación muestra qué hace clase para cumplir con la interfaz. A diferencia de las clases anteriores, **ListaOrdenada** es abstracta pues hay métodos que deberemos implementar en cada clase derivada que creemos a partir de ella. Pero todo lo demás ya está definido.

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()	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***.
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<T> 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*.

Uso de ListaOrdenada

La primera salvedad es que esta estructura de datos está implementada como una clase abstracta. Por lo tanto, siempre debe ser implementada como una extensión de esta.

Sin embargo, la implementación de estas clases derivadas es tan cortita que podemos poner una de ellas completa aquí, sin problemas. Lo único que necesitamos es hacer que la clase haga lo que su superclase no puede hacer: definir cómo se ordenarán los nodos:

```
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();
	}

}
```

En este caso elegimos una lista de empleados que necesita ser ordenada por el número de legajo. Este dato es un número entero: por eso el primer parámetro de clase es **Integer**, ya que no pueden usarse los tipos de datos nativos (siempre debe ser algo que derive de ***Object***). El segundo sí se refiere al elemento, que en este caso es la clase **Empleado**.

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 calcular esto si las claves no son numéricas?

Es relativamente simple: 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());
	}

}
```

En este caso lo que se usa es el método ***compareTo****(valor)* que hace exactamente lo mismo que lo detallado en el ejemplo previo.

Veamos ahora cómo usar las clases que acabamos de crear. A diferencia de los casos anteriores (Pila y Cola) aquí usaremos las clases que acabamos de crear:

```
	EmpleadosPorLegajo listaPorLegajo = new EmpleadosPorLegajo();
	EmpleadosPorNombre listaPorNombre = new EmpleadosPorNombre();
```

Luego, no es más que utilizarlos tal como si fuese *casi* como un ArrayList, con el agregado de que la búsqueda de un elemento no la tenemos que hacer nosotros, sino que ya lo hace la propia clase a través de su método *search(...)*.

Las listas son *iterables*

Una cosa que no podemos perder en nuestras listas es la posibilidad de recorrerlas secuencial y completamente con *for-each*. Gracias a la implementación de las interfaces *Iterable* e *Iterator* (algo que es más fácil de lo que parece y que está disponible en la librería completa que acompaña este apunte, en esta misma unidad) podemos usar *for-each* para recorrer la lista completa:

```
for (Empleado empleado : empleadosPorLegajo) {
	System.out.println(empleado);
}
```

Ahora lo que resta es combinar las estructuras, usando las más convenientes según lo que requiera el ejercicio o el problema en cada caso.