

# Colecciones

Java Intermedio

14 de Enero de 2019

# Conceptos Importantes

- Interface: Clase especial que todos sus métodos son abstractos.
- Framework: Conjunto de herramientas que una persona o conjuntos de personas ya desarrollo y podemos utilizarlas a nuestro antojo.
- Conurrencia: Capacidad de que distintas partes de un algoritmo puedan ejecutarse casi simultáneamente sin afectar el resultado final.
- Dato: Representación simbólica de un atributo.

# Estructuras de Datos

Una estructura de datos es la forma en la cual una computadora organiza los datos para ser manejado de forma eficiente. Las estructuras de datos se pueden clasificar de varias formas, en cuanto a tamaño de la estructura:

- **Estructura de Datos Estática:** Siempre posee un tamaño fijo. (Por ejemplo: Un arreglo estático)
- **Estructura de Datos Dinámica:** Su tamaño varía en tiempo de ejecución. (Por ejemplo: Una lista)

# Estructuras de datos

En cuanto a la relación de los datos se clasifican en dos grupos, estos grupos son:

- **Estructuras de datos lineales:** Todos los datos poseen máximo un sucesor y mínimo un antecesor.
- **Estructuras de datos no lineales:** Los datos pueden tener n sucesores y n antecesores. Siendo n un número entero mayor o igual a cero.

# Estructuras de Datos

Las estructuras de datos pueden determinar la complejidad de un algoritmo, en general la elección de estructuras de datos eficientes permite la creación de algoritmos eficientes. Algunos métodos de diseño de software destacan las estructuras de datos en lugar de los algoritmos para diseñar.

Cada estructura de datos debe poseer un conjunto mínimo de acciones (métodos), a estos se les conoce como el abc, o CRUD (create-Read-Update-Delete), y consiste en las siguientes acciones:

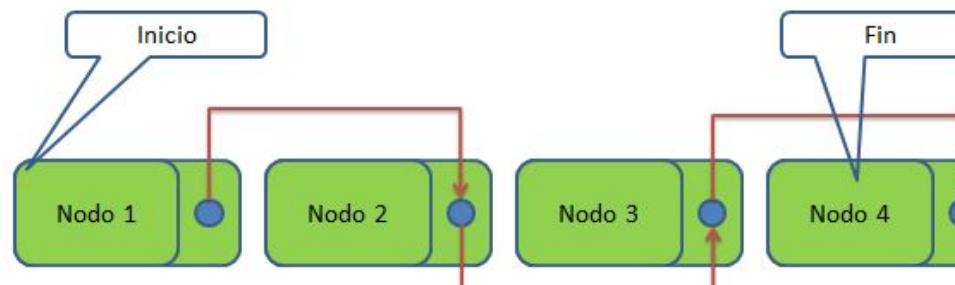
- Crear: Acción que permite crear un elemento en la estructura de datos (inserción).
- Read: Acción que permite leer la información almacenada en la estructura de datos.
- Update: Acción que permite actualizar un dato que se encuentra almacenado en la estructura de datos.
- Delete: Acción que permite borrar un elemento de la estructura de datos.

# Estructuras de Datos Lineales

Las estructuras de datos lineales son aquellas en las que cada dato posee máximo un sucesor y máximo un antecesor.

## Almacenamientos:

- Almacenamiento Contiguo: Cada dato se encuentra almacenado seguidamente de otro en memoria.
- Almacenamiento Ligado: Cada dato se encuentra almacenado en una posición x de memoria, pero cada dato posee la dirección de memoria del dato sucesor (nulo si es el último).



Lista ligada – Podemos recorrer la estructura solo hacia adelante

# Almacenamiento Contiguo vs Ligado

No existe un almacenamiento 100% mejor que el otro, sino que uno se acopla más a las circunstancias en las que nos encontramos. Así que analizaremos cada uno en sus funciones básicas:

- Almacenamiento Contiguo: Este permite agregar o eliminar un dato en tiempo constante, aunque ocupa un poco más de memoria porque requiere la dirección de memoria del dato sucesor, el acceso a un elemento depende de la cantidad de datos en la estructura.
- Almacenamiento Ligado: Para poder agregar o eliminar un dato, copia todos los demás en un conjunto de localidades donde puedan estar juntos, por lo que el tiempo depende de la cantidad de datos. El acceso a un elemento es constante, ya que se sabe la localidad de memoria a la que se desea acceder.

*En conclusión:* El almacenamiento contiguo es mejor cuando trabajamos con una cantidad de datos pequeña y el almacenamiento ligado es mejor cuando trabajamos con grandes cantidades de información.

# Estructura de Datos Lineal

Una estructura de datos lineal es en la que cada elemento sólo puede ir enlazado al siguiente o al anterior.

- Arreglo
- Conjunto
- Pila
- Cola
- Lista Enlazada

# Arreglo

Arreglo: Los datos se almacenan contiguamente en memoria. Puede ser de dos formas:

- Arreglo Estático: Arreglo que nunca cambia la cantidad de elementos en tiempo de ejecución
- Arreglo Dinámico: Arreglo que cambia la cantidad de elementos en tiempo de ejecución

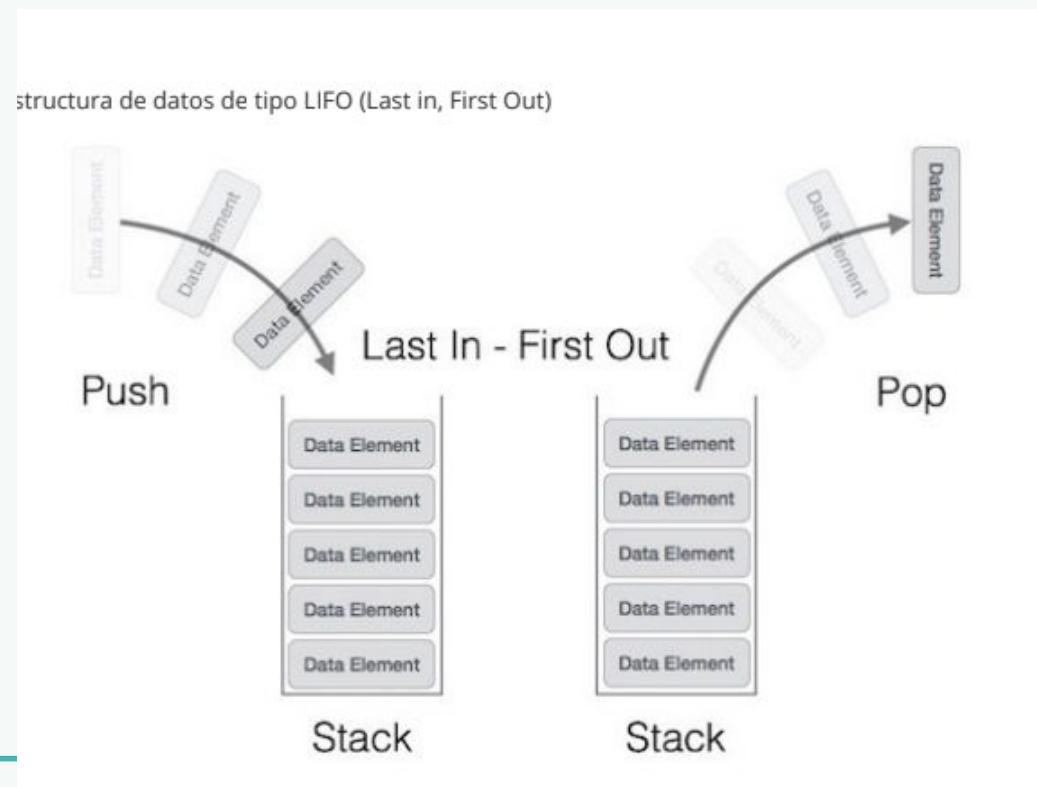
# Conjunto

Colección de varios datos únicos.

conjunto = {1,2,3,4}

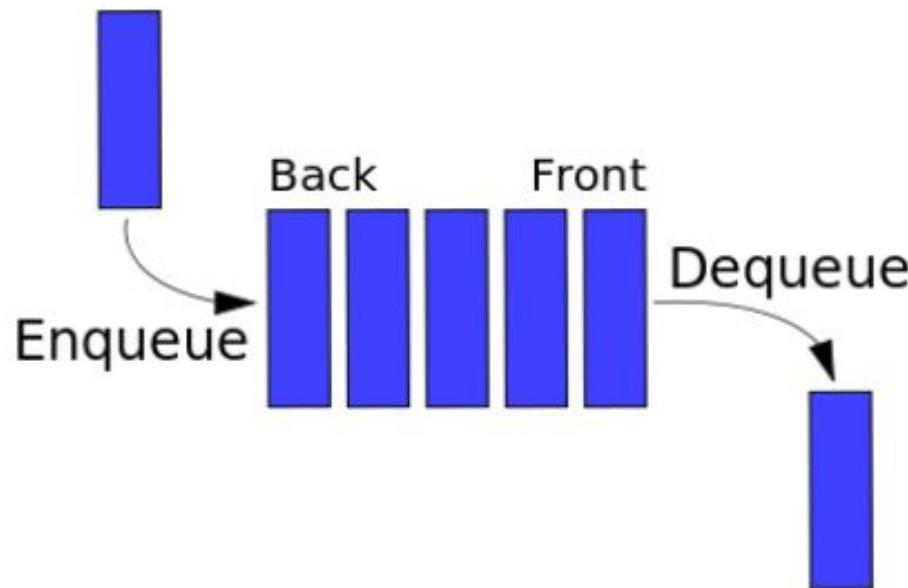
no\_conjunto = {1,1,2,3}

Estructura de datos lineal que sigue la filosofía LIFO (Last in- First Out), es decir, el primer elemento que entra es el primer elemento que sale



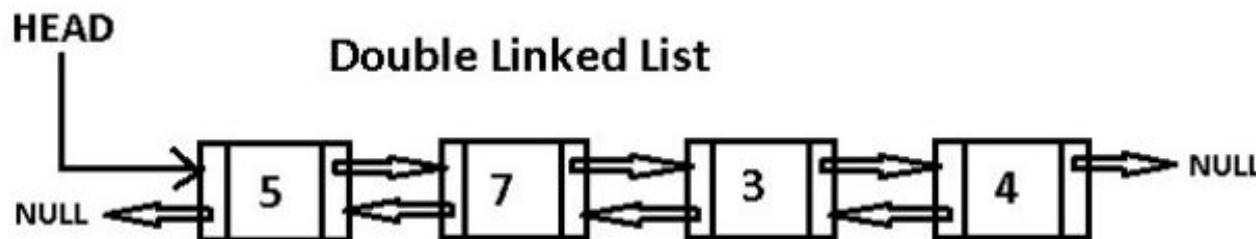
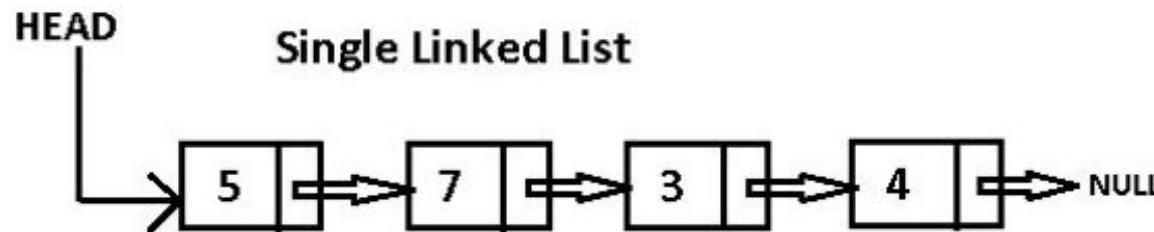
# Cola

Una cola es una estructura de datos lineal que sigue la filosofía FIFO (First in- First Out), es decir, el último elemento en entrar es el primero en salir.



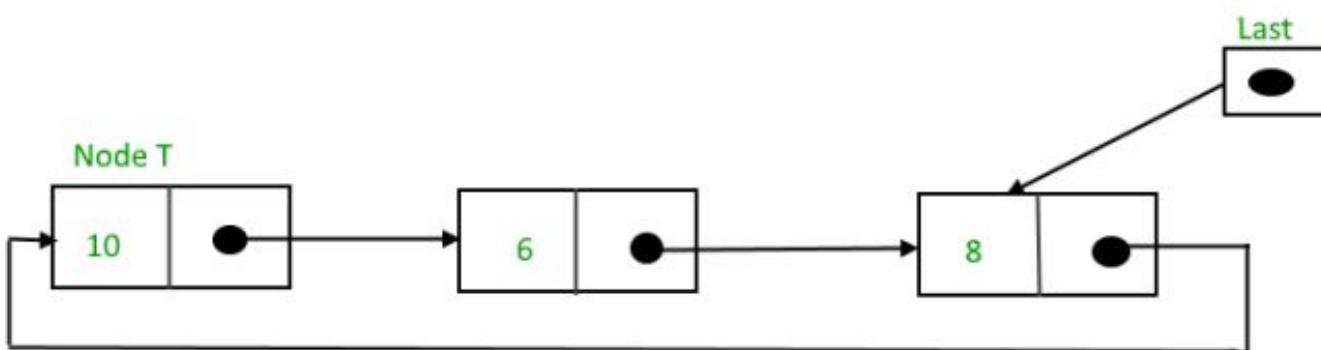
# Listas Enlazadas

Estructuras de datos lineales que almacenan el elemento posterior y pueden almacenar el antecesor, y además almacenan un valor



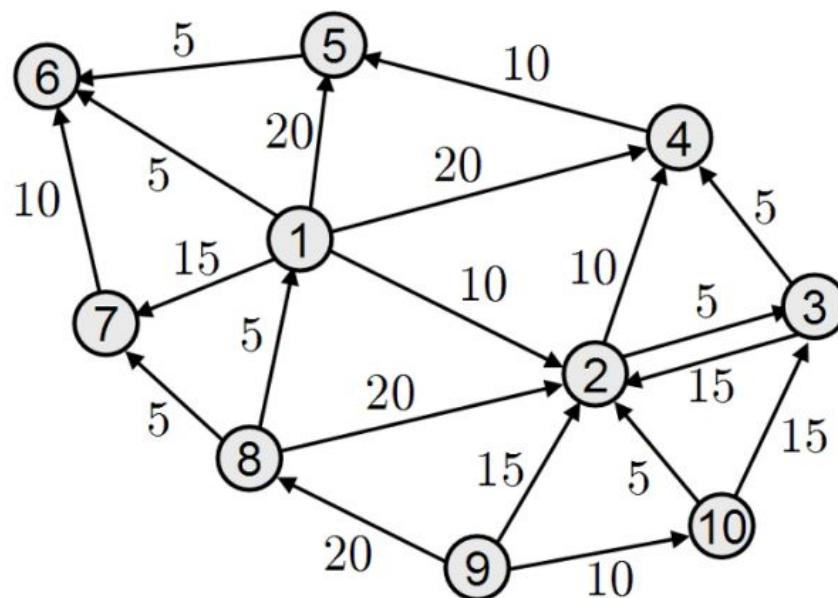
# Lista Ligada Circular

Lista en la que el sucesor del último elemento es el primer elemento.



# Estructuras de Datos no Lineales

Estructuras de datos en las que un elemento puede ir enlazado con dos o más elementos

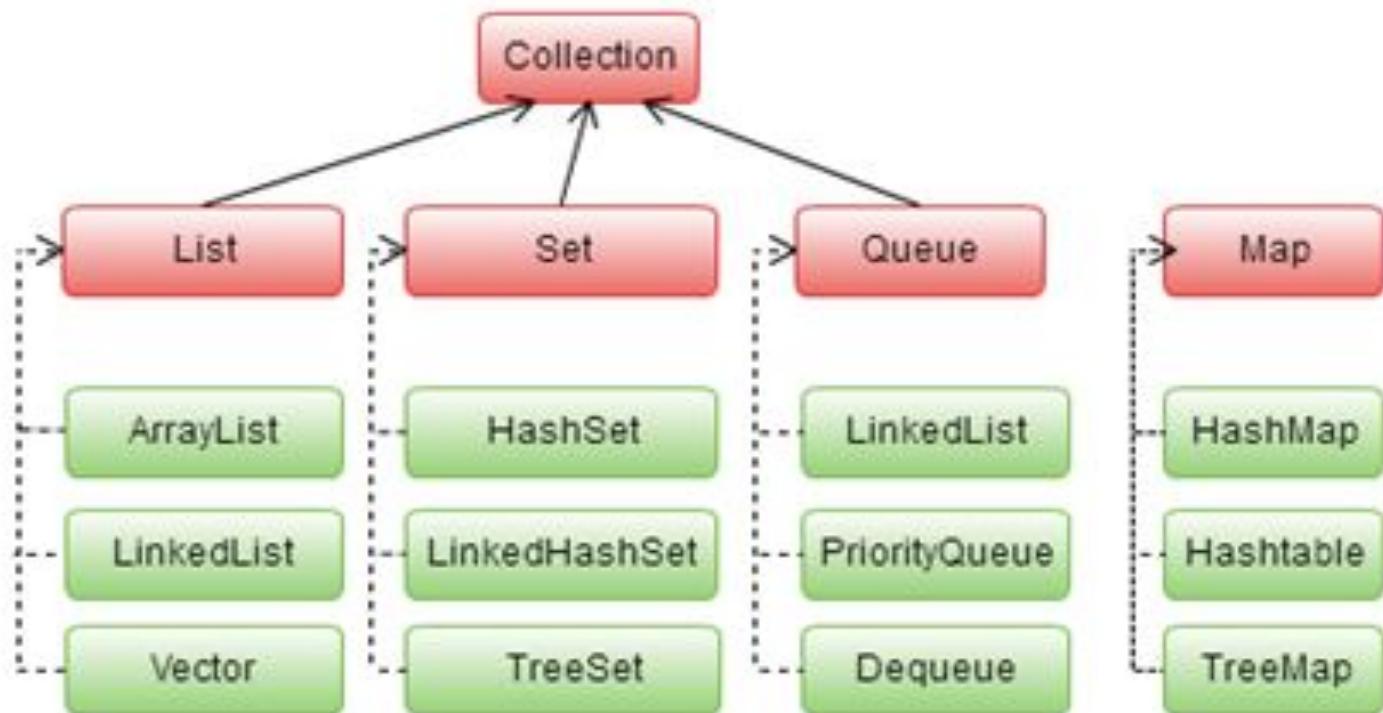


# Java Collections

Una colección es cualquier conjunto de objetos, Java provee una interface Collection, y todas las estructuras lineales la implementan.



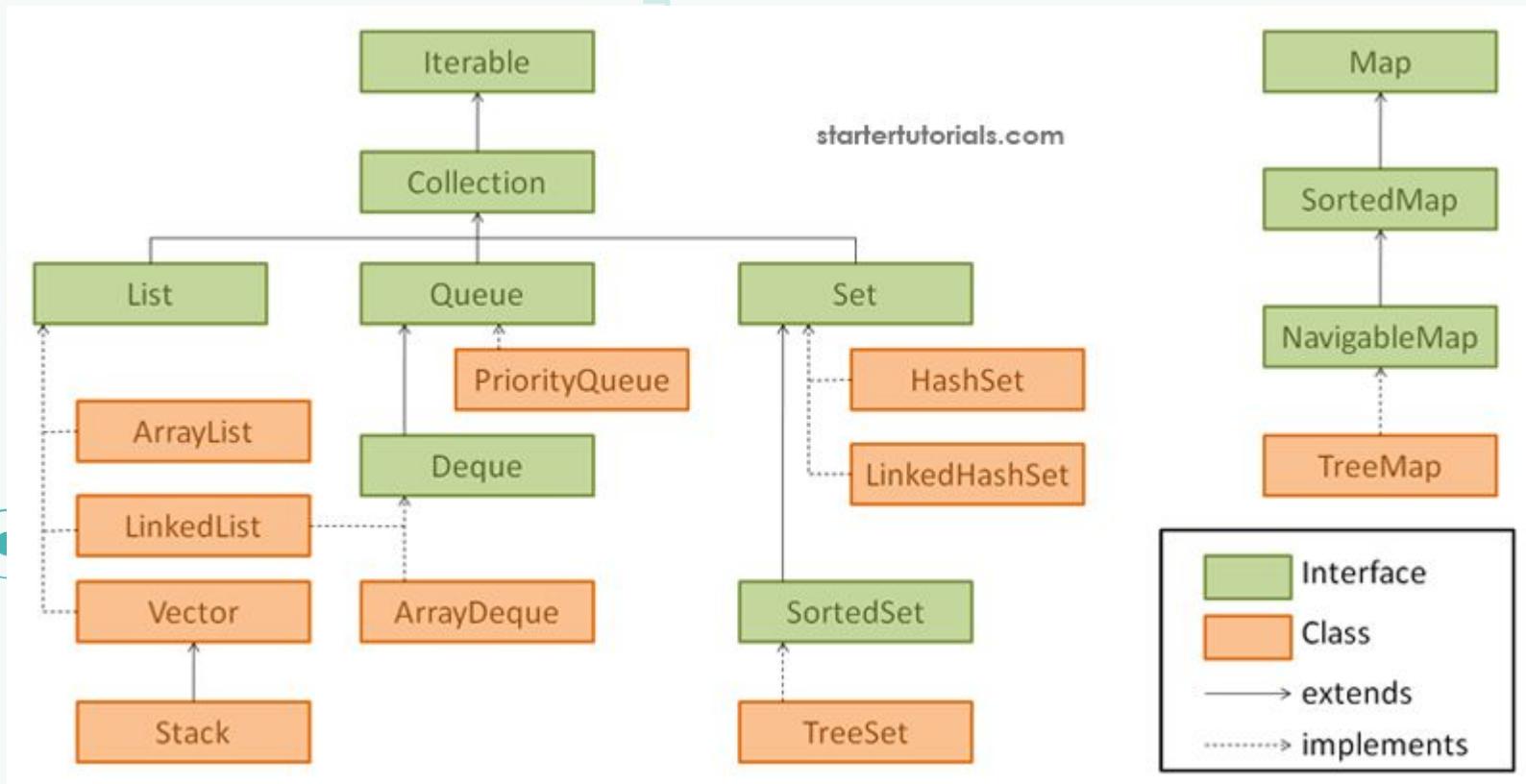
# Colecciones



# Composición

- **Colección de Interfaces:** La base del framework son estas interfaces, que implementan los distintos tipos de colección (estructuras de datos lineales) como pilas, colas, listas, conjuntos.
- **Clases Principales:** Son clases que implementan una o una serie de colección de interfaces y son las que usamos comunmente.
- **Implementaciones Abstractas:** Existen implementacion abstractas de las clases principales, que mediante estas, podemos dar una configuracion específica para su uso.
- **Algoritmos:** Métodos estáticos que permiten realizar acciones muy comunes sobre cualquier objeto creado a partir de una colección del jfc, por ejemplo ordenar una lista.
- **Implementación en Concurrencia:** Existen clases que estan hechas para trabajar con hilos, por ejemplo java.util.Vector

# Diagrama de Clases



- Las principales interfaces usadas son **Collection** y **Map**
- **LinkedList** implementa dos interfaces **List** y **Queue**

# Ventajas

- Reduce el tiempo de programar
- Reduce el tiempo de aprender api's sobre estructuras de datos, puesto que es suficiente aprender una.



# Interfaces base y sus métodos

- **Interfaz Iterator**
- **Interfaz Iterable**
- **Interface Collection**
- **Interfaz Map**
- **Interface List**
- **Interface Queue**
- **Interface Dequeue**
- **Interface Set**

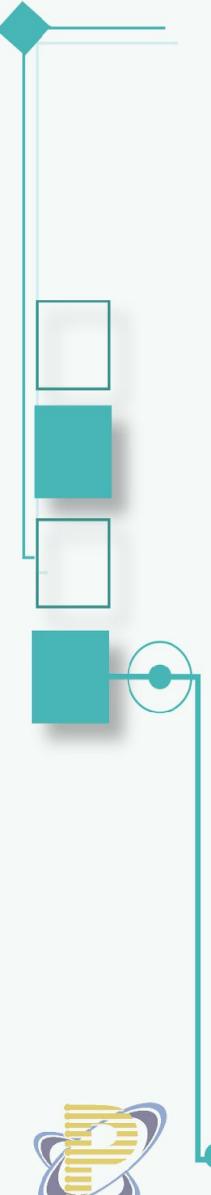
# Interfaz Iterator

La interfaz Iterator<E> permite que un objeto que implementa esta clase posea una serie de elementos, uno en a la vez.

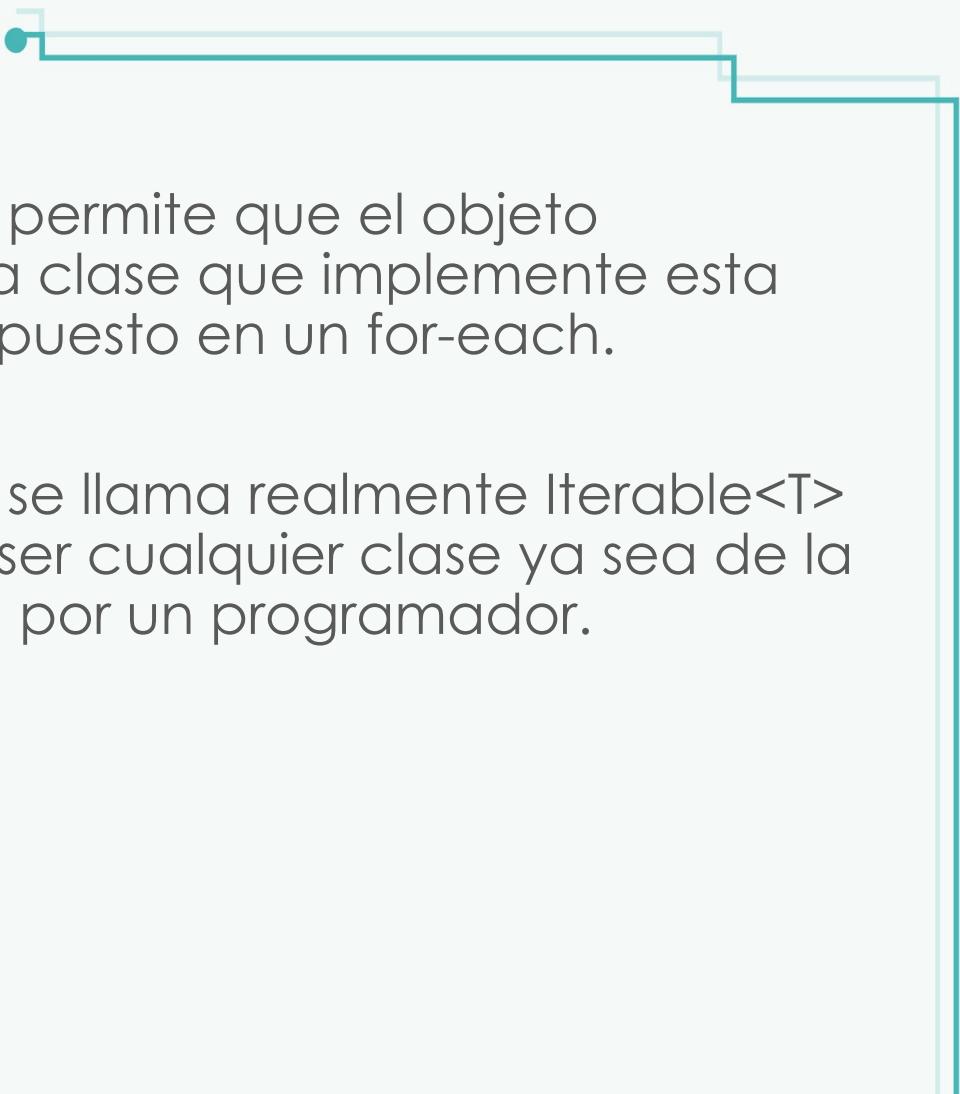
Donde <E> es una clase cualquiera con la que deseamos trabajar. Esta interfaz posee la misma funcionalidad que Enumeration<E>.

Esta interfaz es una mejora de la interfaz Enumeration<E>, se diferencia en que posee métodos con nombres más cortos, y posee un método para eliminar un elemento de la serie de elementos.

# Interfaz Iterable



La interfaz Iterable permite que el objeto instanciado de una clase que implemente esta interfaz pueda ser puesto en un for-each.



La interfaz Iterable se llama realmente Iterable<T> donde <T> puede ser cualquier clase ya sea de la Java Api o creada por un programador.

# Interfaz Collection

La interfaz Collection<E> hereda de Iterable<E> y es la raíz de todas las colecciones del JCF, existen colecciones que están ordenadas otras que no, existen colecciones que son elementos único otras que no.

Collection es el padre de 3 interfaces que cumplen con este objetivo: Set, List, Queue.

# Características Collection

Todas las clases que implementen en algún momento Collection poseen por lo menos 2 constructores:

- *Con parámetros nulos:* Permite la creación de una estructura vacía.
- *Como parámetro un objeto Collection:* Realiza una copia del objeto pasado como parámetro en el objeto a crear.

# Métodos Collection

Dato de Retorno	Cabecera	Descripción
boolean	add(E ejemplo)	Agrega un elemento a la colección. Regresa true si pudo hacerlo.
void	clear()	Borra todos los elementos de la colección
boolean	contains(E ejemplo)	Regresa true si se encuentra el objeto ejemplo en la colección
boolean	isEmpty()	Regresa true si la colección esta vacia.
boolean	remove(Tipo ejemplo)	Borra el objeto ejemplo de la colección, regresa true si lo pudo hacer y false si no existe el objeto en la colección
int	size()	Determina el tamaño de la colección
E[]	toArray()	Regresa un arreglo estático de la colección

# Interfaz Map

La interfaz Map no está relacionada con Collection, porque su filosofía es almacenar elementos que posean una llave única, es decir, almacena de la forma clave-valor.

Cabe resaltar que cada clave es única, y dos o más claves pueden estar asociadas a un mismo valor. Esta interfaz es implementada por varias clases que hablaremos de ellas más adelante.

# Interfaz Map

En una interfaz Map<K, V> K es la clase a las que pertenece el conjunto de llaves y V es la clase a las que pertenece el conjunto de valores. Hay que tener cuidado con el conjunto de claves, puesto que no se pueden repetir, y con usar objetos mutables como llaves.

Dato de Retorno	Cabecera	Descripción
void	clear()	Borra todas las claves-valor del mapa
boolean	containsKey(K clave)	Determina si la clave está en el mapa
boolean	containsValue(V valor)	Determina si existe el valor asociado al mapa
V	get(K clave)	Regresa el valor asociado a la clave pasada como parámetro
boolean	isEmpty()	Regresa true si el mapa está vacío y false en caso contrario
Set <K>	keySet()	Regresa un set con todas las claves del mapa
K	remove(K clave)	Elimina la clave (y su valor asociado) del mapa, y regresa el valor asociado
V	put(K clave, V valor)	Inserta la clave-valor en el mapa y regresa el valor pasado por parámetro
K	replace(K clave, V valor)	Reemplaza el valor de la clave (pasada por parámetro), con el nuevo valor pasada por parámetro, y regresa el antiguo valor
int	size()	Regresa la cantidad almacenada de clave-valor en el mapa
Collection <V>	values()	Regresa una colección con los valores del mapa

# Interfaz List

La interfaz List<E> hereda de la interfaz Collection, por lo que posee todos sus métodos. Un objeto que proviene de List posee una colección de objetos donde se pueden repetir.

Aunque se llame List no necesariamente es un estructura de datos lista. En una interfaz List el usuario puede insertar elementos, eliminar elementos y buscar elementos a su antojo.

# Métodos List

Dato de Retorno	Cabecera	Descripción
E	remove(int indice)	Elimina el objeto ubicado en el índice y lo regresa.
E	set(int indice, E objeto)	Sobreescribe el objeto que se encuentra en el índice con el recibido como parámetro y regresa el anterior objeto.
int	lastIndexOf(Tipo objeto)	Regresa el índice de la última ocurrencia del objeto dado.
int	indexOf(Tipo objeto)	Regresa el índice de la primera ocurrencia del objeto dado.
E	get(int indice)	Regresa el objeto ubicado en el índice dado.

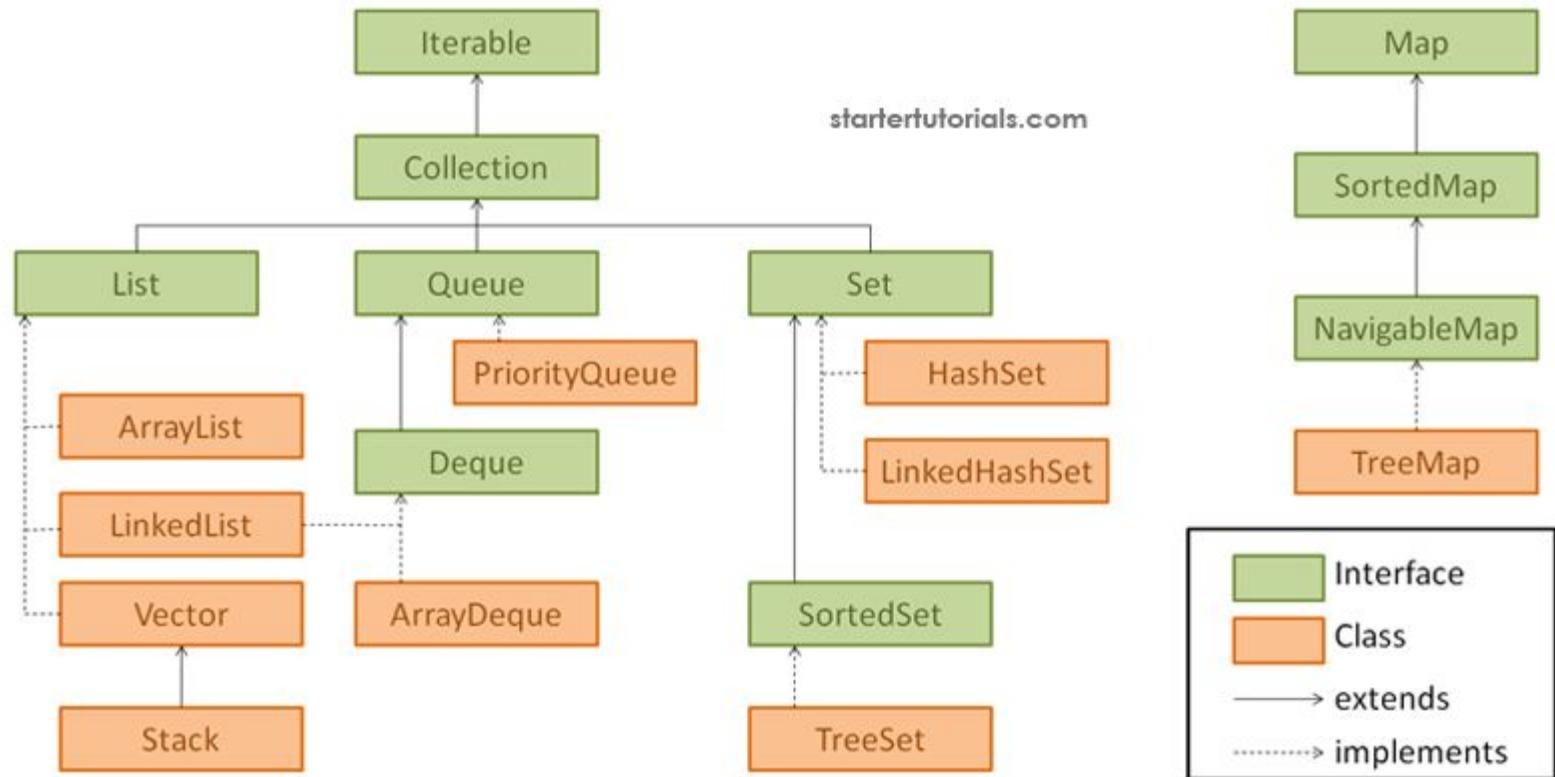
# Interfaz Queue

La interfaz Queue<E> hereda de la interfaz Collection, por lo que posee todos sus métodos. Puede almacenar elementos repetidos, ademas se incluyen métodos para una organización de la información de la forma LIFO.

Hay 6 métodos nuevos, estos se agrupan por parejas puesto que realizan la misma funcionalidad con diferencia de que es lo que ocurre en caso de un fallo.

# Métodos Queue

Funcionalidad	Arroja Excepcion	Regresa Booleano
Insertar	add(E elemento)	offer(E elemento)
Eliminar	remove()	poll()
Ver	element()	peek()



# Interfaz Deque

La interfaz `Deque<E>` hereda de la interfaz `Queue`, y abstrae el concepto de una cola doble. Al ser una cola doble es posible añadir al principio y al final, esto genera nuevos métodos.

Los métodos se clasifican de la misma forma que en la interfaz `Queue`.

# Metodos Dequeue

Los que se aplican sobre el primer elemento:

Acción	Arroja Excepción	Regresa Booleano
Insertar	addFirst(E elemento)	offerFirst(E elemento)
Eliminar	removeFirst()	pollFirst()
Ver	getFirst()	peekFirst()

Los que se aplican sobre el ultimo elemento:

Acción	Arroja Excepción	Regresa Booleano
Insertar	addLast(E elemento)	offerLast(E elemento)
Eliminar	removeLast()	pollLast()
Ver	getLast()	peekLast()

# Interfaz Set

La interfaz Queue<E> hereda de la interfaz Collection, por lo que posee todos sus métodos. Se caracteriza porque almacena elementos únicos.

# Clases provenientes de List

- ArrayList
- LinkedList
- Vector
- Stack

# Clase ArrayList

La clase `ArrayList<E>` implementa `List` por lo que posee todos sus métodos. Un `ArrayList` es la implementación de un arreglo dinámico, las funciones `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` se ejecutan en tiempo constante, pero los métodos `add`, y `remove` tardan de acuerdo al tamaño de elementos de la estructura.

```
import java.util.ArrayList;  
ArrayList<E> arreglo = new ArrayList();
```

# Clase LinkedList

La clase `LinkedList<E>` implementa `List` por lo que posee todos sus métodos. `LinkedList` es la implementación de una lista ligada doble.

```
import java.util.LinkedList;  
LinkedList<E> arreglo = new LinkedList();
```

# Clase Vector

La clase Vector implementa List por lo que posee todos sus metodos. Vector esta hecho para trabajar en un ambiente de hilos, posee 3 atributos:

```
import java.util.Vector;  
LinkedList<E> arreglo = new Vector();
```

Modificador	Identificador	Descripción
protected	capacityIncrement	Entero que indica la capacidad que aumentará cuando supere el tamaño, si es menor a uno incrementara el doble de su tamaño
protected	elementCount	Número que indica la cantidad de elementos del vector
protected	elementData	Arreglo donde estan contenidos todos los elementos del vector

Possee metodos relacionados con el trabajo concurrente:

Tipo		Descripción
void	setSize(int tamano)	Establece el tamaño del vector
E	firstElement()	Regresa el primer elemento del vector
int	capacity()	Regresa la capacidad del vector (cantidad máxima que puede guardar)
int	size()	Regresa la cantidad de elementos que tiene guardado

# Clase Stack

Posee todos los métodos y atributos de Vector, puesto que hereda de ella.

Esta clase abstrae el concepto de una pila, no tenemos que programar propiamente nada, puesto que java nos proporciona todos los métodos para trabajar con ella de una forma excelente.



- Si se trabaja con hilos, lo mejor es ocupar Vector.
- Si no se trabaja con hilos, y se trabaja con poca cantidad de datos, lo mejor es usar ArrayList.
- Si no se trabaja con hilos, y se trabaja con gran cantidad de datos, lo mejor es un LinkedList.
- Stack nunca, mejor usar ArrayDeque

# Clases provenientes de Queue y Dequeue

PriorityQueue  
ArrayDeque  
LinkedList

# PriorityQueue

PriorityQueue implementa la interfaz Queue así que posee todos sus métodos, la filosofía que sigue a la hora de añadir y sacar elementos es la de insertar en donde corresponda (comparando) y sacar el primero (el que sea menor a todos). Esta clase no permite la inserción de elementos nulos ni tampoco permite la inserción de objetos que provengan de una clase que no haya implementado Comparator.

Se llama PriorityQueue porque siempre saca primero el mas importante (el que sea el menor a todos).

# Clase LinkedList

LinkedList además de implementar List implementa Deque, por lo que posee métodos tanto de List como los de Deque.

Al implementar ambas interfaces permite al programador usarlo de diversas formas por la variedad de métodos. Aunque si se desea usar solamente como cola ArrayDeque es una mejor opción.

# vs Queue

En general la elección de uno u otro depende del uso que se quiera dar.

- Si se desea trabajar con objetos que posean una prioridad mayor a otros lo mejor es usar PriorityQueue.
- Si se desea trabajar con una cola ya sea simple o doble lo mejor es usar ArrayQueue.
- Si se desea trabajar con objetos que posean métodos tanto de listas como de colas lo mejor es usar LinkedList.

# Queue

No se puede crear “propiamente” una cola normal puesto que no existe una clase Queue para poder instanciarla, tenemos que ayudarnos de un ArrayDeque para poder usarla.

# Clases proveniente de Set

HashSet  
LinkedHashSet  
TreeSet

# Clase HashSet

HashSet hereda de la clase Set y posee todos sus métodos, esta clase almacena elementos únicos y no acepta repetidos.

HashSet no posee conserva un orden en cuanto a la inserción de elementos, este va cambiando conforme se insertan datos.

# Clase LinkedHashSet

LinkedHashSet hereda de la clase Set y posee todos sus métodos, esta clase almacena elementos únicos y no acepta repetidos.

LinkedHashSet conserva el orden en el cual los elementos fueron insertados.

# Clase TreeSet

TreeSet hereda de la clase Set y posee todos sus métodos, esta clase almacena elementos únicos y no acepta repetidos.

Conforme se insertan los datos en un TreeSet este los va almacenando de menor a mayor, para usar esta estructura los objetos insertados deben de ser Comparator.

# vs Set

En general HashSet es más rápido que  
LinkedHashSet y TreeSet, pero si se desea  
conservar algún tipo de orden específico lo mejor  
será utilizar LinkedHashSet y TreeSet.

# Clases provenientes de Map

HashMap  
LinkedHashMap  
TreeMap

# Clase HashMap

HashMap hereda de la clase Map y posee todos sus métodos, almacena elementos de la forma clave-valor, siendo las claves únicas y los valores se pueden repetir.

HashMapSet no posee conserva un orden en cuanto a la inserción de elementos, este va cambiando conforme se insertan datos.

# Clase LinkedHashMap

LinkedHashMap hereda de la clase Map y posee todos sus métodos, almacena elementos de la forma clave-valor, siendo las claves únicas y los valores se pueden repetir.

LinkedHashMap conserva el orden en el cual los elementos fueron insertados.

# Clase TreeMap

TreeMap hereda de la clase Map y posee todos sus métodos, almacena elementos de la forma clave-valor, siendo las claves únicas y los valores se pueden repetir.

Conforme se insertan los datos en un TreeMap este los va almacenando de menor a mayor, para usar esta estructura los objetos insertados deben de ser Comparator.

# vs Map

En general HashMap es más rápido que LinkedHashMap y TreeMap, pero si se desea conservar algún tipo de orden específico lo mejor será utilizar LinkedHashMap y TreeMap.