Módulo profesional Programación

UD 8 – Estructuras de datos dinámicas y TDA (Tipos de datos abstactos)

Introducción

Si quieres escribir un programa que trate a los objetos (como los sellos de la izquierda) como una colección de elementos tienes varias opciones.

Por supuesto, puedes usar un array, pero los informáticos han inventado otros mecanismos que pueden ser más adecuados para esta tarea.

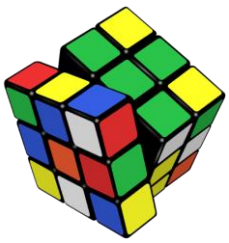
Introducción

En este apartado, presentamos las clases e interfaces que ofrece el **marco de colecciones de Java (JCF, Java Collections Framework)**

Aprenderás a utilizar las clases de este marco y elegir el tipo de colección más apropiado para un problema.

Introducción

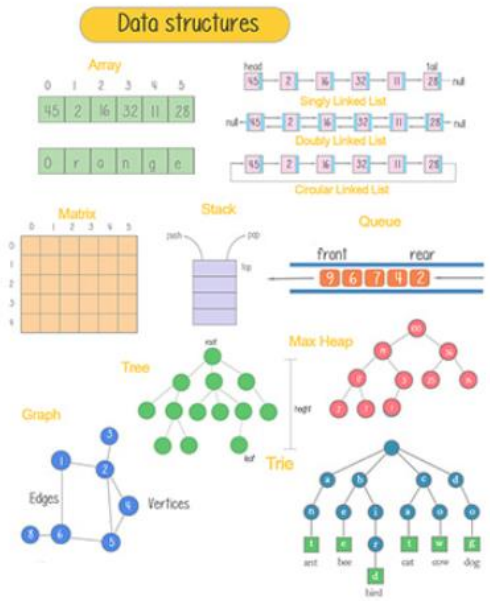
Antes repasemos el concepto de **estructura de datos** y las distintas clasificaciones



Estructuras de datos

Una **estructura de datos** es:

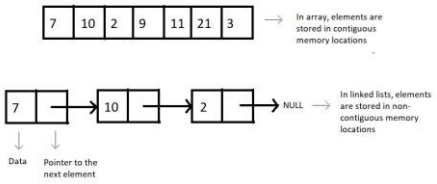
- una **agrupación de datos** que se tratan como una unidad en su conjunto.
- una forma particular de **almacenar y organizar información** para que pueda ser recuperada y utilizada de manera más eficiente.



Clasificación

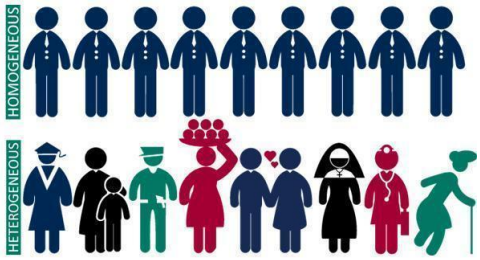
Podemos clasificar a las **estructuras de datos** según:

- Su **ubicación en la memoria del ordenador, contiguas o enlazadas**: las contiguas ocupan posiciones sucesivas de memoria y las enlazadas no necesariamente por lo que para acceder desde un elemento a otro se hace mediante el enlace (dirección de memoria) de donde se ubica el siguiente elemento.



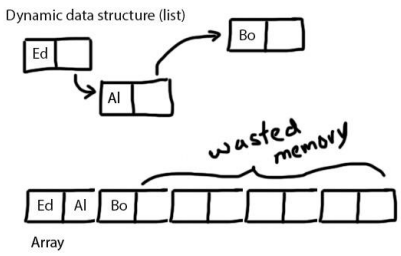
Clasificación

- El **tipo de dato** que contiene, **homogéneas o heterogéneas:** en las homogéneas los elementos son del mismo tipo y las heterogéneas agrupan elementos de distinto tipo.



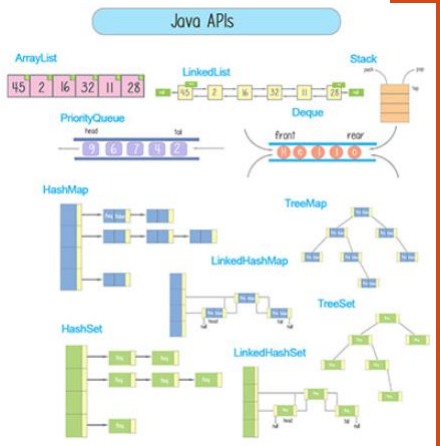
Clasificación

- El **tamaño**, número fijo o puede crecer, **estáticas** o **dinámicas**: el tamaño de las estructuras de datos estáticas no puede crecer en tiempo de ejecución en cambio las dinámicas son aquellas que pueden ir modificando el número de elementos que contienen en tiempo de ejecución a medida que va siendo necesario guardar más información.



Java Collections Framework

JCF (Java Collections Framework*) es una librería que contiene una jerarquía de **interfaces** y **clases** que sirven para almacenar colecciones de **objetos** como listas, conjuntos, mapas, etc.

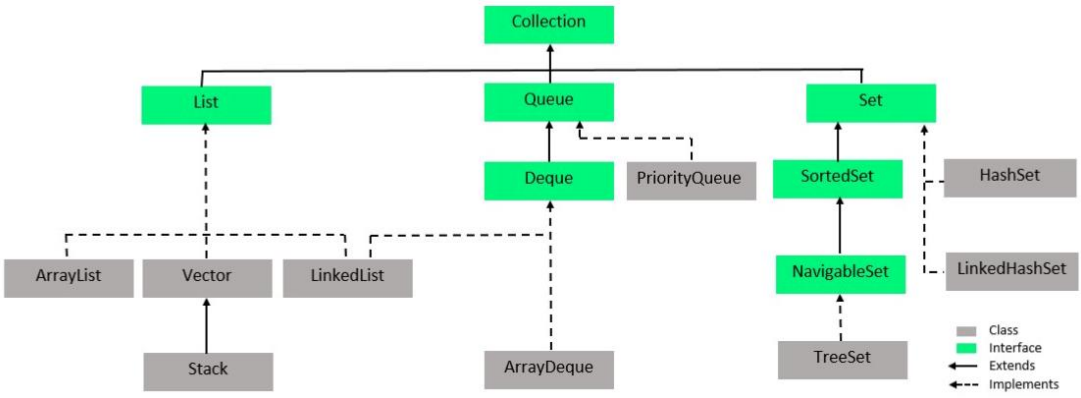


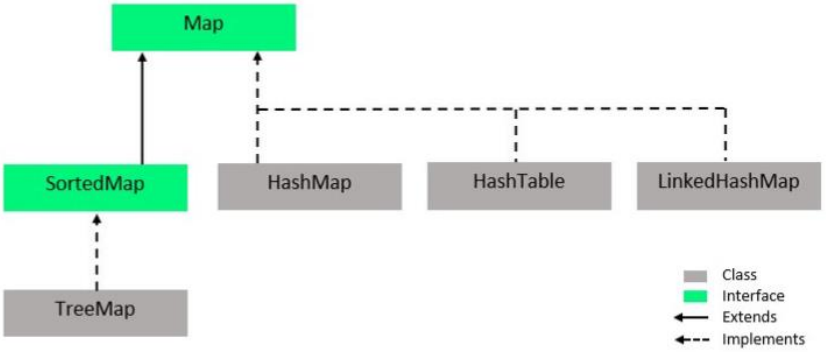
*framework en java: proporciona arquitectura prefabricada de clases e interfaces.

Java Collection Framework

JCF tiene dos **interfaces** principales:

- La interface **Collection**
- La interface **Map**.


Collection
List
Queue
Set
Deque
PriorityQueue
SortedSet
HashSet
ArrayList
Vector
LinkedList
NavigableSet
LinkedHashSet
Stack
ArrayDeque
TreeSet
Class
Interface
Extends
Implements


Map
SortedMap
HashMap
HashTable
LinkedHashMap
TreeMap
Class
Interface
Extends
Implements

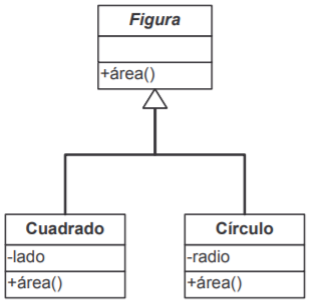
Interfaces

Una **interfaz** en Java es una colección de **métodos** abstractos (sin implementación) y definiciones de **constantes**. Las cases que implementan la interfaz tienen que proveer estos métodos.

En las interfaces se especifica **qué** se debe hacer pero no **cómo**.

Cuando una clase declara que implementa una interfaz, el compilador comprueba que proporciona todos los métodos denegados por la interfaz.

Interfaces




```
public interface Figura {
    public double area();
}
```

```
public class Circulo implements Figura {
    private double radio;

    public Circulo(double radio){
        this.radio=radio;
    }

    public double area (){
        return Math.PI*radio*radio;
    }
}
```

```
public class Cuadrado implements Figura {
    private double lado;

    public Cuadrado(double lado){
        this.lado=lado;
    }

    public double area (){
        return lado*lado;
    }
}
```

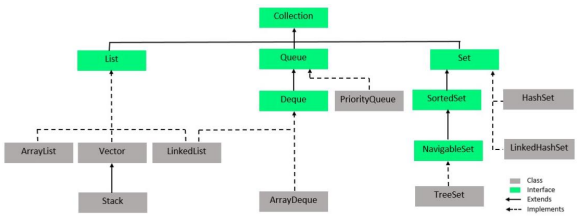
En Java, para indicar que una clase implementa una interfaz se utiliza la palabra reservada **implements**

Interfaces

Por lo tanto, la **interfaz** define un conjunto de comportamientos comunes. Las clases implementan la interfaz de acuerdo con estos comportamientos y proporcionan su propia implementación a los comportamientos.

Si sabes que una **clase implementa una interfaz**, entonces sabes que la clase contiene implementaciones concretas de los métodos declarados en esa interfaz, y se te garantiza que podrás invocar estos métodos de manera segura.

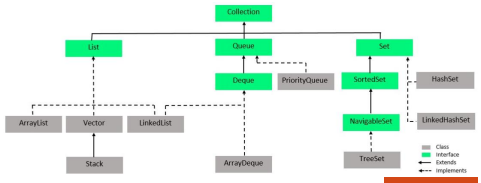
Interfaces



Volviendo a JCF, en la raíz de la jerarquía está la interfaz Collection. Esta interfaz tiene métodos para añadir y eliminar elementos, etc.

Como todas las colecciones implementan esta interfaz, sus métodos están disponibles para **todas las clases**. Por ejemplo, el método size informa del número de elementos en cualquier colección.

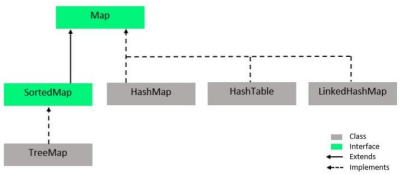
Java Collection Frame



La interfaz de la **Collection** es la interfaz que implementan todas las clases del framework. Declara los métodos que tendrá cada colección. Tiene 3 interfaces hijas:

- **List:** Representa una estructura de datos de tipo lista en la que podemos almacenar una **colección ordenada de objetos**. Puede tener valores **duplicados**. La interfaz de List se implementa mediante las clases ArrayList, LinkedList, Vector y Stack.
- **Set**: Representa el **conjunto no ordenado de elementos** que **no** permite almacenar elementos **duplicados**. Set está implementado por las clases HashSet, LinkedHashSet y TreeSet.
- **Queue**: mantiene un orden FIFO (de primero en entrar, primero en salir). Se puede definir como una lista. Las clases LinkedList, PriorityQueue y ArrayDeque que implementan la interfaz Queue.

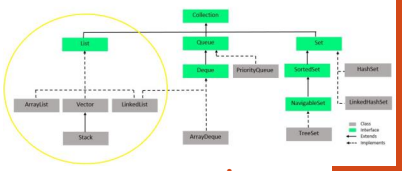
Java Collection Fram



Map: Un mapa asocia valores a una clave, es decir, sus elementos son el par clave y valor. Cada par de **clave y valor** se conoce como una entrada.

Un mapa contiene claves únicas. Un mapa es útil si tiene que buscar, actualizar o eliminar elementos en función de una clave. Las clases Hashtable, LinkedHashMap, HashMap, TreeMap implementan la interfaz Map

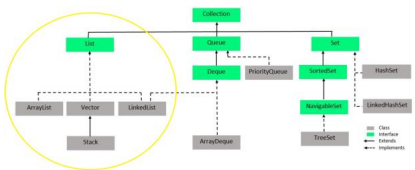
Listas. La interfaz List



Al igual que los arrays, una **lista** es una secuencia ordenada de elementos. Esto significa que podemos acceder a un elemento dentro de una lista por su índice.

La interfaz denota lo que significa ser una Lista, cualquier clase que implemente esta interfaz tiene que proporcionar un conjunto particular de métodos, incluyendo add, get, remove, y unos 20 más.

Listas. La interfaz List



Una lista puede aumentar de tamaño dinámicamente a medida que se agregan elementos. Y además las listas pueden tener elementos duplicados.

Hemos usado los **arrays** tradicionales de Java para almacenar secuencias de elementos. Los arrays no cambian automáticamente su tamaño en tiempo de ejecución para dar cabida a elementos adicionales.

Listas. La interfaz List

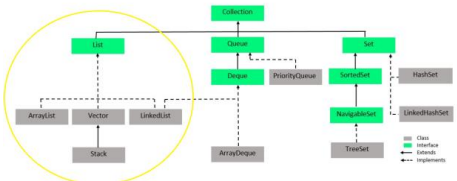
Al ser **List** una interfaz, es necesario instanciar utilizando alguna de sus clases que la implementan para poder utilizarla. Existen varias implementaciones (clases) de la interfaz List y estas implementaciones proveen los métodos definidos en List:

- ArrayList
- LinkedList
- Vector
- Stack

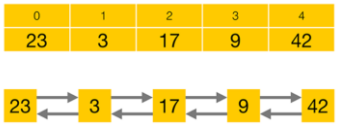
Listas. La interfaz List

ArrayList y LinkedList proporcionan los métodos definidos en List, por lo que se pueden utilizar de forma de forma intercambiable.

Un método escrito para trabajar con una List funcionará con una ArrayList, LinkedList, o cualquier otro objeto que implemente List.



Listas. La interfaz List



Las clases que implementan la interfaz List son muy similares a usar. Su principal diferencia es su implementación, lo que causa un rendimiento diferente para diferentes operaciones.

- **ArrayList**: se implementa como un **array** de tamaño variable. A medida que se agregan más elementos a ArrayList, su tamaño aumenta dinámicamente.
- **LinkedList**: se implementa como una lista doblemente enlazada. Esto significa que cada elemento de la lista tiene una referencia a los elementos siguiente y anterior de la lista. Esta clase también implementa la interface **Queue**

Listas. La interfaz List

- **Vector**: Vector es casi idéntico a ArrayList, y la diferencia es que Vector está **sincronizado**. Tiene todos sus métodos **syncronized** previstos para la programación concurrente. Debido a esto, tiene un desempeño peor que ArrayList.
- **Stack**: Esta clase hereda de la Vector y con 5 operaciones permite tratar un vector a modo de pila.

Métodos para trabajar con listas (list)

`void` **`add(int indice, Object obj)`**

Inserta un elemento (objeto) en la posición de la lista dada por el índice indicado.

`Object` **`get(int indice)`**

Obtiene el elemento (objeto) de la posición de la lista dada por el índice indicado.

`int` **`indexOf(Object obj)`**

Nos dice cual es el índice de dicho elemento (objeto) dentro de la lista. Nos devuelve -1 si el objeto no se encuentra en la lista.

`Object` **`remove(int indice)`**

Elimina el elemento que se encuentre en la posición de la lista indicada mediante dicho índice, devolviéndonos el objeto eliminado.

`Object` **`set(int indice, Object obj)`**

Establece el elemento de la lista en la posición dada por el índice al objeto indicado, sobrescribiendo el objeto que hubiera anteriormente en dicha posición. Nos devolverá el elemento que había previamente en dicha posición.

La clase ArrayList

La clase **Arraylist** es un array de **dimensión modificable** y ofrece dos beneficios significativos:

- Los ArrayList pueden aumentar y disminuir según sea necesario. No es necesario declarar su tamaño como pasa con los arrays.
- La clase ArrayList nos provee métodos para muchas tareas comunes, tales como la inserción y eliminación de elementos.

La clase ArrayList

De forma general un **ArrayList** en Java se crea de la siguiente forma:

```
ArrayList nombreArray = new ArrayList();
```

Esta instrucción crea el ArrayList nombreArray vacío.

La clase ArrayList

Un ArrayList declarado así puede contener objetos de cualquier tipo. Por ejemplo:

```
ArrayList lista = new ArrayList();
lista.add("Lenguaje");
lista.add(3);
lista.add('a');
```

Los elementos del arrayList **lista** son: "Lenguaje" 2 'a'.

Es decir, un ArrayList puede contener objetos de tipos distintos.

La clase ArrayList

En este ejemplo, el primer **objeto** que se añade es el String "Lenguaje".

El **resto "no son objetos".** Son datos de tipos primitivos pero el compilador los convierte automáticamente en objetos de su **clase envolvente** (clase contenedora o wrapper) antes de añadirlos al array.

```
ArrayList lista = new ArrayList();
lista.add("Lenguaje");
lista.add(3);
lista.add('a');
```

La clase ArrayList

Un array al que se le pueden asignar elementos de distinto puede tener alguna complicación a la hora de trabajar con él. Por eso, una alternativa a esta **declaración es indicar el tipo de objetos que contiene**. En este caso, el array solo podrá contener objetos de ese tipo.

De forma general:

```
ArrayList<tipo> nombreArray = new ArrayList<>();
```

tipo debe ser una clase. Indica el tipo de objetos que contendrá el Arraylist

La clase ArrayList

No se pueden usar tipos primitivos. Para un tipo primitivo se debe utilizar su clase envolvente. Por ejemplo:

```
ArrayList<Integer> numeros = new ArrayList<>();
```

Crea una lista de números Integer.

```
ArrayList<String> cadenas = new ArrayList<>();
```

Crea una lista de cadenas

Nota: Se le denomina operador diamante por la forma que tiene el operador "<>"

La clase ArrayList

Uso de ArrayList como List

```
List<String> ejemploLista = new ArrayList<>();
```

(más adelante explicaremos porqué hay que utilizarlo de esta manera)

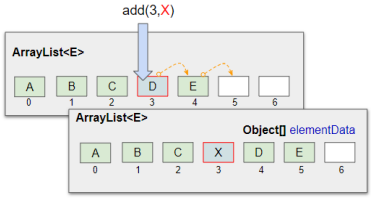
Con nuestra lista creada podemos empezar a introducir datos en ella.

Supongamos que queremos agregar los siguientes nombres: Juan, Pedro, José, María, Sofía.

```
ejemploLista.add("Juan");
ejemploLista.add("Pedro");
ejemploLista.add("José");
ejemploLista.add("María");
ejemploLista.add("Sofía");
```

Actividad

Agregar elementos




```
[Steve, Tim, Lucy, Pat, Angela, Tom]
[Steve, Tim, Lucy, Charlie, Pat, Angela, Tom]
```

```java
import java.util.ArrayList;
import java.util.List;

public class ArrayListExample {

    public static void main(String[] args) {

            List<String> aList = new ArrayList<>();
            aList.add("Steve");
            aList.add("Tim");
            aList.add("Lucy");
            aList.add("Pat");
            aList.add("Angela");
            aList.add("Tom");

            //mostranto los elementos
            System.out.println(aList);

            //agregando un elemento en la cuarta posición
            aList.add(3,"Charlie");

            //mostrando los elementos
            System.out.println(aList);
    }
}
```

La clase ArrayList

Podemos obtener la cantidad de elementos que posee la lista:

```java
ejemploLista.size();
```

```java
List<String> ejemploLista = new ArrayList<>();
```

Para consultar la lista se utiliza:

```java
ejemploLista.get(0);
```

```java
ejemploLista.add("Juan");
ejemploLista.add("Pedro");
ejemploLista.add("José");
ejemploLista.add("María");
ejemploLista.add("Sofía");
```

Donde 0 es el índice en el que se encuentra la información que queremos. En este caso, el índice 0 vendría siendo Juan

La clase ArrayList

```
List<String> ejemploLista = new ArrayList<>();
ejemploLista.add("Juan");
ejemploLista.add("Pedro");
ejemploLista.add("José");
ejemploLista.add("María");
ejemploLista.add("Sofía");
```

Si queremos eliminar determinado elemento:

```
ejemploLista.remove(0);
```

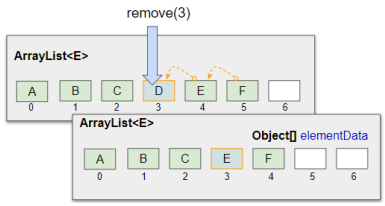
El número representa el índice que queremos eliminar.

Otra forma de eliminar un elemento es por su nombre:

```
ejemploLista.remove("Juan");
```

Actividad

Remover elementos




```
[Steve, Tim, Lucy, Pat, Angela, Tom]
[Tim, Lucy, Pat, Tom]
[Tim, Lucy, Tom]
```

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListRemoveExample {

    public static void main(String[] args) {

            List<String> aList = new ArrayList<>();
            aList.add("Steve");
            aList.add("Tim");
            aList.add("Lucy");
            aList.add("Pat");
            aList.add("Angela");
            aList.add("Tom");

            //mostranto los elementos
            System.out.println(aList);

            //eliminando elementos
            aList.remove("Steve");
            aList.remove("Angela");

            //mostrando los elementos
            System.out.println(aList);

            //eliminando el tercer elemento
            aList.remove(2);

            //mostrando los elementos
            System.out.println(aList);

    }
}
```

La clase ArrayList

Podemos recorrer un ArrayList mediante:

- Un bucle (for, while, do-while)
- Un bucle for-each
- Un objeto iterador y un bucle
- El método forEach() con expresiones lambda (**new)**

La clase ArrayList

Podemos recorrerlo de forma clásica con un bucle for:

```
for(int i = 0;i<ejemploLista.size();i++)
    System.out.println(ejemploLista.get(i));
```

La clase ArrayList

Con un bucle **for each**:

```
List<Integer> aList = new ArrayList<>();

aList.add(14);
aList.add(7);
aList.add(39);
aList.add(40);

for (Integer num: aList)
    System.out.println(num);
```

Actividad

Recorriendo una lista

```
import java.util.ArrayList;
import java.util.List;

public class IteratingArrayListExample {

    public static void main(String[] args) {

            List<String> aList = new ArrayList<>();

            aList.add("Steve");
            aList.add("Tim");
            aList.add("Lucy");
            aList.add("Pat");
            aList.add("Angela");
            aList.add("Tom");

            for (String name: aList)
                System.out.println(name);

    }
}
```

La clase ArrayList

Utilizando un objeto **Iterator**. Cada colección proporciona un iterador que son objetos que están acoplados con ella. Para iterar una colección se debe tener primero una referencia al objeto iterador.

La ventaja de utilizar un **Iterador** es que no necesitamos indicar el tipo de objetos que contiene la lista. Iterator tiene como métodos:

- **hasNext**: devuelve true si hay más elementos en la lista.
- **next**: devuelve el siguiente objeto contenido en la lista.

La clase ArrayList

Recorrer un ArrayList con un iterador

```
List<Integer> aList = new ArrayList<>();

aList.add(14);
aList.add(7);
aList.add(39);
aList.add(40);

Iterator it = aList.iterator();
while (it.hasNext())
    System.out.println(it.next());
```

Actividad

Recorrer una lista

- bucle for
- bucle foreach
- while
- Iterador

```
Bucle FOR
14
7
39
40
Bucle FOR-EACH
14
7
39
40
Bucle WHILE
14
7
39
40
Iterador
14
7
39
40
```

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Iterator;

public class IteratingArrayListExample2 {

    public static void main(String[] args) {

        List<Integer> aList = new ArrayList<>();

        aList.add(14);
        aList.add(7);
        aList.add(39);
        aList.add(40);

        /* bucle for */
        System.out.println("Bucle FOR");
        for(int i=0;i<aList.size();i++)
            System.out.println(aList.get(i));

        /* bucle for-each */
        System.out.println("Bucle FOR-EACH");
        for (Integer num: aList)
            System.out.println(num);

        /* bucle while */
        System.out.println("Bucle WHILE");
        int i=0;
        while (aList.size()>i){
            System.out.println(aList.get(i));
            i++;
        }

        /* iterador */
        System.out.println("Iterador");
        Iterator it = aList.iterator();
        while (it.hasNext())
            System.out.println(it.next());

    }
}
```

La clase ArrayList

Recorrer un ArrayList con el método **forEach** con expresiones lambda

```java
import java.util.ArrayList;
import java.util.List;

public class IteratingArrayListExample3 {

    public static void main(String[] args) {

            List<String> listaNombres = new ArrayList<>();
            listaNombres.add("Steve");
            listaNombres.add("Tim");
            listaNombres.add("Lucy");
            listaNombres.add("Pat");
            listaNombres.add("Angela");
            listaNombres.add("Tom");

            listaNombres.forEach(nombre -> System.out.println(nombre));

    }
}
```

La clase ArrayList

```
listaNombres.forEach(nombre -> System.out.println(nombre));
```

¡Increíble! Este código parece aún más compacto y más legible el bucle for-each. Podemos leer la línea anterior así: por cada elemento de listaNombres, imprime el valor de la variable nombre.

Desde Java 8, cada colección tiene un método forEach() que implementa la iteración internamente. Ten en cuenta que este método toma una expresión Lambda o, en otras palabras, los programadores pueden pasar su código -o función- a este método. Como se muestra en el ejemplo anterior, el código para imprimir cada elemento se pasa al método.

Si no conoces las expresiones Lambda, consulta este tutorial.

No veremos expresiones lambda ni programación funcional en este curso.

```
Exception in thread "main" java.util.ConcurrentModificationException
        at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1013)
        at java.base/java.util.ArrayList$Itr.next(ArrayList.java:967)
        at RemovingArrayListExample.main(RemovingArrayListExample.java:16)
```

La clase ArrayList

Remover un elemento de un ArrayList

El resultado usando un bucle **for each** es más legible. Sin embargo en ocasiones los iteradores aportan cosas interesantes que los bucles for each no pueden abordar.

Vamos a borrar un los objeto de una lista con un nombre concreto

```
List<String> aList = new ArrayList<>();

aList.add("Damasco");
aList.add("Banana");
aList.add("Ciruela");
for (String fruta: aList)
    aList.remove("Banana");
```

No funciona y lanza una excepción **ConcurrentModificationException**

Como podemos ver, antes de terminar nuestra iteración estamos eliminando un elemento. Eso es lo que desencadena la excepción.

La clase ArrayList

Remover un elemento de un ArrayList

Soluciones

1) Usar un iterador

la interfaz **Iterator** dispone de un método adicional que permite eliminar objetos de una lista mientras la recorremos. Es el método **remove**.

```
Iterator it = aList.iterator();
while (it.hasNext()){
    String fruta = (String)it.next();
    System.out.println(fruta);
    if (fruta.equals("Banana"))
        it.remove();
}
```

La clase ArrayList

2) No remover durante la iteración

Si queremos mantener nuestro bucle for-each, entonces podemos hacerlo. Es solo que necesitamos esperar hasta después de iterar antes de eliminar los elementos. Probemos esto agregando lo que queremos eliminar a una lista **toRemove** a medida que iteramos:

```
List<String> aList = new ArrayList<>();
List<String> toRemove = new ArrayList<>();

aList.add("Damasco");
aList.add("Banana");
aList.add("Ciruela");
for (String fruta: aList)
    if (fruta.equals("Banana"))
        toRemove.add(fruta);

aList.removeAll(toRemove);
```

La clase ArrayList

Para **eliminar** todos los elementos de la lista usamos:

```
ejemploLista.clear();
```

Si deseamos saber si nuestra lista contiene algún elemento utilizamos:

```
ejemploLista.isEmpty();
```

Esto nos devolverá un true o un false.

La clase ArrayList

```
List<String> ejemploLista = new ArrayList<>();
ejemploLista.add("Juan");
ejemploLista.add("Pedro");
ejemploLista.add("José");
ejemploLista.add("María");
ejemploLista.add("Sofía");
```

En caso de que contenga algún elemento podemos verificar si entre esos elementos tenemos alguno en específico. Por ejemplo si queremos saber si en nuestra lista está la cadena José, utilizamos:

```
ejemploLista.contains("José");
```

Esto también nos devolverá un true o un false. Y si por alguna razón queremos modificar algún dato de nuestra lista, por ejemplo el índice 1 que contiene la cadena Pedro, utilizamos el siguiente método:

```
ejemploLista.set(1, "Félix");
```

Habremos cambiado la cadena en el índice 1 (Pedro) por la nueva cadena (Félix).

La clase ArrayList

Si queremos extraer una lista que contenga los elementos entre un índice y otro podemos utilizar:

```
ejemploLista.subList(0, 2);
```

Actividad

Sublistas

```
import java.util.ArrayList;
import java.util.List;

public class SublistExample {

    public static void main(String[] args) {

            List<String> aList = new ArrayList<>();
            aList.add("Steve");
            aList.add("Tim");
            aList.add("Lucy");
            aList.add("Pat");
            aList.add("Angela");
            aList.add("Tom");

            //mostranto los elementos
            System.out.println("Elementos de la lista original: " + aList);

            //Sublista del ArrayList
            List<String> aSublist = new ArrayList(aList.subList(1,4));
            System.out.println("Sublista almacenada en un ArrayList: " + aSublist);


            //Sublista a List
            List<String> anotherList = aList.subList(1,4);
            System.out.println("Sublista almacenada en un List: " + anotherList);

    }
}
```

Actividad

Escribe un programa que lea números enteros y los almacene en un ArrayList hasta que se lea un 0. Debe mostrar la cantidad de números almacenados, su suma y su media.

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class ArrayListActivity {

    public static void main(String[] args) {

            Scanner sc = new Scanner(System.in);
            List<Integer> numeros = new ArrayList<>();
            int suma=0, n;
            do {
                System.out.println("Introduce números enteros -0 para terminar");
                System.out.println("Número: ");
                n = sc.nextInt();
                if (n!=0){
                    numeros.add(n);
                    suma += n;
                }
            } while (n!=0);

            int cantidad = numeros.size();
            System.out.println("Ha introducido " + cantidad + " números");
            System.out.println("Suma: " + suma);
            int media = 0;
            if (cantidad != 0)
                media = suma/cantidad;
            System.out.println("Media: " + media);

    }
}
```

Conversiones entre List y array

Conversión de array a ArrayList se hace utilizando **Arrays.asList()**

Se debe indicar, además, que se requiere la utilización de objetos y no funciona directamente con arrays de tipos primitivos.

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Arrays;

public class ArrayToArrayList {

    public static void main(String[] args) {

        //Declaración e inicialización de un array
        String[] ciudades = {"Cádiz","Sevilla","Córdoba","Huelva"};

        //Conversión de array a ArrayList
        List<String> listaCiudades = new ArrayList<>(Arrays.asList(ciudades));

        //Mostrar elementos
        System.out.println(listaCiudades);

        //Agregar ciudades a la lista
        listaCiudades.add("Granada");
        listaCiudades.add("Almería");
        listaCiudades.add("Jaén");

        //Mostrar elementos
        System.out.println(listaCiudades);

    }
}
```

Conversiones entre List y array

La conversión de cualquier **lista a un array de cadenas** se resuelve utilizando el método **toArray()** que implementan todas las clases de la interfaz List.

https://www.geeksforgeeks.org/arraylist-array-conversion-java-toarray-methods/

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Arrays;

public class ArrayListToStringArray {

    public static void main(String[] args) {

        //Declaración e inicialización de un array
        List<String> amigos = new ArrayList<>();

        //Agregar elementos
        amigos.add("Ana");
        amigos.add("Gabriel");
        amigos.add("Gema");
        amigos.add("Antonio");

        //Conversión de ArrayList a list
        String [] vectorAmigos = amigos.toArray(new String[amigos.size()]);

        //Mostrar elementos
        System.out.println(Arrays.toString(vectorAmigos));
    }
}
```

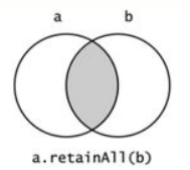
Otras operaciones con listas (List)

lista1:[1, 2, 3]

lista2:[1, 3, 4, 5]

lista2.**retainAll**(lista1);

lista2:[1, 3]

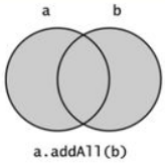


lista1:[1, 2, 3]

lista2:[1, 3, 4, 5]

lista2.**addAll**(lista1);

lista2:[1, 3, 4, 5, 1, 2, 3]

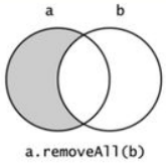


lista1:[1, 2, 3]

lista2:[1, 3, 4, 5]

lista2.**removeAll**(lista1);

lista2:[4,5]



Array vs ArrayList

	Size	They can hold	Iteration	How to get size?	Multi-dimensional	How to add elements?
Array	Fixed	Primitives as well as objects	Only through *for* loop or *for-each* loop	*Length* attribute	Yes	Using assignment operator
ArrayList	Re-sizable	Only objects	Iterators or *for* loop or *for-each* loop	*size()* method	No	Using *add()* method

Array vs ArrayList

Para la mayoría de las tareas de programación, los ArrayList son más fáciles de usar que los arrays. Los ArrayList pueden crecer y reducirse. Por otro lado, los arrays tienen una sintaxis más agradable para el acceso a los elementos y la inicialización de elementos.

¿Cuál de los dos debería elegir?

Array vs ArrayList

Aquí tienes algunas recomendaciones.

- Si el tamaño de una colección no cambia nunca, utiliza un array.
- Si agrupa una larga secuencia de valores de tipo primitivo y te preocupa la eficiencia, utiliza un array.
- En caso contrario, utiliza ArrayList

Longitud y tamaño

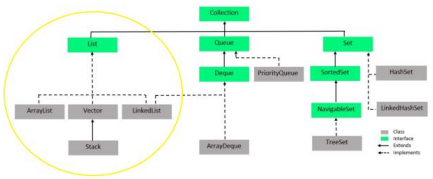
Common Error 7.4

Length and Size

Unfortunately, the Java syntax for determining the number of elements in an array, an array list, and a string is not at all consistent. It is a common error to confuse these. You just have to remember the correct syntax for every data type.

Data Type	Number of Elements
Array	`a.length`
Array list	`a.size()`
String	`a.length()`

La clase LinkedList



Java proporciona una estructura de datos de tipo lista completamente dinámica implementada con elementos doblemente enlazados internamente. Sus nodos o elementos contienen un valor y un puntero al anterior y al siguiente elemento en la lista.

Each node in a linked list is connected to the neighboring nodes.

La clase LinkedList

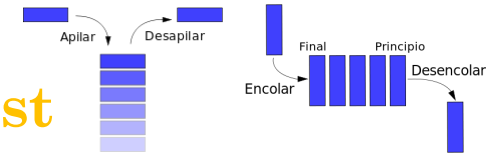
La clase **LinkedList** funciona de manera similar a una **lista doblemente enlazada**

La clase LinkedList

LinkedList tiene los mismos métodos que todas las List de Java. Los principales son:

`void add(Object)`	- Añade un elemento nuevo en el primer sitio libre.
`void add(int, Object)`	- Inserta un elemento en la posición indicada por índice.
`void clear()`	- Elimina todos los elementos de la lista.
`boolean contains(Object)`	- Comprueba si el elemento está incluido en la lista.
`Object get(int)`	- Obtiene el elemento de la posición indicada por índice.
`int indexOf(Object)`	- Devuelve la posición del elemento.
`boolean isEmpty()`	- Comprueba si la lista está vacía.
`boolean remove(Object)`	- Elimina el elemento correspondiente.
`Object remove(int)`	- Elimina el elemento de la posición indicada por índice.
`int size()`	- Devuelve el número de elementos almacenados.

La clase LinkedList



`void` **`addFirst(Object obj)`** / `void` **`addLast(Object obj)`**

Añade el objeto indicado al principio / final de la lista respectivamente.

`Object` **`getFirst()`** / `Object` **`getLast()`**

Obtiene el primer / último objeto de la lista respectivamente.

`Object` **`removeFirst()`** / `Object` **`removeLast()`**

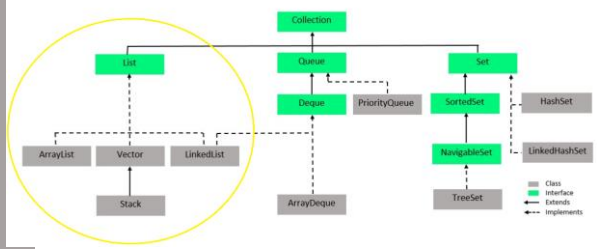
Extrae el primer / último elemento de la lista respectivamente, devolviéndonos dicho objeto y eliminándolo de la lista.

Hemos de destacar que estos métodos nos permitirán trabajar con la lista como si se tratase de una **dequeue** o de una **queue**.

En el caso de la pila realizaremos la inserción y la extracción de elementos por el mismo extremo, mientras que para la cola insertaremos por un extremo y extraeremos por el otro.

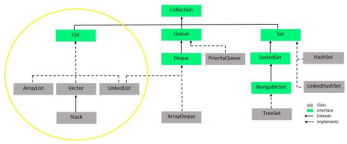
Actividad

Ejemplo de LinkedList




```java
import java.util.LinkedList;
import java.util.List;
import java.util.Iterator;

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

      LinkedList<String> list=new LinkedList<>();

      //Adding elements to the Linked list
      list.add("Steve");
      list.add("Carl");
      list.add("Raj");

      //Adding an element to the first position
      list.addFirst("Negan");

      //Adding an element to the last position
      list.addLast("Rick");

      //Adding an element to the 3rd position
      list.add(2, "Glenn");

      //Iterating LinkedList
      Iterator<String> iterator=list.iterator();
      while(iterator.hasNext()){
        System.out.println(iterator.next());
      }
   }
}
```

Arraylist vs Linkedlist



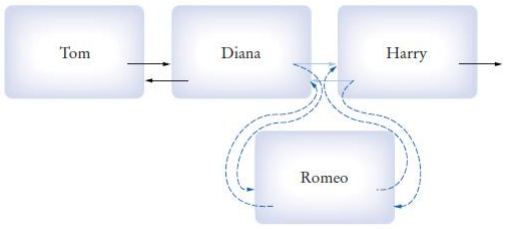
LinkedList debe ser preferido sobre **ArrayList** si:

- No hay gran cantidad de acceso aleatorios. Acceso es principalmente secuencial.
- Hay un gran número de operaciones de agregar/remover elementos en los extremos de la lista

En cambio utilizar **ArrayList**

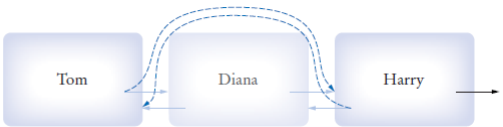
- El acceso es aleatorio. El acceso por índices es muy rápido. Se accede a la misma velocidad a cada elemento, independientemente del total.

Inserciones/eliminaciones en LinkedList



Inserting a Node into a Linked List

Removing a Node from a Linked List



¿Cuál es la diferencia entre ambas declaraciones?

```
ArrayList<Integer> lista1 = new ArrayList<>();

List<Integer> lista2 = new ArrayList<>();
```

En la declaración 2) **List** es una interfaz y **ArrayList** es una implementación de esta interfaz.

Para el tipo de una variable, es mejor usar la interfaz que implementa. De esa manera, puedes cambiar la implementación más adelante sin tener que cambiar el resto del código. Es decir, puedes cambiar la declaración de ArrayList a LinkedList de forma transparente.

```
List<Integer> lista2 = new LinkedList<>();
```

Programación basada en la interfaz

Este estilo se denomina "**programación basada en la interfaz**" o, más informalmente, "programación para una interfaz"

Cuando se utiliza una librería, el código debe depender únicamente de la interfaz, como List. No debería depender de una implementación específica, como ArrayList. De esta manera, si la implementación cambia en el futuro, el código que la utiliza seguirá funcionando.

Por otro lado, si la interfaz cambia, el código que depende de ella tiene que tiene que cambiar también. Por eso los desarrolladores de librerías evitan cambiar las interfaces a menos que sea que sea absolutamente necesario.

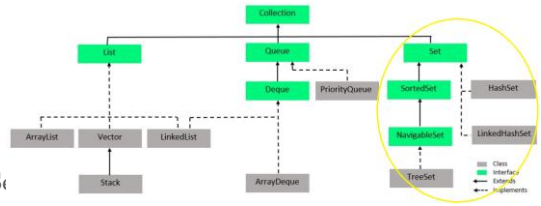
Interfaz Set

La estructura de datos **Conjunto** se basa en el concepto matemático de conjunto: **colección de elementos no duplicados.**

- Consideramos que un elemento de un conjunto está repetido si tenemos dos objetos o1 y o2 iguales, comparándolos mediante el operador **o1.equals(o2).**
- De esta forma, si el objeto a insertar en el conjunto estuviese **repetido**, **no nos dejará insertarlo**.
- Recordemos que el método **add** devolvía un valor booleano, que servirá para este caso, devolviéndonos true si el elemento a añadir no estaba en el conjunto y ha sido añadido, o false si el elemento ya se encontraba dentro del conjunto. Un conjunto podrá contener a lo sumo un elemento null.

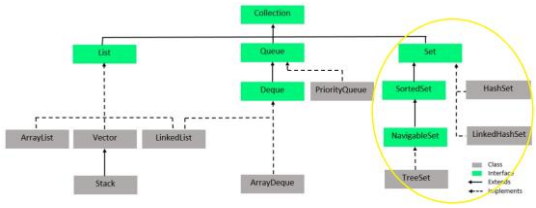
Interfaz Set

Los **conjuntos** se definen en la interfaz S diferentes implementaciones:



- **HashSet:** conjunto implementado con un una tabla hash (HashMap). Es rápido y no permite duplicados, pero no tiene ningún tipo de ordenación. Al imprimir los elementos puede dar distintos resultados. El rendimiento de HashSet es mejor en comparación con LinkedHashSet y TreeSet.
- **LinkedHashSet:** es un HashSet (con un LinkedList) que incluye ordenación de elementos por orden de inserción.
- **TreeSet:** conjunto implementado con un árbol (TreeMap). Los elementos están ordenados.

Interfaz Set



Crear un conjunto

```
Set<Integer> enteros = new HashSet();
```

Añadir elementos

```
enteros.add(4);
enteros.add(5);
enteros.add(7);
```

Añadir elemento repetido

```
boolean repetido = enteros.add(4);
```

no añade el 4 porque ya está en el conjunto y además devuelve 'false'

Actividad

Insertar elementos

Como puede ver allí, todos los valores duplicados no están presentes en la salida, incluido el valor nulo duplicado.

```
[null, Apple, Grapes, Fig, Mango, Orange]


-----------------
(program exited with code: 0)
```

```
import java.util.HashSet;
import java.util.Set;

public class HashSetExample {

    public static void main(String[] args) {

            //Crea un conjunto
            Set<String> frutas = new HashSet<>();

            //Agregar elementos
            frutas.add("Apple");
            frutas.add("Mango");
            frutas.add("Grapes");
            frutas.add("Orange");
            frutas.add("Fig");

            //Agregar elementos duplicados
            frutas.add("Apple");
            frutas.add("Mango");

            //Agregar valores nulos
            frutas.add(null);
            frutas.add(null);

            //Mostrar elementos
            System.out.println(frutas);

    }
}
```

Interfaz Set

```
Set<Integer> enteros = new HashSet<>();
enteros.add(4);
enteros.add(5);
enteros.add(7);
```

Tamaño de un conjunto

```
enteros.size();
```

me devuelve 3: el conjunto contiene 3 elementos.

Saber si un elemento está en el conjunto

```
enteros.contains(7); // me devolverá 'true'
enteros.contains(8); // me devolverá 'false'
```

Interfaz Set

Un Set no tiene los métodos **get()** e **indexOf()**, los elementos no están en ninguna posición en particular. Básicamente, con un conjunto lo que podemos hacer es añadir elementos, eliminarlos y preguntar si un elemento pertenece al conjunto.

Recorrer todos los elementos del conjunto

Con un bucle for-each como éste:

```
for (Integer e: enteros)
    System.out.println(e);
```

Actividad

Recorriendo un conjunto con un iterador

```java
import java.util.HashSet;
import java.util.Set;
import java.util.Iterator;

public class IterateHashSetExample {

    public static void main(String[] args) {

            //Crea un conjunto
            Set<String> frutas = new HashSet<>();

            //Agrega elementos
            frutas.add("Apple");
            frutas.add("Mango");
            frutas.add("Grapes");
            frutas.add("Orange");
            frutas.add("Fig");

            Iterator<String> it = frutas.iterator();
            while (it.hasNext())
                System.out.println(it.next());


    }
}
```

Interfaz Set

Eliminar un objeto del conjunto

```java
boolean eliminado = enteros.remove(7);
```

Devuelve 'true'. Si no hay ningún 7 devolvería 'false'.

Saber si un conjunto está vacío o vaciarlo

```java
boolean estaVacio = enteros.isEmpty(); // devolverá 'false'
enteros.clear(); // ahora sí que está vacío
```

Listas y conjuntos

Como puedes observar, hay operaciones sobre las listas y los conjuntos que son iguales. Y es que tanto las listas como los conjuntos son colecciones de objetos y por tanto comparten algunas operaciones. Esto es así porque ambas clases son a su vez del tipo Collection, de manera que podemos hacer lo siguiente:

```
List<Integer> listaEnteros= new ArrayList();
Set<Integer> conjuntoEnteros = new HashSet();

Collection<Integer> coleccion = listaEnteros;
coleccion.add(10);
coleccion = conjuntoEnteros;
coleccion.add(11);
```

Actividad

Convertir un HashSet a un ArrayList/List

Aquí tenemos HashSet de String y estamos creando una ArrayList de Strings copiando todos los elementos de HashSet a ArrayList.

El siguiente es el código completo:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.HashSet;

public class HashSetToArrayList {

    public static void main(String[] args) {

        //Declaración e inicialización de un array
        Set<String> conjuntoAmigos = new HashSet<>();

        //Agregar elementos
        conjuntoAmigos.add("Ana");
        conjuntoAmigos.add("Gabriel");
        conjuntoAmigos.add("Gema");
        conjuntoAmigos.add("Antonio");

        //Mostrar los elementos del HashSet
        System.out.println(conjuntoAmigos);

        //Conversión a List de un HashSet
        List<String>listaAmigos = new ArrayList(conjuntoAmigos);

        //Mostrar elementos del ArrayList
        System.out.println(listaAmigos);
    }
}
```

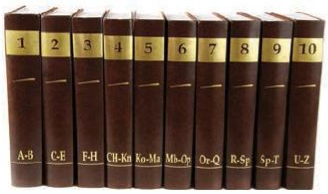


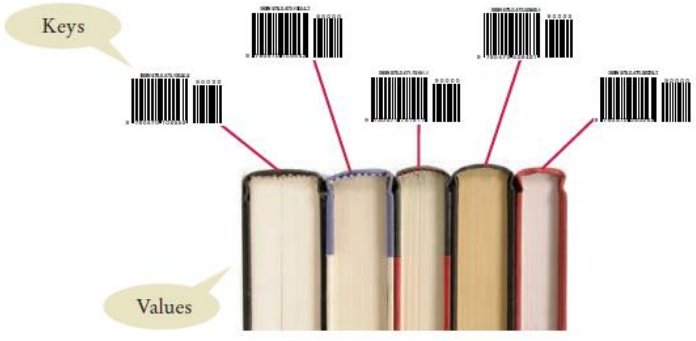
Figure 2 A List of Books

Figure 3 A Set of Books

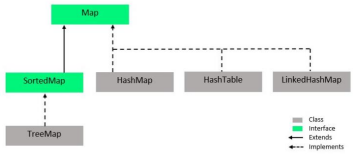
Figure 4 A Stack of Books

Mapas

Un mapa es una colección cuyos elementos son pares: **clave-valor .**



Mapas. La interfaz Map

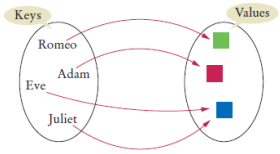


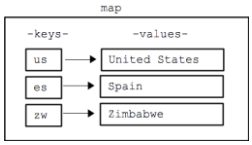
Aunque muchas veces se hable de los mapas como una colección, en realidad no lo son, ya que **no** heredan de la interfaz Collection.

Los **mapas se definen en la interfaz Map.** Un mapa es una colección cuyos elementos son pares: **clave-valor .**

Contendrá un conjunto de claves, y a cada clave se le asociará un determinado valor.

Tanto la clave como el valor puede ser cualquier objeto. Esta interfaz no puede contener claves duplicadas y; cada una de dichas claves, sólo puede tener asociado un valor como máximo.





Mapas. La interfaz Map

Los métodos básicos para trabajar con estos elementos son los siguientes:

`Object` **`get(Object clave)`**

Nos devuelve el valor asociado a la clave indicada

`Object` **`put(Object clave, Object valor)`**

Inserta una nueva clave con el valor especificado. Nos devuelve el valor que tenía antes dicha clave, o *null* si la clave no estaba en la tabla todavía.

`Object` **`remove(Object clave)`**

Elimina una clave, devolviendonos el valor que tenía dicha clave.

`Set` **`keySet()`**

Nos devuelve el conjunto de claves registradas

`int` **`size()`**

Nos devuelve el número de parejas (clave,valor) registradas.

Mapas. La interfaz Map

Encontramos distintas implementaciones de los mapas:

- **HashMap:** La implementación más genérica de Map. Un mapa que no garantiza el orden de las claves.
- **LinkedHashMap:** Implementación de map que garantiza orden de claves por su orden de inserción. Itera más rápido que un HashMap, pero inserta y elimina mucho peor.
- **TreeMap:** Un Map que permite que sus claves puedan ser ordenadas, de la misma forma que TreeSet. Úsalo en lugar de un HashMap solo si necesitas esta ordenación, ya que su rendimiento es mucho peor que el de HashMap en casi todas las operaciones (excepto iteración).
- **HashTable: deprecated**

Mapas. La interfaz Map

Crear un mapa y añadir elementos

```
Map <Integer, Punto> mapa = new HashMap<>();
mapa.put(1,new Punto(0,0));
mapa.put(2,new Punto(2,6));
mapa.put(3,new Punto(1,2));
```

Obtener un valor

Se pasa una clave como parámetro al método get

```
Punto punto = mapa.get(1);
```

Vaciar un mapa

```
mapa.clear();
```

Mapas. La interfaz Map

Verificar si contiene una clave

```
boolean tieneClave = mapa.containsKey(5);
```

Verificar si contiene un valor

```
Map <Integer, Punto> mapa = new HashMap<>();
Punto punto1 = new Punto(0,0);
Punto punto2 = new Punto(2,6);
Punto punto3 = new Punto(1,2);
mapa.put(1, punto1);
mapa.put(2, punto2);
mapa.put(3, punto3);

boolean tieneValor = mapa.containsValue(punto3);
System.out.println(tieneValor);
tieneValor = mapa.containsValue(new Punto(1,2));
System.out.println(tieneValor);
```

La primera llamada a containsValue devuelve true y la segunda false.

Mapas. La interfaz Map

Insertar todos los elementos de otro mapa

```
Map <Integer, Punto> mapaA = new HashMap<>();
mapaA.put(1,new Punto(0,0));
mapaA.put(2,new Punto(2,6));
mapaA.put(3,new Punto(1,2));

Map <Integer, Punto> mapaB = new HashMap<>();
mapaB.putAll(mapaA);
```

Mapas. La interfaz Map

Recorriendo las claves

```java
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;
public class IteratingKeysMap {

    public static void main(String[] args) {

        Map <Integer, Punto> mapa = new HashMap<>();

        Punto punto1 = new Punto(0,0);
        Punto punto2 = new Punto(2,6);
        Punto punto3 = new Punto(1,2);

        mapa.put(1, punto1);
        mapa.put(2, punto2);
        mapa.put(3, punto3);

        //Mostrar un mapa
        System.out.println(mapa);

        /*RECORRIENDO CLAVES*/

        //Con un iterador
        Iterator iterator = mapa.keySet().iterator();
        while(iterator.hasNext()){
          Integer clave  = (Integer)iterator.next();
          Punto punto = mapa.get(clave);
          System.out.println(clave + " - " + punto);
        }

        //Bucle for-each
        for(Integer clave : mapa.keySet()) {
            Punto punto = mapa.get(clave);
            System.out.println(clave + " - " + punto);
        }

        //método forEach
        mapa.forEach((clave, punto) -> System.out.println(clave + " - " + punto));

    }
}
```

Mapas. La interfaz Map

Recorriendo los valores

```java
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;
public class IteratingValuesMap {

    public static void main(String[] args) {

        Map <Integer, Punto> mapa = new HashMap<>();

        Punto punto1 = new Punto(0,0);
        Punto punto2 = new Punto(2,6);
        Punto punto3 = new Punto(1,2);

        mapa.put(1, punto1);
        mapa.put(2, punto2);
        mapa.put(3, punto3);

        /*RECORRIENDO VALORES*/

        //Con un iterador
        Iterator<Punto> iterator = mapa.values().iterator();
        while(iterator.hasNext()) {
            Punto punto = (Punto) iterator.next();
            System.out.println(punto);
        }

        //Bucle for-each
        for(Punto punto : mapa.values()){
            System.out.println(punto);
        }
    }
}
```

```
Clave: 16 -> Valor: Busquets
Clave: 1 -> Valor: Casillas
Clave: 18 -> Valor: Pedrito
Clave: 3 -> Valor: Pique
Clave: 5 -> Valor: Puyol
Clave: 6 -> Valor: Iniesta
Clave: 7 -> Valor: Villa
Clave: 8 -> Valor: Xavi Hernandez
Clave: 11 -> Valor: Capdevila
Clave: 14 -> Valor: Xabi Alonso
Clave: 15 -> Valor: Ramos
```

HashMap

```
Clave: 1 -> Valor: Casillas
Clave: 3 -> Valor: Pique
Clave: 5 -> Valor: Puyol
Clave: 6 -> Valor: Iniesta
Clave: 7 -> Valor: Villa
Clave: 8 -> Valor: Xavi Hernandez
Clave: 11 -> Valor: Capdevila
Clave: 14 -> Valor: Xabi Alonso
Clave: 15 -> Valor: Ramos
Clave: 16 -> Valor: Busquets
Clave: 18 -> Valor: Pedrito
```

TreeMap

```
Clave: 1 -> Valor: Casillas
Clave: 15 -> Valor: Ramos
Clave: 3 -> Valor: Pique
Clave: 5 -> Valor: Puyol
Clave: 11 -> Valor: Capdevila
Clave: 14 -> Valor: Xabi Alonso
Clave: 16 -> Valor: Busquets
Clave: 8 -> Valor: Xavi Hernandez
Clave: 18 -> Valor: Pedrito
Clave: 6 -> Valor: Iniesta
Clave: 7 -> Valor: Villa
```

LinkedHashMap

```
public class PruebaHashMap {
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "Casillas");
        map.put(15, "Ramos");
        map.put(3, "Pique");
        map.put(5, "Puyol");
        map.put(11, "Capdevila");
        map.put(14, "Xabi Alonso");
        map.put(16, "Busquets");
        map.put(8, "Xavi Hernandez");
        map.put(18, "Pedrito");
        map.put(6, "Iniesta");
        map.put(7, "Villa");

        // Imprimimos el Map con un Iterador
        Iterator it = map.keySet().iterator();
        while(it.hasNext()){
          Integer key = (Integer) it.next();
          System.out.println("Clave: " + key +
                             " -> Valor: " + map.get(key));
        }
    }
}
```

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

        Map<Integer, String> treeMap = new TreeMap<>();
        treeMap.put(1, "Casillas");
        treeMap.put(15, "Ramos");
        treeMap.put(3, "Pique");
        treeMap.put(5, "Puyol");
        treeMap.put(11, "Capdevila");
        treeMap.put(14, "Xabi Alonso");
        treeMap.put(16, "Busquets");
        treeMap.put(8, "Xavi Hernandez");
        treeMap.put(18, "Pedrito");
        treeMap.put(6, "Iniesta");
        treeMap.put(7, "Villa");

        // Imprimimos el Map con un Iterador
        Iterator it = treeMap.keySet().iterator();
        while(it.hasNext()){
          Integer key = (Integer)it.next();
          System.out.println("Clave: " + key +
                             " -> Valor: " + treeMap.get(key));
        }
    }
}
```

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

        Map<Integer, String> linkedHashMap = new LinkedHashMap<>();
        linkedHashMap.put(1, "Casillas");
        linkedHashMap.put(15, "Ramos");
        linkedHashMap.put(3, "Pique");
        linkedHashMap.put(5, "Puyol");
        linkedHashMap.put(11, "Capdevila");
        linkedHashMap.put(14, "Xabi Alonso");
        linkedHashMap.put(16, "Busquets");
        linkedHashMap.put(8, "Xavi Hernandez");
        linkedHashMap.put(18, "Pedrito");
        linkedHashMap.put(6, "Iniesta");
        linkedHashMap.put(7, "Villa");

        // Imprimimos el Map con un Iterador
        Iterator it = linkedHashMap.keySet().iterator();
        while(it.hasNext()){
          Integer key = (Integer)it.next();
          System.out.println("Clave: " + key +
                             " -> Valor: " + linkedHashMap.get(key));
        }
    }
}
```

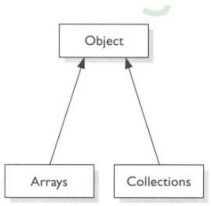
- HashMap: vemos que nos da los objetos sin un "orden lógico“
- TreeMap : vemos que nos ordena los objetos por clave en "orden natural“
- LinkedHashMap: vemos que nos ordena los objetos tal y como los hemos ido introduciendo:

La clase Collections y la clase Arrays

La clase **Collections** (no confundir con la interfaz **Collection**) dispone exclusivamente de métodos estáticos que operan sobre colecciones.

Contiene métodos que operan sobre colecciones, los cuales devuelven una nueva colección obtenida a partir la colección original.

La clase Collections y la clase Arrays




```
// Utility Class java.util.Collections
static <T> int binarySearch(List<? extends T> lst, T key, Comparator<? super T> comp)
static <T> int binarySearch(List<? extends Comparable<? super T>> lst, T key)
   // Searches for the specified key using binary search

static <T> void sort(List<T> lst, Comparator<? super T> comp)
static <T extends Comparable<? super T>> void sort(List<T> lst)

static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp)
static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp)
static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)
```

```
// Utility Class java.util.Collections
static void swap(List<?> lst, int i, int j)   // Swaps the elements at the specified indexes
static void shuffle(List<?> lst)              // Randomly permutes the List
static void shuffle(List<?> lst, Ramdon rnd)  // Randomly permutes the List using the specified source or randomness
static void rotate(List<?> lst, int distnace) // Rotates the elements by the specified distance
static void reverse(List<?> lst)              // Reverses the order of elements
static <T> void fill(List<? super T>, T obj)  // Replaces all elements with the specified object
static <T> void copy(List<? super T> dest, List<? extends T> src)  // Copies all elements from src to dest
static <T> boolean replaceAll(List<T> lst, T oldVal, T newVal)     // Replaces all occurrences
```

Actividad

Ordenar una lista

```
[India, US, China, Denmark]
[China, Denmark, India, US]
```

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Collections;

public class ArrayListSort  {

    public static void main(String args[]){

        List<String> listaPaises = new ArrayList<>();

        listaPaises.add("India");
        listaPaises.add("US");
        listaPaises.add("China");
        listaPaises.add("Denmark");

        /* Lista no ordenada */
        System.out.println(listaPaises);

        /* Ordena la lista */
        Collections.sort(listaPaises);

        /* Lista ordenada */
        System.out.println(listaPaises);

    }
}
```

Actividad

```java
public class StringPrimitiveComparableTest {
   public static void main(String[] args) {
      // Sort/search an "array" of Strings using Arrays.sort() and Arrays.binarySearch()
      //   using ordering specified in compareTo()
      String[] strArray = {"Hello", "hello", "Hi", "HI", "Hello"};  // has duplicate elements

      Arrays.sort(strArray);  // sort in place and mutable
      System.out.println(Arrays.toString(strArray));  //[HI, Hello, Hello, Hi, hello]

      // The array must be sorted for binarySearch()
      System.out.println(Arrays.binarySearch(strArray, "Hello")); //2
      System.out.println(Arrays.binarySearch(strArray, "HELLO")); //-1 (insertion at index 0)

      // Sort/search a List<Integer> using Collections.sort() and Collections.binarySearch()
      List<Integer> lst = new ArrayList<>();
      lst.add(22);  // int auto-box to Integer
      lst.add(11);
      lst.add(44);
      lst.add(11);  // duplicate element
      Collections.sort(lst);    // sort in place and mutable
      System.out.println(lst);  //[11, 11, 22, 44]
      System.out.println(Collections.binarySearch(lst, 22)); //2
      System.out.println(Collections.binarySearch(lst, 35)); //-4 (insertion at index 3)
   }
}
```

Actividad

Desordenar una lista. ¿Reconocéis este código?

```java
static private final String[] palabras_secretas = {"reingenieria", "cubeta","tunelizacion", "protocolo",
"puertos", "conexion", "broadcasting", "direccion","internet", "router", "switch", "wifi", "estandar",
"socket", "transporte","enlace", "capas", "arquitectura", "cliente", "servidor", "proxy", "firewall","redes",
"LAN", "WAN", "MAN", "hub", "concentrador", "datagrama", "puente","fibra", "TCP", "UDP", "mascara", "gateway",
"servidor", "DNS", "cliente","conmutacion", "circuito", "satelite", "coaxial", "microondas", "seÃ±al","ingrarrojos",
"token", "anillo", "bus", "control", "flujo", "congestion","enrutamiento", "aplicacion", "correo", "peertopeer"};

public static String damePalabraAleatoria(){
    List<String> palabras = Arrays.asList(palabras_secretas);
    Collections.shuffle(palabras);
    return palabras.get(0);
}
```

Actividad

Matriz con números aleatorios sin repetir

```java
public static void generaMatrizConArrayListConsecutivos(int[][] matriz){

    //Ejemplo base usando el algoritmo de Fisher-Yates:
    List<Integer> lista= new ArrayList();
    for (int j = 1; j <= 100; j++ )
        lista.add(j);

    Collections.shuffle(lista);
    System.out.println(lista);

    int i=0;
    for (int f=0; f<matriz.length;f++){
        for (int c=0; c<matriz[0].length;c++){
                matriz[f][c]=lista.get(i);
                i++;
                System.out.print(matriz[f][c] + " ");
        }
        System.out.println();
    }

}
```

Operador diamante <>

Antes de Java 1.5, en la API de Collections no había forma de parametrizar los argumentos construir una colección:

```java
List cars = new ArrayList();
cars.add(new Object());
cars.add("car");
cars.add(new Integer(1));
```

Esto permitía añadir cualquier tipo y conducía a posibles excepciones de conversión de tipos en tiempo de ejecución.

Operador diamante <>

En Java 1.5 se introdujeron los genéricos, que permitían parametrizar las clases, incluidas las de la API de Colecciones, al declarar y construir objetos:

```
List<String> cars = new ArrayList<String>();
```

En este punto, tenemos que especificar el tipo parametrizado en el constructor, que puede ser algo ilegible:

```
Map<String, List<Map<String, Map<String, Integer>>>> cars
 = new HashMap<String, List<Map<String, Map<String, Integer>>>>();
```

Operador diamante <>

La versión 1.7 de Java soporta la inferencia de tipos en la instanciación para simplificar el código:

```
List<String> cars = new ArrayList<>();
```

```
// Prior to JDK 7, you need to write
List<String> coffeeLst = new ArrayList<String>();

// JDK 7 can infer on type from the type of the assigned variable
List<String> coffeeLst = new ArrayList<>();
```

El método equals

Durante varios capítulos has utilizado el operador == para comparar **igualdades**.

Has visto que este operador no se comporta como se espera cuando se usa con objetos, ya que es posible tener dos objetos distintos con estados equivalentes, como dos objetos Punto con las coordenadas (7, 2).

```
Point p1 = new Point(7, 2);
Point p2 = new Point(7, 2);
```

Esta observación nos recuerda que un objeto tiene una identidad y es distinto de otros objetos, aunque otro objeto tenga el mismo estado.

El método equals

El operador == no se comporta como se espera con objetos, porque comprueba si dos objetos tienen la misma identidad. En realidad, la comparación == comprueba si dos variables se refieren al mismo objeto, no si dos objetos distintos tienen el mismo estado.

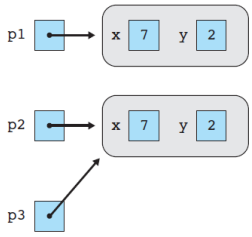
Considere las siguientes tres declaraciones de variables:

```
Point p1 = new Point(7, 2);
Point p2 = new Point(7, 2);
Point p3 = p2;
```

El método equals

El siguiente diagrama representa el estado de estos objetos y las variables que hacen referencia a ellos. Observa que p3 no es una referencia a un tercer objeto, sino una segunda referencia al mismo objeto al que hacía referencia p2.

```
Point p1 = new Point(7, 2);
Point p2 = new Point(7, 2);
Point p3 = p2;
```



El objeto al que se refiere p1 tiene el mismo estado que el objeto de p2, pero tienen **identidades diferentes.** La expresión p2 == p3 se evaluaría como verdadera, porque p2 se refiere al mismo objeto que p3.

El método equals

A menudo, al comparar dos objetos, queremos saber si tienen el mismo estado. Para realizar una comparación de este tipo, utilizamos un método especial llamado **equals**. Cada objeto Java contiene un método equals que utiliza para compararse a sí mismo con otros objetos.

Un **método equals adecuado realiza una comparación de los estados de dos objetos y devuelve true si los estados son iguales.**

El método equals

Con los objetos Point anteriores, nos gustaría que las expresiones p1.equals(p2), p1.equals(p3), y p2.equals(p3) se evalúen a true porque todos los puntos tienen las mismas coordenadas (x, y).

Puedes imaginarte un código que examine dos objetos Punto para ver si tienen los mismos valores en los campos x e y:

```
if (p1.getX() == p2.getX() && p1.getY() == p2.getY()) {
    // the objects have equal state
    ...
}
```

Pero la funcionalidad de igualación **debería implementarse en la propia clase Punto y no en el código que usa los objetos Punto.**

El método equals

El siguiente código es una implementación inicial del método equals que tiene varios **defectos**:

```
// a flawed implementation of an equals method
public boolean equals(Point p2) {
    if (x == p2.getX() && y == p2.getY()) {
        return true;
    } else {
        return false;
    }
}
```

El método equals

```
// a flawed implementation of an equals method
public boolean equals(Point p2) {
    if (x == p2.getX() && y == p2.getY()) {
        return true;
    } else {
        return false;
    }
}
```

Un defecto inicial que podemos corregir es que el código anterior no hace un buen uso de un booleano. Recuerde que cuando su código usa una sentencia if/else para devolver un valor booleano verdadero o falso, a menudo puede devolver directamente el valor de la condición de la sentencia if:

```
return x == p2.getX() && y == p2.getY();
```

Es legal que el método equals acceda directamente a los campos de p2, así que opcionalmente podemos modificar esto aún más. Los campos privados son visibles para toda la clase, incluyendo otros objetos de la misma clase. Por lo que es legal que un objeto Point examine los campos de otro:

```
return x == p2.x && y == p2.y;
```

El método equals

Para mantener nuestro método equals coherente con otras clases Java, también debemos hacer un cambio en su cabecera. **El parámetro del método equals no debe ser de tipo Point**. En su lugar, el método debe aceptar un parámetro de tipo Object:

```
public boolean equals(Object o)
```

Una variable o parámetro de tipo Object puede referirse a cualquier objeto Java, lo que significa que cualquier objeto puede ser pasado como parámetro al método equals. Así, podemos comparar objetos Punto con cualquier tipo de objeto, no sólo con otros Puntos.

Por ejemplo una expresión como **p1.equals("hola")** sería ahora legal. El método equals debería devolver false en este caso porque el parámetro no es un Punto.

```
public boolean equals(Object o)
```

El método equals

Se podría pensar que el código siguiente compararía correctamente los dos objetos Point y devolvería el resultado correcto.
Desafortunadamente, ni siquiera compila correctamente.

```
return x == o.x && y == o.y; // does not compile
```

El método equals

Si queremos tratar a o como un objeto Point, debemos convertirlo de tipo Object a tipo Punto.

Ya hemos hablado de la conversión entre tipos primitivos, como por ejemplo dedoble a int.

La conversión entre tipos objeto tiene un significado diferente. **Un cast de un objeto es una promesa al compilador**. El cast es la garantía de que la referencia se refiere realmente a un tipo diferente y que **el compilador puede tratarlo como ese tipo.**

El método equals

En nuestro método, escribiremos una sentencia que transforme el parámetro o en un objeto Point para que el compilador confíe en que podemos acceder a sus campos x e y:

```
// returns whether the two Points have the same (x, y) values
public boolean equals(Object o) {
    Point other = (Point) o;
    return x == other.x && y == other.y;
}
```

No olvides que si tu objeto tiene atributos que son objetos en sí mismos, como un atributo String o Point, entonces esos atributos deben ser comparados por igualdad usando su método equals y no usando el operador ==.

NO ENTRAN PILAS Y COLAS EN EL EXAMEN

Tipos de datos abstractos: pilas y colas

Cabe abrir un paréntesis e incorporar este apartado para introducir el concepto de tipo abstracto de datos (TAD) y aclarar la diferencia existente con el concepto de estructura de datos ya que a menudo se utilizan como sinónimos y no lo son.

Tipos de datos abstractos: pilas y colas

El **tipo de datos abstracto** puede definirse como una colección de datos y operaciones asociadas que se especifican con precisión independientemente de cualquier implementación en particular.

Una **estructura de datos** es una representación (implementación) de un TAD en un lenguaje de programación.

Un TAD puede tener varias implementaciones. Por ejemplo, una pila se puede implementar con un vector o una lista.

Tipo de dato abstracto: Pila

Imaginemos un puerto marítimo de mercancías donde los contenedores se colocan uno encima del otro con una grúa para organizar el espacio; se puede poner un contenedor nuevo en la parte superior, así como quitar uno que esté en lo alto.

Tipo de dato abstracto: Pila

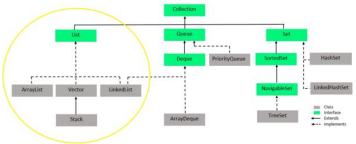
Para mover el contenedor que está en la parte inferior, primero hay que mover los contenedores que tenga por encima. La estructura de datos clásica de **Stack (pila)** funciona de manera que se pueden añadir elementos, uno a uno, en la parte superior y permite que se retire el último elemento que ha sido añadido, de uno en uno, pero no los anteriores (los que están debajo de él).

Tipo de dato abstracto: Pila

Hay muchas aplicaciones para las pilas en la informática. Considere la función de deshacer de un procesador de textos. Guarda los comandos emitidos en una pila. Cuando se selecciona "Deshacer", se deshace el último comando, luego el penúltimo, y así sucesivamente.

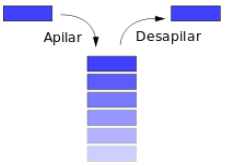
The Undo key pops commands off a stack so that the last command is the first to be undone.

Pilas y la clase Stack



En programación y sistemas operativos, las **pilas** son estructuras de uso común. Se utilizan internamente para gestionar muchas características. La pila es una estructura de datos, que implementa el comportamiento **LIFO (Last In - First Out)** (último en entrar - primero en salir). Como con los contenedores de mercancías, se podrían añadir elementos y quitados sólo en la parte superior de la pila.

Una pila se caracteriza por tres operaciones fundamentales:



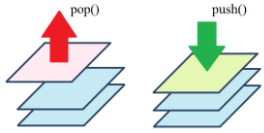
- `push()` - Añade un elemento en la parte superior de la pila.
- `pop()` - Extrae el elemento existente en la parte superior de la pila.
- `peek()` - Lee el elemento de la parte superior de la pila sin eliminarlo.

Una pila pueden tener implementaciones dinámicas y estáticas. La clase **Stack** es una implementación dinámica de una pila

Actividad

Utilizando pilas

```
stack: []
stack: [10, 15, 80]
80
15
10
```




```java
import java.util.Stack;

public class EjemploPila {

   public static void main(String args[]) {

      Stack<Integer> pila = new Stack<>();
      System.out.println("stack: " + pila);

      pila.push(10);
      pila.push(15);
      pila.push(80);

      System.out.println("stack: " + pila);

      while (!pila.isEmpty())
            System.out.println(pila.pop());

   }
}
```

Tipo de dato abstracto: Cola

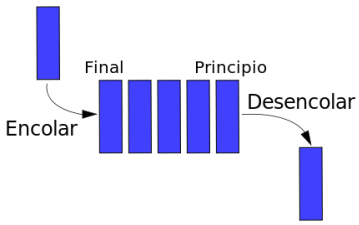
Las **colas** son estructuras lineales adecuadas para tareas y tratamientos de naturaleza secuencial en el que el orden y la prioridad son importantes.

Por ejemplo una cola de espera para la impresión de documentos, los procesos de espera para acceder a un recurso común, y otros.

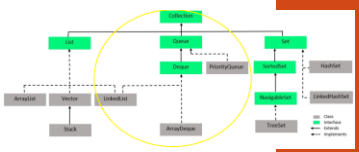
Tipo de dato abstracto: Cola

En las colas, de manera normal se **añaden elementos en la parte del final y recuperan o extraen de la cabeza o frente.**

Por ejemplo, para comprar una entrada para un concierto; probablemente haya que hacer cola delante de la taquilla y habrá que esperar a que sean atendidos todos los clientes que tengamos delante.



Colas y la interfaz Queue



Una cola cumple el comportamiento **FIFO (first in - first out)** (primero en entrar - primero en salir). Los elementos añadidos a una cola, lo hacen por final, y cuando se extraen los elementos, se hace por el principio de la cola (en el orden en que fueron añadidos).

Así, la cola se comporta como una lista con dos extremos (cabeza y final). Una cola pueden tener implementaciones dinámicas y estáticas. La interfaz **Queue** es una implementación dinámica de una cola.

La interfaz Queue

La implementación en Java de **Queue** tiene, entre otros, los siguientes métodos básicos son:

`Object peek()`	- Obtiene el elemento de la cabeza de la cola sin quitarlo.
`Object poll()`	- Extrae el elemento situado en la cabeza de la cola.
`void offer(Object)`	- Añade un elemento al final de la cola.

Actividad

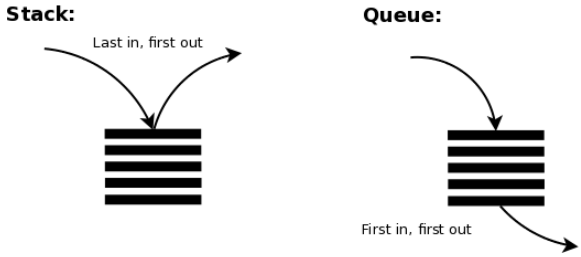
Utilizando colas

```
queue: []
queue: [One, Two, Three, Four, Five]
One
Two
Three
Four
Five
```

```java
import java.util.LinkedList;
import java.util.Queue;
import java.util.Iterator;

public class EjemploCola {

    public static void main(String[] args) {

        Queue<String> cola = new LinkedList<>();

        System.out.println("queue: " + cola);

        cola.offer("One");
        cola.offer("Two");
        cola.offer("Three");
        cola.offer("Four");
        cola.offer("Five");

       System.out.println("queue: " + cola);

        while (!cola.isEmpty())
            System.out.println(cola.poll());
    }
}
```

Actividad Pilas y Colas

Crea una aplicación Java de nombre **Palindromo** que permita introducir, por teclado, una frase. A partir de esa frase, crea una **pila** y una **cola** de letras (utilizar las clases **LinkedList** para la cola y **Stack** para la pila) para determinar si la frase es palíndromo.




```java
import java.util.LinkedList;

public class Palindrome {

    public static void main(String[] args) {

            Scanner sc = new Scanner(System.in);

            String cadena = sc.nextLine();

            Queue<Character> cola = new LinkedList<>();
            Stack<Character> pila = new Stack<>();

            int longitud = cadena.length();

            //Llenamos la pila y la cola con los valores de la frase
            for(int i =0; i< longitud; i++){
                cola.offer(cadena.charAt(i));
                pila.push(cadena.charAt(i));
            }

            System.out.println("Pila: " + pila);
            System.out.println("Cola: " + cola);

            //ahora comparamos la pila y la cola para decidir si la
            //frase es palíndrome
            boolean palindrome = true;
            while(!cola.isEmpty()){
                if(cola.poll() != pila.pop()){
                    palindrome = false; // con un solo caracter que sea
                                        // distinto ya la cadena no es palíndrome
                }
            }

            if(palindrome)
               System.out.println("La cadena es palíndrome");
            else
               System.out.println("la cadena NO es palíndrome");
        }
}
```

Recursos

- Curso youtube : Java desde 0
- Libro Java 9. Manual imprescindible. F. Javier Moldes Teo. Editorial Anaya
- App SoloLearn: Aprende a Programar. Curso Java
- Tutorial de Java Collections