En los lenguajes de programación suele haber dos tipos de datos almacenados que son:

* Estáticos. Son los arrays que ya has manejado, debemos saber de antemano el número de elementos con los que deseamos trabajar.
* Dinámicos. No sabemos cuántos elementos vamos a tener y según necesitamos elementos los vamos incorporando. Son los que están agrupados en las colecciones o Collection. Que son las listas, pilas, colas, árboles, grafos, etc…

Los datos que forman las colecciones de datos dinámicos son siempre objetos y cada uno de esos objetos recibe el nombre de nodo. Esos objetos tienen un campo llamado campo de enlace, cuyo tipo es el nombre de la clase donde está y se utiliza para enlazar con otro elemento. Estas estructuras pueden ser:

1. Lineales. Listas, pilas y colas.
2. No lineales. Árboles y grafos.

Tenemos una serie de clases e interfaces para manipular estos tipos de estructuras. Que son Collection, Map y Iterator.

Diagrama, Esquemático

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

INTERFAZ COLLECTION

Se encuentra en **java.io.Collection**. Solo se pueden recorrer los datos en el mismo orden en que fueron insertados y debemos pasar por todos los anteriores hasta el elemento deseado. Podemos eliminar cualquier elemento de la colección. Y los elementos nuevos van al final de la colección.

Podemos tener las interfaces List y Set.

1. Interfaz List. Es una colección de nodos que están en cierto orden, guarda la información de cómo están colocados cosa que no hace Collection, permite duplicados y permite el acceso a los datos a partir de una posición. Esta implementada por dos clases: ArrayList y LinkedList.

* ArrayList, no tiene un tamaño fijo de nodos o elementos. Indicamos su posición y accedemos al elemento directamente. Se pierde tiempo al borrar un elemento ya que tiene que recolocar todos los elementos que van detrás del elemento borrado al igual que insertamos un elemento.

**Ejemplos:**

**ArrayListEjemplo.java**

**ArrayListEjemplo\_String.java**

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

* Stack libro de Mario mirarlo
* LinkeList, son listas doblemente enlazadas, cada nodo o elemento tiene dos campos de enlace para saber quién le precede y quien le antecede. Se insertan o borran elementos de forma más fácil, al solo cambiar las referencias de los enlaces.

Los elementos de la colección no se guardan por el contenido sino por el identificador que le asigna a cada objeto, que es único para cada objeto.

Algunos de los métodos de Collection son:

* boolean add( Object ), añade un objeto a la colección, devuelve true si la operación ha sido correcta.
* boolean remove( Object ), elimina un objeto de la colección que el identificador coincida con el objeto que recibe.
* boolean contains( Object ), comprueba si hay un objeto con ese identificador.
* Object next( ), devuelve el siguiente objeto. Si no tenemos más dará el error NoSuchElementException.
* boolean hasNext( ), devuelve true o false si hay un elemento después del objeo. Pero no se mueve de posición.
* void remove( ), elimina el objeto donde apunta el iterador.

Ejemplo:

public static void main(String [] args)

{

//Collection es un interfaz y no puede instanciarse

Collection elemento = new ArrayList();

elemento.add(25);

elemento.add(54);

elemento.add(23);

int valor;

Iterator nodo = elemento.iterator();

while( nodo.hasNext())

{

valor = (Integer)elemento.next( );

System.out.println(valor+” “);

}

}

La interfaz List añade nuevos métodos para trabajar como pueden ser añadir un elemento entre dos nodos, mostrar un elemento indicando su posición (se enumeran de 0 en adelante) que son:

* boolean add( int, Object ), inserta un elemento en la lista en la posición indicada.
* Object remove(int), elimina el elemento que está en la posición indicada.
* Object set(int, Object), sustituye un elemento en la lista. Devuelve el objeto antiguo.
* Object get(int), devuelve el elemento que se encuentra en la posición indicada.

Si intentamos acceder a una posición que no existe se produce la excepción IndexOutOfBoundsException.

Para recorrer los elementos de una lista podemos utilizar el iterador Iterator o la interfaz ListIterator, que nos da mayores posibilidades, al poder modificar, añadir objetos y recorrer la lista en ambos sentidos.

Sus métodos son:

* void add(Object), inserta un elemento.
* void set(Object), sustituye un objeto.
* int nextIndex( ), devuelve la posición del siguiente elemento.
* int previousIndex( ), devuelve la posición del objeto al que apunta.
* Object previous( ), devuelve el objeto al que apunta.

1. Interfaz Set. Se usa cuando queremos que en la colección no tengamos elementos repetidos en la colección. Esta implementada por dos clases: HashSet y TreeSet.

* HashSet, es la clase más utilizada para implementar una colección sin duplicados, pero no se garantizan su orden. Utiliza una tabla, guardando los diferentes códigos de identificación de la colección. Cada posición de la tabla apunta a los diferentes nodos que tienen el mismo código de identificación. Debemos sobreescribir los métodos equals (decidimos cuando son iguales dos objetos) y hashCode (determinamos cual es el código de identificación de cada objeto). No admite acceso secuencial.

**HashSet nombre = new HashSet( );**

Ejemplo: **HashSet <String> palabra = new HashSet<String>();**

**Ejemplos:**

**HastSetEjemplos.java**

**HastSetEjemplos\_String.java**

https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/HashSet.html

* LinkedHashSet, no guardar duplicados, pero si mantiene el orden de los valores introducidos.

**LinkedHashSet nombre = new LinkedHashSet();**

Ejemplo: **LinkedHashSet <Integer> numeros = new LinkedHashSet<>();**

**Ejemplos:**

**LinkedHastSetEjemplos.java**

**LinkedHastSetEjemplos\_String.java**

https://docs.oracle.com/javase/10/docs/api/java/util/LinkedHashSet.html

* TreeSet, implementa la interfaz SortedSet, los elementos se encuentran ordenados en estructura en árbol binario. Con este tipo debemos implementar Comparable a la clase.

**TreeSet nombre = new TreeSet( );**

**Ejemplos:**

**TreeSetEjemplos.java**

**TreeSetEjemplos\_String.java**

Si los tipos de datos tienen una ordenación diferente a la esperada debemos utilizar lo siguiente:

**public class nombre implements Comparable <nombre clase>**

Comparable es un interfaz y tenemos que recodificar el método public int compareTo(Objeto). Si devuelve positivo el objeto que le llama va después que el objeto que recibe, 0 si son iguales y negativo el objeto que le llama va antes que el objeto que recibe.

https://docs.oracle.com/javase/10/docs/api/java/util/TreeSet.html

INTERFAZ MAP

Los mapas guardan parejas de objetos, asociando un objeto por su clave con otro objeto por un valor concreto. Las claves no pueden estar repetidas y solo se pueden asociar con un valor. Tenemos las clases TreeMap ( hace que los elementos estén ordenados por clave ) y HashMap ( hace algo similar a HashMap ).

INTERFAZ ITERATOR

Es para poder recorrer la los elementos declarados con el interfaz Collection, y por ello utilizamos el objeto Iterator. La Collection tiene el método iterator( ), que devuelve un objeto del tipo Iterator, con ese objeto y sus métodos podemos recorrer el objeto. Tenemos la interfaz ListIterator para recorrer listas ArraList y LinkedList. No se puede utilizar para recorrer otro tipo de colecciones.

En las colecciones podemos guardar cualquier tipo de objeto o dato. Si creamos clases o métodos genéricos solo podremos guardar el objeto indicado y si intentamos guardar otra cosa daría error.

ArrayList <clase> nombre = new ArrayList<clase> ( );

Al tener genéricos, podemos recorrer una colección con un for del siguiente tipo:

for( clase objeto: colección)

También podemos tener el iterador genérico.

Iterator <clase> objeto = colección.iterator( );

**REPASAR CON LO ANTERIOR ESCRITO Y REALIZAR EJERCICIOS.**

Es una interfaz que define los siguientes métodos para trabajar con listas:

* **boolean add(Object o)**. Añade el objeto a la colección. Devuelve true si se pudo completar la operación. Si no cambió la colección como resultado de la operación devuelve false
* **boolean remove(Object o)**. Elimina al objeto indicado de la colección.
* **int size( )**. Devuelve el número de objetos almacenados en la colección.
* **boolean isEmpty( )**. Indica si la colección está vacía.
* **boolean contains(Object o)**. Devuelve true si la colección contiene a o.
* **void clear( )**. Elimina todos los elementos de la colección.
* **boolean addAll( Collection otra)**. Añade todos los elementos de la colección otra a la colección actual.
* **boolean removeAll( Collection otra)**. Elimina todos los objetos de la colección actual que estén en la colección otra.
* **boolean retainAll(Collection otra)**. Elimina todos los elementos de la colección que no estén en la otra.
* **boolean containsAll(Collection otra)**. Indica si la colección contiene todos los elementos de otra
* **Object[] toArray(** **)**. Convierte la colección en un array de objetos.
* **Iterator iterator(** **)**. Obtiene el objeto iterador de la colección.

La interfaz Iterator (también en java.util) define objetos que permiten recorrer los elementos de una colección. Los métodos definidos por esta interfaz son:

* **Object next( )**. Obtiene el siguiente objeto de la colección. Si se ha llegado al final de la colección y se intenta seguir, da lugar a una excepción de tipo: **NoSuchElementException** (que deriva a su vez de **RunTimeException**)
* **boolean hasNext( )**. Indica si hay un elemento siguiente (y así evita la excepción).
* **void remove( )**. Elimina el último elemento devuelto por **next**.

LISTAS ENLAZADAS

**List** es una interfaz de **java.util**. Es una colección en la que si importa el orden de los elementos. Hereda los métodos de Collections pero los propios de su interfaz son:

* **void add(int índice, Object elemento)** Añade el elemento indicado en la posición índice de la lista.
* **void remove(int índice)** Elimina el elemento cuya posición en la colección la da el parámetro índice.
* **Object set(int índice, Object elemento)** Sustituye el elemento número índice por uno nuevo. Devuelve además el elemento antiguo.
* **int indexOf(Object elemento)** Devuelve la posición del elemento. Si no lo encuentra, devuelve -1.
* **int lastIndexOf(Object elemento)** Devuelve la posición del elemento comenzando a buscarle por el final. Si no lo encuentra, devuelve -1.
* **void addAll(int índice, Collection elemento)** Añade todos los elementos de una colección a una posición dada.
* **ListIterator listIterator( )** Obtiene el iterador de lista que permite recorrer los elementos de la lista.
* **ListIterator listIterator(int índice)** Obtiene el iterador de lista que permite recorrer los elementos de la lista. El iterador se coloca inicialmente apuntando al elemento cuyo índice en la colección es el indicado.

ListIterator

Es un interfaz que define clases de objetos para recorrer listas. Es heredera de la interfaz

**Iterator.** Aporta los siguientes métodos

**método uso**

* **void add(Object elemento)** Añade el elemento delante de la posición actual del iterador
* **void set(Object elemento)** Sustituye el elemento señalado por el iterador, por el elemento indicado

https://www.arquitecturajava.com/java-foreach-y-sus-opciones/