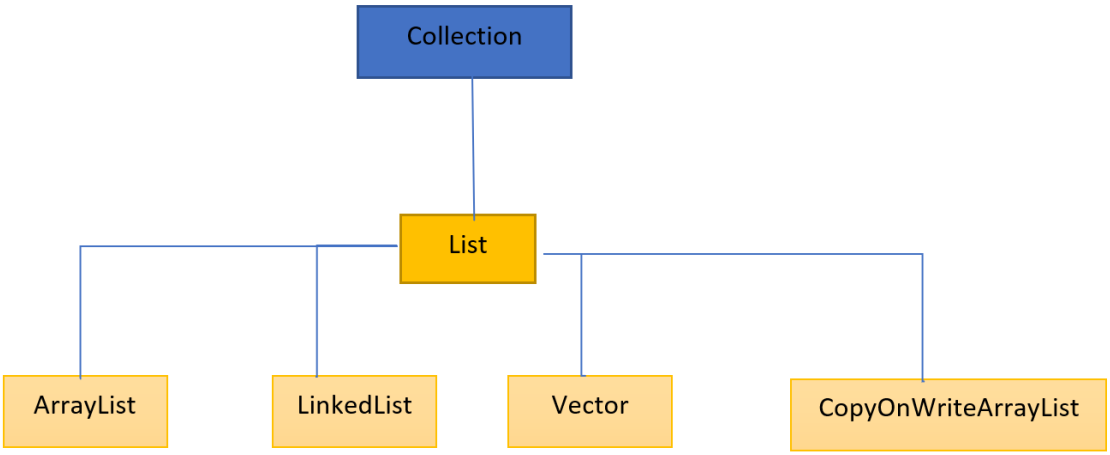
Interfaz List



List es una interfaz (de java.util) que se utiliza para definir listas doblemente enlazadas. En este tipo de listas importa la posición de los objetos, de modo que se pueden recolocar. Deriva de la interfaz Collection por lo que tiene disponible todos sus métodos, pero además aporta métodos mucho más potentes:

Método	Uso
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
Object get(int índice)	Obtiene el elemento almacenado en la colección en la posición que indica el índice
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
List subList(int desde, int hasta)	Obtiene una lista con los elementos que van de la posición desde a la posición hasta

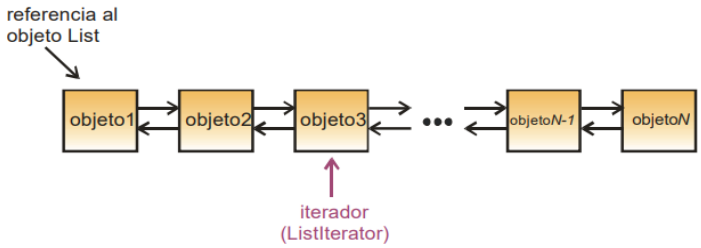
Nota: Cualquier error en los índices produce IndexOutOfBoundsException

Interfaz 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
Object previous()	Obtiene el elemento previo al actual. Si no lo hay provoca excepción: NoSuchElementException
boolean hasPrevious()	Indica si hay elemento anterior al actualmente señalado por el iterador
int nextIndex()	Obtiene el índice del elemento siguiente
int previousIndex()	Obtiene el índice del elemento anterior

Los iteradores de este tipo son mucho más potentes que los de tipo Iterator ya que admiten recorrer la lista en cualquier dirección e incluso ser utilizados para modificar la lista. Además contiene todos los métodos de Iterator (como remove por ejemplo) ya que los hereda



Clase ArrayList

Un ArrayList representa una colección basada en índices, en la que cada objeto de la misma tiene asociado un número (índice) según la posición que ocupa dentro de la colección, siendo 0 la posición del primer elemento

Posee tres constructores:

- ArrayList(). Constructor por defecto. Simplemente crea un ArrayList vacío
- ArrayList(int capacidadInicial). Crea una lista con una capacidad inicial indicada.
- ArrayList(Collection c). Crea una lista a partir de los elementos de la colección indicada.

Creación de un ArrayList

variable_objeto = new ArrayList();

Donde ***variable_objeto*** es la variable que contendrá la referencia al objeto ArrayList creado. Por ejemplo:

ArrayList v = new ArrayList();

Una vez creado, podemos hacer uso de los métodos de la clase **ArrayList** para realizar las operaciones habituales con una colección

Métodos de la clase ArrayList

boolean add(Object o). Añade un nuevo objeto y la colección (su referencia) y lo sitúa **al final** de la misma, devolviendo el valor *true.* Los objetos añadidos pueden ser de cualquier tipo no siendo necesario que todos pertenezcan a la mismn clase:

```
ArrayList v=new ArrayList ();
v.add ("hola"); //añade la cadena en la primera
                //posición del ArrayList
v.add(new Integer(6)); //añade el número en la
                       //segunda posición
```

En este ejemplo observamos cómo, debido a que el ArrayList no puede almacenar tipos básicos, ha sido necesario envolver el número en un objeto *Integer* para poder añadirlo a la colección. Sin embargo, a partir de la versión 5 del lenguaje, gracias a la característica del *autoboxing* es posible añadir directamente el número sin envolverlo previamente en un objeto pues, como ya sabemos, esta operación se lleva a cabo implícitamente, Según esto, la instrucción para añadir el número entero a la colección se puede reescribir para la versión 1.5 de la siguiente forma;

```
v.add(6); //autoboxing
```

void add(int índice, object o). Añade un objeto al ArrayList **en la posición** especificada por *indice,* desplazando hacia delante el resto de los elementos de la colección. El índice de la primera posición es 0.

Object get(int indice). Devuelve el objeto que ocupa la posición indicada. Hay que tener en cuenta que el tipo de devolución es Object, por tanto, para guardar la referencia al objeto devuelto en una variable de su tipo será necesario realizar una conversión explícita:

```
ArrayList v=new ArrayList();
v.add ("texto");
//Conversión explícita a String
String s=(String)v.get(0);
```

En el caso de que se almacenen objetos numéricos, es necesario recordar que la llamada a ***get()*** devuelve el objeto de envoltorio, y no el número:

```
ArrayList v=new ArrayList();
v.add(new Integer(6));
        //Recupera el objeto y muestra un número
Integer i=(Integer)v.get(0);
System.out.println(i.intValue());
```

Utilizando autoboxing/autounboxing la operación anterior podría realizarse de la siguiente manera:

```
ArrayList v=new ArrayList();
v.add(6); //autoboxing
        //Recupera el objeto numérico
int i=(Integer)v.get(0); //unboxing
                        //Muestra el número
System.out.println(i);
```

La realización de conversiones cuando se recupera una referencia almacenada en una colección suele generar cierta confusión entre los programadores Java novatos. Algunas veces se tiende a realizar operaciones como ésta;

```
ArrayList v=new ArrayList();
v.add("35");
//Intenta recuperarlo como objeto numérico
Integer s=(Integer)v.get( 0 );
```

El código anterior compilaría sin ningún problema, sin embargo, al ejecutar la última línea **se produciría una excepción ClassCastException,** dado que no se puede convertir explícitamente un objeto String (eso es lo que se ha almacenado en el objeto de colección) en un Integer. Únicamente **se puede convertir explícitamente el objeto *a su* tipo original.** En el caso de que tengamos que recuperar como número un dato numérico almacenado como de texto en la colección, deberíamos hacerlo del siguiente modo:

```
ArrayList v=new ArrayList();
v.add("35");
//Se recupera en su tipo original
String s=(String)v.get (0);
//Se convierte la cadena a número entero
int n=Integer.parseInt(s);
```

Object remove(int índice). **Elimina** de la colección el objeto **que ocupa la posición** indicada, desplazando hacía atrás los elementos de las posiciones siguientes. Devuelve el objeto eliminado

voíd clear(). **Elimina lodos** los elementos de la colección.

int indexOf(Object o). Localiza en el ArrayLisl **el objeto** indicado como parámetro, **devolviendo su posición**. En caso de que el objeto no se encuentre en la colección, la llamada al método devolverá como resultado el valor -1.

int size(). Devuelve el número de elementos almacenados en la colección. Utilizando este método conjuntamente le con *get(),* se puede recorrer la colección completa. El siguiente método realiza el recorrido de un ArrayList de cadenas para mostrar su contenido en pantalla:

```
public void muestra (ArrayList v)
{
      for(int i=0;i<v.size();i++)
      {
         System.out println((String)v.get(i));
      }
}
```

Como sabemos, a partir de la versión Java *5* se puede utilizar la varíante *for-each* para simplificar el recorrido de colecciones, así el método anterior quedaría:

```
public void muestra (ArrayList v)
{
   for(Object ob:v)
   {
         System.out.println((String)ob);
   }
}
```

Clase LinkedList

Es una clase heredera de las anteriores e implementa métodos que permiten crear listas de adición tanto por delante como por detrás (listas dobles). Desde esta clase es sencillo implantar estructuras en forma de pila o de cola. Añade los métodos:

Método	Uso
Object getFirst()	Obtiene el primer elemento de la lista
Object getLast()	Obtiene el último elemento de la lista
void addFirst(Object o)	Añade el objeto al principio de la lista
void addLast(Object o)	Añade el objeto al final de la lista
void removeFirst()	Borra el primer elemento
void removeLast()	Borra el último elemento

Los métodos están pensados para que las listas creadas mediante objetos LinkedList sirven para añadir elementos por la cabeza o la cola, pero en ningún caso por el interior. En el caso de implementar pilas, los nuevos elementos de la pila se añadirían por la cola (mediante addLast) y se obtendrían por la propia cola (getLast, removeLast). En las colas, los nuevos elementos se añaden por la cola, pero se obtienen por la cabeza (getFirst, removeFirst).


referencia a
la cabeza
referencia a
la cola
objeto1
objeto2
objeto3
objeto N-1
objeto N