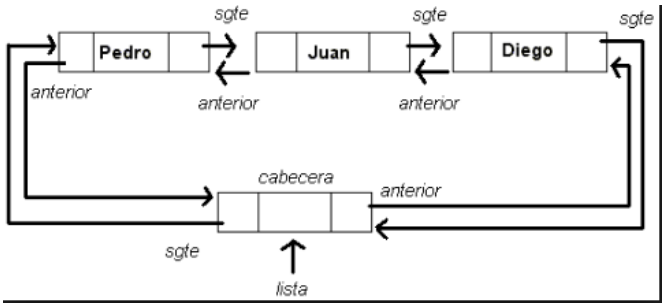
PRÁCTICA A003.- LISTA DOBLEMENTE ENLAZADA CIRCULAR CON CABECERA

Se va a utilizar una **lista doblemente enlazada circular con cabecera** genérica, cuya estructura de datos estará definida por una única referencia a un nodo doble:

- **cab**: referencia a un nodo vacío (sin elemento) que **siempre va a tener la lista**.

  Si la lista es vacía solamente tendrá este nodo cabecera.



Los nodos de la lista doblemente enlazada tienen la siguiente estructura:

- **next**: referencia al siguiente nodo, **o a cab si es el último nodo**
- **previous**: referencia al nodo anterior , o **a cab si es el primer nodo**
- **content**: de tipo genérico, contenido del nodo

PASOS PARA LA REALIZACIÓN DE LA PRÁCTICA:

1. **Descargar el proyecto edi_a003_2019** de la página de la asignatura en agora.unileon.es.
2. **Importar** dicho proyecto en Eclipse : Import… **General**…..**Existing proyects into Workspace**… Select archive file… (indicar el archivo ZIP descargado)
3. El proyecto edi-a003-2019 contiene el fichero **DoubleLinkedList**, donde define el interface para la lista doblemente enlazada circular. **Este fichero NO SE PUEDE MODIFICAR.**

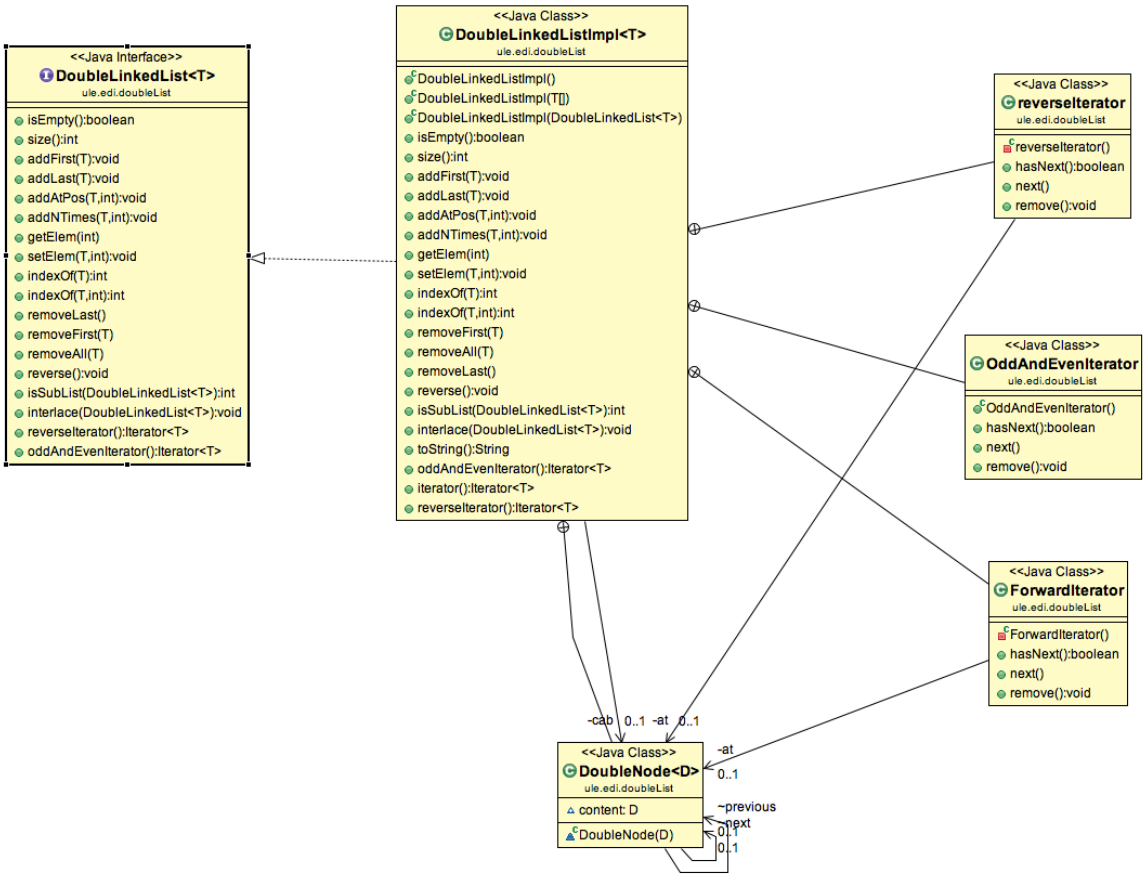
```
package ule.edi.doubleList;

import java.util.Iterator;

public interface DoubleLinkedList<T> extends Iterable<T> {

	/**
	 * indica si la lista está vacía.
	 *  @return true si la lista es vacía, false en caso contrario
	 */
	public boolean isEmpty();

	/**
	 * Devuelve el número de elementos de la lista.
	 *  @return numero de elementos de la lista
	 */
	public int size();

	/**
	 * Inserta un elemento como primero.
	 * @param element a insertar
	 */
	public void addFirst(T element);

	/**
	 * Inserta un elemento como último.
	 * @param element a insertar
```

```
     */
    public void addLast(T element);


    /**
     * Inserta el elemento en la posición p, desplazando los elementos a partir de esa posición.
     * Si la lista tiene menos de n elementos lo insertará como último elemento.
     *
     * Si la lista era [A, B, C] :
     *       lista.addAtPos("Z", 1) dejará la lista como [Z, A, B, C].
     *       lista.addAtPos("Z", 3) dejará la lista como [A, B, Z, C].
     *       lista.addAtPos("Z", 5) dejará la lista como [A, B, C, Z].
     *
     * @param element a insertar
     * @param pos posicion en la que se insertará el elemento, desplazando los siguientes
     */
    public void addAtPos(T element, int p);

    /**
     * inserta n veces el elemento al final de la lista.
     * Si lista=[A, B, C], lista.addNTimes("Z", 4) dejará la lista como: [A, B, C, Z, Z, Z, Z]
     *
     * @param element a insertar
     * @param p posicion en la que se insertará el elemento, desplazando los siguientes
     */
    public void addNTimes(T element, int n);

    /**
     * Devuelve el elemento que ocupa la posición p en la lista (las posiciones empiezan en 1).
     *  Dispara IndexOutOfBoundsException si no existen p elementos (la lista tiene menos de p elementos).
     *
     * @param p posicion que ocupa el elemento a devolver
     * @return el elemento que ocupa la posición p en la lista
     * @throws IndexOutOfBoundsException si no existen p elementos (la lista tiene menos de p elementos).
     */
    public T getElem(int p);

    /**
     * Sustituye el elemento que ocupa la posición p en la lista (las posiciones empiezan en 1).
     *  Dispara IndexOutOfBoundsException si no existen p elementos (la lista tiene menos de p elementos).
     *
     * @param p posicion que ocupa el elemento a devolver
     * @throws IndexOutOfBoundsException si no existen p elementos (la lista tiene menos de p elementos).
     */
    public void setElem(T elem, int p);

    /**
     * Indica la posición donde se encuentra la primera aparición de elem desde el principio de la lista
     * (las posiciones empiezan en 1).
     * Dispara la excepción NoSuchElementException si no se encuentra el elemento en la lista.
     *
     * @param elem  el elemento a buscar
     * @return la posición que ocupa el elemento en la lista
     * @throws NoSuchElementException si no se encuentra el elemento en la lista.
     */
    public int indexOf(T elem);

    /**
     * Indica la posición donde se encuentra la primera aparición de elem desde desde la posición p incluida
     * Dispara la excepción IndexOutOfBoundsException si no hay p elementos en la lista.
     * Dispara la excepción NoSuchElementException si no se encuentra el elemento en la lista a partir
     * de la posición indicada incluida habiendo al menos p elementos en la lista.
     *
     * @param elem  el elemento a buscar
     * @param p posicion a partir de la que se busca el elemento
     * @return la posición que ocupa el elemento en la lista a partir de la posición p
     * @throws IndexOutOfBoundsException si no existen p elementos (la lista tiene menos de p elementos).
     * @throws NoSuchElementException si no se encuentra el elemento en la lista a partir de la posición p (habiendo al menos
     * p elementos en la lista)
     */
    public int indexOf(T elem, int p);

    /**
     * Elimina el último elemento de la lista.
     *
     * @return el elemento que es eliminado
     * @throws EmptyCollectionException si la lista está vacía
     */
    public T removeLast() throws EmptyCollectionException;

    /**
     * Elimina la primera aparición del elemento.
     * Si la lista es vacía dispara la excepción EmptyCollectionException.
     *
     * Si lista=[A, C, B, C, D, C]
     *                 lista.removeFirst("C") dejará a lista=[A, B, C, D, C]
     *
     * @param elem  el elemento a eliminar
     * @return el elemento que es eliminado
     * @throws EmptyCollectionException si la lista está vacía
```

```
     * @throws NoSuchElementException si no se encuentra el elemento en la lista
     */
    public T removeFirst(T elem) throws EmptyCollectionException;

    /**
     * Elimina todas las apariciones del elemento.
     * Si la lista es vacía dispara la excepción EmptyCollectionException.
     *
     * Si lista=[A, C, B, C, D, C]
     *                 lista.removeAll("C") dejará a lista=[A, B, D]
     *
     * @param elem  el elemento a eliminar
     * @return la última aparición del elemento que es eliminado
     * @throws EmptyCollectionException si la lista está vacía
     * @throws NoSuchElementException si no se encuentra el elemento en la lista
     */
    public T removeAll(T elem) throws EmptyCollectionException;

    /**
     * Invierte la lista actual.
     * Por ejemplo, si la lista era [A, B, C], después de la llamada al método será [C,B,A]
     *
     */
    public void reverse();

    /**
     * Indica a partir de qué posición se encuentra la sublista pasada como parámetro en la lista actual
     *          o -1 si no se encuentra.
     * Si part es vacía devuelve 1
     *
     * Ejemplos:
     *
     *  [A, B, A, B, C], con part=[B, A, X], devolvería -1
     *  [A, B, A, B, C], con part=[B, A], devolvería 2
     *
     *  [A, B, A, B], con part=[A, B], devolvería 1
     *
     *  [A, B, A, B, C, X, A], con part=[B, C, X], devolvería 4
     *
     * @param part lista a comprobar si es sublista de la actual
     * @return posición a partir de la que se encuentra la sublista en la lista actual
     */
    public int isSubList(DoubleLinkedList<T> part);

    /**
     * inserta los elementos de la lista pasada como parámetro en la lista actual, entrelazando sus elementos.
     * Es decir, el primer elemento de other, lo dejará detrás del primer elemento de la lista actual, y así sucesivamente.
     * La lista other no se modifica, por lo que deben crearse nuevos nodos en la lista actual con el mismo contenido que los
     *     nodos de other.
     *  Es decir, no se utilizarán los mismos nodos de other para insertarlos en la lista actual.
     *
     *  Por ejemplo,
     *      Si lista=[A, B, C, D, E] y other=[X,Y,Z]:    lista.interlace(other) dejará a lista=[A, X, B, Y, C, Z, D, E]
     *      Si lista=[A, B, C] y other=[V, W, X,Y,Z]:    lista.interlace(other) dejará a lista=[A, V, B, W, C, X, Y, Z]
     *      Si lista=[A, B, C] y other=[]:               lista.interlace(other) dejará a lista=[A, B, C]
     *      Si lista=[] y other=[A, B]:                  lista.interlace(other) dejará a lista=[A, B]
     *
     * @param other lista a intercalar con la actual
     */
    public void interlace(DoubleLinkedList<T> other);

    /// ITERADORES

    /**
     * Devuelve un iterador que recorre la lista en orden inverso.
     *
     * Por ejemplo, para una lista x con elementos [A, B, C, D, E]
     *
     * el iterador creado con x.reverseIterator() devuelve en sucesivas llamadas a next(): E, D, C, B y A.
     *
     * @return iterador para orden inverso.
     */
    public Iterator<T> reverseIterator();

    /**
     * Devuelve un iterador que recorre la lista recorriendo primero los elementos que ocupan posiciones pares y luego los
     *     que ocupen posiciones impares.
     *
     * Por ejemplo, para una lista x con elementos [A, B, C, D, E]
     *
     *          el iterador creado con x.oddAndEvenIterator() devuelve en sucesivas llamadas a next(): B, D, A, C y E.
     *
     * @return iterador para orden pares e impares.
     */
    public Iterator<T> oddAndEvenIterator();
}
```

4. El proyecto edi-a003-2019, contiene otro fichero java **DoubleLinkedListImpl** que corresponde con la implementación del interface mediante una lista doble circular con nodo cabecera descrita anteriormente.

ESTRUCTURAS DE DATOS UTILIZADAS:




```
public class DoubleLinkedListImpl<T> implements DoubleLinkedList<T> {
	/**
	 * Nodo doblemente enlazado.
	 *
	 * Como es estática, no tiene en ámbito el parámetro 'T' de la clase que la contiene. El parámetro 'D' será sustituído
	 * por un tipo particular cuando se use el nodo, por ejemplo:
	 *
	 *                  DoubleNode<T> cab;        en la lista.
	 *
	 * @param <D>
	 */
	static class DoubleNode<D> {

		DoubleNode(D element) {
			this.content = element;
			this.previous = null;
			this.next = null;
		}

		//        dato a almacenar en el nodo
		D content;

		DoubleNode<D> previous;

		DoubleNode<D> next;
	}

	/**
	 * El atributo cab apunta al nodo cabecera; siempre habrá un nodo vacío (sin elemento) que actúa de cabecera
	 *  OJO!!! ESTE NODO CABECERA DEBERÁ CREARSE EN CADA CONSTRUCTOR DE LA LISTA
	 */
	private DoubleNode<T> cab;
```

```
//////////////////////
/////  CONSTRUCTORES
//////////////////////


/**
 * Construye una lista vacía.
 */
public DoubleLinkedListImpl() {
          //TODO
          // Deberá crear el nodo cabecera vacío
}

/**
 * Construye una lista con los elementos dados.
 *
 * Java creará un array 'elements' con los dados en la llamada al constructor; por ejemplo:
 *
 *        x = new DoubleLinkedList<String>("A", "B", "C");
 *
 * ejecuta este método con un array [A, B, C] en 'elements'.
 *
 * @param elements
 */
public DoubleLinkedListImpl(T ... elements) {
          //TODO
}

/**
 * Construye una lista a partir de otra.
 *
 * Las listas tienen nodos independientes, con los mismos contenidos.
 */
public DoubleLinkedListImpl(DoubleLinkedList<T> other) {
          //TODO
}

//////////////////////
/////  ITERADORES
//////////////////////

private class ForwardIterator implements Iterator<T> {

          private DoubleNode<T> at ;

          @Override
          public boolean hasNext() {
                    return false;
                    // TODO Auto-generated method stub
          }

          @Override
          public T next() {
                    return null;
                    // TODO Auto-generated method stub
          }

          @Override
          public void remove() {
                    // TODO Auto-generated method stub
                    throw new UnsupportedOperationException();
          }
};


private class reverseIterator implements Iterator<T> {

          …

};

private class OddAndEvenIterator implements Iterator<T> {

          // Definir los atributos necesarios para implementar el iterador
};


////// FIN DE ITERADORES ///////
////////////////////////////////
```

5. A la vez que se van desarrollando las clases anteriores se deben crear las correspondientes clases de pruebas JUnit 4 (cuyo nombre debe acabar en Test) para ir comprobando su correcto funcionamiento.
6. **Se deberá entregar en agora.unileon.es la versión final de la práctica.** Para ello habrá que exportar el proyecto edi-a003-2019 como zip (Export… General… Archive File)

FECHA LIMITE DE ENTREGA DE ESTA PRÁCTICA: 14 de Abril de 2019 23:55

- **IMPORTANTE**: Utilizar JUNIT 4.

- Se valorará la cobertura total de los test implementados, por lo que debe usarse un plugin de Eclipse para comprobar la cobertura de los test generados. Se buscará el 100% en la cobertura del fichero DoubleLinkedListImpl.