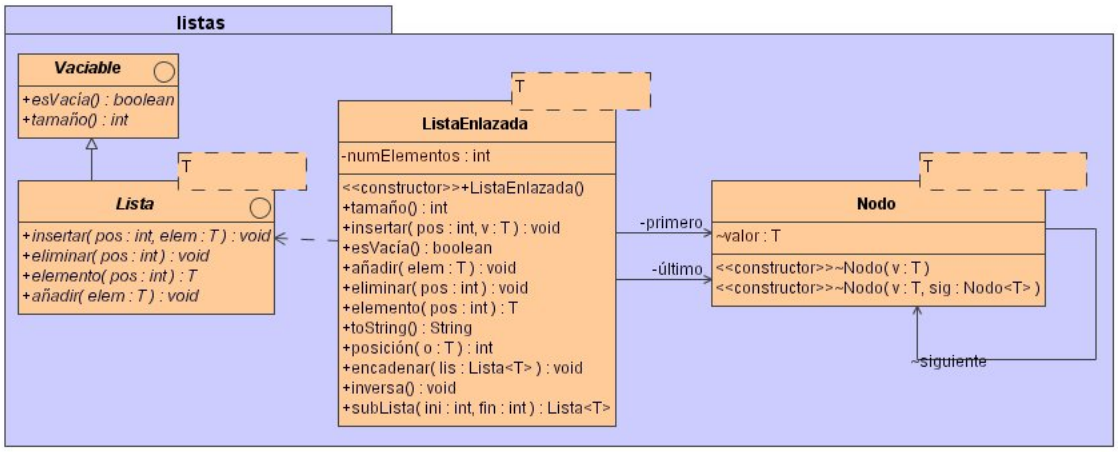
En esta práctica construiremos y usaremos dos paquetes: el primero, listas, que implementa listas genéricas y el segundo, colas, que implementa colas y colas de prioridades genéricas.

Ejercicio 1 (proyecto prListas, paquete listas)

Se deberá implementar un paquete `listas` que deberá contener dos interfaces y dos clases tal y como aparece en el siguiente diagrama:



El protocolo que deben implementar las listas se definirá en la interfaz `Lista<T>`, que heredará de una interfaz `Vaciable` que se refiere a todas aquellas clases de objetos contenedores que pueden aumentar y disminuir de tamaño y, eventualmente, quedarse vacíos. Las clases serán: `ListaEnlazada<T>`, que corresponde a una implementación genérica de listas como secuencias enlazadas de nodos con acceso directo al primero y al último nodos, y la clase `Nodo<T>`, que deberá ser una clase privada interna de la clase anterior para que sus objetos no sean accesibles desde fuera.

Los métodos de la clase `ListaEnlazada<T>` son autoexplicativos salvo el método `añadir` que se refiere a una operación para añadir un elemento al final de la lista. El paquete se probará con la clase `TestListas` siguiente (esta clase de prueba no debe formar parte del paquete):

```
import listas.*;
public class TestListas {
    public static void main(String[] args) {
        Lista<Number> lis = new ListaEnlazada<Number>();
        lis.insertar(0, new Integer(1000));
        System.out.println (lis);
        for (int i = 0; i < 10; i++) {
            lis.añadir(i);
        }
        lis.insertar(11, 1.5);
        System.out.println(lis);
        lis.eliminar(11);

        System.out.println(lis);
        System.out.println("Contenido de la lista: " +  lis
                                        + " 0: " + lis.elemento(0)
                                        + " 5: " + lis.elemento(5)
                                        + " 10: " + lis.elemento(10));
        lis.insertar(1, 2000);
        System.out.println(lis);
    }
}
```

Ejercicio 2 (proyecto `prPrimos`)

Implementad una aplicación `ListaPrimos` que calcule los números primos menores que un número determinado, introducido desde teclado, para lo cual utilizaremos la lista del apartado anterior de forma que cada uno de sus nodos contenga un número primo. El primer nodo contendrá el número 2 y el último el mayor primo menor o igual que el número dado al ejecutar el programa.

La lista de celdas que se deberá generar será como la siguiente:

2	3	5	7	11	...	n

Para construir esta lista la aplicación introducirá el número 2 en la primera posición de una lista inicialmente vacía, y realizará un proceso iterativo desde el 3 hasta el tope dado de manera que se filtre cada número con los números de la lista. El procedimiento de filtrado comienza en la primera posición y actúa de la siguiente manera sobre cada número i:

- Si i es múltiplo del primo contenido en la posición actual no hacemos nada (detenemos el proceso).
- Si i no es múltiplo del primo que contiene la posición actual pasamos a considerar la siguiente posición:
- Si esa posición es una posición válida procedemos a realizar el filtrado desde esa posición.
- Si esa posición no es una posición válida (no tenemos más elementos en la lista) introducimos el valor i en dicha posición de la lista (al final de ésta).

Veamos un ejemplo de creación de la lista: 3, 4, 5, 6, 7, 8, ..., tope. Partimos de la lista:

2

Como el 3 no es múltiplo de 2 y la posición actual (con el 2) es el último elemento de la lista concluimos que el 3 es primo y lo añadimos al final.

2	3

El 4 se ignora porque es múltiplo del 2, valor de la primera celda. El 5 pasa por el 2, sigue con el 3 y se inserta al final.

2	3	5

El 6 se filtra con el 2. El 7 pasa por el 2, el 3, el 5 y se inserta al final.

2	3	5	7

El 8 se filtra con el 2. El 9 pasa el 2, pero se filtra con el 3. El 10 se filtra con el 2. El 11 pasa por el 2, el 3, el 5, el 7 y se inserta al final.

2	3	5	7	11

El proceso continua de la misma forma con cada uno de los números entre 12 y tope.

Ejercicios adicionales

1.- Agregar los siguientes métodos a la interfaz `Lista<T>` e implementarlos en `ListaEnlazada<T>`.

`int posición(T elem);` que devuelve la posición de `elem` en la lista receptora o –1 si no está. La comparación de elementos se hará con `equals`.
`void encadenar(Lista<? extends T> m);` `que` encadena los elementos de la lista `m` tras los de la lista receptora.
`void inversa();` que invierte los elementos de la lista receptora.
`Lista<T> subLista(int in, int fin);` `que` devuelve una nueva lista con los elementos de la receptora que ocupan las posiciones entre `in` (inclusive) y `fin` (exclusive). Si no hay elementos en este rango se devuelve una lista vacía.

2.- Una operación muy frecuente sobre las estructuras de datos compuestas es la de "iterar" sobre sus elementos, es decir, pasar uno por uno por cada uno de los componentes de dicha estructura, realizando alguna operación sobre ellos. Para realizar estos recorridos Java recomienda crear objetos especiales, cuyo comportamiento se recoge en la interfaz `Iterator<T>`, asociados a las estructuras y declarar que las clases de las estructuras implementan la interfaz `Iterable<T>`. La interfaz `Iterator<T>` está en el paquete `java.util`, y define un protocolo muy sencillo:

```
public interface Iterator<T> {
   public boolean hasNext();
   public T next();
   public void remove();
}
```

con operaciones para controlar el iterador, mientras que la interfaz `Iterable<T>`:

```
public interface Iterable<T> {
   Iterator<T> iterator();
}
```

tan solo exige la implementación de un método `iterator()` que debe ser el encargado de generar un iterador para la estructura. De esta forma, se consigue también disponer el bucle "*for each*" -utilizado sobre arrays- habilitado para todas aquellas clases que implementen la interfaz `Iterable<T>`. Lo que pretendemos es tener una forma de iterar sobre los elementos de una estructura de datos, utilizando el siguiente procedimiento: mientras haya elementos (`hasNext() == true`) dame el siguiente elemento (`next()`).

Para hacer que nuestras listas sean iterables, procederemos del siguiente modo:

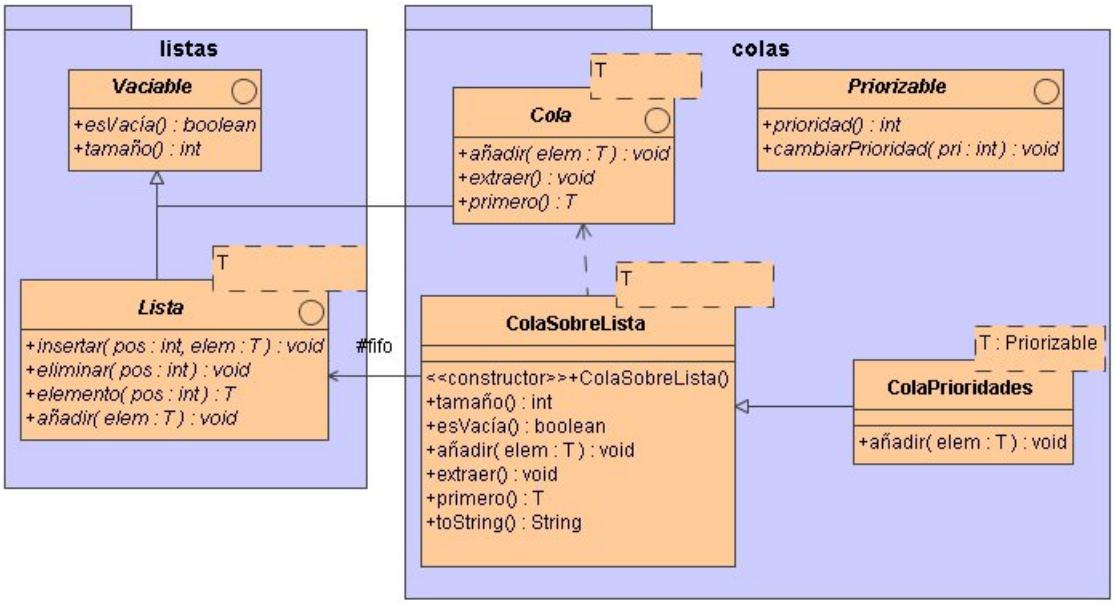
(a) indicando en al cabecera de la interfaz `Lista<T>` que también extiende `Iterable<T>`.
(b) implementando el método `iterator()` en la clase `ListaEnlazada<T>`; pero como este método debe devolver un iterador sobre listas enlazadas, habrá que definir la clase correspondiente a estos objetos. Como cada iterador debe acceder al estado de la correspondiente estructura, y el estado es privado, la clase del iterador debe ser una clase anidada de la clase `ListaEnlazada<T>`, no estática, en concreto una clase `IterListaEnlazada` que implemente la interfaz `Iterator<T>`:

```
protected class IterListaEnlazada implements Iterator<T> {
   Nodo<T> posición; // indicará la posición actual del recorrido
   protected IterListaEnlazada() {
          posición = primero;
   }
   public boolean hasNext() {
          return posición != null;
   }
   public T next() {
          if (posición == null) {
                throw new NoSuchElementException();
          }
          Nodo<T> aux = posición;
          posición = posición.siguiente;
          return aux.valor;
   }
   public void remove() { // no se implementa realmente
          throw new UnsupportedOperationException();
   }
}
```

Ejercicio 3 (proyecto prColas, paquete colas)

El segundo paquete a considerar es el paquete `colas`, de colas y colas de prioridades, cuyas interfaces y clases se muestran en la figura siguiente. Este paquete necesita usar interfaces y clases del paquete `listas`. En particular, la interfaz `Cola<T>` heredará de la interfaz `Vaciable`, y la clase `ColaSobreLista<T>` que implementa la interfaz `Cola<T>` representará una cola sobre una lista de elementos del tipo correspondiente.

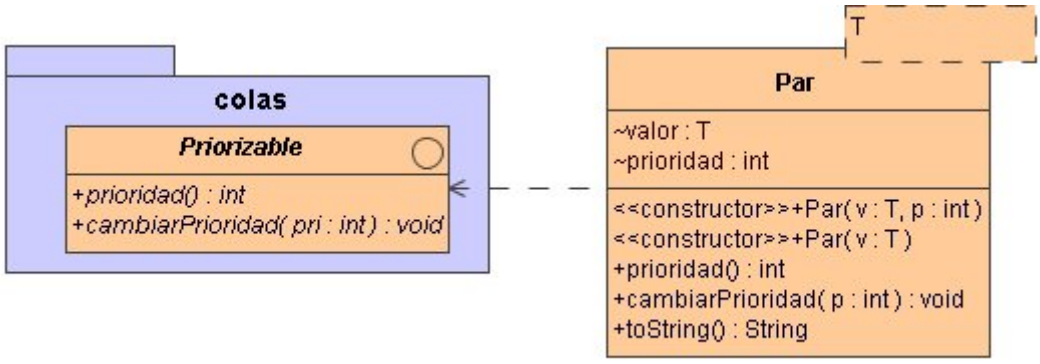
Además de la interfaz `Cola<T>`, que describe el comportamiento de una cola y hereda de la interfaz `Vaciable`, el paquete incluirá una segunda interfaz `Priorizable`. En él también se definen dos clases: `ColaSobreLista<T>`, que implementa la interfaz `Cola<T>`, y la clase `ColaPrioridades<T extends Priorizable>`, que hereda de `ColaSobreLista<T>` y restringe el tipo genérico exigiendo que responda a la interfaz `Priorizable`. Con esta interfaz aseguramos que todo elemento que se introduzca en una cola de prioridades responda a la interfaz `Priorizable`, y por tanto nos asegure que de una forma o de otra todo elemento en la cola tiene asociada una prioridad.



Implementad el paquete `colas` teniendo en cuenta:

- La clase `ColaSobreLista<T>` responde a la interfaz `Cola<T>` y responde un protocolo FIFO (el primer elemento en entrar debe ser el primero en salir).
- Una cola de prioridades describe una estructura donde los elementos que se almacenan tienen asociada una prioridad (de tipo entero), y se organizan igual que una cola, pero teniendo en cuenta no sólo el orden de inserción de los elementos en la cola, sino también la prioridad asociada. En la clase `ColaPrioridades<T extends Priorizable>` se añade un elemento a la cola teniendo en cuenta que éste ha de ser extraído antes que cualquiera de prioridad menor o cualquier otro de la misma prioridad insertado posteriormente (FIFO).

Probad el paquete con la clase `TestColas` y la clase `Par<T>` dadas más abajo. Obsérvese que la clase `Par<T>` implementa la interfaz `Priorizable` almacenando explícitamente un valor de tipo `T` y su prioridad asociada. Otras clases pueden implementar dicha interfaz de distinta forma.




```
import colas.*;

class TestColas {
    public static void main (String[] args) {
        Cola<Integer> c = new ColaSobreLista<Integer>();
        for (int i = 0; i < 10; i++) {
            c.añadir(i * 2);
        }
        System.out.println("La cola es: " + c);
        c.extraer();
        c.extraer();
        System.out.println("La cola es ahora: " + c);
        System.out.println("Y el primer elemento: " + c.primero());

        ColaPrioridades<Par<Integer>> cp = new ColaPrioridades<Par<Integer>>();
        System.out.println(cp);
        for (int i = 0; i < 10; i++) {
            cp.añadir(new Par<Integer>(i * 2, i % 3));
        }
        System.out.println("La cola es: " + cp);
    }
}
```

```
import colas.*;

class Par<T> implements Priorizable {
    T valor;
    int prioridad;

    public Par(T v, int p) {
        valor = v;
        prioridad = p;
    }

    public Par(T v) {
        valor = v;
        prioridad = 0;
    }

    public int prioridad() {
        return prioridad;
    }

    public void cambiarPrioridad(int p) {
        prioridad = p;
    }

    public String toString() {
        return "(" + valor + ", " + prioridad + ")";
    }
}
```

Ejercicios adicionales

1.- Realizad una implementación estática de la interfaz `Lista<T>`, basada en arrays.
2.- Realizad una implementación estática de la interfaz `Cola<T>`, basada en arrays