

Tema 1 – S1

Estructuras de Datos (EDAs), en Java

Contenidos

1. Estructuras de Datos (EDAs): motivación, definición y clasificación
2. Diseño de una EDA en Java
 - 2.1. Jerarquía Java de una EDA: componentes y nomenclatura
 - 2.2. Organización de jerarquías de EDAs en librerías (*packages BlueJ*)
 - 2.3. Criterios de diseño de las clases de una jerarquía. Ejemplos para las jerarquías Lineales de *Pila* y *Cola*

1. Estructuras de Datos

Motivación

- Cualquier aplicación informática exige manipular/gestionar Colecciones de Datos de talla elevada y, por lo general, dinámicas ...
 - EFICIENTEMENTE
 - Permitiendo la REUTILIZACIÓN DEL SOFTWARE
- ➔ Resulta imprescindible elegir la mejor Estructuración –u organización- **de los Datos (EDA)** ...

1. Estructuras de Datos

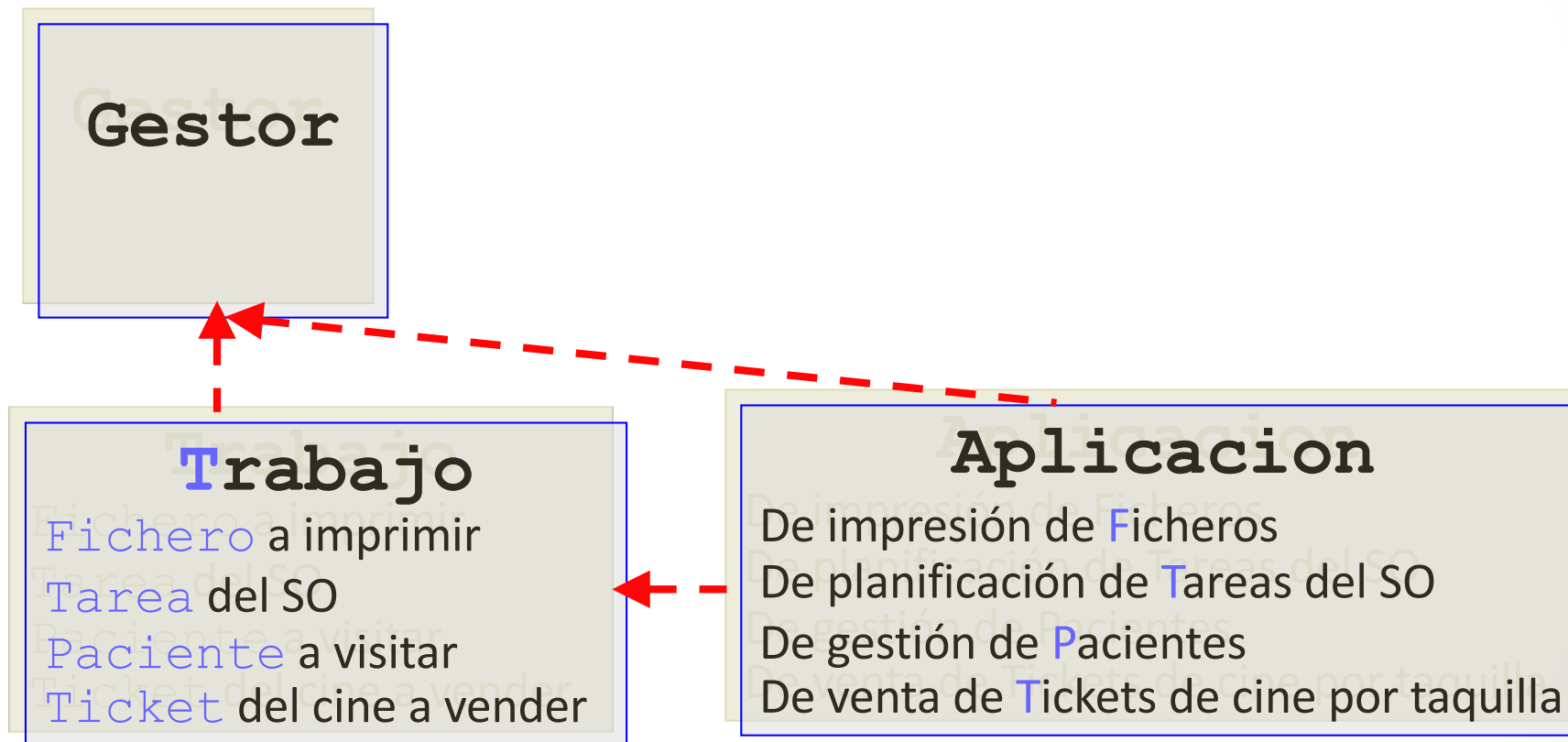
Motivación

Algunos ejemplos de modelos de gestión de datos, así como del coste asociado a su implementación

1. Estructuras de Datos

Ejemplo 1: aplicación de gestión de una Colección de Trabajos (I)

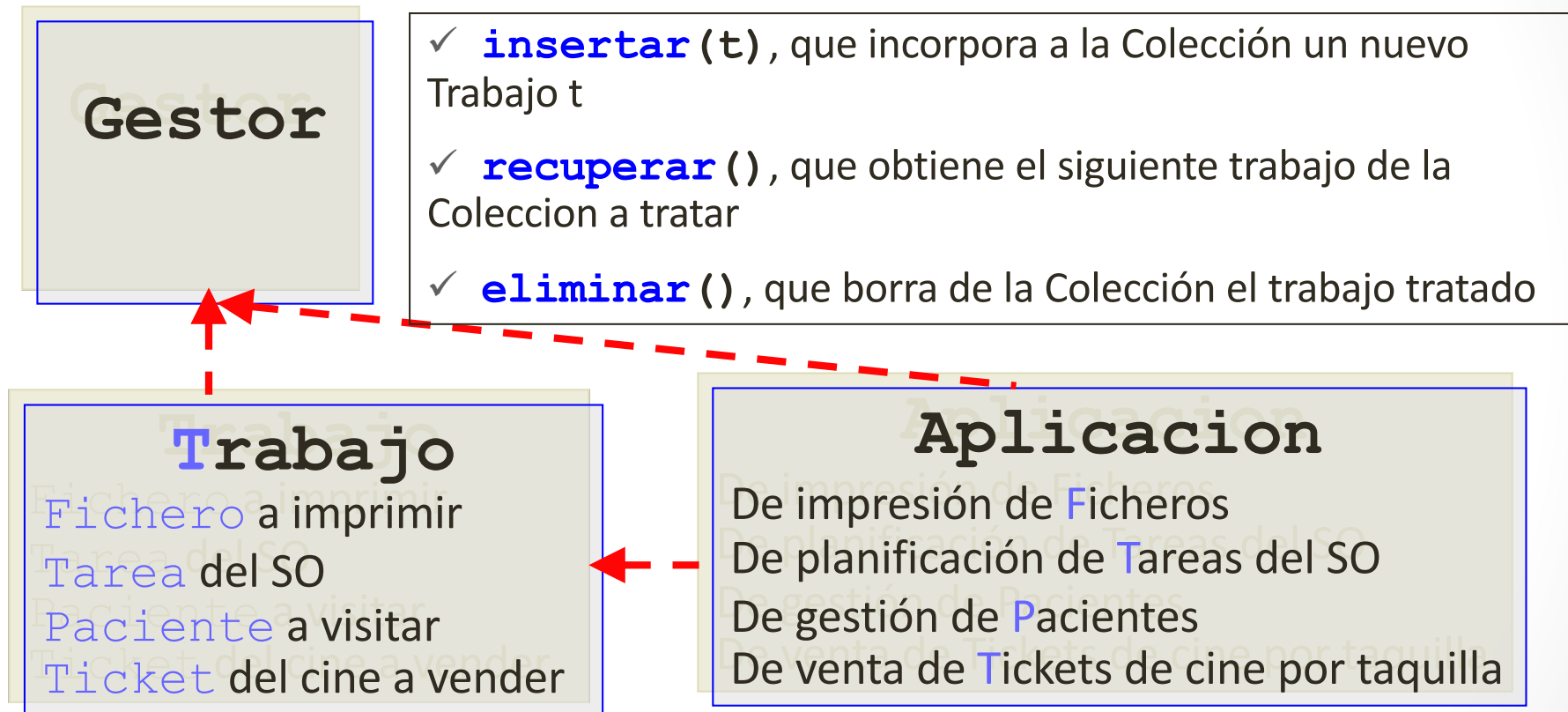
(a) Clases de la aplicación y relaciones entre ellas



1. Estructuras de Datos

Ejemplo 1: aplicación de gestión de una Colección de Trabajos (II)

(b) Operaciones básicas para la gestión de los Trabajos

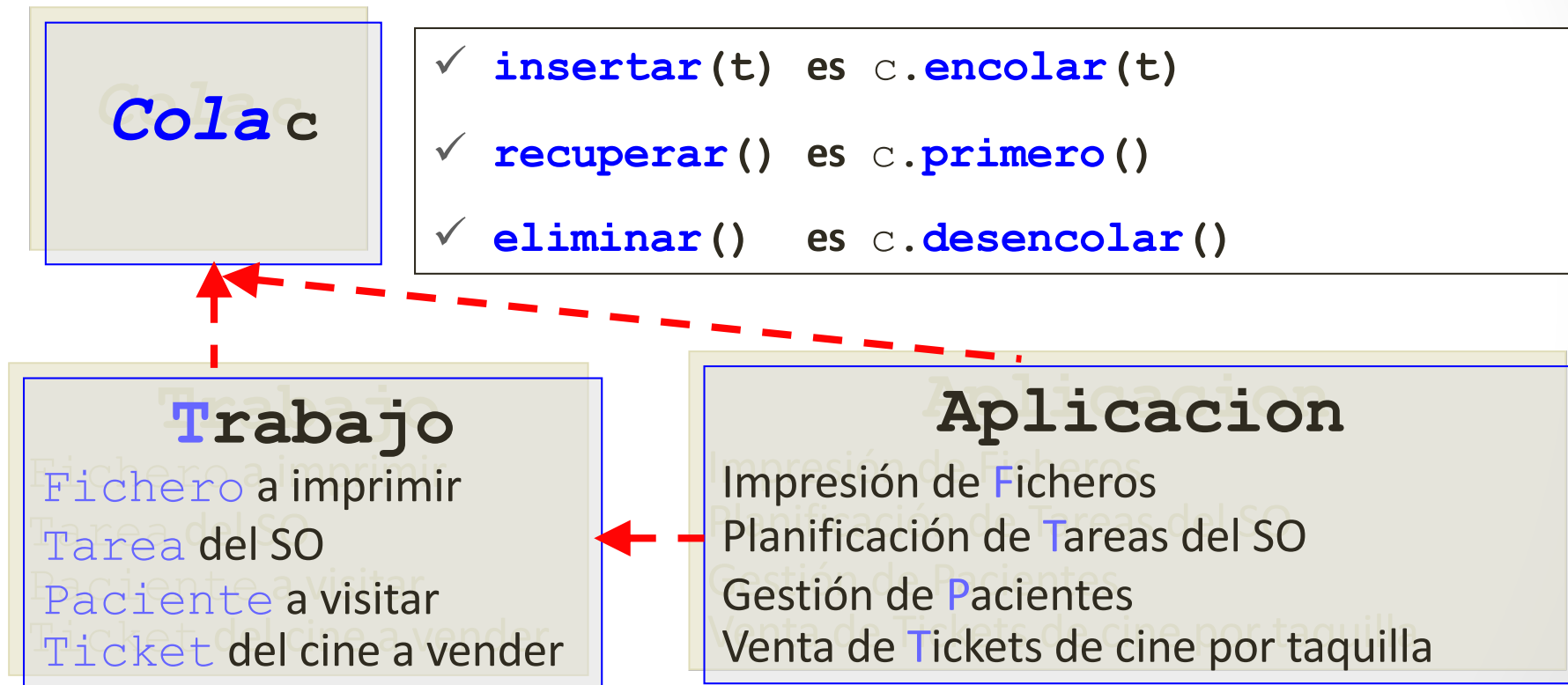


¿Cuál es el **modelo** de gestión de trabajos, o gestor, más simple y eficiente que conoces?

1. Estructuras de Datos

Ejemplo 1: aplicación de gestión de una Colección de Trabajos (III)

(b. 1) Operaciones básicas para la gestión de los Trabajos: Modelo **FIFO**



1. Estructuras de Datos

Ejemplo 1: aplicación de gestión de una Colección de Trabajos (IV)

(c. 1) Coste de la gestión **FIFO**

Operación del modelo

`c.encolar(t)`

`c.desencolar()`

`c.primer()`

Coste promedio asociado

constante

constante

constante

Soporte de Datos en memoria:
array o **LEG**

1. Estructuras de Datos

***Ejemplo 1:** aplicación de gestión de una Colección de Trabajos (V)*
Cuestión sobre el modelo de gestión FIFO

A pesar de su sencillez y eficiencia ...

¿Sirve un modelo de gestión FIFO cuando los trabajos de la Colección se quieren **tratar en función de su prioridad** -máxima prioridad, mínimo tiempo de espera?

1. Estructuras de Datos

Ejemplo 1: aplicación de gestión de una Colección de Trabajos (VI)

(b. 2) Operaciones básicas para la gestión de los Trabajos: Modelo **Cola de Prioridad**

**Cola de
Prioridad** qP

- ✓ **insertar**(t) según su prioridad, o qP .**insertar**(t)
- ✓ **recuperar**() el Trabajo de Máxima prioridad, o qP .**recuperarMin**()
- ✓ **eliminar**() el Trabajo de Máxima prioridad, o qP .**eliminarMin**()

Trabajo con prioridad

Fichero a imprimir (longitud)

Tarea del SO (tipo)

Paciente a visitar (gravedad)

Aplicación de optimización

Impresión de **Ficheros** según tiempos

Planificación de **Tareas** del SO eficiente

Gestión de **Pacientes** según gravedad

- Modelo de *Búsqueda Dinámica del Dato con Máxima Prioridad*
- Los trabajos deben ser **Comparable**s -dados dos trabajos, ¿cuál es el de máxima prioridad?
- A igualdad de prioridades, la Cola de Prioridad funciona como una Cola

1. Estructuras de Datos

Ejemplo 1: aplicación de gestión de una Colección de Trabajos (VII)
(c. 2) Coste de la gestión de una **Cola de Prioridad**

Operación del modelo

`insertar(t)`
`recuperarMin()`
`eliminarMin()`

Coste promedio asociado

lineal
constante
constante

Soporte de Datos en memoria:
LEG ordenada

constante

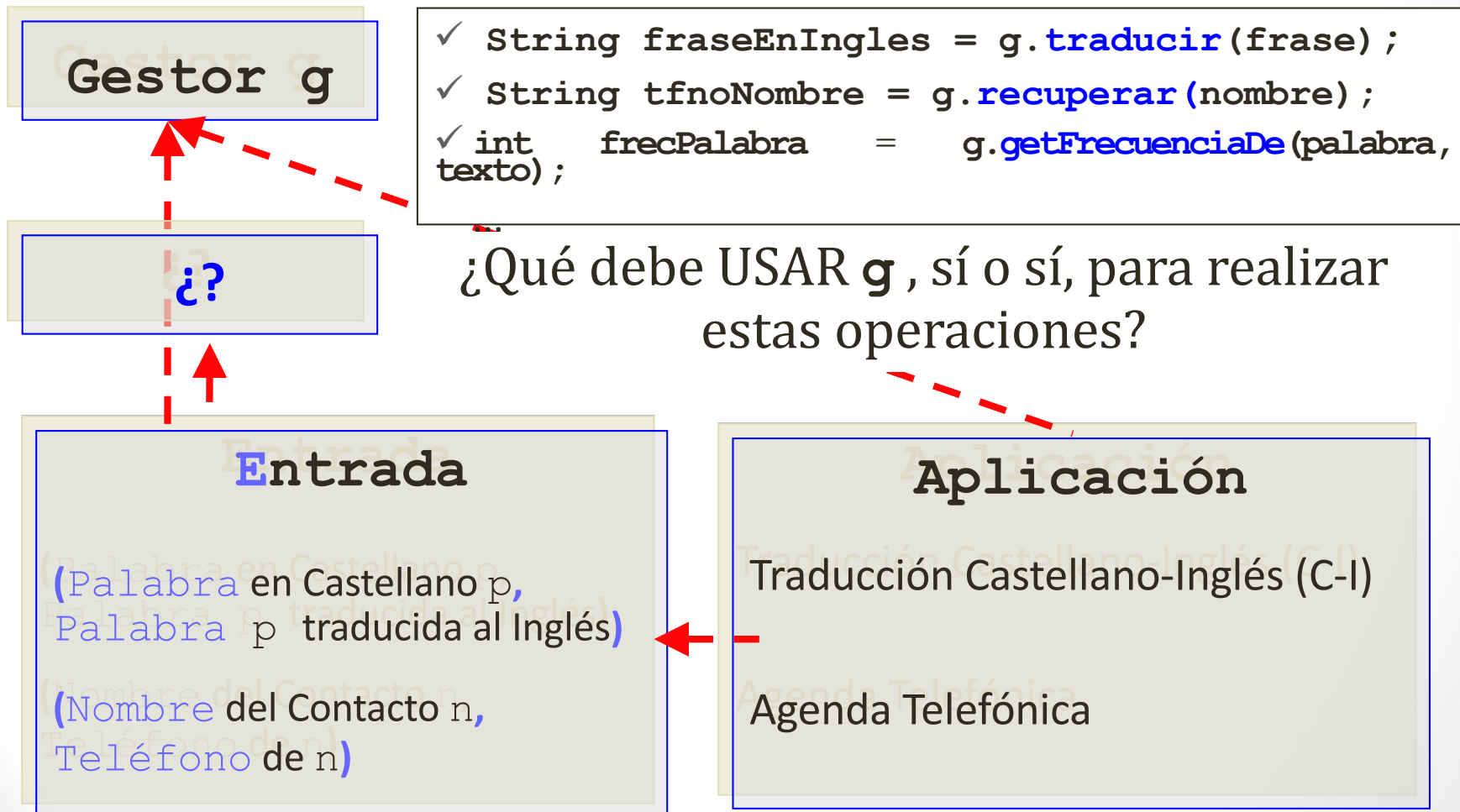
constante

logarítmico

Soporte de Datos en memoria:
Heap, un array "especial"

1. Estructuras de Datos

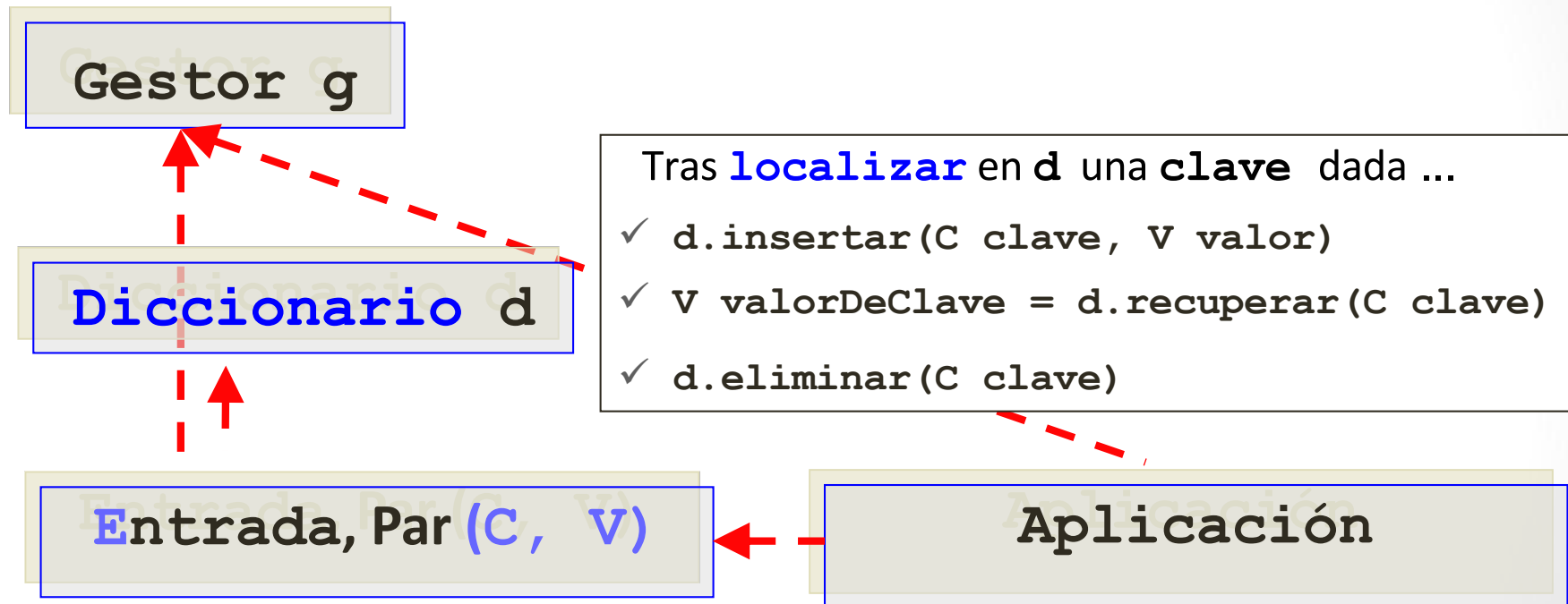
Ejemplo 2: aplicación de gestión de una Colección de Entradas (I)



1. Estructuras de Datos

Ejemplo 2: aplicación de gestión de una Colección de Entradas (II)

(b) Operaciones **básicas** para la gestión de Entradas: Modelo **Diccionario**



- Modelo de *Búsqueda Dinámica (del Valor)* de una Entrada de clave dada
- Dos Entradas son **iguales** si tienen la misma clave
- No hay Entradas repetidas

1. Estructuras de Datos

Ejemplo 2: aplicación de gestión de una Colección de Entradas (III)
(c) Coste de la gestión de un **Diccionario**

Operación del modelo

`localizar(C clave)`

Coste promedio asociado

lineal

Soporte de Datos en memoria:
LEG ordenada

constante

Soporte de Datos en memoria:
Tabla Hash, un array "especial"

1. Estructuras de Datos

Definición (sigue ...)

- Una **EDA** es el conjunto formado por las operaciones que definen el comportamiento o funcionalidad de una Colección de Datos y la posible representación en memoria de ésta

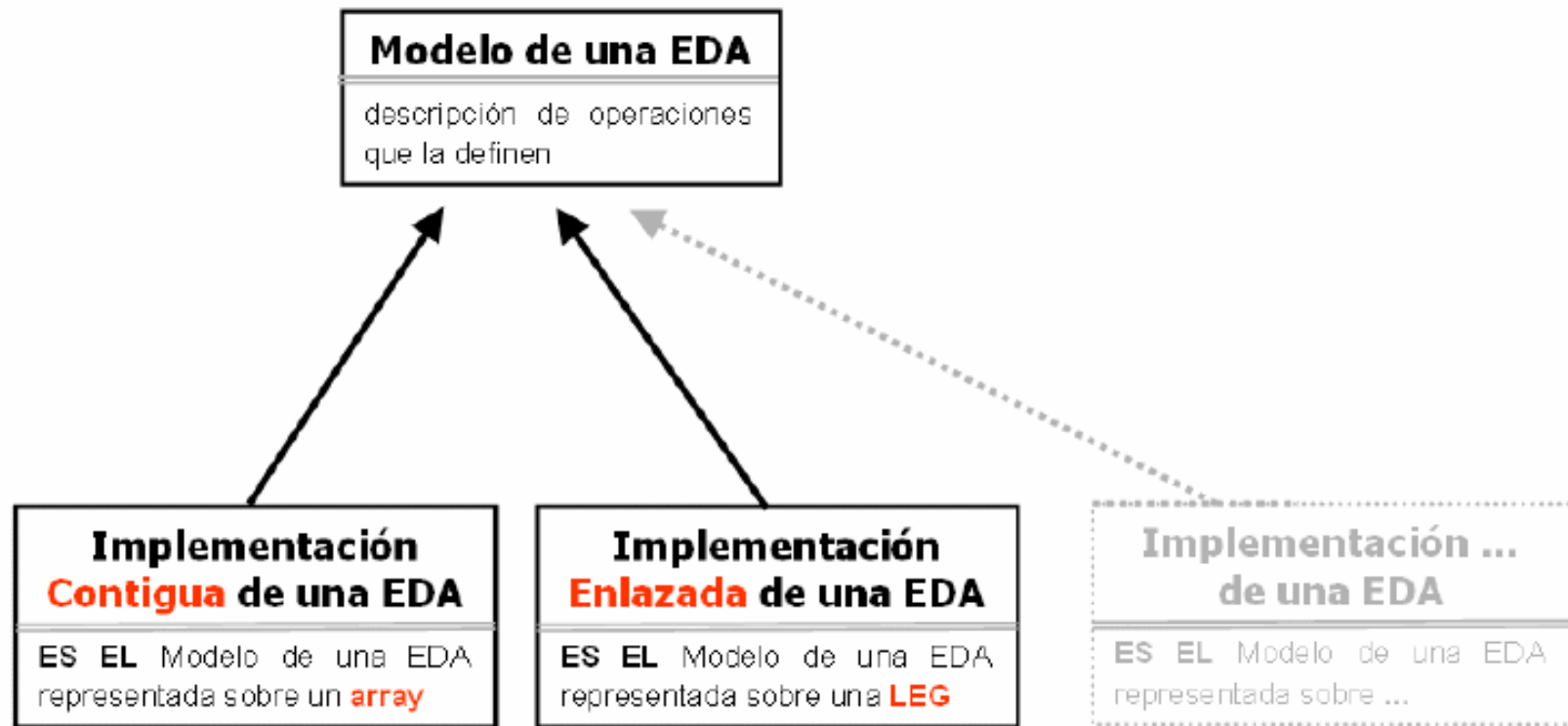
➔ Para describir una EDA resulta imprescindible, a su vez, describir 2 niveles de abstracción o componentes:

- **MODELO**, o Especificación, de una EDA: descripción de las operaciones que definen su funcionalidad, el tipo de gestión de Datos que realiza, con independencia de su posterior representación en memoria
- **IMPLEMENTACIÓN** de una EDA: representación en memoria de los Datos (soporte Contiguo, Enlazado, Mixto) y, en base a ésta, la implementación de las operaciones que define su Modelo

1. Estructuras de Datos

Definición

- De la definición de una EDA se desprende que la **relación** que guardan las distintas Implementaciones de una EDA con su Modelo es **Jerárquica**



1. Estructuras de Datos

Clasificación en función del Modelo

- Cada aplicación exige un Modelo determinado, cuya eficacia vendrá dada por la Implementación más o menos ajustada que se haga de él

- **LINEAL:** **Pila, Cola y Lista**

gestión Secuencial de Datos, i.e. en base al orden en el que se han ido incorporando (LIFO, FIFO, SECUENCIAL)

- **DE BÚSQUEDA:** **Diccionario y Cola de Prioridad**

gestión basada en la Búsqueda Dinámica de un Dato dado (Búsqueda por clave o por prioridad)

- **DE RELACIÓN o GRAFO:**

gestión de Datos que guardan entre sí una Relación Binarias para obtener, por ejemplo, el “Camino Mínimo” que los une

2. Diseño de una EDA en Java

2.1. Jerarquía Java de una EDA

- Al ser jerárquica la relación entre las Implementaciones de una EDA y su Modelo, una EDA se describe en Java mediante una Jerarquía compuesta por:

- Una **interface** Raíz de tipo genérico, que describe su Modelo
- Cada Derivada de la Raíz (vía **implements**) de tipo genérico, que describe una de sus Implementaciones -Contigua, Enlazada, etc.

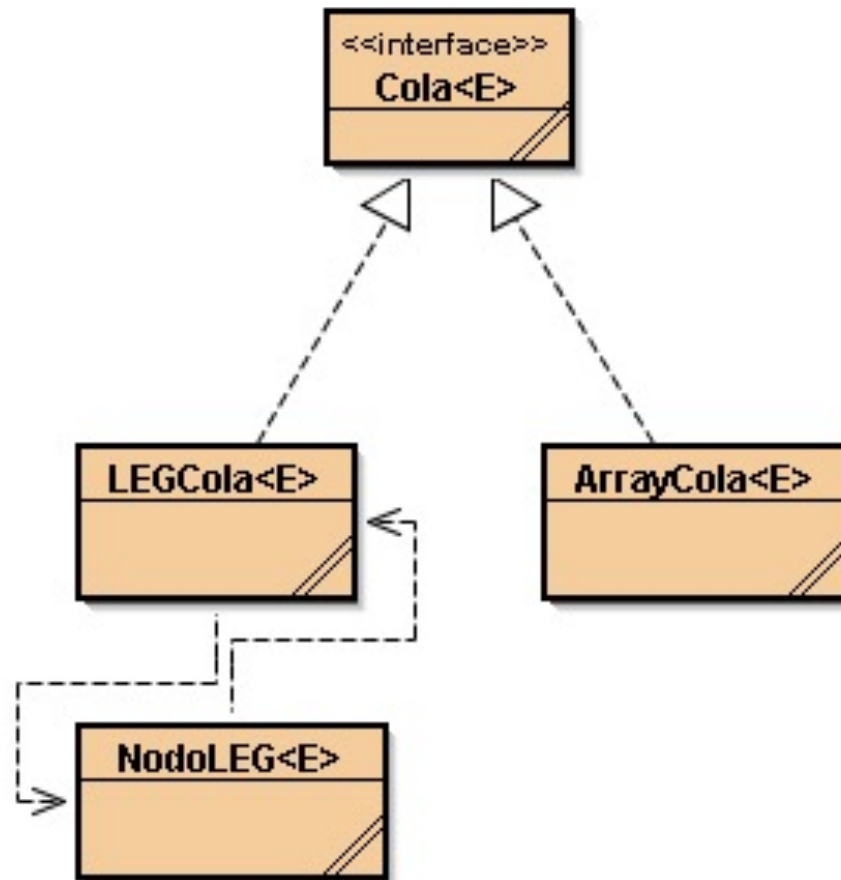
- **Ejemplo:** Jerarquía Java de una Cola

- `public interface Cola<E>{...}` // Modelo Java
- `public class ArrayCola<E>{...}` // Implementación Java Contigua
- `public class LEGCola<E>{...}` // Implementación Java Enlazada

2. Diseño de una EDA en Java

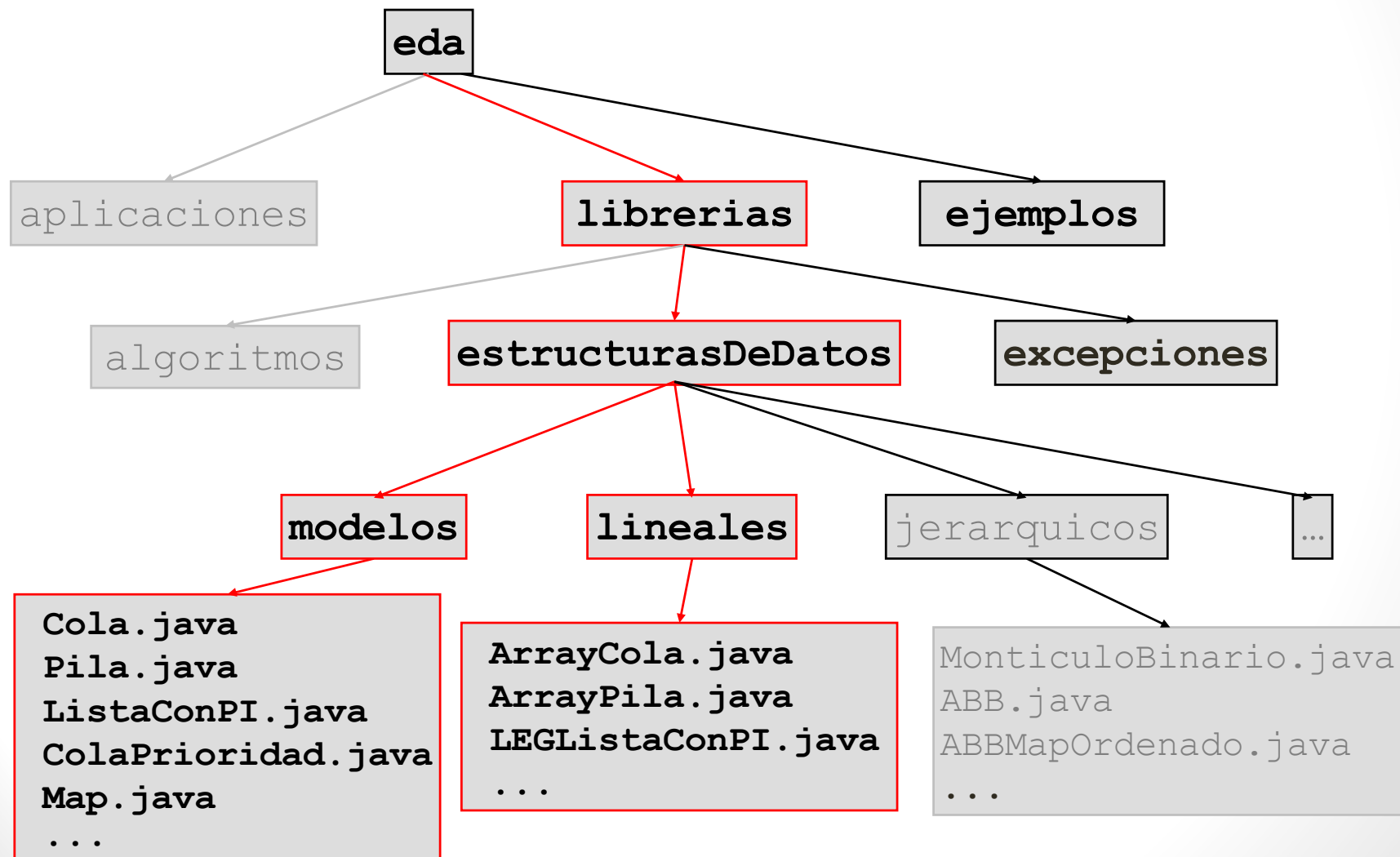
2.1. Jerarquía Java de una EDA

Clases de la jerarquía Java de una *Cola* y relaciones entre ellas



2. Diseño de una EDA en Java

2.2. Organización de jerarquías de EDAs en librerías (packages BlueJ)




2. Diseño de una EDA en Java

2.3. Criterios de diseño *de la Raíz de la jerarquía* -aplicados a Cola

Sabiendo que una Cola es una Colección de Elementos que se gestionan siguiendo un criterio FIFO, i.e. permitiendo la consulta sólo del primero de ellos en orden de inserción ... ¿Cómo se consigue diseñar la siguiente *interface*?

```
package librerias.estructurasDeDatos.modelos;

public interface Cola<E> {
    void encolar(E e); //  $\Theta(1)$ 
    /** SII !esVacia() */ E desencolar(); //  $\Theta(1)$ 
    /** SII !esVacia() */ E primero(); //  $\Theta(1)$ 
    boolean esVacia(); //  $\Theta(1)$ 
}
```

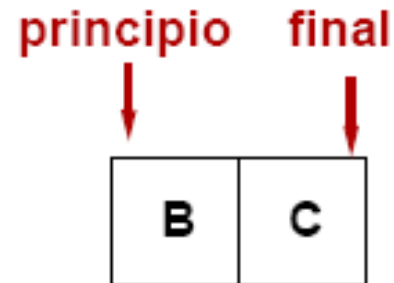
1. Definir el nº mínimo *–kernel–* de métodos abstractos que permiten la gestión FIFO 
 - Revisar los parámetros de los métodos definidos para que respeten la estructura de la EDA –puntos de acceso para inserción, borrado y consulta
 - No sobrescribir los métodos `toString` ni `equals`
2. Evitar en lo posible lanzar Excepciones **checked**, substituyéndolas por Precondiciones
3. Establecer el coste máximo estimado de los métodos definidos



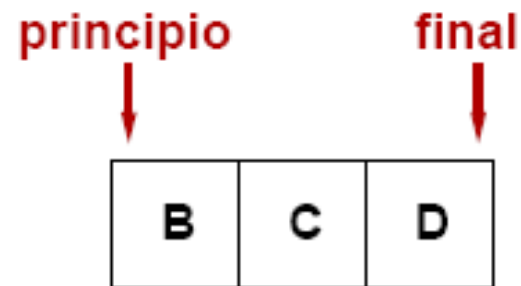
¿Qué operaciones definen el Modelo FIFO?



Cola de tres Datos de tipo `Character`, siendo A el `primero()` de ellos



Al `desencolar()` se elimina A de la **Cola**, por lo que B es ahora el `primero()`



Al `encolar(new Character('D'))` se inserta D al **final** la **Cola**, en la que B sigue siendo el `primero()`



2. Diseño de una EDA en Java

2.3. Criterios de diseño *de una Derivada* de la jerarquía -aplicados a la Implementación **Contigua** (**array**) de una Cola

Si un **ArrayCola<E>** es una clase que implementa la interfaz **Cola<E>**

1. TIENE UN **array** genérico como soporte de datos en memoria
2. Para satisfacer las restricciones de coste de los métodos de la interfaz :
 - Se simula un **array** CIRCULAR
 - TIENE UN **finalC**, **principioC**, **talla**
3. Sobrescribe el método **toString**

```
package librerias.estructurasDeDatos.lineales;  
import librerias.estructurasDeDatos.modelos;  
public class ArrayCola<E> implements Cola<E> {  
    protected E[] elArray; protected static final int C_P_D =...;  
    protected int finalC, principioC, talla;  
  
    public ArrayCola() {...}  
    public String toString() {...} //  $\Theta(x)$   
    public void encolar(E e) {...} //  $\Theta(1)$   
    private void duplicarArrayCola() {...} //  $\Theta(x)$   
    private int incrementar(int indice) {...} //  $\Theta(1)$   
    ...  
}
```

2. Diseño de una EDA en Java

2.3. Criterios de diseño *de una Derivada* de la jerarquía -aplicados a la Implementación **Contigua** (**array**) de una Cola



↑
principioC

↑
finalC

talla=4
elArray.length=6

encolar(24)



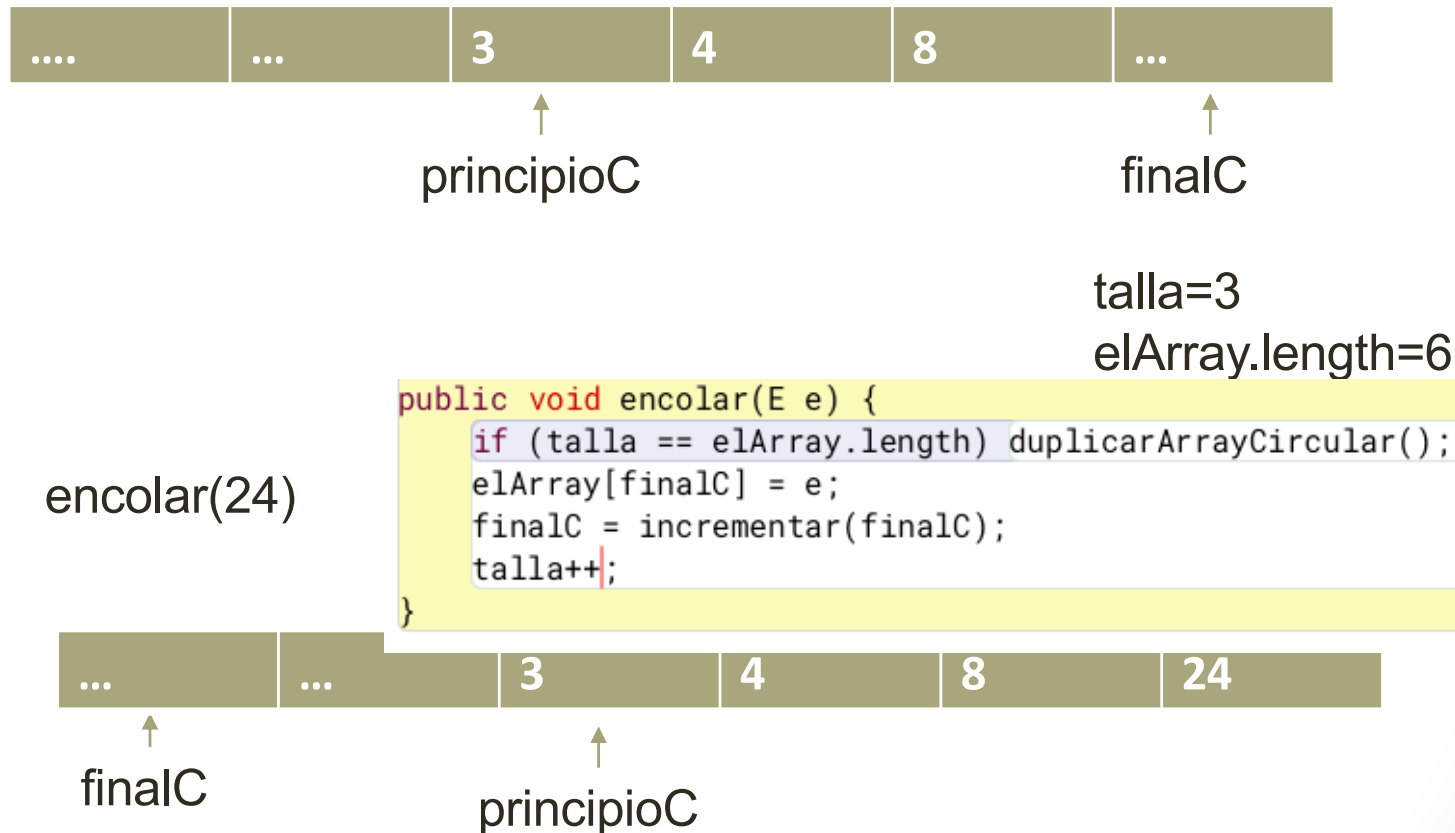
↑
finalC

↑
principioC

(23)

2. Diseño de una EDA en Java

2.3. Criterios de diseño *de una Derivada* de la jerarquía -aplicados a la Implementación **Contigua** (**array**) de una Cola



```
// incrementa un indice de un array circular
```

```
protected int incrementar(int indice) {  
    if (++indice == elArray.length) indice = 0;
```

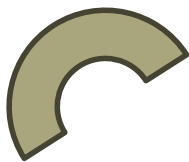
(24)

2. Diseño de una EDA en Java

2.3. Criterios de diseño *de una Derivada* de la jerarquía -aplicados a la Implementación **Contigua** (**array**) de una Cola



```
// incrementa un indice de un array circular
protected int incrementar(int indice) {
    if (++indice == elArray.length) indice = 0;
    return indice;
}
```



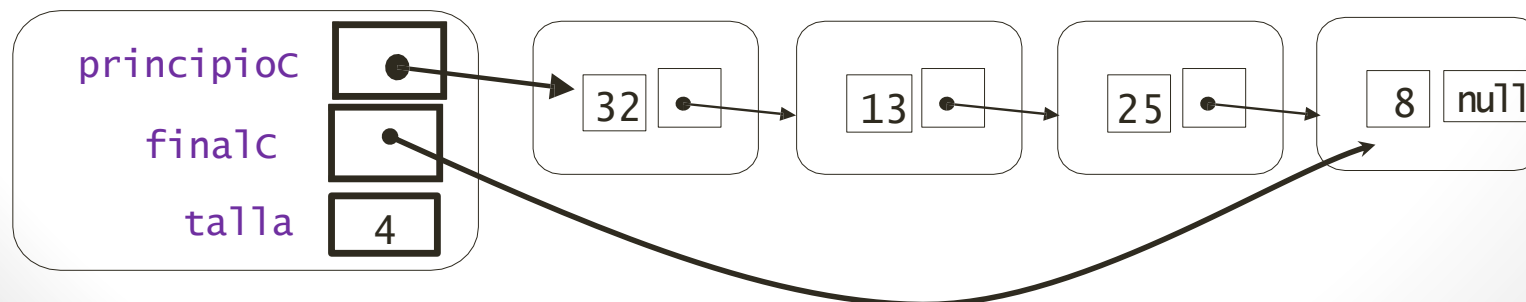
2. Diseño de una EDA en Java

2.3. Criterios de diseño *de una Derivada* de la jerarquía -aplicados a la Implementación *Enlazada* (*LEG*) de una Cola

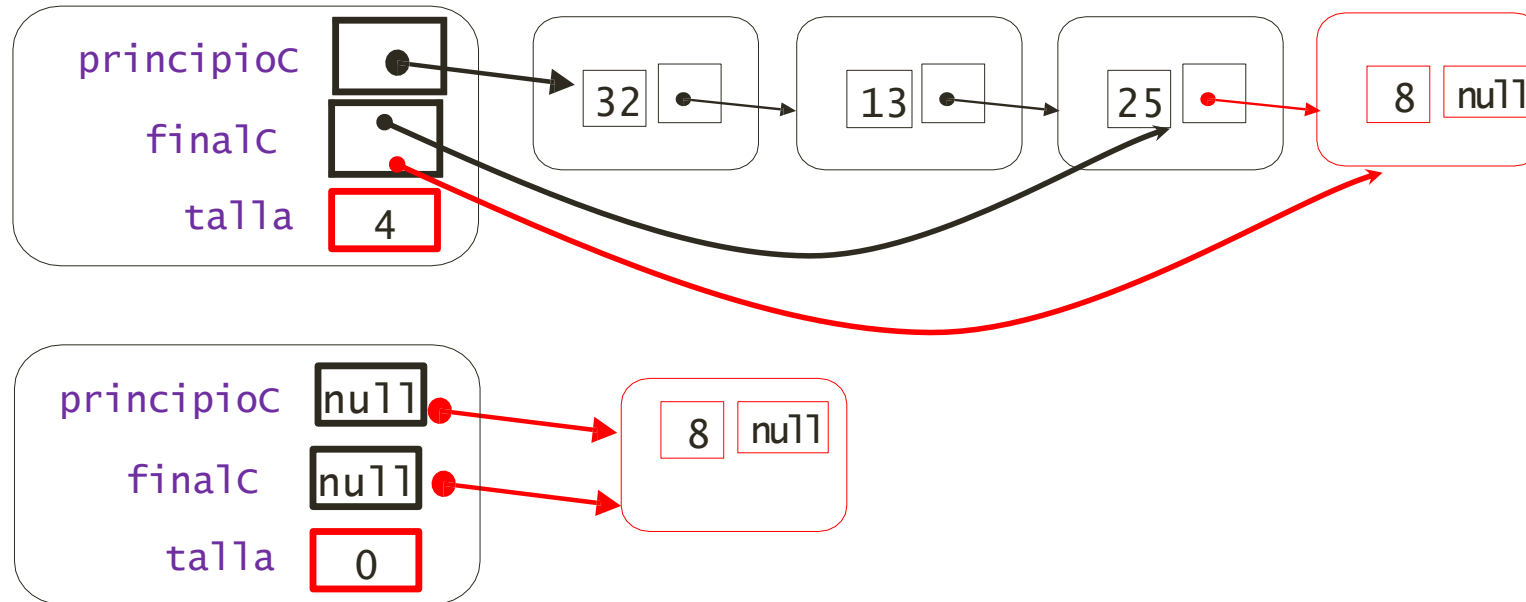
Ejercicio 1.2: Concluir el diseño de la clase `LEGCola<E>` para que implemente la interfaz `Cola<E>`

- La implementación del modelo **Cola** mediante una secuencia de nodos enlazados genéricos (una **LEG**: lista enlazada genérica).

```
public class NodoLEG<E> {  
    protected E dato;  
    protected NodoLEG<E> siguiente;  
    ...  
}
```

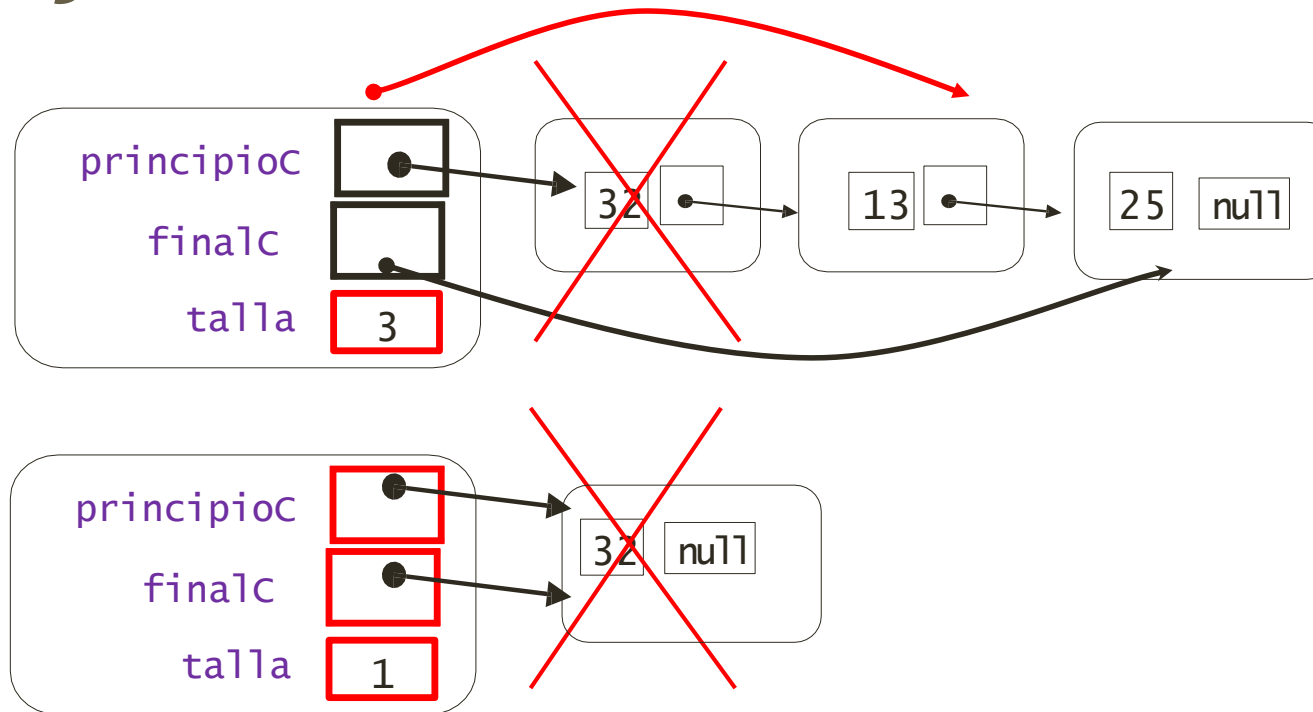


Ejercicio 1.2: encolar



```
public void encolar(E e) {  
    NodoLEG<E> n = new NodoLEG<E>(e);  
    if (finalC != null) finalC.siguiente = n;  
    else principioC = n;  
    finalC = n;  
    talla++;  
}
```

Ejercicio 1.2: desencolar



```
public E desencolar() {  
    E e = principioC.dato;  
    principioC = principioC.siguiente;  
    if (principioC == null) finalC = null;  
    talla--;  
    return e;  
}
```

Ejercicio 1.2

LEGCola: métodos encolar y desencolar

```
public class LEGCola<E> implements Cola<E> {
    ...
    public void encolar(E e) {
        NodoLEG<E> n = new NodoLEG<E>(e);
        if (finalC != null) finalC.siguiente = n;
        else principioC = n;
        finalC = n;
        talla++;
    }
    public E desencolar() {
        E e = principioC.dato;
        principioC = principioC.siguiente;
        if (principioC == null) finalC = null;
        talla--;
        return e;
    }
}
```

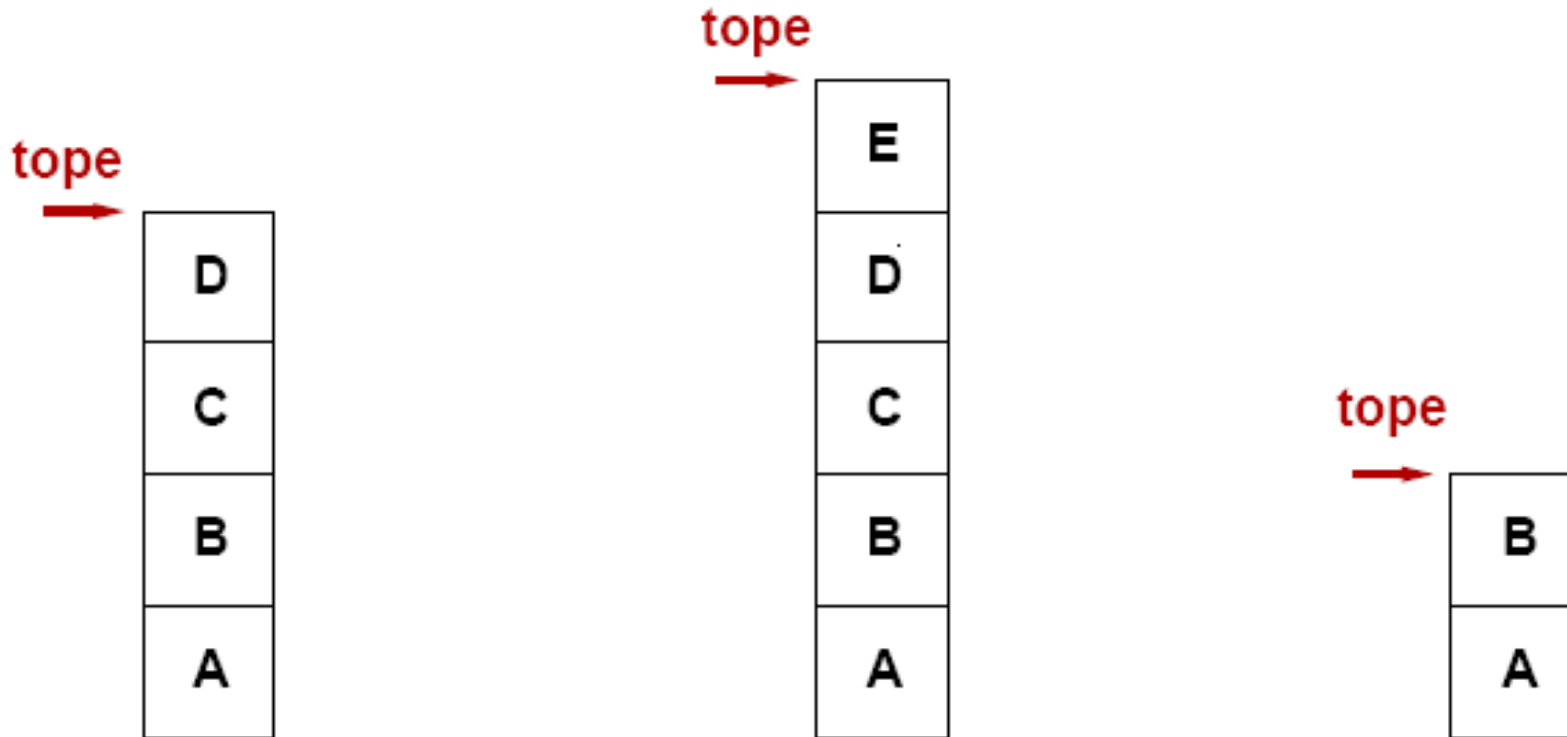
2. Diseño de una EDA en Java

*2.3. Criterios de diseño de las clases de la jerarquía Java de una EDA -aplicados a **Pila***

Ejercicio 1.1: Completar el diseño de la
Implementación Contigua de la interfaz `Pila<E>`



¿Qué operaciones definen el Modelo LIFO?



Pila de 4 Datos de tipo `Character`, cuyo **tope** (`tope()`) ocupa D

Al `apilar(new Character('E'))` se sitúa E en el **tope** de la **Pila**

Al `desapilar()` tres veces se borran de la **Pila**, en este orden, E, D y C, por lo que B ocupa ahora su **tope**



Ejercicio 1.2

ArrayPila

```
public class LEGCola<E> implements Cola<E> {  
    ...  
    public ArrayPila() {  
        elArray = (E[]) new Object[CAPACIDAD_POR_DEFECTO];  
        tope = -1;  
    }  
  
    /** inserta el Elemento e en una Pila, o lo situa en su  
    tope **/  
    public void apilar(E e) {  
        if(tope+1==elArray.length) duplicarArray();  
        tope++;  
        elArray[tope]=e;  
    }  
}
```


2. Diseño de una EDA en Java

*2.3. Criterios de diseño de las clases de la jerarquía Java de una EDA -aplicados a **Pila***

Ejercicio Para Casa: Completar el diseño de la Implementación Enlazada de la interfaz `Pila<E>`

Ejercicio 1.1

LEGPila: métodos apilar y desapilar

```
// en la asignatura EDA
public class LEGPila<E>
implements Pila<E> {
    ...
    public void apilar(E e) {
        tope = new NodoLEG<E>(e, tope);
        talla++;
    }
    public E desapilar() {
        E e = tope.dato;
        tope = tope.siguiente;
        talla--;
        return e;
    }
}
```

Tema 1 – S2

Estructuras de Datos (EDAs), en Java

Contenidos

3. Uso de la jerarquía Java de una EDA
4. EDAs en el estándar de Java: la jerarquía **Collection**

3. Uso de la jerarquía Java de una EDA

Modalidades

Como sucede con cualquier clase Java, la jerarquía de una EDA se puede reutilizar convenientemente instanciada para diseñar nuevas clases ...

1. **Vía `import`**

- Programa ejemplo *TestEDACola*

2. **Vía Composición (TIENE UN)**

- Clase ejemplo *GestorDePacientes*

3. **Vía Herencia (ES UN)**

- Subinterfaz ejemplo *ColaPlus*

4. **Combinando** cualquiera de las posibilidades anteriores

- Clase ejemplo *GestorDePacientesPlus*

¿En qué situaciones y bajo qué condiciones se usan cada una de estas modalidades?

3. Uso de la jerarquía Java de una EDA

3.1 Vía *import*

- Compila el programa **TestEdaCola**, en el paquete **ejemplos** del **tema1**.

Si surge algún error de compilación, comprueba que....

- Has usado la directiva **import** de acceso a **Cola** y **ArrayCola**.
 - Has instanciado convenientemente el tipo de los Elementos de la Cola.
 - NO estás intentando acceder a los atributos de **ArrayCola** de forma incorrecta: aún si tienes acceso al código de la clase, SOLO se puede acceder a sus atributos **protected** vía Herencia y a los **private** vía métodos consultores y modificadores.
 - NO estás intentando usar los métodos de **ArrayCola** de forma incorrecta: aún si son públicos, SOLO se pueden usar los métodos de la **interface Cola** que implementa, entre los que por defecto están los que hereda de **Object**.
- El error persiste... ¿Qué solución se te ocurre para, SIN modificar la interfaz **Cola**, poder compilar y ejecutar **TestEdaCola** sin problemas? Impleméntala.

```

package ejemplos.tema1;

// modificación 1: incluir importaciones
import librerias.estructurasDeDatos.modelos.*;
import librerias.estructurasDeDatos.lineales.*;

public class TestEDACola {
    public static void main(String[] args) {
        // modificación 2: instanciar la cola a números enteros
        Cola<Integer> q = new ArrayCola<Integer>();

        // modificaciones relacionadas con el tamaño
        // declarar una variable local, tallaQ, con la misma funcionalidad
        // ya que, en este ejercicio
        // no se permiten modificaciones en Cola ni en ArrayCola.

        int tallaQ = 0; // modificación 3
        System.out.println("Creada una Cola con " + /*q.talla()*/tallaQ
            + " Integer, q = " + q.toString());
        q.encolar(new Integer(10));
        tallaQ++; // modificación 4
        q.encolar(new Integer(20));
        tallaQ++; // modificación 5
        q.encolar(new Integer(30));
        tallaQ++; // modificación 6
        System.out.println("La Cola de Integer actual es q = " + q.toString());
        System.out.println("Usando otros metodos para mostrar sus Datos el resultado
es ...");
        String datosQ = "";
        while (!q.esVacia()) {
            Integer primero = q.primer();
            if (primero.equals(q.desencolar())) datosQ += primero + " ";
            else datosQ += "ERROR ";
            tallaQ--; // modificación 7
        }
        System.out.println(" el mismo, " + datosQ
            + ", PERO q se vacia, q = " + q.toString());
    }
}

```

3. Uso de la jerarquía Java de una EDA

3.2 Vía Composición (TIENE UN)

- Las reglas señaladas con la ayuda de **TestEDACola** se aplican igualmente al uso de una EDA mediante Composición, pues la única diferencia entre esta modalidad y la anterior es **declarar como un atributo de la clase la variable de la Jerarquía a reutilizar**, en lugar de como variable local del **main** de un programa.

- **Ejemplo:**

Se ha diseñado una aplicación para gestionar la atención diaria a los Pacientes de una consulta: peticiones de cita, lista diaria de Pacientes e historiales en orden de visita, etc.

Modifíquese la clase **GestorDePacientes**, también en **ejemplos** del **tema1**, para que pueda ejecutarse sin problemas con toda su funcionalidad.

3. Uso de la jerarquía Java de una EDA

3.3 Vía Herencia (ES UN) (sigue ...)

- Se emplea en situaciones análogas a las del cálculo de la talla de una *Cola* en TestEDACola:

clases que reutilizan vía `import` o Composición la jerarquía de una EDA, PERO cuyo diseño exige la aplicación de operaciones que no figuran en su Modelo `-interface-` actual

➔ La **mejor solución**, en términos de Reutilización del Software, consiste en **ampliar vía Herencia la Jerarquía Java de la EDA**

- Diseñar una subinterfaz que defina la nueva funcionalidad, el método `talla()` para el caso de Cola
- Diseñar una clase que implemente TAN SOLO a dicha subinterfaz, i.e. una clase que solo implemente la nueva funcionalidad, el método `talla()` para el caso de Cola

3. Uso de la jerarquía Java de una EDA

3.3 Vía Herencia (ES UN) (sigue ...)

Ampliación vía Herencia de la Jerarquía Java Cola

- Diseñar una subinterfaz que defina el método `talla()`

```
package librerias.estructurasDeDatos.modelos;  
public interface ColaPlus<E> extends Cola<E> { int talla(); }
```

- Diseñar una clase que implemente TAN SOLO a dicha subinterfaz, i.e. al método `talla()`

```
package librerias.estructurasDeDatos.lineales;  
import librerias.estructurasDeDatos.modelos;  
public class ArrayColaPlus<E> extends ArrayCola<E>  
    implements ColaPlus<E> {  
    public ArrayColaPlus() { super(); }  
    public talla() { return super.talla; }  
}
```

¿Qué ocurre si NO se tiene acceso a los atributos de la clase?

3. Uso de la jerarquía Java de una EDA

3.3 Vía Herencia (ES UN) (sigue ...)

Ampliación vía Herencia de la Jerarquía Java Cola

```
package librerias.estructurasDeDatos.lineales;  
import librerias.estructurasDeDatos.modelos;  
  
public class ArrayColaPlus<E> extends ArrayCola<E>  
                                implements ColaPlus<E> {  
  
    public ArrayColaPlus() { super(); }  
    public talla() {  
        int res = 0;  
        while (!this.esVacia()) {  
            E primero = this.desencolar();  
            res++;  
        }  
        return res;  
    }  
}
```

PROBLEMA: esta solución suele ser menos eficiente

VENTAJA: esta solución es siempre posible

OTRO PROBLEMA, MÁS GRAVE: el método **talla** es consultor, y **NO** debería modificar **this**

3. Uso de la jerarquía Java de una EDA

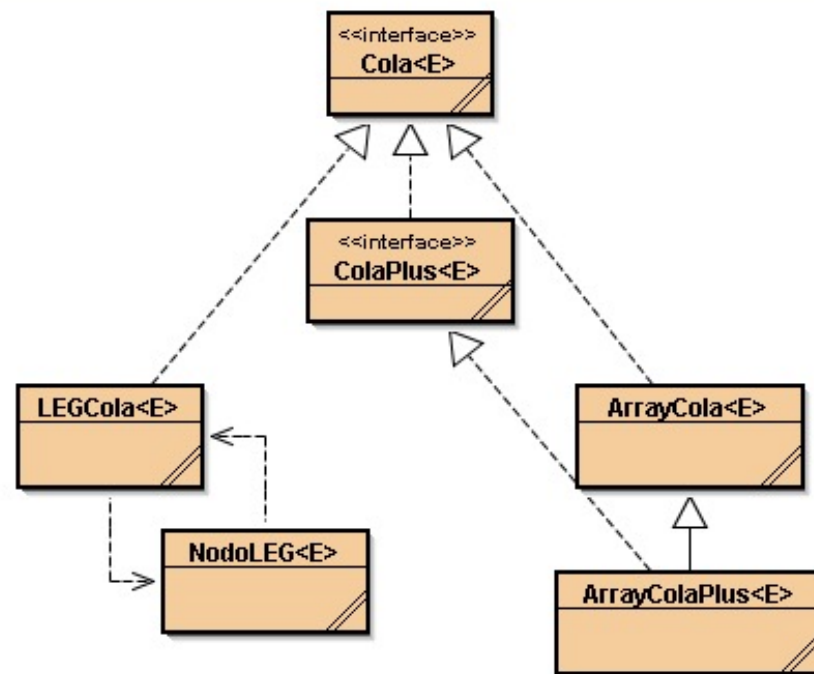
Ejercicio para casa

Ejercicio 2.4: Corregir errores de compilación en la clase TestEdaCola (en ejemplos/tema1), sin modificar la interfaz Cola ni la clase ArrayCola que implementa la interfaz. Para ello utiliza la clase ArrayColaPlus que os proporcionamos

3. Uso de la jerarquía Java de una EDA

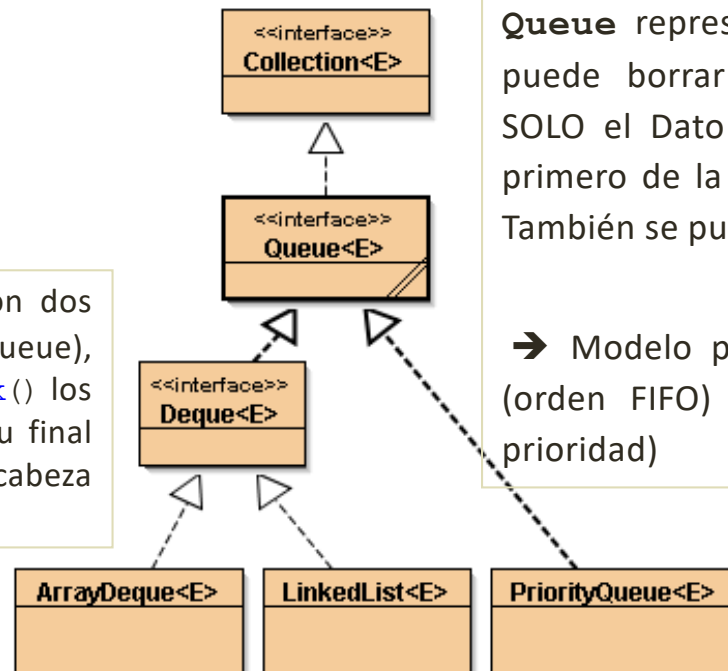
3.3 Vía Herencia (ES UN) (sigue ...)

Ampliación vía Herencia la Jerarquía Java Cola



4. EDAs en el estándar de Java: la jerarquía `Collection`

Deque ES UNA **Queue** con dos extremos (**Double ended queue**), que añade a `poll()` y `peek()` los métodos de inserción en su final (`addLast(e)`) y en su cabeza (`addFirst(e)`)



ArrayDeque y **LinkedList** son Implementaciones eficientes de **Pila** y **Cola**

Queue representa **Collection** en las que se puede borrar (`poll()`) y recuperar (`peek()`) SOLO el Dato que ocupa su cabeza (head), el primero de la Colección según un cierto orden. También se puede añadir a la cola (`add(e)`)

➔ Modelo parcial de **Pila** (orden LIFO), **Cola** (orden FIFO) y **Cola de Prioridad** (orden de prioridad)

PriorityQueue ES UNA **Queue** para Datos **Comparable**, que se insertan según su prioridad –método `add(e)`– para que en su cabeza siempre figure el Dato de máxima prioridad
➔ Implementación eficiente de **Cola de Prioridad**

4. EDAs en el estándar de Java: la jerarquía `Collection`

Ejercicio 3.1: Utilizando la clase de la jerarquía java `Collection` `ArrayDeque` vía herencia, diseña la clase `ArrayDequeCola` que implemente la interfaz `Cola`.

Ejercicio 3.2: Utilizando vía Herencia la clase `ArrayDequeCola` anterior, escribe la clase `ArrayDequeColaPlus` que implemente la interfaz `ColaPlus`.

Ejercicio 3.3 : Utilizando la jerarquía `Deque` escribe un nuevo diseño de la clase `TestEdaCola`, llamado `TestEDAColaVDeque`, equivalente a `TestEDACola`.

[Deque Documentacion](#)

Ejercicio 3.1.

```
package librerias.estructurasDeDatos.lineales;

import librerias.estructurasDeDatos.modelos.*;
import java.util.ArrayDeque;
import java.util.Iterator;

public class ArrayDeQueCola<E> extends ArrayDeque<E> implements Cola<E> {
    protected ArrayDeque elArray;

    public ArrayDeQueCola() {
        elArray = new ArrayDeque();
    }

    public void encolar(E e) {
        elArray.add(e);
    }

    public E desencolar() {
        return (E) elArray.poll();
    }

    /** SII !isEmpty():
     *  ** retira pero no borra el elemento al principio de la cola, en orden
     *  * de inserción **/
    public E primero() { return (E) elArray.peek(); }

    /** comprueba si la cola está vacía **/
    public boolean esVacia() { return elArray.size() == 0; }

    /** obtiene un String con los Elementos de una Cola en orden FIFO,
     *  * u orden de inserción, y en el formato utilizado en el estándar Java.
     *  * Así, por ejemplo, si tienes una Cola con Enteros 1 a 4,
     *  * en orden FIFO, toString devuelve [1, 2, 3, 4];
     *  * si la cola está vacía, devuelve [].*/

    public String toString() {
        // NOTA: Se utiliza la clase StringBuilder en lugar de String,
        // por razones de eficiencia
        StringBuilder res = new StringBuilder();
        res.append("[");

        //Iterator i = elArray.iterator();
```

Ejercicio 3.2

```
package librerias.estructurasDeDatos.lineales;

import librerias.estructurasDeDatos.modelos.*;

public class ArrayDeQueColaPlus<E> extends ArrayDeQueCola<E> implements ColaPlus<E> {

    public int talla() {
        return elArray.size();
    }
}
```


Ejercicio 3.3

```
package ejemplos.tema1;

import librerias.estructurasDeDatos.modelos.*;
import librerias.estructurasDeDatos.lineales.*;

public class TestEDAColaVDeque {
    public static void main(String[] args) {
        ColaPlus<Integer> q = new ArrayDequeColaPlus<Integer>();

        System.out.println("Creada una Cola con " + q.talla()
            + " Integer, q = " + q.toString());
        q.encolar(new Integer(10));
        q.encolar(new Integer(20));
        q.encolar(new Integer(30));
        System.out.println("La Cola de Integer actual es q = " + q.toString());
        System.out.println("Usando otros metodos para mostrar sus Datos el resultado es ...");
        String datosQ = "";
        while (!q.esVacia()) {
            Integer primero = q.primerO();
            if (primero.equals(q.desencolar())) datosQ += primero + " ";
            else datosQ += "ERROR ";
        }
        System.out.println(" el mismo, " + datosQ
            + ", PERO q se vacia, q = " + q.toString());
    }
}
```

Tema 1 – S3

Estructuras de Datos (EDAs), en Java

Contenidos

5. Lista con Iterador: la jerarquía **ListaConPI**

5. Lista con Iterador o Punto de Interés

Motivación : el modelo Lista

- A diferencia de una pila o de una cola, una **lista** es una secuencia en la que el acceso puede realizarse en cualquier posición, pero se requiere previamente desplazarse hasta esa posición.

```
package librerias.estructurasDeDatos.modelos;
public interface Lista<E> {
    /** SII 0<=i<=talla() */ void insertar(E e, int i); //  $\Omega(1)$ ,  $O(n)$ 
    /** SII 0<=i<=talla() */ void eliminar(int i);      //  $\Omega(1)$ ,  $O(n)$ 
    /** SII 0<=i<=talla() */ E recuperar(int i);       //  $\Omega(1)$ ,  $O(n)$ 
    boolean esVacia();                                //  $\Theta(1)$ 
    int talla();                                       //  $\Theta(1)$ 
}
```

- El argumento **i** en los métodos de la interfaz **Lista** indica la posición de la lista donde se quiere insertar, eliminar o recuperar un dato, respectivamente.
- Los 3 métodos tienen como precondition que el valor de **i** sea una posición válida de la lista.
- Los 3 métodos se pueden implementar con un coste **$\Omega(1)$, $O(n)$** usando una representación enlazada en memoria.

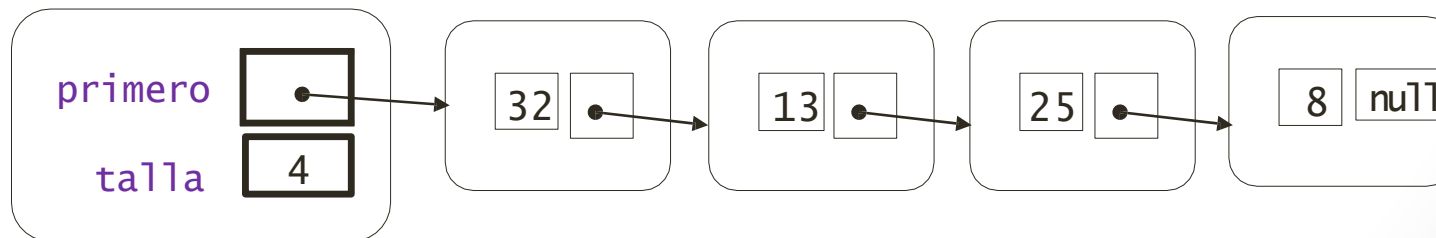
5. Lista con Iterador o Punto de Interés

Motivación: el modelo Lista

- La implementación del modelo **Lista** mediante una secuencia de nodos enlazados genéricos (una **LEG**: lista enlazada genérica).

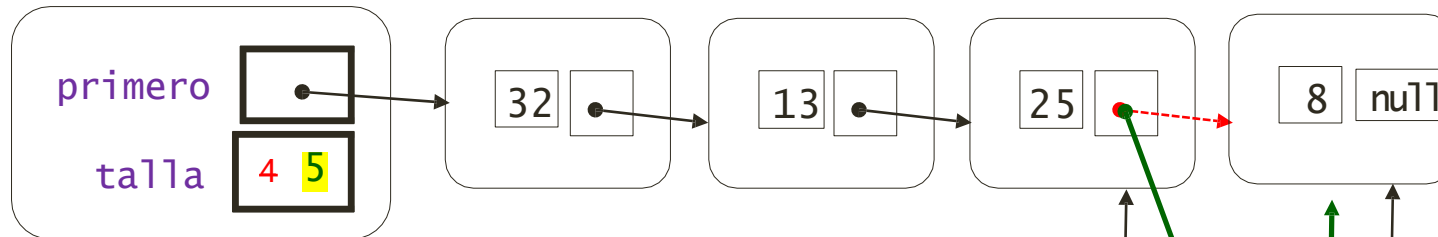
```
public class NodoLEG<E> {  
    protected E dato;  
    protected NodoLEG<E> siguiente;  
    ...  
}
```

```
public class LEGLista<E> implements Lista<E> {  
    protected NodoLEG<E> primero;  
    protected int talla;  
    public LEGLista() { primero = null; talla = 0; }  
    ...  
}
```



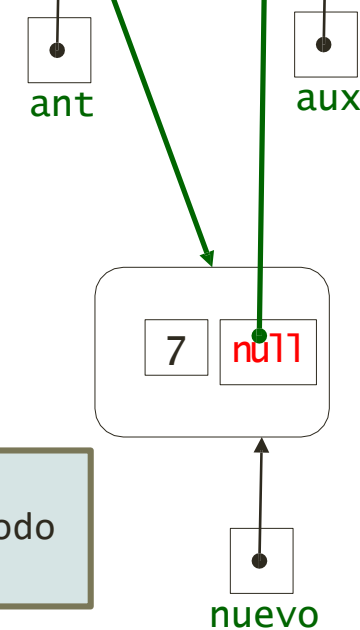
5. Lista con Iterador o Punto de Interés

Motivación: el modelo Lista



```
public class LEGLista<E> implements Lista<E> {  
    ...  
    public void insertar(E e, int i) {  
        NodoLEG<E> nuevo = new NodoLEG<E>(e);  
        talla++;  
        NodoLEG<E> aux = primero;  
        NodoLEG<E> ant = null;  
        for (int j = 0; j < i; j++) {  
            ant = aux;  
            aux = aux.siguiente;  
        }  
        nuevo.siguiente = aux;  
        if (ant == null) primero = nuevo;  
        else ant.siguiente = nuevo;  
    }  
    ...  
}
```

bucle for,
coste del método
 $\Omega(1)$, $O(n)$



5. Lista con Iterador o Punto de Interés

Motivación: los modelos Lista y ListaConPI

- Una **lista** es una secuencia en la que el acceso puede realizarse en cualquier posición, pero se requiere previamente desplazarse hasta esa posición (coste lineal en el caso peor).
- Una **lista con punto de interés** (PI) es una lista en la que se supone que hay una posición actual, llamada punto de interés o **cursor**.
- Si la lista tiene n datos, $n \geq 0$, numerados de 0 a $n-1$, el punto de interés puede estar:
 - en una posición $0 \leq i \leq n-1$: *sobre el elemento i -ésimo*
 - en la posición $i = n$: *después del último elemento*
- En una **lista con PI**, las operaciones de insertar un dato, borrar un dato, etc., se refieren al punto de interés. Coste exactamente constante, sin instancias significativas.
- En el modelo de lista con PI que se presenta, el PI solo se puede mover hacia la derecha (hacia adelante), o situarse al inicio o al final.

5. Lista con Iterador o Punto de Interés

Punto de Interés de una Lista

○ **Idea básica:** En cada instante de la gestión secuencial solo es posible acceder a un Dato de la [Lista](#), el que ocupa en dicho instante el **Punto de Interés (PI)**

➔ El acceso al Dato que ocupa el PI se puede realizar en tiempo constante

➔ La posición que ocupa un Dato en la Lista deja de ser un parámetro en las operaciones de acceso secuencial, pues ...

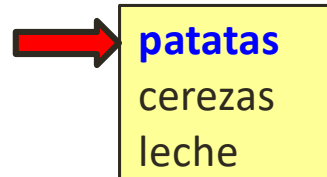
- Al **inicio** de la gestión secuencial el PI solo puede estar situado sobre el **primero** de los Datos de la *Lista*
- Para acceder al **siguiente** Dato de una *Lista* durante su gestión secuencial basta desplazar el PI sobre él ➔ Al **insertar** o **recuperar** un Dato de una Lista el PI permanecerá **inamovible**; al **eliminar** un Dato de una Lista el PI se situará sobre el dato siguiente (si lo hubiera)
- Si una Lista **está vacía**, o bien el acceso secuencial ha llegado a su **fin**, no hay Dato alguno que ocupe el PI ➔ Intentar **recuperar** un Dato, **eliminarlo** o pasar al **siguiente** provocará una excepción

5. Lista con Iterador o Punto de Interés

PI de una Lista. Ejemplo de búsqueda e inserción al final de una ListaConPI (TestListaConPIDeLaCompra)

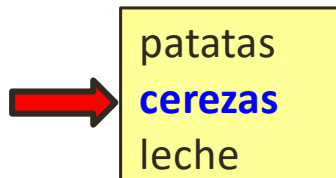
- **Inicialización de la búsqueda:** instrucciones y estado de la lista tras ejecutarla (la flecha roja indica el PI)

```
esta = false  
l.inicio();
```

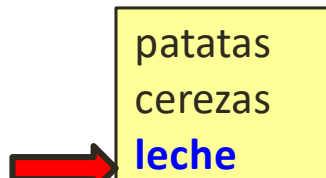


- **Búsqueda:** iteraciones y estado de la lista tras cada una de ellas

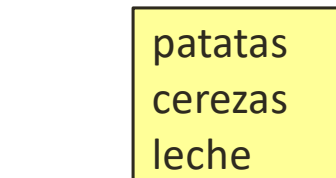
```
// patatas ≠ perejil  
l.siguiente();
```



```
// cerezas ≠ perejil  
l.siguiente();
```

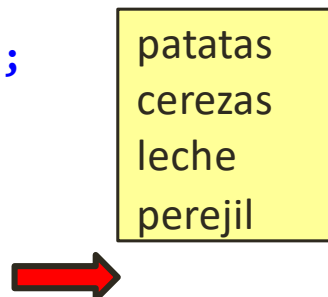


```
// leche ≠ perejil  
l.siguiente();
```



- **Resolución de la búsqueda:** instrucción y estado de la lista tras ejecutarla

```
l.insertar(e);
```



5. Lista con Iterador o Punto de Interés

PI de una Lista. Ejemplo de inserción al inicio de una ListaConPI

- **Terminación de la búsqueda:** estado de la lista tras la última iteración

patatas
cerezas
leche



// l.esFin() == true && esta == false

- **Resolución de la búsqueda:** instrucciones y estado de la lista tras ejecutarlas

l.inicio();



patatas
cerezas
leche

l.insertar(e);





perejil
patatas
cerezas
leche

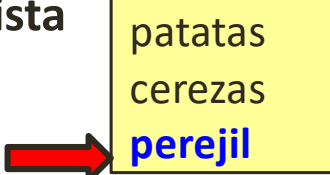
5. Lista con Iterador o Punto de Interés

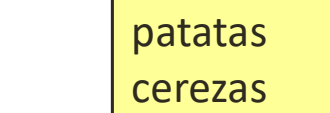
PI de una Lista. Ejemplos de borrado en una ListaConPI

Eliminar el último de la lista

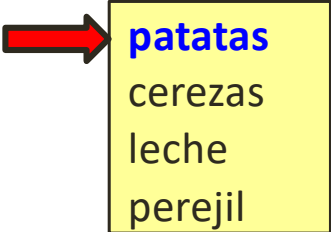

`// l.esFin() == false
l.eliminar();`

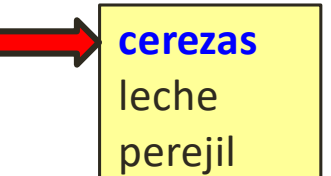

`// l.esFin() == true &&
l.esVacia() == true`


`// l.esFin() == false
l.eliminar();`

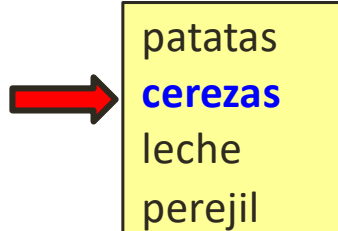

`// l.esFin() == true &&
l.esVacia() == false`

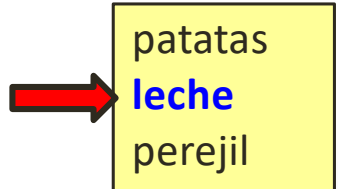
Eliminar el primero de la lista


`l.eliminar();`



Eliminar en otras posiciones


`l.eliminar();`



5. Lista con Iterador o Punto de Interés

*Diseño de la Raíz de la jerarquía **ListaConPI***

Una **Lista Con PI** es una Colección de Datos que se gestionan **secuencialmente** (i.e. uno tras otro desde el primero al último en orden de inserción). El Dato de la Lista que resulta accesible en cada momento de esta gestión es entonces el que ocupa su Punto de Interés.

```
package librerias.estructurasDeDatos.modelos;

public interface ListaConPI<E> {           // Todas las operaciones con coste  $\Theta(1)$ 

    void insertar(E e); // inserta e en una Lista, antes del Elemento en PI,
                        // que permanece inalterado

    void eliminar();    // SII !esFin(): elimina de una Lista el Elemento en PI,
                        // que permanece inalterado

    E recuperar();      // SII !esFin(): obtiene el Elemento en PI de una Lista

    void inicio();      // sitúa el PI de una Lista en su inicio

    void siguiente();   // SII !esFin(): avanza el PI de una Lista

    boolean esFin();    // comprueba si el PI de una Lista está en su fin

    boolean esVacía();  // comprueba si una Lista Con PI está vacía

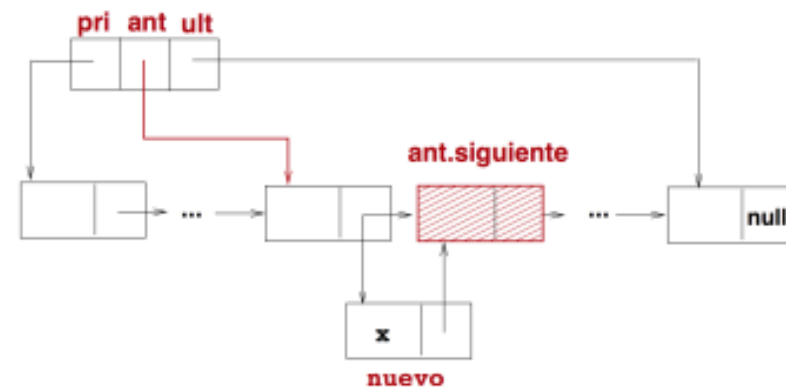
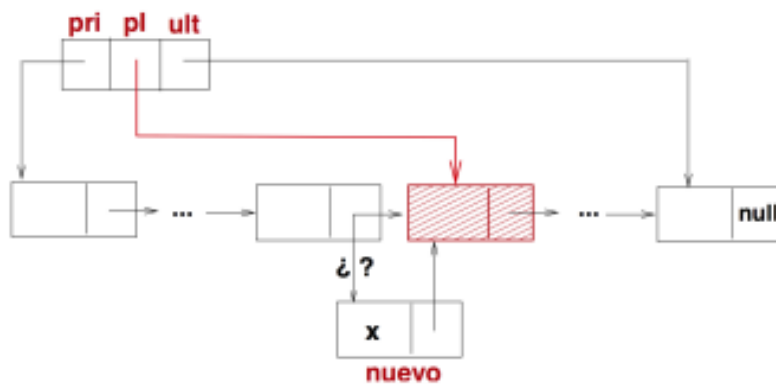
    void fin();         // sitúa el PI de una Lista en su fin

    int talla();        // devuelve la talla de una Lista Con PI
}
```

5. Lista con Iterador o Punto de Interés

*Implementación enlazada del modelo **ListaConPI***

- Como insertar en tiempo constante?



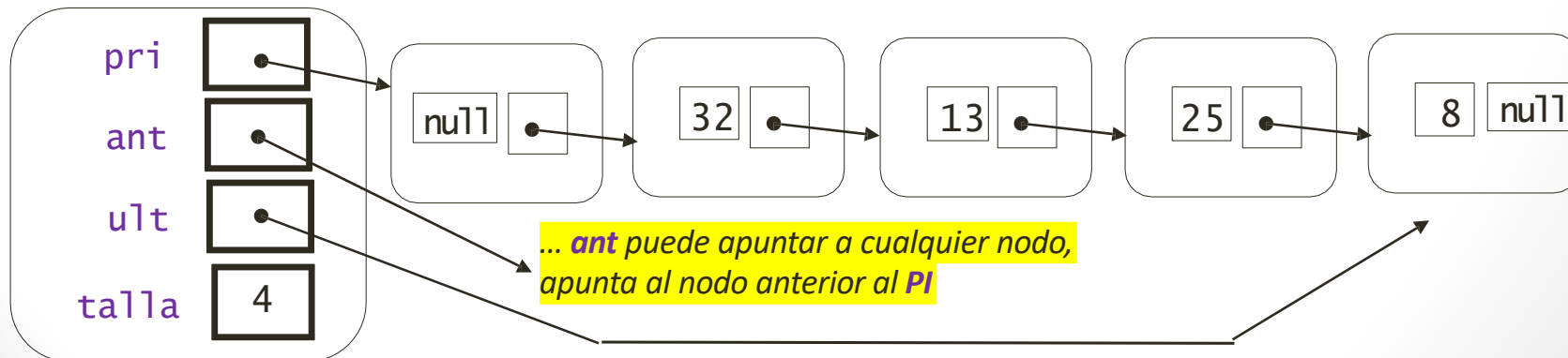
ant no esta definido en inicio de Lista!

5. Lista con Iterador o Punto de Interés

*Implementación enlazada del modelo **ListaConPI***

- La implementación del modelo **ListaConPI** mediante una secuencia de nodos enlazados genéricos (una **LEG**: lista enlazada genérica).

```
public class LEGListaConPI<E> implements ListaConPI<E> {  
    // referencias al nodo primero, al anterior del PI, y al último  
    protected NodoLEG<E> pri, ant, ult;  
    protected int talla;  
    public LEGListaConPI() {  
        // crea lista vacía, con un nodo cabecera ficticio  
        pri = ult = ant = new NodoLEG<E>(null);  
        talla = 0;  
    }  
    ...  
}
```



5. Lista con Iterador o Punto de Interés

*Implementación enlazada del modelo **ListaConPI***

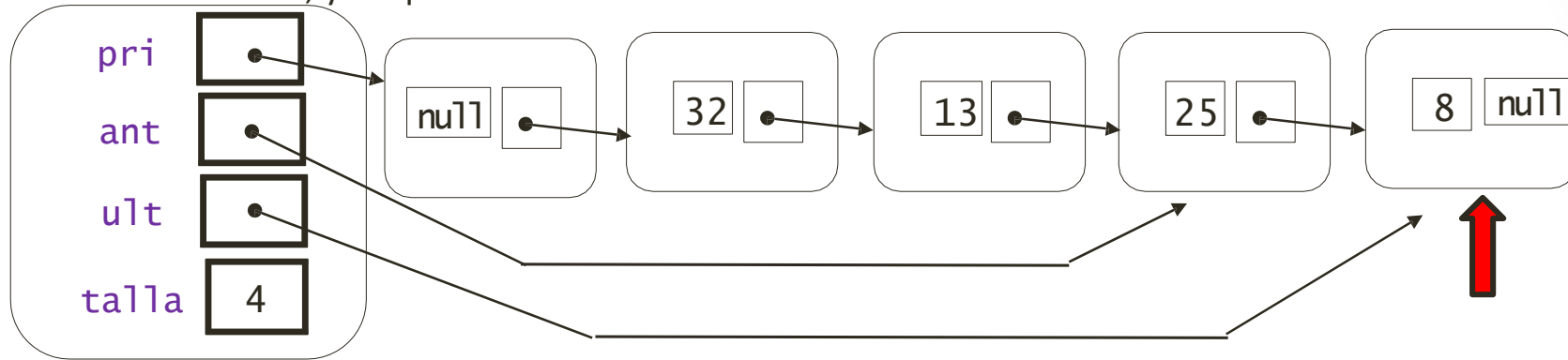
- La implementación del modelo **ListaConPI** mediante una secuencia de nodos enlazados genéricos (una **LEG**: lista enlazada genérica).

```
public class LEGListaConPI<E> implements ListaConPI<E> {  
    ...  
    public void insertar(E e) {  
        NodoLEG<E> nuevo = new NodoLEG<E>(e, ant.siguiente);  
        ant.siguiente = nuevo;  
        if (nuevo.siguiente == null) ult = nuevo;  
        ant = ant.siguiente;  
        talla++;  
    }  
    ...  
}
```

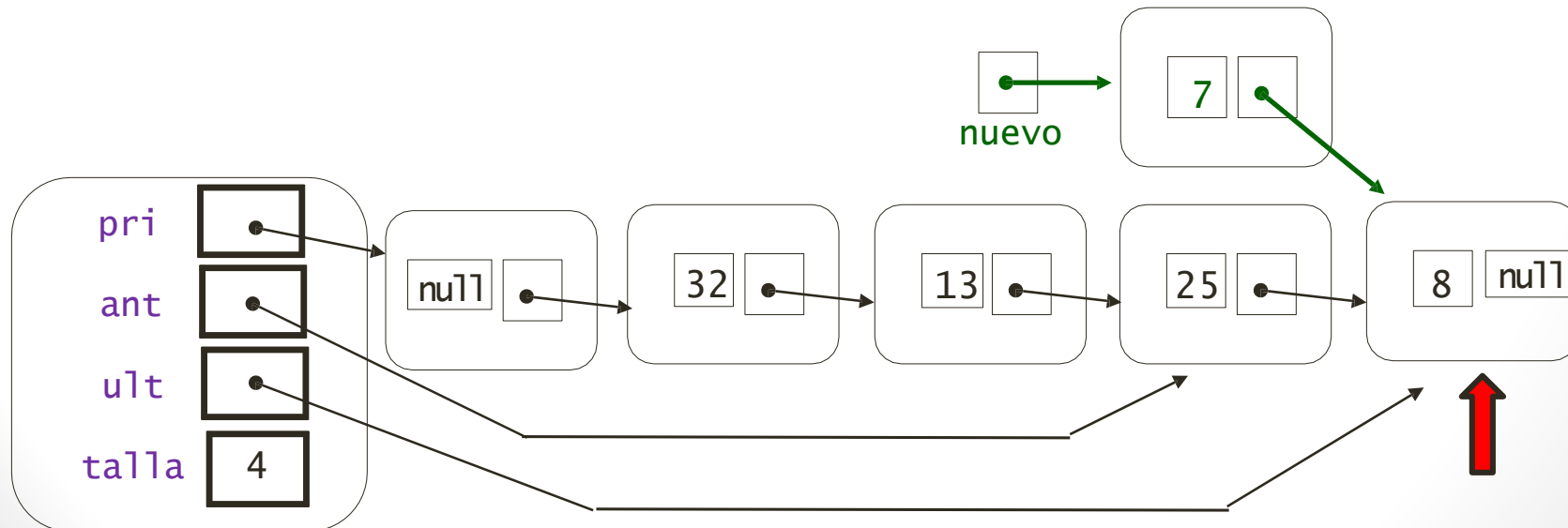
5. Lista con Iterador o Punto de Interés

*Implementación enlazada del modelo **ListaConPI***

- **Ejemplo de inserción paso a paso.** Secuencia inicial {32,13,25,8}, PI sobre último elemento, y se quiere insertar elemento 7:



```
NodoLEG<E> nuevo = new NodoLEG<E>(e, ant.siguiete);
```

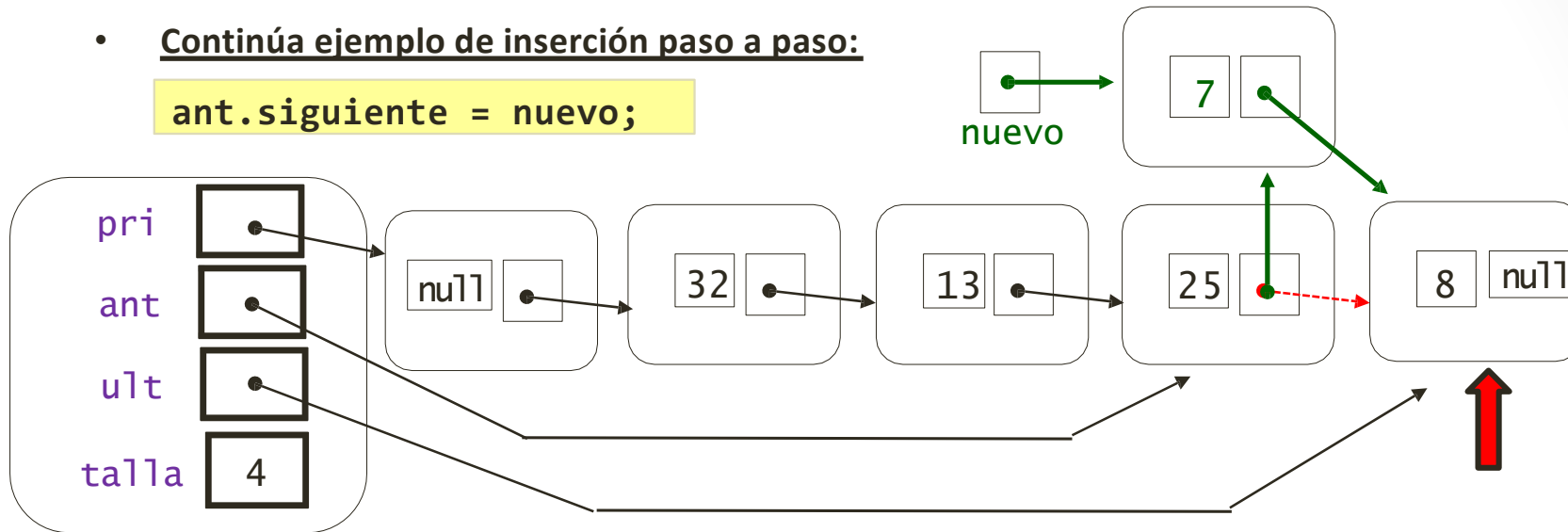


5. Lista con Iterador o Punto de Interés

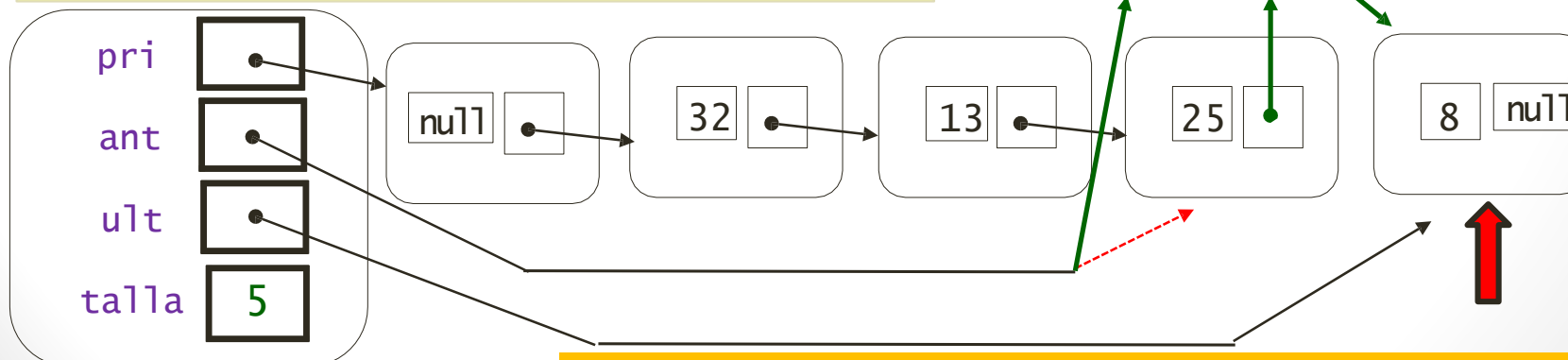
*Implementación enlazada del modelo **ListaConPI***

- Continúa ejemplo de inserción paso a paso:

```
ant.siguiente = nuevo;
```



```
if (nuevo.siguiente == null) ult = nuevo;  
ant = ant.siguiente;  
talla++;
```

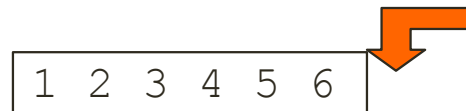


Secuencia final {32,13,25,7,8} y PI sigue sobre último elemento.

5. Lista con Iterador o Punto de Interés

*Uso de la jerarquía **ListaConPI***

Cuestión 1: ¿Qué secuencia de instrucciones Java permite obtener la siguiente **ListaConPI 1** a partir de su creación (vacía)?



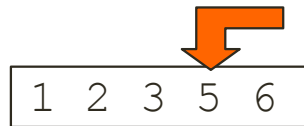
Cuestión 2: Dada la anterior **ListaConPI 1**,

- ¿Qué sucede al ejecutar **1.eliminar()**?
- ¿Qué resultado da **1.esFin()**?
- ¿Qué sucede al ejecutar **1.fin()**?
- ¿Qué sucede al ejecutar **1.inicio()**?

5. Lista con Iterador o Punto de Interés

*Uso de la jerarquía **ListaConPI***

Cuestión 3: Sea la siguiente **ListaConPI** 1,



- Desde su inicio, ¿cómo se ha situado el PI sobre el 5?

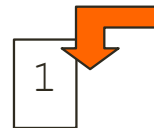
- La secuencia de instrucciones

`1.insertar(new Integer(4)); 1.eliminar();`

¿es equivalente a la secuencia

`1.eliminar(); 1.insertar(new Integer(4)); ?`

¿Y si la Lista es la siguiente?



[42]

5. Lista con Iterador o Punto de Interés

Ejercicio 4.1: Modifica la subinterfaz **ListaConPIPlus** que extienda la funcionalidad de EDA **ListaConPI** vía herencia con los siguientes métodos:

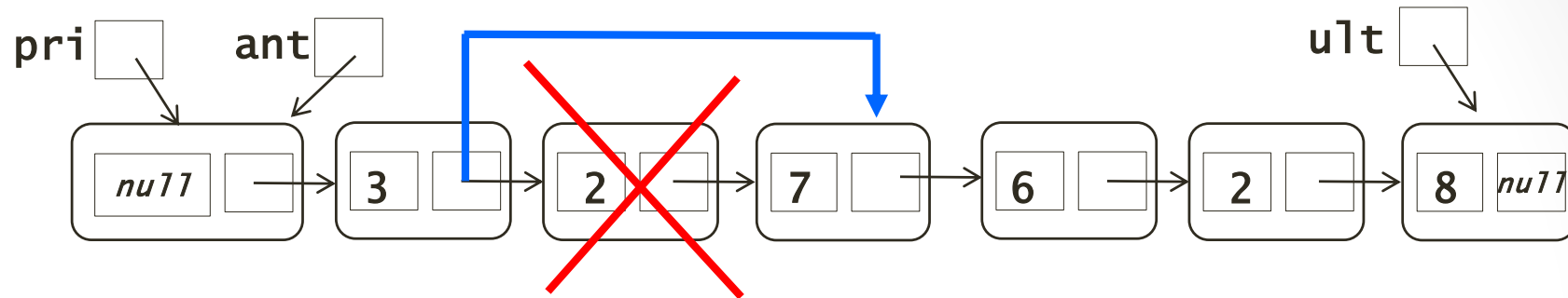
- ☐ boolean **contiene**(E e): devuelve True/False si la lista incluye o no el elemento
- ☐ boolean **eliminarPrimero**(E e): elimina el primer elemento e de la lista
- ☐ boolean **eliminarUltimo**(E e): elimina el último elemento e de la lista
- ☐ boolean **eliminarTodos**(E e): elimina todos los elementos iguales a e de la lista
- ☐ método void **concatenar**(ListaConPI<E>): une una nueva lista a esta lista
- ☐ void **vaciar**(): borra la lista (elimina todos los elementos)
- ☐ void **buscar**(E e): coloca el PDI en e. Si no se encuentra e, el PDI se colocará al final de la lista
- ☐ void **invertir**(): invierte el orden de los elementos de la lista

Ejercicio 4.2: Crea la clase **LEGListaConPIPlus**, que será una implementación enlazada del (sub)modelo **ListaConPIPlus**. Utiliza únicamente los métodos existentes en el modelo **ListaConPI** para implementar los métodos.

Ejercicio 4.3: Crea un método en **LEGListaConPIPlus** que desplace todos los elementos de la lista una posición hacia la izquierda, de forma que el primer elemento pase a ser el último.

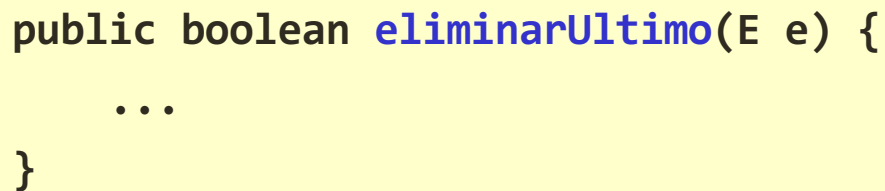
Ejercicio 4.2

ListaConPIPlus: eliminar primera aparición



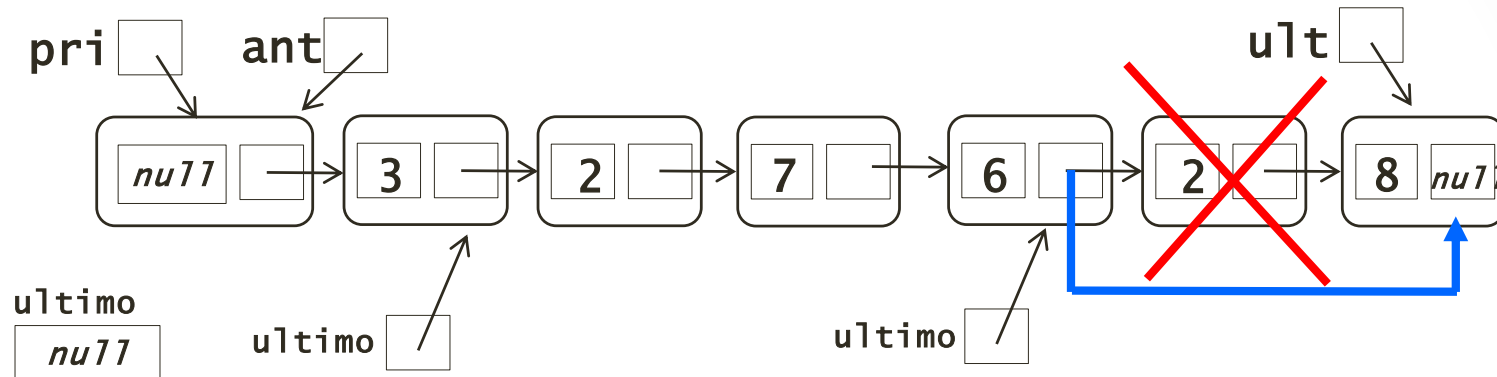
```
public boolean eliminarPrimero(E e) {  
    inicio();  
    while (!esFin()) {  
        if (recuperar().equals(e)) {  
            eliminar();  
            return true;  
        }  
        siguiente();  
    }  
    return false;  
}
```

ListaConPIPlus: eliminar última aparición



Ejercicio 4.2

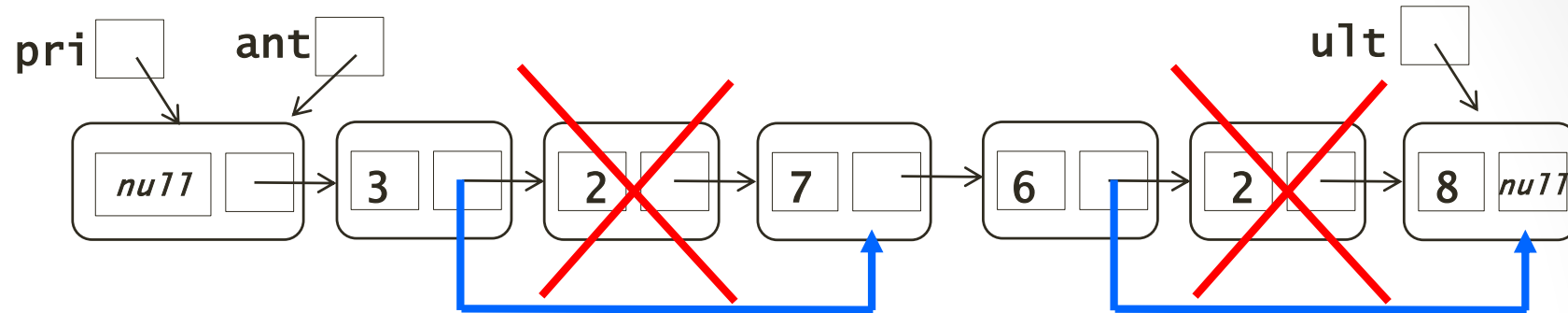
ListaConPIPlus: eliminar última aparición



```
public boolean eliminarUltimo(E e) {  
    NodoLEG<E> ultimo = null;  
    inicio();  
    while (!esFin()) {  
        if (recuperar().equals(e)) ultimo = ant;  
        siguiente();  
    }  
    if (ultimo == null) return false;  
    else {  
        ant = ultimo; eliminar(); return true;  
    }  
}
```

Ejercicio 4.2

ListaConPIPlus: eliminar todas las apariciones



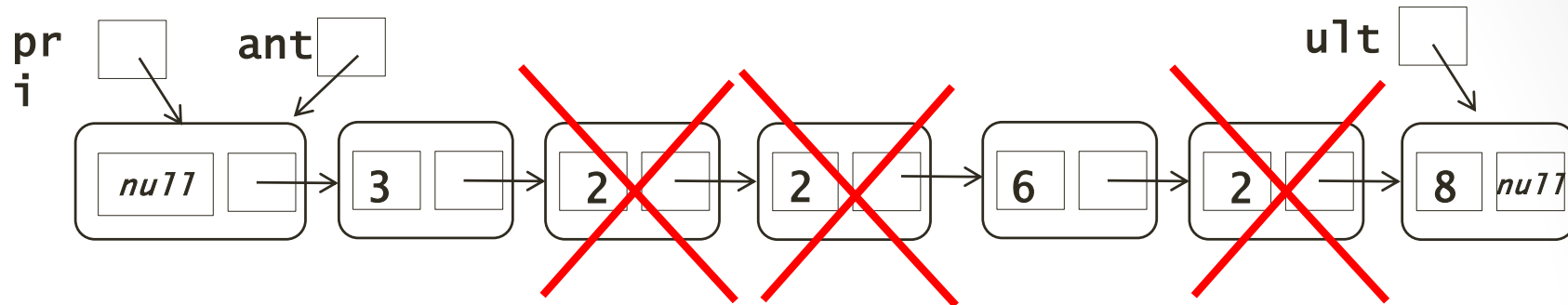
```
public boolean eliminarPrimero(E e) {  
    inicio();  
    while (!esFin()) {  
        if (recuperar().equals(e)) {  
            eliminar();  
            return true;  
        }  
        siguiente();  
    }  
    return false;  
}
```

```
public boolean eliminarTodos(E e) {  
    inicio();  
    boolean b = false;  
    while (!esFin()) {  
        if (recuperar().equals(e)) {  
            eliminar();  
            b = true;  
        }  
        siguiente();  
    }  
    return b;  
}
```

¿¿ Esto es
correcto ??

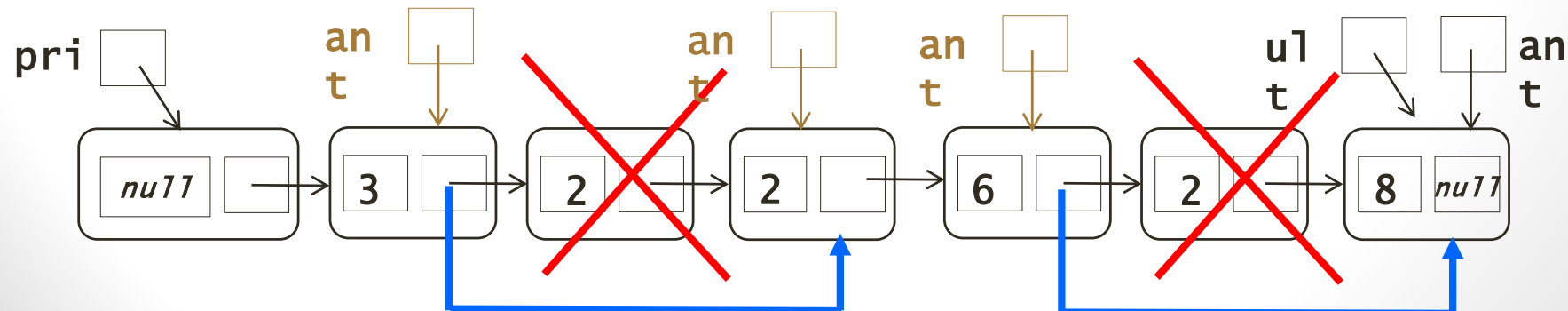
Ejercicio 4.2

ListaConPIPlus: eliminar todas las apariciones



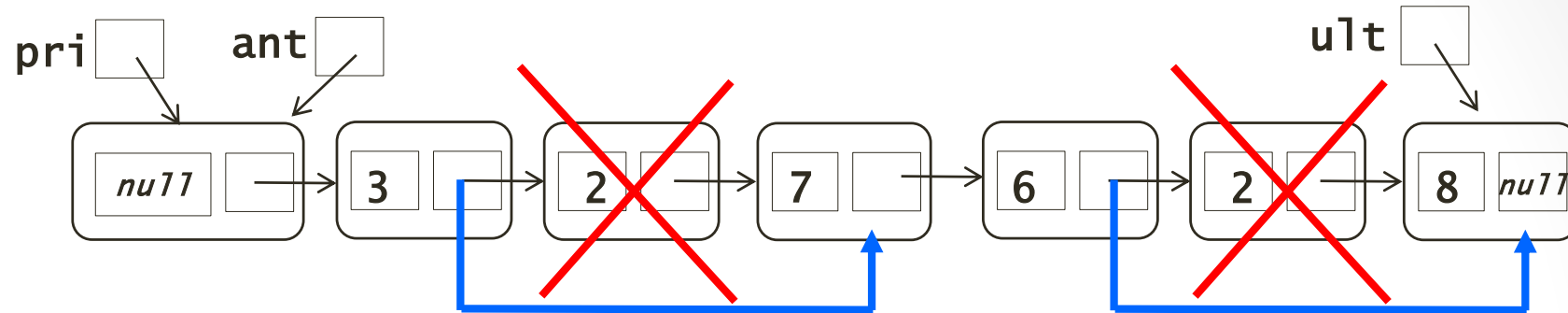
```
while (!esFin()) {  
    if (recuperar().equals(e)) {  
        eliminar(); b = true;  
    }  
    siguiente();  
}
```

Esto es
incorrecto



Ejercicio 4.2

ListaConPIPlus: eliminar todas las apariciones



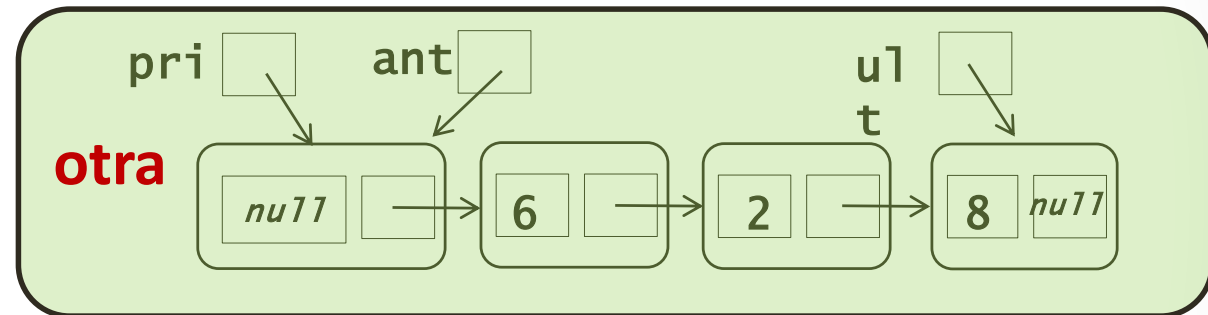
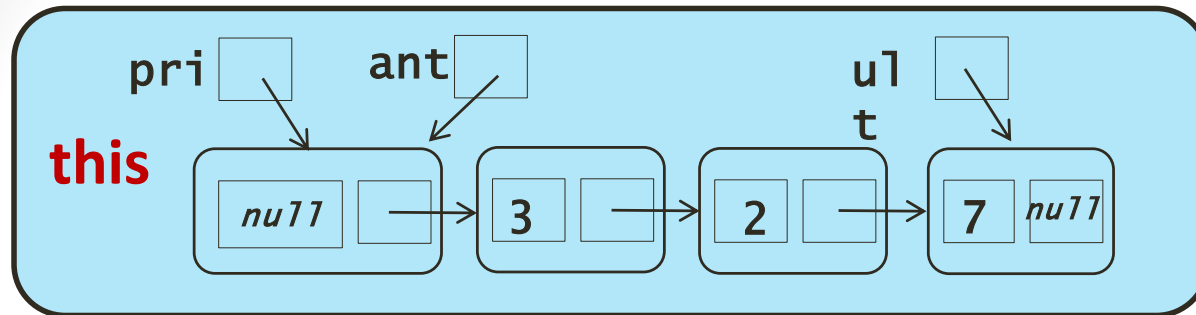
```
public boolean eliminarPrimero(E e) {  
    inicio();  
    while (!esFin()) {  
        if (recuperar().equals(e)) {  
            eliminar();  
            return true;  
        }  
        siguiente();  
    }  
    return false;  
}
```

```
public boolean eliminarTodos(E e) {  
    inicio();  
    boolean b = false;  
    while (!esFin()) {  
        if (recuperar().equals(e)) {  
            eliminar();  
            b = true;  
        }  
        else siguiente();  
    }  
    return b;  
}
```

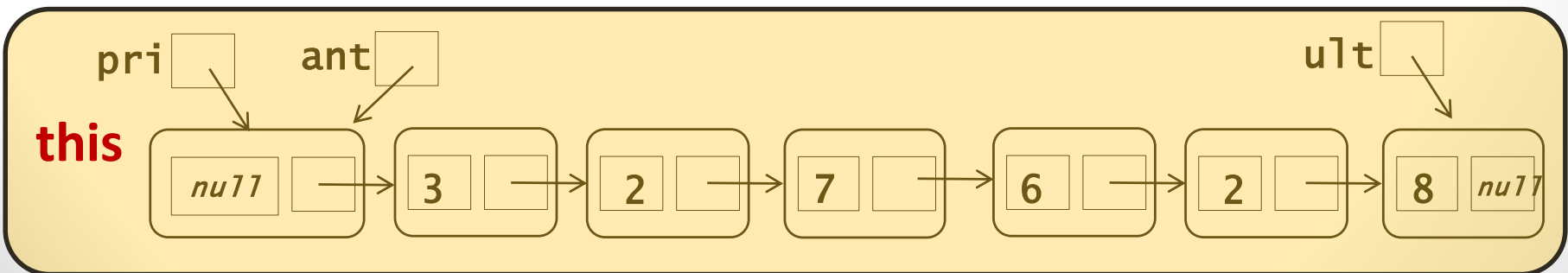
Ahora Sí es
correcto

Ejercicio 4.2

ListaConPIPlus: concatenar



```
public void concatenar(ListaConPI<E> otra) {  
    ...  
}
```



Ejercicio 4.2

ListaConPIPlus: concatenar

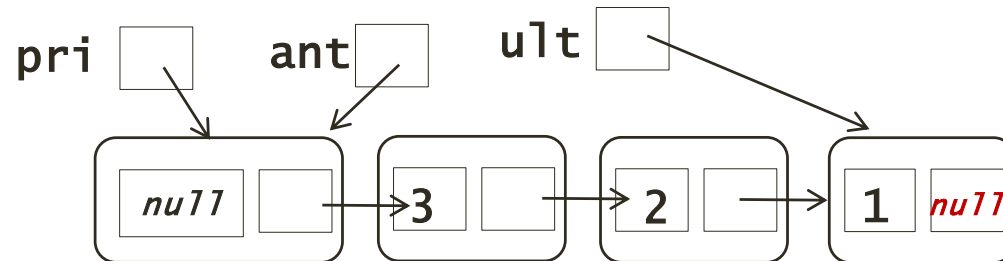
```
public void concatenar(ListaConPI<E> otra) {  
    this.fin();  
    otra.inicio();  
    while (!otra.esFin()) {  
        E dato = otra.recuperar();  
        this.insertar(dato);  
        otra.siguiente();  
    }  
}
```

Implementaciones equivalentes:
con bucle **while**, o con bucle **for**
this se puede omitir

```
public void concatenar(ListaConPI<E> otra) {  
    this.fin();  
    for (otra.inicio(); !otra.esFin(); otra.siguiente()) {  
        E dato = otra.recuperar();  
        this.insertar(dato);  
    }  
}
```

Ejercicio 4.2

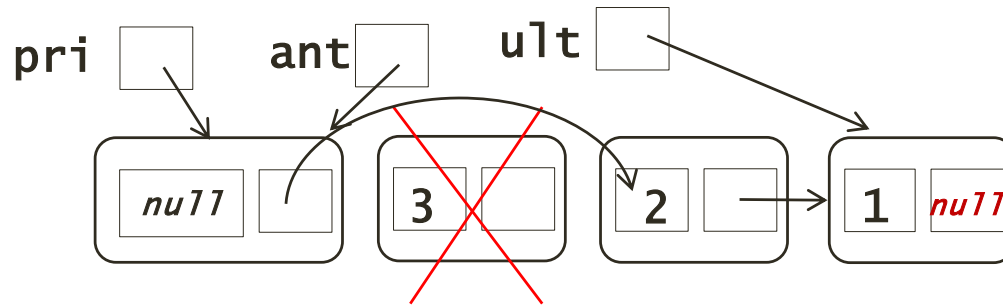
ListaConPIPlus: invertir



```
/** invierte los elementos de una lista */  
void invertir(){  
    if(!esVacia()){  
        inicio();  
        E dato=recuperar();  
        eliminar();  
        invertir();  
        insertar(dato);  
    }  
}
```

Ejercicio 4.2

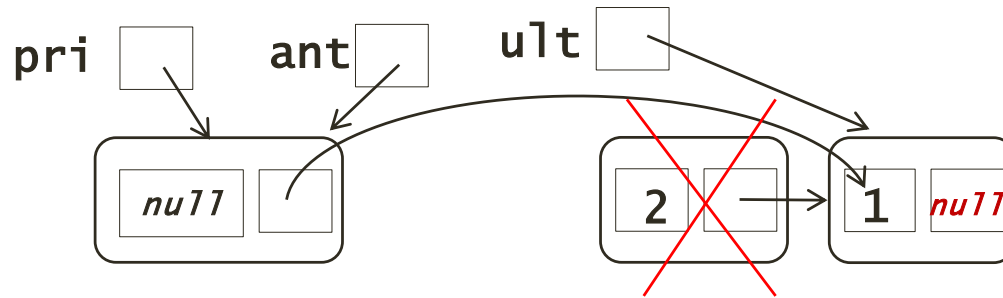
ListaConPIPlus: invertir



```
/** invierte los elementos de una lista */  
void invertir(){  
    if(!esVacia()){  
        inicio();  
        E dato=3  
        eliminar();  
        invertir()  
        insertar(dato);  
    }  
}
```

Ejercicio 4.2

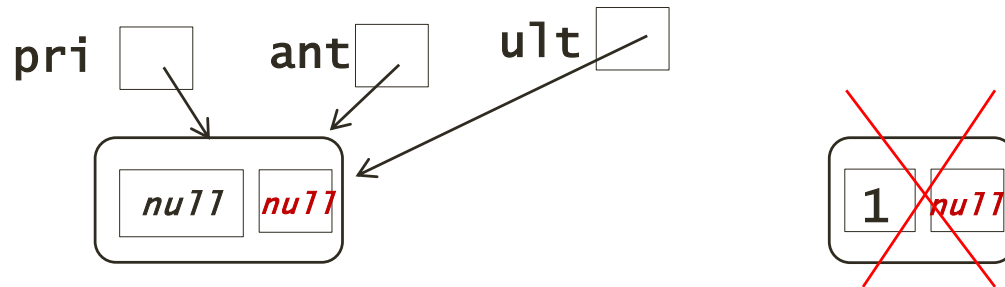
ListaConPIPlus: invertir



```
/** invierte los elementos de una lista **/  
void invertir(){  
    if(!esVacia()){  
        inicio();  
        E dato=2  
        eliminar();  
        invertir()  
        insertar(dato);  
    }  
}
```

Ejercicio 4.2

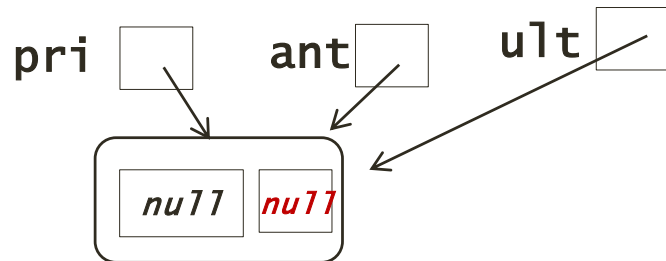
ListaConPIPlus: invertir



```
/** invierte los elementos de una lista **/  
void invertir(){  
    if(!esVacia()){  
        inicio();  
        E dato=1  
        eliminar();  
        invertir()  
        insertar(dato);  
    }  
}
```

Ejercicio 4.2

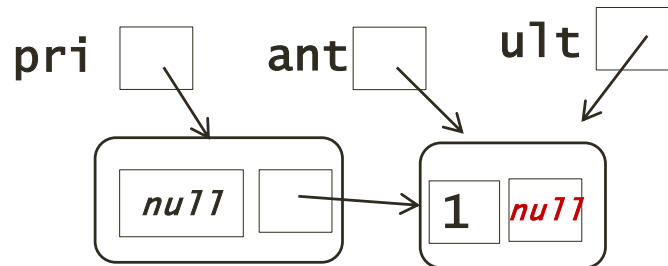
ListaConPIPlus: invertir



```
/** invierte los elementos de una lista **/  
void invertir(){  
    if(!esVacia()){  
        inicio();  
        E dato=1  
        eliminar();  
        invertir()  
        insertar(dato);  
    }  
}
```


Ejercicio 4.2

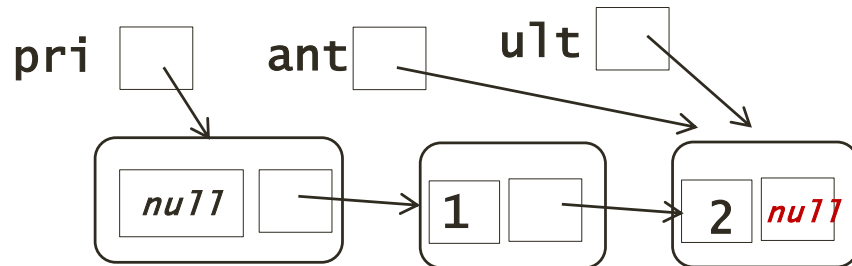
ListaConPIPlus: invertir



```
/** invierte los elementos de una lista */  
void invertir(){  
    if(!esVacia()){  
        inicio();  
        E dato=1  
        eliminar();  
        invertir()  
        insertar(dato);  
    }  
}
```

Ejercicio 4.2

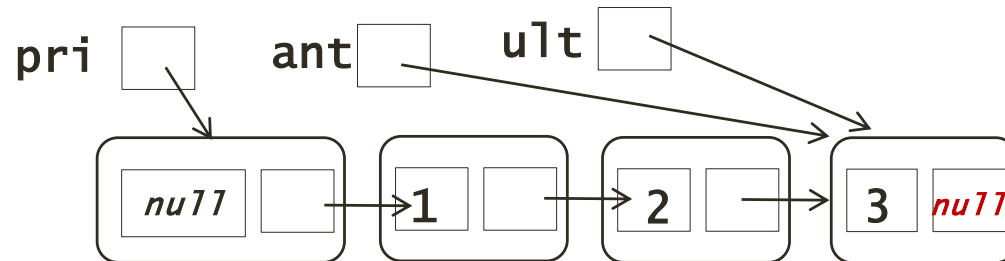
ListaConPIPlus: invertir



```
/** invierte los elementos de una lista */  
void invertir(){  
    if(!esVacia()){  
        inicio();  
        E dato=2  
        eliminar();  
        invertir()  
        insertar(dato);  
    }  
}
```

Ejercicio 4.2

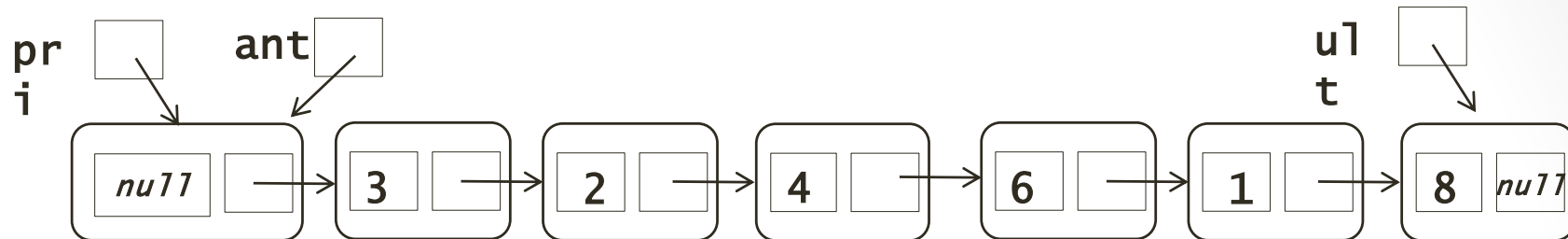
ListaConPIPlus: invertir



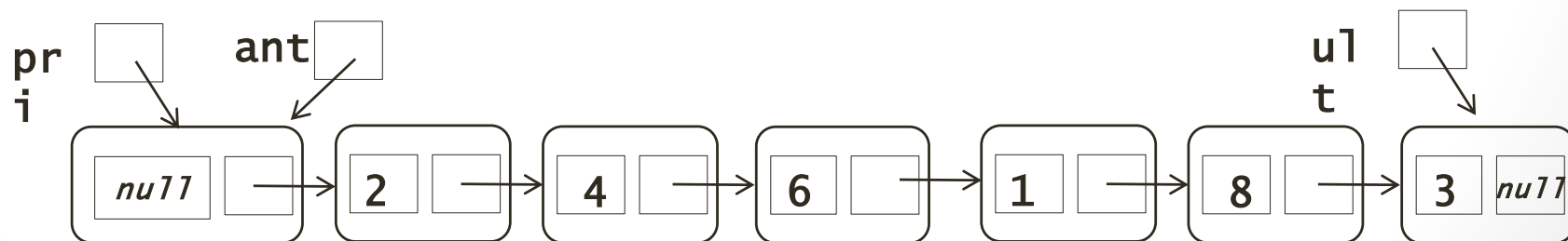
```
/** invierte los elementos de una lista */  
void invertir(){  
    if(!esVacia()){  
        inicio();  
        E dato=3  
        eliminar();  
        invertir();  
        insertar(dato);  
    }  
}
```

Ejercicio 4.3

ListaConPIPlus: mover a izquierda

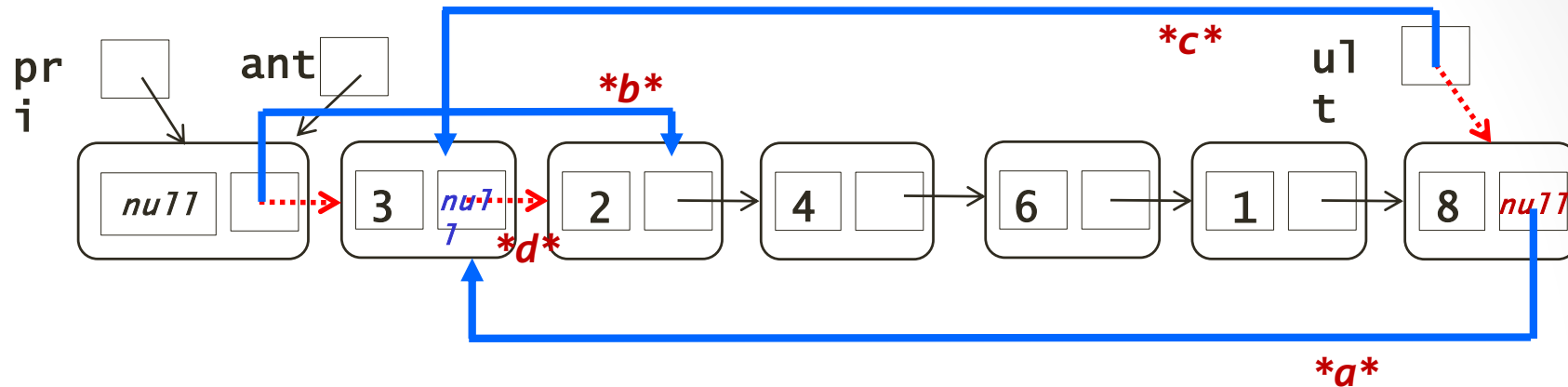


```
// desplaza todos los elementos de la lista una posición  
// hacia la izquierda, de forma que  
// el primer elemento deberá pasar a ser el último  
public void moverAizquierda() {  
    ...  
}
```



Ejercicio 4.3

ListaConPIPlus: mover a izquierda



*// desplaza todos los elementos de la lista una posición
// hacia la izquierda, de forma que
// el primer elemento deberá pasar a ser el último*

```
public void moverAizquierda() {  
    if (talla <= 1) { return; }  
    ult.siguiente = pri.siguiente; // *a*  
    pri.siguiente = pri.siguiente.siguiente; // *b*  
    ult = ult.siguiente; // *c*  
    ult.siguiente = null; // *d*  
}
```

Tema 1 – S4

Estructuras de Datos (EDAs), en Java

Contenidos

6. Clases de tipo genérico restringido por **Comparable**

6. Clases de tipo restringido por *Comparable*

Motivación

¿Cómo resolver los siguientes problemas?

- Mantener **ordenados por antigüedad** los empleados de una empresa.
 - Se suponen disponibles las clases **Empleado** y **Empresa**, y la clase **Empresa** TIENE UNA **ListaConPI<Empleado> l**, tal que, en su constructor, **l = new LEGListaConPI<Empleado>();**
- Mantener **ordenadas por área** las figuras de un grupo,
 - Se suponen disponibles las clases **Figura** y **GrupoDeFiguras**, y la clase **GrupoDeFiguras** TIENE UNA **ListaConPI<Figura> l**, tal que, en su constructor, **l = new LEGListaConPI<Figura>();**
- Mantener **ordenada por palo y valor** una mano de un juego de cartas.
 - Se suponen disponibles las clases **Carta** y **Mano**, y la clase **Mano** TIENE UNA **ListaConPI<Carta> l**, tal que, en su constructor, **l = new LEGListaConPI<Carta>();**

Como no se puede usar el método **insertar** de **ListaConPI**, para insertar en orden ...

Se definiría en cada una de estas clases (**Empresa**, **GrupoDeFiguras** y **Mano**) un método de inserción **en orden**

6. Clases de tipo restringido por *Comparable*

Motivación

```
public void insertar(Empleado aIns) {  
    l.inicio();  
    while (!l.esFin() && l.recuperar().antiguo() < aIns.antiguo())  
        l.siguiente();  
    l.insertar(aIns);  
}
```

```
public void insertar(Figura aIns) {  
    l.inicio();  
    while (!l.esFin() && l.recuperar().area() < aIns.area())  
        l.siguiente();  
    l.insertar(aIns);  
}
```

```
public void insertar(Carta aIns) {  
    l.inicio();  
    while (!l.esFin() && l.recuperar().puntos() < aIns.puntos())  
        l.siguiente();  
    l.insertar(aIns);  
}
```

PROBLEMA: el método **insertar** NO es reutilizable porque su código varía en función del **criterio de comparación de los Elementos de la Lista** -**Empleado** por **antigüedad**, **Figura** por **área**, etc.

¿ **PROBLEMA** ?

¿No proporciona Java uno **genérico**, uno tipo **equals**?


6. Clases de tipo restringido por *Comparable*

Motivación

- Java proporciona un criterio de comparación genérico, un método **compareTo(otro)** que se pueda utilizar casi como un **equals(otro)** ...

→ **Vía Herencia** se puede diseñar una clase genérica **LEGListaConPIOrdenada** que **extends LEGListaConPI** para sobrescribir **insertar**

```
public void insertar(E aIns) {  
    this.inicio();  
    while (!this.esFin() && this.recuperar().compareTo(aIns) < 0)  
        this.siguiente();  
    super.insertar(aIns);  
}
```



→ **Vía Composición** las clases **Empresa**, **GrupoDeFiguras** y **Mano** pueden usar una **LEGListaConPIOrdenada** para mantener en orden sus **Empleado**, **Figura** y **Carta**

```
public void insertar(Empleado aIns) { l.insertar(aIns); }  
public void insertar(Figura aIns) { l.insertar(aIns); }  
public void insertar(Carta aIns) { l.insertar(aIns); }
```

FUNCIONA (por Enlace Dinámico) **SII** las clases **Empleado**, **Figura** y **Carta** sobrescriben **compareTo** para definir su propio criterio de comparación - **antigüedad** en **Empleado**, **área** en **Figura**, etc.

6. Clases de tipo restringido por *Comparable*

El método *compareTo* de la interfaz genérica *Comparable*

- El método **compareTo** NO está definido en **Object**



NO se puede usar exactamente igual que **equals**

- El método **compareTo** está **definido** en la interfaz `java.lang.Comparable`, el modelo estándar y genérico que proporciona Java para la comparación de cualesquiera dos objetos de tipo genérico **E**
- El método, **compareTo** es el **único método** de la interfaz **Comparable**

```
/** compara un elemento con otro y devuelve un valor int ...  
* MENOR QUE 0 si un elemento (this) es MENOR QUE el otro  
* MAYOR QUE 0 si un elemento (this) es MAYOR QUE el otro  
* IGUAL A 0 si un elemento (this) es IGUAL A el otro  
*/  
public abstract int compareTo(E otro);
```

6. Clases de tipo restringido por *Comparable*

El método *compareTo* de la interfaz genérica *Comparable*

Para poder comparar dos objetos de la clase **E**,
obligatoriamente **E implements Comparable<E>**
y, por ello, sobrescribe su método **compareTo**

Así ...

- Integer, Double, String **implements** –respectivamente–
Comparable<Integer>, **Comparable<Double>**, **Comparable<String>**
- Empleado, Figura, Carta **implements** –respectivamente–
Comparable<Empleado>, **Comparable<Figura>**, **Comparable<Carta>**

Recomendación:

si **x.compareTo(y)==0** entonces **x.equals(y)**

sino, incluir en la documentación de la clase una advertencia tipo ...

“Note: this class has a natural ordering that is inconsistent with equals”

6. Clases de tipo restringido por *Comparable*

La clase *Figura Comparable*

```
public abstract class Figura implements Comparable<Figura> {  
    ...  
    public int compareTo(Figura f) {  
        double areaF = f.area(), areaThis = this.area();  
        if (areaThis < areaF) return -1;  
        if (areaThis > areaF) return +1;  
        return 0;  
    }  
}
```

// Implementación alternativa:

```
public int compareTo(Figura f) {  
    return (int) Math.signum(this.area() - f.area());  
}
```


El método lanza la excepción **ClassCastException** si el objeto pasado como argumento no es *Figura*

6. Clases de tipo restringido por *Comparable* *E extends Comparable<E>*

Suponiendo que ...

- La clase **LEGListaConPIOrdenada** está disponible en el paquete lineales

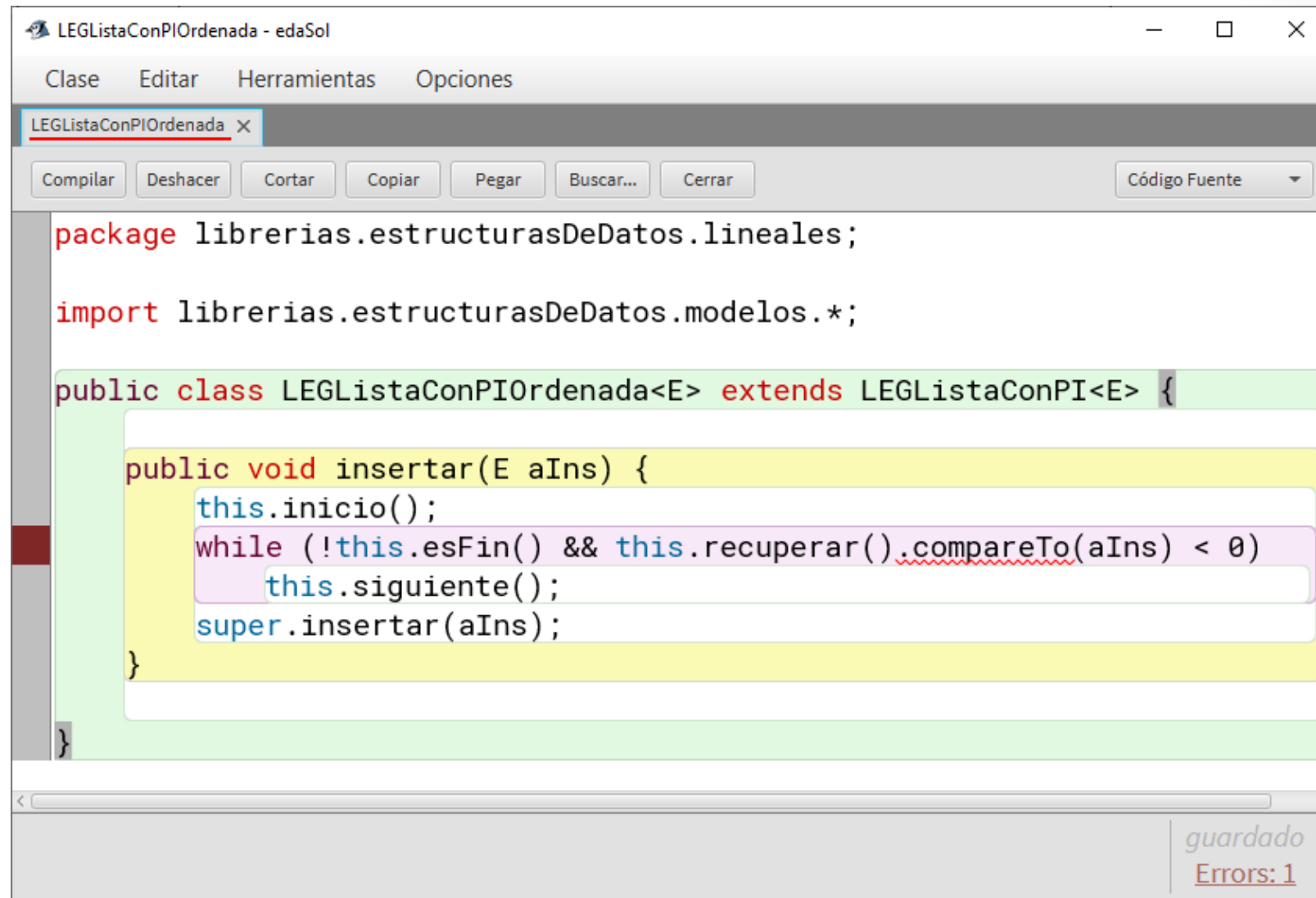
```
public class LEGListaConPIOrdenada<E> extends LEGListaConPI<E> {  
    ...  
    public void insertar(E aIns) {  
        this.inicio();  
        while (!this.esFin() && this.recuperar().compareTo(aIns) < 0)  
            this.siguiente();  
        super.insertar(aIns);  
    }  
}
```



- La clase **GrupoDeFiguras** TIENE UNA **ListaConPI<Figura>** implementada mediante una **LEGListaConPIOrdenada<Figura>**
- La clase **Figura** implements **Comparable<Figura>**, sobrescribiendo **compareTo**

¿Qué ocurre al compilar **LEGListaConPIOrdenada**?

6. Clases de tipo restringido por *Comparable* *E extends Comparable<E>*



```
package librerias.estructurasDeDatos.lineales;

import librerias.estructurasDeDatos.modelos.*;

public class LEGListaConPIOrdenada<E> extends LEGListaConPI<E> {

    public void insertar(E aIns) {
        this.inicio();
        while (!this.esFin() && this.recuperar().compareTo(aIns) < 0)
            this.siguiente();
        super.insertar(aIns);
    }
}
```

guardado
Errors: 1

¿Por qué se produce este error? ¿Cómo resolverlo?

6. Clases de tipo restringido por *Comparable*

E extends Comparable<E>

```
public class LEGListaConPIOrdenada<E extends Comparable<E>>
extends LEGListaConPI<E> {
    public void insertar(E aIns) {
        this.inicio();
        while (!this.esFin() && this.recuperar().compareTo(aIns) < 0)
            this.siguiente();
        super.insertar(aIns);
    }
}
```

La clase debe ser de tipo genérico
restringido por **Comparable**

6. Clases de tipo restringido por *Comparable*, o *T extends Comparable<T>*

Ejercicio 5.1

Ampliar la funcionalidad de la EDA **Pila** mediante herencia (**PilaExt**, **LEGPilaExt**) para añadir un nuevo método que devuelva el elemento más pequeño de la pila. Implementa este método:

- a) Accediendo a los atributos de **LEGPila**.
- b) Utilizando únicamente los métodos del modelo.

Ejercicio 5.1

Subinterfaz

```
public interface PilaExt<E> extends Comparable<E>> extends
Pila<E>
{
    // IFF !esVacía()
    E minimo();
}
```

Ejercicio 5.1

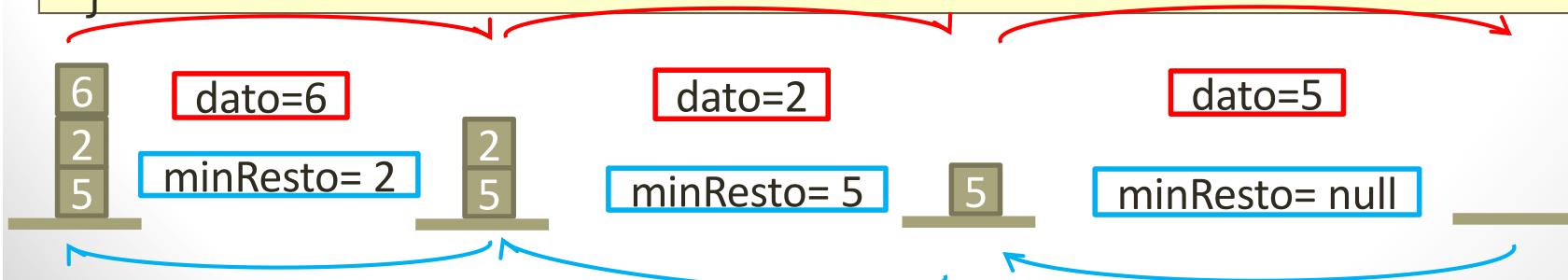
Solución a)

```
public class LEGPilaExt<E extends Comparable<E>> extends
LEGPila<E> implements PilaExt<E> {
    /** devuelve el elemento más pequeño de la pila
    * accediendo a los atributos de LEGPila */
    public E minimo() {
        NodoLEG<E> aux = tope;
        E min = null;
        while (aux != null) {
            if (min == null || aux.dato.compareTo(min) < 0)
                min = aux.dato;
            aux = aux.siguiente;
        }
        return min;
    }
}
```

Ejercicio 5.1

Solución b)

```
public class LEGPilaExt<E extends Comparable<E>> extends  
LEGPila<E> implements PilaExt<E> {  
    /** devuelve el elemento más pequeño de la pila  
     * utilizando sólo los métodos del modelo */  
    public E minimo() {  
        if (esVacia()) return null;  
        E dato = desapilar();  
        E minResto = minimo();  
        apilar(dato);  
        if (minResto == null || dato.compareTo(minResto) < 0)  
            return dato;  
        return minResto;  
    }  
}
```



6. Clases de tipo restringido por *Comparable*, o *T extends Comparable<T>*

Ejercicio 5.2

En **LEGListaConPi** crea un método estático tal que, dadas dos **ListaConPi** genéricas, ambas sin elementos repetidos y ordenadas de forma ascendente, elimine de dichas listas todos los elementos que tengan en común y los devuelva almacenados en una Cola.

Ejercicio 5.2.

Método estático encolarRepetidos

*/** método estático tal que, dadas dos ListasConPI genéricas, ambas sin elementos repetidos y ordenados ascendentemente, elimine de dichas listas todos los elementos que tengan en común y los devuelva almacenados en una Cola */*

```
public static <E extends Comparable<E>> Cola<E> encolarRepetidos
(ListaConPI<E> l1, ListaConPI<E> l2) {
    Cola<E> c = new ArrayCola<E>();
    l1.inicio(); l2.inicio();
    while (!l1.esFin() && !l2.esFin()) {
        E e1 = l1.recuperar(); E e2 = l2.recuperar();
        int cmp = e1.compareTo(e2);
        if (cmp == 0) {
            c.encolar(e1); l1.eliminar(); l2.eliminar();
        }
        else if (cmp < 0) { l1.siguiente(); }
        else { l2.siguiente(); }
    }
    return c;
}
```

Ejercicio muy similar a otro de examen, ver examen resuelto de fecha **31 marzo 2021**