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
   - Jerarquía Java de una EDA: componentes y nomenclatura
   - Organización de jerarquías de EDAs en librerías (*package**s*** BlueJ)
   - Criterios de diseño de las clases de una jerarquía. Ejemplos para las jerarquía Lineal de ***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 **Estructura**ción –u organización- **de** los **Datos** (**EDA**)

1. Estructuras de Datos

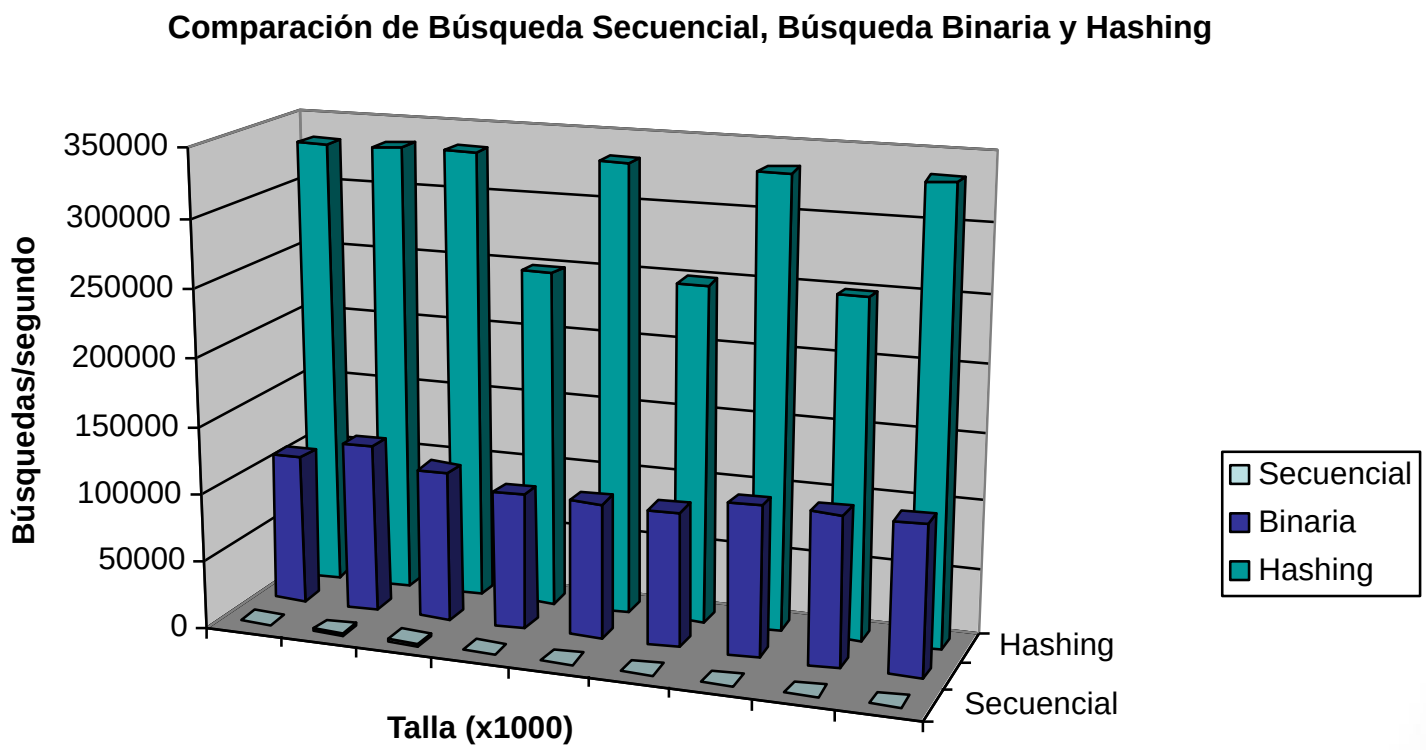
*Motivación: ejemplos de gestión de una Colección de Datos y el **coste** asociado a su implementación*

Ejemplo 1 - Aplicación de gestión de una Colección de Trabajos y su coste: **Modelos Cola y Cola de Prioridad**

Ejemplo 2 - Aplicación de gestión de una Colección de Entradas y su coste: **Modelo Diccionario (o *Map*)**

1. Estructuras de Datos

*¿Motivación?... Comparativa del **coste** de la gestión de un Diccionario con una Lista o una Tabla Hash del Ejemplo 2*



Secuencial	1471	2000	1351	1053	826	680	578	503	441
Binaria	111110	125002	111110	100000	100000	100000	111110	111110	111110
Hashing	333331	333357	333331	250003	333331	250003	333331	250003	333331

1. Estructuras de Datos

Definición

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:

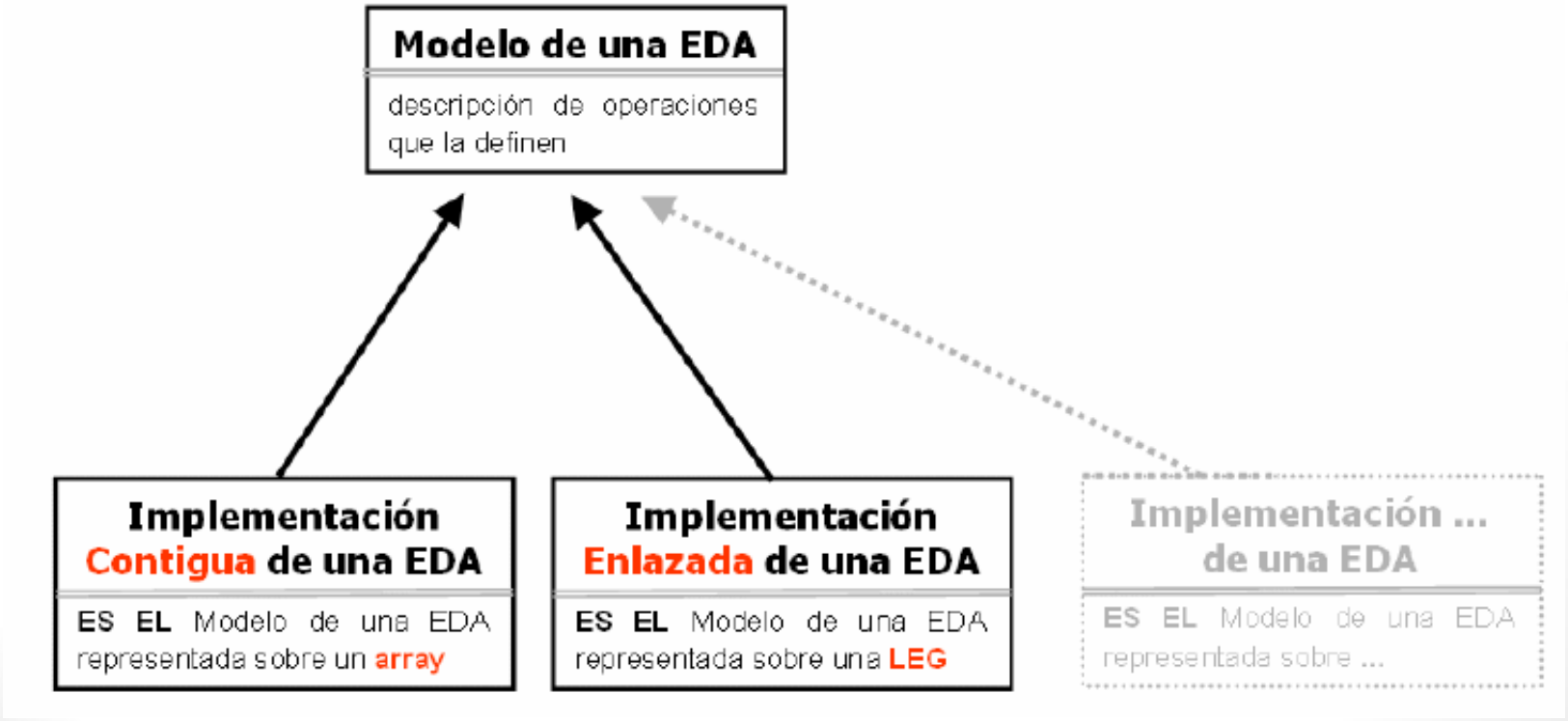
- **MODELO**, o Especificación, de una EDA: descripción de las operaciones que definen su funcionalidad, el tipo de gestión de Datos que realiza, con independencia de su posterior representación en memoria

- **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

Jerarquía de una EDA

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 Binaria, por ejemplo para obtener el "Camino Mínimo" que los une

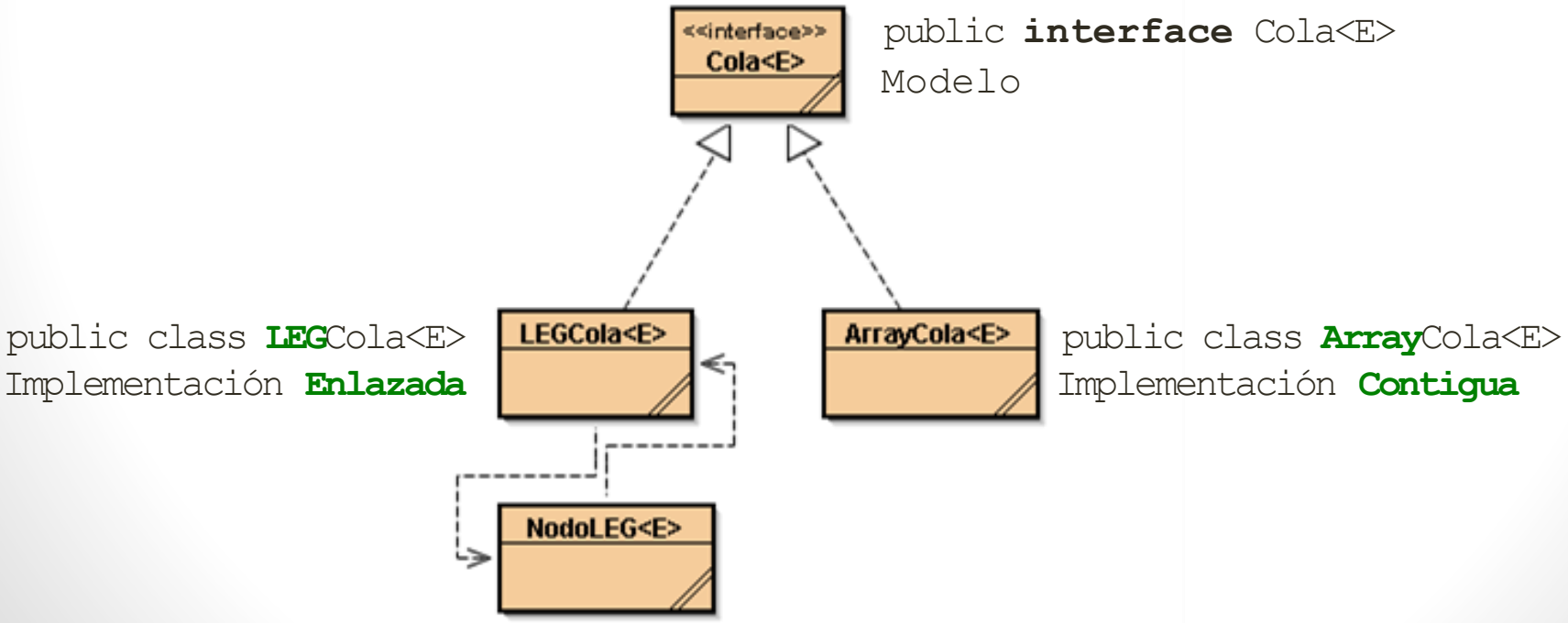
2. Definición (diseño) de una EDA en Java

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
- Una Derivada de la Raíz (vía `implements`) de tipo genérico por cada una de sus Implementaciones -Contigua, Enlazada, etc.

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



2. Definición (diseño) de una EDA en Java

Organización de jerarquías de EDAs en librerías (packages BlueJ)

2. Definición (diseño) de una EDA en Java

*Criterios de diseño **de la Raíz** de la jerarquía - aplicados al Modelo de una Cola*

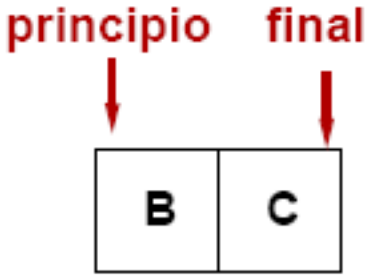
Sabiendo que una Cola es una Colección de **E**lementos que se gestionan siguiendo un criterio FIFO, i.e. permitiendo la consulta sólo del primero de ellos en orden de inserción**... ¿Cómo se consigue diseñar la siguiente `interface`?**

```
package librerias.estructurasDeDatos.modelos;
public interface Cola<E> {
    void encolar(E  e);                                  // Θ(1)
    /** SII !esVacia()**/ E desencolar();                // Θ(1)
    /** SII !esVacia()**/ E primero();                   // Θ(1)
    boolean esVacia();                                   // Θ(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. Definición (diseño) de una EDA en Java

*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() {…}                    // Θ(x)
  public void encolar(E e) {…}                    // Θ(1)
  private void duplicarArrayCola() {…}            // Θ(x)
  private int incrementar(int indice) {…}         // Θ(1)
  …
}
```

Ejercicios de la Sesión 1:

Para que te familiarices con los conceptos básicos sobre definición de una EDA y su Jerarquía Java, hemos preparado un cuestionario en PoliformaT

Ejercicio 1: Examen poli formaT

Tema 1 - S1: Cuestiones sobre el diseño de la Jerarquía de una EDA

Para resolverlo, puedes consultar el documento disponible en PoliformaT...

"Apuntes (1) – Diseño y Uso de la Jerarquía de una Estructura de Datos (EDA)"

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`**
2. **Vía Composición** (TIENE UN)
3. **Vía Herencia** (ES UN)
4. **Combinando** cualquiera de las posibilidades anteriores

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

3. Uso de la jerarquía Java de una EDA

Vía `import` - Programa ejemplo `TestEdaCola`

Para recordar cómo usar vía `import` la Jerarquía Java de una EDA, así como las limitaciones que presenta, responde a las preguntas del cuestionario PoliformaT...

Ejercicio 2: Examen poli formaT

Tema 1 - S2: Cuestiones sobre el uso de la Jerarquía de una EDA vía import

Para resolverlo, puedes consultar el documento disponible en PoliformaT...

"Apuntes (2) – Herencia y Polimorfismo"

3. Uso de la jerarquía Java de una EDA

Vía Composición (TIENE UN(A)) - Programa ejemplo `GestorDePacientes`

- 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. **Modifica** la clase `GestorDePacientes`, también en el paquete `tema1` de `ejemplos`, para que pueda ejecutarse sin problemas con toda su funcionalidad.

3. Uso de la jerarquía Java de una EDA

Vía Herencia (ES UN) - Ampliación de la Jerarquía Java de una EDA

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

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

Vía Herencia (ES UN) - Ejemplo: Ampliación de `Cola` (I)

- 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

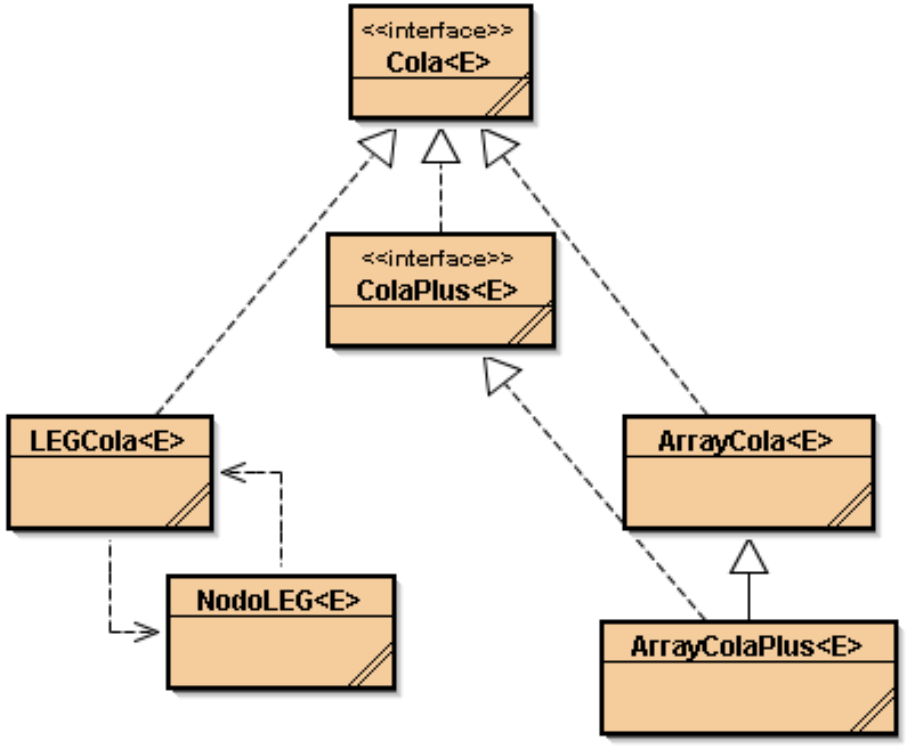
Vía Herencia (ES UN) - Ejemplo: Ampliación de `Cola` (II)

```
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 <u>menos eficiente</u>**
VENTAJA: **esta solución es <u>siempre posible</u>**

3. Uso de la jerarquía Java de una EDA

Vía Herencia (ES UN) - Ejemplo: Ampliación de `Cola` (III)



3. Uso de la jerarquía Java de una EDA

Combinando todas las vías - Programa ejemplo `GestorDePacientesPlus`

Tras ampliar la Jerarquía Java `Cola` ya se puede emplear el método `talla()` en aplicaciones como `TestEDACola` o `GestorDePacientes`:

basta con declarar variables de tipo Estático `ColaPlus`, en lugar de `Cola`, y tipo Dinámico `ArrayColaPlus`, en lugar de `ArrayCola`

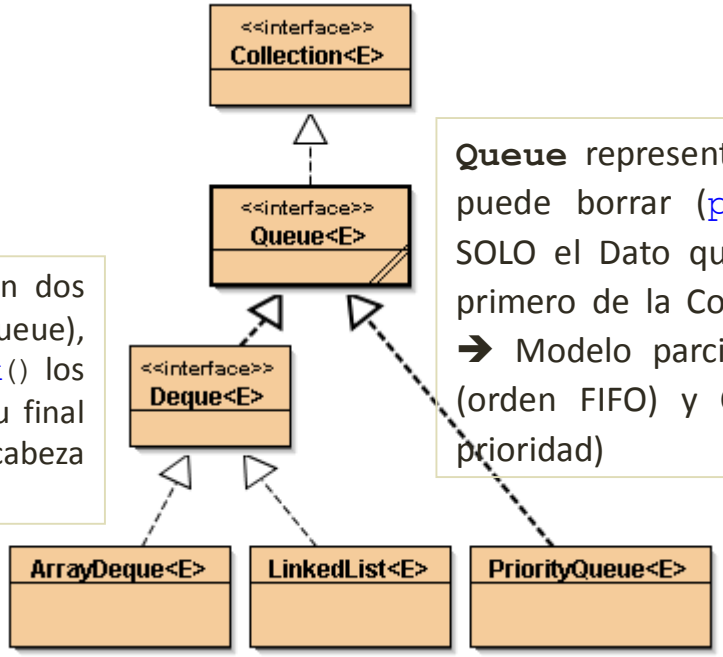
Ejercicio 3: La clase GestorDePacientesPlus

(clave: CCDFJ00Z)

Concluye el diseño de la clase `GestorDePacientesPlus`, que usa una `ColaPlus` implementada mediante un `ArrayColaPlus`

4. EDAs en el estándar de Java

La jerarquía `Collection`



`Deque` ES UNA **`Queue`** con dos extremos (**D**ouble **e**nded **q**ueue), que añade a `poll()` y `peek()` los métodos de inserción en su final (`addLast(e)`) y en su cabeza (`addFirst(e)`)

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

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

`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 ➔Implentación eficiente de **Cola de Prioridad**

Ejercicios de la Sesión 2:

Ejercicio 4: La clase ArrayDequeCola

(clave CCDFI00Z)

Usando `ArrayDeque` vía Herencia, diseña la clase `ArrayDequeCola` que implementa la interfaz `Cola` y es equivalente a `ArrayCola`

Ejercicio 5: usando vía Herencia la clase `ArrayDequeCola` (ejercicio 4), escribe la clase `ArrayDequeColaPlus` que implementa la interfaz `ColaPlus`

Ejercicio 6: usa la jerarquía `Deque` en el diseño de la clase `TestEDAColaVDeque`

Ejercicio 7: La clase GestorDePacientesVDeque

(clave CCDFK00Z)

Usa la (sub)jerarquía `Deque` en el diseño de la clase `GestorDePacientesVDeque`

Tema 1 – S3

Estructuras de Datos (EDAs), en Java

Contenidos

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

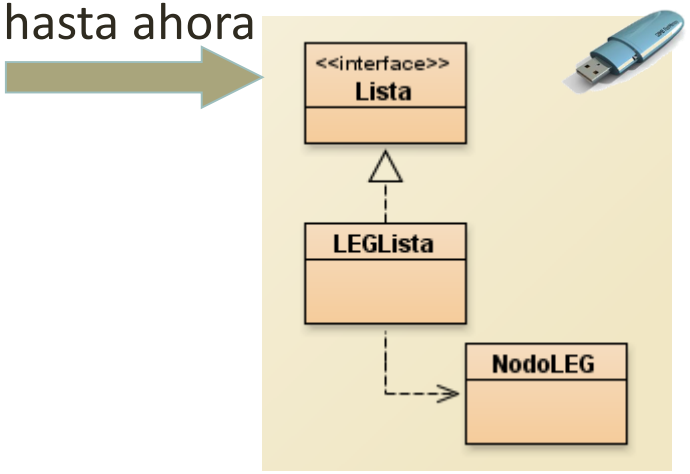
5. Lista con Iterador (o Punto de Interés)

Motivación

Una **Lista** es una Colección de Datos que se gestionan **Secuencialmente**, i.e. accediendo uno tras otro a cada uno de sus Datos, desde el primero al último, en orden de inserción

- **Ejemplo:** la Lista de la Compra

  - Para no sobrecargar el coste ”natural” de la gestión Secuencial, proporcional a la talla x (O(x)) si el coste de tratar cada elemento es O(1), cada uno de los métodos del Modelo **Lista** debería poder implementarse con un **coste máximo estimado estrictamente del orden de una constante**, lo que **NO** es posible con una jerarquía Java como la propuesta hasta ahora



5. Lista con Iterador (o Punto de Interés)

Modelo: definición del Punto de Interés de una Lista y sus consecuencias sobre la gestión de esta

Idea básica: por definición, en cada instante de la gestión secuencial sólo es posible acceder a un Dato de la Lista, el que ocupa en dicho instante su ***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 resulta obvio que...
  - 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, recuperar** o **eliminar** un Dato de una Lista el PI permanece **inamovible**
  - 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, **eliminar**lo o pasar al **siguiente** provoca una excepción

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

5. Lista con Iterador (o Punto de Interés)

Modelo Java, o Raíz de la jerarquía `ListaConPI`

```
package librerias.estructurasDeDatos.modelos;

public interface ListaConPI<E> {
  /** Inserta e en una Lista ANTES del Elemento que ocupa su PI,
    * que permanece INALTERADO */ void insertar(E e);                    // Θ(1)
  /** SII !esFin(): elimina de una Lista el Elemento que ocupa su PI,
    * que permanece INALTERADO */ void eliminar();                       // Θ(1)
  /** SII  !esFin(): obtiene el Elemento que ocupa el PI
    * de una Lista */ E recuperar();                                     // Θ(1)
  /** Sitúa el PI de una Lista en su inicio */ void inicio();            // Θ(1)
  /** SII !esFin(): avanza el PI de una Lista */ void siguiente();       // Θ(1)
  /** Comprueba si el PI de una Lista está en su fin */ boolean esFin(); // Θ(1)
  /** Comprueba si una Lista Con PI está vacía */ boolean esVacia();     // Θ(1)
  /** Sitúa el PI de una Lista en su fin */ void fin();                  // Θ(1)
  /** devuelve la talla de una Lista Con PI */ int talla();              // Θ(1)
}
```

Implementación Java: la clase `LEGListaConPI`

5. Lista con Iterador (o Punto de Interés)

Ejercicios de la Sesión 3

Para comprobar si has entendido el funcionamiento de una Lista Con PI, responde a las preguntas del cuestionario PoliformaT...

Ejercicio 8: Examen poliformaT

Tema 1 - S3: Cuestiones sobre el Modelo Lista Con PI y su uso

Para resolverlo, puedes consultar el documento disponible en PoliformaT...

"Apuntes (4) – Lista, Pila y Cola: Modelo y Aplicaciones"

+ Ejercicios de la Sesión 3:

Ejercicio 9: la clase `LEGListaConPIPlus` es una Implementación preliminar del (sub)Modelo `ListaConPIPlus`, pues sus siguientes métodos, bien no se han diseñado aún, bien contienen algún error. Diséñalos o corrígelos para obtener la versión definitiva de esta clase

`ListaConPIPlus: contiene`
(clave: CCDFGH11[)

`ListaConPIPlus: eliminar`

(clave: CCDFGJ11[)

`ListaConPIPlus: concatenar`

(clave: CCDFGK11[)