Estructuras de Datos y Algoritmos.
Unidad Didáctica I: Conceptos de Java para Estructuras de Datos.

Tema 6. Diseño y uso de la Jerarquía Java de una Estructura de Datos (EDA)

Mabel Galiano Natividad Prieto

Departamento de Sistemas Informáticos y Computación
Escuela Técnica Superior de Ingeniería Informática

Índice

1. Introducción a las EDAs: definición y clasificación **3**

2. Diseño de la Jerarquía Java de una EDA **6**

3. Organización de Jerarquías de EDAs en librerías BlueJ **8**

4. Uso de una EDA: reutilización de su Jerarquía Java **9**

5. EDAs en el estándar de Java: características y uso de la Jerarquía `Collection` **12**

Objetivos y Bibliografía

El objetivo principal de este tema es introducir la Jerarquía como la herramienta imprescindible para la representación y uso eficientes de una Estructura de Datos (EDA) en Java, por lo que la presentación de sus contenidos se realiza como sigue: en su primer apartado se define una EDA como la organización concreta de los Datos de una Colección que requiere una aplicación y se clasifica como Lineal (Pila, Cola y Lista), de Búsqueda (Diccionario y Cola de Prioridad) o Grafo según el tipo de gestión de Datos que realice; a partir de esta introducción, y utilizando como ejemplo ilustrativo la organización de los Trabajos de una Colección según una Cola y una Cola de Prioridad, los tres siguientes apartados del tema se dedican a exponer las características generales de, respectivamente, el diseño, la organización y el uso de la Jerarquía de clases que representa una EDA en Java; finalmente se presenta la Jerarquía `Collection` de `java.util`, la representación en el estándar de Java de las EDAs más frecuentemente utilizadas e instrumento principal del lenguaje para el diseño de las EDAs de Usuario.

De esta forma, por su objetivo y contenidos, este tema se puede considerar un epílogo de la primera Unidad Didáctica de la asignatura y un prólogo de las tres restantes: en él se aplican al ejemplo paradigmático de las Estructuras de Datos (EDAs) los principales conceptos Java

estudiados hasta la fecha (Ocultación de la Información, Herencia, Genericidad y Eficiencia) y, al mismo tiempo, en él se introducen las ideas y nomenclatura básica que requieren el diseño y reutilización en Java de las distintas EDAs que se estudian en los restantes temas del curso.

Como bibliografía básica del tema se recomienda la lectura de los apartados `6.1`, `6.3` y `6.8` del Capítulo `6` del libro de Weiss M.A. **Estructuras de datos en Java** (Adisson-Wesley, 2000); asimismo, se pueden consultar las características de la Jerarquía `Collection` en la dirección **http://java.sun.com/docs/books/tutorial/collections/index.html**.

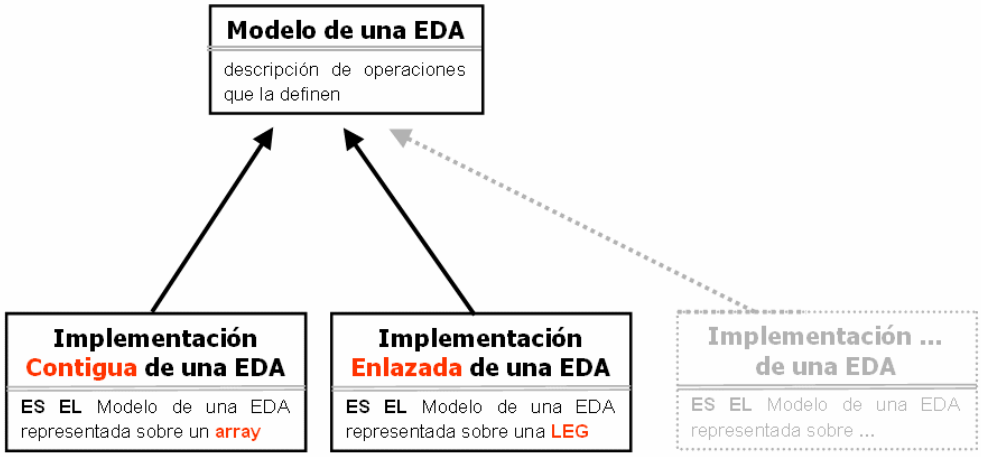
1. Introducción a las EDAs: definición y clasificación

Cualquier aplicación informática exige manipular Colecciones de Datos de talla elevada; básicamente se debe poder insertar, borrar y recuperar un Dato de la Colección, recorrerla para tratar todos sus Datos de una cierta manera y buscar aquel de sus Datos que satisfaga una propiedad relevante para la aplicación. Para ello resulta imprescindible realizar una estructuración de la Colección y sus Datos que no sólo posibilite su manipulación eficaz sino que también, más importante aún, permita su reutilización en otras aplicaciones. Específicamente,

> se denomina **E**structura **D**e **D**atos (**EDA**) al conjunto formado por las operaciones que definen el comportamiento o funcionalidad de una Colección de Datos y la posible representación de ésta en memoria.

A partir de esta definición resulta fácil observar que para describir una EDA es necesario, a su vez, describir dos niveles de abstracción o tipos de componentes: el **Modelo** o **Especificación** de la EDA, esto es la descripción del conjunto de operaciones que definen su funcionalidad con independencia de su posterior representación en memoria, y la **Implementación** de la EDA, esto es la descripción de la Representación o soporte en memoria elegido para sus Datos (Contiguo, Enlazado o mixto) así como la implementación de las operaciones de su Modelo en base a tal Representación.

La descripción a dos niveles de una EDA resulta muy útil desde el punto de vista de la Programación Orientada a Objetos. Por un lado, permite distinguir al Modelo de la EDA como Factor-Común de sus diversas Implementaciones y, por tanto, establece una relación Jerárquica ES UN(A) entre éstas y el Modelo: como resume gráficamente la siguiente figura, una Implementación Contigua de una EDA ES LA descripción que se obtiene de sus operaciones cuando se usa un `array` como soporte en memoria de sus Datos; en cambio, si el soporte elegido es una Lista Enlazada entonces lo que se obtiene ES LA Implementación Enlazada de su Modelo; etc.



Como se verá en los siguientes apartados, una vez establecida la relación entre el Modelo y las Implementaciones de una EDA queda clara la regla básica a seguir para que su diseño y uso en Java resulten óptimos, o favorezcan al máximo la Reutilización del Software: aplicar sistemáticamente los mecanismos de Genericidad y Herencia (vía `extends` o vía `implements`).

Por otro lado, la descripción de una EDA en términos de un Modelo único (pero) con diversas Implementaciones permite que al desarrollar una aplicación concreta sea posible elegir el Modelo de EDA que más eficientemente gestiona sus Datos sin conocer su representación en memoria y

luego, tras alcanzar un nivel de abstracción más bajo, considerar cuál es la Implementación más eficiente de éste que es factible realizar. En concreto, las EDAs que se estudian en la asignatura se puede clasificar en base a la eficiencia de su Modelo como sigue:

- **Lineales** o las que cabe emplear en aplicaciones en las que interesa una **gestión Secuencial** (uno tras otro) de sus Datos, i.e. atendiendo única y exclusivamente al orden de incorporación de éstos; en estos casos -tanto si el criterio de gestión es LIFO (*Last In First Out*), FIFO (*First In First Out*) o general- un sencillo Modelo Lineal -Pila, Cola o Lista Con Punto de Interés- caracteriza eficientemente cuál es el siguiente dato a tratar.

  En cuanto a las Implementaciones que se pueden realizar de una EDA Lineal, se obtienen utilizando una Representación Lineal (`array` o LEG) de los Datos tal que permita implementar en el orden de una constante cualquier operación de su Modelo.
- **de Búsqueda** o, como su propio nombre indica, las que cabe emplear en aplicaciones donde la Búsqueda de un Dato dado es el objetivo principal; piénsese por ejemplo en un proceso de Traducción de un idioma a otro, donde hay que buscar la traducción de cada palabra en un Diccionario antes de poder traducirla efectivamente, o en la gestión de las Urgencias de un hospital, donde se requiere determinar en cada momento cuál de todos los pacientes de una sala es el más grave antes de poder atenderlo.

  Como no resulta difícil deducir, elegir una EDA Lineal para una aplicación de éste tipo resulta claramente ineficiente: la Búsqueda Secuencial tiene un coste lineal con la talla del problema y la Búsqueda Binaria sólo se puede realizar sobre un `array` ordenado y, por tanto, a costa de insertar nuevos Datos en tiempo lineal en vez de constante. Precisamente por ello es necesario definir nuevas EDAs en las que la Búsqueda se realice en tiempo subLineal, como Diccionario y Cola de Prioridad según se busque el Dato asociado a una cierta Clave o el de mayor prioridad respectivamente; como se verá en temas posteriores, conseguir este objetivo requiere una sofosticada Representación de los Datos, un soporte no Lineal (Jerárquico, como el Árbol Binario de Búsqueda o el Montículo Binario, o Tabla de Dispersión) en el que el Dato a buscar bien se encuentra en una posición específica conocida de antemano o bien en una zona concreta junto con aproximadamente la mitad de los Datos.
- **de Relación o Grafo** o la que cabe emplear en las aplicaciones cuyos Datos guardan entre sí una Relación Binaria -más general que la Lineal y la Jerárquica- y cuyo objetivo principal es obtener el "Camino Mínimo" -u orden de visita óptimo- que pudiera existir entre los Datos conforme a tal Relación; así por ejemplo, según se interprete el término Camino Mínimo, aplicaciones de este tipo serían establecer la ruta óptima (por distancia, coste económico, interés turístico, etc.) entre dos puntos de una red (aeropuertos, estaciones de tren, ciudades o calles de una ciudad, ordenadores, etc.), planificar de manera óptima los cursos/asignaturas necesarios para obtener un título universitario o las actividades de administración y desarrollo de un proyecto de envergadura, interconectar con la menor cantidad posible de cable los `N` pines de un circuito con `N-1` cables, ...

  Para garantizar su eficiencia, en la Implementación de un Grafo intervienen todas las EDAs Lineales y de Búsqueda presentadas en los puntos anteriores; en concreto, para la Representación eficiente de sus Datos y las relaciones que éstos guardan entre sí es necesario usar Diccionario y Lista Con Punto de Interés y para la implementación de sus operaciones de Exploración, las de Camino Mínimo incluidas, es necesario usar Pila, Cola y Cola de Prioridad.

Para ilustrar de manera intuitiva los conceptos más importantes introducidos hasta el momento, considérese el siguiente caso: durante el desarrollo de una aplicación de gestión de una Colección de Trabajos -los Ficheros a imprimir tras una sesión de prácticas, las Tareas a realizar por un Sistema Operativo (SO), las Entradas de cine a vender por taquilla, los Pacientes a visitar en una consulta médica, ...- se ha identificado como clase principal `Gestor`, la que lleva a cabo la gestión de los Trabajos mediante las siguientes operaciones:

- `insertar(Trabajo t)`, que incorpora a la Colección un nuevo Trabajo `t` a tratar -un nuevo Fichero, una nueva Tarea, un nuevo Paciente, ...
- `recuperar()`, que selecciona el siguiente Trabajo de la Colección a tratar -el siguiente Fichero a imprimir, la siguiente Tarea a ejecutar, el siguiente Paciente a visitar,...
- `eliminar()`, que quita de la Colección el Trabajo que acaba de ser tratado -el Fichero ya impreso, la Tarea ya ejecutada, el Paciente ya atendido, ...
- `esVacia()`, que comprueba si queda algún Trabajo de la Colección aún por tratar.

Especificada la funcionalidad o trabajo básico del `Gestor` sólo queda por establecer el criterio o Modelo a seguir para llevarlo a cabo, i.e. resta por establecer cuál de todos los Trabajos de la Colección es el que se debe `recuperar()` para ser tratado en cada momento -si el primero que se insertó o el último o uno elegido al azar o el más urgente o prioritario. Como es fácil imaginar, lo más sencillo y eficaz es `recuperar` siempre el Trabajo que se insertó primero o, en otros términos, seguir el criterio ***F**irst **I**n **F**irst **O**ut* (`FIFO`) de gestión; de esta forma,

- `insertar(Trabajo t)` consiste en añadir `t` al final de la Colección, por lo que resulta más natural denominar a esta operación `encolar(Trabajo t)`;
- `recuperar()` consiste en obtener el Trabajo de la Colección que lleva más tiempo encolado, de donde `primero()` es un nombre más apropiado para esa operación;
- `eliminar()` consiste en quitar el primer Trabajo de la Colección una vez tratado, por lo que resulta más apropiado notarla como `desencolar()`.

Por tanto y en resumen, la clase `Gestor` usa una **Cola** para organizar la Colección de Trabajos.

Otro caso de interés que se deriva del presentado es aquel en el que los Trabajos a gestionar tienen una cierta prioridad -la un Fichero a imprimir sería inversamente proporcional a su longitud, la de una Tarea del SO una función del grupo al que pertenece su propietario, la de un Paciente a visitar proporcional a la gravedad de su estado, etc. Nótese entonces que si se atienden los Trabajos de acuerdo con su prioridad, mínimo tiempo de espera para el Trabajo de máxima prioridad, se consigue una gestión que permite no sólo tratarlos a todos como en el ejemplo anterior sino también minimizar o maximizar una cierta función objetivo -maximizar la utilización que hace de una impresora un grupo prefijado de usuarios en determinados intervalos horarios, optimizar los recursos y servicios de un SO para un determinado número de Tareas y usuarios, optimizar el funcionamiento de un servicio de urgencias con un número de empleados dado, etc. Como obviamente este nuevo criterio de gestión **no** es FIFO, la EDA que usa el `Gestor` de la aplicación **no** puede ser una Cola sino una tal que:

- `insertar(Trabajo t)` consiste en añadir `t` a la Colección atendiendo a su prioridad;
- `recuperar()` es obtener el Trabajo de la Colección cuyo tratamiento exige un mínimo tiempo de espera, el de máxima prioridad; por ello `recuperarMin` suele ser su nombre;
- `eliminar()` consiste en quitar de la Colección, una vez tratado, el Trabajo que requiere el mínimo tiempo de espera; por ello `eliminarMin` suele ser su nombre.

Una EDA con esta funcionalidad se denomina en la literatura **Cola de Prioridad** y, como se desprende fácilmente de su Especificación, cualquiera que sea el tipo de sus Datos éstos tienen que ser `Comparable`s para insertarlos, recuperarlos y eliminarlos en base a su prioridad; nótese también que una Cola de Prioridad se comporta como una simple Cola cuando algunos de sus Datos tienen la misma prioridad.

Ya para concluir resulta de interés presentar algunos datos que ilustren cómo influye la elección de la Representación de una EDA en la eficiencia de su Implementación; se usará el caso de la recién introducida Cola de Prioridad, una EDA cuyas operaciones tienen un coste que para una talla dada `x` varía en función de su Representación como indica la siguiente tabla:

Representación (tipo)	coste promedio de `insertar`	coste promedio de `recuperarMin`	coste promedio de `eliminarMin`
Lista Enlazada Ordenada (Lineal)	lineal con `x`	constante	constante
Árbol Binario de Búsqueda (Jerárquica Enlazada)	logaritmo de `x`, (pero `O(x)`)	logaritmo de `x` (pero `O(x)`)	logaritmo de `x` (pero `O(x)`)
Montículo Binario (Jerárquica Contigua)	constante (pero `O(log x)`)	constante	logaritmo de `x`

2. Diseño de la Jerarquía Java de una EDA

Como ya se ha comentado, la Relación que guardan las Implementaciones de una EDA con su Modelo es Jerárquica; por tanto es obvio que una EDA se describe en Java mediante una Jerarquía compuesta por una clase Base o Raíz que describe el Modelo de la EDA, i.e. su Modelo Java, y cada una de las Derivadas de ésta que describe una Implementación de la EDA (Contigua, Enlazada o mixta), i.e. cada una de sus Implementaciones Java. Ahora bien, se necesita algún criterio más de diseño para poder implementar la Jerarquía Java de una EDA; en concreto, y como marcan las reglas de la Herencia, se debe establecer básicamente el tipo concreto de clase Java que se utilizará como Raíz: si simplemente `public` como `Throwable` o `public` y `abstract` como `Figura` y `Number` o `interface` y genérica como `Comparable`.

Así, pensando en el posterior grado de reutilización de una EDA **la primera condición a imponer a su Modelo Java es que sea una clase genérica**, por lo que sus distintas Implementaciones Java también lo serán. Por ilustrar este primer criterio de diseño con un caso que resulte familiar, supóngase por ejemplo que un programador tiene que diseñar la Jerarquía Java de la Cola que utilizará siempre que tenga que realizar la gestión FIFO de una Colección de Trabajos -Ficheros a imprimir tras una sesión de prácticas, Tareas de un Sistema Operativo (SO), Entradas de cine a expedir en taquilla, Pacientes a visitar en una consulta médica, etc. Claramente la solución no pasa por diseñar tantos Modelos Java de Cola como tipos específicos de Trabajos se tengan que gestionar -`Ficheros`, `TareasSO`, `TicketsCine`, `Pacientes`, etc.- sino tan sólo uno genérico `Cola<E>` que pueda reutilizar convenientemente instanciado -`Cola<Fichero>`, `Cola<TareaSO>`, `Cola<TicketCine>`, `Cola<Paciente>`, etc.

Pensando también que el Modelo de una EDA debe ser por definición independiente de sus Implementaciones **la segunda condición a imponer a su Modelo Java es que sea una clase** `interface`. Así por ejemplo, sea la siguiente interfaz genérica la que se obtiene transcribiendo a Java el Modelo de una Cola presentado en el apartado anterior:

```
/** Cola<E>: Base de la Jerarquía que representa el Modelo de una Cola de
  * Elementos de tipo E, o describe en Java las operaciones de una Cola **/
public interface Cola<E> {
   /** inserta el Elemento e al final de una Cola, o lo encola **/
   void encolar(E e);
   /** SII !esVacia(): obtiene el primer Elemento insertado en una Cola **/
   E primero();
   /** SII !esVacia(): obtiene y elimina de una Cola el primer Elemento insertado **/
   E desencolar();
   /** comprueba si una Cola está vacía **/
   boolean esVacia();
}
```

Entonces, por definición de `interface` Java,

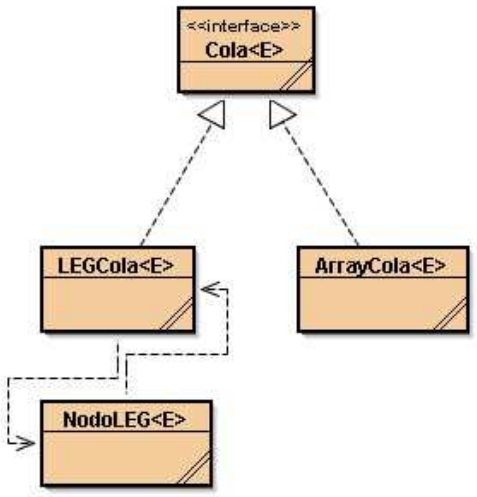
- `Cola` especifica en Java las operaciones de una Cola sin dar ningún detalle sobre la Representación en memoria de sus Datos ni sobre su implementación;
- a su vez, cualquier clase que declare implementar `Cola` no sólo debe sobrescribir por fuerza sus métodos sino que también, para poder hacerlo, se ve obligada a declarar al menos tantos atributos como requiera la Representación que realice de sus Datos en memoria; así por ejemplo, si la clase `ArrayCola` implementa la interfaz `Cola` mediante un `array` entonces su estructura mínima sólo puede ser la siguiente:

```
/** ArrayCola<E>: subclase de la Jerarquía que representa la
  * Implementación Contigua, sobre un array, de una Cola **/
public class ArrayCola<E> implements Cola<E> {
  // una Representación Contigua de una Cola exige como atributos mínimos:
  protected E elArray[];
  protected final static int CAPACIDAD_POR_DEFECTO = ...
  // en base a sus atributos, los métodos que implementa son como mínimo:
  //    (a) un constructor por defecto, que construye una Cola vacía
  public ArrayCola(){...}
  //    (b) los que hereda y, por tanto, sobrescribe de Cola
  public void encolar(E e){...}
  public E desencolar(){...}
  public E primero(){...}
  public boolean esVacia(){...}
}
```

  Aunque se detallará en un tema posterior, conviene señalar ahora que **un diseño eficiente** de `ArrayCola` **exige incorporar nuevas componentes a la clase**, atributos como `talla` y métodos auxiliares como `duplicarArray`, y también sobrescribir el método `toString` de `Object`, que no aparece explícitamente en la `interface Cola`.

Cuestión: tomando como ejemplo `ArrayCola`, obténgase la estructura mínima que debe tener la clase genérica `LEGCola` para ser una Implementación Enlazada de una Cola.

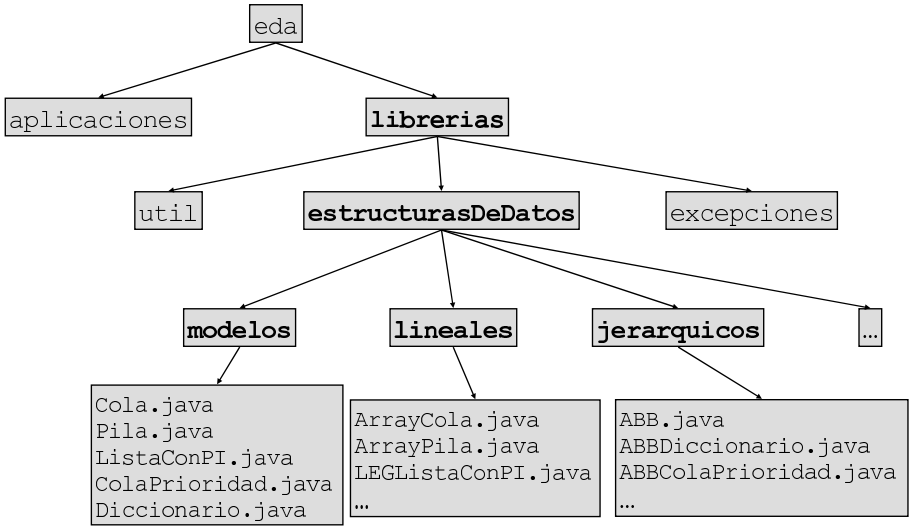
Por todo lo anterior, la conclusión que cierra este apartado es la siguiente: **para que el diseño de una EDA en Java refleje en todos sus extremos su definición y al mismo tiempo favorezca al máximo su reutilización resulta imprescindible representar su Modelo mediante una `interface` genérica y cada una de sus Implementaciones en una clase que implemente tal `interface`**, como resume la siguiente figura para una Cola.



Ejercicio propuesto: usando los mismo pasos que para una Cola, diséñese una Jerarquía Java de Cola de Prioridad cuya `interface ColaPrioridad` tenga como implementación Enlazada una versión conveniente de la clase `LEGOrdenada` presentada en el tema `4`.

3. Organización de Jerarquías de EDAs en librerías BlueJ

Todas las Jerarquías Java de una EDA que se diseñen este curso estarán contenidas en un subpaquete de `librerias` de `eda` denominado `estructurasDeDatos`; a su vez, y cara a su posterior reutilización, las componentes de estas Jerarquías se situarán según su tipo en uno u otro de los paquetes de `estructurasDeDatos` que se muestran en la siguiente figura.



Así, se ubicarán en el paquete `modelos` todas las interfaces que representen el Modelo de una EDA, en el paquete `lineales` todas sus implementaciones Lineales (`ArrayCola` y `LEGCola` de `Cola`, `LEGOrdenada` de `ColaPrioridad` y `Diccionario`, etc.), en el paquete `jerarquicos` las implementaciones según un Árbol de `ColaPrioridad` y `Diccionario`, en el paquete `noLineales` la implementación según una Tabla Hash de `Diccionario`, `...`

Finalmente, todas las Excepciones de Usuario que lancen las clases de `estructurasDeDatos` estarán en el subpaquete `excepciones` de `librerias` de `eda`.

4. Uso de una EDA: reutilización de su Jerarquía Java

Como sucede con cualquier clase Java, la Jerarquía de una EDA se puede reutilizar convenientemente instanciada para diseñar nuevas clases bien vía `import`, bien vía Composición (TIENE UN(A)) o bien vía Herencia (ES UN(A)). En lo que sigue se plantean diversos ejemplos de uso de la Jerarquía Cola antes diseñada, aunque empleando implementaciones eficientes de `ArrayCola` y `LEGCola`, para introducir e ilustrar gradualmente las situaciones y condiciones que conlleva cada una de estas formas de uso de una EDA.

Uso de la Jerarquía Java de una EDA vía `import`

Se ha escrito el siguiente programa para crear y manipular vía `import` una `Cola` de `Integer` `q`, esto es un objeto de tipo Estático `Cola<Integer>` y tipo Dinámico `ArrayCola<Integer>`:

```
package ejemplos.tema6;
import librerias.estructurasDeDatos.modelos.*;
import librerias.estructurasDeDatos.lineales.*;
public class TestEDACola {
  public static void main(String args[]){
    Cola<Integer> q = new ArrayCola<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 métodos 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 vacía, q = ["+q.toString()+"]");
  }
}
```

Aún siendo su código muy simple, la compilación de `TestEDACola` permite señalar tres reglas básicas sobre el uso correcto de una EDA. La primera de ellas es que **la reutilización de su Jerarquía Java exige como mínimo instanciar dos clases genéricas**, la `interface` que representa su Modelo (`Cola`, ubicada en `librerias.estructurasDeDatos.modelos`) y una de las clases que la implementa (`ArrayCola`, ubicada en `librerias.estructurasDeDatos.lineales`); en caso de no respetarla, bien por omitir el `import` de acceso a éstas o bien por no instanciar sus variables de tipo debidamente, el intérprete Java lo advierte con un error de compilación.

Cuestión: indíquese el error de compilación que se produce al eliminar el `import` de `modelos` de `TestEDAJava` y si es el mismo que provoca eliminar el `import` de `lineales`. Determínese asimismo el tipo de error que provocan las siguientes instrucciones: «`Cola q = new ArrayCola();`», «`Cola q = new ArrayCola<Integer>();`» y «`Cola<Integer> q = new ArrayCola();`».

La segunda regla que la compilación de `TestEDACola` deja patente es que **única y exclusivamente se puede acceder a los atributos `protected` de una Implementación Java de una EDA vía Herencia**, ni siquiera cuando se tiene acceso al código de dicha clase; así, como `TestEDACola` **no** ES UNA `Cola` **ni** tampoco ES UN `ArrayCola`, al compilar «`System.out.println("Creada una Cola con "+q.talla+" Integer, q = [ "+q+" ]");`» el intérprete Java lanza el error «`cannot find symbol - variable talla`» para advertir que el atributo `talla` de `ArrayCola` no es visible, ni por tanto accesible, desde de su método `main`.

Finalmente hay que substituir `q.talla` por `q.talla()` en `TestEDACola` para que su compilación ilustre con claridad la tercera de las reglas a observar: **vía `import` sólo se pueden reutilizar (aplicar vía `.`) los métodos de la `interface` que representa al Modelo de una EDA, incluidos los que hereda de `Object`**, y por tanto ningún otro de los métodos públicos que pueda definir cualquier implementación de esta interfaz. Así, al compilar «`System.out.println("Creada una Cola con "+q.talla()+" Integer, q = [ "+q+"]");`» el intérprete Java lanza el error «`cannot find symbol - method talla()`» para advertir que no hay un método con el mismo nombre en la `interface Cola`, el tipo Estático de `q`, por lo que no puede aplicarle el de la clase de su tipo Dinámico `ArrayCola`, tanto si está definido en ella como sino. Observar también que la aplicación del método `toString()` a la variable `q` no es una excepción a esta regla: al tratarse de un método heredado de `Object` siempre se puede ejecutar vía Enlace Dinámico, tanto si se sobrescribe en `ArrayCola` como sino.

Uso de la Jerarquía Java de una EDA vía Composición

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

Ejercicio propuesto: se han diseñado las siguientes clases 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 primera de ellas para que no contenga errores de compilación.

```
package ejemplos.tema6;
import librerias.estructurasDeDatos.lineales.*;
public class GestorDePacientes {
  private Cola<Paciente> q;  private double horaCita;
  private static final int MAXIMO_DIARIO_PACIENTES = 40;
  private static final double HORA_INICIO_CONSULTA = 9.00, TIEMPO_MEDIO_VISITA = 0.15;
  public GestorDePacientes(){
    q = new ArrayCola<Paciente>();  horaCita = HORA_INICIO_CONSULTA;
  }
  public String darCita(Paciente x){
    String res = "Espere un momento; consulto si le pueden atender mañana ... ";
    boolean aceptado = ( q.talla <= MAXIMO_DIARIO_PACIENTES );
    if ( !aceptado ) res += "\nLo siento. Mañana no podemos atenderle";
    else{ q.encolar(x);
          res += "\nConfirmado, le esperamos mañana a las "+String.format("%2.2f", horaCita);
          horaCita += TIEMPO_MEDIO_VISITA;
          if ( horaCita - Math.round(horaCita) < 0.0 )   horaCita = Math.round(horaCita);
    }
    return res;
  }
  public String toString(){
    return "Historiales de sus "+q.talla+" Pacientes de mañana en orden de visita\n"+q;
  }
}
public class Paciente{
    private int numero;
    public Paciente(int n){ numero = n; }
    public String toString(){ return "Paciente número "+numero+" | " ;}
}
```

Uso de la Jerarquía Java de una EDA vía Herencia

Esta tercera forma de uso de una EDA se emplea en situaciones análogas a las que plantea el cálculo de la talla de una Cola en `TestEDACola` o `GestorDePacientes`, i.e. en clases que reutilizan vía `import` o Composición una EDA dada **pero** cuyo diseño exige la aplicación de operaciones que **no** figuran en su Modelo actual; en estos casos la única solución general y eficaz, i.e. la mejor en términos de Reutilización del Software, consiste en añadir vía Herencia a la Jerarquía Java de la EDA dos nuevas clases: una `interface` Derivada de su Raíz actual para describir en Java la nueva funcionalidad de la EDA y una clase que implementa esta nueva `interface` extendiendo alguna de las implementaciones disponibles de su Raíz actual. Así por ejemplo, para el caso concreto de la Jerarquía `Cola` se deben dar los siguientes pasos para aplicar el método `talla()` a la `Cola q` de `TestEDACola` o `GestorDePacientes`:

1. Establecer la Especificación y perfil del método que describe en Java la nueva operación con la que se ampliará la funcionalidad de la EDA Cola, o método enriquecedor de Cola:

```
/** obtiene la talla de una Cola **/
public int talla();
```

2. Diseñar una `interface` que represente en Java el Modelo de una Cola con `talla`; para ello basta definir la siguiente **sub**`interface` de `Cola`, una nueva interfaz que hereda los métodos de `Cola` y sólo define explícitamente el método enriquecedor `talla()`:

```
package librerias.estructurasDeDatos.modelos;
/** Una ColaPlus ES UNA Cola con talla() **/
public interface ColaPlus<E> extends Cola<E> {
    /** obtiene la talla de una Cola **/
    int talla();
}
```

3. Diseñar una clase que implemente la sub`interface ColaPlus` definida, o que represente en Java una Implementación de una Cola con `talla`; para ello basta definir la Derivada de `ArrayCola` que se presenta a continuación, una nueva clase Java que implementa el único método `talla()` de `ColaPlus` en base a las componentes que hereda de `ArrayCola`:

```
package librerias.estructurasDeDatos.lineales;
import librerias.estructurasDeDatos.modelos.*;
/** Un ArrayColaPlus ES UN ArrayCola que implementa ColaPlus **/
public class ArrayColaPlus<E> extends ArrayCola<E>
                              implements ColaPlus<E>, Cola<E> {
    /** obtiene la talla de una Cola **/
    public final int talla(){ ... }
}
```

   Cabe observar ahora que, por heredar `ArrayColaPlus` tanto los atributos como los métodos de `ArrayCola`, la implementación de `talla()` puede realizarse de dos formas:

   a) **usando únicamente los métodos que hereda de** `ArrayCola`, i.e.

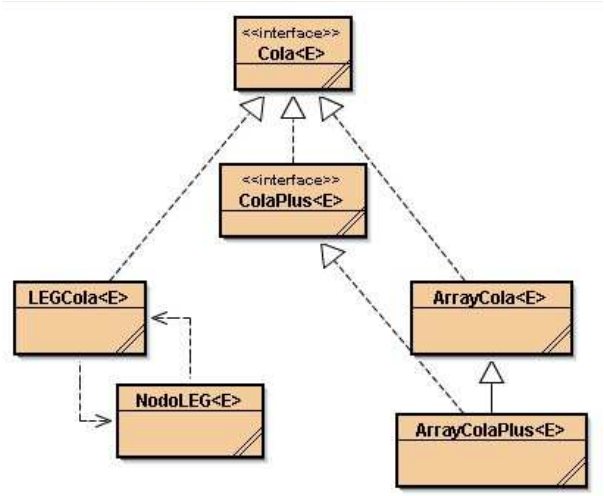
```
public int talla(){
   int res = 0;
   while ( !this.esVacia() ){ E primero = this.desencolar(); res++; }
   return res;
}
```

   b) **usando únicamente los atributos** `protected` **que hereda de** `ArrayCola`, i.e.

```
public int talla(){ return super.talla; }
```

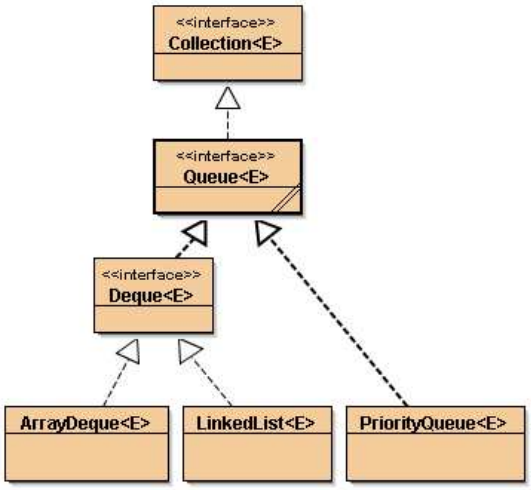
Obviamente la segunda implementación presentada para `talla()` es mucho más eficiente que la primera y, además, calcula la talla de la Cola **sin vaciarla**; sin embargo también es menos general porque usa el atributo `protected talla` de la clase `ArrayCola`, lo que no siempre es posible en un entorno Java donde el Principio de Ocultación de la Información se suele mantener a rajatabla para garantizar la Reutilización del Software.

4. Tras ampliar la Jerarquía `Cola` como ilustra la siguiente figura, 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`.



5. EDAs en el estándar de Java: características y uso de la Jerarquía `Collection`

La Jerarquía `Collection` de `util` es la herramienta principal que proporciona el lenguaje Java en su *Java Collections Framework* para la representación y manipulación de Colecciones (*Collection*) de Datos. Aunque los objetivos y características de esta vasta Jerarquía se describen en detalle en **http://java.sun.com/docs/books/tutorial/collections/index.html**, a modo de introducción se puede decir que esta Jerarquía juega en el caso de las EDAs el mismo papel que la Jerarquía `Throwable` en el caso de los Fallos de Programación: a partir de su Raíz `Collection`, la interfaz genérica que describe la funcionalidad básica de una Colección de objetos Java, se representan en el estándar de Java vía Herencia (simple o múltiple) los Modelos e Implementaciones más eficientes de las EDAs más frecuentemente utilizadas en el desarrollo de aplicaciones. Entre las EDAs representadas en `Collection` se encuentran todas las que se estudian en esta asignatura **pero**, obviamente, organizadas de manera distinta a la propuesta en el tercer apartado de este tema y con denominaciones y operaciones que no siempre coinciden con las de sus traducciones al Inglés. Así por ejemplo, las tres Jerarquías Java disjuntas con las que se representan en esta asignatura la EDAs Pila, Cola y Cola de Prioridad aparecen integradas en el estándar de Java en la (sub)Jerarquía de `Collection` que, a grandes rasgos, muestra la siguiente figura.



La Raíz de esta Jerarquía, la interfaz `Queue`, es la Derivada directa de `Collection` que representa a todas las EDAs en las que se puede borrar y recuperar únicamente el Dato que ocupa su cabeza (*head*), el primero de sus Datos según un cierto orden; dicho en otros términos, `Queue` es en el estándar de Java el Modelo parcial o "Factor-Común" de Pila (orden LIFO), Cola (orden FIFO) y Cola de Prioridad (orden de prioridad): con métodos como `poll` y `peek` especifica las operaciones que `borrar` y `recuperar` el último Dato insertado en una Pila, el primer Dato insertado en una Cola y el Dato de máxima prioridad en una Cola de Prioridad; sin embargo, sus métodos de inserción **ni** especifican la inserción de `x` en cabeza de Pila para mantener el orden LIFO, **ni** en fin (*tail*) de Cola para mantener el orden FIFO **ni** en base a su prioridad en Cola de Prioridad, que además exigiría que `Queue` implementara `Comparable`. Por todo ello, considerando que sólo los Datos de una Cola de Prioridad deben ser `Comparable` y que el orden de una Pila es el inverso al de una Cola, la clase `PriorityQueue` y la subJerarquía `Deque` especializan vía Herencia a la interfaz `Queue` para representar en el estándar de Java, respectivamente, la Implementación de una Cola de Prioridad según un Montículo Binario y el Modelo e Implementaciones de una Pila y una Cola. En concreto,

- `PriorityQueue` ES UNA `Queue` tal que `implements Queue` para Datos `Comparable` que se insertan según su prioridad y de forma que en su cabeza siempre figure el Dato de máxima prioridad; usando sus métodos `add(x)`, `peek()` y `poll()` se pueden implementar respectivamente los métodos `insertar(x)`, `recuperarMin()` y `eliminarMin()` del Modelo `ColaPrioridad` de esta asignatura;
- la interfaz `Deque` ES UNA `Queue` con dos extremos, o una *Double ended queue*, tal que `extends Queue` para definir entre otros los métodos de inserción en su final y en su cabeza, respectivamente `addLast(x)` y `addFirst(x)`; así por ejemplo, si a un objeto `Deque` se le aplican los métodos `peek()`, `poll()` y `addLast(x)` funciona como una Cola **pero** si se le aplican `peek()`, `poll()` y `addFirst(x)` entonces funciona como una Pila. Obviamente, sus Derivadas directas `ArrayDeque` y `LinkedList` son las Implementaciones eficientes de las EDAs Pila y Cola y equivalen a las de los Modelos `Cola` y `Pila` de esta asignatura.

En base a esta descripción de `Queue` no resulta muy difícil proponer varios ejemplos que demuestran que, en general, un programador puede simplificar los procesos de diseño y uso de una EDA reutilizando aquellas clases de `Collection` que más le convengan. Así, y como haría con las clases de `Exception` para diseñar las Excepciones de Entrada-Salida de una aplicación, si tiene que diseñar un programa como `TestEDACola` para crear y manipular una

Cola `q` de `Integer` puede ahorrarse el esfuerzo de diseñar la Jerarquía `Cola` y usar vía `import` la subJerarquía `Deque` de `Collection` como sigue:

```
package ejemplos.tema6;
import java.util.*;
public class TestEDAColaVDeque {
  public static void main(String args[]){
    Deque<Integer> q = new ArrayDeque<Integer>();
    System.out.println("Creada una Cola con "+q.size()+" Integer, q = [ "+q.toString()+"]");
    q.add(new Integer(10)); q.add(new Integer(20)); q.add(new Integer(30));
    System.out.println("La Cola de Integer actual es q = [ "+q.toString()+"]");
    System.out.println("Usando otros métodos para mostrar sus Datos el resultado es ... ");
    String datosQ = "";
    while ( q.size() != 0){
      Integer primero = q.peek();
      if ( primero.equals(q.poll()) ) datosQ += primero+" "; else datosQ += "ERROR ";
    }
    System.out.println(" el mismo, "+datosQ+", PERO q se vacía, q = ["+q.toString()+"]");
  }
}
```

Ejercicio propuesto: modifíquese la clase `GestorDePacientes` del apartado anterior, que TIENE UNA Cola `q` de `Paciente`, para que reutilice vía Composición la subJerarquía `Deque` de `Collection`.

Adicionalmente, y como haría con `Exception` para diseñar una Excepción de Usuario, un programador puede reutilizar vía Herencia la Jerarquía `Collection` para diseñar con las máximas garantías y el menor esfuerzo su propia Jerarquía Java de una EDA. Por ejemplo, en lugar de diseñar `ArrayCola` desde cero podría obtener una Implementación Contigua de `Cola` sin más que extender la clase `ArrayDeque` de `Collection` como sigue:

```
package librerias.estructurasDeDatos.lineales;
import librerias.estructurasDeDatos.modelos.*;
import java.util.*;
/** ArrayDequeCola<E>: subclase de la Jerarquía que representa la
  * Implementación Contigua, sobre un ArrayDeque, de una Cola **/
public class ArrayDequeCola<E> extends ArrayDeque<E> implements Cola<E>, Deque<E>, Queue<E>{
  // Hereda los atributos private de ArrayDeque, PERO NO tiene acceso a ellos
  // en base a la Especificación de ArrayDeque y Cola, los métodos que implementa son:
  //    (a) un constructor por defecto, que construye una Cola vacía
   public ArrayDequeCola(){ super(); }
  //    (b) los métodos que hereda de Cola, y que por tanto sobrescribe
   public void encolar(E e){ addLast(e); }
   public E desencolar(){ return poll(); }
   public E primero(){ return peek(); }
   public boolean esVacia(){ return ( size() == 0 ); }
}
```

Cuestión: supóngase que, en lugar de `ArrayCola`, en `TestEDAJava` se elige `ArrayDequeCola` como tipo Dinámico de `q`; como `ArrayDequeCola` hereda los métodos `size()` y `toString()` de `ArrayDeque`, ¿por qué motivo no se puede evaluar correctamente el resultado de la expresión «`q.size()`» pero sí el de la expresión «`q.toString()`»?