

# Tema 1

Estructuras de datos en Java

# Objetivos

---

- Definir las EDAs más frecuentemente utilizadas en programación, clasificándolas en base a los modelos de gestión de datos que representan
- Revisar las herramientas que ofrece Java para el diseño y uso eficientes de una EDA mediante ejemplos ilustrativos; en concreto, se hará especial hincapié en lo siguiente:
  - Definición y uso de la jerarquía Java de una EDA de tipo genérico, en su caso restringido por **Comparable**, combinando adecuadamente *herencia*, *polimorfismo* y *genericidad*
  - Uso y características de la Jerarquía **java.util.Collection**, la definición en el estándar de Java de las EDAs más frecuentemente utilizadas en programación

# Contenidos (4 sesiones aprox.)

---

1. Estructuras de Datos (EDAs): definición y clasificación
2. Diseño de una EDA en Java
  - 2.1. Jerarquía Java de una EDA: componentes y nomenclatura
  - 2.2. Criterios de diseño de las clases de una jerarquía. Ejemplos para las jerarquías lineales de **Pila** y **Cola**
3. Uso de la jerarquía Java de una EDA
4. EDAs en el estándar de Java: la jerarquía **Collection**
5. Lista con iterador: la jerarquía **ListaConPI**
6. Clases de tipo genérico restringido por **Comparable**: la jerarquía **ColaPrioridad**

# 1. Estructuras de datos

---

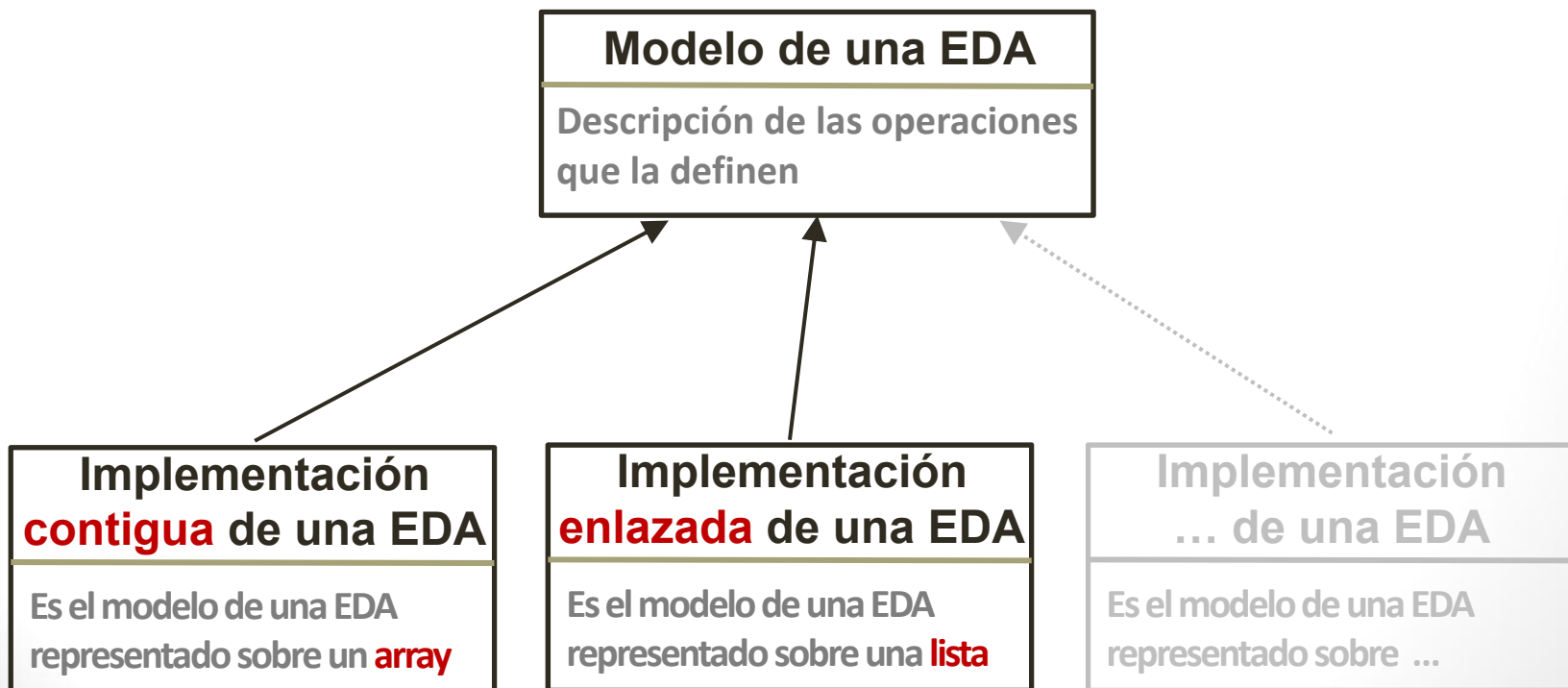
## *Definición*

- 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 describir dos 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:** *Map y Cola de Prioridad*

Gestión basada en la búsqueda dinámica de un dato dado (búsqueda por clave o por prioridad)

- **DE RELACIÓN:** *Grafo*

Gestión de datos que guardan entre sí una relación binaria

# 2. Diseño de una EDA en Java

---

## 2.1. Jerarquía Java de una EDA

- Una EDA se describe en Java mediante una jerarquía compuesta por:
  - Una **interface**, de tipo genérico, que describe su **modelo**
  - Cada clase genérica derivada del modelo, que describe cada una de sus **implementaciones**

### Ejemplo: jerarquía Java de una **Cola**

- `public interface Cola<E> {...}`  
`// Modelo Java`
- `public class ArrayCola<E> implements Cola<E> {...}`  
`// Implementación contigua`
- `public class LEGCola<E> implements Cola<E> {...}`  
`// Implementación enlazada`

# 2. Diseño de una EDA en Java

---

## 2.2. *Criterios de diseño del modelo*

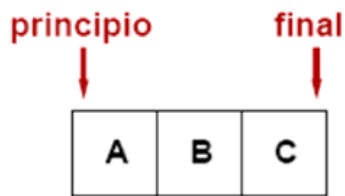
1. Definir el número mínimo de operaciones que permiten la gestión de sus elementos
2. Evitar en lo posible el lanzamiento de excepciones *comprobadas*, substituyéndolo por el uso de precondiciones
3. Establecer el coste máximo estimado de los métodos definidos



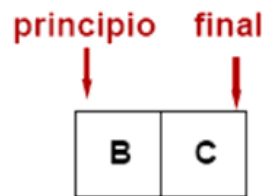
# 2. Diseño de una EDA en Java

## 2.2. Ejemplo: el modelo Cola

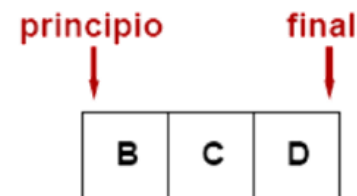
Una **Cola** es una colección de elementos que se gestionan siguiendo un criterio **FIFO**, i.e. permitiendo la consulta sólo del primero de ellos en orden de inserción:



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('D')** se inserta D al final la cola, en la que B sigue siendo el **primero()**

```
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)  
}
```

# 2. Diseño de una EDA en Java

## 2.2. Criterios de diseño de la implementación contigua de una Cola

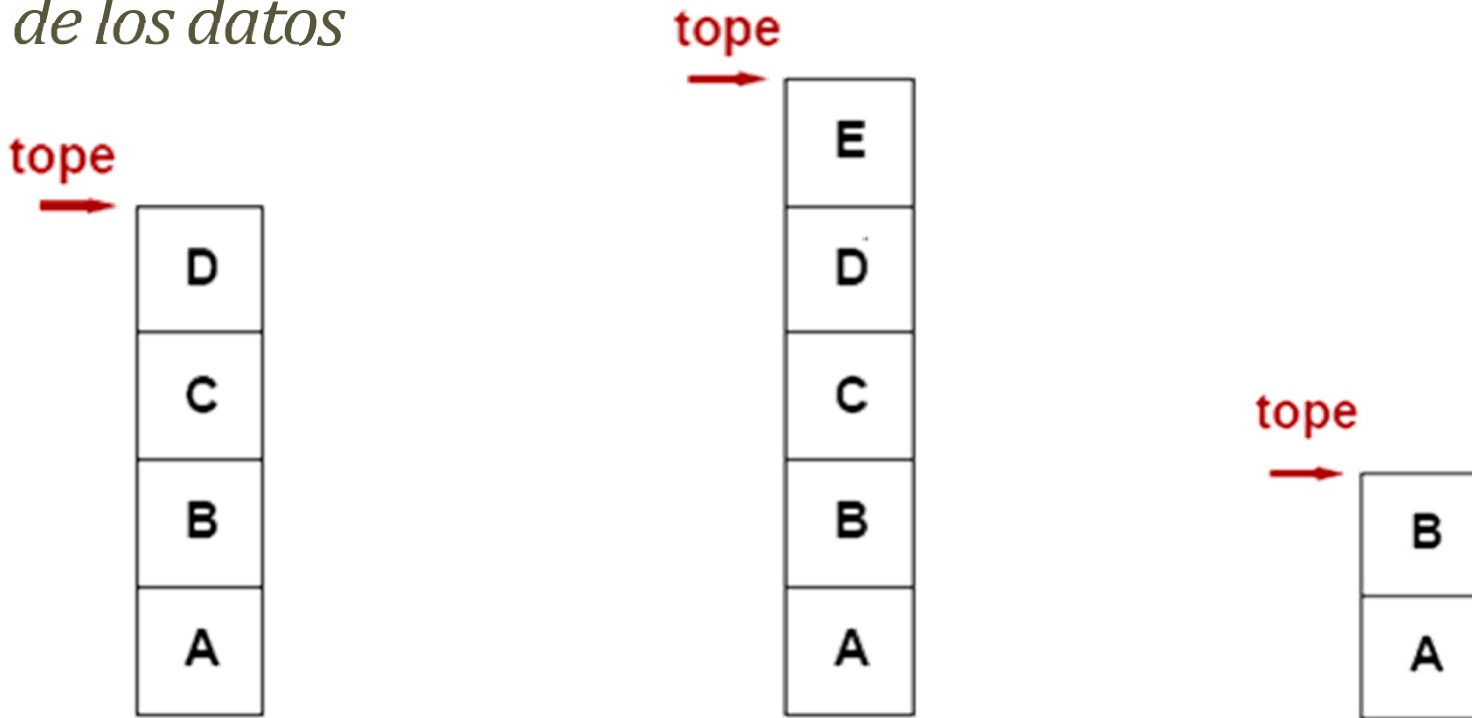
Un **ArrayCola<E>** es una clase que implementa la interfaz **Cola<E>**:

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 índice para el `final`, el `principio` y la `talla`
3. Sobrescribe el método `toString`

```
public class ArrayCola<E> implements Cola<E> {  
    // Atributos  
    protected E elArray[];  
    protected int final, principio, talla;  
    public ArrayCola() {...}           // Constructor  
    public void encolar(E e) {...}     // Métodos del modelo  
    ...  
    public String toString() {...}    // Método toString  
}
```

## 2. Diseño de una EDA en Java

### 2.2. El modelo Pila: operaciones para una gestión LIFO de los datos



Pila de cuatro datos de tipo *Character*, cuyo **tope()** ocupa D

Al **apilar('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

### 3. Uso de la jerarquía Java de una EDA

---

#### *Modalidades*

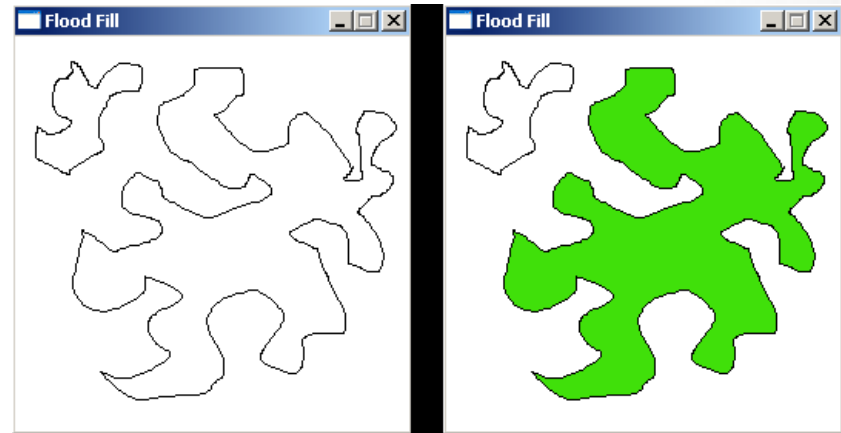
Como sucede con cualquier clase Java, la jerarquía de una EDA se puede reutilizar para diseñar nuevas clases ...

1. Vía composición ( TIENE UN )
2. Vía herencia ( ES UN )

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

### 3. Uso de la jerarquía Java de una EDA *vía composición (TIENE UN)*

- Utilización de la EDA, correctamente instanciada, para una aplicación concreta
- Ejemplo: rellenar una figura



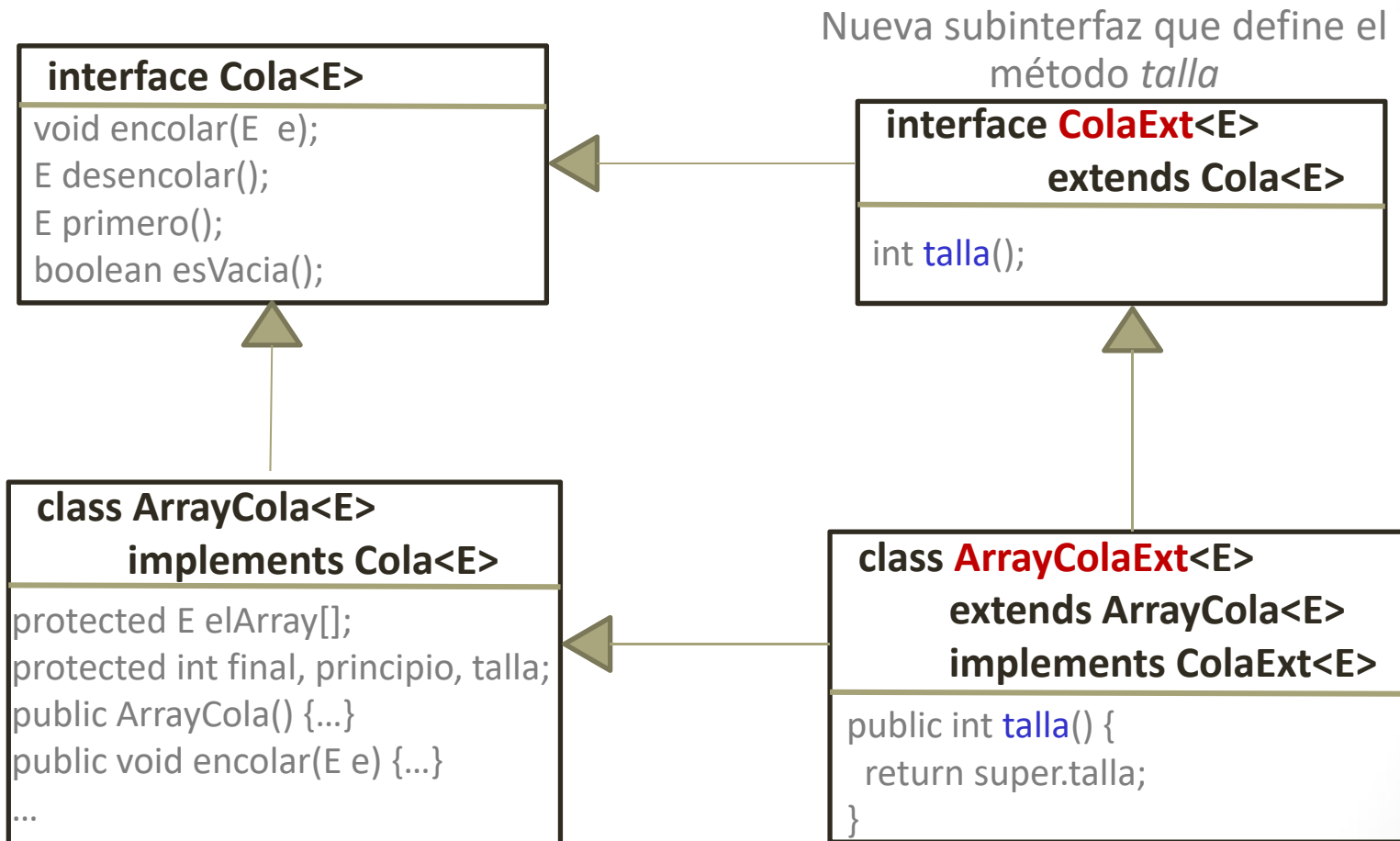
```
public static void rellenar(BufferedImage img, Point p,
    int colorFondo, int nuevoColor) {
    Cola<Point> c = new ArrayCola<Point>();
    c.encolar(p);
    while (!c.esVacía()) {
        p = c.desencolar();
        if (img.getRGB(p.x, p.y) == colorFondo) {
            img.setRGB(p.x, p.y, nuevoColor);
            if (img.getRGB(p.x + 1, p.y) == colorFondo)
                c.enconlar(new Point(p.x + 1, p.y));
        }
    }
}
```

### 3. Uso de la jerarquía Java de una EDA *vía herencia (ES UN)*

---

- Se emplea en aplicaciones cuyo diseño exige el uso 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; por ejemplo, un método *talla()* para el caso de la *Cola*
  - Diseñar una clase que implemente dicha subinterfaz. Sólo es necesario implementar la nueva funcionalidad (el método *talla()* en nuestro ejemplo)

### 3. Uso de la jerarquía Java de una EDA *vía herencia (ES UN)*



¿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)*

---

- **Ampliación vía herencia**, utilizando los métodos del modelo

```
public class ArrayColaExt<E> extends ArrayCola<E>
    implements ColaExt<E> {

    public int talla() {
        E marca = null;
        this.encolar(marca);
        return talla(marca);
    }

    private int talla(E marca) {
        E tmp = this.desencolar();
        if (tmp == marca) return 0;
        this.encolar(tmp);
        return 1 + talla(marca);
    }
}
```

**PROBLEMA:** esta solución suele ser menos eficiente

**VENTAJA:** esta solución es siempre posible



# 4. EDAs en el estándar de Java

## Las jerarquías *Collection* y *Map*

- Java contiene en su librería *java.util* las jerarquías *Collection* y *Map* para la representación y manipulación de colecciones de datos

Interfaces	Implementaciones				
	Tabla hash	Array	Árbol Balanceado	Lista enlazada	Tabla hash + lista enlazada
<b>Set</b>	HashSet		TreeSet		LinkedHashSet
<b>List</b>		Vector/ ArrayList		LinkedList	
<b>Deque</b>		ArrayDeque		LinkedList	
<b>Map</b>	HashMap/ Hashtable		TreeMap		LinkedHashMap

**Set:** manipulación de conjuntos

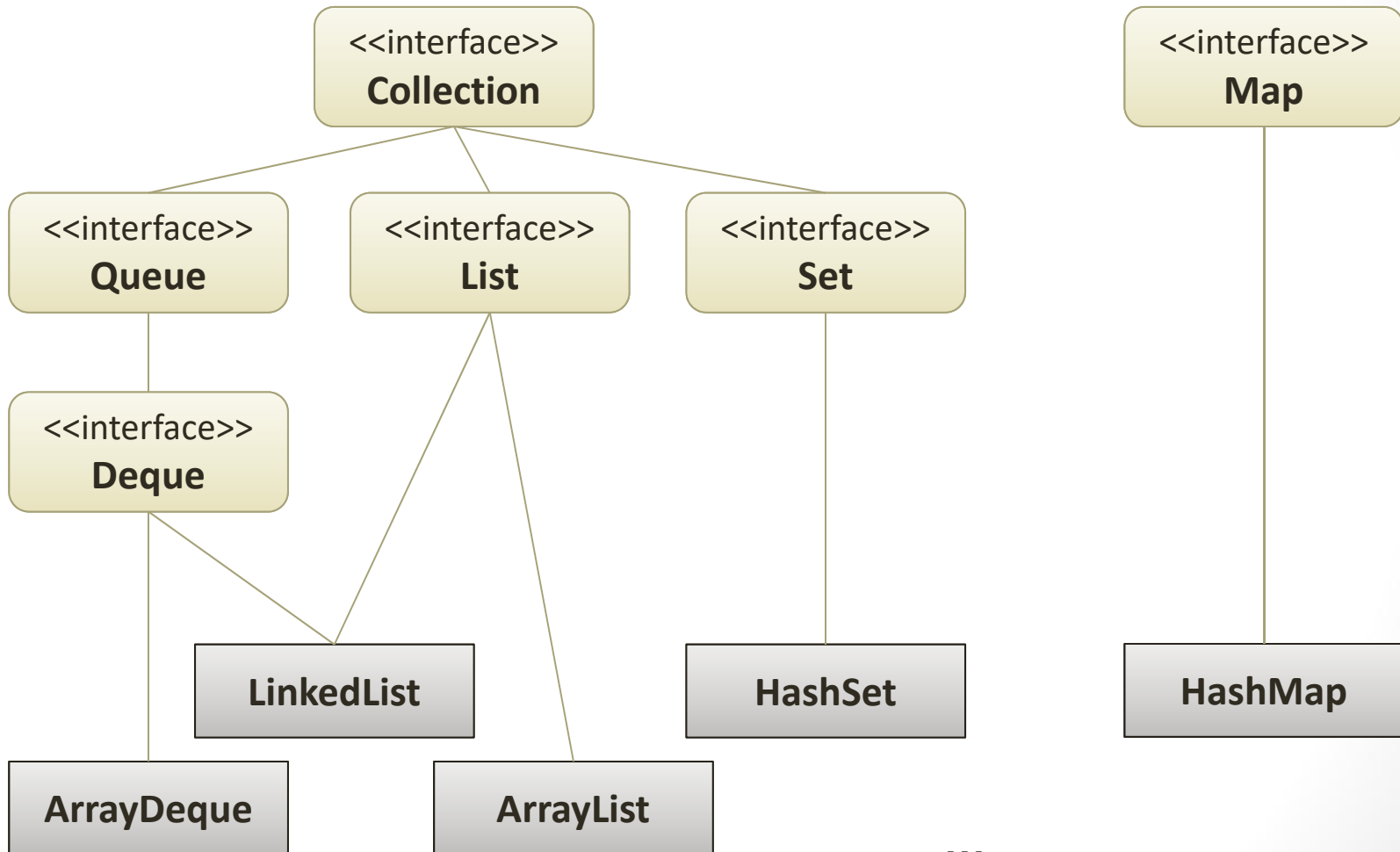
**List:** colección ordenada / secuencia

**Deque:** fusión de los modelos *Pila* y *Cola*

**Map:** diccionario

# 4. EDAs en el estándar de Java

## Las jerarquías *Collection* y *Map*



# 5. Lista con iterador

---

## *Punto de interés de una lista*

- **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 el ***punto de interés (PI)***
  - El acceso al dato que ocupa el PI se realiza en tiempo constante
  - La posición que ocupa un dato en la *lista* deja de ser un parámetro en las operaciones de acceso secuencial pues:
    - Inicialmente el PI está situado sobre el primer dato de la *lista*
    - El recorrido de la lista se realiza desplazando el PI al siguiente elemento
    - Al **insertar**, **recuperar** o **eliminar** un dato el PI permanece **inamovible**
    - Si una lista está **vacía** o bien el acceso secuencial ha llegado al **final** no hay dato alguno que ocupe el PI

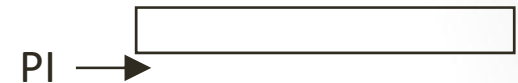
# 5. Lista con iterador

*Ejemplo: inserción al final de una ListaConPI*

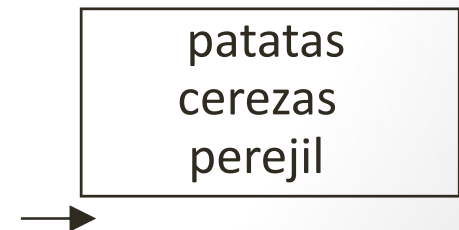
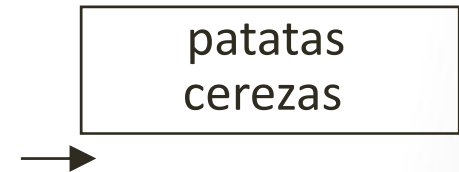
```
lpi.insertar("patatas");
```

```
lpi.insertar("cerezas");
```

```
lpi.insertar("perejil");
```



*Se inserta antes del PI*



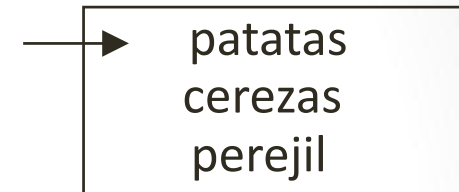
*Al insertar en una ListaConPI, el PI permanece inalterado*

# 5. Lista con iterador

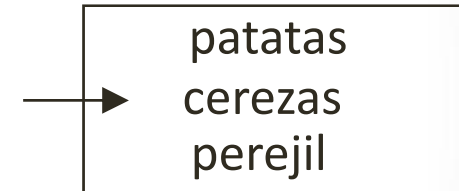
## Ejemplo: borrado en una ListaConPI

```
lpi.inicio();
```

*inicio() sitúa el PI sobre  
el primer elemento de la lista*



```
lpi.siguiente();
```



```
System.out.println(lpi.recuperar());
```

*Imprime "cerezas"*

```
lpi.eliminar();
```

*eliminar() borra el elemento sobre el  
que está el PI. El PI queda apuntando  
al siguiente elemento de la lista*



```
lpi.eliminar();
```



# 5. Lista con iterador

*Ejemplo: inserción al principio de una ListaConPI*

```
lpi.inicio();
```



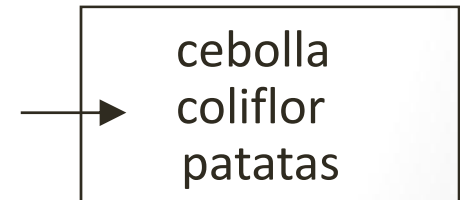
```
lpi.insertar("coliflor");
```



```
lpi.inicio();
```



```
lpi.insertar("cebolla");
```



# 5. Lista con iterador

## Diseño del modelo ListaConPI

```
public interface ListaConPI<E> {  
    /** inserta e antes del PI, que permanece inalterado **/  
    void insertar(E e); //  $\Theta(1)$   
    /** SII !esFin(): elimina el dato que ocupa su PI **/  
    void eliminar(); //  $\Theta(1)$   
    /** SII !esFin(): devuelve el dato que ocupa el PI **/  
    E recuperar(); //  $\Theta(1)$   
    /** sitúa el PI de una Lista en su inicio **/  
    void inicio(); //  $\Theta(1)$   
    /** SII !esFin(): avanza el PI de una Lista **/  
    void siguiente(); //  $\Theta(1)$   
    /** comprueba si el PI está tras su último dato **/  
    boolean esFin(); //  $\Theta(1)$   
    /** comprueba si una Lista Con PI está vacía **/  
    boolean esVacía(); //  $\Theta(1)$   
    /** sitúa el PI tras el último elemento **/  
    void fin(); //  $\Theta(1)$   
    /** devuelve la talla de una Lista Con PI **/  
    int talla(); //  $\Theta(1)$   
}
```

## 6. Clases de tipo restringido por Comparable

### Motivación

¿Cómo resolverías los siguientes problemas?

- Mantener **ordenados por antigüedad** los empleados de una empresa; supón disponibles las clases *Empleado* y *Empresa* y que la clase *Empresa* tiene una *ListaConPI<Empleado>*
- Mantener **ordenadas por área** las figuras de un grupo; supón disponibles las clases *Figura* y *GrupoDeFiguras* y que la clase *GrupoDeFiguras* tiene una *ListaConPI<Figura>*

Definir un método de inserción en orden en *Empresa* y *GrupoDeFiguras*, pues el de *ListaConPI* no sirve

¿Qué problema presenta esta solución?



## 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);
}
```

**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.)


## 6. Clases de tipo restringido por Comparable

### La interfaz Comparable

- Java proporciona un criterio de comparación genérico, el método `compareTo(otro)` de la interfaz `Comparable`
- El método **`compareTo`** no está definido en la clase `Object`
  - No se puede usar exactamente igual que **`equals`**
  - Es el único método de la interfaz `java.lang.Comparable`, el modelo estándar y genérico que proporciona Java para la comparación de dos objetos de tipo genérico

```
public interface Comparable<T> {  
    public int compareTo(T x);  
}
```

Devuelve un número



$\left\{ \begin{array}{ll} < 0 & , \text{ si } \text{this} < x \\ > 0 & , \text{ si } \text{this} > x \\ == 0 & , \text{ si } \text{this} \text{ es igual a } x \end{array} \right.$

**Nota:** se recomienda que `a.compareTo(b) == 0` sea equivalente a `a.equals(b)`

## 6. Clases de tipo restringido por Comparable

### Ejemplo: La clase Figura comparable

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

# Bibliografía

---

- Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java (4th edition)*. John Wiley & Sons, Inc., 2005.
  - Capítulos 1, 2, 3, y 5, sobre conceptos generales de Java y EDAs. Capítulo 6, sobre Listas e Iteradores Java. Secciones 1.3, 2.1 y 2.2 del Capítulo 8, sobre la EDA Cola de Prioridad.
- Weiss, M.A. *Estructuras de Datos en Java*. Adisson-Wesley, 2000.
  - Apartados 1, 3 y 8 del Capítulo 6, sobre EDAs y su representación en Java
- *The Java™ Tutorials: Collections*
  - <http://java.sun.com/docs/books/tutorial/collections/index.html>
- La subjerarquía *Queue de Collection* en el API de Java
  - <http://java.sun.com/javase/6/docs/api/java/util/Queue.html>