# UD10.- Estructuras de datos II

Módulo: Programación
1.º DAM

Unión Europea
Fondo Social Europeo
*El FSE invierte en tu futuro*

# CONTENIDOS

- Estructuras estáticas *versus* estructuras dinámicas
- Interfaces
- Colecciones

# Estructuras estáticas *versus* Estructuras dinámicas

- En prácticamente todos los lenguajes de programación existen estructuras para almacenar colecciones de datos:
  - Un conjunto de datos identificados por un único nombre.
- Estructuras **estáticas**
  - Se tiene que saber el número de elementos que formarán parte de la colección en tiempo de compilación.
  - Ejemplo: Arrays
- Estructuras **dinámicas**
  - El número de elementos se decide y modifica en tiempo de ejecución.
  - El número de elementos es ilimitado.
  - Son clásicas de la programación. Algunos ejemplos son las colas, las pilas, las listas enlazadas, los árboles, los grafos, etc.

# Interfaces (I)

- Es un elemento de Java que indica qué se ofrece, pero no como se hace. Define un COMPORTAMIENTO.
- Una interfaz proporciona:
  - Valores constantes, que son variables “public static final”.
  - Métodos, que son “public”.
  - NO incluye constructores.
- Las interfaces:
  - Se pueden extender con nuevas constantes y/o métodos
  - Se pueden implementar totalmente → clases.
  - Se pueden implementar parcialmente → clases abstractas.

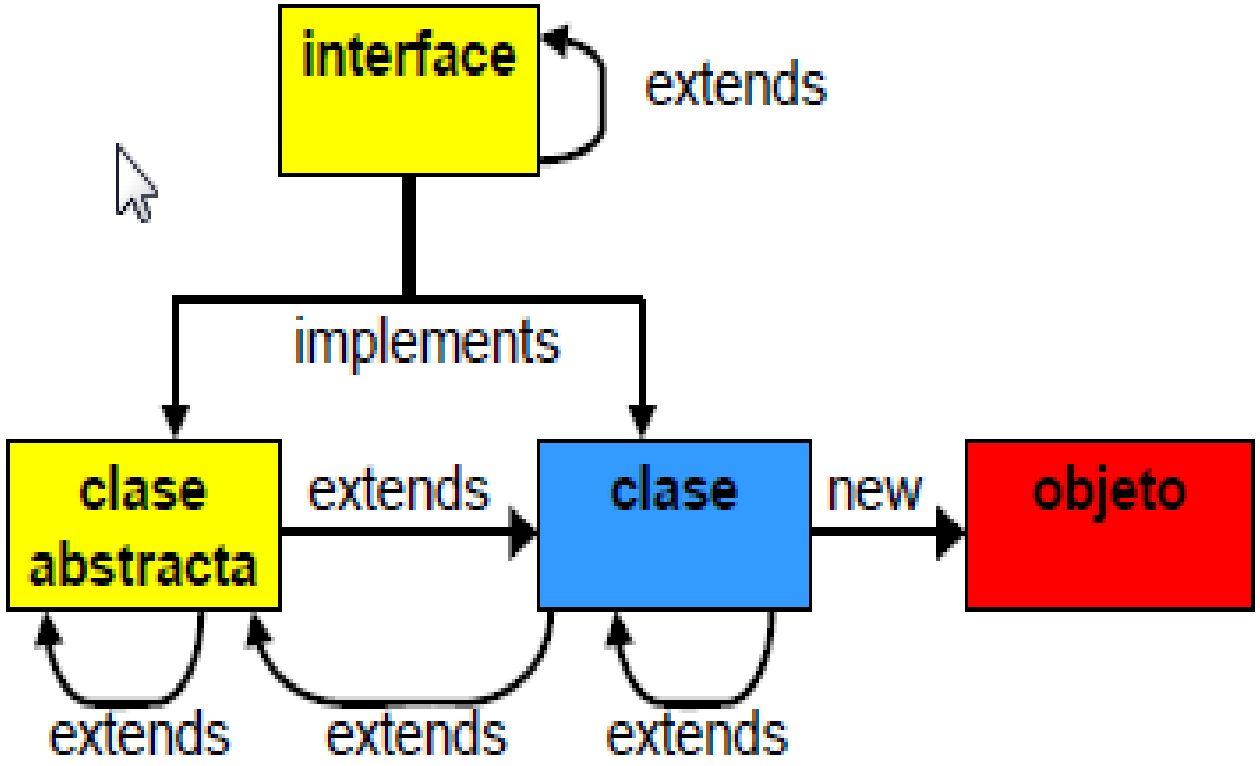
# Interfaces (II)

- ¿Cuando emplear interfaces?
  - Cuando sabemos el que queremos pero no sabemos (todavía) como hacerlo.
  - Lo hará otro.
  - Lo haremos de varias formas.

# Interfaces (III)

- Una clase implementa una interfaz cuando proporciona código concreto para los métodos definidos en la interfaz.
  - De una misma interfaz pueden derivarse varias implementaciones.
  - Una misma clase puede implementar varias interfaces (implementación múltiple).
  - Si una clase no implementa todos los métodos definidos en una interfaces sino solo una parte, el resultado es una clase abstracta (implementación parcial).

# Interfaces (IV)



# Colecciones (I)

- Interfaces principales:

| | |
|---|---|
| **Collection** | Contiene la definición de todos los métodos genéricos que tienen que implementar las colecciones. |

| | |
|---|---|
| **List** | Colección de objetos con una secuencia determinada. |
| **Set** | Colección de objetos donde no se admiten duplicados. |
| **Map** | Almacena parejas de objetos (clave-valor). |

# Colecciones (II)

- Interfaces e implementaciones

| | | | Implementaciones | | | | |
|---|---|---|---|---|---|---|---|
| | | | Tabla Hash | Array redimensionable | Árbol balanceado | Listas enlazadas | Tabla Hash + Listas enlazadas |
| Interface | Collection | Set | HashSet | | TreeSet | | LinkedHashSet |
| | | List | | **ArrayList** | | **LinkedList** | |
| | | Map | HashMap | | TreeMap | | LinkedHashMap |

# ArrayList

- La clase ArrayList permite el almacenamiento de datos en memoria de forma parecida a los *arrays* convencionales pero con una gran ventaja: la cantidad de elementos que puede guardar es dinámica.
  - La cantidad de elementos de un *array* convencional está limitado por el número indicado en el momento de su creación o inicialización.
  - Los *ArrayList* pueden guardar un número variable de elementos sin estar limitado por un número prefijado.
- Forma parte del paquete java.util.ArrayList

# Declaración de una variable ArrayList

- De forma genérica: `ArrayList nomLlista;`
  - De esta forma no se especifica el tipo de datos. Esto nos permite listas heterogéneas, pero es necesario hacer un *casting* en la recuperación de los elementos.
  - Es recomendable especificar el tipo de datos que contendrá la lista. Así se utilizarán las operaciones y métodos adecuados para el tipo de datos concreto.
- Para especificar el tipo de datos: `ArrayList<nomClasse> nomLlista;`
  - En caso de guardar datos de un tipo básico de Java (char, int, double, etc…), se tiene que especificar el nombre de la clase asociada (Wrapper class): Character, Integer, Double, etc.
  - Ejemplos:
    - `ArrayList<String> paises;`
    - `ArrayList<Integer> edades;`

# Creación de un objeto *ArrayList* (I)

```
ArrayList<nomClasse> nomLlista;

nomLlista = new ArrayList();
```

- Se puede declarar la lista a la vegada que se crea:
  - `ArrayList<nomClasse> nomLlista = new ArrayList<nomClasse>();`
- Ejemplos:

```
ArrayList<String> paises = new ArrayList<String>();

ArrayList<Alumno> alumnos = new ArrayList<Alumno>();
```

# Creación de un objeto *ArrayList* (II)

- Tiene 3 constructores:
  - `ArrayList()`: constructor por defecto. Crea un ArrayList vacío.
  - `ArrayList(int capacitatInicial)`: crea una lista con una capacidad inicial indicada.
  - `ArrayList(Collection c)`: crea una lista a partir de los elementos de la colección indicada.

# Añadir elementos al final de la lista

- `boolean add(Object elementoAInsertar)`
  - Los elementos que se van añadiendo se colocan después del último elemento.
  - El primer elemento se colocará en la posición 0.
- Ejemplo:

```
ArrayList<String> paisos = new ArrayList<String>();
paisos.add("España"); //Ocupa la posición 0
paisos.add("Francia"); //Ocupa la posición 1
paisos.add("Portugal"); //Ocupa la posición 2

//Se pueden crear ArrayList para guardar datos numéricos
ArrayList<Integer> edades = new ArrayList<Integer>();
edades.add(22);
edades.add(31);
edades.add(18);
```

# Añadir elementos en una posición determinada

- `void add(int posicion, Object elementoAInsertar)`
  - Inserta el elemento en la posición indicada y desplaza todos los elementos uno hacia la derecha.
  - Si se intenta insertar en una posición que no existe, se producirá la excepción IndexOutOfBoundsException.
- Ejemplo:

```
ArrayList<String> paisos = new ArrayList<String>();
paisos.add("España");
paisos.add("Francia");
paisos.add("Portugal");

//El orden hasta ahora es: España, Francia, Portugal

paisos.add(1, "Italia");

//El orden ahora es: España, Italia, Francia, Portugal
```

# Consultar un elemento de una lista

- `Object get(int posicion)`
  - Permite obtener el elemento guardado en una determinada posición.
  - Ejemplo:

```
System.out.println(paisos.get(3));
//Siguiendo con el ejemplo anterior, mostraría: Portugal
```

# Modificar un elemento de la lista

- `Object set(int posicion, Object nuevoElemento)`
  - Permite modificar un elemento que previamente ha sido guardado en la lista.
    - El primer parámetro indica la posición que ocupa el elemento a modificar.
    - El segundo parámetro indica el nuevo elemento que sustituirá al anterior.
  - Ejemplo:

```
paisos.set(1,”Alemania”));
```

# Buscar un elemento

- `int indexOf(Object elementoBuscado)`
  - Devuelve la posición del elemento buscado.
    - Si el elemento se encuentra más de una vez, indicará la posición de la primera aparición.
    - El método lastIndexOf obtiene la posición del último elemento encontrado.
    - Si el elemento no se encuentra, devolverá -1
  - Ejemplo:

```
String paisBuscat = "Francia";
int pos = paisos.indexOf(paisBuscat);
if(pos!=-1)
   System.out.println(paisBuscat + " en la posición: "+pos);
else
   System.out.println(paisBuscat + " no se ha encontrado");
```

# Recorrido de una lista (I)

- `size()`
  - Devuelve el número de elementos de la lista.
  - Ejemplo:

```
for(int i=0; i < paisos.size(); i++)
   System.out.println(paisos.get(i));
```

```
for(String pais:paisos)
   System.out.println(pais);
```

# Recorrido de una lista (II)

- También se pueden recorrer utilizando un iterador
  - Ejemplo:

```
Iterator iter = paisos.iterator();
while(iter.hasNext()) { //True si han más elementos
    System.out.println(iter.next()); //devuelve el elemento
}                                    //y apunta al siguiente
```

# Otros métodos

- `boolean remove(Object o)`
  - Elimina de la colección lo object indicado.
- `void clear()`
  - Borra todo el contenido de la lista.
- `Object clone()`
  - Devuelve una copia de la lista.
- `boolean contains(Object o)`
  - Devuelve true si el elemento se encuentra en la lista y false en caso contrario.
- `boolean isEmpty()`
  - Devuelve true si la lista está vacía.
- `Object[]toArray()`
  - Convierte la lista en un array.

# Array *versus* ArrayList

| *arrays* | *List* |
|---|---|
| `String[] x;` | `List<String> x;` |
| `x = new String[1000];` | `x = new ArrayList<String>();` |
| `... = x[20];` | `... = x.get(20);` |
| `x[20] = "1492";` | `x.set(20, "1492");` |
| | `x.add("2001");` |

# Map

- La interfaz Map (java.io.Map) permite representar una estructura de datos para almacenar pares “clave | valor” de forma que para una determinada clave solo tenemos un valor.
- Esta estructura de datos es conocida en otros lenguajes de programación como “Diccionarios”. Aunque en cada lenguaje la implementación puede variar, la idea final es la misma.
- Al igual que las listas, también los podemos recorrer con iteradores.
- Java tiene varías interfaces implementadas de Map. Las más empleadas son:
  - **HashMap**: los elementos no tienen un orden específico. Permite una clave con valor nulo y puede tener varios valores nulos.
  - **LinkedHashMap**: los elementos están ordenados según se han insertado. Permite una clave con valor nulo y puede tener varios valores nulos.
  - **TreeMap**: los elementos se ordenan por la clave de forma “natural”. Por ejemplo, si la clave son valores enteros los ordena de menor a mayor. No permite claves con valor null pero puede tener varios valores nulos.

# Declaración de una variable Map

- De forma genérica:

```
HashMap nomHashMap;

TreeMap nomTreeMap;

LinkedHashMap nomLinkedHashMap;
```

  - De esta forma no se especifica el tipo de datos. Esto nos permite listas heterogéneas, pero es necesario hacer un *casting* en la recuperación de los elementos.
  - Es recomendable especificar el tipo de datos que contendrá la lista. Así se utilizarán las operaciones y métodos adecuados para el tipo de datos concreto.

- Para especificar el tipo de datos:

```
HashMap<Integer, String> nomHashMap;

TreeMap<String, String> nomTreeMap;

LinkedHashMap<String, Double> nomLinkedHashMap;
```

# Creación de un objeto Map

```
HashMap<Integer, String> nomHashMap;

nomHashMap = new HashMap<Integer, String>();
```

- Se puede declarar la lista al mismo tiempo que se crea:

```
TreeMap<String, String> nomTreeMap = new TreeMap<String, String>();
```

- Ejemplos:

```
HashMap<Integer, String> map = new HashMap<Integer, String>();

TreeMap<String, Integer> map = new TreeMap<String, Integer>();

LinkedHashMap<String, Integer> map = new LinkedHashMap<String,Integer>();
```

# Principales Métodos (I)

- `Object put(Object key, Object value)`
  - Asocia el valor `value` con el elemento que tiene la clave `key`. Si ya existe un elemento con esa clave reemplaza su valor. Si el elemento con la clave key ya tenía un valor, devuelve el valor viejo, en caso contrario devuelve null.
- `Object get(Object key)`
  - Obtiene el elemento del Map que tiene como clave `key`. Devuelve null si no encuentra el elemento.
- `Collection<Object> values()`
  - Obtiene la lista de los valores que están al Map.
- `Set<Object> keySet()`
  - Obtiene el conjunto de claves que están al Map.
- `boolean replace(Object key, Object value)`
  - Cambia el valor del elemento con la clave `key` por `newValue`. Devuelve true si se ha podido cambiar el valor.
- `boolean replace(Object key, Object oldValue, Object newValue)`
  - Cambia el valor el elemento con la clave `key` que tiene el valor `oldValue` por `newValue`. Devuelve true si se ha podido cambiar el valor.

# Principales Métodos (II)

- Los métodos de la interfaz Map son similares a los que hemos visto para las listas.
- `void clear()`
  - Borra todo el contenido del Map.
- `Object clone()`
  - Devuelve una copia del Map.
- `boolean containsKey(Object key)`
  - Devuelve true si hay algún elemento que tenga como clave `key`.
- `boolean containsValue(Object value)`
  - Devuelve true si hay algún elemento que tenga como valor `value`.
- `boolean isEmpty()`
  - Devuelve true si lo Map está vacío.
- `int size()`
  - Devuelve el número de elementos que tiene el Map.