# 6. Colecciones (java.util)



# 6. Colecciones (java.util)

- Java ofrece para el almacenamiento de objetos dos tipos de estructuras:
  - **Arrays**
    - La estructura más sencilla para el almacenamiento de objetos.
    - Gestión de memoria estática.
    - Permite el almacenamiento de tipos básicos y objetos.
    - Sólo permite almacenar datos de un mismo tipo.
  - **Clases contenedoras de objetos o Colecciones**
    - Son estructuras complejas basadas en clases Java que realizan una gestión interna del almacenamiento y recuperación de los elementos.
    - Gestión de memoria dinámica.
    - Permite el almacenamiento sólo de objetos.

# 6.1 Arrays

- La característica principal de un array es que una vez dimensionado no se puede variar su tamaño.
- La estructura se manipula directamente, por este motivo es la que ofrece un mejor rendimiento en los accesos directos e iteraciones.
- Definición de un array:
  - Se hace uso de los corchetes.
  - No se dimensiona.

```
tipoArray nombreArray[];
int temperaturas[];
Persona listado[];
```

- Creación de un array:
  - Se emplea el operador new para realizar la reserva de memoria con la dimensión dada.

```
nombreArray = new tipoArray[dimension];
temperaturas = new int[10];
listado = new Persona[20];
```

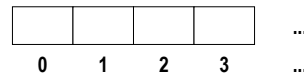
- La definición y creación también se puede efectuar en la misma línea:

```
int temperaturas[] = new int[10];
Persona listado[] = new Persona[20];
```



# 6.1 Arrays

- La forma de direccionar un array es a través de un índice numérico.




- Por defecto, un array se inicializa al valor por defecto del tipo (0) u objeto (null).
- Agregar un elemento a una posición:

```
temperaturas[2] = 34;
listado[1] = new Persona("Luis", 17);
```

- Extraer un elemento de una posición:

```
int a = temperaturas[2];
Persona p = listado[1];
```

- Eliminar un elemento:

```
temperaturas[2] = 0; //puede llevar a equívoco
listado[1] = null; //con objetos, no
```

- Recorrer la estructura:
  - Se utiliza el atributo length para obtener el número de posiciones del array, no para saber el número de elementos de la estructura.

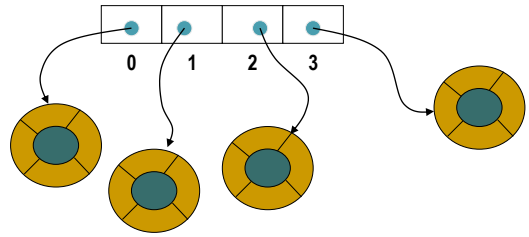
```
temperaturas.length → 10
listado.length → 20;
```

# 6.1 Arrays

- También se puede inicializar un array en el momento de su definición:

```
int temperaturas[] = {34, 36, 23, 40};
```

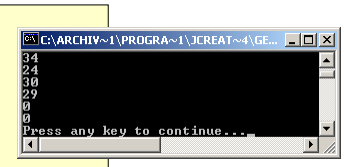
- Un array es considerado como un tipo específico de objeto.
- Cuando se intenta acceder a una posición que no existe se lanza la excepción: *java.lang.ArrayIndexOutOfBoundsException*.
- En memoria se comporta como un conjunto de referencias:



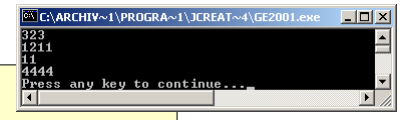



# 6.1 Arrays - *Ejemplo*

```
int temperaturas[] = new int[6];

temperaturas[0] = 34;
temperaturas[1] = 24;
temperaturas[2] = 30;
temperaturas[3] = 29;

for(int i=0;i<temperaturas.length;i++)
        System.out.println(temperaturas[i]);
```




```
Persona agenda[] = new Persona[4];

agenda[0] = new Persona("Juan", 323);
agenda[1] = new Persona("Miguel", 1211);
agenda[2] = new Persona("David", 11);
agenda[3] = new Persona("Ramón", 4444);

for (int i=0;i<agenda.length;i++)
        System.out.println(agenda[i].getDni());
```



## 6.2 Colecciones (java.util)

- Las clases que permiten agrupar objetos se denominan contenedores o colecciones de objetos.
- Son estructuras más complejas que los arrays, ya que además de poseer un interfaz para realizar todas las operaciones básicas de agregación y extracción, poseen otras más complejas (algoritmos), para realizar inserciones, borrado y búsquedas de objetos.
- Los tipos de objetos deberán ser encapsulados a través de los wrappers.
- Además de estas clases, Java proporciona un conjunto de Interfaces que permiten trabajar de una forma abstracta con estas estructuras omitiendo los detalles de sus implementaciones.

Los Interfaces enfocados a realizar iteraciones, reciben el nombre de Iterators.



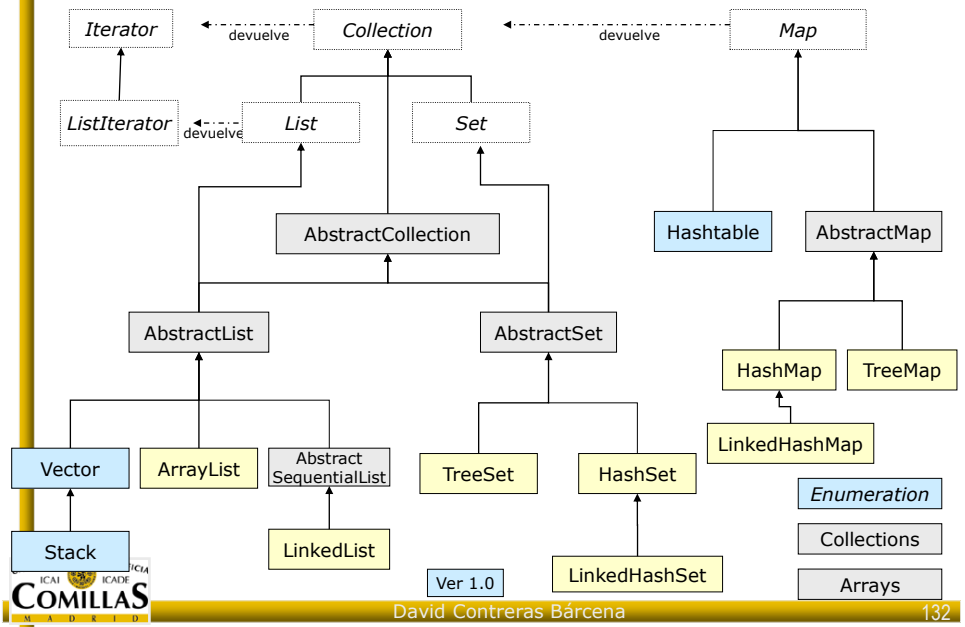
## 6.2 Colecciones (java.util)

- Existen tres tipos genéricos de contenedores definidos por sendos interfaces. Cada uno de ellos dispone de dos o tres implementaciones.
- Contenedores:

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

- Los *List* y *Set* generalizan su comportamiento en otro Interface llamado *Collection*. Cuando se haga referencia en esta documentación a colecciones, se referirá al conjunto total de clases contenedoras de objetos.

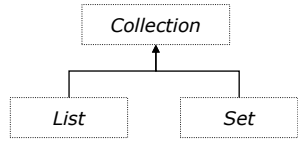
# 6.2 Colecciones (java.util)





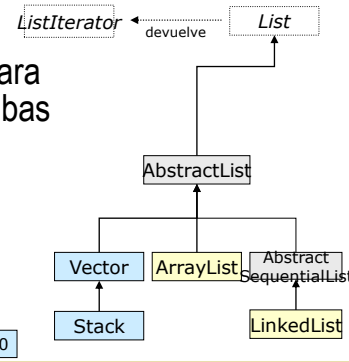

# 6.3 Interface Collection (java.util) 1.2

- El interfaz de *Collection* permite básicamente añadir, eliminar y recorrer la estructura gracias a un *Iterator*.
- El interfaz de *List* refina el comportamiento de *Collection* permitiendo también, extraer, buscar y reemplazar ocurrencias. Además ofrece la posibilidad de recorrer la estructura con un *ListIterator*.
- El interfaz de Set no amplia el comportamiento de Collection.



# 6.4 Interface List (java.util) 1.2

- La característica más importante de una List es el orden de almacenamiento, asegurando que los elementos siempre se mantendrán en una secuencia determinada.
- El List añade un conjunto de métodos a Collection que permiten la inserción y borrado de elementos en mitad de la lista.
- Permite generar un ListIterator para moverse a través de las lista en ambas direcciones.
- Sublcases:
  - ArrayList y LinkedList.




Ver 1.0



# 6.4 Interface List (java.util) 1.2

- **Clase ArrayList**
  - Lista volcada en un array.
  - Se debe utilizar en lugar de Vector como almacenamiento de objetos de propósito general.
  - Permite un acceso aleatorio muy rápido a los elementos, pero realiza con bastante lentitud las operaciones de insertado y borrado de elementos en medio de la Lista.
  - Se puede utilizar un ListIterator para moverse hacia atrás y hacia delante en la Lista, pero no para insertar y eliminar elementos. Para ello LinkedList.
  - Otras características: No sincronizada y fail-fast.
- **Clase LinkedList**
  - Proporciona un óptimo acceso secuencial, permitiendo inserciones y borrado de elementos de en medio de la Lista muy rápidas.
  - Por el contrario, es bastante lento el acceso aleatorio, en comparación con la ArrayList.
  - También dispone de los métodos addLast(), getFirst(), getLast(), removeFirst() y removeLast(), que no están definidos en ningún interfaz o clase base y que permiten utilizar la Lista Enlazada como una pila, una cola o una cola doble.

# 6.4 Interface List (java.util) 1.2

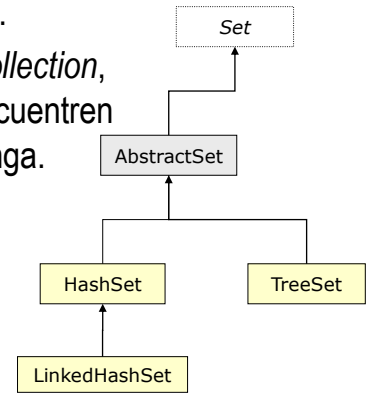
- Comparación de rendimiento:

| | |
|---|---|
| **ArrayList** | -SI: Acceso directo (3070)<br>- NO: Iteración (12200), inserción (500) y borrado (46850) |
| **LinkedList** | - SI: Iteración (9110), inserción (110) y borrado (60).<br>- NO: Acceso directo(16320) |
| ***array*** | - Acceso directo (1430) y Iteración (3850) |



# 6.5 Interface Set (java.util) 1.2

- No se admiten objetos duplicados, por lo que cada elemento que se añada a un *Set* debe ser único.
- Los elementos incorporados al conjunto deben tener redefinido el método *public boolean equals(Object)*. Mecanismo para evitar duplicados.
- *Set* tiene el mismo interfaz que *Collection*, y no garantiza el orden en que se encuentren almacenados los objetos que contenga.
- Sublcases:
  - HashSet, TreeSet y LinkedHashSet.



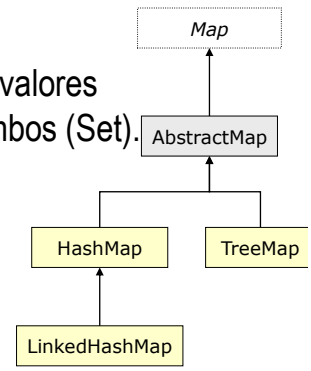
# 6.5 Interface Set (java.util)

- **Clase HashSet** 1.2
  - Solución habitual de Sets.
  - Emplea una tabla hash internamente, por ese motivo los objetos almacenados deben implementar el método hasCode() –ver Interface Map-.
  - Cuando el tiempo de búsqueda sea crítico.
  - No se respeta el orden de inserción.
- **Clase LinkedHashSet** 1.4
  - Lista doblemente enlazada donde se garantiza el orden de inserción.
- **Clase TreeSet** 1.2
  - Es un *Set* ordenado, almacenado según un árbol balanceado.
  - Forma sencilla de extraer una secuencia ordenada.
  - Los objetos a almacenar deben implementar el *Interface Comparable*.



# 6.5 Interface Map (java.util) 1.2

- Los Mapas almacenan parejas de valores, relacionando un objeto (clave) con otro objeto (valor).
- Las claves no pueden estar duplicadas y sólo puede tener una valor asociado.
- Los valores pueden estar duplicados.
- Se pueden extraer las claves (Set), los valores (Collection) o la entrada conjunta de ambos (Set).
- Implementaciones:
  - HashMap, TreeMap y LinkedHashMap.



# 6.5 Interface Map (java.util) 1.2

- ***HashMap***
  - Excelente rendimiento realizando búsquedas. Implementación basada en una tabla hash. Se debe utilizar en lugar de **Hashtable**.
  - Proporciona un rendimiento constante al insertar y localizar la pareja de valores.
  - El rendimiento se puede ajustar a través de la capacidad y el factor de carga de la tabla hash.
  - Acepta nulos en la clave y en el valor.
  - El objeto clave tiene que redefinir los métodos:
    - *public int hashCode()*
    - *public boolean equals(Object)*
- ***TreeMap***
  - Implementación basada en un árbol balanceado.
  - Las claves de las parejas se ordenan.
  - Permite recuperar los elementos en un determinado orden.
  - Es el único mapa que define el método *subMap()*, que permite recuperar una parte del árbol solamente.
  - Los objetos clave deben implementar el interface **Comparable**.



- **LinkedHashMap** 1.4
  - Lista doblemente enlazada donde se garantiza el orden de inserción.

# 6.5 Recorrer las colecciones

- Java proporciona interfaces para recorrer las colecciones de forma genérica sin necesidad de conocer su estructura interna (una forma abstracta de realzar iteraciones sobre cualquier colección).
  - **List**: Permite comprobar si hay objetos, avanzar y eliminar.
    - boolean hasNext(): ¿hay más objetos en la iteración?
    - Object next(): devuelve el objeto siguiente de la iteración.
    - void remove(): elimina el objeto actual.

```
ArrayList lista = new ArrayList();
Iterator it = lista.iterator();
while(it.hasNext())
{
        Object o = it.next();
        System.out.println(o);
}
```



# 6.5 Recorrer las colecciones

- **ListIterator**: Refina el comportamiento del anterior permitiendo además, retroceder, agregar, actualizar y devolver el índice actual.