

# El framework de Colecciones

## ① Las interfaces Centrales

`Collection`

- `Set` y `SortedSet`
- `List`
- `Queue`

`Map` y `SortedMap`

## ② Las interfaces Secundarias

`Iterator` y `ListIterator`

## ③ Las implementaciones convencionales

`HashSet`, `HashMap`, `ArrayList`, `LinkedList`, `TreeSet`,  
`TreeMap`, *etc.*

# Colecciones

## Composición

Una colección es un objeto que representa a un grupo de objetos. Se usa para almacenar, recuperar y manipular un conjunto de datos.

Un *framework* de colecciones permite representar y manipular colecciones de una manera unificada, independientemente de los detalles de implementación. El *frameworks* de colecciones de JAVA cuentan con:

- Interfaces: son tipos de datos abstractos que representan colecciones y que permiten manejarlas en forma independiente de su implementación. Forman jerarquías de herencia.
- Implementaciones: son implementaciones concretas de las interfaces. Son estructuras de datos.
- Algoritmos: son métodos de clase que realizan operaciones útiles (búsquedas y ordenamientos) sobre objetos que implementan alguna de las interfaces de colecciones.

Java incluye en su librería, implementaciones de las estructuras de datos más comunes. Esta parte de la librería es conocida como API de colecciones.

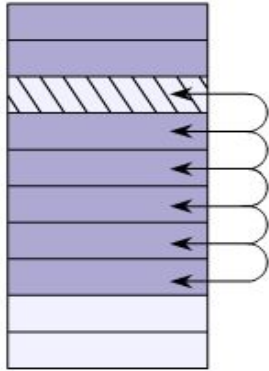
# Colecciones

## Tecnologías de almacenamiento

Existen cuatro tecnologías de almacenamiento básicas disponibles para almacenar objetos: arreglo, lista enganchada, árbol y tabla de hash.

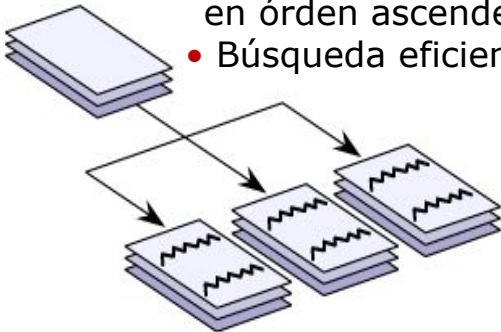
### Arreglo

El acceso es muy eficiente.  
Es ineficiente cuando se agrega/elimina un elemento.  
Los elementos se pueden ordenar.



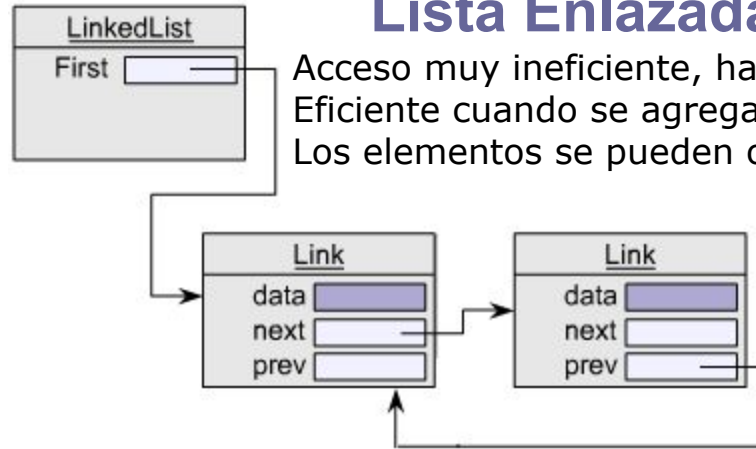
### Arbol

- Almacenamiento de valores en orden ascendente.
- Búsqueda eficiente.



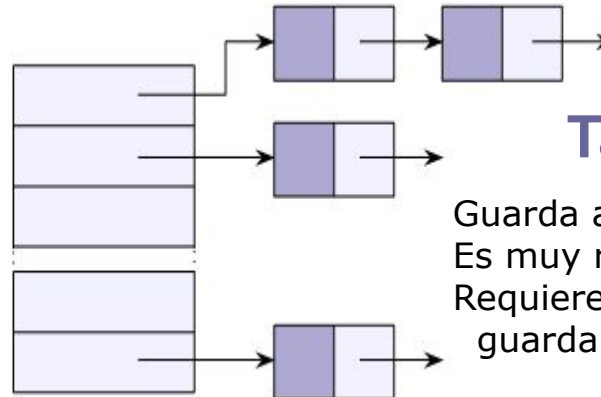
### Lista Enlazada

Acceso muy ineficiente, hay que recorrer la lista.  
Eficiente cuando se agrega/elimina un elemento.  
Los elementos se pueden ordenar.



### Tabla de hash

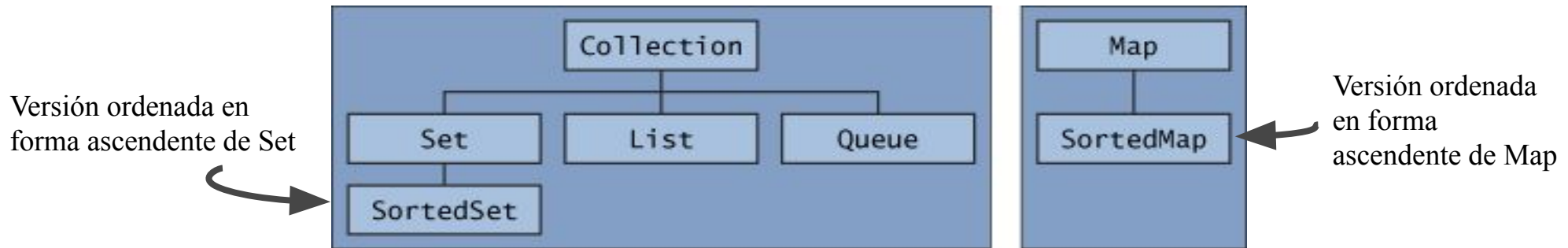
Guarda asociaciones (clave, valor).  
Es muy rápido, accede por clave  
Requiere memoria adicional para guardar las claves (tabla).



# Colecciones

## Jerarquías de Interfaces

Encapsulan distintos tipos de colecciones y son el fundamento del framework de colecciones.



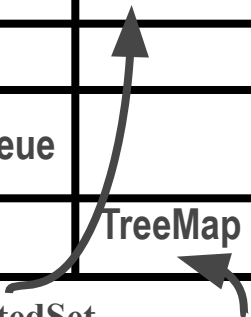
- **Collection:** representa un conjunto de objetos, llamados elementos. Es la raíz de la jerarquía de colecciones. La plataforma Java no provee una implementación directa para la interface Collection, pero si para sus subinterfaces Set, List y Queue. Permite elementos duplicados.
- **Set:** extiende Collection y no permite elementos duplicados. Modela el concepto de conjunto matemático.
- **List:** extiende Collection, es una colección que permite elementos duplicados (también llamada secuencia) y que incorpora acceso posicional mediante índices.
- **Queue:** extiende Collection proveyendo operaciones adicionales para inserción, extracción e inspección de elementos. Típicamente los elementos de una Queue están ordenados usando una estrategia FIFO (First In First Out). Se incorporó a partir de la versión jse 5.0.
- **Map:** permite tener pares de objetos que “mapean” claves con valores. No permite claves duplicadas. Cada clave mapea a lo sumo con un valor.

# Colecciones

## Implementaciones de las interfaces

La tabla muestra las implementaciones de propósito general que vienen con la plataforma java, las cuales siguen una convención de nombre, combinando la estructura de datos subyacente con la interface del framework:

Interfaces	Tecnologías de almacenamiento				
	Tabla de Hashing	Arreglo de tamaño variable	Árbol	Lista Encadenada	Tabla de Hashing + Lista Encadenada
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue		ArrayBlockingQueue		LinkedBlockingQueue	
Map	HashMap		TreeMap		LinkedHashMap



Todas las implementaciones de propósito general:

- Tienen implementado el método `toString()`, el cual retorna a la colección de una manera legible, con cada uno de los elementos separados por coma.
- Tienen por convención al menos 2 constructores: el nulo y otro con un argumento `Collection`:  
`TreeSet()` y `TreeSet(Collection c)`  
`LinkedHashSet()` y `LinkedHashSet(Collection c)`

...

# Colecciones

## La interface Collection

La interface **Collection** representa un conjunto de objetos de cualquier tipo. Esta interface se usa cuando se necesita trabajar con grupos de elementos de una manera muy genérica.

```
public interface Collection<E> extends Iterable<E> {  
  
    // operaciones básicas  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element);  
    boolean remove(Object element);  
    Iterator iterator();  
  
    // operaciones en "masa"  
    boolean containsAll(Collection<E> c);  
    boolean addAll(Collection<E> c);  
    boolean removeAll(Collection<E> c);  
    boolean retainAll(Collection<E> c);  
    void clear();  
  
    // operaciones de Arreglos  
    Object[] toArray();  
}
```

Las clases que implementan esta interface pueden iterarse con un objeto Iterator.

Este método retorna un iterador que permite recorrer la colección desde el comienzo hasta el final

Convierte la colección a un arreglo

# Colecciones

## La interface List

Un objeto **List** es una secuencia de elementos donde puede haber duplicados. Además de los métodos heredados de **Collection**, define métodos para recuperar y modificar valores en la lista por posición como: **E get(int index)**; **E set(int index, E element)**; **boolean add(E element)**; **void add(int index, E element)**; **E remove(int index)**.

La plataforma java provee 2 implementaciones de List, que son **ArrayList** y **LinkedList**.

```
import java.util.*;
public class DemoIterador{
    public static void main(String[] args){
        List<Integer> lista = new ArrayList<Integer>();
        lista.add(1);
        lista.add(new Integer(2));
        lista.add(190);
        lista.add(90);
        lista.add(7);
        lista.remove(new Integer(2));
        System.out.print(lista.toString());
    }
}
```

**[1, 190, 90, 7]**

salida

Se podría reemplazar por  
**new LinkedList<Integer>()** y  
todo sigue funcionando

### boxing/unboxing

Convierte automáticamente datos de tipo primitivo int a objetos de la clase Integer. Mejora a partir JSE 5.0

# Colecciones

## La interface Set

Un objeto Set es una colección que no contiene elementos duplicados. Tiene exactamente los mismos métodos que la interface **Collection**, pero agrega la restricción de no mantener duplicados.

La plataforma java provee implementaciones de propósito general para Set. Por ejemplo **HashSet** (mejor performance, almacena los datos en una tabla de hash) y **TreeSet** (más lento pero ordenados).

```
public class DemoIterador{  
    public static void main(String[] args) {  
        Set<String> instrumentos= new HashSet<String>();  
        instrumentos.add(" Piano");  
        instrumentos.add(" Saxo");  
        instrumentos.add(" Violin");  
        instrumentos.add(" Flauta");  
        instrumentos.add(" Flauta");  
        System.out.println(instrumentos.toString());  
    }  
}
```

La interface Set es útil para crear colecciones sin duplicados desde una colección `c` con duplicados.



```
Set<String> sinDup=new TreeSet<String>(c);
```

salida

[Violin, Piano, Saxo, Flauta]

Implementa **SortedSet**

Cambiando únicamente la instanciación por un objeto **TreeSet()**, obtenemos una colección ordenada:

salida

```
Set<String> instrumentos= new TreeSet<String>();  
[Flauta, Piano, Saxo, Violin]
```

En este caso el compilador chequea que los objetos que se insertan (add()) sean **Comparables!!**



# Colecciones

## La interface Map

Un objeto Map mapea claves con valores. No puede contener claves duplicadas y cada clave mapea con a lo sumo un valor.

La plataforma java provee implementaciones de propósito general para **Map**. Por ejemplo, las clases **HashMap** y **TreeMap** con un comportamiento y performance análogo a las implementaciones mencionadas para la interface Set.

```
public interface Map <K,V> {  
    // Operaciones Básicas  
    V put(K clave, V valor);  
    V get(K clave);  
    V remove(K clave);  
    boolean containsKey(K clave);  
    boolean containsValue(V valor);  
    int size();  
    boolean isEmpty();  
    // Operaciones en "masa"  
    void putAll(Map <K,V> t);  
    void clear();  
    // Vistas  
    Set<K> keySet();  
    Collection<V> values();  
    ...  
}
```

```
public class DemoMap {  
    public static void main(String[] args) {  
        Map<String, Integer> numeros=new HashMap<String,Integer>();  
        numeros.put("uno", new Integer(1));  
        numeros.put("dos", new Integer(2));  
        numeros.put("tres", new Integer(3));  
        System.out.println(numeros.toString());  
    }  
}
```

**{tres=3, uno=1, dos=2}**

salida

Implementa  
**SortedSet0**

Cambiando únicamente la instanciación por un objeto **TreeMap()**, obtenemos una colección ordenada:

salida

```
Map<String, Integer> numeros=new TreeMap<String, Integer>();
```

**{dos=2, tres=3, uno=1}**

En este caso el compilador chequea que los objetos que se insertan (put()) sean **Comparables!!**

# Colecciones

## La interface Queue

Un objeto **Queue** es una colección diseñada para mantener elementos que esperan por procesamiento. Además de las operaciones de **Collection**, provee operaciones para insertar, eliminar e inspeccionar elementos. No permite elementos nulos.

La plataforma java provee una implementación de **Queue**: **PriorityQueue** (es una cola con prioridades) en el paquete `java.util` y varias en el paquete `java.util.concurrent` como **DelayQueue** y **BlockingQueue** que implementan diferentes tipos de colas, ordenadas o no, de tamaño limitado o ilimitado, etc.

```
public interface Queue<E> extends Collection<E>{
```

```
// Búsqueda
```

```
E peek();
```

```
boolean offer(E e);
```

```
E poll();
```

```
}
```

Recupera, pero no elimina la cabeza de la cola.

Inserta el elemento en la cola si es posible

Recupera y elimina la cabeza de la cola.

```
public class DemoQueue{  
    public static void main(String[] args) {  
        PriorityQueue<String> pQueue
```

```
        PriorityQueue<String>();  
        pQueue.offer("Buenos Aires");  
        pQueue.offer("Montevideo");  
        pQueue.offer("La Paz");  
        pQueue.offer("Santigao");  
        System.out.println(pQueue.peek());  
        System.out.println(pQueue.poll());  
        System.out.println(pQueue.peek());  
    }  
}
```

PriorityQueue chequea que los objetos que se insertan sean **Comparables!!**

= new

Buenos Aires  
Buenos Aires  
La Paz

salida

# Colecciones

## Mecanismos para recorrerlas - for-each

Supongamos que tenemos una cartera de personas, representada por un List de objetos de tipo Person. El siguiente ejemplo imprime el nombre de todos los miembros contenidos en la colección usando un bucle for-each:

```
List<Integer> lista= new ArrayList<Person>();

for (Person p : lista) {
    System.out.println(p.getName());
}
```

```
package lambdas;
import java.time.LocalDate;
public class Person {
    public enum Sex {
        MALE, FEMALE
    }
    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;
    public int getAge() {}
    public String getName() {}
    . . .
    public void toString() {}
}
```

Si quisiéramos imprimir todos los miembros que cumplan con alguna característica, podríamos hacer métodos específicos:

```
public static void printPersonsOlderThan(List<Person> roster, int age) {
    for (Person p : roster) {
        if (p.getAge() >= age) {
            p.printPerson();
        }
    }
}
```

```
public static void printPersonsOlderThanSex(List<Person> roster, int age) {
    for (Person p : roster) {
        if (low<=p.getAge() && p.getAge()<high && p.gender==Person.Sex.MALE){
            p.printPerson();
        }
    }
}
```

Cuando hay muchas condiciones se hace engorroso

# Colecciones

## Mecanismos para recorrerlas - Iteradores

Este ejemplo usa `iterator()` y `listIterator()` para recorrer la collection, desde el inicio al final y viceversa

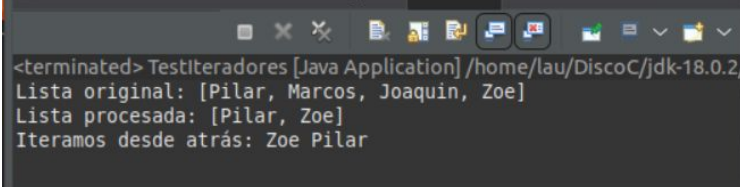
```
package pruebas;
import java.util.*;

public class TestIteradores {
    public static void main(String[] args) {
        List<Person> lista = new ArrayList<Person>();
        Person p1 = new Person("Pilar", Person.Sex.FEMALE);
        Person p2 = new Person("Marcos", Person.Sex.MALE);
        Person p3 = new Person("Joaquin", Person.Sex.MALE);
        Person p4 = new Person("Zoe", Person.Sex.FEMALE);
        lista.add(p1); lista.add(p2); lista.add(p3); lista.add(p4);

        System.out.println("Lista original: " + lista);
        Iterator<Person> it1 = lista.iterator();
        while (it1.hasNext()) {
            Person p = it1.next();
            if (p.getGender() == Person.Sex.MALE)
                it1.remove();
        }
        System.out.println("Lista procesada: " + lista);

        System.out.print("Iteramos desde atrás: ");
        ListIterator<Person> it = lista.listIterator(lista.size());
        while (it.hasPrevious())
            System.out.print(it.previous()+" ");
    }
}
```

Si no se define al iterador de tipo `<Person>`, hay que castear cuando se recupera el objeto, dado que es de tipo `Object` y no lo puede asignar a `Person`



```
<terminated> TestIteradores [Java Application] /home/lau/DiscoC/jdk-18.0.2/
Lista original: [Pilar, Marcos, Joaquin, Zoe]
Lista procesada: [Pilar, Zoe]
Iteramos desde atrás: Zoe Pilar
```

# Colecciones

## Mecanismos para recorrerlas - operaciones agregadas

En las versiones posteriores al JDK 8, el método preferido para iterar sobre una colección es obtener un stream y realizar operaciones agregadas sobre él. Las operaciones agregadas a menudo se utilizan junto con expresiones lambda para hacer la programación más expresiva, utilizando menos líneas de código.

```
package lambdas;
import java.util.ArrayList;
import java.util.List;
public class TestStream {
    public static void main(String[] args) {
        List<Person> lista = new ArrayList<Person>();
        Person p1 = new Person("Pilar", Person.Sex.FEMALE, "pili@gmail.com");
        Person p2 = new Person("Marcos", Person.Sex.MALE, "mar@gmail.com");
        Person p3 = new Person("Joaquin", Person.Sex.MALE, "joaki@gmail.com");
        lista.add(p1); lista.add(p2); lista.add(p3);

        lista
            .stream().filter(p -> p.getGender() == Person.Sex.MALE && p.getAge() <= 25)
            .map(p -> p.getEmailAddress())
            .forEach(email -> System.out.println(email));
    }
}
```

Referencia:

<https://docs.oracle.com/javase/tutorial/collections/streams/index.html>

las operaciones agregadas filter(), map() y foreach() procesan elementos desde el stream, no desde la colección.

se obtiene la fuente de datos	stream()
se filtran objetos que coinciden con un predicado	filter()
se aplica una función a cada elemento y produce un nuevo stream	map()
utiliza un iterador para ejecutar una acción sobre cada elemento	forEach()