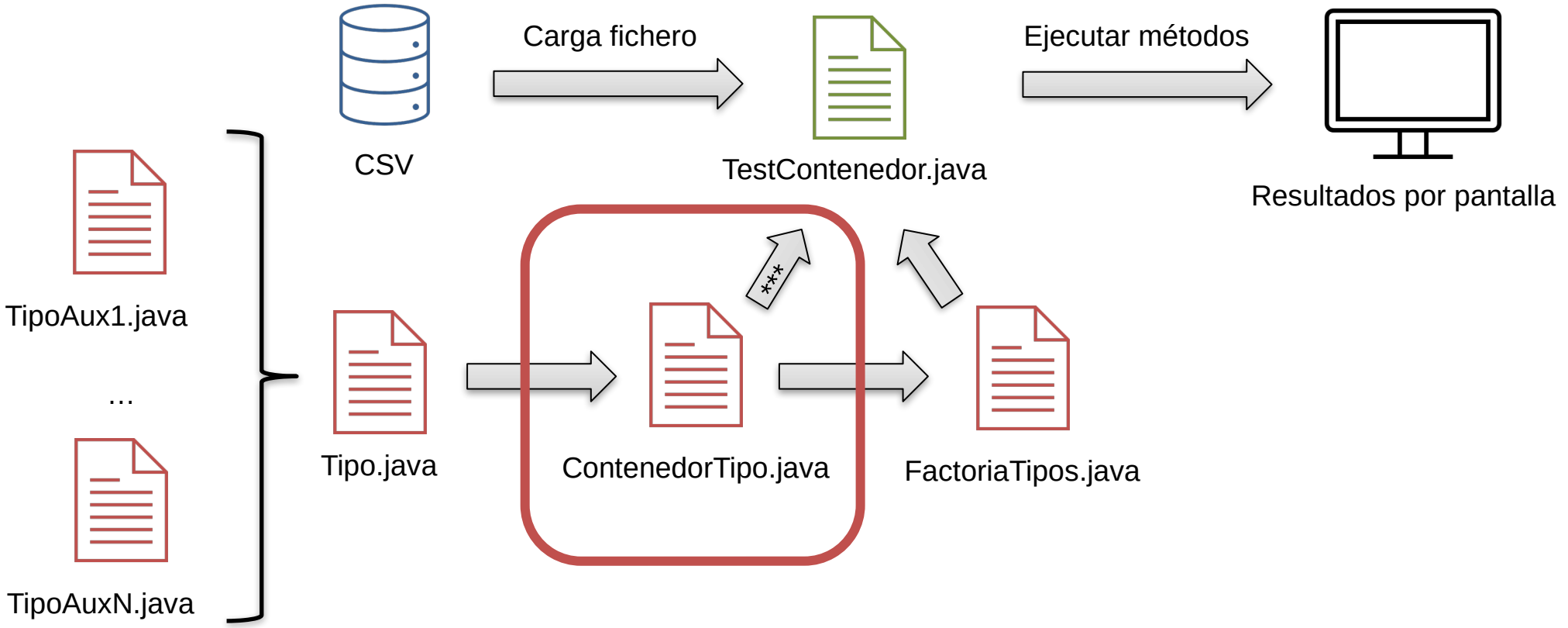
FUNDAMENTOS DE PROGRAMACIÓN

3 - Colecciones

UNIVERSIDAD DE SEVILLA

¿Qué se ve en este tema?



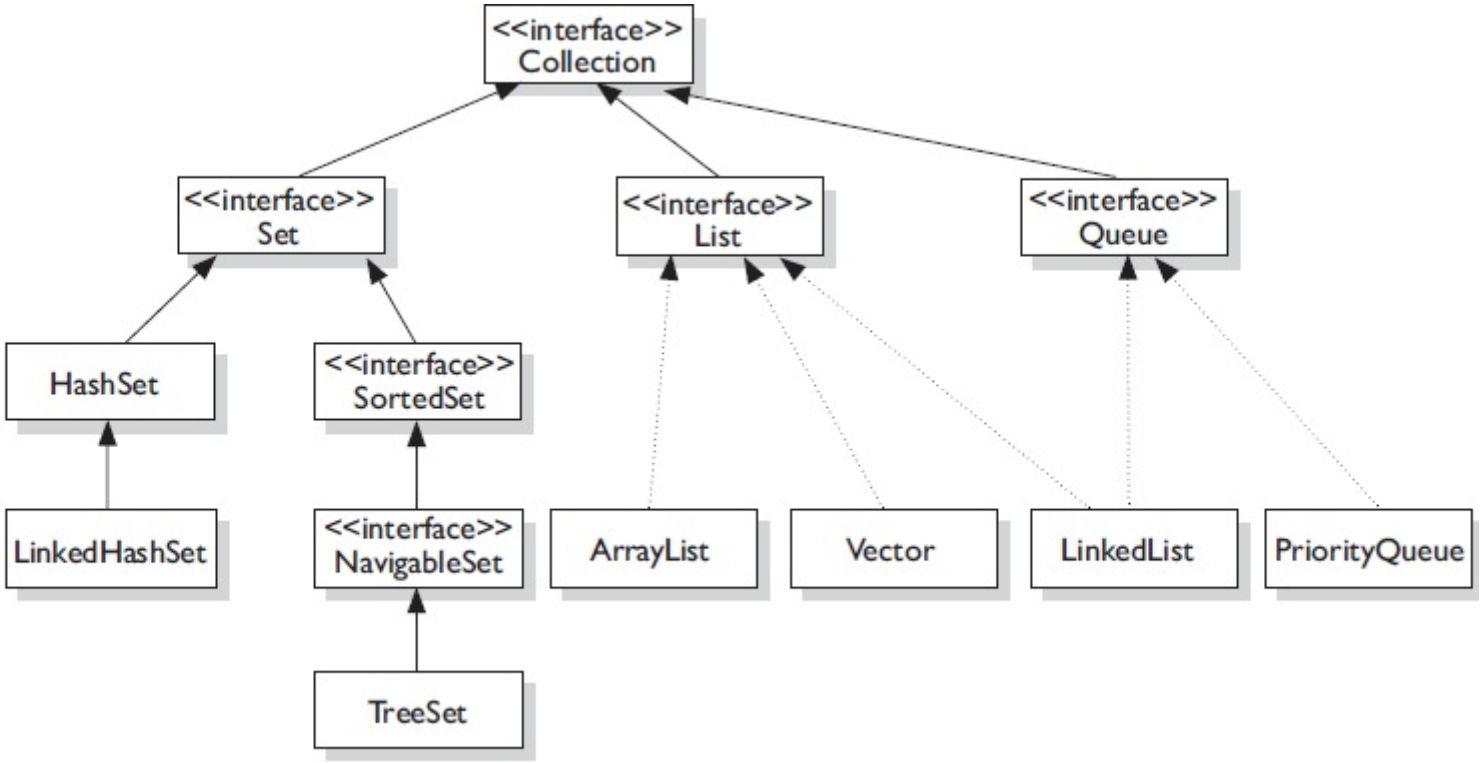
ÍNDICE

ÍNDICE

1. Introducción

- En este capítulo vamos a ver las colecciones, modeladas mediante la interfaz *Collection* y dos de las interfaces que heredan de ella:
  - *List*, que modela las listas.
  - *Set*, que modela los conjuntos.
  - *Map*, que modela el concepto matemático de Aplicación.

1. Introducción



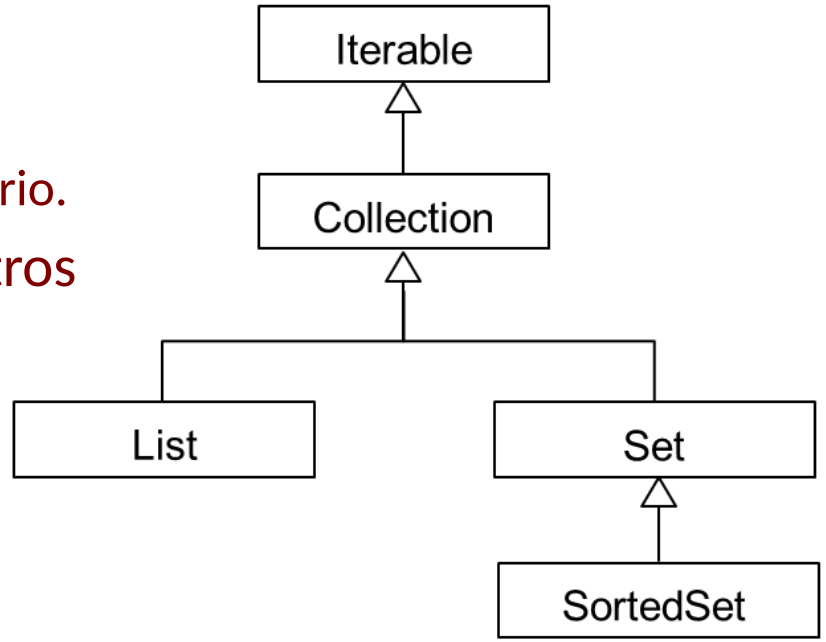
* ¿Qué es una interfaz?

- Una interfaz en Java es una colección de métodos abstractos y propiedades constantes.
- En las interfaces se especifica qué se debe hacer, pero no su implementación. Serán las clases que implementen estas interfaces las que describen la lógica del comportamiento de los métodos.
- El uso de las interfaces Java proporciona las siguientes ventajas:
  - Organizar la programación.
  - Permiten declarar constantes que van a estar disponibles para todas las clases implementen la interfaz.
  - Obligar a que ciertas clases utilicen los mismos métodos (nombres y parámetros).
  - Establecer relaciones entre clases que no estén relacionadas.

ÍNDICE

2. La interfaz Collection

- Definida en el paquete *java.util*.
- Tipo general, que agrupa objetos de un mismo tipo.
  - Pueden admitir elementos duplicados o no
  - Elementos pueden estar ordenados o no según determinado criterio.
- El tipo *Collection* se utiliza para declarar variables o parámetros donde se quiere la máxima generalidad posible.
- La interfaz *Collection* es genérica, por lo que hablaremos de *Collection<E>*.
- Hereda de *Iterable<E>*:
  - Son iterables
  - Se puede utilizar con ellas el *for* extendido



2. La interfaz Collection

API: https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

- `boolean` **`add`**`(E e)`
  - Añade un elemento a la colección, devuelve *false* si no se añade
- `boolean` **`addAll`**`(Collection<? extends E> c)`
  - Añade todos los elementos de c a la colección que invoca. Es el operador unión. Devuelve *true* si la colección original se modifica
- `boolean` **`remove`**`(Object o)`
  - Borra el objeto o de la colección que invoca; si no estuviera se devuelve *false*
- `boolean` **`removeAll`**`(Collection<?> c)`
  - Borra todos los objetos de la colección que invoca que estén en c. Devuelve *true* si la colección original se modifica

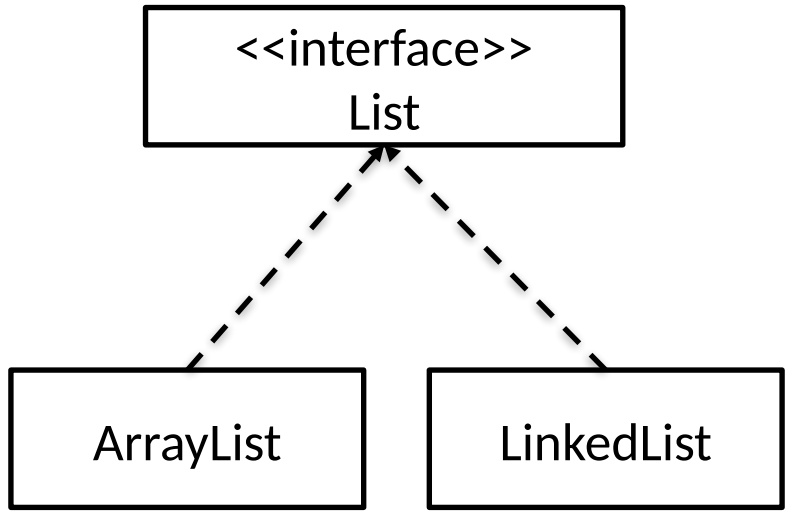
2. La interfaz Collection

- `boolean` **`contains`**`(Object o)`
  - Devuelve *true* si o está en la colección invocante
- `boolean` **`containsAll`**`(Collection<?> c)`
  - Devuelve *true* si la colección que invoca contiene todos los elementos de c
- `boolean` **`isEmpty`**`()`
  - Devuelve *true* si la colección no tiene elementos
- `boolean` **`retainAll`**`(Collection<?> c)`
  - En la colección que invoca sólo se quedarán aquellos objetos que están en c. Por tanto, es la intersección entre ambas colecciones. Devuelve *true* si la colección original se modifica
- `void` **`clear`**`()`
  - Borra todos los elementos de la colección

ÍNDICE

3. El tipo List

- Representan colecciones de elementos en los **que importa cuál es el primero, el segundo, etc**.
- Cada elemento está **referenciado mediante un índice**.
- Las listas pueden contener **elementos duplicados**.
- La interfaz *List* hereda de la interfaz *Collection*.
  - La operación *addAll* añade los elementos de la lista que se pasa al final de la lista sobre la que se invoca.
  - Si en la lista sobre la que se invoca hay elementos duplicados, la operación *removeAll* elimina de esta todas las instancias de los elementos que aparecen en la lista que se pasa.
  - De manera análoga se comporta *retainAll*: si en la lista sobre la que se invoca un elemento aparece *n* veces, y este aparece en la lista que se pasa (independientemente del número de veces que aparezca), en la lista resultado permanecerán las *n* apariciones del elemento.



3. El tipo List


```
import java.util.LinkedList;
import java.util.List;

public class Test {
    public static void main(String[] args) {
        List<String> l1 = new LinkedList<String>();
        List<String> l2 = new LinkedList<String>();
        l1.add("A");
        l1.add("B");
        l1.add("C");
        l2.add("B");
        l2.add("B");
        l1.removeAll(l2);
        System.out.println("l1 después de l1.removeAll(l2): " + l1);
    }
}
```

3. El tipo List


```
import java.util.LinkedList;
import java.util.List;

public class Test {
    public static void main(String[] args) {
        List<String> l1 = new LinkedList<String>();
        List<String> l2 = new LinkedList<String>();
        l1.clear();
        l2.clear();
        l1.add("A");
        l1.add("B");
        l1.add("C");
        l2.add("B");
        l2.add("B");
        l2.retainAll(l1);
        System.out.println("l2 después de l2.retainAll(l1): " + l2);
    }
}
```

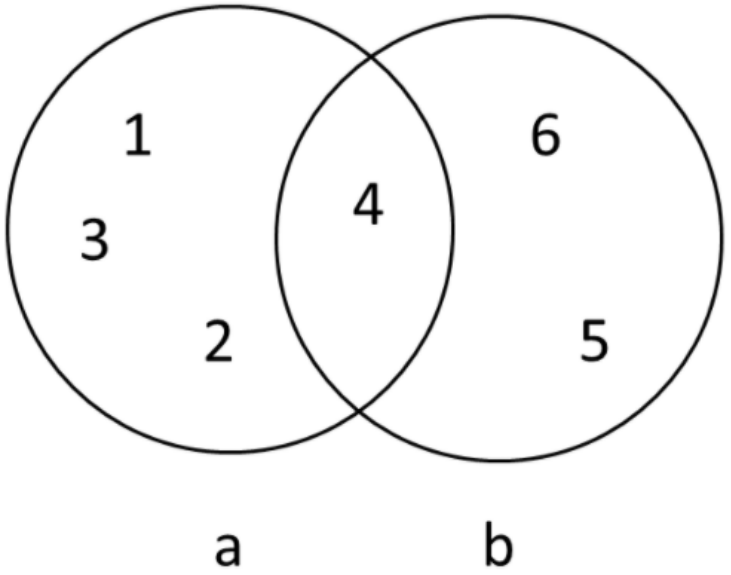
3. El tipo List

- `void add(int index, E element)`
  - Inserta el elemento especificado en la posición especificada
- `E get(int index)`
  - Devuelve el elemento de la lista en la posición especificada
- `int indexOf(Object o)`
  - Devuelve el índice donde se encuentra por primera vez el elemento o (si no devuelve -1)
- `E remove(int index)`
  - Borra el elemento de la posición especificada
- `List<E> subList(int fromIndex, int toIndex)`
  - Devuelve una vista de la porción de la lista entre fromIndex, inclusive, and toIndex, sin incluir.

ÍNDICE

4. El tipo Set

- Corresponde con el concepto matemático de **conjunto**:
  - Agregado de elementos en el que **no hay orden** (no se puede decir cuál es el primero, el segundo, el tercero, etc.)
  - **No puede haber elementos repetidos**

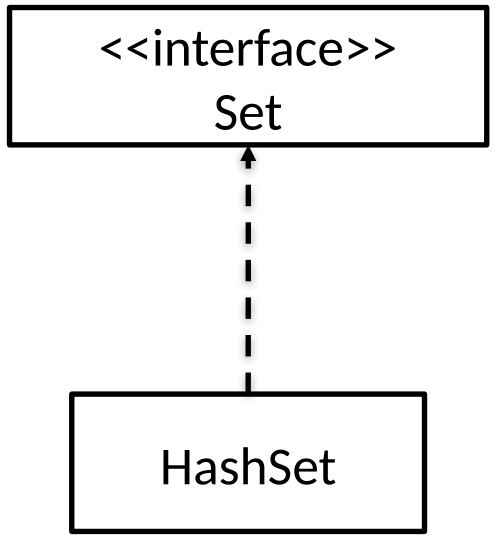



```
a = {1, 2, 3, 4}     b = {4, 5, 6}

a ∪ b = {1, 2, 3, 4, 5, 6}
a - b = {1, 2, 3}
a ∩ b = {4}
```

4. El tipo Set

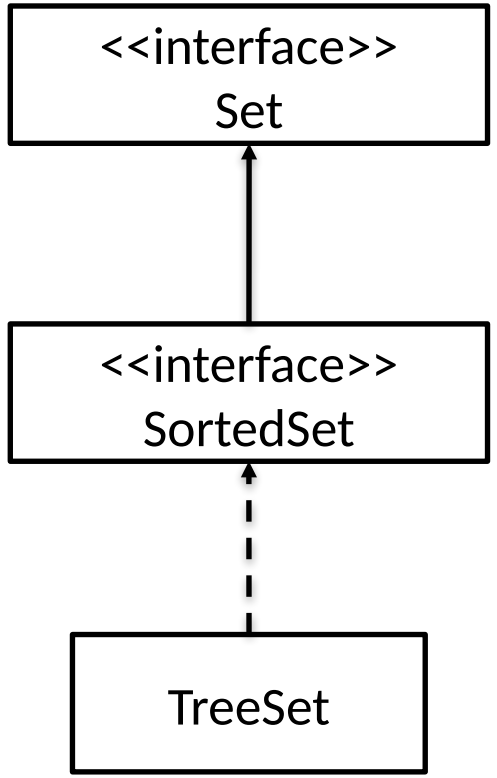
- La interfaz *Set* no aporta ningún método extra a los que ya tiene *Collection*.
  - `boolean addAll(Collection<? extends E> c)`
    - Unión entre conjuntos
  - `boolean retainAll(Collection<?> c)`
    - Intersección entre los dos conjuntos
  - `boolean removeAll(Collection<?> c)`
    - Diferencia de conjuntos
  - `boolean contains(Object o)`
    - Equivale a la pertenencia en conjuntos (∈)
  - `boolean containsAll(Collection<?> c)`
    - Corresponde con la de subconjunto (⊆).
- La implementación más habitual del tipo *Set* es la clase *HashSet*. Tiene dos constructores:
  - Uno vacío, que construye un conjunto vacío
  - Otro que recibe una colección y construye un conjunto con los elementos de la colección (sin duplicados).



ÍNDICE

5. El tipo SortedSet

- Subtipo de los conjuntos (por tanto, los elementos **no están indexados** y **no puede haber elementos repetidos**), en el que **existe una relación de orden** entre los elementos que permite decir cuál va antes y cuál después.
- La implementación de los conjuntos ordenados es la clase *TreeSet*. Esta clase tiene varios constructores:
  - TreeSet()
    - Crea un conjunto ordenado vacío donde los elementos se ordenarán según su orden natural
  - TreeSet(Collection<? extends E> c)
    - Crea un conjunto ordenado con los elementos de la colección ordenados según su orden natural (los elementos de la colección tienen que implementar Comparable).
  - TreeSet(Comparator<? super E> comparator)
    - Crea un conjunto ordenado vacío cuyos elementos se ordenarán según el orden inducido por el comparador
  - TreeSet(SortedSet<E> s)
    - Crea un conjunto ordenado con los mismos elementos que el que recibe como argumento y usando su mismo orden.



5. El tipo SortedSet

- El for extendido sobre los elementos de un SortedSet los devuelve en el orden que tienen inducido:

```
import java.util.SortedSet;
import java.util.TreeSet;

public class Test {
    public static void main(String[] args) {
        SortedSet<Character> ss = new TreeSet<Character>();
        ss.add('X');
        ss.add('C');
        ss.add('R');
        ss.add('Q');

        for (Character ch : ss) { // (Python) for ch in ss:
            System.out.println(ch);
        }
    }
}
```

ÍNDICE

6. La clase de utilidad Collections

- El paquete java.util contiene la clase de utilidad Collections
- `static <T> boolean addAll(Collection<? super T> c, T... elements)`
  - Añade a la colección los elementos indicados en elements.
- `static void fill(List<? super T> l, T o)`
  - Reemplaza todos los elementos de la lista l por o.
- `static <T> max(Collection<? extends T> coll)`
- `static <T> min(Collection<? extends T> coll)`
  - Devuelve el elemento máximo/mínimo de la colección según el orden natural de sus elementos.

6. La clase de utilidad Collections

- `static void reverse(List<?> list)`
  - Invierte los elementos de la lista list.
- `static void shuffle(List<?> list)`
  - Mezcla aleatoriamente los elementos de la lista list.
- `static <T extends Comparable<?super T>> void sort(List<T> list)`
  - Ordena la lista según el orden natural del tipo.

6. La clase de utilidad Collections


```
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

public class Test {
    public static void main(String[] args) {
        List<String> l = new LinkedList<String>();
        l.add("R");
        l.add("T");
        l.add("M");
        System.out.println(l);

        Collections.reverse(l);
        System.out.println(l);

        Collections.sort(l);
        System.out.println(l);

        Collections.fill(l, "X");
        System.out.println(l);
    }
}
```

Ejercicios

- A continuación, cree un fichero de test en el que realice las siguientes operaciones una tras otra:
  - Cree una lista de tipo ArrayList que va a usar para introducir números enteros.
  - Agregue los siguientes números a la lista: 2, 3, 3, 6, 8, 12, 17, 17, 27
  - Elimine los elementos repetidos.
  - Ordene la lista por orden descendente.
  - A continuación, implemente las siguientes funciones que reciben como parámetro una lista de enteros:
    - Implemente una función que devuelva una lista que contenga los números primos de la lista recibida como parámetro.
    - Implemente una función que determine el número más bajo de la lista. Para ello no podrá usar ninguna función ya implementada en java como sort, min, max, etc.

Ejercicios

	name	slots	empty_slots	free_bikes	latitude	longitude
2	149_CALLE ARROYO	20	11	9	37.397829929383	-5.97567172039552
3	257_TORRES ALBARRACIN	20	16	4	37.38376948792722	-5.908921914235877
4	243_GLORIETA DEL PRIMERO DE MAYO	15	6	9	37.380439481169994	-5.953481197462845
5	109_AVENIDA SAN FRANCISCO JAVIER	15	1	14	37.37988413609134	-5.974382770011586
6	073_PLAZA SAN AGUSTIN	15	10	4	37.38951386231434	-5.984362789545622
7	096_CALLE BETIS	19	17	2	37.3835407311529	-5.999910722822249
8	256_MIGUEL MONTORO	29	27	1	37.386100496778624	-5.910090919676698
9	082_CALLE LUIS MONTOTO	17	12	5	37.387098477016366	-5.974843147214583
10	226_AVENIDA DOCTOR EMILIO LEMOS	15	10	5	37.402862871745285	-5.921753676327738
11	103_AVENIDA EDUARDO DATO	20	20	0	37.384432463478554	-5.982738173371375
12	016_CALLE DE MANUEL VILLALOBOS	20	17	3	37.40705079945115	-5.982430505819379
13	209_AVENIDA ALEMANIA	15	3	11	37.34531731164308	-5.982656150368391
14	026_AVENIDA DE MIRAFLORES	17	16	1	37.40249453741322	-5.977678166024575
15	221_AVENIDA ALCALDE LUIS URUÑUELA	15	11	3	37.40560856297865	-5.930524002164018
16	128_VIRGEN DE LORETO	18	6	12	37.37645342505249	-6.000632231559362
17	077_PLAZA CHAPINA	20	13	7	37.389629475123165	-6.007581263540937