

# UT6

## Colecciones de tamaño flexible

Módulo – Programación (1º)

Ciclos – Desarrollo de Aplicaciones Multiplataforma | Desarrollo de Aplicaciones Web

CI María Ana Sanz

---

# Contenidos

---

- Introducción al framework de colecciones en Java
- La clase ArrayList
  - operaciones sobre ArrayList
- El bucle for mejorado
- Iteradores para recorrer colecciones
- Las clase envolventes
  - convertir tipos primitivos en clases wrapper
  - convertir clases wrapper en tipos primitivos
  - String y clases envolventes

# Contenidos

---

- La clase HashSet (TreeSet / LinkedHashSet)
  - Operaciones sobre HashSet
- La clase HashMap (TreeMap / LinkedHashMap)
- Generar documentación de nuestras clases
- Creación de paquetes
  - Ficheros jar
- Más sobre colecciones
  - clase Stack, clase LinkedList
  - clase Collections

# Ventajas / Desventajas de los arrays

## ■ Arrays

- son objetos
- tamaño fijo - `String[] array = new String[10];`
  - se puede aumentar su tamaño creando un nuevo array  
`String[] nuevoArray = new String[20]; array = nuevoArray;`
- valores de tipos primitivos y tipos referencia (objetos)
- accesos rápidos por índice (valor entero a partir de 0)
- inserciones / borrados poco eficientes
  - hay que desplazar elementos
- colecciones de tamaño fijo del mismo tipo de elementos

# Qué vamos a ver?

- **Collection Framework (framework de colecciones)**
  - conjunto de clases, interfaces y métodos estáticos (algoritmos) para procesar colecciones
- Qué es una **colección**?
  - un contenedor de objetos
    - un objeto que agrupa a muchos otros objetos

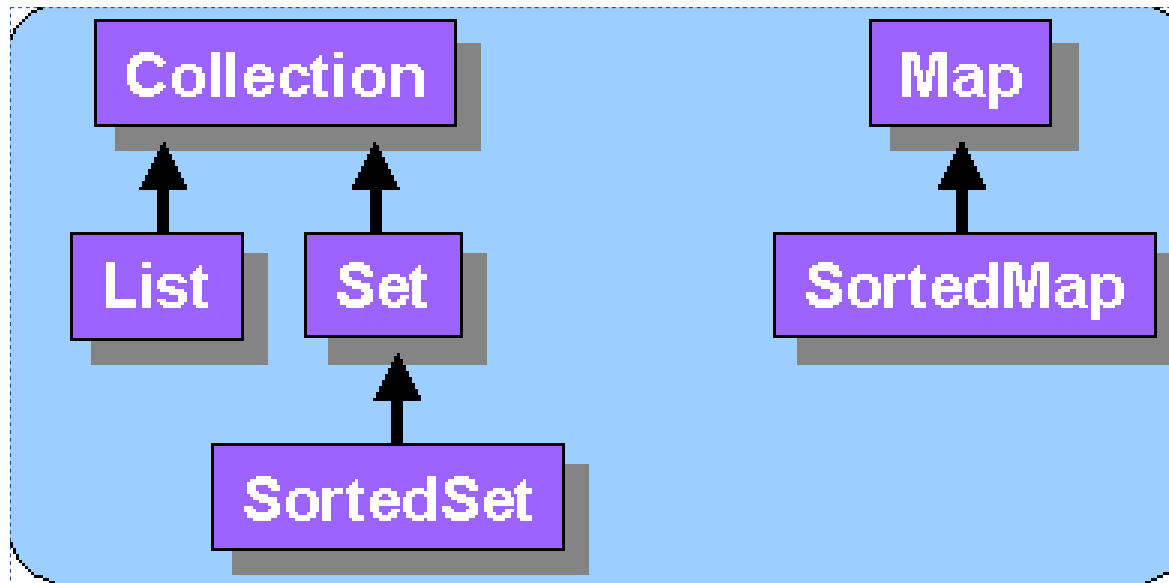
Un array es una coleccion de tamaño fijo, que tambien admite datos de tipo primitivo.

# Collection Framework

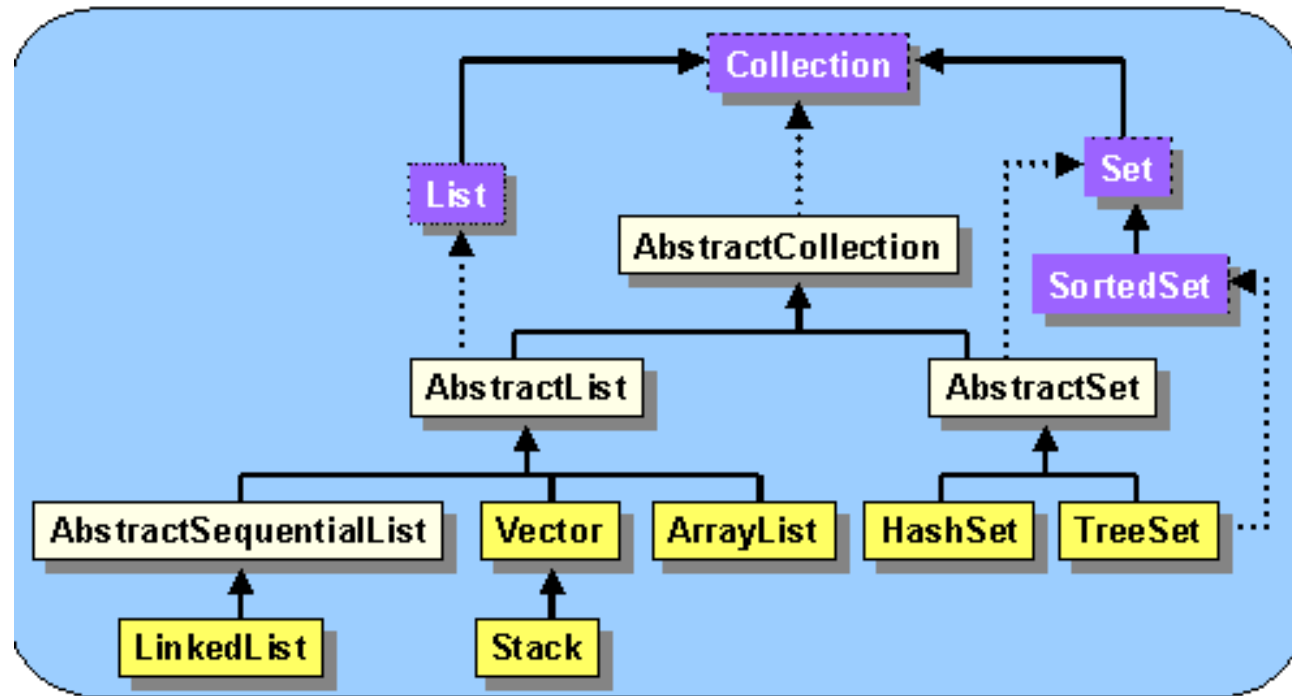
- Arquitectura multicapa para representar y manipular colecciones
  - **Interfaces**
    - tipos abstractos que representan las colecciones
    - describen **qué** se puede hacer con ellas, **no cómo** se hace
    - son especificaciones, contratos
  - **Implementaciones**
    - implementaciones concretas de los interfaces
    - clases que los implementan
  - **Algoritmos**
    - métodos (muchos estáticos) para realizar operaciones comunes sobre las colecciones
      - ordenar, buscar, ....
- Es posible extender nuevas clases a partir de las existentes

# Collection Framework en Java

 Interfaces



# Collection Framework en Java



**Interfaces**



**Clases abstractas**

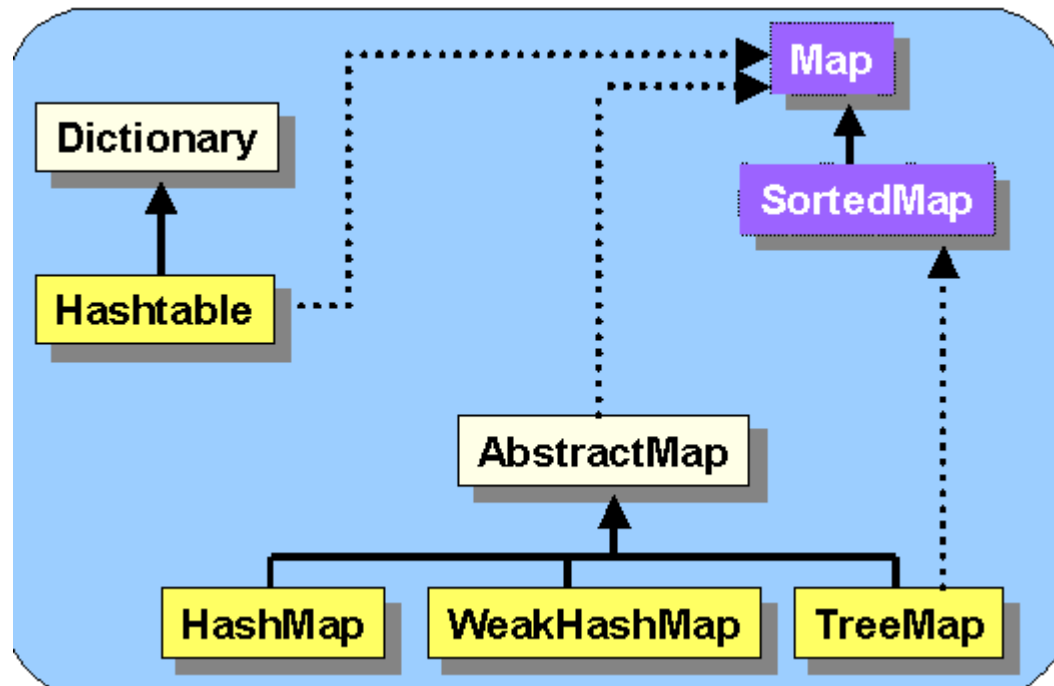


**Clases concretas**

Detrás de las colecciones subyacen estructuras de datos: (arrays, listas enlazadas, árboles, tablas hash, ...)



# Collection Framework en Java



**Interfaces**



**Clases abstractas**



**Clases concretas**

Detrás de las colecciones subyacen estructuras de datos: (arrays, listas enlazadas, árboles, tablas hash, ...)

# La clase ArrayList

- en **java.util**
- una colección ArrayList es un **objeto**, un contenedor de otros objetos
- **Modela una colección de**
  - objetos
  - contiene un nº arbitrario de objetos
  - crece / decrece dinámicamente a medida que se añaden/borran objetos de la colección
  - contador interno que indica el nº de elementos de la colección
  - mantiene el orden de los elementos (tal como se insertaron)
    - los elementos de la colección se recuperan en el orden en que se introdujeron
    - es una secuencia ordenada
  - admite duplicados

# La clase Agenda

---

- permite guardar notas
- no tiene límite en el nº de notas que puede almacenar
- permite mostrar notas individuales
- permite indicar cuántas notas hay almacenadas

# La clase Agenda

```
import java.util.ArrayList;
import java.util.Iterator;
/**
 * Una clase que mantiene una lista
 * con un nº arbitrario de notas.
 * Las notas se numeran de forma externa
 * por el usuario
 */
public class Agenda
{
    // Almacén de notas
    private ArrayList<String> notas;
    /**
     * Constructor
     */
    public Agenda()
    {
        notas = new ArrayList<String>();
    }
}
```

# La clase Agenda

```
import java.util.ArrayList;
import java.util.Iterator;
/**
 * Una clase que mantiene una lista
 * con un nº arbitrario de notas.
 * Las notas se numeran de forma externa
 * por el usuario
 */
public class Agenda
{
    // Almacén de notas
    private ArrayList<String> notas;
    /**
     * Constructor
     */
    public Agenda()
    {
        notas = new ArrayList<>();
    }
}
```

A partir de Java 7  
se puede omitir al  
instanciar la  
colección el tipo (el  
compilador infiere  
el tipo)

# La clase Agenda

- **import java.util.ArrayList;**

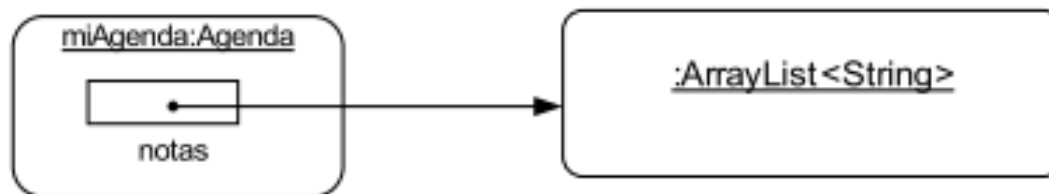
importar el paquete

- **private ArrayList<String> notas;**

especificar tipo de los objetos que contendrá la colección.

- **miAgenda = new Agenda();**

Colecciones **genéricas**,  
**parametrizadas**

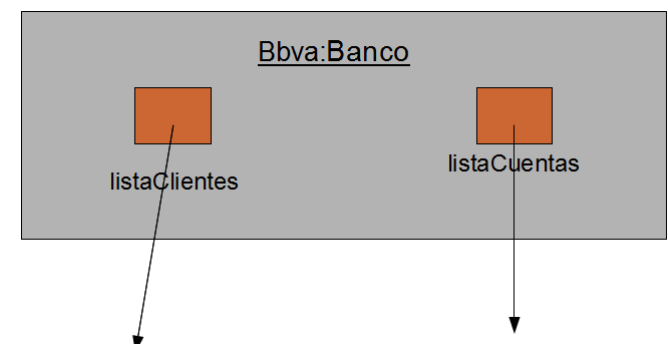


# Ejer 6.1

## ■ Ejer 6.1

```
import java.util.ArrayList;
public class Banco
{
    private ArrayList<Cliente> listaClientes;
    private ArrayList<Cuenta> listaCuentas;
    /**
     * Constructor
     */
    public Banco()
    {
        listaClientes = new ArrayList<Cliente>();
        listaCuentas = new ArrayList<Cuenta>();
    }
    .....
}
```

Banco bbva = new Banco();



# Operaciones sobre ArrayList

**public int size()** Tamaño de la colección

**public boolean add(E elemento)** añadir un elemento (un objeto de tipo E) a la colección

**public E get(int index)** Obtener el elemento de posición *index*

**public E remove(int index)** Borrar el elemento de posición *index*

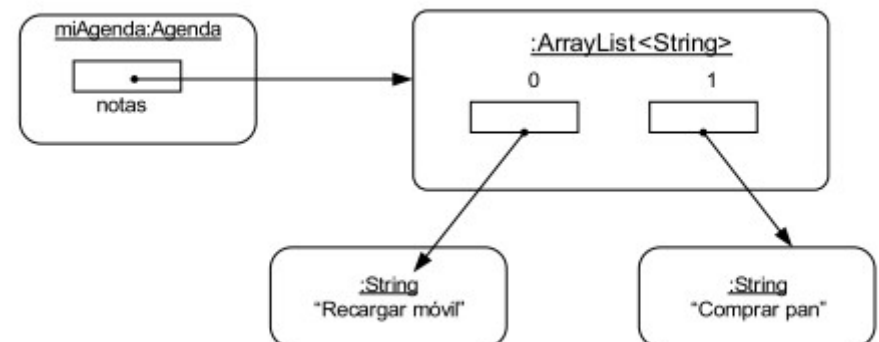


# Añadir un objeto a la colección ArrayList: add()

```
/**
 * Almacenar una nueva nota
 * @param nota La nota que se almacena
 */
public void añadirNota(String nota)
{
    notas.add(nota);
}
```

Objeto que se añade a la colección. Se **añade al final de la colección**. El tamaño se incrementa automáticamente.

```
miAgenda = new Agenda();
miAgenda.añadirNota("Recargar móvil");
miAgenda.añadirNota("Comprar pan");
```



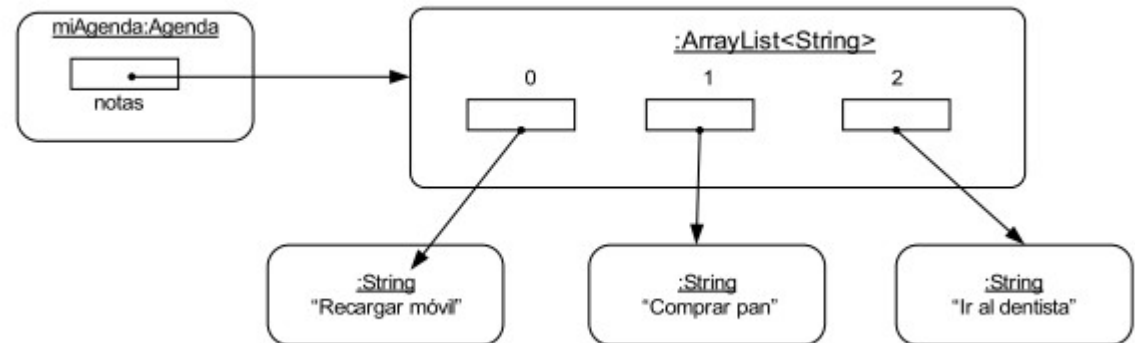
# Añadir un objeto a la colección ArrayList: add()

```
/**
 * Almacenar una nueva nota
 * @param nota La nota que se almacena
 */
public void añadirNota(String nota)
{
    notas.add(nota);
}
```

El objeto ArrayList contiene 2 atributos enteros y un array de objetos que inicialmente tiene un tamaño de 0 y va incrementando 5 en 5 su tamaño.

Objeto que se añade  
A la colección. Se añade al final  
de la colección.  
El tamaño se incrementa  
automáticamente.

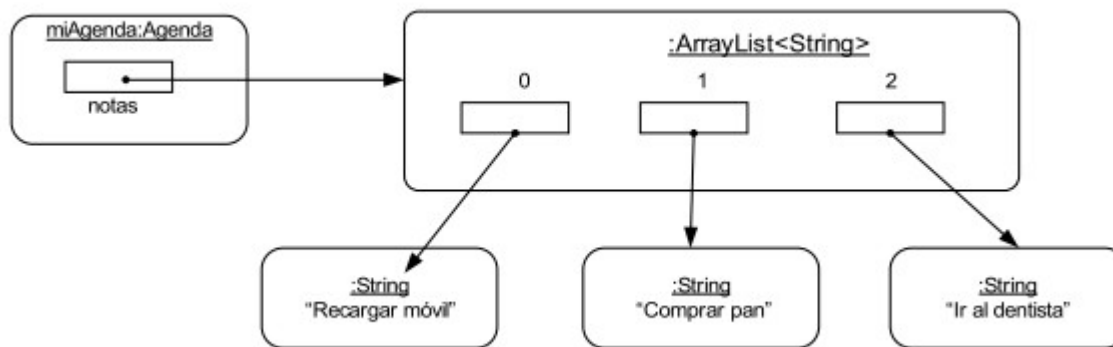
```
miAgenda.añadirNota("Ir al dentista");;
```



# Obtener el tamaño de la colección ArrayList: size()

```
/**  
 * @return El nº de notas actualmente almacenadas  
 */  
public int numeroNotas()  
{  
    return notas.size();  
}
```

Devuelve el tamaño actual, nº de objetos que almacena



Tamaño 3

# Obtener un elemento de la colección ArrayList: get()

```
/**
 * Mostrar una nota
 * @param numeroNota El nº de nota a mostrar
 */
public void mostrarNota(int numeroNota)
{
    if (numeroNota >= 0 && numeroNota < numeroNotas()) {

        System.out.println((numeroNota + 1) + " " +
                           notas.get(numeroNota));
    }
    else {
        System.out.println("Indice incorrecto");
    }
}
```

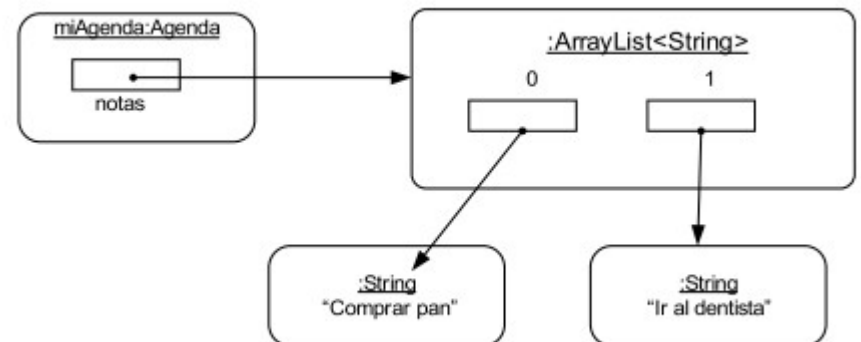
miAgenda.mostraNota(1) ----- “Comprar pan”.  
devuelve un String

Posición de los objetos  
en la colección:  
**0 – primero**  
**size() - 1 – último**

# Borrar un elemento de la colección ArrayList: remove()

```
/**
 * Mostrar una nota
 * @param numeroNota El nº de nota a borrar
 */
public void borrarNota(int numeroNota)
{
    if (numeroNota >= 0 && numeroNota < numeroNotas()) {
        notas.remove(numeroNota);
    }
    else {
        System.out.println("Indice incorrecto");
    }
}
```

miAgenda.borrarNota(0)



# Procesando la colección completa

```
/**
 * Mostar todas las notas
 */
public void listarNotas()
{
    int cuantas = notas.size();
    int indice = 0;
    while (indice < cuantas) {
        System.out.println((indice + 1) + " " + notas.get(indice));
        indice++;
    }
}
```

# El bucle for mejorado (a partir de Java 1.5)

```
for (tipo elemento: colección) {  
    .....  
}
```

Para recorrer  
colecciones y  
arrays

**tipo** – representa el tipo de los objetos almacenados en la colección (String, Cliente, ...). Si se trata de un array es el tipo de los elementos del array

**elemento** – es el nombre de la variable asociada al bucle que tomará uno a uno los valores de la colección

**colección** – es la colección (o el array) que va ser recorrida

# El bucle for mejorado (a partir de Java 1.5)

```
public void listarNotas()
{
    for (String nota: notas) {
        System.out.println(nota);
    }
}
```

```
public double calcularMedia(int[ ] numeros)
{
    int suma = 0;
    for (int num: numeros) {
        suma += num;
    }
    return suma / (double) numeros.length;
}
```



# Otros métodos de ArrayList

**public boolean isEmpty()** *true* si la colección no contiene elementos

**public E set(int indice, E elemento)** reemplaza el **valor** de la posición *indice* por *elemento*, devuelve el elemento que estaba previamente en esa posición

**public boolean contains(Object obj)** devuelve *true* si el objeto *obj* está contenido en la colección

**public int indexOf(Object obj)** devuelve el índice de la primera ocurrencia del objeto *obj* en la colección

**public void add(int indice, E elemento)** inserta el elemento E en la posición especificada por *índice*  
y los demas avanzan una posicion a la derecha

**public Iterator<E> iterator()** devuelve un objeto iterador para recorrer los elementos de la colección en secuencia

# Ejer. Curso Estudiante

---

Curso Estudiante

# Iterador para recorrer una colección

- Es posible recorrer una colección `ArrayList` utilizando un mecanismo especial, un objeto `Iterator`
  - se recorre así la colección sin utilizar índices
- El uso de iteradores no es específico de `ArrayList` sino de muchas otras colecciones (`HashSet`, `HashMap`, ....)
- Qué es un **iterador**?

Un objeto de tipo `Iterator` que permite recorrer una colección

# Iterador para recorrer una colección

- método **iterator()** de `ArrayList`
  - devuelve un objeto de tipo `Iterator`
  - la clase `Iterator` en *java.util*
- Métodos de la clase `Iterator` (es realidad es un interfaz)
  - **hasNext()**
    - devuelve true si queda aún elementos en la colección que se está recorriendo
  - **next()**
    - devuelve el siguiente elemento de la colección
  - **remove()**
    - borra el último elemento de la colección que fue obtenido con `next()`

# listarNotasConIterador()

```
import java.util.ArrayList;  
import java.util.Iterator;
```

importar la clase

```
.....  
public void listarNotasConIterador()  
{
```

```
    Iterator<String> it = notas.iterator();
```

```
    while (it.hasNext()) {
```

```
        String nota = it.next();
```

```
        System.out.println(nota);
```

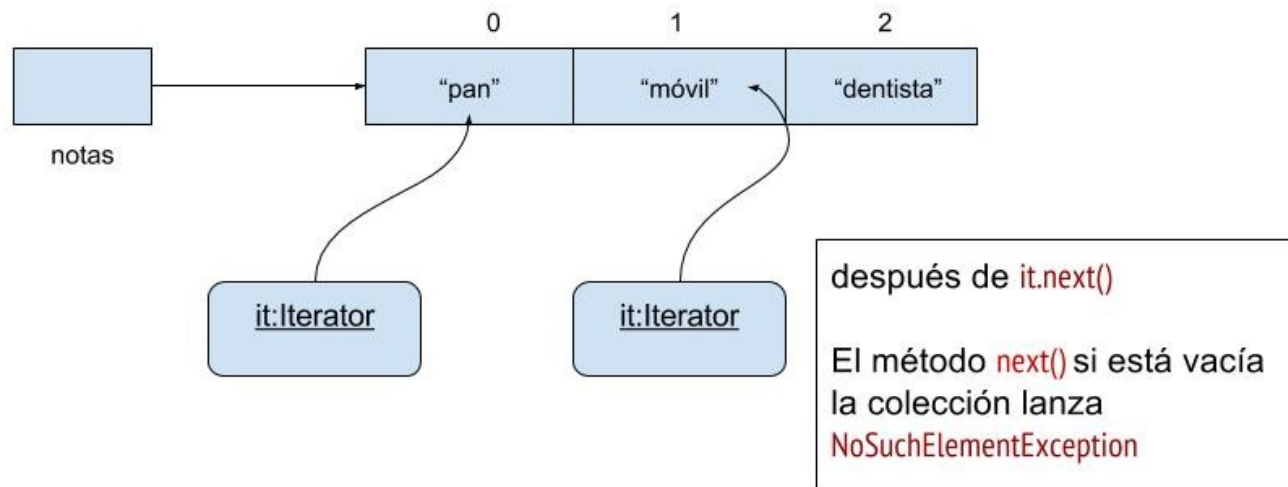
```
    }
```

```
}
```

definir el  
iterador /  
obtener el  
objeto  
Iterator, se le  
pide a la  
colección

obtener el siguiente objeto de la colección, se pide al  
iterador

# listarNotasConIterador()



# Recorrido completo de una colección ArrayList

- Métodos para recorrer una colección ArrayList
  - con *while* (utilizando índices)
  - con *for* normal (con índices)
  - con *for* mejorado (sin índices)
  - con *iterador* (sin índices)
- Para borrar varios elementos de una colección ArrayList
  - con *while*
  - con *iterador*
  - con *for normal* empezando desde el final
  - **no** con *for normal* empezando desde el principio
  - **no** con *for mejorado*

**Nota** – con la introducción a partir de Java 8 de los *interfaces funcionales*, *expresiones lambda* y *Streams* que permiten la **programación funcional** hay otros recorridos posibles para las colecciones (*iteración interna*) que de momento no veremos.

En la 3ª evaluación cuando introduzcamos el concepto de interfaz funcional y expresión lambda veremos algún ejemplo.

## Ejer. 6.2

```
import java.util.Iterator;

.....
private ArrayList<Estudiante> listaEstudiantes;

.....
listaEstudiantes = new ArrayList<Estudiante>();

.....
public void borrarMenoresDeEdad()
{
    Iterator<Estudiante> it = listaEstudiantes.iterator();
    while (it.hasNext()) {
        Estudiante e = it.next();
        if (e.getEdad() < 18) {
            it.remove();
        }
    }
}
```



## Ejer. 6.2

```
public void borrarMenoresDeEdad()
{
    int indice = 0;
    while (indice < listaEstudiantes.size()) {
        Estudiante e = listaEstudiantes.get(indice);
        if (e.getEdad() < 18){
            listaEstudiantes.remove(indice);
        }
        else {
            indice++;
        }
    }
}
```

# Uso de índice o iteradores?

---

- Además de `ArrayList` Java proporciona otras muchas colecciones
- El uso de índices con alguna de ellas es muy ineficiente
- En esos casos hay que usar iterador
- Los iteradores están disponibles en todas las colecciones del lenguaje

# Ejercicios

---

Ejer. 6.3 Proyecto Club

Ejer. 6.4 Gestor Stock Producto

EJAD01 Urna y Bola

EJAD02 Biblioteca

## Ejer 6.20

```
/**
 *
 * @param pila la pila de la que desempilaremos
 * @return la cadena formada por los caracteres
 *         que se desempilan
 */
private String desempilar(Stack<Character> pila)
{
    StringBuilder sb = new StringBuilder();
    while (!pila.empty()) {
        sb.append(pila.pop());
    }
    return sb.toString();
}
```

# Clases envolventes (Wrapper classes)

---

- Las clases que representan colecciones como `ArrayList` permiten almacenar objetos
- Java distingue entre tipos primitivos y tipos referencia.
- ¿Qué podemos hacer si queremos guardar valores de tipo `int`, por ejemplo, en una colección?
  - Utilizar las **clases envolventes** o *wrapper classes*.

# Convirtiendo tipos primitivos en clases

- Cada tipo simple (primitivo) en Java tiene su correspondiente clase envolvente
  - representa al mismo tipo, “*envuelve*” el valor del tipo en un objeto.

Tipo primitivo	Clase envolvente (wrapper)
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

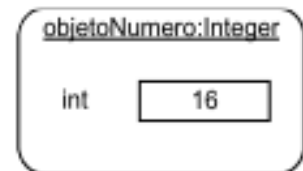
# Convirtiendo tipos primitivos en clases

- Cada clase tiene un constructor que toma como argumento un valor del tipo primitivo que representa.
  - Los valores de los tipos primitivos se convierten en objetos llamado al constructor apropiado.

```
int numero = 16;
```

```
Integer objetoNumero = new Integer(numero);
```

```
Character objetoCharacter = new Character('M');
```



- Los objetos de las clases wrapper son **inmutables**
  - no cambian una vez han sido creados

no setters

# Convirtiendo clases envolventes en tipos primitivos

- De objeto a tipo primitivo
  - `intValue()` para `Integer`
  - `charValue()` para `Character`
  - `doubleValue()` para `Double`

```
Integer objetoNumero = new Integer(7);  
int numero = objetoNumero.intValue(); // devuelve 7
```

- Todas las clases envolventes incluyen un método `toString()`

```
String strNumero = objetoNumero.toString(); // devuelve "7"
```

getter



# Colecciones y tipos primitivos

- Definir una colección `ArrayList` de enteros
  - `ArrayList<Integer> listaEnteros = new ArrayList<>();`
- Añadir un entero a la colección
  - `listaEnteros.add(new Integer(6));`
- Recuperar un elemento de la colección
  - `int numero = listaEnteros.get(0).intValue();`

# Autoboxing / unboxing

- A partir de Java 1.5
  - no se especifica explícitamente el proceso de *wrapping* / *unwrapping*
  - Se efectúa el **autoboxing / unboxing** por parte del compilador
- **Autoboxing** - *envolver* – de tipo primitivo a objeto. Se aplica cuando
  - al pasar un valor de tipo primitivo como parámetro a un método que espera un tipo objeto de una clase wrapper  
`listaEnteros.add(6);` // en lugar de `listaEnteros.add(new Integer(6))`
  - al asignar un valor de un tipo primitivo a una variable de tipo wrapper  
`Integer numero = 7;` // en lugar de `Integer numero = new Integer(7);`

# Autoboxing / unboxing

- **Unboxing** - *desenvolver* – de tipo objeto a tipo primitivo. Se aplica cuando
  - un valor de una clase wrapper se pasa como parámetro a un método que espera un tipo primitivo

```
public void sumar(int valor)
.....
sumar(new Integer(7));
// en lugar de sumar(new Integer(7).intValue())
```

- se almacena en una variable de tipo primitivo un valor de tipo wrapper

```
int numero = listaEnteros.get(0);    // en lugar de listaEnteros.get(0).intValue()
```

realiza proceso de unboxing al comparar, no ==, solo pasa al llamar al constructor.  
Cuando haces la reacion sin llamar al constructor, java ubica los objetos en memoria, tengo en este pull algun objeto que valga 10, si.

# !!! Atención unboxing !!!

**Atención! -**

**El proceso de *unboxing* no funciona con `==` ni con `!=`**

**(se hace automáticamente cuando se asigna, en la correspondencia de parámetros, en valores de retorno pero **NO** al comparar)**

**Si tenemos:**

```
ArrayList<Integer> lista1;  
ArrayList<Integer> lista2;
```

**y hacemos**

```
if (lista1.get(0) == lista2.get(0))
```

**se comparan referencias,  
no se hace *unboxing* convirtiéndolo a *int***

realiza proceso de unboxing al comparar, no `==`, solo pasa al llamar al constructor.

Cuando haces la reaccion sin llamar al constructor, java ubica los objetos en memoria, tengo en este pull algun objeto que valga 10, si.

usar `equals`

## Ejer. 6.5

```
public class ColeccionEnteros
{
    private ArrayList<Integer> miLista;

    /**
     * Constructor
     */
    public ColeccionEnteros()
    {
        miLista = new ArrayList<>();
        inicializarColeccion();
    }
}
```

## Ejer. 6.5

```
private void inicializarColeccion()
{
    int total = 0;
    int alea = (int) (Math.random() * 21);
    while (alea != 0 && total < 10) {
        total++;
        miLista.add(alea); // miLista.add(new Integer(alea));
        alea = (int) (Math.random() * 21);
    }
}
```

## Ejer. 6.5

```
public int sumar()
{
    int total = 0;
    Iterator<Integer> it = miLista.iterator();
    while (it.hasNext()) {
        // Integer entero = it.next();
        // total += entero.intValue();
        int entero = it.next();
        total += entero;
    }
    return total;
}
```

## Ejer. 6.5

```
public int sumarV2()
{
    int total = 0;
    for (int entero: miLista) {    //hace automáticamente el unboxing
        total += entero;
    }
    return total;
}
```



## Ejer. 6.5

```
public String toString()
{
    StringBuilder sb = new StringBuilder();
    for (int entero: miLista) {    //hace automáticamente el unboxing
        sb.append(entero + " , ");
    }
    return sb.toString();
}
```

# Strings y clases envolventes – String a objeto wrapper

- Convertir un string que representa un valor numérico en un objeto wrapper - Ej. “123” en un objeto Integer
- Todas las clases envolventes incluyen el método estático **valueOf(String s)**
  - crea un nuevo objeto inicializándolo al valor representado por el string
  - `Double obj = Double.valueOf("12.4");`
  - `Integer objOtro = Integer.valueOf("13");`
- Se genera un error, excepción `NumberFormatException`, si el string no representa un nº válido
  - `Integer.valueOf("5,45")` genera un error.

# Strings y clases envoltantes – String a tipo primitivo

- Convertir un `String` directamente a un tipo primitivo
  - `int numero = Integer.valueOf("653").intValue();` 0
  - `int numero = Integer.valueOf("653");` haciendo unboxing
- Todas las clases envoltantes proporcionan métodos estáticos que obtienen directamente el tipo primitivo que corresponde a un string:
  - **`parseInt()`, `parseDouble()`, .....**  
`int numero = Integer.parseInt("653");`

## Ejer. 6.6 / 6.7 / 6.8

- Ejer 6.6

```
Integer unEntero = new Integer(2);  
Integer[] arrayEnteros =  
    {new Integer(1), new Integer(2), new Integer(3)};
```

- Ejer 6.8

- ```
TextField txtNumero = new TextField(25);  
String strNumero = txtNumero.getText();  
int numero = Integer.valueOf(strNumero).intValue();
```

## Ejer. 6.6 / 6.7 / 6.8

---

- Ejer 6.7

if (Character.isLetter(c)) // c es de tipo char

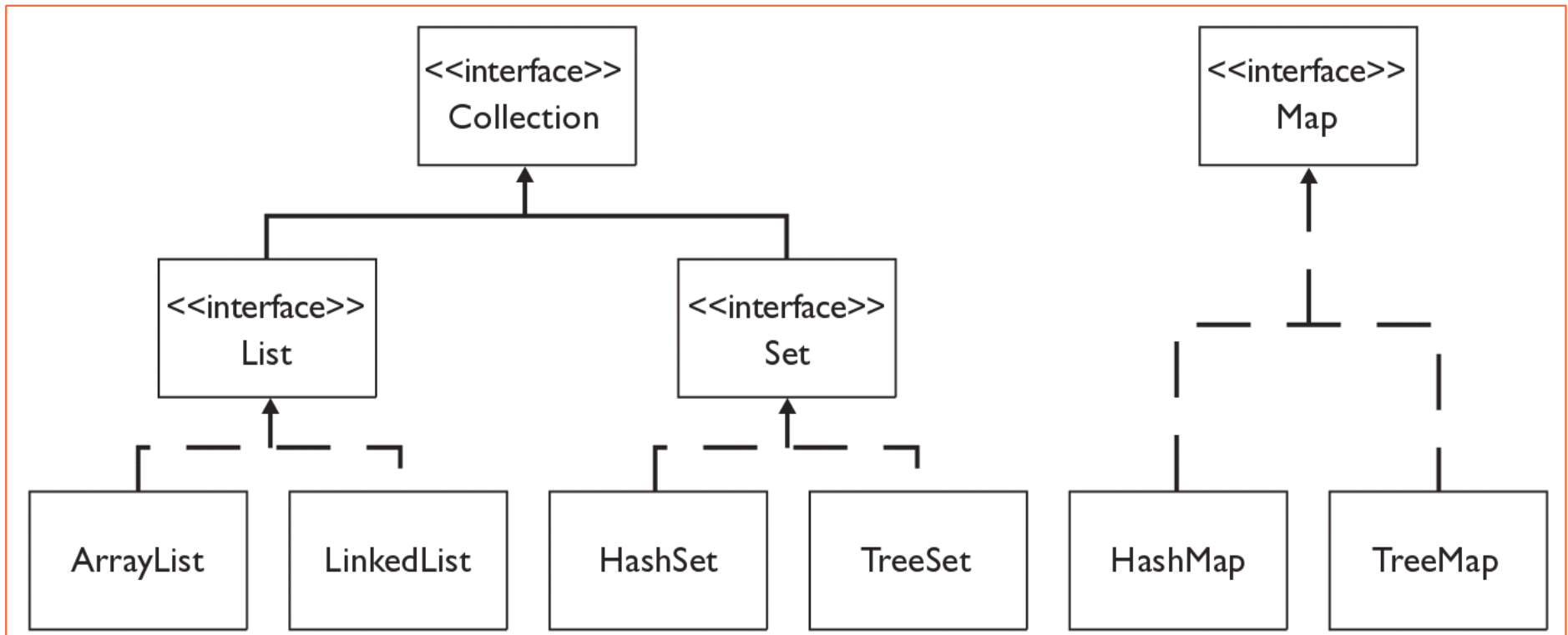
if (Character.isLowerCase(c)) // c es de tipo char

# Ejercicios

---

EJAD03 Lista de objetos enteros (Fusión / Intersección)

# ArrayList versus arrays



# ArrayList versus arrays

| arrays                                                  | ArrayList                                                                                                         |
|---------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|
| <b>Límite de capacidad</b>                              | Sin límites en cuanto al tamaño. Inicialmente capacidad 10, si se llena se reemplaza por uno nuevo redimensionado |
| <b>Más rápidos</b>                                      | Menos eficientes que los arrays (más lentas)                                                                      |
| <b>Utilizar si se sabe de antemano el tamaño máximo</b> | Utilizar si no se sabe de antemano el tamaño máximo                                                               |
| Almacena tipos primitivos y objetos                     | Almacena solo objetos                                                                                             |
|                                                         | Estructura subyacente: un array                                                                                   |



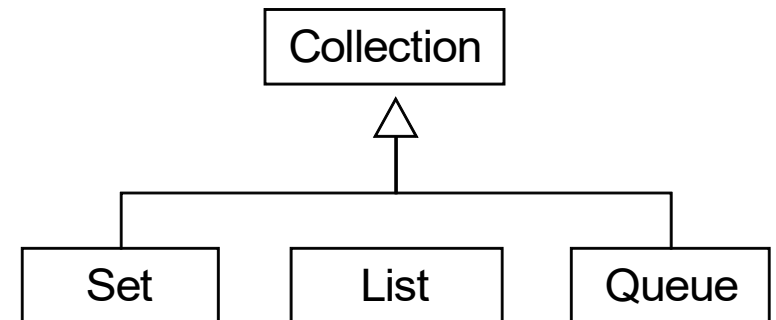
# Las colecciones HashSet / TreeSet

## ■ Interface List

- colecciones ordenadas (primero, segundo, ...)
- admite duplicados
- ArrayList, LinkedList

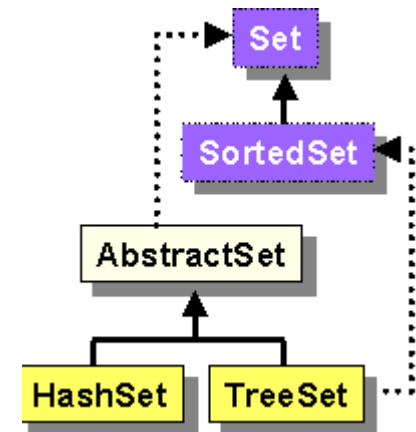
## ■ Interface Set

- abstracción matemática *conjunto*
- colecciones de objetos sin duplicados
  - HashSet – colección de objetos no ordenados
  - TreeSet – colección de objetos ordenados



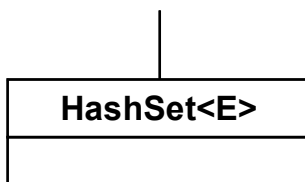
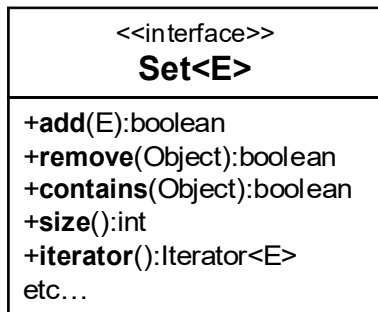
# La colección HashSet

- Colección flexible que almacena objetos
- especialización del interface Set
- colección no ordenada de objetos únicos, no se permiten duplicados
  - no se indica ningún índice para acceder a sus elementos
  - no hay método `get()` para acceder a un set
- estructura subyacente – *tabla hash*
  - *array + lista enlazada*
- en `java.util`



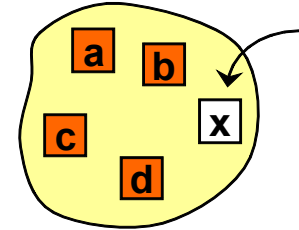
# La colección HashSet

- Operaciones sobre un conjunto
  - recorrer el conjunto (sin índices)
    - con iterador o for mejorado
  - añadir un nuevo objeto al conjunto
  - ver si está vacío
  - comprobar si un elemento pertenece o no al conjunto
  - borrar un elemento del conjunto



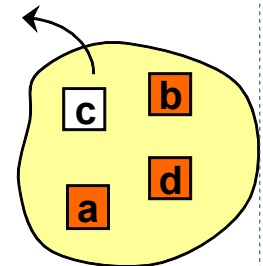
**add(x)**  
→ true

**add(b)**  
→ false



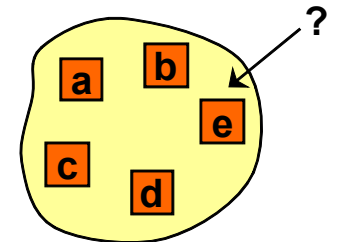
**remove(c)**  
→ true

**remove(x)**  
→ false



**contains(e)**  
→ true

**contains(x)**  
→ false



**isEmpty()**  
→ false

**size()**  
→ 5

# La colección HashSet

```
import java.util.HashSet;
```

```
HashSet<String> conjuntoNombres = new HashSet<>();
```

```
conjuntoNombres.add("Pedro");  
conjuntoNombres.add("Juan");  
conjuntoNombres.add("Pedro");
```



no se añade porque  
ya está en el  
conjunto

```
System.out.println("Nº de nombre en el conjunto " +  
    conjuntoNombres.size());
```

```
for (String nombre : conjuntoNombres) {  
    System.out.print(nombre + "\t");  
}
```

a) Pedro Juan  
b) Juan Pedro



el orden en que  
se muestran es  
indeterminado

# Métodos de Set (HashSet)

**public boolean add()** añade el objeto al conjunto. Devuelve *true* si se ha añadido, *false* si no (porque había un duplicado)

**public boolean remove(E elemento)** borra el elemento especificado del conjunto. *equals()* y *hashCode()* tienen que estar redefinidos en la clase E

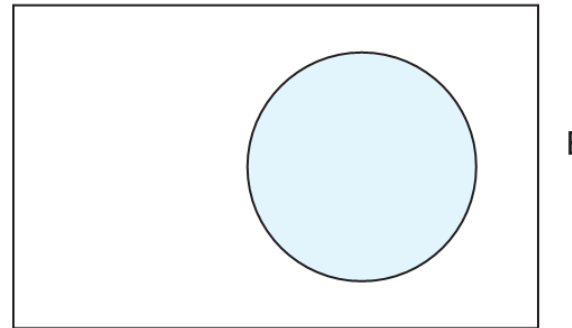
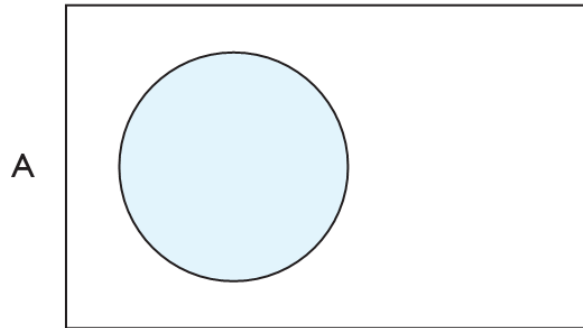
**public int size()** devuelve el nº de objetos del conjunto

**public boolean isEmpty()** devuelve *true* si el conjunto está vacío

**public boolean contains(E elemento)** devuelve *true* si elemento está en el conjunto. *equals()* y *hashCode()* tienen que estar redefinidos en la clase E

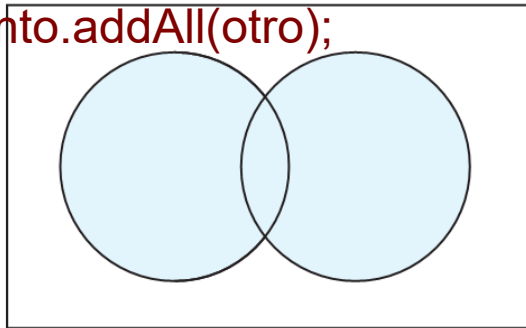
**public Iterator<E> iterator()** devuelve un objeto iterador para recorrer los elementos del conjunto

# Otros métodos de Set (HashSet)



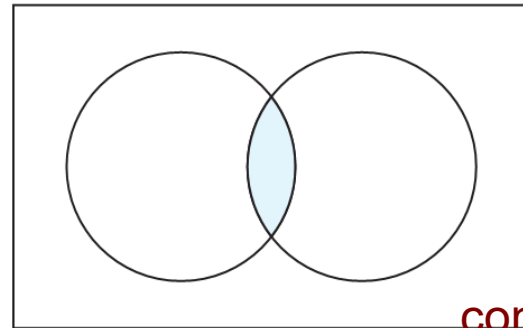
`conjunto.addAll(otro);`

$A \cup B$   
Union



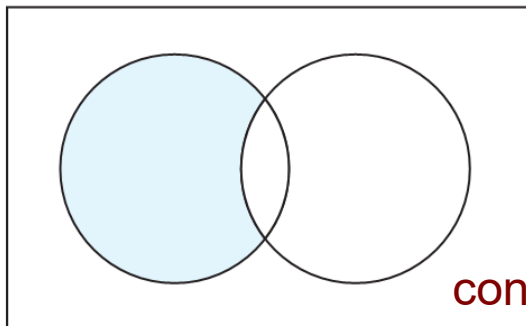
$A \cap B$   
Intersect

`conjunto.retainAll(otro);`



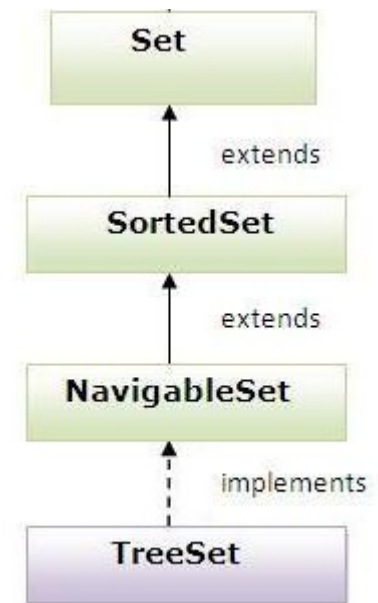
$A - B$   
Difference

`conjunto.removeAll(otro);`



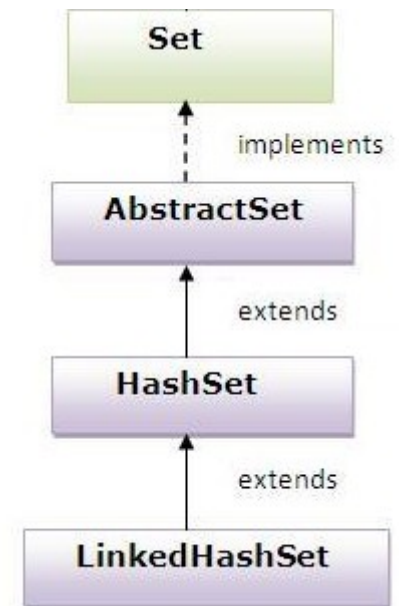
# La clase TreeSet

- Funcionalidad idéntica a HashSet
  - colección que no admite duplicados pero
  - las claves se recuperan en orden
- Estructura subyacente de un TreeSet
  - un árbol binario



# La clase `LinkedHashSet`

- Funcionalidad idéntica a `HashSet`
  - colección que no admite duplicados pero
  - las claves se recuperan en el orden en que se introdujeron
- Estructura subyacente de un `LinkedHashSet`
  - tabla hash + lista doblemente enlazada





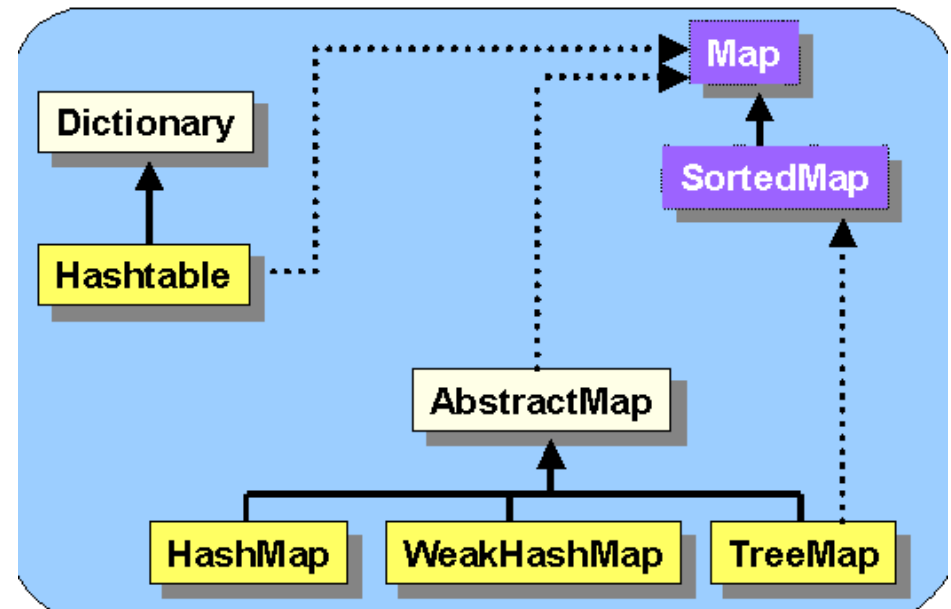
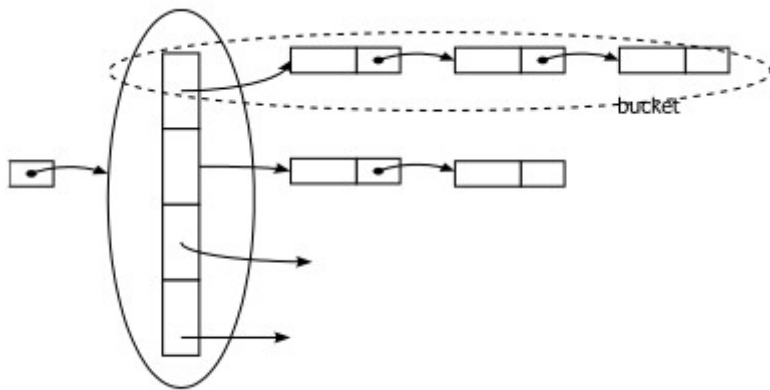
# Ejercicios

---

EJAD04 Conjunto de hobbies y personas

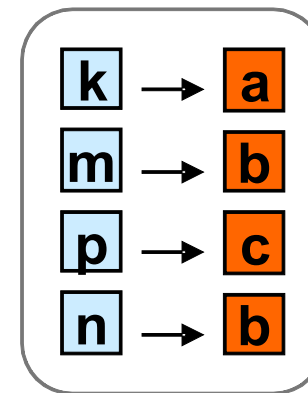
# La clase HashMap

- Implementa el interfaz Map
  - los map asocian claves con valores
- en java.util
- estructura subyacente
  - una tabla hash



# La clase HashMap

- Un map es
  - una colección de objetos denominados **entradas**
- Cada entrada es un par de objetos
  - el objeto **clave**
  - el objeto **valor** asociado a la clave
- Podemos pensar en un map como en un array asociativo
  - los índices en lugar de ser enteros son objetos (las claves)



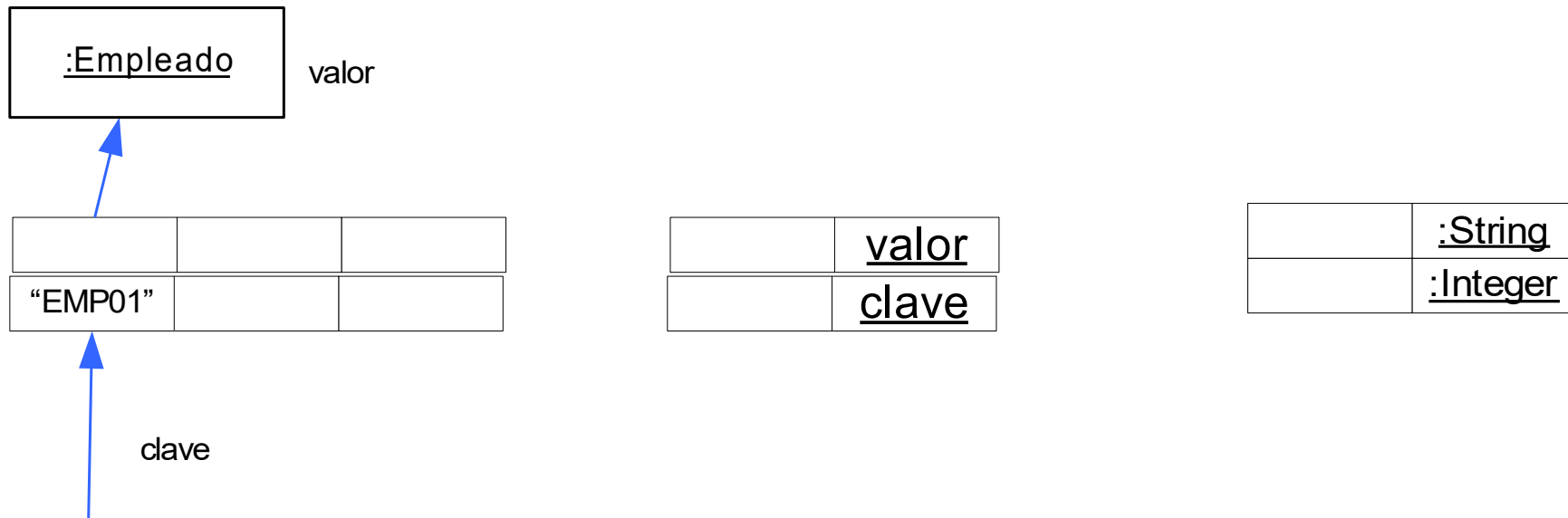
**keys**   **values**

# La clase HashMap

|                 |               |         |  |  |              |
|-----------------|---------------|---------|--|--|--------------|
| <u>"pencil"</u> | <u>"book"</u> | "table" |  |  | <u>valor</u> |
| "lápiz"         | "libro"       | "mesa"  |  |  | <u>clave</u> |

entrada

# La clase HashMap



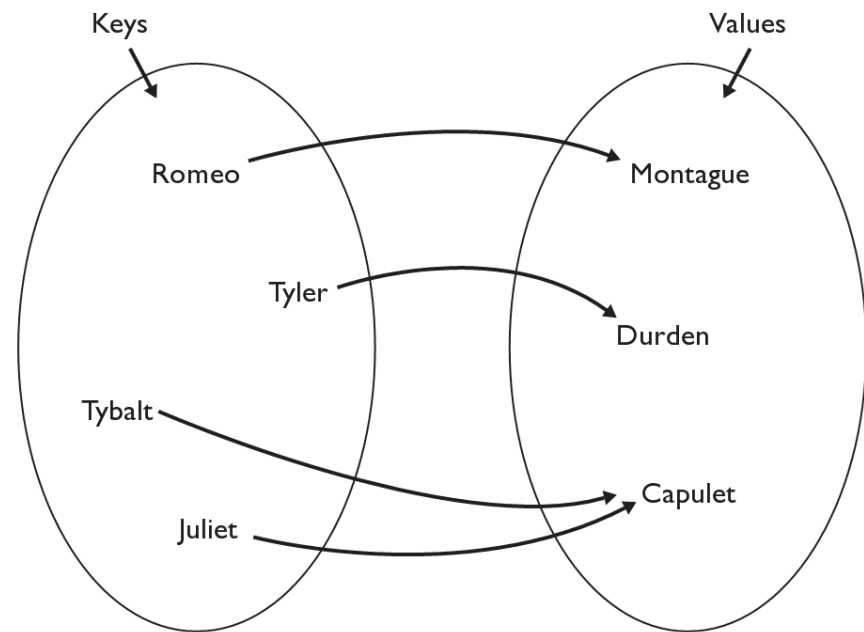
# La clase HashMap

- Ejemplos
  - dada una palabra obtener su traducción
  - dado un nombre de persona obtener su teléfono
  - dado un código postal obtener el barrio asociado
  - dada una letra obtener la lista de palabras que empiezan por ella
- los map son muy útiles y eficientes para búsquedas en una dirección
  - dada una clave obtener su valor asociado
- dado un valor obtener la clave
  - aunque puede hacerse es menos eficiente

# La clase HashMap

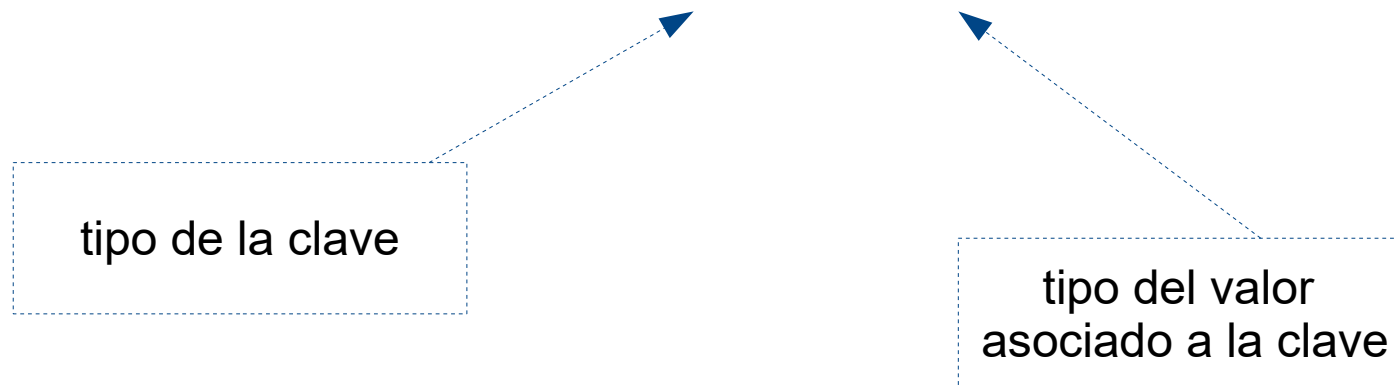
- Las claves en un **HashMap** no están ordenadas (no es previsible el orden en que se obtendrán)
  - si queremos recuperar claves en orden utilizar **TreeMap**
  - si queremos recuperar claves en el orden en que se introdujeron utilizar **LinkedHashMap**

- Las claves en un map no se duplican
- Los valores en un map pueden duplicarse
- Dos claves pueden tener asociado el mismo valor



# Definir e instanciar un HashMap

- map que asocia nombres de personas con su teléfono
  - `HashMap<String, Telefono> listin;`
  - `listin = new HashMap<String, Telefono>();`



- `HashMap<String, Telefono> listin =  
new HashMap<String, Telefono>();`



# Definir e instanciar un HashMap (otro ejemplo)

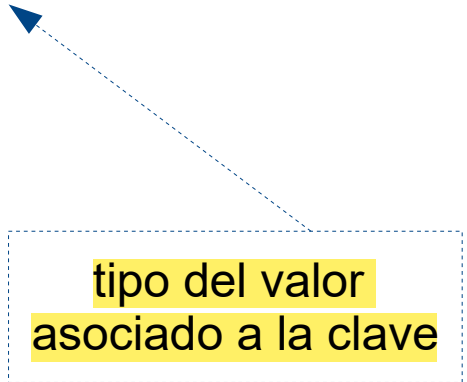
- map que asocia palabras y su traducción en inglés
  - `HashMap<String, String> diccionario;`
  - `diccionario = new HashMap<String, String>();`

tipo de la clave



A dashed blue box containing the text 'tipo de la clave' has a dashed blue arrow pointing from its top-right corner to the first '<String>' in the code 'HashMap<String, String> diccionario;'.

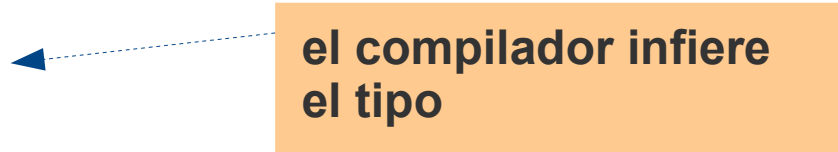
tipo del valor  
asociado a la clave



A dashed blue box containing the text 'tipo del valor asociado a la clave' has a dashed blue arrow pointing from its top-left corner to the second '<String>' in the code 'HashMap<String, String> diccionario;'.

- `diccionario = new HashMap<>();`

el compilador infiere  
el tipo



An orange box containing the text 'el compilador infiere el tipo' has a dashed blue arrow pointing from its left side to the '<>' in the code 'diccionario = new HashMap<>();'.

## Ejer 6.9

- `HashMap<Integer, Estudiante> estudiantes =  
new HashMap<Integer, Estudiante>();`

ó

- `HashMap<Integer, Estudiante> estudiantes =  
new HashMap<>();`

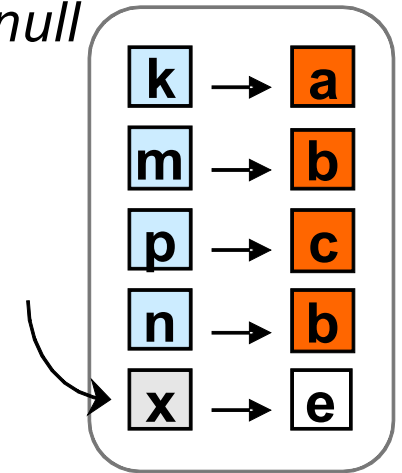
# Añadir una nueva entrada al map – put(K, V)

V put(K, V)

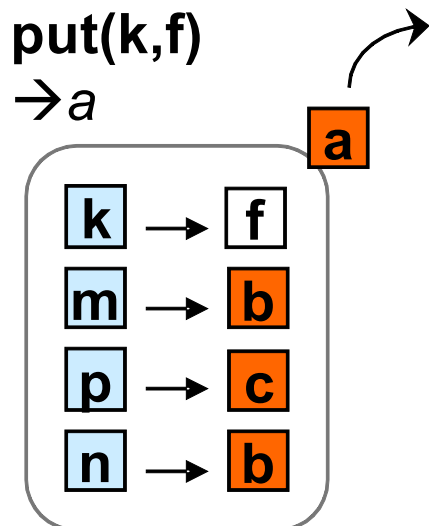
## ■ put(K, V)

- añade una nueva entrada al map, el objeto clave K asociado al objeto valor V
  - si la clave existía se sobrescribe
  - devuelve el valor previo (o null) asociado a esa clave
- 
- `diccionario.put("pan", "bread");`
  - `listin.put("Luis", new Telefono(23452425));`

put(x,e)  
→ null



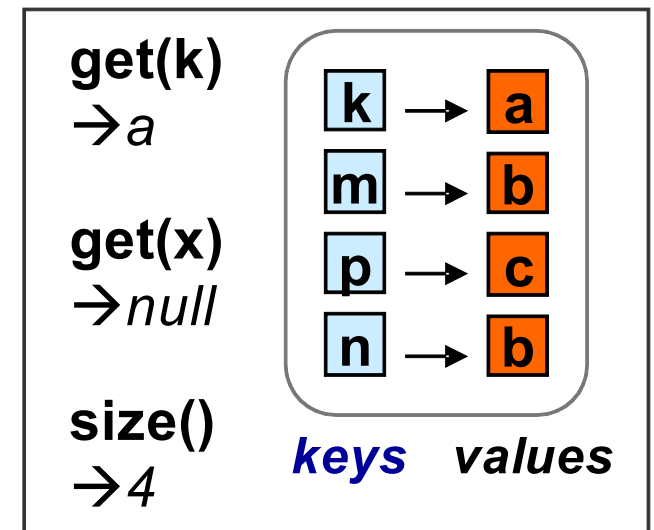
put(k,f)  
→ a



# Obtener el valor asociado a una clave – get(K)

V get(Object)

- **get(K)**
  - dada una clave K devuelve su valor asociado V
  - devuelve null si la clave no existe
  - la clave proporcionada puede ser de cualquier tipo
- String pal = diccionario.get("pan");
- Telefono telefono = listin.get("Luis");



# Tamaño del map - size() / borrar entrada del map - remove()

## ■ size()

int size()

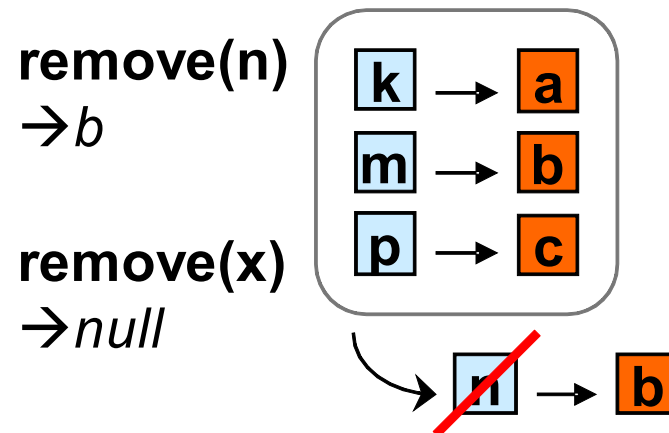
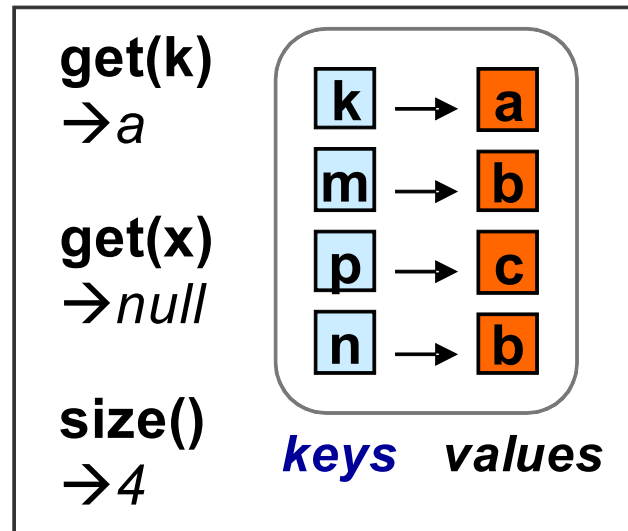
- devuelve el nº de entradas del map

## ■ remove(K)

V remove(Object)

- borra el objeto valor asociado a la clave K
    - la entrada entera desaparece
    - subyace la utilización de `equals()` y `hashCode()`
  - devuelve el valor previo (o null) asociado a esa clave
  - la clave proporcionada puede ser de cualquier tipo
  - si la clave no existe devuelve *null*
- 
- `diccionario.remove("pan");`
  - `listin.remove("Luis");`

# Tamaño del map - size() / borrar entrada del map - remove()



# Otras operaciones sobre un HashMap

**public boolean containsKey(K)**

devuelve *true* si el map contiene la clave indicada. Subyace equals().  
La clave puede ser de cualquier tipo referencia

**public boolean containsValue(V)**

devuelve *true* si el map contiene el valor indicado. Subyace equals().  
El valor puede ser de cualquier tipo referencia

**public Set<K> keySet()**

devuelve el conjunto (Set) de todas las claves del map. Es una “*vista*” de las claves del map, no es un conjunto independiente

**public Collection<V> values()**

devuelve una colección (Collection) con todos los valores del map.  
Es una “*vista*” de los valores del map

**public Set<Map.Entry<K,V>> entrySet()**

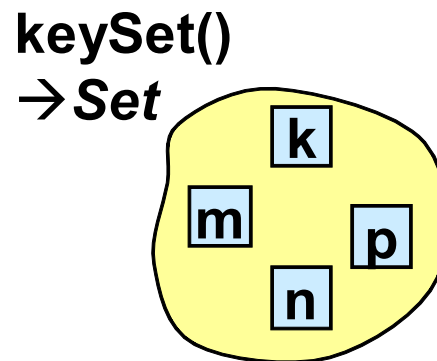
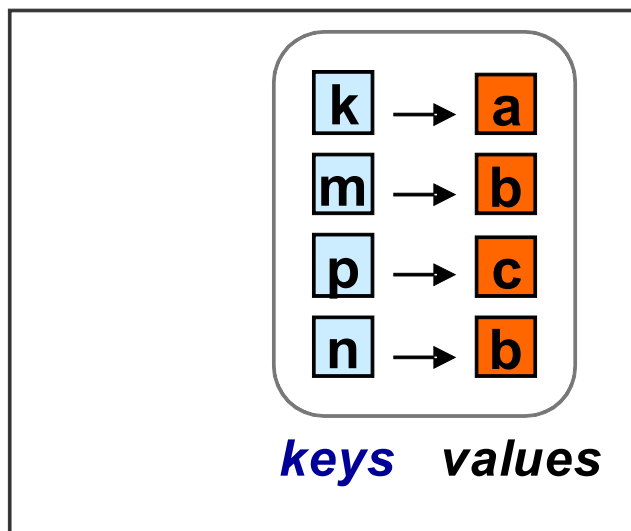
devuelve el conjunto (Set) de todas las entradas (de tipo Map.Entry) del map. Es una “*vista*” sobre las entradas del map, no es un conjunto independiente

# Obtener el conjunto de claves de un HashMap - `keySet()`

```
Set<K> keySet( )
```

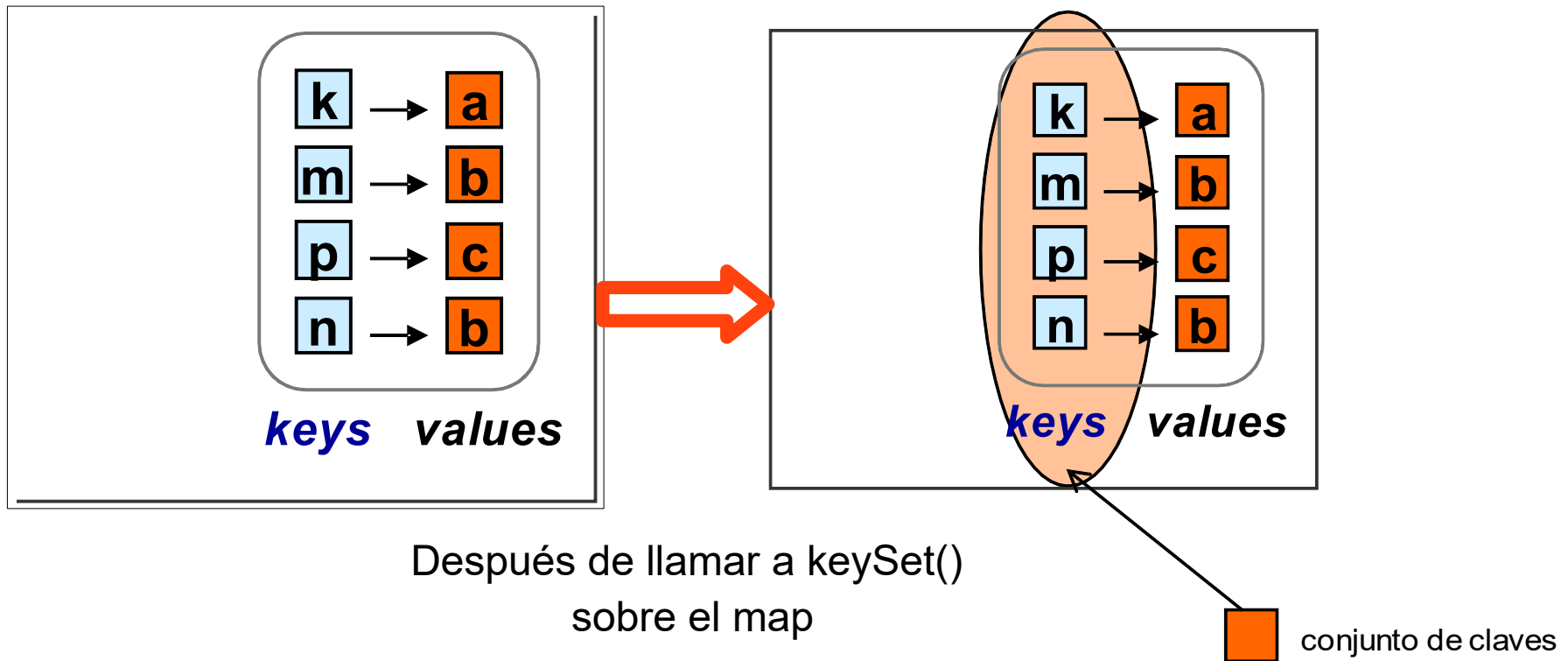
## ■ `keySet()`

- devuelve el conjunto de claves del map
- no es un conjunto independiente
  - una vista





# Obtener el conjunto de claves de un HashMap - `keySet()`



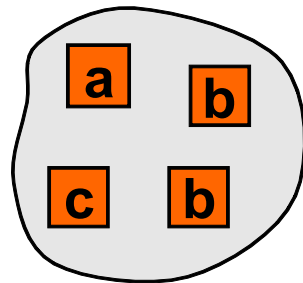
# Obtener el conjunto de valores de un HashMap - values()

`Collection<V> values( )`

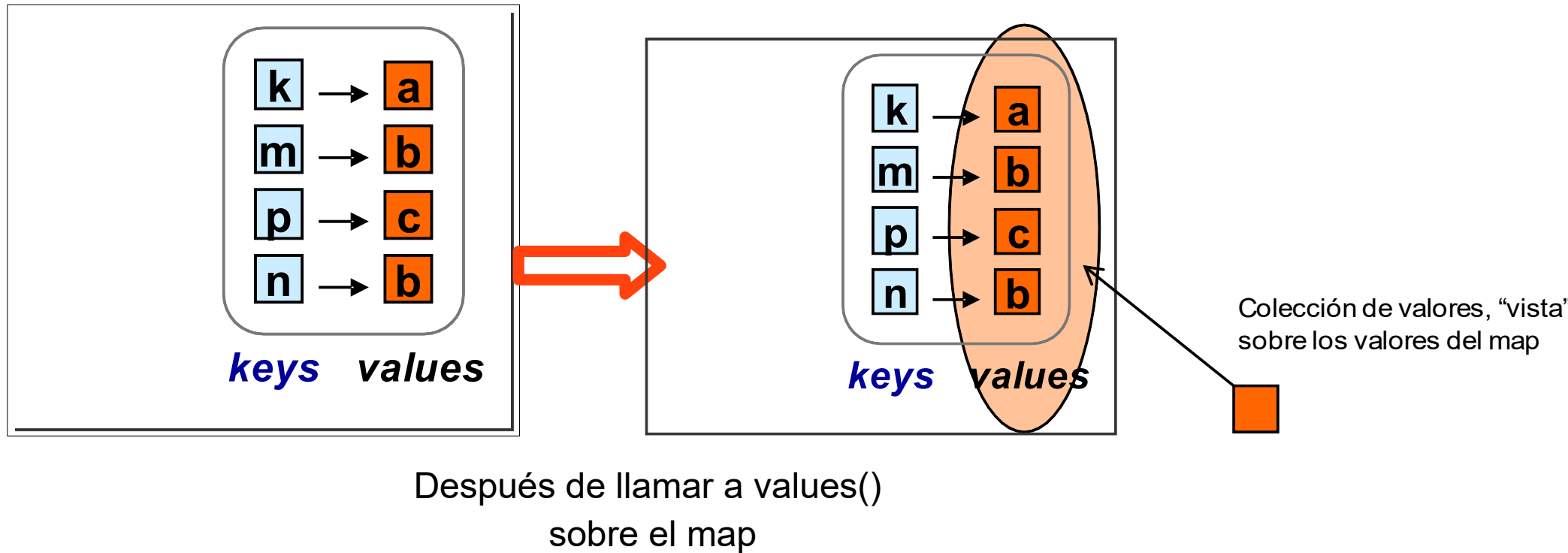
## ■ values()

- devuelve el conjunto de valores del map
  - una colección `Collection` no independiente
- es una vista sobre los valores del map

**values()**  
→ ***Collection***

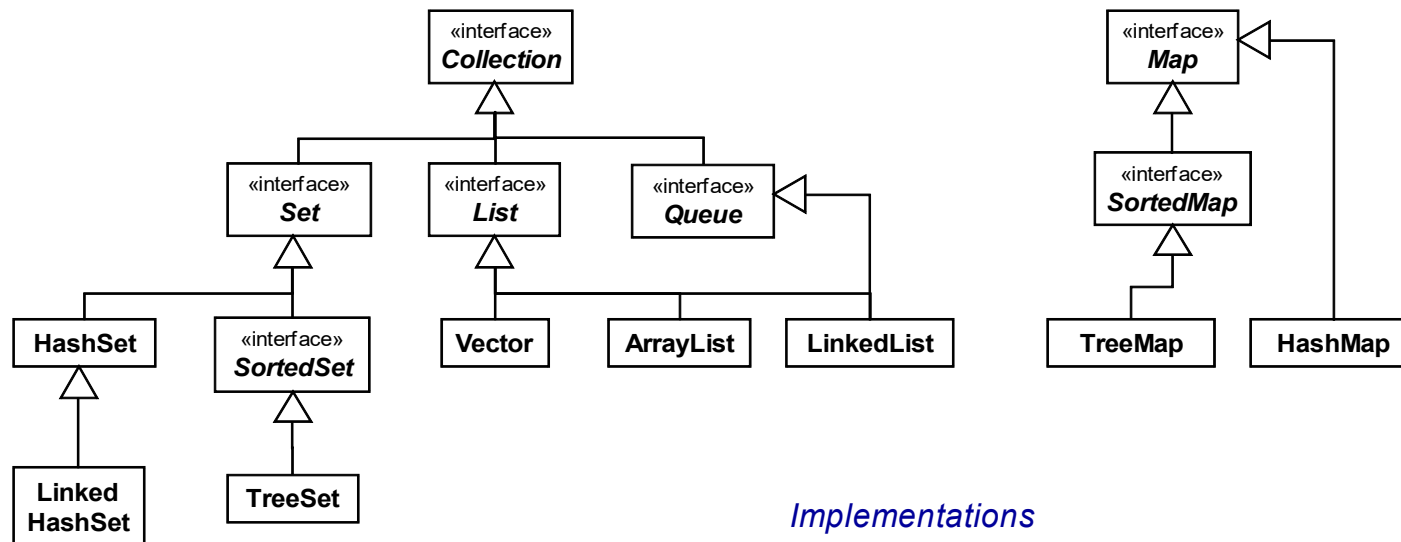


# Obtener el conjunto de valores de un HashMap - values()



# Cómo iterar sobre un HashMap?

- Un map no es en realidad una colección (no de tipo `Collection`)
  - en particular no hay iteradores para los map (no `iterator()`)



# Cómo iterar sobre un HashMap?

- Para recorrer un map de forma completa

**A**

- obtener el conjunto de claves con `keySet()`
- utilizar un for mejorado o un iterador para recorrer ese conjunto de claves y escribir el map

**B**

- obtener el conjunto de entradas con `entrySet()`
- utilizar un for mejorado o un iterador para recorrer ese conjunto de entradas y escribir el map

# Iterar sobre un HashMap utilizando keySet()

```
HashMap<String, String> diccionario = new HashMap<>();
```

```
.....
```

```
Set<String> conjuntoClaves = diccionario.keySet();
```

```
Iterator<String> it = conjuntoClaves.iterator();
```

```
while (it.hasNext()) {  
    String clave = it.next();  
    String traduccion = diccionario.get(clave);  
    System.out.println(clave + " - " + traduccion);  
}
```

# Iterar sobre un HashMap utilizando keySet()

```
HashMap<String, String> diccionario = new HashMap<>();
```

```
.....
```

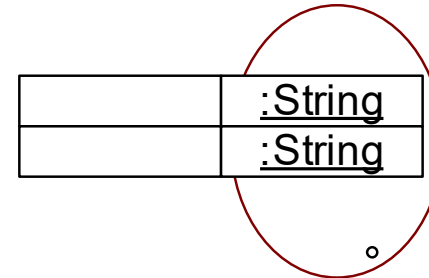
```
Set<String> conjuntoClaves = diccionario.keySet();
```

```
for (String clave: conjuntoClaves) {  
    String traduccion = diccionario.get(clave);  
    System.out.println(clave + " - " + traduccion);  
}
```

# Iterar sobre un HashMap utilizando entrySet()

## ■ **Map.Entry (interface)**

- cada elemento en un map es un par clave / valor
- un objeto de tipo **Map.Entry**
- el conjunto de entradas del map se obtiene con el método **entrySet()**
  - devuelve una vista sobre las entradas
- podemos iterar sobre un map recorriendo este conjunto de entradas



Entrada de tipo  
Map.Entry

- Si cada entrada del map es de tipo Map.Entry entonces
  - **entrada.getKey()** devuelve la clave de la entrada
  - **entrada.getValue()** devuelve el valor de la entrada



# Iterar sobre un HashMap utilizando entrySet()

```
HashMap<String, String> diccionario = new HashMap<>();
```

```
.....
```

```
Set<Map.Entry<String, String>> conjuntoEntradas =  
    diccionario.entrySet();
```

```
Iterator<Map.Entry<String, String>>> it =  
    conjuntoEntradas.iterator();
```

```
while (it.hasNext()) {  
    Map.Entry<String, String> entrada = it.next();  
    System.out.println(entrada.getKey() + " - " +  
        entrada.getValue());  
}
```

# Iterar sobre un HashMap utilizando entrySet()

```
HashMap<String, String> diccionario = new HashMap<>();
```

```
.....
```

```
Set<Map.Entry<String, String>> conjuntoEntradas =  
    diccionario.entrySet();
```

```
for (Map.Entry<String, String> entrada :conjuntoEntradas) {  
    System.out.println(entrada.getKey() + " - " +  
        entrada.getValue());  
}
```

## Ejer 6.10

```
HashMap<String, Cuenta> cuentasBancarias =  
    new HashMap<>();
```

.....

**A**

```
public void addCuenta(String nombre, int numCuenta, int balance)  
{  
    cuentasBancarias.put(nombre, new Cuenta(numCuenta, balance));  
}
```

## Ejer 6.10

**B**

```
public void addCuenta(String nombre, int numCuenta, int balance)
{
    cuentasBancarias.put(nombre, new Cuenta(numCuenta, balance));
}
```

## Ejer 6.10

C

```
public void listarClientes( )
{
    Set<String> conjuntoClaves = cuentasBancarias.keySet();
    for (String clave: conjuntoClaves) {
        System.out.println("Id. cliente " + clave);
    }
}
```

## Ejer 6.11

```
import java.util.HashMap;
public class ListinTelefonico
{
    .....

    public String buscarNumero(String nombre)
    {
        listin.get(nombre);
    }

    public void escribirListin()
    {
        for (String numero: listin.keySet()) {
            System.out.println(nombre + " - " + numero);
        }
    }
}
```

## Ejer 6.11

```
import java.util.HashMap;
public class ListinTelefonico
{
    private HashMap<String, String> listin;
    public ListinTelefonico( )
    {
        listin = new HashMap<>();
    }

    public void listarClientes( )
    {
        Set<String> conjuntoClaves = cuentasBancarias.keySet();
        for (String clave: conjuntoClaves) {
            System.out.println("Id. cliente " + clave);
        }
    }
}
```

## Ejer 6.12

- **A**
  - se sobrescribe la entrada
- **B**
  - se añade la entrada con la nueva clave y el valor estará duplicado
- **C**
  - con el método `containsKey()`
- **D**
  - método `size()`



## Ejer 6.13

- `HashMap<String, String> m = new HashMap<>();`  
`m.add("hola");`
  - NO CORRECTO, se añade con el método `put(K, V)`
- `HashMap<String, String> m = new HashMap<>();`  
`ArrayList<String> lista = new ArrayList<>();`  
`m.put("hola", lista);`
  - NO CORRECTO, el valor asociado en el map es de tipo `String` y se está añadiendo un valor de tipo `ArrayList<String>`

## Ejer 6.13

- `HashMap<String, String> m = new HashMap<>();`  
`String s = "27";`  
`m.put("hola", s);`  
`m.put("adios", s);`
  - CORRECTO, se añaden dos claves diferentes con el mismo valor
- `HashMap<String, String> m = new HashMap<>();`  
`m.put("hola", "27");`  
`m.put("hola", "327");`
  - CORRECTO, se sobrescribe la primera entrada

# Otro ejemplo – Personas y sus colores favoritos

- Clase `ElectorColor`
  - incluye atributo que asocia nombres de personas con sus colores favoritos
  - los colores de cada persona no se repiten
  - no nos importa el orden de los colores
- `private HashMap<String, HashSet<String>> mapPersonas;`

## Otro ejemplo – Personas y sus colores favoritos

```
import java.util.HashMap;
public class ElectorColor
{
    private HashMap<String, HashSet<String>> mapPersonas;

    public ElectorColor( )
    {
        mapPersonas = new HashMap<String, HashSet<String>>();
    }
}
```

# Otro ejemplo – Personas y sus colores favoritos

```
import java.util.HashMap;  
public class ElectorColor  
{
```

```
.....
```

```
public void addPersona(String nombre, String color )  
{  
    if (!mapPersonas.containsKey(nombre)) {  
        HashSet<String> colores = new HashSet<String>();  
        colores.add(color);  
        mapPersonas.put(nombre, colores);  
    }  
    else {  
        mapPersonas.get(nombre).add(color);  
    }  
}
```

# Otro ejemplo – Personas y sus colores favoritos

```
import java.util.HashMap;
public class ElectorColor
{
    .....

    public void addPersonaV2(String nombre, String color )
    {
        if (!mapPersonas.containsKey(nombre)) {
            HashSet<String> colores = new HashSet<String>();
            mapPersonas.put(nombre, colores);
        }
        mapPersonas.get(nombre).add(color);
    }
}
```

# Otro ejemplo – Personas y sus colores favoritos

```
import java.util.HashMap;
public class ElectorColor
{
    .....

    public int personasLesGustaColor(String color)
    {
        int cuantos = 0;
        for (String nombre: mapPersonas.keySet()) {
            if (mapPersonas.get(nombre).contains(color)) {
                cuantos++;
            }
        }
        return cuantos;
    }
}
```

# Ejercicios adicionales

---

EJAD05 Generador conjunto

EJAD06 Serie múltiplos

EJAD07 Frecuencia palabras

EJAD08 Tesauro

EJAD09 Equipo Jugador



# Documentación de las clases

---

- Comentarios *javadoc*
- etiquetas @
- Qué documentar?
  - elementos públicos de la clase
- *javadoc* y BlueJ
- *javadoc* desde línea de comandos

# Paquetes y ficheros jar

---

Presentación aparte

# Más sobre colecciones

- Java Collections Framework
  - conjunto de interfaces y clases (abstractas y concretas) que permiten organizar y manipular los datos eficientemente
  - **interfaces** (tipos)
    - describen tipos lógicos (representan funcionalidad, son las abstracciones de la implementación)
  - **clases** (implementaciones)
    - implementan el tipo lógico (incluyen además en algunos casos métodos estáticos – *algoritmos* – de utilidad para trabajar con diferentes tipos de colecciones).

# Más sobre colecciones

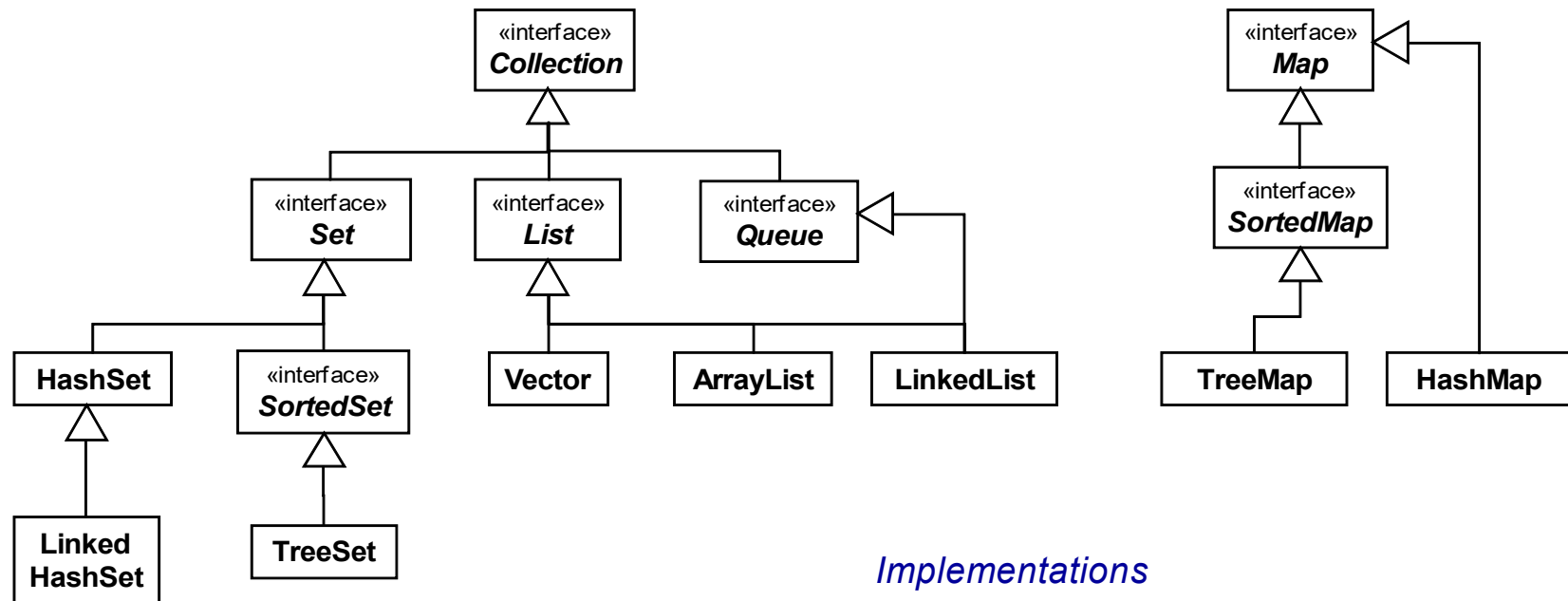
---

- Ventajas de un framework
  - reducen el esfuerzo de programación
  - incrementan la velocidad de desarrollo y la calidad
  - hay que hacer menos esfuerzo para utilizar colecciones
  - se reutiliza el código

# Java Collection Framework

- Java soporta tres tipos de colecciones
  - List, Set y Map
- Definidas por tres interfaces:
  - **Collection** – interfaz raíz que representa a una colección de elementos
    - **Set** – interfaz que describe una colección de elementos no duplicados y sin orden (HashSet, TreeSet, LinkedHashSet)
    - **List** – interfaz que representa a una colección de elementos ordenados que permite duplicados y el acceso por índice (ArrayList, LinkedList)
  - **Map** – describe una colección de objetos en la que se asocia una clave con un objeto. No contiene claves duplicadas (HashMap, TreeMap, LinkedHashMap).

# Java Collection Framework



# Implementación de las colecciones

- Estructuras subyacentes utilizadas en las clases que representan colecciones
  - arrays redimensionables
  - listas enlazadas (linked lists)
  - árboles (tree)
  - tablas asociativas (hash table)

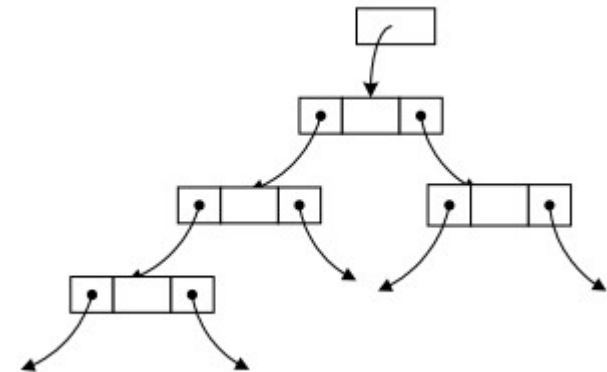
| Implementaciones |           |           |         |            |
|------------------|-----------|-----------|---------|------------|
| Interfaces       | hashtable | array     | tree    | linkedlist |
| <b>List</b>      |           | ArrayList |         | LinkedList |
| <b>Set</b>       | HashSet   |           | TreeSet |            |
| <b>Map</b>       | HashMap   |           | TreeMap |            |

# Implementación de las colecciones

- *array redimensionable* – array que “crece”
- *lista enlazada* – estructura lineal formada por nodos dónde cada nodo “apunta” o “enlaza” con el siguiente



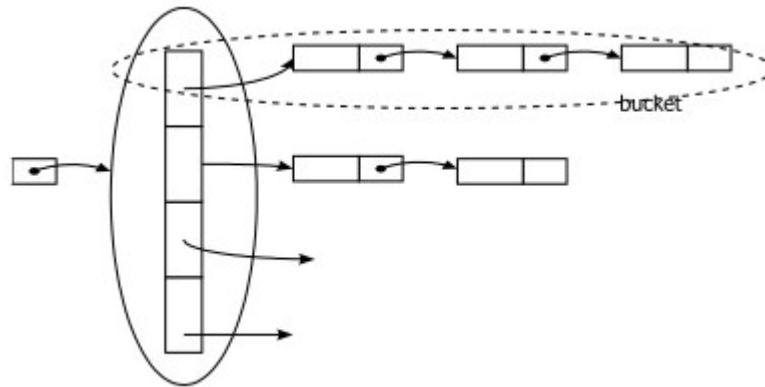
- *árbol (tree)* - estructura no lineal de nodos enlazados en forma de árbol (generalmente binario)





# Implementación de las colecciones

- *hashtable* – combinación de array y lista enlazada



# Otras colecciones

- Clase **Collections**
  - no representa a ninguna colección, simplemente contiene utilidades (métodos estáticos) para trabajar con colecciones de diferentes tipos
  - `max()`, `reverse()`, `shuffle()`, `sort()`
- Clase **Vector**
  - misma funcionalidad que `ArrayList`
  - es sincronizada lo que la hace menos eficiente para trabajar con hilos

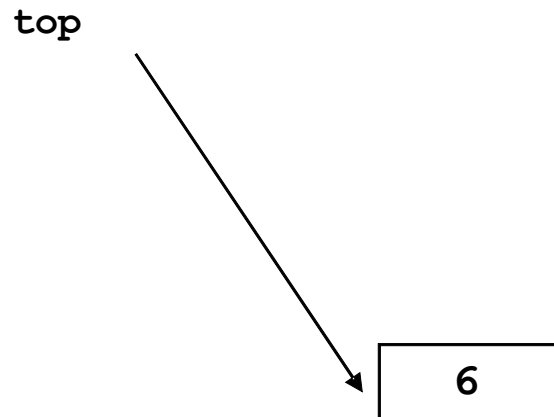
# Otras colecciones: Stack

- Clase **Stack**
  - define una colección de elementos que se gestionan como una “pila” (estructura **LIFO – Last In First Out**)
  - definida como una extensión de la clase Vector
  - en java.util

```
boolean isEmpty ()  
E push (E obj)  
E pop ()  
E peek ()
```

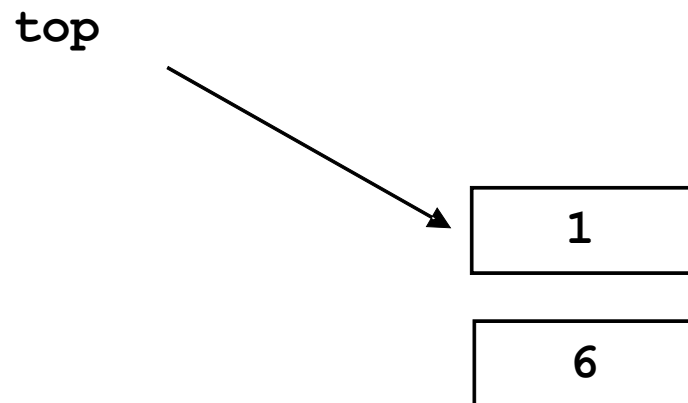
# Ejemplo Stack

```
Stack<Integer> st = new Stack<>();  
st.push(6);
```



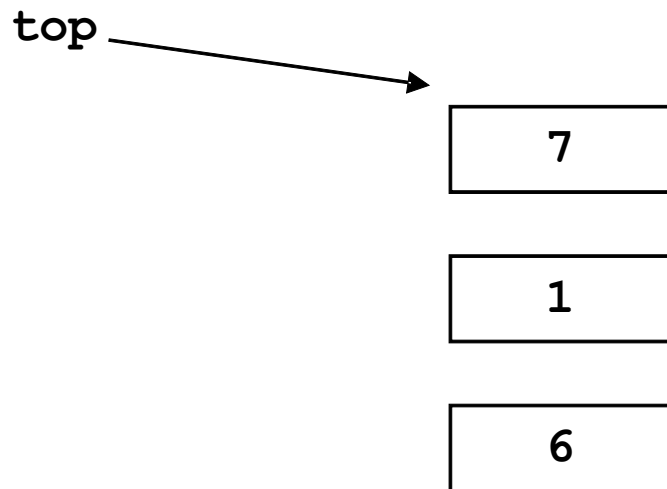
# Ejemplo Stack

```
Stack<Integer> st = new Stack<>();  
st.push(6);  
st.push(1);
```



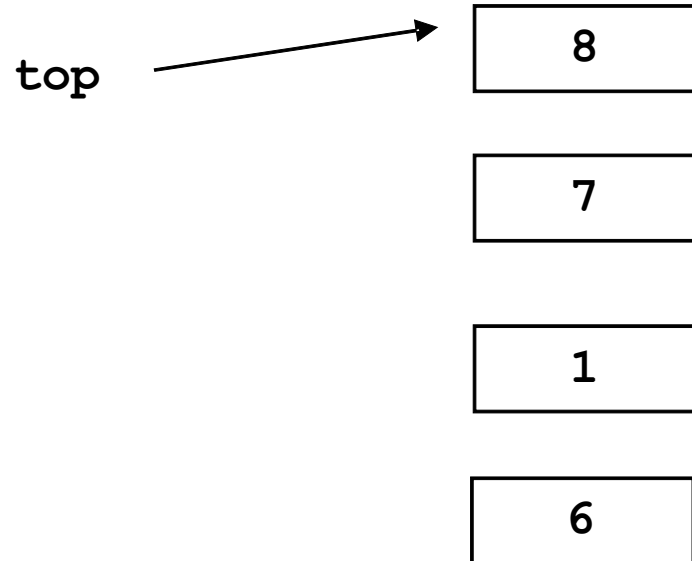
# Ejemplo Stack

```
Stack<Integer> st = new Stack<>();  
st.push(6);  
st.push(1);  
st.push(7);
```



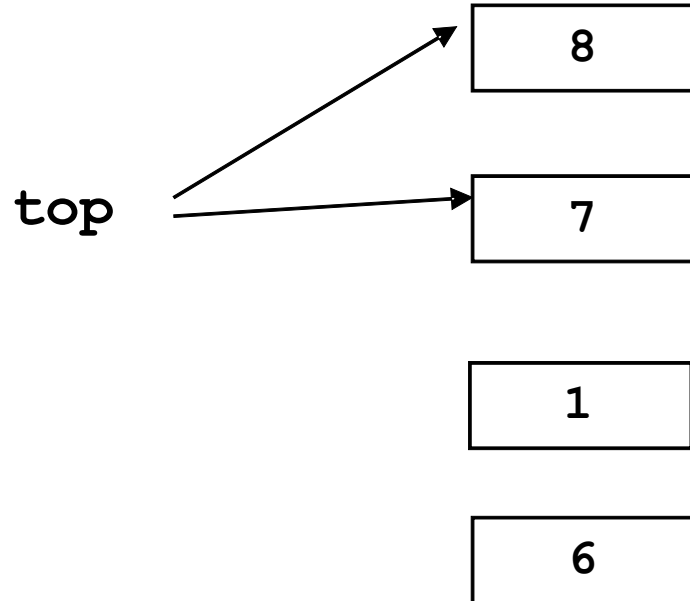
# Ejemplo Stack

```
Stack<Integer> st = new Stack<>();  
st.push(6);  
st.push(1);  
st.push(7);  
st.push(8);
```



# Ejemplo Stack

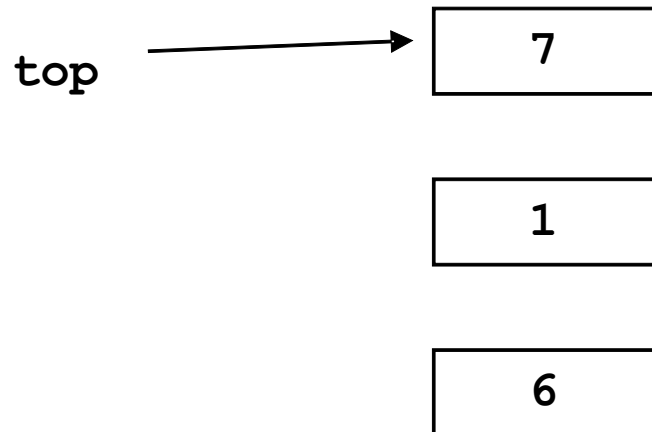
```
Stack<Integer> st = new Stack<>();  
st.push(6);  
st.push(1);  
st.push(7);  
st.push(8);  
st.pop();
```





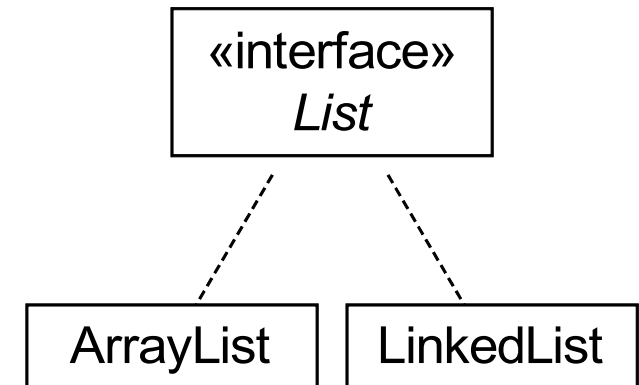
# Ejemplo Stack

```
Stack<Integer> st = new Stack<>();  
st.push(6);  
st.push(1);  
st.push(7);  
st.push(8);  
st.pop();
```



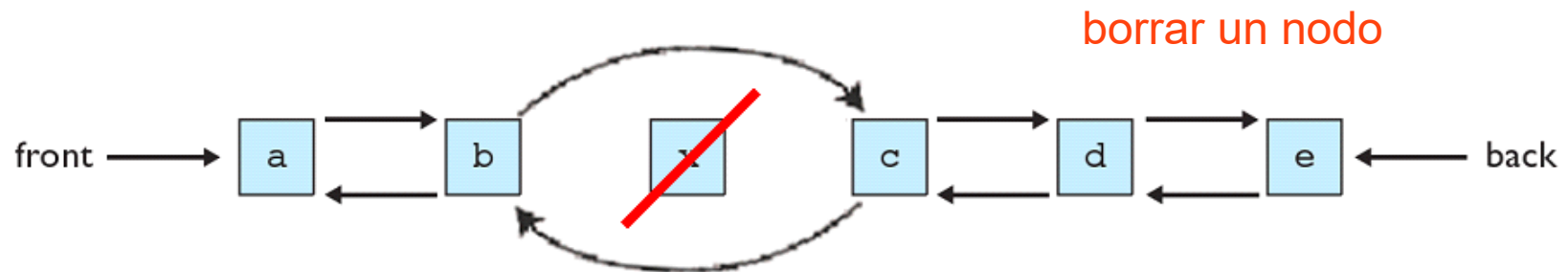
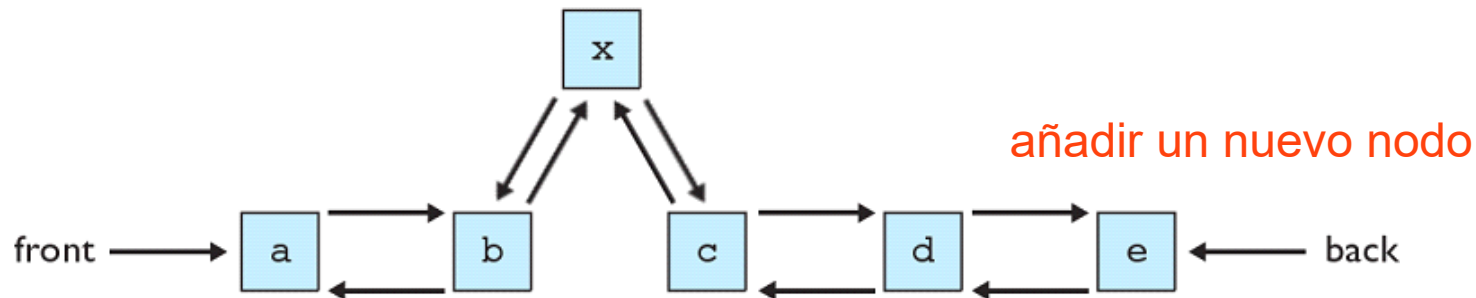
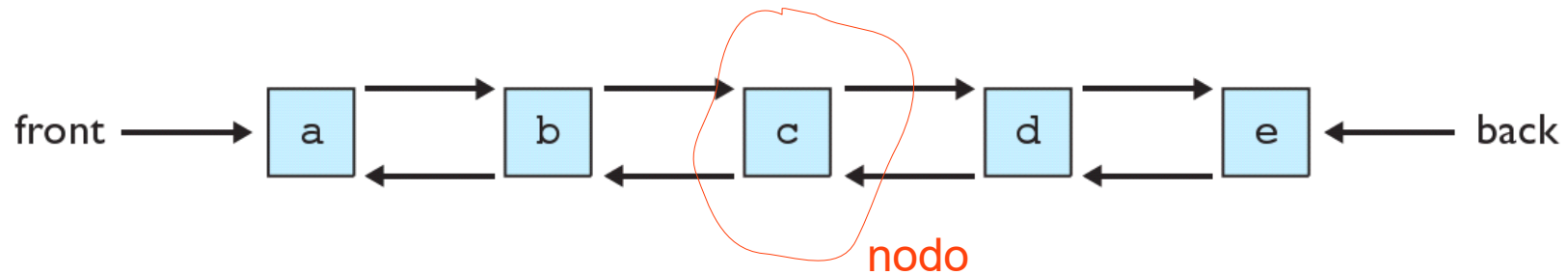
# Otras colecciones: LinkedList

- Clase **LinkedList**
  - misma funcionalidad que ArrayList
  - se implementa como una estructura de datos dinámica, la lista enlazada
  - permite que las inserciones y los borrados sean muy rápidos
- añade algunos métodos propios



```
void addFirst (E obj)
void addLast (E obj)
E getFirst ()
E getLast ()
E removeFirst ()
E removeLast ()
```

# Otras colecciones: LinkedList



# ¿Qué colección elegir?

- Cada colección tiene sus ventajas y sus desventajas
  - ArrayList permite accesos más rápidos que LinkedList pero
  - inserciones y borrados son más lentos
- Hay que
  - pensar en el problema a resolver y elegir el interface (funcionalidad) más adecuado
  - decidir si lo que importan son los accesos, o hay muchas operaciones de inserción y borrado, ...
  - elegir la clase (implementación) que implemente el interfaz

## Ejer 6.20

```
import java.util.Stack;
import java.util.Arrays;

public class DetectorPalindromos
{

    private String cadena;

    /**
     * Constructor de la clase DetectorPalindromos
     */
    public DetectorPalindromos(String cadena)
    {
        this.cadena = cadena.trim().toLowerCase();
        eliminarEspacios();
    }
}
```

## Ejer 6.20

```
/**
 * elimina los blancos que haya en la cadena
 */
private void eliminarEspacios()
{
    cadena = cadena.replace(" ", "");
}

/**
 * @return true si es palindromo
 */
public boolean esPalindromo()
{
    Stack<Character> pila = new Stack<>();
    empilar(cadena, pila);
    String inversa = desempilar(pila);
    return inversa.equals(cadena);
}
```

## Ejer 6.20

```
/**
 *
 * @param cadena la cadena a empilar
 * @param pila la pila sobre la que se empila
 */
private void empilar(String cadena, Stack<Character> pila)
{
    for (int i = 0; i < cadena.length(); i++) {
        pila.push(cadena.charAt(i));
    }
}
```