

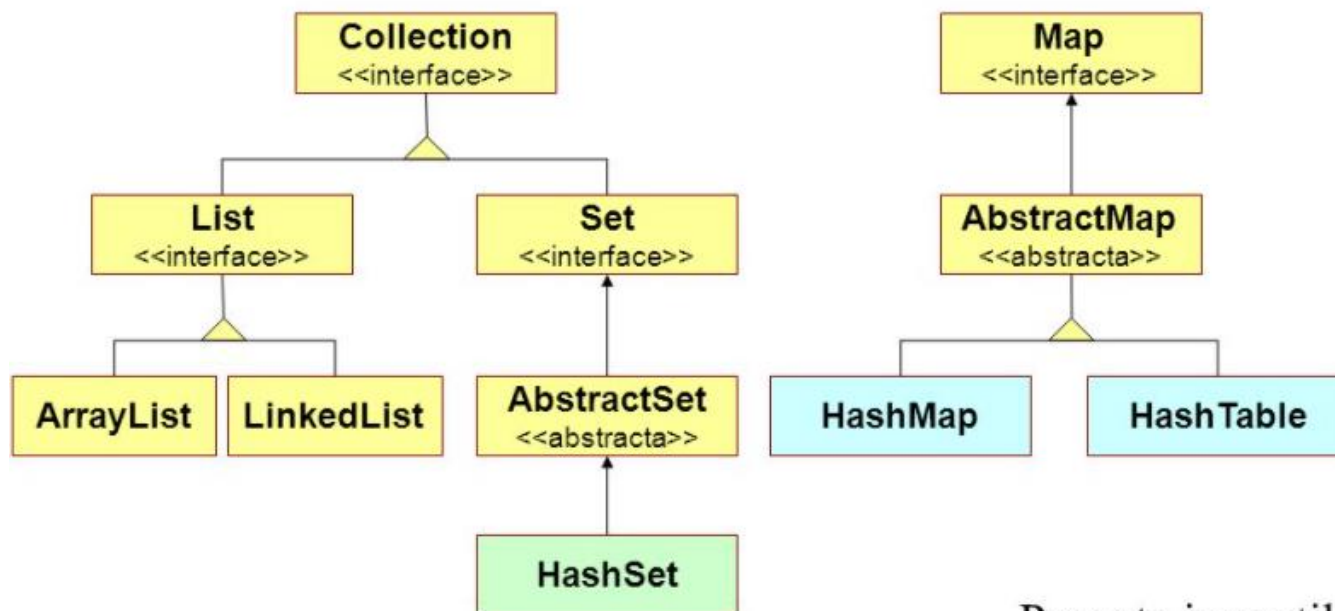
DAM PROGRAMACIÓN



UD6-ESTRUCTURAS DE ALMACENAMIENTO

Colecciones

- Una **colección** es un **grupo de objetos almacenados de forma conjunta en una misma estructura**.
- **Las colecciones** almacenan de grupos de **objetos** que **mantienen alguna relación** y facilitan operaciones de búsqueda, ordenación, etc.
- En algunos casos es necesario que los objetos almacenados en las colecciones **implementen determinadas interfaces** para poder realizar ciertas operaciones.



Paquete java.util

Interfaz Collection

- La **interfaz** presente en muchas de las colecciones en Java es **Collection**, gracias a la cual podemos almacenar cualquier objeto y realizar operaciones como añadir, eliminar, obtener el tamaño de la colección, etc.
- Entre los **métodos más destacados** de la interfaz **Collection** están:
 - **int size()** → devuelve el nº de elementos de la colección.
 - **boolean isEmpty()** → devuelve true si la colección está vacía.
 - **boolean contains (E element)** → devuelve true si el elemento está almacenado en la colección.
 - **boolean add (E element)** → permite añadir el elemento a la colección.
 - **boolean remove (E element)** → elimina el elemento de la colección.
 - **Object[] toArray** → permite pasar la colección a un array de objetos de tipo Object.
 - **void clear()** → vacía la colección.

Tipos de colecciones

Set

HashSet

TreeSet

LinkedHashSet

List

ArrayList

LinkedList

Map

HashMap

TreeMap

LinkedHashMap

Creación de objetos de las colecciones

```
Set<E> almacen = new HashSet<E>();  
Set<E> almacen = new LinkedHashSet<E>();  
Set<E> almacen = new TreeSet<E>();
```

Creación de objetos que implementan **Set**
No permite duplicados

```
List<E> almacen = new ArrayList<E>();  
List<E> almacen = new LinkedList<E>();
```

Creación de objetos que implementan **List**.
Cada elemento está identificado por su posición

```
Map<K,V> almacen = new HashMap<K,V>();  
Map<K,V> almacen = new TreeMap<K,V>();  
Map<K,V> almacen = new LinkedHashMap<K,V>();
```

Creación de objetos que implementan **Map**.
Cada elemento está asociado a una clave y un valor

Todas las listas pueden no especificar el objeto (<E>), en tal caso, el objeto que almacenan es de tipo **Object**.



Set

- La **interfaz Set** define una colección que **no puede contener elementos duplicados**. Se encarga de gestionar conjuntos.
- Esta interfaz contiene, únicamente, los métodos heredados de **Collection** añadiendo la restricción del almacenamiento de elementos duplicados.
- Los **objetos almacenados** con **Set** deben tener implementados los métodos **equals()** y **hashCode()** ya que estos métodos serán utilizados para comprobar si dos elementos son iguales o no.
- Las clases que implementan la interfaz **Set** son:
 - **HashSet**:
 - Almacena los elementos en una tabla **hash**.
 - No garantiza ningún orden a la hora de realizar iteraciones.
 - Se debe definir el tamaño inicial.
 - Es la implementación de **Set** con mejor rendimiento.
 - **TreeSet**:
 - Almacena los elementos ordenándolos en función de sus valores.
 - Los elementos almacenados deben implementar la interfaz Comparable.
 - Es la implementación con un rendimiento más lento
 - **LinkedHashSet**:
 - Almacena los elementos en función del orden de inserción.



```
public class Persona {
```

```
    private String nombre;  
    private String nif;
```

```
    public Persona (String nombre, String nif){  
        this.nombre=nombre;  
    }
```

```
@Override
```

```
public int hashCode(){  
    final int prime = 31;  
    int result = 1;  
    result = result * prime * ((nombre == null)?0:nombre.hashCode());  
    result = result * prime * ((nif == null)?0:nif.hashCode());  
    return result;  
}
```

```
@Override
```

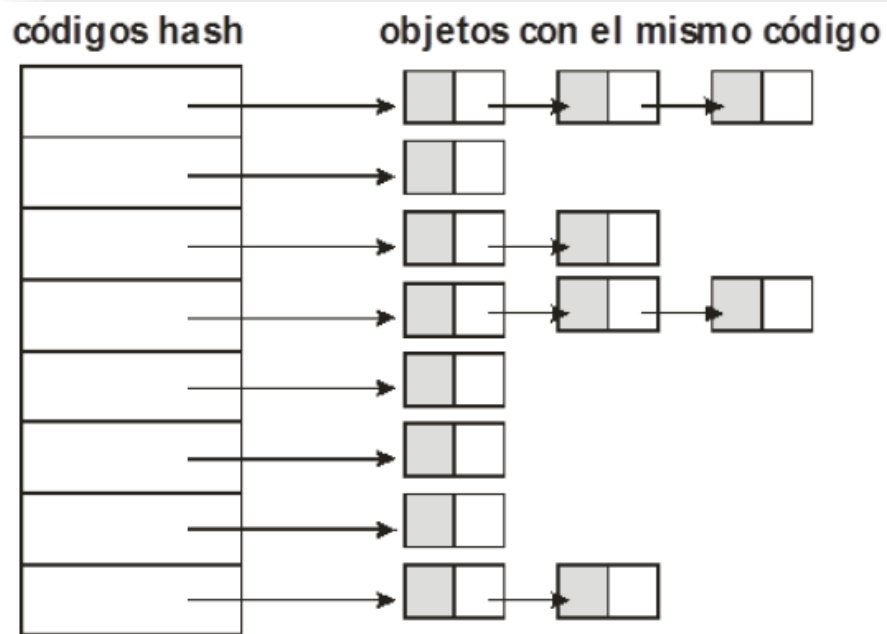
```
public boolean equals (Object obj){  
    if (this == obj) return true;  
    if (Objects.isNull(obj)) return false;  
    if (this.getClass() != obj.getClass()) return false;  
    Persona other = (Persona) obj;  
    if (nif != other.nif) return false;  
    if (nombre != other.nombre) return false;  
    return true;  
}
```

```
}
```

Para que los objetos de una clase puedan ser almacenados en un **Set** deben tener implementados la función **hashCode()** y **equals()**

Set → HashSet

Una **tabla hash** es una estructura de datos formada básicamente por un array donde **la posición de los datos va determinada por una función hash**, permitiendo localizar la información por medio del resultado de dicha función

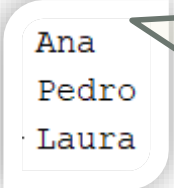


Todos los objetos almacenados en un HashSet **deben tener implementado el método hashCode()** para realizar el almacenamiento en una tabla hash


```
public static void main(String[] args) {  
    Persona p1 = new Persona("Laura", "3453435");  
    Persona p2 = new Persona("Ana", "3434375");  
    Persona p3 = new Persona("Pedro", "3458635");
```

```
    HashSet<Persona> almacen = new HashSet<Persona>();  
    almacen.add(p1);  
    almacen.add(p2);  
    almacen.add(p3);  
    //Este objeto no se almacena  
    almacen.add(p3);
```

```
    for (Persona aux : almacen) {  
        System.out.println(aux.getNombre());  
    }  
}
```



Ana
Pedro
Laura

El orden viene
dado por el
hash de cada
objeto, no por el
orden de
inserción

Set → LinkedHashSet

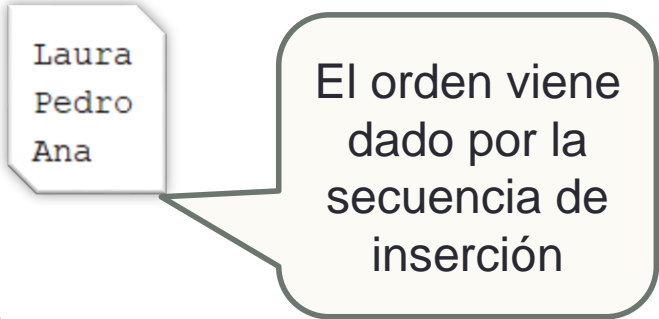
```
public static void main(String[] args) {
```

```
    Persona p1 = new Persona("Laura", "3453435");  
    Persona p2 = new Persona("Pedro", "3458635");  
    Persona p3 = new Persona("Ana", "3434375");
```

```
    LinkedHashSet<Persona> almacen = new LinkedHashSet<Persona>();
```

```
    almacen.add(p1);  
    almacen.add(p2);  
    almacen.add(p3);  
    //Este objeto no se almacena  
    almacen.add(p3);
```

```
    for (Persona aux : almacen) {  
        System.out.println(aux.getNombre());  
    }  
}
```



Laura
Pedro
Ana

El orden viene
dado por la
secuencia de
inserción

Set → TreeSet

- Para utilizar **TreeSet** para almacenar objetos, éstos tienen que tener implementado la interfaz **Comparable**, por ello debemos añadir a la clase Persona una implementación para el método **compareTo(Object o)**

```
@Override  
public int compareTo(Object o) {  
    return nombre.compareTo(((Persona)o).getNombre());  
}
```

cadena1.compareTo(cadena2)

- devuelve < 0, si cadena1 < cadena2
- devuelve == 0 si cadena1 == cadena2
- devuelve > 0, si cadena1 > cadena2

Determinamos que los objetos se ordenen según su atributo nombre

```
public static void main(String[] args) {
```

```
    Persona p1 = new Persona("Laura", "3453435");  
    Persona p2 = new Persona("Pedro", "3458635");  
    Persona p3 = new Persona("Ana", "3434375");
```

```
    TreeSet<Persona> almacen = new TreeSet<Persona>();
```








```
    almacen.add(p1);  
    almacen.add(p2);  
    almacen.add(p3);  
    //Este objeto no se almacena  
    almacen.add(p3);
```

```
    for (Persona aux : almacen) {  
        System.out.println(aux.getNombre());  
    }
```

```
}
```

Ana
Laura
Pedro

Los objetos fueron almacenados según el orden alfabético de su atributo *nombre*



```
final Set hashSet = new HashSet(1_000_000);
final Long startHashSetTime = System.currentTimeMillis();
for (int i = 0; i < 1_000_000; i++) {
    hashSet.add(i);
}
final Long endHashSetTime = System.currentTimeMillis();
System.out.println("Time spent by HashSet: " +
    (endHashSetTime - startHashSetTime));
```

Algoritmo de análisis de rendimiento de las distintas colecciones de tipo Set

```
Tiempo utilizado con HashSet: 100
Tiempo utilizado con TreeSet: 211
Tiempo utilizado con LinkedHashSet: 106
```

```
final Set treeSet = new TreeSet();
final Long startTreeSetTime = System.currentTimeMillis();
for (int i = 0; i < 1_000_000; i++) {
    treeSet.add(i);
}
final Long endTreeSetTime = System.currentTimeMillis();
System.out.println("Time spent by TreeSet: " + (endTreeSetTime - startTreeSetTime));
```

En el caso de no definir ningún tipo de objeto al definir **Set**, el objeto que almacena es **Object**

```
final Set linkedHashSet = new LinkedHashSet(1_000_000);
final Long startLinkedHashSetTime = System.currentTimeMillis();
for (int i = 0; i < 1_000_000; i++) {
    linkedHashSet.add(i);
}
final Long endLinkedHashSetTime = System.currentTimeMillis();
System.out.println("Time spent by LinkedHashSet: " +
    (endLinkedHashSetTime - startLinkedHashSetTime));
```

El modificador **final** determina que no puede modificarse el dato que está a la derecha del =

List

- La interfaz **List** define una sucesión de elementos.
- A diferencia de la interfaz **Set**, la interfaz **List** **sí admite elementos duplicados**.
- **Cada elemento de la lista tiene un índice y una posición.**
- A parte de los métodos heredados de **Collection**, añade métodos que permiten mejorar los siguientes puntos:
 - **boolean add(int índice, E e)** → Añadir un objeto a la colección en una posición determinada.
 - **E get(int índice)** → Devuelve el elemento de la posición *índice*.
 - **int indexOf(E e)** → Devuelve la primera posición en el que se encuentra el elemento.
 - **int lastIndexOf(E e)** → Devuelve la última posición en el que se encuentra el elemento.
 - **E remove(int índice)** → Eliminar un objeto en la posición indicada
 - **E set(int índice, E e)** → Reemplaza el elemento en la posición indicada.

Clases que implementan List

- Las clases que implementan la interfaz **List** son:
 - **ArrayList**:
 - Es la clase más utilizada para representar colecciones de datos.
 - Su funcionamiento se basa en un array dinámico en tamaño.
 - Puede incluir elemento **null**.
 - **LinkedList**:
 - Permiten crear listas que se recorren hacia adelante y hacia atrás.
 - Implementa una lista doblemente enlazada.
 - Su iteración es más lenta que la de **ArrayList** pero la agregación de elemento por la cola o la cabeza es más rápida, así como la eliminación de un elemento.
- Si se van a realizar muchas operaciones de eliminación de elementos sobre la lista, o si las inserciones van a ser por la cola o la cabeza, conviene usar una lista enlazada (**LinkedList**), pero si mayoritariamente se van a insertar y consultar elementos por posición, conviene usar una lista basada en arrays redimensionados (**ArrayList**).

List → ArrayList

- Un **ArrayList** es una colección ordenada pero no clasificada.
- La característica más importante es el rápido acceso por posición, así como la iteración de todos los elementos.
- Para crear un **ArrayList** tenemos los siguientes constructores:
 - **ArrayList()**. Constructor por defecto. Simplemente crea un **ArrayList** vacío
 - **ArrayList(int capacidadInicial)**. Crea una lista con una capacidad inicial indicada.
 - **ArrayList(Collection c)**. Crea una lista a partir de los elementos de la colección indicada.

```
ArrayList a=new ArrayList();  
a.add("Hola");  
a.add("Adiós");  
a.add("Hasta luego");  
a.add(0,"Buenos días");  
for (Object o:a){  
    System.out.println(o);  
}
```



```
Buenos días  
Hola  
Adiós  
Hasta luego
```











```
public class Persona{
```

```
    private int idPersona;  
    private String nombre;  
    private int altura;
```

```
    public Persona(int idPersona, String nombre, int altura)  
{  
        this.idPersona = idPersona;  
        this.nombre = nombre;  
        this.altura = altura;}  
}
```

```
    public int getAltura() { return altura; }  
    //Omitimos otros métodos get y set para simplificar
```

```
    @Override  
    public String toString() {  
        return "Persona-> ID: "+idPersona+" Nombre: "+  
            nombre+" Altura: "+altura+"\n";  
    }  
}
```



```
public static void main(String[] args) {  
    List<Persona> lp = new ArrayList<Persona>();  
  
    Random r = new Random();  
  
    Persona temp = null;  
  
    int sumaaltura = 0;  
  
    for (int i = 0; i < 10; i++) {  
        lp.add(new Persona(i, "Persona" + i, r.nextInt(100) + 1));  
    }  
  
    for(Persona p:lp){  
        sumaaltura += p.getAltura();  
    }  
  
    System.out.println("La media de altura del conjunto de Personas es: " +  
        sumaaltura / lp.size());  
}
```

List → LinkedList

- Añade los siguientes métodos a la clase
 - **Object** **getFirst()** → Obtiene el primer elemento de la lista
 - **Object** **getLast()** → Obtiene el último elemento de la lista
 - **void** **addFirst(Object o)** → Añade el objeto al principio de la lista
 - **void** **addLast(Object o)** → Añade el objeto al final de la lista
 - **void** **removeFirst()** → Borra el primer elemento
 - **void** **removeLast()** → Borra el último elemento
- En el caso de definir una lista especificando el elemento genérico de la lista, deberíamos substituir **Object** por el objeto concreto **E**.

Algoritmo de análisis
de rendimiento de las
distintas colecciones
de tipo **List**

```
public static void main(String[] args) {  
    List<Persona> listaarray = new ArrayList<Persona>();  
    List<Persona> listalinked = new LinkedList<Persona>();  
    long antes;  
  
    for(int i=0;i<10000;i++) {  
        listaarray.add(new Persona(i,"Persona"+i,i));  
        listalinked.add(new Persona(i,"Persona"+i,i));  
    }  
  
    System.out.println("Tiempo invertido en insertar una persona en"+  
        "listaarray (en nanosegundos):");  
  
    antes = System.nanoTime();  
    // Inserción en posicion 0 de una persona  
    listaarray.add(0,new Persona(10001,"Prueba",10001));  
    System.out.println(System.nanoTime()- antes);  
  
    System.out.println("Tiempo invertido en insertar una persona en"+  
        "listalinked (en nanosegundos):");  
  
    antes = System.nanoTime();  
    // Inserción en posicion 0 de una persona  
    listalinked.add(0,new Persona(10001,"Prueba",10001));  
  
    System.out.println(System.nanoTime()- antes);  
}
```

Iterfaz iterator I

- La interfaz **Iterator** pertenece al *framework* **Collections**.
- Esta interfaz permite recorrer una colección de elemento y operar con el elemento al que se accede en cada momento:
 - **boolean hasNext()** → Indica si hay un elemento siguiente, así evita que se produzca una excepción
 - **E next()** → Obtiene el siguiente objeto de la colección. El intento de acceso más allá del final de la colección da lugar a una excepción de tipo: **NoSuchElementException** (que deriva a su vez de **RuntimeException**)
 - **void remove()** → Elimina el último elemento devuelto por **next()**.
- Se puede usar para todas las clases que implementan **Collection**.

Iterfaz iterator. Ejercicio

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

```
Iterator it= lista.iterator();
```

```
while(it.hasNext()){  
    String s = (String)it.next();  
    System.out.println(s);  
}
```

Al usar un iterador sin tipo, se debe utilizar casting para extraer el objeto.

Se puede extraer un objeto `Iterator` de cualquier objeto que implemente `Collection` con el método `iterator()`

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

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

```
while(it.hasNext()){  
    String s = it.next();  
    System.out.println(s);  
}
```

Al usar un iterador de tipo `String`, NO se necesita hacer *casting* para extraer el objeto y evitamos posibles problemas

Práctica

- Crear una clase denominada **Vehiculo** con los atributos **idVehiculo (int)** y **tipo (String)**, donde **tipo** podrá tomar los valores *Coche*, *Camión*, *Furgoneta* o *Moto*.
- Crea una clase con el método *main* donde se introduzcan **5000 vehículos** en una lista de tipo estático **List**. El atributo **tipo** debe establecerse para cada objeto de forma **aleatoria**.
- **Con la lista se deben realizar las siguientes operaciones:**
 - Un resumen de cuántos vehículos hay de cada tipo.
 - Recorrerse la lista y eliminar todos los vehículos que no sean de tipo **Coche**.
 - Añadir tantos vehículos de tipo **Coche** como se hayan eliminado, al final de la lista, de modo que los nuevos **ids** comenzarán a partir del último existente en la lista anterior.
 - Mostrar de nuevo el resumen de cuántos vehículos hay de cada tipo y el tiempo empleado desde que comenzó la eliminación de elementos hasta que terminó la inserción de elementos.
- **Responde a estas preguntas:**
 - Implementa el programa usando **ArrayList**. ¿Cuál es el resultado que obtienes?
 - Implementa el programa usando **LinkedList** ¿Cuál es el resultado que obtienes?
 - Haz varias ejecuciones y compara los resultados. ¿Observas diferencias entre la ejecución con **ArrayList** y con **LinkedList**? Si observas diferencias, ¿cuáles son y a qué crees que se deben?
 - ¿cuáles de las operaciones realizadas con **iterator** podrían realizarse con **for-each**?

Práctica

Clase
Vehiculo para
analizar
iterator()

```
public class Vehiculo {  
  
    private int idVehiculo;  
    private String tipo;  
  
    public Vehiculo(int idVehiculo, String tipo) {  
        this.idVehiculo = idVehiculo;  
        this.tipo = tipo;  
    }  
  
    public void setIdVehiculo(int idVehiculo) {  
        this.idVehiculo = idVehiculo;  
    }  
  
    public void setTipo(String tipo) {  
        this.tipo = tipo;  
    }  
  
    public int getIdvehiculo() {  
        return idVehiculo;  
    }  
  
    public String getTipo() {  
        return tipo;  
    }  
}
```



```
List<Vehiculo> lista = new LinkedList<Vehiculo>();
```

```
Random r = new Random();
```

```
//Se llena la lista con 5000 elementos de tipo aleatorio.
```

```
for (int i = 0; i < 5000; i++) {
```

```
    String tipo;
```

```
    switch (r.nextInt(4)) {
```

```
        case 0:
```

```
            tipo = "Coche";
```

```
            break;
```

```
        case 1:
```

```
            tipo = "Camión";
```

```
            break;
```

```
        case 2:
```

```
            tipo = "Furgoneta";
```

```
            break;
```

```
        case 3:
```

```
            tipo = "Moto";
```

```
            break;
```

```
    }
```

```
    lista.add(new Vehiculo(i + 1, tipo));
```

```
}
```

Algoritmo para
rellenar la lista

Map <K,V>

- La clase interfaz **Map** permite definir colecciones de *elementos que poseen pared de datos clave-valor*.
- Cada elemento tiene asociado una clave y un valor.
- **La clave** es utilizada para **acceder** de forma muy **rápida** a un **elemento**.
- Java permite que la **clave** sea cualquier **tipo de objeto** pero se suele utilizar Integer o String.
- Los mapas no permiten insertar objetos nulos ya que provocarían excepciones de tipo **NullPointerException**.
- **Ejemplo** de declaración de un **Map** con clave **Integer** y valor **String**.

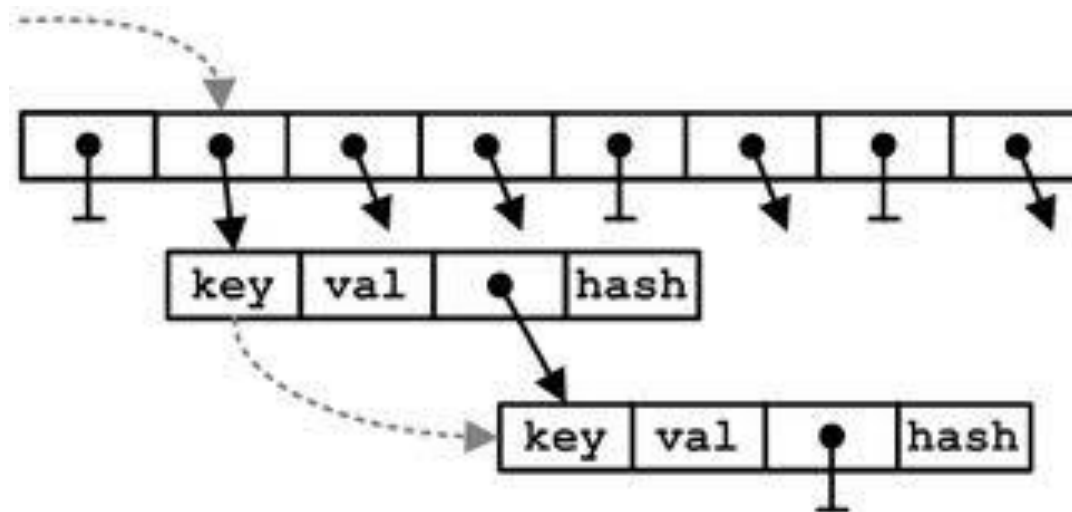
```
Map<Integer, String> nombreMap =  
    new HashMap<Integer, String>();
```

Map <K,V>. Métodos.

- Suponemos que K es el tipo de la clave y V el tipo del valor.
- **Los métodos destacados de Map son:**
 - **int size()** → Devuelve el **numero de elementos** del objeto Map.
 - **boolean isEmpty()** → Devuelve *true* si el objeto está **vacío**.
 - **void put(K clave, V valor)** → **Inserta** en el mapa el **elemento v** con la **clave k**.
 - **V get(K clave)** → **Devuelve** el **valor** de la clave que se le pasa como parámetro o **null** si la clave no existe.
 - **void clear()** → **Borra** todos los componentes del Map
 - **V remove(K clave)** → **Borra el par clave/valor** de la clave que se le pasa como parámetro.
 - **boolean containsKey(K clave)** → Devuelve *true* si en el *map* hay una clave que coincide con **K**.
 - **boolean containsValue(V valor)** → Devuelve *true* si en el map hay un Valor que coincide con **V**.
 - **Collection<V> values()** → Devuelve una Collection con los valores del objeto Map.
 - **Set<K> keySet()** → Devuelve un Set con los claves del objeto Map.

Map <K,V>. Condiciones

- Las operaciones fundamentales son **get**, **put** y **remove**.
- **El conjunto de claves no puede repetir la clave.**
- Las claves se almacenan en una tabla **hash** (es decir es una estructura de tipo **Set**) por lo que para detectar si una clave está repetida, **la clase a la que pertenecen las claves del mapa deben definir** (si no lo está ya) adecuadamente los métodos **hashCode()** y **equals()**.



Map <K,V>. Recorrer I

- Usando iterator.

- El método **keySet()** devuelve un **set** de claves.
- Se recuperan los valores del **map** utilizando las claves del **set** obtenido.

```
Iterator<String> it = map.keySet().iterator();  
while(it.hasNext()){  
    String key = it.next();  
    System.out.println("DNI: " + key + " -> Nombre: " + map.get(key));  
}
```

- Transformando Map en un Collection.

```
Collection <Persona> collection = map.values();  
Iterator <Persona> it = collection.iterator();  
while(it.hasNext()){  
    System.out.println(it.next().getNombre());  
}
```

Map <K,V>. Recorrer II

- Usando un bucle for.

```
//Para cada elemento key del conjunto map.keySet()
for (String key : map.keySet()){
    System.out.println(key + "=> " + map.get(key).getNombre());
}
```

```
//1 Entry es un key-value pair
for(Map.Entry<String, Persona> entry : map.entrySet()){
    String key = entry.getKey();
    Persona3 value = entry.getValue();
    System.out.println(key + "=> " + value.getNombre());
}
}
```

Map → HashMap

```
public static void main(String[] args) {
```

```
    Alumno a1 = new Alumno("Ana", 7.0);  
    Alumno a2 = new Alumno("Mateo", 8.5);  
    Alumno a3 = new Alumno("Andrea", 7.2);  
    Alumno a4 = new Alumno("Tomás", 8.0);  
    Alumno a5 = new Alumno("Gonzalo", 7.0);  
    Alumno a6 = new Alumno("Sofía", 8.0);
```

```
    HashMap<Integer, Alumno> lista = new HashMap<Integer, Alumno>();
```

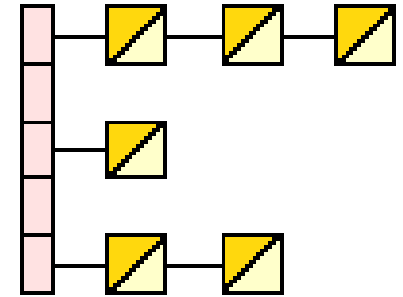
```
    lista.put(1, a1);  
    lista.put(2, a2);  
    lista.put(3, a3);  
    lista.put(4, a4);  
    lista.put(5, a5);  
    lista.put(6, a6);
```

```
    System.out.println(lista.get(4));
```

```
    lista.remove(4);
```

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

```
}
```



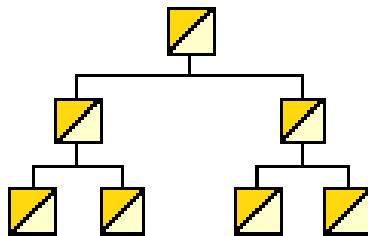
HashMap es la clase básica que implementa la interfaz **Map**, no añade ningún método ni funcionamiento en particular.

Usa una tabla hash para almacenar los datos.

El orden de entrada no se tiene en cuenta y no hay ningún orden establecido.

Map \rightarrow TreeMap<K,V>

- Almacena los datos en un árbol binario equilibrado ordenado.
- Los elementos clave/valor de un **TreeMap** se **ordenan** en **sentido ascendente** según la clave.
- Para que el orden previsto se pueda implementar es necesario que la claves implementen la interfaz **Comparable** y se **sobrescriba** el método **CompareTo**.
- Las operaciones de búsqueda y modificación son menos eficiente que en **HashMap**<K,V>.




```
public class Persona implements Comparable {
```

```
    private int idPersona;  
    private String nombre;  
    private int altura;
```

```
    private TreeMap<String, String> agendatel;
```

```
    public Persona(int idPersona, String nombre, int altura) {  
        this.idPersona = idPersona;  
        this.nombre = nombre;  
        this.altura = altura;  
        //inicialmente el mapa está vacío  
        this.agendatel = new TreeMap<String, String>();  
    }
```

```
    public TreeMap<String, String> getAgendatel() {  
        return agendatel;  
    }
```

```
@Override
```

```
    public String toString() {  
        return "Persona-> ID: " + idPersona + " Nombre: " + nombre + "  
Altura: " + altura + " \nAgenda:\n" +  
        agendatel.toString().replaceAll(",", "\n");  
    }
```

```
    /*Continúa en la otra diapositiva*/
```

Para cada objeto
se crea un
TreeMap para
almacenar datos
de teléfonos

Para ver el TreeMap lo
convertimos en String y
substituimos todas las
comas por saltos de línea

```
@Override
    public int hashCode() {
        int hash = 5;
        hash = 53 * hash + this.idPersona;
        return hash;
    }
    @Override
    public boolean equals(Object obj) {
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        final Persona other = (Persona) obj;
        if (this.idPersona != other.idPersona) return false;
        return true;
    }
    @Override
    public int compareTo(Persona p) {
        // Ascendente
        int resultado = this.nombre.compareTo(p.nombre);
        if (resultado == 0) {
            // Descendente
            resultado = Integer.compare(p.altura, this.altura);
        }
        return resultado;
    }
}
```

Se debe implementar el método `compareTo()` para determinar el orden en se almacenarán los objetos en `TreeMap`

```
public static void main(String[] args) {  
    Persona p = new Persona(1, "María", 167);  
  
    p.getAgendatel().put("Trabajo", "954825748");  
    p.getAgendatel().put("Oficina", "958746362");  
    p.getAgendatel().put("Móvil", "666555444");  
    p.getAgendatel().put("Casa", "952473456");  
  
    System.out.println("Información de Maria" + p)  
}
```

```
Información de MariaPersona-> ID: 1 Nombre: María Altura: 167  
Agenda:  
{Casa=952473456  
Móvil=666555444  
Oficina=958746362  
Trabajo=954825748}
```

Damos valor a los datos de la agenda de teléfonos del objeto persona que se acaba de crear

Al visualizar los datos de la agenda, se muestran ordenados por orden alfabético de la clave.

Map → LinkedHashMap<K,V>

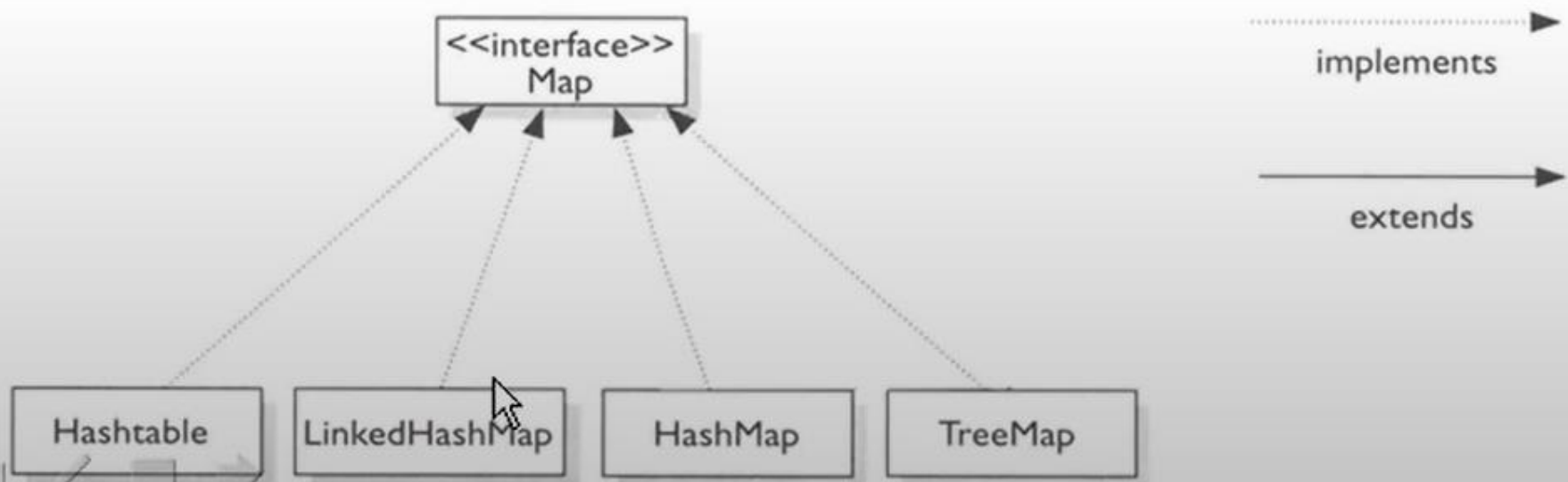
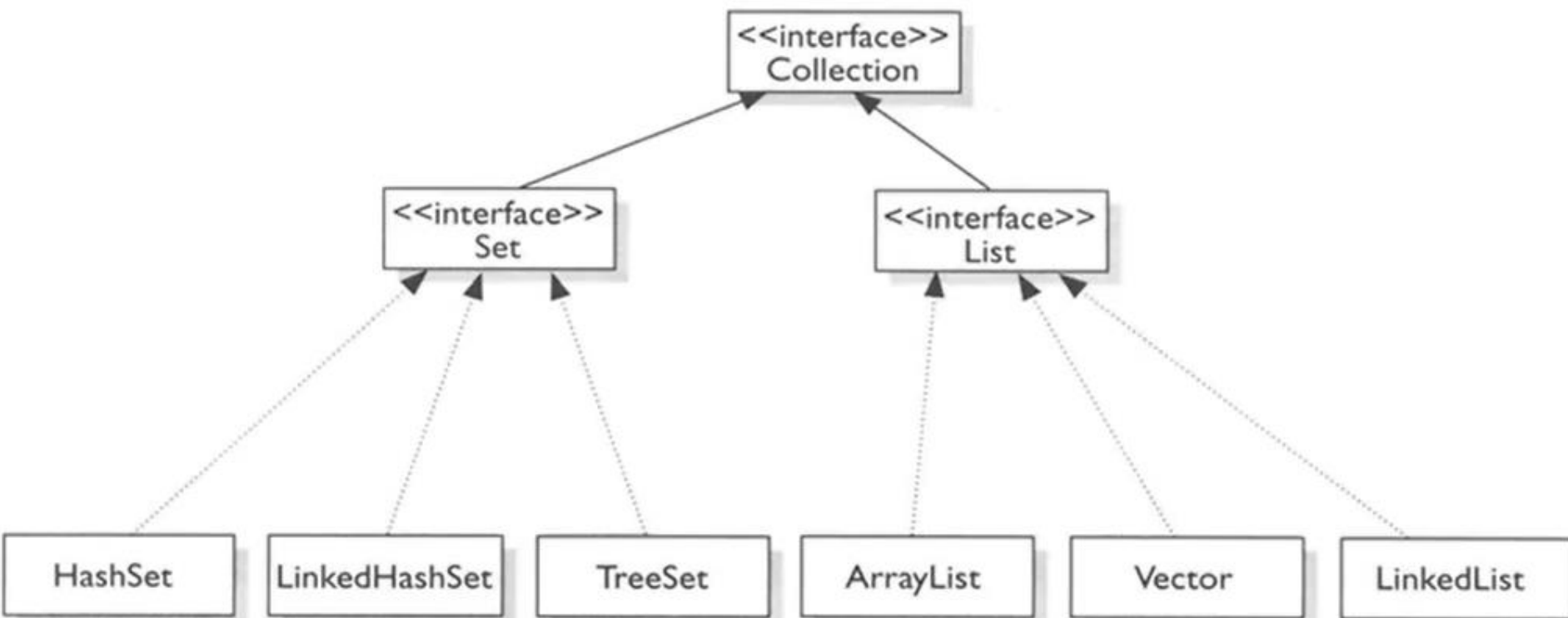
- **LinkedHashMap** respeta el orden en que fueron **insertados** los objetos del Mapa.
- Su característica más destacada es que permite que el orden de los elementos se refiera al último acceso realizado en ellos. Los más recientemente accedidos aparecerán primero.
- **Ejemplo:** Si en el anterior ejemplo cambiamos el tipo **TreeMap** por **LinkedHashMap**, la visualización de la información introducida respetaría el orden de introducción de datos.

```
Información de MariaPersona-> ID: 1 Nombre: María Altura: 167
Agenda:
{Trabajo=954825748
Oficina=958746362
Móvil=666555444
Casa=952473456}
```



Map. Ejercicio

- Almacena en un HashMap los códigos postales de las provincias de Galicia y Castilla-Leon. *Utiliza un iterador extraído de keySet()*
 - a. Muestra por pantalla los datos introducidos
 - b. Pide un código postal y muestra la provincia asociada si existe, sino avisa al usuario
 - c. Elimina las provincias León, Lugo y Valladolid.
 - d. Muestra por pantalla los datos



.....>
implements

————>
extends