Sesión 6: Colecciones

Índice

- **El Framework Collections**
- **Implementaciones de propósito general:**
  - ArrayList
  - LinkedList
  - HashMap
  - TreeMap
  - HashSet
  - TreeSet
- **Iteradores**

java.util: Colecciones

Colecciones

Java provee un API de colecciones con decenas de interfaces y clases dedicadas a ello.

Beneficios:

- Menos esfuerzo de programación para hacer lo mismo
- Aumenta la calidad del código y la velocidad de las operaciones
- Interoperabilidad: Todas las clases pertenecen a la misma API
- Curva de aprendizaje pequeña
- Reusabilidad

java.util: Colecciones

Collection

Las colecciones permitirán almacenar cualquier grupo de objetos y trabajar con ellos mediante el uso de métodos y funciones comunes.

Principales ***tipos de colecciones*** (heredan de Collection):

- **List:** ArrayList, LinkedList, Vector...
- **Set:** HashSet, TreeSet...
- **Map:** HashMap, TreeMap…

El framework Collections:
https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/doc-files/coll-index.html

java.util: Colecciones

Dificultades con los arrays

- Hay que conocer el tamaño previamente.
- Una vez se ha creado el array, el tamaño no puede ser modificado.
- Problemas para insertar objetos en posiciones intermedias o al inicio.
- No son realmente objetos.

```
int miArray[] = new int [10];
int array2[] = {5, 6, 7, 8};
```

List: ArrayList

ArrayList

Clase hija de List, es la colección más utilizada.

- Implementa una lista de elementos.
- Array dinámico: crece según se necesite.
- Cada elemento se guarda en una posición y tiene un índice.
- Los índices van desde 0 hasta tamaño - 1
- Permite elementos duplicados.
- Creación y consulta rápidas
- Insercion y eliminacion (al principio) ineficientes.

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ArrayList.html

Implementaciones de List

List: ArrayList

Constructores de un ArrayList

- **ArrayList()**
  - Construye una lista vacía.
- **ArrayList(Collection c)**
  - Construye un ArrayList a partir de una colección existente
- **ArrayList(int initialCapacity)**
  - Construye un ArrayList indicando una capacidad inicial

List: ArrayList

Algunos métodos de ArrayList

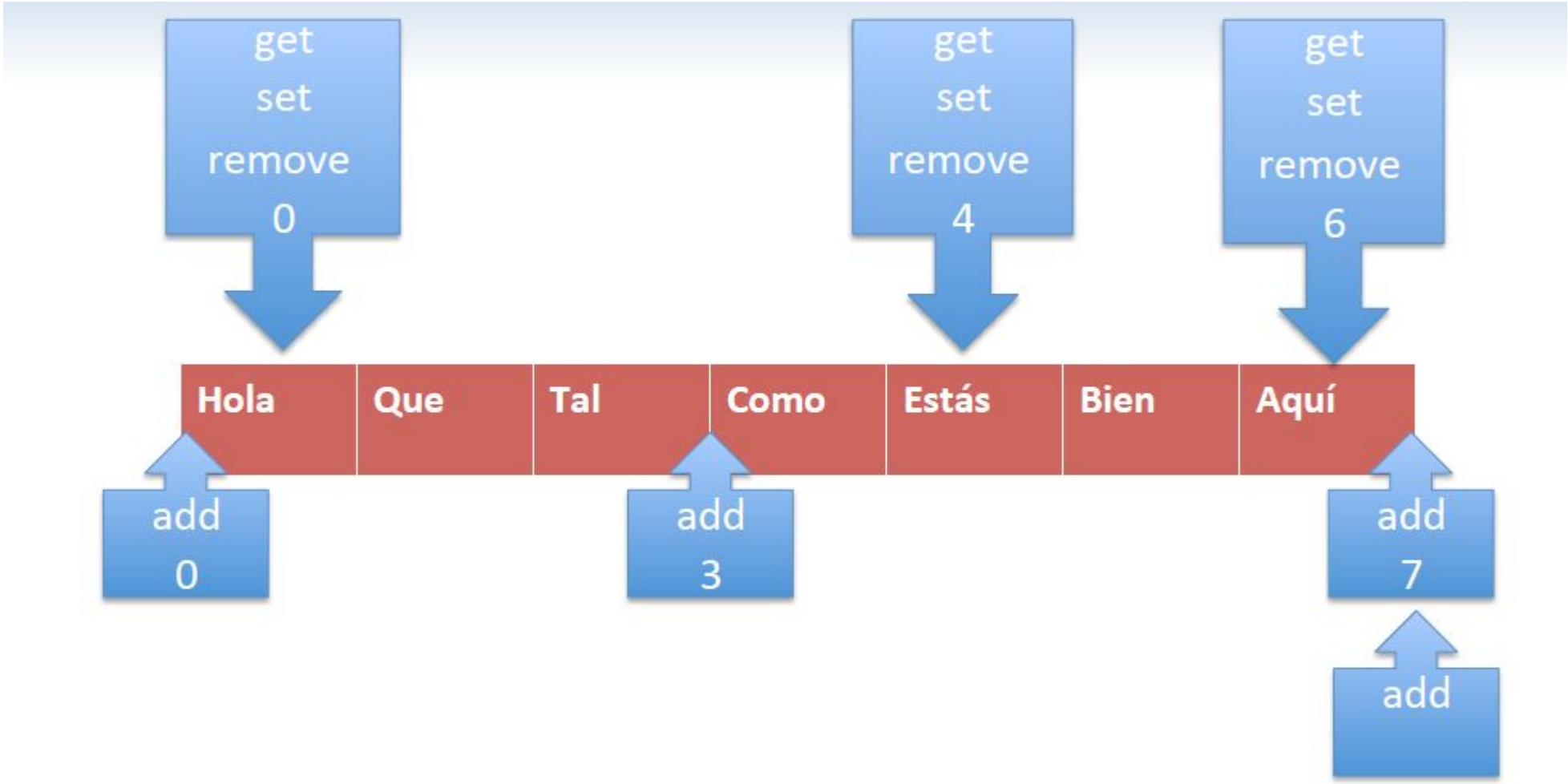
boolean **add** (Object obj) boolean add (int pos, Object o)	Añade un elemento (obj) al final de la lista. Devuelve true si el elemento se ha añadido correctamente, o false en caso contrario.
boolean **addAll** (Collection coll)	Añade los elementos de una Colección coll a la colección que llama al método. Devuelve true si se ha realizado la operación y false en caso contrario
void **clear**()	Elimina todos los elementos de la colección.
boolean **contains**(Object obj)	Comprueba si un elemento (objeto) está en la lista.
Object **get** (int indice)	Devuelve el elemento almacenado en la posición especificada.
boolean **isEmpty**()	Indica si la colección está vacía (no tiene ningún elemento).

List: ArrayList

Algunos métodos de ArrayList

int **indexOf**(Object o)	Devuelve la posición de la primera ocurrencia del elemento que se indica entre paréntesis.
boolean **remove**(Object o) boolean **remove**(int pos)	Elimina un determinado elemento (objeto) de la colección, devolviendo *true* si dicho elemento estaba contenido en la colección, y *false* en caso contrario.
Object **set**(int indice, Object obj)	Establece el objeto (obj) en la posición de la lista especificada, sobreescribiendo el objeto que hubiera en dicha posición.
int **size**()	Devuelve el número de elementos de la colección.
List **subList** (int inicio, int fin)	Devuelve una lista que incluye los elementos desde la posición indicada (inicio) hasta la final - 1 (fin - 1).
Object [] **toArray**()	Devuelve la colección de elementos como un array.

Algunos métodos de ArrayList



List: ArrayList

Implementación

Se implementa así:

```
ArrayList al = new ArrayList();
```

Aunque se recomienda asignar un tipo de datos al ArrayList:

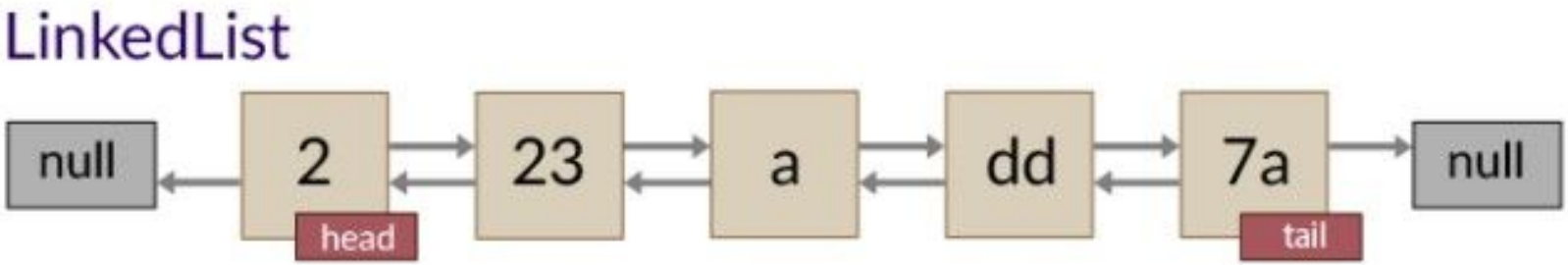
```
ArrayList<String> al = new ArrayList<>();
al.add("Pablo"); //añade un String al listado
al.add("Bernardo"); //añade otro String
al.add(1, "Nuria"); //añade el String en el índice 1, el resto
se mueve
al.set(1, "Alicia"); //Asigna el String en el índice 1,
sobreescribe valor
al.remove(0); //Elimina el elemento en el índice 0
```

List: LinkedList

- Implementa una lista vinculada de elementos.
- Cada elemento está doblemente vinculado: tiene un puntero al elemento anterior y otro al siguiente elemento.
- Se suele utilizar cuando se quiere implementar un listado que va a funcionar como una pila o cola de datos, en el que se realiza la inserción o extracción de elementos por un extremo de la pila.
- Tiene métodos especiales para manejar los elementos que están en los extremos de la pila
- Inserción rápida, acceso aleatorio ineficiente.

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/LinkedList.html

Implementaciones de List



List: LinkedList

Algunos métodos propios:

void **addFirst**(Object obj) void **addLast**(Object obj)	Añade un objeto (obj) al inicio o final de la lista.
Object **getFirst**() Object **getLast**()	Devuelve el primer o último objeto de la lista
Object **removeFirst**() Object **removeLast**()	Elimina el primer o último objeto de la lista
Object **pollFirst**() Object **pollLast**()	Devuelve y elimina el primer o último objeto de la lista

List: LinkedList

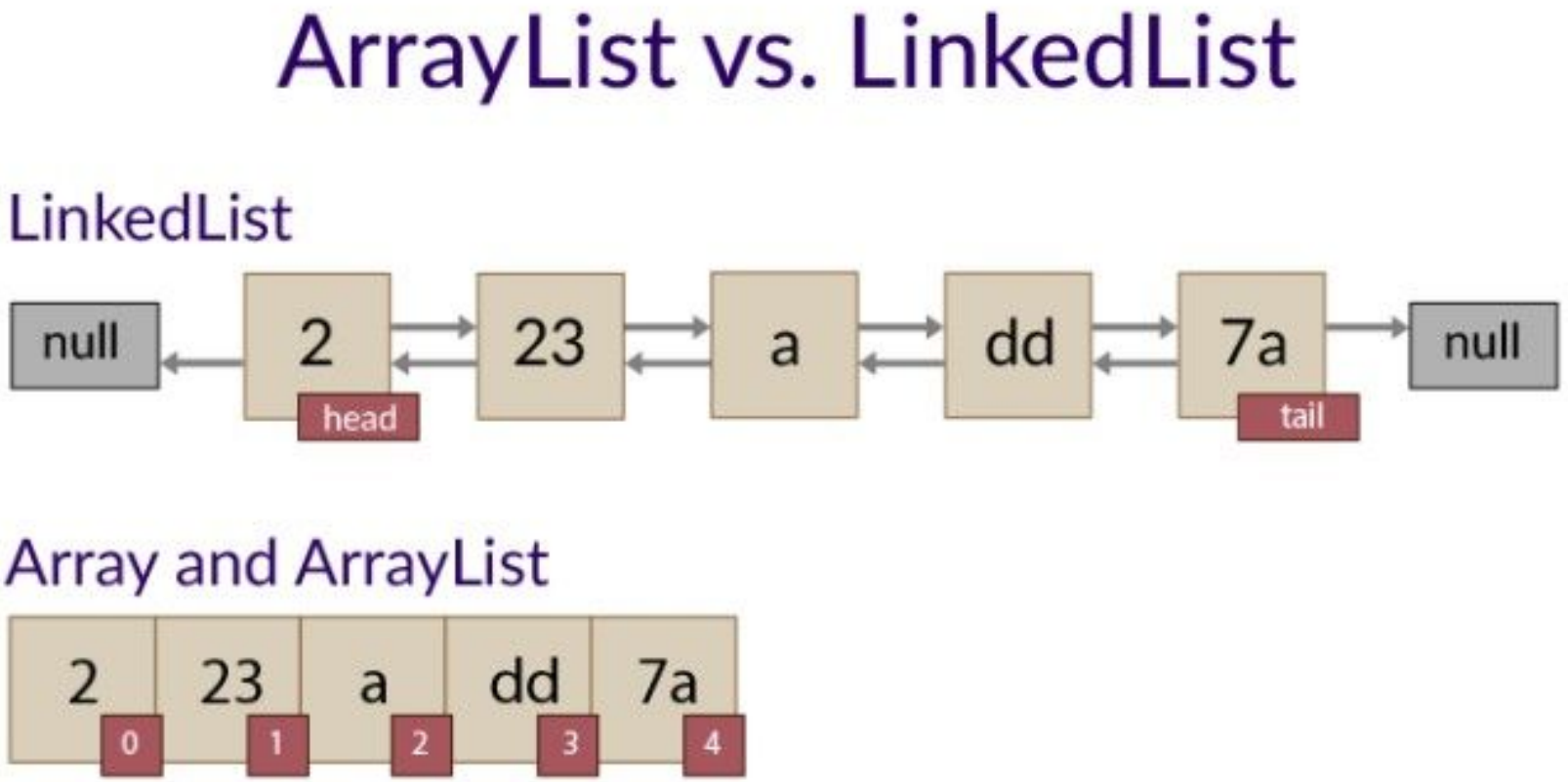
Implementación

```
LinkedList linked = new LinkedList();
```

Es recomendable asignar un tipo de datos al LinkedList:

```
LinkedList<String> lilist = new LinkedList<>();
lilist.add("E");
lilist.addFirst("A");  //añade en la primera posición
lilist.removeFirst();  //elimina la primera
String letra = lilist.getFirst();  //extrae el primer elemento
```

Implementaciones de List



Colecciones: Map

Map (Mapas)

Un mapa es un objeto que relaciona una clave (key) con un valor.

Por tanto, un mapa contendrá un conjunto de claves, y cada clave estará asociada a un elemento que se almacene en el mapa. Métodos propios:

Object **get**(Object clave)	Devuelve el valor asociado a la clave indicada
Object **put**(Object clave, Object valor)	Inserta una nueva clave con el valor especificado. Devuelve el valor que tenía antes dicha clave, o null si la clave no estaba en la tabla todavía.
Object **remove**(Object clave)	Elimina una clave, devolviendonos el valor que tenía dicha clave.
boolean **containsKey**(Object c) boolean **containsValue**(Object c)	Devuelve *true* si el mapa contiene la clave. Devuelve *true* si el mapa contiene el valor..
int **size**()	Devuelve el número de parejas (clave,valor) registradas.

Map: HashMap

Clases de Map:

- **HashMap**

  Almacena el mapa en una tabla *hash* (utiliza el hash code para identificar los datos).

  Tiempo de ejecución de operaciones básicas constante, incluso en mapas grandes.

  No garantiza el orden de los elementos

  https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashMap.html

- Implementación:

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

hm.put("Juan", 32);

hm.put(“Maria”, 22);
```

Map: TreeMap

Clases de Map:

- **TreeMap**

  TreeMap implementa la interfaz Map usando una estructura árbol para el almacenamiento.

  Por tanto, los elementos se muestran ordenados por sus claves, en orden ascendente.

  Los tiempos de acceso y lectura de datos son rápidos, aunque más lentos que HashMap. Su rendimiento es *Log(N)* en operaciones básicas.

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/TreeMap.html

- Implementación:

```
TreeMap<String> tm = new TreeMap<String>();

tm.add("B");
```

Colecciones: Set

Set (Conjuntos)

Un conjunto es un grupo de elementos en el que no puede haber ningún elemento repetido.

Para determinar si un elemento es repetido, hace uso interno del método equals.

No implementa métodos propios, hereda los métodos de List: add, contains, remove…

No implementa el método get, por lo que se necesita un iterador para recorrer el Conjunto comprobando si está nuestro objeto.

Set: HashSet

Clases de Set:

- **HashSet**

  Almacena los elementos en una tabla *hash.*

  Utiliza el hash code para identificar los datos.

  Es la implementación Set con mejor rendimiento y, por tanto, la más utilizada.

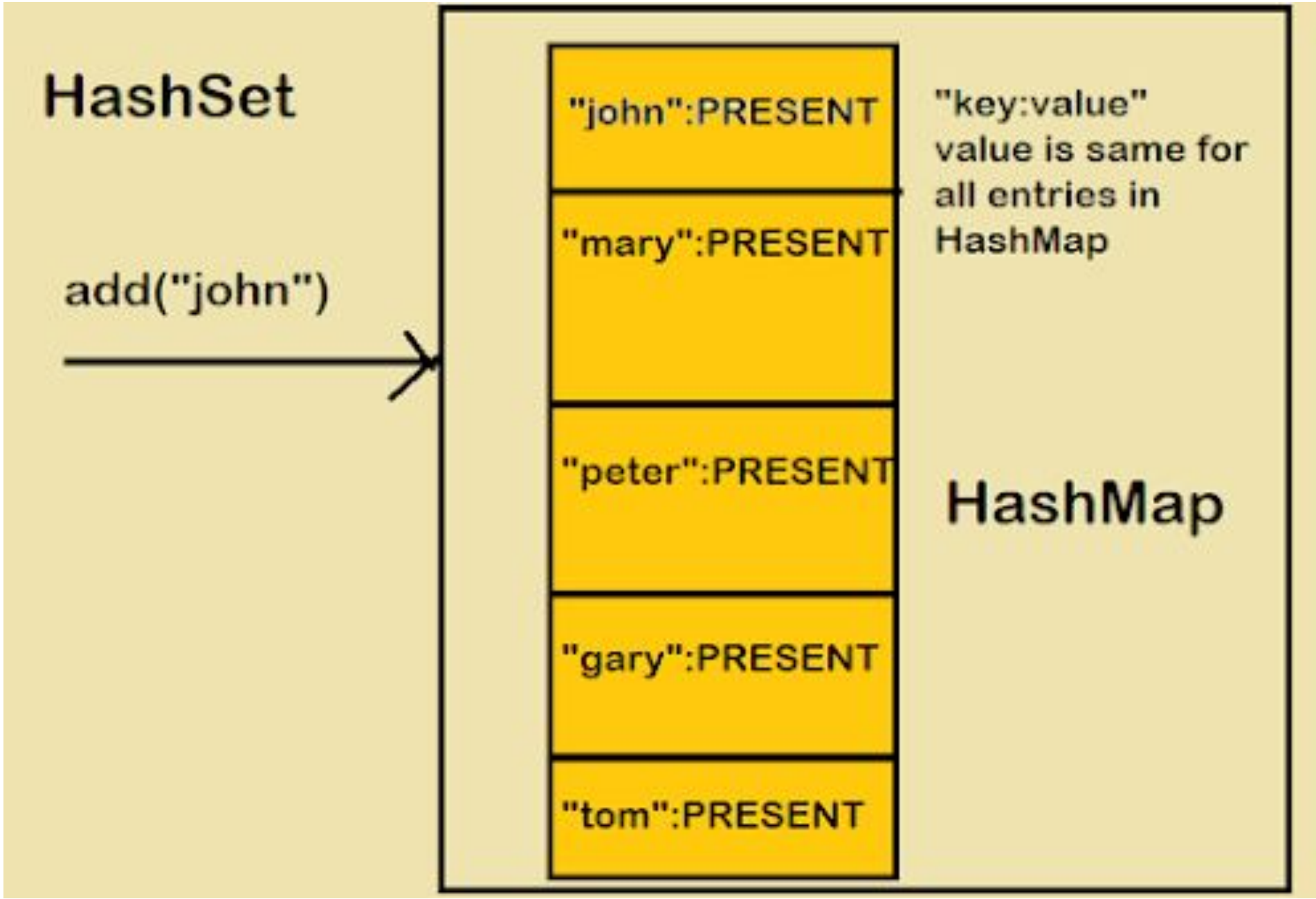
  No garantiza un orden de los elementos

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashSet.html

- Implementación:

```
HashSet<String> hs = new HashSet<>();
hs.add("B");
```

Un HashSet es internamente un HashMap



Set: TreeSet

Clases de Set:

- **TreeSet**

  Almacena los elementos en estructura árbol.

  Por tanto, los elementos se muestran ordenados, en orden ascendente.

  Los tiempos de acceso y lectura de datos son rápidos, aunque más lentos que HashSet, ya que tiene que comparar elementos para su orden. Su rendimiento es logarítmico (*Log(N))* en operaciones básicas.

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/TreeSet.html

- Implementación:

```
TreeSet<String> ts = new TreeSet<String>();

ts.add("B");
```

Colecciones: Iterator

Un iterador permite recorrer los elementos de una colección de forma eficiente y segura.

Cada una de las clases de Collection proporciona un método iterator(); Por tanto, haciendo un uso de un Iterador y un Bucle, podremos acceder a cada elemento.

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Iterator.html

Implementación:

```
Iterator it = col.iterator()  //col es cualquier colección:
ArrayList, HashMap, etc..
```

Pasos:

1. Obtener un iterador de la colección llamando al método iterator()
2. Montar un bucle (while) que llame al método hasNext(), que devuelve true siempre que haya un elemento en la siguiente iteración.
3. Dentro del bucle, se obtiene cada elemento llamando al método next();

Colecciones: Iterator

Métodos:

boolean **hasNext**()	Devuelve *true* si hay más elementos. De lo contrario devuelve *false*.
Object **next**()	Devuelve el siguiente elemento
void **remove**()	Elimina el elemento actual. Produce una excepción si se llama a remove sin llamar antes a next()

Implementación:

```
ArrayList<String> al = new ArrayList<String>();
Iterator<String> itr = al.iterator();

while (itr.hasNext()) {
    String letra = itr.next();
    System.out.println(letra);
}
```

Colecciones: Iterator

Implementación con Map:

Para iterar en una colección Map con un Iterator, necesitamos obtener su conjunto de claves.

Map proporciona un método que devuelve una colección (Set) de sus claves (keys).

Una vez se dispone de la colección de claves, se puede iterar sobre ella.

```
HashMap<String, Double> hm = new HashMap<String, Double>();
Set<String> keys = hm.keySet();
Iterator<String> ith = keys.iterator();

while (ith.hasNext()) {
  String key = ith.next();
  hm.put(key, 0.0);
}
```

Colecciones: Otros iteradores

- **for-each**

  Una forma de iterar usando el bucle *for*, pero reduciendo significativamente el código.

  Como desventaja frente al iterator, no permite eliminar elementos. Solo acceder al elemento.

  Sintaxis:

```
for (<tipo dato array> <nombre variable> : <Array/Lista/Coleccion>)
```

  Ejemplo:

```
ArrayList<String> al = new ArrayList<String>();
for (int i = 0; i < al.size(); i++)
 System.out.println (al.get(i));

for (String string : al)
 System.out.println (string);
```

Colecciones: Otros iteradores

- **ListIterator**

  ListIterator extiende (o hereda) de Iterator.

  Permite atravesar una lista bidireccionalmente.

  Posee sus propios métodos para ello.

  https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ListIterator.html

  Ejemplo:

```
ArrayList<String> al = new ArrayList<String>();
   ListIterator<String> itr = al.listIterator();

while (itr.hasPrevious()) {
    String letra = itr.previous();
    System.out.println(letra);
}
```