# Introducción a Java

16.10.2025

Enrique Krause Buedo

# Lectura de datos a través de la consola del sistema

## Sistema de lectura de datos en la consola del sistema; clase Scanner

```
import java.util.*;
public class DameTusDatos{
 public static void main(String args[]){
  Scanner sc = new Scanner (System.in);
  System.out.print ("Introduce tu nombre: ");
  String nombre = sc.nextLine();
  System.out.print ("Introduce tu edad: ");
  String aux = sc.nextLine();
  int edad = Integer.parseInt (aux);
  System.out.print ("Introduce tu estatura: ");
  aux = sc.nextLine(); double estatura = Double.parseDouble (aux);
  System.out.println("Tus datos son:");
  System.out.printf ("Nombre:%s\nEdad:%d\nEstatura: %5.2f", nombre,
        edad, estatura);
}
```

```
Introduce tu nombre:
Cristina Introduce tu
edad: 24 Introduce tu
estatura: 1.65 Tus datos
son; Nombre:Cristina
Edad: 24
Estatura: 1,65
```

*Uso de wrapper*

*Uso de ”mascarillas” (se les dice así...)*

# Lectura de datos a través de la consola del sistema

## Con validación

Son muchas las ocasiones en las que se requiere que el usuario introduzca un dato y este dato tenga que cumplir unas determinadas condiciones específicas, así por ejemplo, no se admitiría que un usuario introdujera una edad negativa o mayor de 110 años. En otras ocasiones se tratará de un dato formado por una cadena de caracteres el que tiene que ser validado, por ejemplo, el programa solicita el nombre del usuario, éste debe estar formado al menos por tres caracteres para que pueda considerarse un nombre válido. En general el proceso de validación de datos en la consola del sistema se realiza envolviendo a la petición de datos dentro de una estructura de bucle repetitivo do-while, de tal forma que, sino se cumplen las condiciones de la validación, el programa repite la petición hasta que se cumplan, y sólo termina la petición cuando el usuario introduce un dato válido.

El siguiente código hace una validación muy paupérrima de un mail

# Lectura de datos a través de la consola del sistema

## Con validación

```
import java.util.*;
public class DameTuCorreo{
  public static void main(String [] args){
  Scanner sc = new Scanner(System.in);
  String correo;
  do{
    System.out.print ("Introduce tu mail ");
    correo = sc.nextLine();
    if(correo.indexOf("@")< 0 || correo.length() < 3)
    System.out.println("Tu correo no es válido");
  }while (correo.indexOf("@")<0 || correo.length() < 3);
  System.out.print ("Tu dirección mail es admitido");
 }
}
```

# 6 Colecciones de longitud fija (arrays)

16.10.2025

Enrique Krause Buedo

# Concepto de array

Un array es una estructura de almacenamiento de un número determinado de variables primitivas o referencias a objetos, todas del mismo tipo y que cada elemento está identificado por uno o varios índices. Con los arrays podemos construir vectores, matrices y demás estructuras repetitivas de datos, es decir, de variables primitivas y referencias a objetos. Los arrays son estructuras de gran utilidad, sólo limitada por el hecho de que su dimensión debe estar previamente declarada y no puede ser ampliado en tiempo de ejecución.

Todo array tiene asociada la propiedad length que indica el número de elementos máximo que puede contener, Cada uno de los elementos del array se identifica mediante su número de orden o índice. Un array puede tener uno o más índices. Por ejemplo, con dos índices podríamos tener matrices, de tal forma que cada elemento se identificará por la fila y la columna en la cual se encuentra. Los índices siempre empiezan en cero, así en un array de tres elementos contaríamos con los elementos Cero, uno y dos.

# Concepto de array

Cuando es definido un array de referencias a objetos, éstos toman por defecto el valor nulo (null) hasta que se le asigne una referencia a un objeto existente. En el caso de variables primitivas toman el valor cero, si son numéricas, y false, si son de tipo boolean.

**Declaración de arrays**

En el lenguaje Java todos los arrays deben ser declarados y construidos antes de ser usados. En los ejemplos siguientes se muestra cómo declarar arrays de varios tipos:

```
//Estas dos declaraciones son equivalentes
int vector[];
int[] vector;
boolean resultados[];
String nombres[];
```

# Definición o construcción de arrays

Un array una vez declarado debe ser definido, proceso que también se puede identificar como construcción, en el cual se reserva el espacio necesario para almacenar los datos o los objetos que va a contener. Los siguientes ejemplos muestran cómo se definen diferentes tipos de arrays.

La siguiente sentencia construye un array de números enteros con un solo índice desde el elemento vector[0] al elemento vector[29]

```
vector = new int[30];
```

La siguiente sentencia construye un array de variable tipo boolean, con un solo índice, desde el elemento resultados [0] al elemento resultados [9]

```
resultados = new boolean[10];
```

La siguiente sentencia construye un array de objetos de la clase String, con un solo índice, desde el elemento nNombres [0] al elemento nombres [nNombres-1], siendo nombres una variable de tipo entero con el número de nombres que puede contener el array.

```
nombres = new String [nNombres];
```

# Definición o construcción de arrays

Un array de dos índices puede tener en cada fila componentes de distinta longitud, de tal forma que se pueden construir, por ejemplo, matrices triangulares. Las siguientes sentencias declaran y definen un array de dos índices, con la primera fila de dos elementos, la segunda de cuatro y la tercera de cinco:

```
int matriz[][]= new int[3];
matriz[0]= new int[2];
matriz[1]= new int[4];
matriz[2]= new int[6];
```

Para dimensionar una matriz de dos índices con todas las columnas del mismo tamaño se puede utilizar la siguiente sentencia:

```
double matriz2D = new double[5] [3];
```

El array matriz2D queda dimensionado con un total de cinco filas y tres columnas.

También es posible dimensionar arrays de más índices.

```
int matriz3D = new int [3] [5] [2];
```

## Definición o construcción de arrays

Es posible combinar la declaración y la definición; en las sentencias siguientes se muestra cómo los dos pasos anteriores de declaración y definición se pueden realizar en una sola sentencia.

```
int vector [ ] = new int[30];
boolean resultados = new boolean[10];
String nombres [ ] = new String [ nNombres];
```

Finalmente, también es posible realizar la declaración, definición e inicia- lización del array dentro de una misma sentencia. En los ejemplos que se ilustran a continuación se muestra cómo realizar las tres tareas en una única sentencia:

```
int valoresPosibles [] = {2, 3, 5, 13, 13};
String nombres [] [] = {{"Juan","Pedro","Benito"}, {"Antonio",
"Manuel", "José}};
```

Para investigar la longitud de un array se obtiene el valor del atributo. length del array. La siguiente sentencia: `int longitud = vector.length;`

## Definición o construcción de arrays

```
int longitud = vector.length;
```

Obtiene el número de elementos del array de nombre vector declarado y definido anteriormente. La sentencia:

```
int nFilas = matriz.length;
```

Obtiene el número de filas del array de dos índices de nombre matriz y la sentencia:

```
int nColumasl = matriz[0].length;
```

Obtiene el número de columnas de la primera fila de la matriz, es decir, del primer array componente de matriz.

*Es importante distinguir entre la propiedad length de un array y el método length() de un objeto de la clase String. Este último siempre se usará con paréntesis, ya que se trata de un método. Un array una vez construido siempre tendrá la misma longitud, tenga o no datos; en un objeto de la clase String puede contener más o menos caracteres, por eso para conocer su longitud hay que acudir a un método para que calcule el número de caracteres que contiene.*

# Ejemplos de uso de arrays

El programa siguiente muestra cómo utilizar la estructura repetitiva for para gestionar un array siguiendo: for (int i=0 ; i< nombres.length ; i++)

Y la estructura for estilo **”foreach”**, de la que no hemos hablado, y en la que se declara que: "para cada elemento del array realizar las sentencias que encierra el for", que en este caso es una única sentencia que muestra por pantalla el contenido de cada elemento del array.

```
for(String nombre : nombres)
```

Igualmente muestra cómo utilizar la propiedad length del array (nombres.length).

Cuando se necesita acceder un atributo de un objeto se puede utilizar la siguiente forma:

```
Nombre_objeto.nombre_atributo;
```

Observe que la sintaxis es la misma que la que tiene el acceso a un método, salvo que en ésta siempre se ponen los paréntesis de los argumentos en el nombre del método.

# Ejemplos de uso de arrays

```
public class SegundoArray{
 public static void main(String arg[]){
  String nombres[]={"Madrid","Barcelona","Toledo","Santiago"};
  nombres [2]="Sevilla";
  //versión según estilo for normal
  //for (int i=0;i< nombres.length; i++)
  //versión siguiendo estilo foreach
  for( String nombre:nombres){
    System.out.println (nombre) ;
  }  }
```

**Resultado**

```
Madrid Barcelona Sevilla Santiago
```

Siempre que tenga que gestionar un array, la estructura de control de flujo más recomendable es la estructura for.

# Utilidades de la clase Arrays

La clase Arrays del paquete java.util soporta utilidades para los array tales como ordenación, comparación entre arrays y relleno con datos. El código muestra cómo es posible usar el método sort de la clase Arrays para ordenar un array.

```
import java.util.*;
public class Ordena{
 public static void main(String args[]){
  double números[] = {32.0,45.5,12.2,28.4,76.3, 2.1,53.0,31.4,29.5};
  System.out.println("Antes de la ordenación");
  for (double número : números)
  System.out.print (número+" ");
  Arrays.sort (números);
  System.out.println("Andespués de la ordenación");
  for(double número : números)
  System.out.print (número+" ");
  }
}
```

```
Resultado  Antes
de la ordenación
32.0 45.5 12.2
28.4 76.3 2.1
53.0 31.4 29.5

Después de la
ordenación 2.1
12.2 28.4 29.5
31.4 32.0 45.5
53.0 76.3
```

# Arrays

***Os toca hacer muchos ejercicios con arrays (hay que conocerlos bien)***

También en la plataforma Moodle, es necesario y conveniente que miremos juntos los puntos

**2.3 Sobre todo la parte relacionada con los métodos**

Entender que en los casos de bastantes ejercicios conviene implementar funcionalidades dentro de métodos (en la clase, pero fuera del método main), de manera que se puedan invocar varias veces

**2.4 Atributos y métodos estáticos**

Nos vendrá bien entender el concepto de cara a lo que veremos

# 7 Colecciones de longitud variable Collections

16.10.2025

Enrique Krause Buedo

# El paquete java.util

El paquete java.util contiene numerosas clases e interfaces que posibilitan la gestión de objetos en listas o colecciones, éstas se agrupan en lo que se denomina la API Collections. Igualmente contiene clases que posibilitan la generación de números aleatorios, gestión de fechas, creación y expansión de ficheros jar y zip, etc. En este libro se expone cóm utilizar algunas de las clases más utilizadas. El lector una vez que sabe aplicar las clases expuestas en este libro, estará en condiciones de explorar otras clases de este paquete.

16.10.2025

Enrique Krause Buedo

# Tipos genéricos

Una de las novedades más destacadas de la versión 5 de java, era la notación de tipos genéricos que simplificaba los procesos de conversión tipos en las colecciones de objetos. Un ejemplo del cambio que supone la nueva notación. Antes de la versión 5 la forma de declarar una colección de objetos con la clase ArrayList era la siguiente:

```
ArrayList lista = new ArrayList ();
```

En la que no había que declarar de qué tipo van a ser los objetos de la lista, esto obligaba a que en la recuperación de dichos objetos había que hacer una transformación de tipos, como:

```
int total = ((Integer)lista.get(0)).intValue();
```

Permitiendo incluso que en la lista existieran objetos de diferente tipo, con el consiguiente problema de gestión e incluso de estabilidad en la lista.

**A partir de la versión 5, es obligatorio declarar de qué tipo van a ser los objetos de la lista, tal como se recoge en la sentencia siguiente:**

# Tipos genéricos

**Es obligatorio declarar de qué tipo van a ser los objetos de la lista, tal como se recoge en la sentencia siguiente:**

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

por lo que en la recuperación de los objetos no es necesario la transformación de tipos (casting) que era obligatorio anteriormente. Así de esta forma se recupera un objeto de la lista, sin especificar la transformación de tipos:

```
int total = lista.get(0).intValue();
//antes de la versión J2SE5
ArrayList lista = new ArrayList();
lista.add(0, new Integer(42));
int total = ((Integer)lista.get(0)).intValue();
//después de la versión J2SE5
ArrayList<Integer> lista = new ArrayList<Integer>();
lista.add(0, Integer.valueOf(42));
int total = lista.get(0).intValue();
```

# Listas y colecciones del paquete java.util

Lista de las clases de uso frecuente para la creación de colecciones de objetos en el lenguaje Java. Estas colecciones tienen ventaja de poder ser ampliadas según las necesidades de la colección, contrario que los arrays que una vez dimensionados no es posible ampliarlos.

Todas estas clases de implementación son del tipo Cloneable y Serializable.

La clonación es el proceso de duplicación de un objeto para que en memoria existan dos objetos idénticos en el mismo instante de tiempo. Se usa, por ejemplo, cuando se tiene que pasar una referencia de un objeto a un método y deseamos garantizar que este método no modifique el objeto. Se suele usar la clonación en de la siguiente forma:

```
cualquierObjeto.cualquierMetodo (miObjetoImportante.clone());
```

# Listas y colecciones del paquete java.util

La serialización es el proceso por el cual los objetos de una clase se pueden archivar y recuperar en un archivo de forma sencilla apelando a los métodos writeObject y readObject de la clase FileOutputStream FileInputStream cuando éstos se aplican a objetos de una clase que mplemente el interfaz Serializable.

Se detalla algunas de las clases más utilizadas.

**ArrayList:** Lista enlazadas similar a la clase Vector, que no permite sincronización.

**LinkedList** Permite crear listas de datos con repetición y ordenadas

**TreeMap** Permite crear listas de datos sin repetición y ordenadas. Una implementación de ***SortedMap*** utilizando un árbol binario equilibrado que mantiene sus elementos ordenados por clave útil para conjuntos de datos ordenados que requieren una búsqueda por clave moderadamente rápida.

# Clases de tipo interface para crear listas y colecciones del paquete java.util

Lista de clases de tipo ***interface*** que permiten crear clases con utilidades que complementan los tipos de listas anteriores. Estas clases de tipo interface tienen la ventaja de que el código que se genera con ellas se separa la especificación de la implementación por lo que es más sencillo reemplazar una clase por otra que impleme la misma interfaz

**Collection** La interfaz raíz de las colecciones. Todos los objetos que derivan de la clase Collection se caracterizan por consi grupos de objetos individuales con una relación de orden entre ellos. Todas las clases que derivan de Collection comparten los siguientes métodos: **add(), isEmpty(), contains() clear() remove(), size(), toArray() e iterator().**

## Clases de tipo interface para crear listas y colecciones del paquete java.util

**Set** Una colección (conjunto) donde no puede haber elementos repetidos, y cuyos elementos no se almacenan necesariamente siguiendo un orden particular.

**SortedSet** Clase abstracta que permite crear conjuntos con elemens ordenados. Añade los siguientes métodos: first(), last(), subSet(), headSet(), tailSet().

**TreeSet** Derivado instanciable de SortedSet en el que los elementos están ordenados mediante una estructura de árbol.

**HashSet** Derivado instanciable de SortedSet implementado con una tabla hash. Como ventaja aporta que el número de objetos contenidos en la estructura no repercute en las velocidad de las operaciones de búsqueda, eliminación e inserción. La desventaja estriba en que no se pueden recorrer según un orden fijado por el programador.

## Clases de tipo interface para crear listas y colecciones del paquete java.util

**List** Colección ordenada que puede tener objetos duplicados. Una colección cuyos elementos permanecen en un orden particular a menos que se modifique la lista. Aporta los siguientes métodos a la clase Collection de la que deriva: **get(int índice), set(int indice, Object), add(int índice, Object), remove, indexOf, subList (min,max).**

**Map** Un lista que asocia una clave con un valor. Los objetos de la clase Map tiene los siguiente métodos **size, isEmpty, containsKey(object), containsValue(Object), get(object), put (object object), remove(Object), clear().**

**SortedMap** Clase abstracta derivada de Map que hace que los elementos que lo constituyen estén ordenados. Permite crear listas cuyas claves están ordenadas.

## Clases de tipo interface para crear listas y colecciones del paquete java.util

**Iterator** Una interfaz para objetos que devuelven elementos de una colección de uno en uno. La interfaz Iterator se usa para recorrer colecciones de objetos.

**ListIterator** Un iterador para objetos List que añade métodos de utilidad relacionados con las listas.

Ejemplo de uso del interface Iterator.

```
HashSet <String> lista=new HashSet <String>();
    lista.add ("Juán");
    lista.add("Antonio");
    lista.add("Ana");
    lista.add ("María");
    @SuppressWarnings("rawtypes")
    Iterator i=lista.iterator();
    while(i.hasNext()){
    System.out.println(i.next().toString());
    }
```

# La clase ArrayList

La clase ArrayList es una lista que utiliza un array de dimensión modificable. Es una lista de uso sencillo, si bien requiere algo más de tiempo de proceso que el deseable, sobre todo en las operaciones de añadir o borrar un elemento.

## Métodos

**boolean add(Object o)** Añade un objeto a la lista

**boolean contains(Object o)** Investiga si un objeto esta en la lista

**void clear()** Borra la lista

**Object remove(int índice)** Borra el objeto que hay en la posición definida por índice

**Object get(int índice)** Obtiene el objeto que esta en la posición índice

**Object set(int índice, Object elemento)** Pone la referencia del objeto en la posición definida por índice

**boolean isEmpty()** Investiga si está vacía la lista

**int indexOf(Object o)** Obtiene el objeto que hay en la posición definida por índice.

# La clase ArrayList

El siguiente código recoge la clase Agenda en la que se crea una lista con la clase arrayList. La sentencia:

```
private ArrayList<Persona> lista;
```

declara lista como un objeto de la clase ArrayList forzando que la lista creada sean objetos de la clase Persona. La sentencia:

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

crea la lista de la clase ArrayList para albergar objetos de la clase Persona
La sentencia:

```
for (Persona i: lista)
```

corre la lista extrayendo el objeto i, tal como se haría con la estructura for en su sintaxis clásica (en estos casos es más aconsejable utilizar el tipo foreach anterior):

```
for(int i=0; i< lista.size();i++)
```

Entonces crearemos: una clase **Agenda**, una clase **Persona**, una **Excepción** propia, y Un fichero/clase **GesionaAgenda**

# La clase ArrayList

```
import java.util.*;
public class Agenda{
 private ArrayList<Persona> lista;
     Agenda(){
        lista = new ArrayList<Persona> ();
     }
     public int anadePersona(Persona p) throws AgendaExcepcion{
      if (buscaPersona(p.dameDNI())!=null) throw new
AgendaExcepcion("Lapersona con el DNI "+ p.dameDNI ()+ " ya está en la
lista");
      else
       lista.add(p);
      return lista.size();   }
```

# La clase ArrayList

```
public boolean borraPersona(Persona p){
        for( Persona i: lista) {
           if (i.dameDNI().equals(p.dameDNI())){
             lista.remove (1);
             return true;       }}
           return false;
}
public String muestraListaPersona(){
      String temp="";
      for(Persona p: lista)
        temp += p.muestraPersona();
        return temp;
    }
    public Persona buscaPersona (String dNI){
      for( Persona p: lista)
        if (p.dameDNI().equals(dNI))
            return p;
        return null;
    }   }
```

La sintáxis se ha seguido permitiendo licencias de modo que entre el código.
Con esta clase declaramos métodos que añaden, borran, muestran todos y buscan a uno de los registros de nuestra agenda
**Lo siguiente sería declarar y desarrollar la clase Persona, el código que gestiona la excepción marcada en rojo, y el main**

# La clase ArrayList

```
public class Persona{
 private String nombre;
 private String dNI;
 Persona(String nom, String dNI){
    nombre=nom;
    dNI=dNI;
 }
 Public String muestraPersona(){
    return ”Nombre: ”+ nombre+ \t + ”DNI: ”+ dNI + ”\n”;
 }
 Public String dameDNI(){
    return dNI;
 }
}

public class AgendaExcepcion
        extends RuntimeException{
  AgendaExcepcion(String mensaje){
        super(mensaje);
   }
}
```

# La clase ArrayList

```
public class Gestionaagenda{
 public static void main(String[]args){
    Agenda agenda = new Agenda();
    Persona p= new Persona("Juan",”111”);
    try{
        agenda.anadePersona(p);
        p= new Persona(”Antonio”,”222”);
        agenda.anadePersona(p);
        p= new Persona(”Manuel”,”333”);
        agenda.anadePersona(p);
        agenda.borraPersona(p);
        agenda.anadePersona(p);
    }catch(Exception e){
        System.out.printl(e.getMessage());
    }
    System.out.printl(agenda.muestraListaPersona());
  }
}
```

```
La persona con el DNI 333 ya está en la lista
Nombre: Juan DNI: 111
Nombre: Antonio DNI: 222
Nombre: Manuel DNI: 333
```

# La clase LinkedList

La clase LinkedList permite crear listas ordenadas con elementos repetidos. El método add() añade un elemento a la lista. Los métodos addFirst y addLast permiten añadir un elemento al principio o al final de la lista. El método size permite investigar cuantos datos hay en la lista. Para acceder a los datos hay que utilizar un objeto de la interface ListIterator. El código muestra cómo utilizar una lista ordenada y con repeticiones.

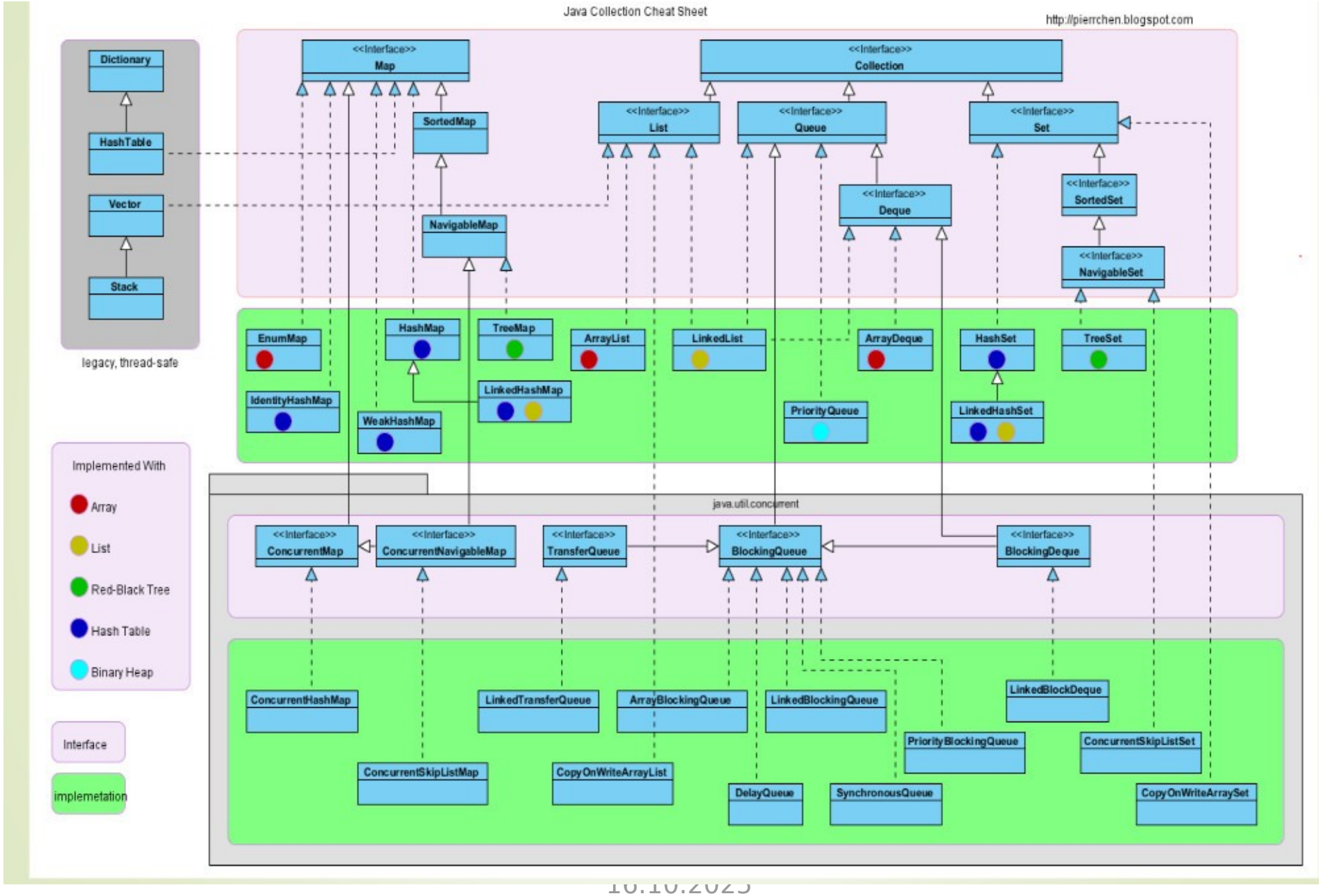
```
import java.util.*;
public class VerLista{
 public static void main(String args[]){
   LinkedList<String> lista = new LinkedList<String>();
   lista.add("Martes"); lista.add("Miercoles"); lista.add("Jueves");
    lista.add("Viernes"); lista.addFirst("Domingo");
    lista.add("Sabado"); lista.addLast("Sabado");
System.out.println("\nLa semana \"ideal\"");
ListIterator i = lista.listIterator (0);
while (i.hasNext ()) System.out.print(i.next().toString()+"  ");
}
```

# La clase TreeMap

La clase TreeMap permite crear listas ordenadas, sin repeticiones, mediante pares de claves y valores. El método put permite añadir un dato a la lista asociada a una clave. Para ver los datos hay que utilizar un objeto de la interface Iterator, para lo cual es necesario crear un objeto de la interface Collection, tal cómo se recoge en el código.

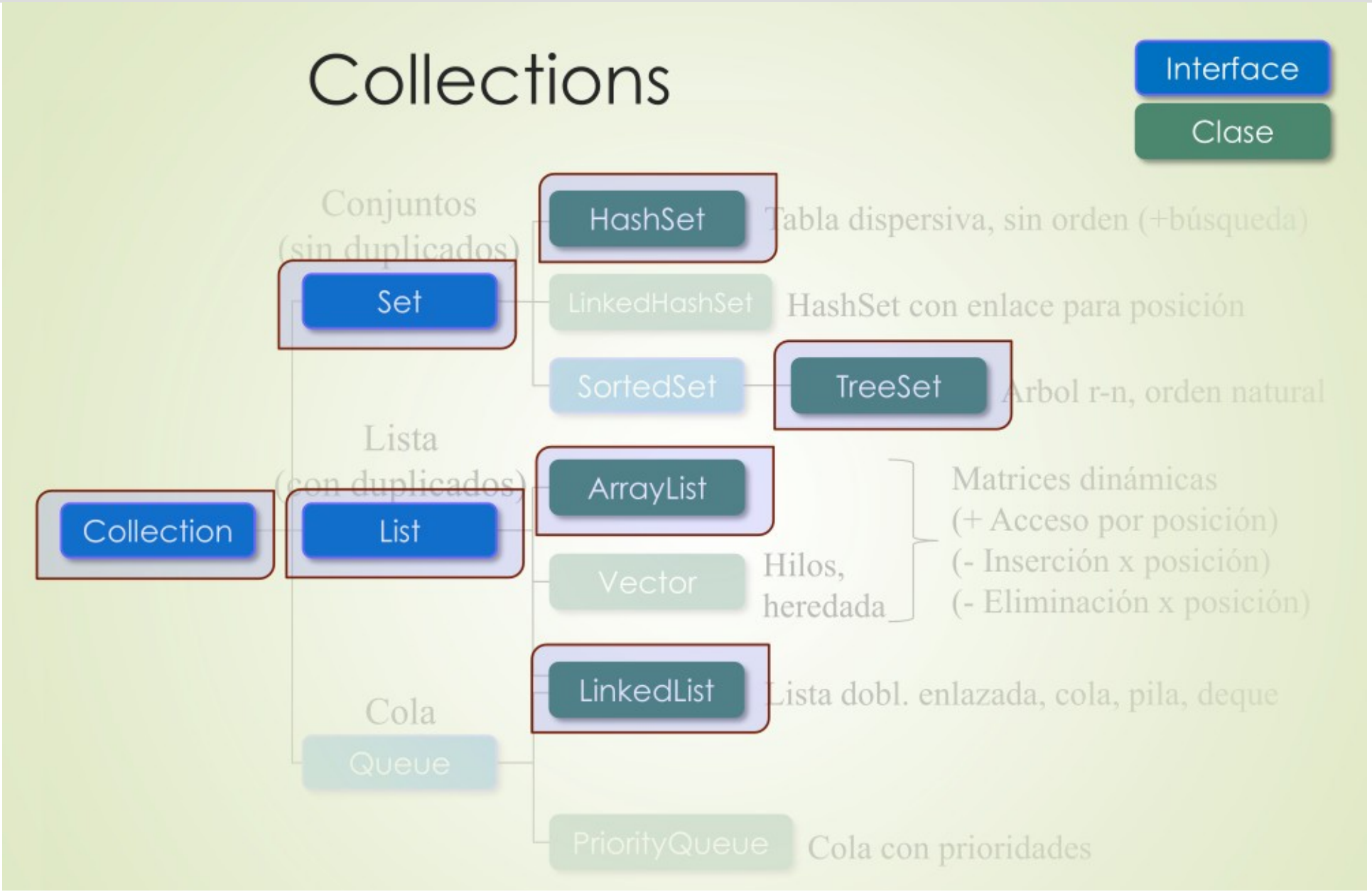
```
public class VerArbol{
 public static void main(String args[]){
  TreeMap<String,String> lista = new TreeMap<String,String>();
  lista.put("1","Lunes"); lista.put("2","Martes"); lista.put("5",
  "Viernes"); lista.put("4","Jueves"); lista.put ("7", "Domingo");
   lista.put ("6","Sábado"); lista.put("3,"Miércoles"); lista.put
   ("6","Sábado");
System.out.println(”La lista tiene:”);
System.out.println(lista.size()+" elementos");
System.out.println("La semana, ordenada y sin repeticiones”); Colletction
colección = lista.entrySet();
Iterator i= colección.iterator();
while (i.hasNext())
System.out.print(i.next().toString ()+" ”);
}}
```

La lista tiene 7 elementos La semana ordenada y sin repeticiones
1=Lunes
2=Martes
3=Miércoles
4=Jueves
5=Viernes
6=Sábado
7=Domingo


Java Collection Cheat Sheet
http://pierrchen.blogspot.com
Dictionary
HashTable
Vector
Stack
legacy, thread-safe
<<Interface>>
Map
SortedMap
NavigableMap
<<Interface>>
Collection
<<Interface>>
List
<<Interface>>
Queue
<<Interface>>
Set
<<Interface>>
Deque
<<Interface>>
SortedSet
<<Interface>>
NavigableSet
EnumMap
HashMap
TreeMap
ArrayList
LinkedList
ArrayDeque
HashSet
TreeSet
IdentityHashMap
WeakHashMap
LinkedHashMap
PriorityQueue
LinkedHashSet
Implemented With
Array
List
Red-Black Tree
Hash Table
Binary Heap
java.util.concurrent
<<Interface>>
ConcurrentMap
<<Interface>>
ConcurrentNavigableMap
<<Interface>>
TransferQueue
<<Interface>>
BlockingQueue
<<Interface>>
BlockingDeque
ConcurrentHashMap
LinkedTransferQueue
ArrayBlockingQueue
LinkedBlockingQueue
LinkedBlockDeque
PriorityBlockingQueue
ConcurrentSkipListSet
ConcurrentSkipListMap
CopyOnWriteArrayList
DelayQueue
SynchronousQueue
CopyOnWriteArraySet
Interface
implemetation


Collections
Interface
Clase
Collection
Conjuntos
(sin duplicados)
Set
HashSet
Tabla dispersiva, sin orden (+búsqueda)
LinkedHashSet
HashSet con enlace para posición
SortedSet
TreeSet
Arbol r-n, orden natural
Lista
(con duplicados)
List
ArrayList
Vector
Hilos,
heredada
Matrices dinámicas
(+ Acceso por posición)
(- Inserción x posición)
(- Eliminación x posición)
LinkedList
Lista dobl. enlazada, cola, pila, deque
Cola
Queue
PriorityQueue
Cola con prioridades

# ANÁLISIS Y PROGRAMACIÓN EN JAVA - IFCD004PO




16.10.2025

Enrique Krause Buedo


List
Interface
Clase
Conjuntos
(sin duplicados)
HashSet
Tabla dispersiva, sin orden (+búsqueda)
Set
LinkedHashSet
HashSet con enlace para posición
SortedSet
TreeSet
Arbol r-n, orden natural
Lista
(con duplicados)
ArrayList
Matrices dinámicas
(+ Acceso por posición)
(- Inserción x posición)
(- Eliminación x posición)
Collection
List
Vector
Hilos,
heredada
LinkedList
Lista dobl. enlazada, cola, pila, deque
Cola
Queue
PriorityQueue
Cola con prioridades

# Interface List (I)

Permite crear matrices dinámicas (Que pueden cambiar su numero de elementos). Clases que implementan la interface List: ArrayList, LinkedList, Vector…

| | |
|---|---|
| void add(Object obj) | Añade un objeto al final |
| Object get(int posicion) | Retorna el objeto que hay en la posicion indicada como parámetro. |
| int size() | Devuelve el número de elementos |
| boolean remove (Object obj)<br>Object remove (int indice) | Elimina el primer objeto con la referencia suministrada o en la posición indicada por el indice. Si no encuentra el objeto retorna false. Si se indica el indice se retorna el objeto eliminado de la lista. |
| void clear () | Quita todos los elementos de la colección. |
| boolean isEmpty() | Devuelve true si no tiene elementos |
| boolean contains(Object obj) | Retorna true si el objeto suminstrado está en la lista. |
| int indexOf(Object obj) | Retorna la posición donde está el objeto. -1 si no encontrado. |

# Interface List (II)

| | |
|---|---|
| Iterator iterator() | Retorna un Iterator con todos los elementos de la lista. |
| Object set(int indice,Object obj) | Remplaza el objeto que hay en la posición indicada por indice con el objeto suministrado. Retornando una referencia al objeto sustituido. |
| T[] toArray(T[] a) | Retorna una matriz con todos los objetos de la lista, pero la matriz es del mismo tipo de la matriz pasada como parámetro. |
| Object [] toArray() | Retorna una matriz de Object con todos los elementos del la lista. |
| boolean addAll(Collection<E> c) | Agrega todos los elementos de c, en el orden que determina el Iterator |
| void add(int index, Object obj) | Inserte un objeto en la posición indicada, incrementando la posición de los objetos que hay detrás. |

# Clase ArrayList

Implementa List. Permite crear matrices dinámicas (Que pueden cambiar su numero de elementos). Como implementa la interface List solamente se indican los métodos adicionales.

| | |
|---|---|
| **ArrayList**() | Lista vacía con capacidad 10 |
| **ArrayList**(Collection <E> c) | Lista con los elementos que contiene c (orden Iterator) |
| **ArrayList**(int capacidadInicial) | Lista vacía con capacidad capacidadInicial |

| | |
|---|---|
| Object **trimToSize**() | Ajusta la capacidad al tamaño actual (size) |

# Ejemplo uso ArrayList

```
ArrayList<String> al = new ArrayList<String>();
al.add("Miercoles");
al.add("Lunes");
al.add("Martes");

System.out.println(al.get(0));

System.out.println("El elemento 2 es: " + al.get(2));

for (int i = 0; i < al.size(); i++) {
        System.out.println(al.get(i));
}

System.out.println(al.size());
```

# Ejemplo uso List

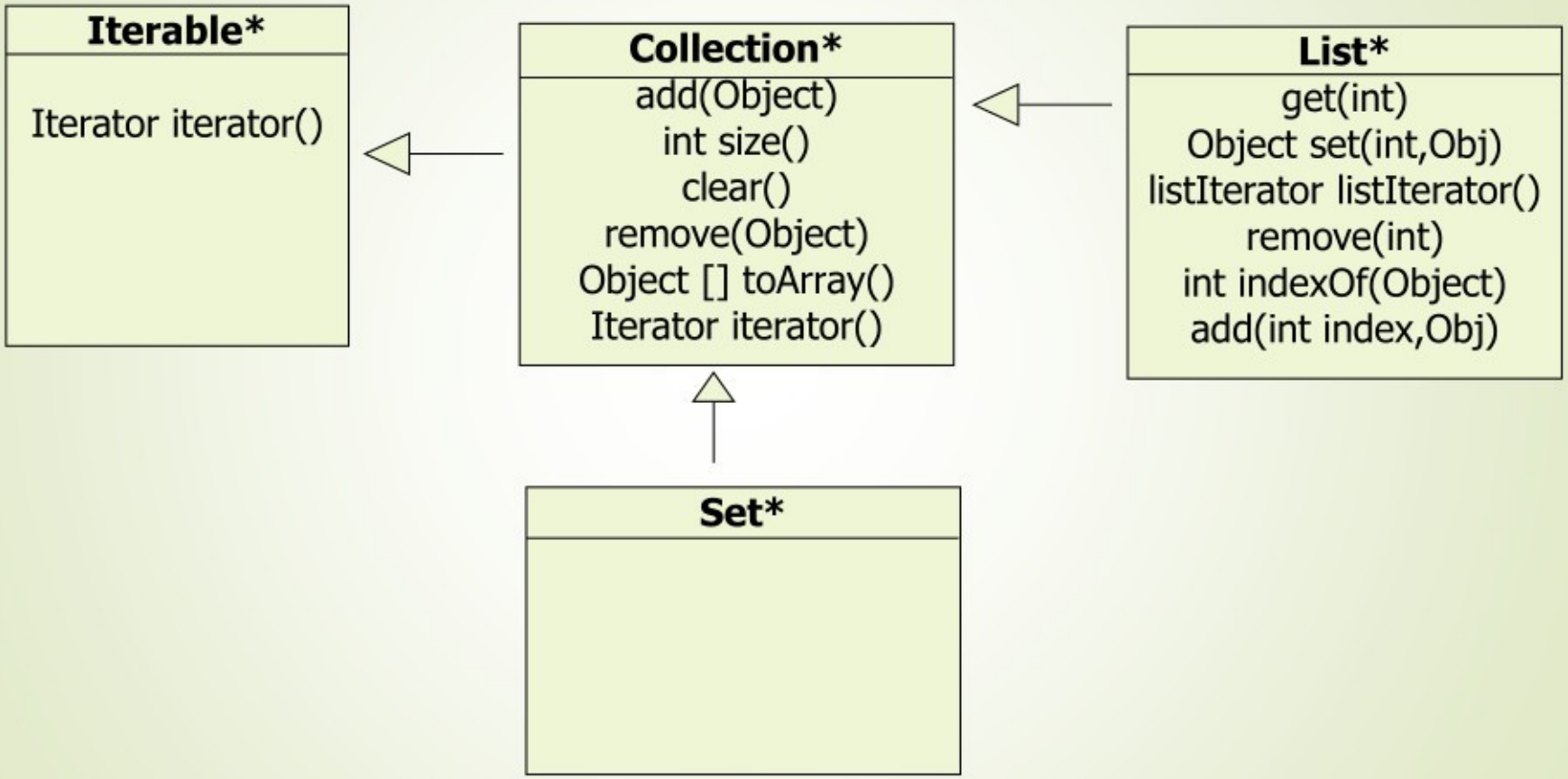
```
List<String> al = new ArrayList<String>();
al.add("Miercoles");
al.add("Lunes");
al.add("Martes");

System.out.println(al.get(0));

System.out.println("El elemento 2 es: " + al.get(2));

for (int i = 0; i < al.size(); i++) {
        System.out.println(al.get(i));
}

System.out.println(al.size());
```

# Iterator

Proporcionan métodos para recorrer secuencialmente los elementos de una colección de objetos.

| **Iterator** | |
|---|---|
| boolean hasNext() | Retorna true si hay más elementos a continuación. |
| Object next() | Retorna una referencia al siguiente objeto de la colección a la que está conectado el Iterator |
| void remove() | Elimina, de la colección a la que está conectado, el elemento que se acaba de acceder. |

# Collection, Set e Iterable



# Iteradores (Iterator)

boolean .hasNext()
Object .next()
Object .remove()

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

while (it.hasNext()) {
   Object dato = it.next();
   ...trabajar con dato...
}
```

# Ejemplo uso List (Iterator)

```
List<String> al = new ArrayList<String>();
al.add("Miercoles");
al.add("Lunes");
al.add("Martes");

System.out.println(al.get(0));

System.out.println("El elemento 2 es: " + al.get(2));

Iterator<String> it = al.iterator();
while (it.hasNext()) {
   System.out.println(it.next());
}

System.out.println(al.size());
```

# for each (o for extendido)

```
for (tipo varTemp : colección){
        varTem.XXX
}
```

```
ArrayList<Figura> lista = new ...;
Iterator it = lista.iterator();

while (it.hasNext()) {
   Figura f = it.next();
   f.area();
   ...
}
```

```
ArrayList<Figura> lista = new ...;

for (Figura f : lista) {
   f.area();
   ...
}
```

# Ejemplo uso List (foreach)

```
List<String> al = new ArrayList<String>();
al.add("Miercoles");
al.add("Lunes");
al.add("Martes");

System.out.println(al.get(0));

System.out.println("El elemento 2 es: " + al.get(2));

for (String actual : al) {
    System.out.println(actual);
}

System.out.println(al.size());
```

# Clase LinkedList

Almacena objetos en una lista doblemente enlazada.
Implementa la interface List.

| | |
|---|---|
| void **add**(int indice, Object elem) | Inserta un elemento en la posición indicada por el índice. Los elementos que había en esa posición y siguientes se desplazan una posición. |
| boolean **add**(Object elem)<br>void **addLast**(Object elem) | Inserta un elemento al final de la lista. |
| void **addFirst**(Object elem) | Inserta un elemento al principio de la lista. |
| Object **set**(int indice, Object elem) | Sustituye el elemento ubicado en la posición indicada por el índice con el objeto suministrado. Retorna una referencia al objeto sustituido. |
| Object[] **toArray**() | Retorna una matriz con los objetos de la lista. |
| Object **remove**(int index)<br>boolean **remove**(Object elem) | Elimina un elemento de la lista, bien por su indice o por la referencia al objeto. |
| ListIterator **listIterator**(int index) | Retorna un ListIterator enlazado a los elementos almacenados a partir del elemento con índice indicado. ListIterator es un Iterator que además tiene métodos para retroceder (hasPrevious() y previous()) |

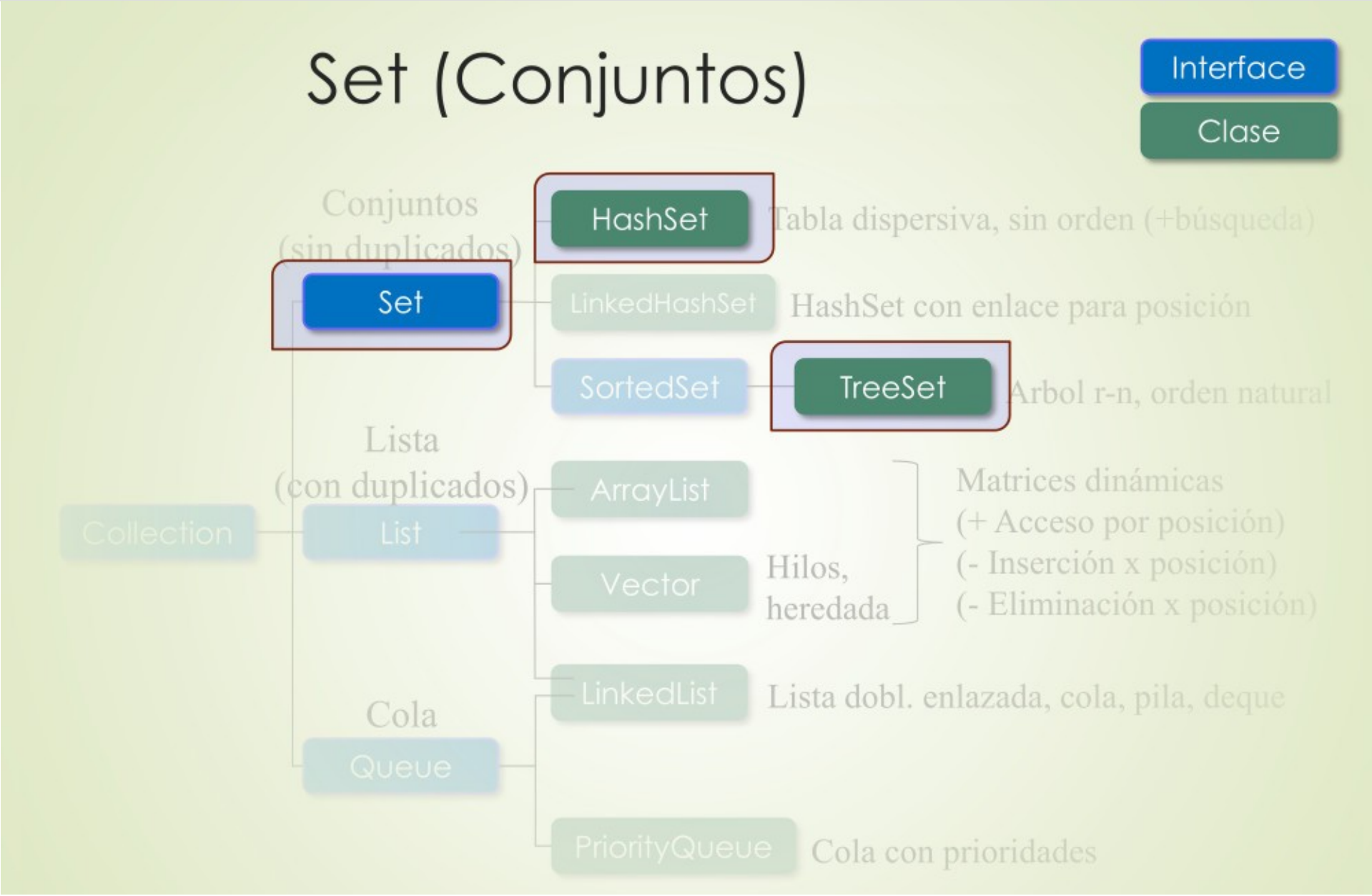
# Clase Stack

Los objetos almacenados se rigen por la filosofía de una pila que sigue la regla: "El último en entrar es el primero en salir".

| | |
|---|---|
| Object **push**(Object item) | Inserta un elemento en la pila.(Retorna la referencia insertada) |
| Object **pop**() | Extrae el último elemento insertado en la pila. |
| Object **peek**() | Retorna el último elemento de la pila, sin sacarlo de esta. |
| boolean **empty**() | Indica si la pila está vacia.(Hace lo mismo que isEmpty, heredado de Vector) |
| int **size**() | Retorna el número de elementos de la pila. |

La clase Stack hereda de Vector, por lo tanto, tiene además de los métodos indicados, los métodos de Vector y de List.


Collections
Interface
Clase
Conjuntos
(sin duplicados)
HashSet
Tabla dispersiva, sin orden (+búsqueda)
Set
LinkedHashSet
HashSet con enlace para posición
SortedSet
TreeSet
Arbol r-n, orden natural
Lista
(con duplicados)
ArrayList
Matrices dinámicas
(+ Acceso por posición)
(- Inserción x posición)
(- Eliminación x posición)
Collection
List
Vector
Hilos,
heredada
LinkedList
Lista dobl. enlazada, cola, pila, deque
Cola
Queue
PriorityQueue
Cola con prioridades




16.10.2025

Enrique Krause Buedo

# Interfaces Set

Set es un conjunto. Set no tiene elementos duplicados. Los métodos son los de Collection

| | |
|---|---|
| Object **add**(Object valor) | Añade un objeto al final |
| boolean **contains**(Object valor) | Indica si contiene o no un objeto |
| boolean **isEmpty**() | Devuelve true si no tiene elementos |
| Iterator **iterator()** | Retorna un iterator para recorrer los objetos que contiene. |
| Object **[] toArray()** | Retorna una matriz con todos los objetos que contiene. |

# Clase HashSet

Almacena objetos, **sin importar el orden** y sin posibilidad de repetir elementos.

- Conjuntos Dispersivos
- Implementa conjuntos mediante Tablas Dispersivas
- Utiliza en método hashCode

| | |
|---|---|
| **HashSet**() | Crea un conjunto |
| **HashSet**(Collection <E> c) | Conjunto con los elementos que contiene c |
| **HashSet**(int capacidadInicial) | Conjunto vacío con una tabla de tamaño capacidadInicial |
| **HashSet**(int capacidadInicial, float factorCarga) | Conjunto vacío con una tabla de tamaño capacidadInicial y factor de Carga* |

* Factor de Carga: valor entre 0.0 y 1.0 que determina el % de carga para realizar la dispersión

# Clase TreeSet

Almacena objetos, **ordenados (orden natural)**.

- Conjunto Ordenado
- Implementa conjuntos mediante árbol rojo-negro
- Extrae los elementos ordenados
- Menos eficiente en búsqueda que Hash pero mucho más que List
- Los objetos que almacena deben implementar la interfaz Comparable (método compareTo( T otro)**

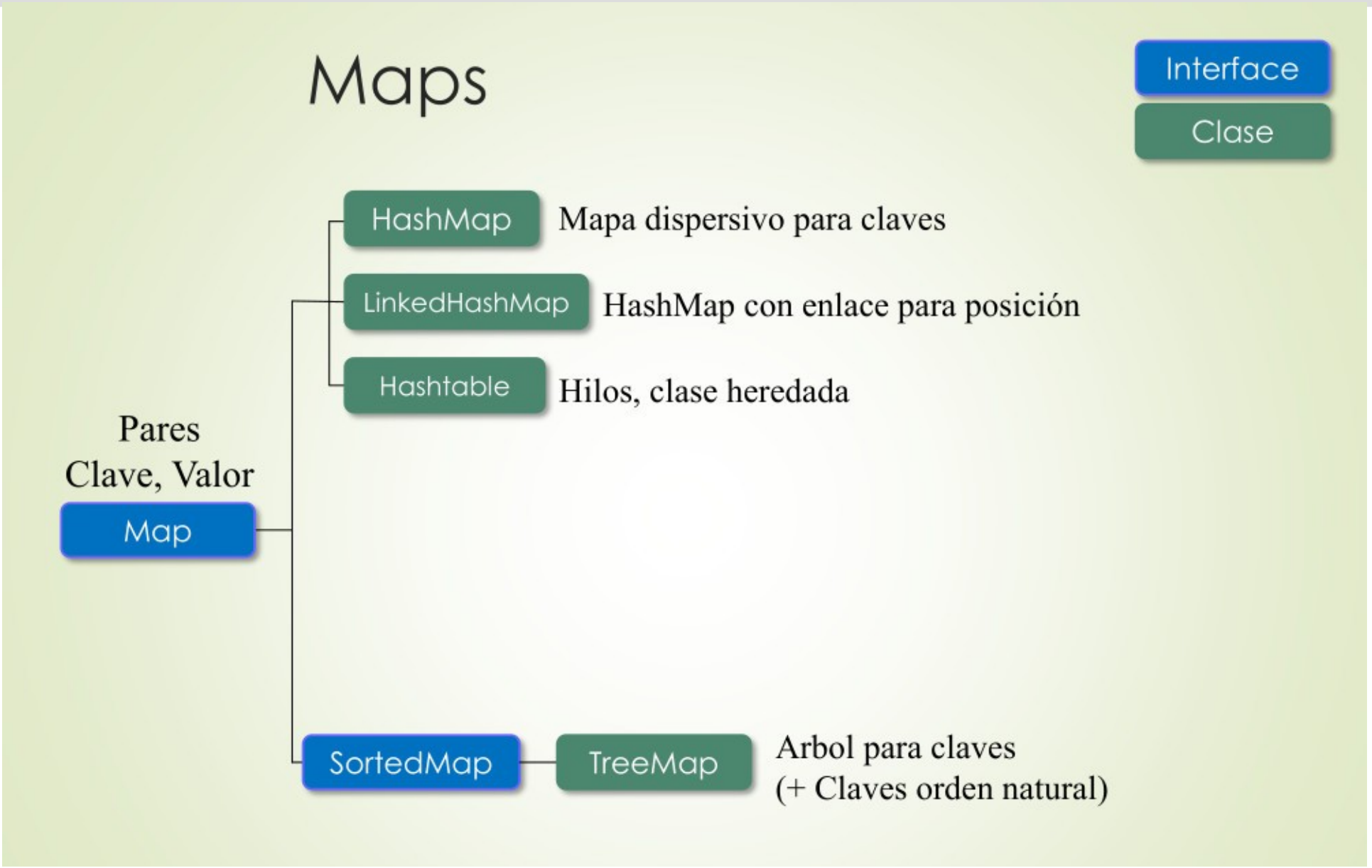
| | |
|---|---|
| **TreeSet**() | Crea un conjunto |
| **TreeSet**(Collection <E> c) | Conjunto con los elementos que contiene c |
| **TreeSet**(Comparator <E> comp) | Conjunto que se ordena de acuerdo a comp* |

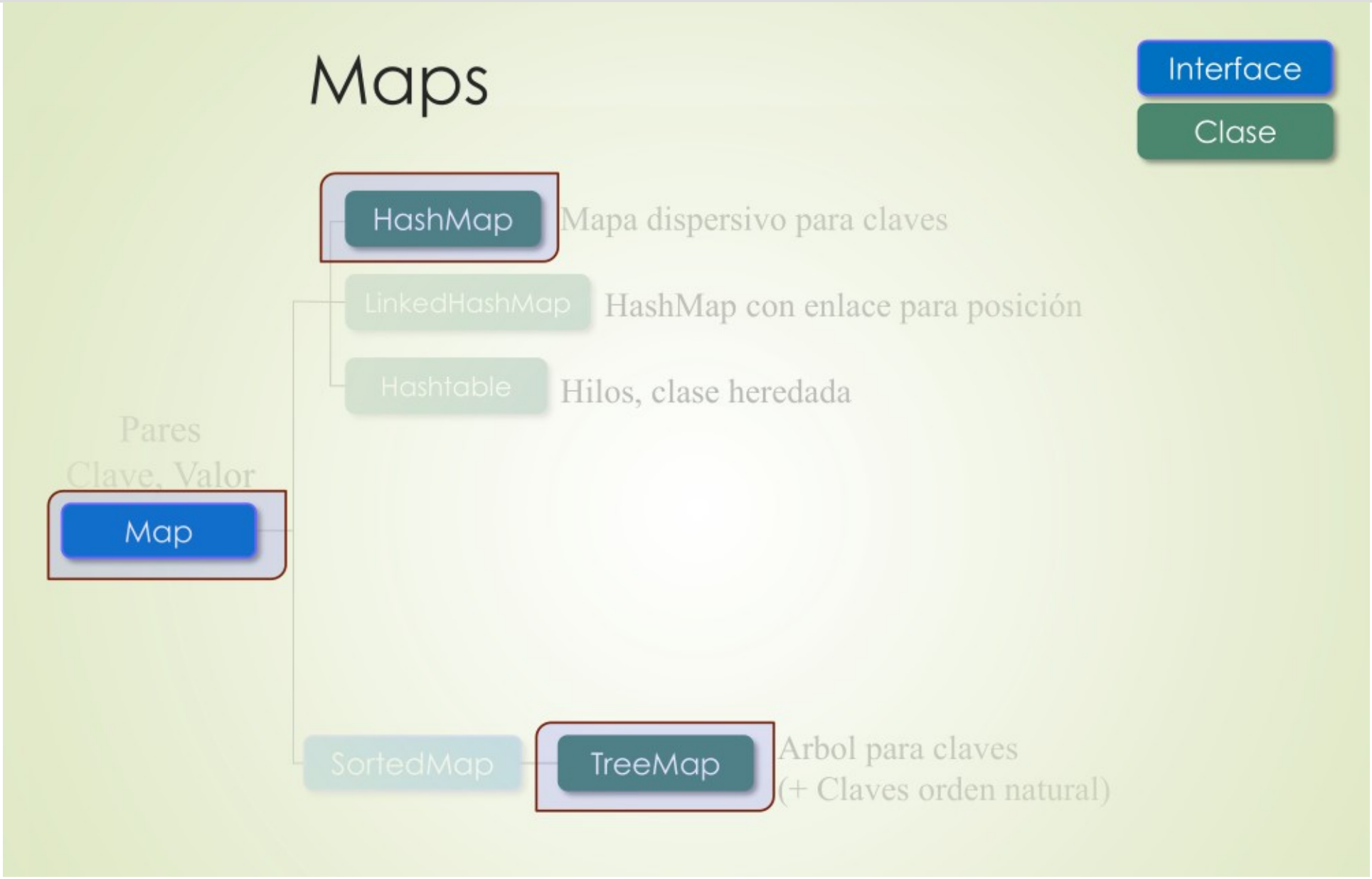
* Ver Interfaz Comparator

# Interfaz Comparator

- Un solo método abstracto, el resto son static o default (1.8)
- Se suele crear una clase anónima

```
Set<Empleado> emp = new TreeSet<>(new Comparator<Empleado>() {
    public int compare(Empleado e1, Empleado e2) {
        return e1.nombre.compareTo(e2.nombre);
    }
});
…

class Empleado{
   int id;
   String nombre;
    …
```


Maps
Interface
Clase
HashMap
Mapa dispersivo para claves
LinkedHashMap
HashMap con enlace para posición
Hashtable
Hilos, clase heredada
Pares
Clave, Valor
Map
SortedMap
TreeMap
Arbol para claves
(+ Claves orden natural)




16.10.2025

Enrique Krause Buedo

# Clase TreeMap

Almacena objetos, asociándole a cada uno una clave, los elementos quedarán ordenados según la clave. (No puede haber repeticiones, es decir dos elementos con la misma clave).

| | |
|---|---|
| Object **put**(Object clave, Object elem) | Inserta un elemento (elem) asociandole una clave. Retorna el objeto previamente asociado a la clave (Si no existía retorna null) |
| Collection **values**() | Retorna una Collection con todos los objetos insertados en el mapa. |
| Set **keySet()** | Retorna un Set con las claves del TreeMap (Nota: Set hereda de Collection ). |
| Object **get**(Object clave) | Retorna el objeto asociado a la clave indicada. |

| | |
|---|---|
| TreeMap() | Constructor sin Parámetros crea un TreeMap cuyos objetos quedarán ordenados por su orden natural, es decir, su método compareTo. |
| TreeMap(Comparator c) | Este constructor permite suministrarle un objeto que implemente la interface Comparator para indicar le criterio de ordenación. Habrá que sobreescribir el método int compare(Object clave1, Object clave2) de Comparator. De modo que retornará negativo si la primera clave es menor, positivo si es mayor y cero si son iguales. |