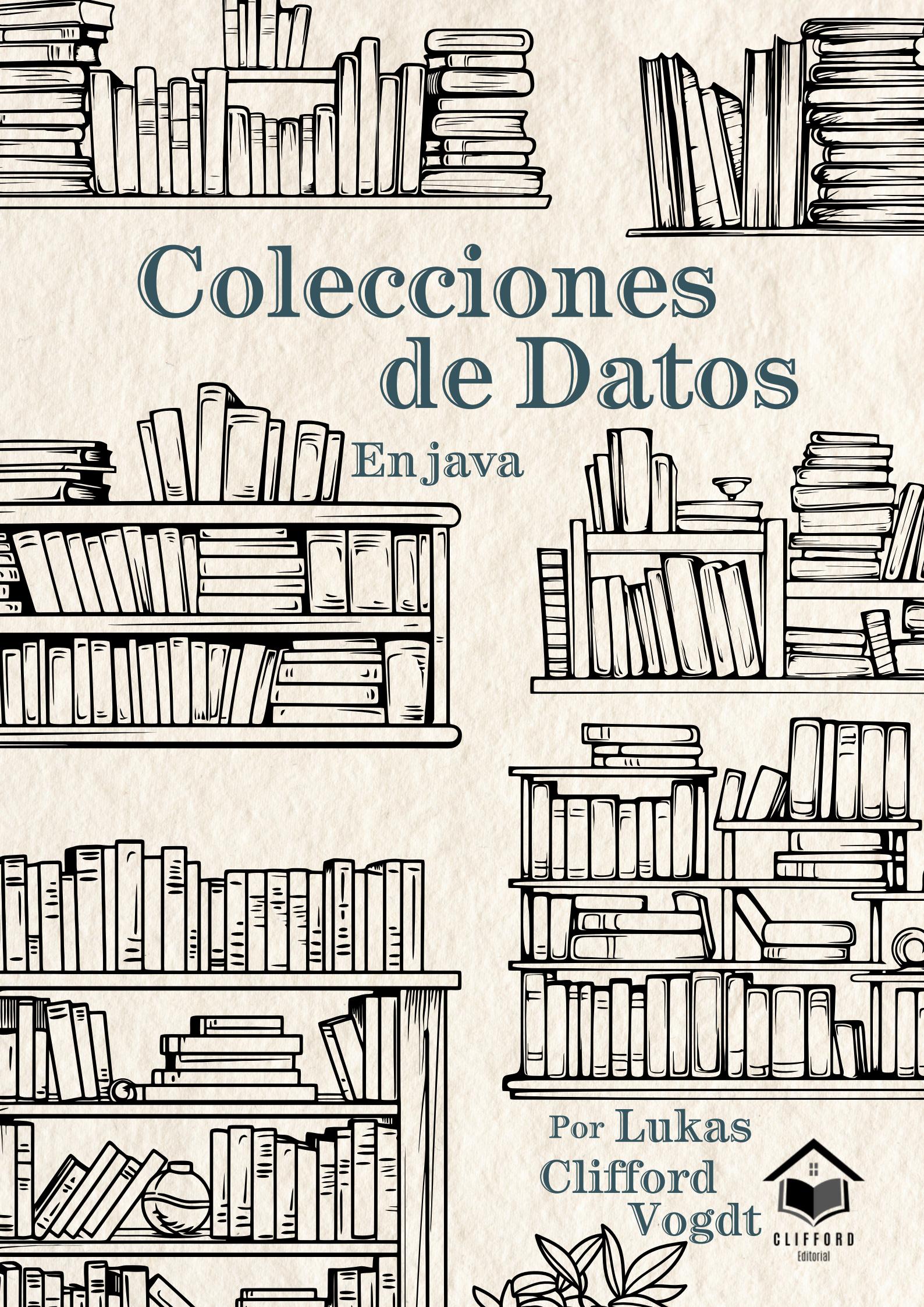


Colecciones de Datos

En java

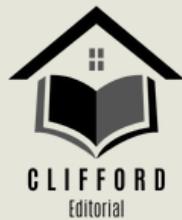


Por Lukas
Clifford
Vogdt



Colecciones de Datos en Java

Por Lukas C. Vogdt Torralba



©Lukas Clifford Vogdt, 2024
©Editorial Clifford Editorial

Primera edición: Diciembre de 2024
Director editorial: Lukas Clifford
Edición: Clifford Vogdt

ISBN: 978-3-16-148410-0

Clifford Editorial
Calle Java 25 +11 , Piso edificio.length - 3**
Ciudad IDE, País Computerlandia.

info@cliffordeditorial.com
www.cliffordeditorial.com

**Impreso por: IMPRIMIMOSTODO S.A. Polígono Industrial Medias Tintas,
Nave 42 Ciudad Canon, País DondeSaleMasBaratoProducirLandia**

**Queda prohibida la reproducción total o parcial de esta obra, su registro o transmisión por cualquier medio, ya sea electrónico, mecánico, biológico, aéreo, bajo el agua, verbal, u otros sistemas, sin la autorización previa y por escrito de la Editorial Clifford Editorial.
Cualquier violación a estos derechos estará sujeta a las sanciones establecidas por las leyes aplicables, supongo.**

Impreso en España - Printed in Spain

INTRODUCCIÓN	6
UN POCO DE HISTORIA	7
Árbol familiar de la familia Collection	9
Árbol familiar de la familia Map	10
Esquema Collection Framework	11
ITERATOR	12
Ejemplo 01	15
ListIterator	16
Ejemplo 02	18
ARRAYLIST	20
Métodos de gestión de valores	21
Ejemplo 03	22
Otros métodos	25
Ejemplo 04	26
Ejemplo 05	28
Ejemplo 06	30
HASHMAP	32
Métodos de gestión de valores	33
Ejemplo 07	33
Otros métodos	35
Ejemplo 08	37
Ejemplo 09	38
CONCLUSIÓN	41
¿De dónde viene la información?	41
Agradecimientos	42

INTRODUCCIÓN

Sobre los datos existe una frase popular “Un dato por sí mismo no constituye información, es el procesamiento de los datos lo que nos proporciona información”. Nosotros, los programadores, nos encanta procesar datos con cientos de variables, organizadas en arrays, listas, tuplas, mapas, sets, pilas... Para abreviar, colecciones. En esto se enfoca este libro, las colecciones de datos, al enfoque del lenguaje de programación Java.

UN POCO DE HISTORIA

En los inicios de la programación Java (JDK 1.0...), los métodos que existían para agrupar objetos eran los Arrays, Vectors y HashTables. La implementación de estas colecciones fue individual, sin relación entre ellas. Esto provocó que los métodos y constructores fueran diferentes para cada colección, haciendo la vida difícil a los programadores. Por ejemplo:

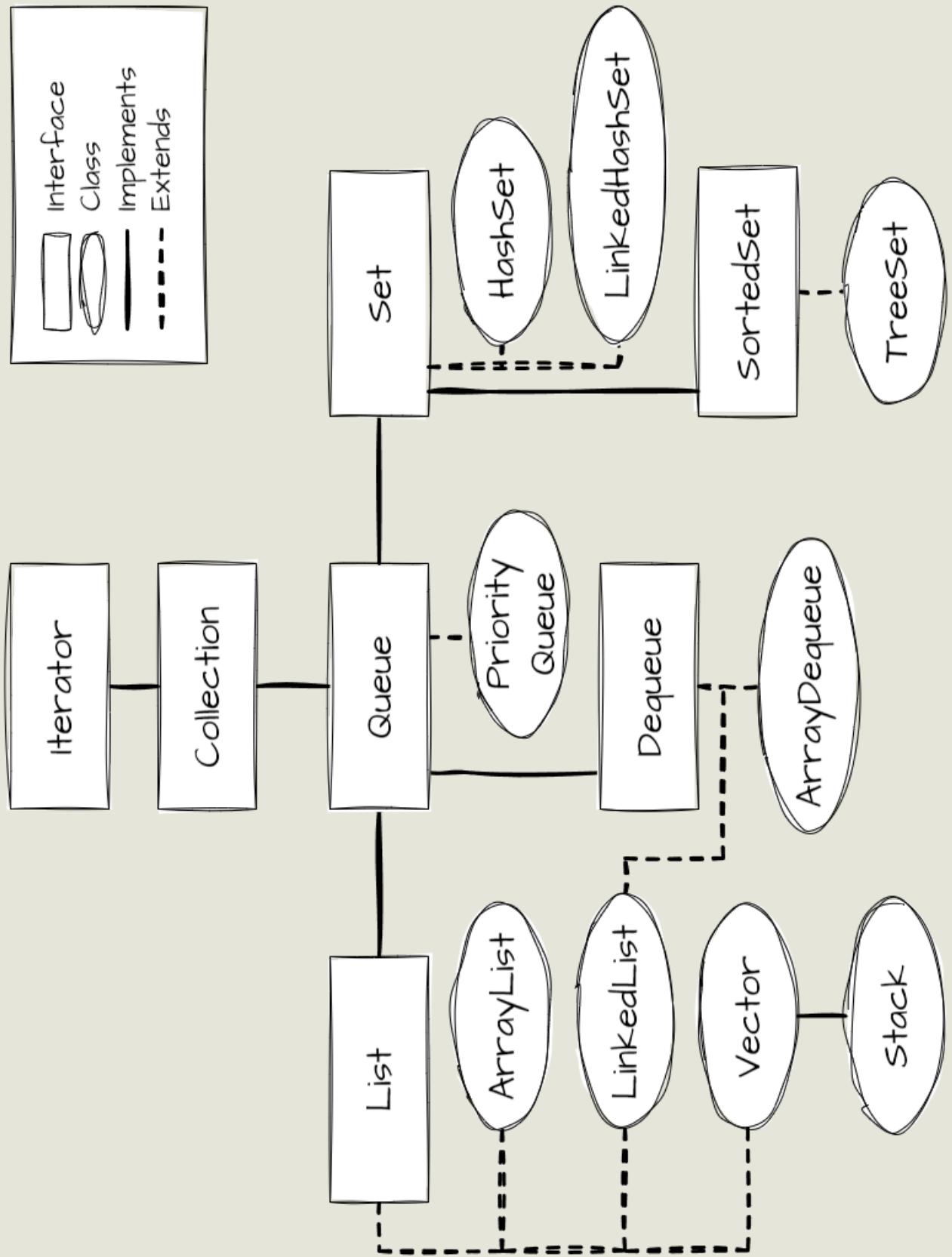
```
vector.addElement(1);  
hashtable.put(1,"uno");  
  
vector.elementAt(0);  
hashtable.get(0);
```

Esta disparidad entre *addElement* y *put* o entre *elementAt* y *get* pedía a gritos una reforma de las colecciones. Es aquí donde entra Joshua J. Boch, principal desarrollador del *Collection Framework*. El neoyorquino, gran conocido en el mundo de Java, fue también desarrollador de la librería `java.math`. Trabajó en *Sun Microsystems* como importante ingeniero para después trabajar en el

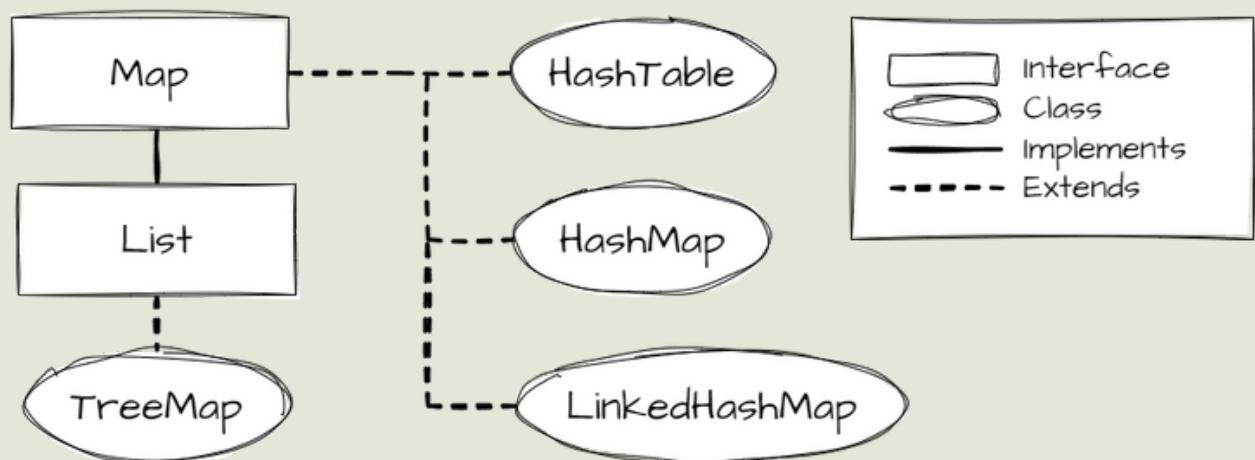
desarrollo de Java en *Google*. Además de escritor de código, también escribió libros conocidos como *Efficient Java*, siendo la más nueva, la tercera edición; y coescritor de *Java Puzzlers*.

Volviendo al Collection Framework... Bueno, ¿y qué es un *framework*? se preguntará alguno. Pues es un conjunto de clases e interfaces que proveen una arquitectura base. Hemos hablado de como no estaban relacionadas las antiguas colecciones (llamadas *Legacy collections*). El Collection Framework, CF por si lo tengo que volver a mencionar, se encarga de estandarizar y ofrecer conexiones entre los tipos de colecciones, y pues claro, añade una gran variedad de tipos de colecciones si cabe decir.

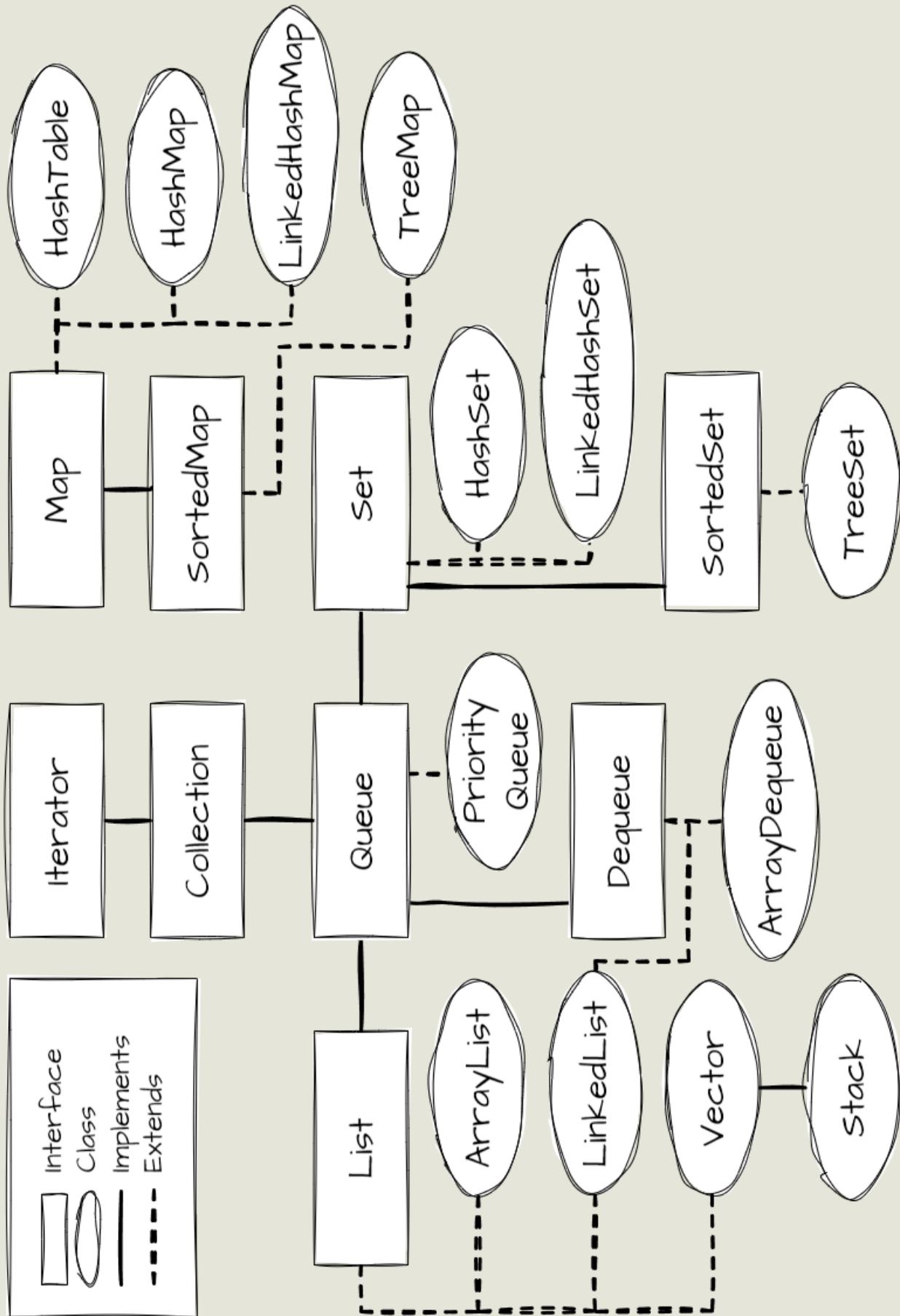
En el CF podemos decir que hay dos familias, una más grande que otra. Empezaremos por la grande, la familia *Collection*. La interfaz del mismo nombre se podría decir que es el padre de todos pero, debemos de tener en cuenta al abuelo *Iterator* que proporciona a toda la familia la posibilidad de utilizar iterators. Más adelante veremos cómo funcionan. De la interfaz *Collection* descienden *List*, *Queue* y *Set*. Cada una tiene características que las distinguen, como que en los sets no pueden repetirse elementos o el establecimiento de prioridades de queue. Véase la imagen esquema para entender la herencia de la familia.



Ahora, tenemos por otro lado la familia *Map*. Esta no es tan grande, se podría comparar a las subfamilias *List* o *Set*. La diferencia está en cómo funcionan, los hijos de *Collection* tratan de conjuntos de un tipo de valor mientras que *Map* posee dos tipos de valores. Piense de ello como un diccionario convencional en el que tiene un valor que es la palabra y otro que es el significado (De hecho, java tiene una clase *Dictionary* pero está en desuso por la creación de *Map*). Por lo tanto, el señor Boch decidió no incluir a *Map* en la herencia porque funcionan de un modo diferente que no encajaría con los métodos de las demás interfaces. Sin embargo, quiero hacer hincapié en que está fuera de la herencia pero no fuera del framework ya que tiene métodos que se relacionan con las demás colecciones. Por ejemplo, los métodos *keySet*, que devuelve un *Set* o *values*, que devuelve un *Collection*. Véase la imagen esquema para entender la familia *Map*.



Esquema del Collection Framework



ITERATOR

Los *Iterators* se utilizan para conseguir elementos uno a uno en cualquier objeto perteneciente a la familia *Collection*. Antiguamente se utilizaba *Enumeration* para leer elementos en las *Legacy Collections* en JDK 1.0. La diferencia está en que los iterators de CF pueden tanto leer como eliminar elementos. Veamos rápidamente los tres métodos que tiene:

- ***hasNext***: Comprueba si la siguiente posición tiene un elemento. En caso afirmativo devolverá *true* y en caso negativo devolverá *false*.
- ***next***: Pone el cursor en la siguiente posición.
- ***remove***: Elimina el elemento en la posición del cursor.

No es de esperar que el lector entienda solo con los métodos como funciona un iterator, así que vamos a ver cómo funcionan exactamente.

Dada una colección, debe imaginar al iterator como un cursor o un puntero que apunta a un elemento. El puntero sirve para leer o eliminar elementos en su posición. Este tiene los métodos mencionados anteriormente. Este es el procedimiento que sigue:

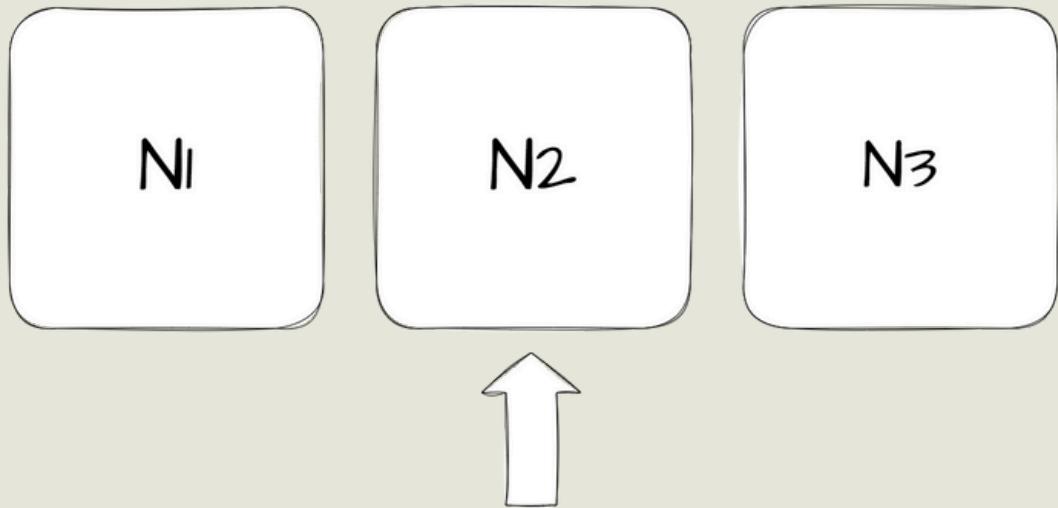
El cursor se sitúa justo antes del primer elemento, como si estuviera en la posición teórica “-1”. Con hasNext se comprueba que haya un elemento delante y luego con next se mueve el cursor una posición.



El cursor se sitúa en el primer elemento, así podremos interactuar con él. Comprobamos con hasNext y seguimos con next.



El cursor se sitúa en el segundo elemento, así podremos interactuar con él. Comprobamos con hasNext y seguimos con next.



El cursor se sitúa en el último elemento. Al comprobar con hasNext, este nos devuelve false, indicando que la colección acaba ahí.



EJEMPLO 01

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

public class Main {
    public static void main(String[] args) {

        //Definición del array
        ArrayList<Integer> numeros = new ArrayList<Integer>
        (List.of(1,2,3));
        System.out.println("Numeros antes de la
modificación: " + numeros + "\n");

        //Definición del iterator
Iterator<Integer> iterator = numeros.iterator();

        //Se comprueba si existe un elemento después de la
        //posición del iterator
        while(iterator.hasNext()) {

            //Se obtiene el número en la posición del
            //iterator
            Integer numero = iterator.next();
            System.out.println("Número: " + numero);

            if(numero == 2) {
                System.out.println("Eliminando número
2...");
                iterator.remove();}
            }
        }

        System.out.println("\nNúmeros después de la
modificación: " + numeros + "\n");
    }
}
```

SALIDA

Números antes de la modificación: [1, 2, 3]

Número: 1

Número: 2

Eliminando número 2...

Número: 3

Números después de la modificación: [1, 3]

Los iterators traen consigo ventajas como su sencillez o integración en todas las colecciones. En adición a eso, brindan eficiencia para recorrer una gran colección y seguridad al eliminar elementos. Sin embargo, si el lector es dado a la observación, se habrá dado cuenta de un inconveniente importante del iterator: es Unidireccional. Solo tiene métodos para ir hacia adelante pero no hacia atrás. Además, solo permite la eliminación de elementos, no su modificación ni adicción. Esto es solucionado por un hijo suyo, *ListIterator*.

ListIterator

Es un tipo de iterator que es solo aplicable a listas (*List*). Tiene más métodos que *Iterator* y es bidireccional. La lista de métodos es la siguiente:

- ***hasNext***: Comprueba si la siguiente posición tiene un elemento. En caso afirmativo devolverá *true* y en caso negativo devolverá *false*. Funciona igual que el de *Iterator*.

- ***next***: Pone el cursor en la siguiente posición. Funciona igual que el de *Iterator*.
- ***nextIndex***: Devuelve el índice del elemento siguiente.
- ***hasPrevious***: Comprueba si la anterior posición tiene un elemento. En caso afirmativo devolverá *true* y en caso negativo devolverá *false*. Funciona al contrario que *hasNext*.
- ***previous***: Pone el cursor en la posición anterior. Funciona al contrario que *next*.
- ***previousIndex***: Devuelve el índice del elemento anterior. Funciona al contrario que *nextIndex*.
- ***remove***: Elimina el elemento en la posición del cursor.
- ***set***: Reemplaza el elemento en la posición del cursor.
- ***add***: Inserta un elemento en la posición del cursor.

Veamos un ejemplo de *ListIterator*.

EJEMPLO 02

```
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

public class Main {
    public static void main(String[] args) {

        //Definición del array
        ArrayList<Integer> numeros = new ArrayList<Integer>
        (List.of(1,2,3));
        System.out.println("Numeros antes de la
modificación: " + numeros + "\n");

        //Definición del iterator
        ListIterator<Integer> listIterator =
        numeros.listIterator();

        //Se comprueba si existe un elemento después de la
        posición del iterator
        while(listIterator.hasNext()) {

            //Se obtiene el número de la posición
            Integer numero = listIterator.next();
            System.out.println("Cursor en número: " +
numero);

            //Uso del método set
            if(numero == 2) {
                System.out.println("Reemplazando el número
                2 por el número 98...");
                listIterator.set(99);
            }
        }
    }
}
```

```
//Uso del método add
if(numero == 3) {
    System.out.println("Añadiendo el valor
99...");
    listIterator.add(120);
}
System.out.println("\nNúmeros después de la
modificación: " + numeros + "\n");
}
```

SALIDA

Números antes de la modificación: [1, 2, 3]

Cursor en número: 1

Cursor en número: 2

Reemplazando el número 2 por el número 98...

Cursor en número: 3

Añadiendo el valor 99...

Números después de la modificación: [1, 99, 3,
120]

ARRAYLIST

Como se prometió en la introducción, vamos a ver cómo se utilizan los *ArrayList* y los *HashMap*. *ArrayList* es una clase hija de *List*, hija de *Collection*, hijo de *Iterator* y *heredero del trono de Gondor...* Volviendo al tema, las listas son un conjunto de elementos del mismo tipo que, al contrario que los arrays, no son estáticos, es decir, se pueden añadir y quitar elementos cambiando el tamaño del conjunto. Además las listas poseen métodos útiles para manipularlos.

Los *ArrayList* se definen de la siguiente manera:

```
ArrayList<TipoDeValor> nombreDeLaLista = new  
    ArrayList<TipoDeValor>();
```

Aquí podemos ver claramente las partes que conforman una lista. Por un lado, tenemos *<TipoDeValor>* con el que se define que tipo de datos va a contener la lista. Por otro lado, tenemos *nombreDeLaLista* al que se le hará referencia la lista. Por favor, los nombres de la lista deben tener sentido, no llame a una lista por lo que no es. Evite usar palabras como array, puesto que las listas y las array son elementos diferentes (tampoco llame a un array como lista).

La anterior instrucción enseña la definición de una lista vacía, lo cual, pues no es muy divertido que digamos. Podemos definir una lista usando `List.of(n1,n2,n3,...,nN)` dentro del paréntesis de la definición (no se preocupe, más adelante habrá ejemplos). Además, podemos añadir, modificar, obtener y eliminar elementos con métodos propios de las listas.

Métodos de gestión de valores

- ***Get***: Obtiene el valor de la lista indicando el índice de la posición (obviamente empezando desde el 0).
- ***Set***: Establece el valor de la lista en la posición indicada el índice de posición (empezando desde el 0 como puede imaginar).
- ***Add***: Añade el valor al final de la lista. Además el método *add* también acepta una segunda forma con dos parámetros, *add(indice, valor)*, con la que se pueden insertar valores en posiciones específicas. Nótese que se ha utilizado el verbo insertar puesto que no sobreescribe el valor en la posición. Se añade el valor y mueve todos los elementos hacia la derecha.
- ***Remove***: Elimina el valor de la lista en la posición indicada el índice de posición (empezando desde el 0...).

EJEMPLO 03

En el ejemplo 03 podrá ver que se ha definido una ArrayList con los elementos 1,2,3, utilizando List.of(). Además, ArrayList es una clase que mediante herencia obtiene el método `toString()`, que no es necesario el uso en este caso pero se utiliza para la demostración. Se muestran ejemplos de los métodos `get`, `set`, `add` y `remove`.

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {

        //Definición del array
        ArrayList<Integer> numeros =
            new ArrayList<Integer>(List.of(1,2,3)) ;

        System.out.println("Numeros: " +
            numeros.toString() + "\n");

        //Método get
        int primerNumero = numeros.get(0) ;
        System.out.println("Primer numero: " +
            primerNumero ) ;

        int segundoNumero = numeros.get(1) ;
        System.out.println("Segundo numero: " +
            segundoNumero) ;

        int tercerNumero = numeros.get(2) ;
        System.out.println("Tercer número: " +
            tercerNumero + "\n") ;
    }
}
```

```

//Método set
System.out.println("Sobreescribiendo el primer
número a -9...");

numeros.set(0, -9);
System.out.println("Numeros: " + numeros +"\n");

//Método add
System.out.println("Añadiendo el valor 4, 5, 6
al final de la lista...");

numeros.add(4);
numeros.add(5);
numeros.add(6);

System.out.println("Numeros: " + numeros +"\n");

//Método add segunda forma
System.out.println("Añadiendo el valor 99 en la
posición 2...");

numeros.add(2, 99);
System.out.println("Numeros: " + numeros +"\n");

//Método remove
System.out.println("Eliminando el valor en
posición 0...");

numeros.remove(0);
System.out.println("Numeros: " + numeros +"\n");
}
}

```

SALIDA

Números: [1, 2, 3]

Primer número: 1

Segundo número: 2

Tercer número: 3

Sobreescribiendo el primer número a -9...

Números: [-9, 2, 3]

Añadiendo el valor 4, 5, 6 al final de la lista...

Números: [-9, 2, 3, 4, 5, 6]

Añadiendo el valor 99 en la posición 2...

Números: [-9, 2, 99, 3, 4, 5, 6]

Eliminando el valor en posición 0...

Números: [2, 99, 3, 4, 5, 6]

Otros métodos

En adición a los métodos que modifican directamente los valores de las listas individualmente, existen otros métodos útiles de las listas que comentaremos:

- ***Size***: Devuelve el número de elementos que tiene la lista.
- ***Contains***: Comprueba si la lista contiene o no un elemento. En caso afirmativo devolverá *true* y en caso negativo devolverá *false*.
- ***IsEmpty***: Comprueba si la lista está vacía o no. En caso afirmativo devolverá *true* y en caso negativo devolverá *false*.
- ***IndexOf***: Devuelve el índice de la primera estancia del elemento indicado como parámetro.
- ***LastIndexOf***: Devuelve el índice de la última estancia del elemento indicado como parámetro.
- ***Clear***: Elimina todos los elementos de la lista.
- ***Sort***: Permite ordenar la lista utilizando de argumento la interfaz *Comparator*.
- ***ToArray***: Devuelve la lista como un *array* de objetos.

- **Iterator:** Devuelve un *Iterator*.
- **SubList:** Devuelve una vista de la lista de un rango establecido por dos índices, de inicio y de final. No crea una nueva lista si no que accede al rango de la lista.

Para mejor comprensión vamos a dividir esta lista de métodos en tres ejemplos. Empezaremos con los métodos size, contains, indexOf y lastIndexOf. Con estos métodos podremos obtener información útil de la lista.

EJEMPLO 04

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {

        //Definición del array de números del 1 al 5
        ArrayList<Integer> numeros =
        new ArrayList<Integer>(List.of(1,2,3,2,1));
        System.out.println(numeros+"\n");

        //Método size
        System.out.println("Cantidad de números: " +
        numeros.size()+"\n");

        //Método contains
        System.out.println("Contiene el número 3?: " +
        numeros.contains(3));

        System.out.println("Contiene el número 7?: " +
        numeros.contains(7)+"\n");
    }
}
```

```
//Método indexOf
System.out.println("Índice del primer número 2:"
+ numeros.indexOf(2)+"\n");

//Método lastIndexOf
System.out.println("Índice del último número 2:"
+ numeros.lastIndexOf(2)+"\n");
}

}
```

SALIDA

[1, 2, 3, 2, 1]

Cantidad de números: 5

Contiene el número 3?: true

Contiene el número 7?: false

Índice del primer numero 2: 1

Índice del último número 2: 3

EJEMPLO 05

En el siguiente ejemplo se verán los métodos sort y clear. Este primero es un método proveniente de la interfaz Collection, que al usarlo directamente en el objeto ArrayList, hay que indicar como parámetro una función lambda. También se puede importar la librería Collections y usar su método sort.

```
import java.util.ArrayList;
import java.util.List;
//import java.util.Collections;

public class Main {
    public static void main(String[] args) {

        //Definición de la lista con números del 1 al 5
        ArrayList<Integer> numeros =
            new ArrayList<Integer>(List.of(2,4,3,1,5));
        System.out.println(numeros + "\n");

        //Método sort
        System.out.println("Organizando todos los
            elementos de la lista numeros...");

        numeros.sort((o1, o2) -> o1.compareTo(o2));
        System.out.println("Numeros ordenados: " +
            numeros + "\n");
        //Método alternativo:
        //Collections.sort(numeros);
    }
}
```

```
//Método clear
System.out.println("Eliminando todos los
elementos de la lista numeros...");  

numeros.clear();
System.out.println("Numeros: " + numeros +"\n");
}  

}
```

SALIDA

[2, 4, 3, 1, 5]

Organizando todos los elementos de la lista
numeros...

Numeros ordenados: [1, 2, 3, 4, 5]

Eliminando todos los elementos de la lista
numeros...

Numeros: []

EJEMPLO 06

En el siguiente ejemplo se verán los métodos toArray, iterator y subList. Como anteriormente se han dado ejemplos de iterator, en este no se indagará.

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

public class Main {
    public static void main(String[] args) {

        //Definición del array de números del 1 al 5
        ArrayList<Integer> numeros =
            new ArrayList<Integer>(List.of(1,2,3,4,5));
        System.out.println(numeros + "\n");

        //Método toArray
        Object[] numerosArray = numeros.toArray();
        System.out.println("Primer número del array: " +
            numerosArray[0] + "\n");

        //Método Iterator
        Iterator<Integer> numerosIterator =
            numeros.iterator();
        System.out.println("Primer número de la lista
accedido por el Iterator: " +
            numerosIterator.next() + "\n");

        //Método SubList
        List<Integer> subLista = numeros.subList(1, 3);
        System.out.println("Sublista de los índices 1 al
3: " + subLista);
    }
}
```

SALIDA

[1, 2, 3, 4, 5]

Primer número del array: 1

Primer número de la lista accedido por el
Iterator: 1

Sublista entre los índices 1 al 3: [2, 3]

HASHMAP

Es hora de sumergirnos en los HashMap. Estos son una implementación básica de la interfaz Map. Se guardan los datos en pares de claves y valores. Para acceder a un valor, hay que hacer referencia a la clave. Tiene la propiedad de claves únicas y valores repetibles. Si se intenta establecer un valor en una clave usada, este se sobreescribe. Internamente utiliza la técnica Hashing, al igual que su antepasado HashTable de las Legacy Collection. Esta técnica trata de asignar índices o localizaciones a objetos en una estructura de datos mediante fórmulas.

Los HashMap se definen de la siguiente manera:

```
HashMap<TipoDeClave, TipoDeValor> nombreDelMapa = new  
    HashMap<TipoDeClave, TipoDeValor>();
```

Al igual que con ArrayList, podemos ver las partes de la definición claramente. En los mapas, se define qué tipo de dato son las claves y por otro lado los valores. Fuera de eso, sigue el mecanismo convencional de las clases parametrizadas. En HashMap no podemos inicializar (en las últimas versiones de Java) directamente con valores como

ocurría con `ArrayList(List.of(1,2,3,4))`, por lo que deberemos introducirlos mediante métodos. Dicho esto, veamos los métodos de gestión de valores.

Métodos de gestión de valores

- ***Get***: Obtiene el valor del mapa indicando la clave.
- ***Put***: Añade y sustituye valores. Al usar `put` con una clave inexistente, se añadirá; al usar `put` con una clave existente, se sustituirá.
- ***Remove***: Elimina el valor del mapa en la posición indicada por la clave.

EJEMPLO 07

En el ejemplo 07 podrá ver que se ha definido un `HashMap` con los elementos: `{"Uno" = "One", "Dos" = "Two", "Tres" = "Three"}`. Al igual que en `ArrayList`, `HashMap` hereda `toString` también. Se muestran ejemplos de los métodos `put`, `get` y `remove`.

```

import java.util.HashMap;

public class Main{
    public static void main(String[] args) {

        //Definición del mapa
        HashMap<String, String> numerosEnIngles =
        new HashMap<String, String>();
        System.out.println("numerosEnIngles vacío: " +
        numerosEnIngles.toString());

        //Método put para añadir valores
        System.out.println("Introduciendo valores...");
        numerosEnIngles.put("Uno", "One");
        numerosEnIngles.put("Dos", "Two");
        numerosEnIngles.put("Tres", "Three");
        System.out.println("numerosEnIngles con
        valores: " + numerosEnIngles.toString() + "\n");

        //Método get
        System.out.println("Número uno en inglés: " +
        numerosEnIngles.get("Uno") + "\n");

        //Método put para sustituir valores
        System.out.println("Intercambiando el valor de
        la clave \"Dos\"");
        numerosEnIngles.put("Dos", "Zwei");
        System.out.println("numerosEnIngles: " +
        numerosEnIngles + "\n");

        //Método remove
        System.out.println("Eliminando la clave \"Dos\""
        y su valor...");
        numerosEnIngles.remove("Dos");
        System.out.println("numerosEnIngles: " +
        numerosEnIngles + "\n");
    }
}

```

SALIDA

```
numerosEnIngles vacío: {}
Introduciendo valores...
numerosEnIngles con valores: {Uno=One, Dos=Two,
Tres=Three}
```

Número uno en inglés: One

```
Intercambiando el valor de la clave "Dos"...
numerosEnIngles: {Uno=One, Dos=Zwei, Tres=Three}
```

```
Eliminando la clave "Dos" y su valor...
numerosEnIngles: {Uno=One, Tres=Three}
```

Otros métodos

En adición a los métodos que modifican directamente los valores de las listas individualmente, existen otros métodos útiles de los mapas que comentaremos:

- ***Size***: Devuelve el número de elementos que tiene el mapa.
- ***ContainsKey***: Comprueba si el mapa contiene o no una clave. En caso afirmativo devolverá true y en caso negativo devolverá false.

- ***ContainsValue***: Comprueba si el mapa contiene o no un valor. En caso afirmativo devolverá *true* y en caso negativo devolverá *false*.
- ***IsEmpty***: Comprueba si el mapa está vacío o no. En caso afirmativo devolverá *true* y en caso negativo devolverá *false*.
- ***Clear***: Elimina todos los elementos del mapa.
- ***EntrySet***: Devuelve una colección *Set* de los elementos del mapa en formato clave=valor. Se debe introducir como tipo de parámetro *Map.Entry<TipoClave, TipoValor>*.
- ***KeySet***: Devuelve una colección *Set* de las claves del mapa.
- ***Values***: Devuelve una *Collection* de los valores del mapa.

Para mejor comprensión vamos a dividir esta lista de métodos en dos ejemplos. Empezaremos con los métodos size, containsKey, containsValue, isEmpty y clear.

EJEMPLO 08

```
import java.util.HashMap;

public class Main{
    public static void main(String[] args) {
        //Definición del mapa
        HashMap<String, String> numerosEnIngles =
            new HashMap<String, String>();
        System.out.println("Introduciendo valores...");  

        numerosEnIngles.put("Uno", "One");
        numerosEnIngles.put("Dos", "Two");
        numerosEnIngles.put("Tres", "Three");
        System.out.println("numerosEnIngles: " +
            numerosEnIngles.toString() + "\n");

        //Método size
        System.out.println("Número de entradas en el
            mapa: " + numerosEnIngles.size() + "\n");

        //Métodos containsKey y containsValue
        System.out.println("Contiene la clave \"Tres\"?:" +
            + numerosEnIngles.containsKey("Tres"));
        System.out.println("Contiene el valor \"Four\"?:" +
            + numerosEnIngles.containsValue("Four") + "\n");

        //Métodos isClear y clear
        System.out.println("Está vacía?: " +
            numerosEnIngles.isEmpty());

        System.out.println("Eliminando elementos...");  

        numerosEnIngles.clear();
        System.out.println("numerosEnIngles: " +
            numerosEnIngles);

        System.out.println("Está vacía?: " +
            numerosEnIngles.isEmpty() + "\n");
    }
}
```

SALIDA

Introduciendo valores...

```
numerosEnIngles: {Uno=One, Dos=Two, Tres=Three}
```

Número de entradas en el mapa: 3

Contiene la clave "Tres"??: true

Contiene el valor "Four"??: false

Está vacía??: false

Eliminando elementos...

```
numerosEnIngles: {}
```

Está vacía??: true

EJEMPLO 09

En el siguiente ejemplo veremos métodos entrySet, keySet y values. Los tres devuelven un tipo de colección de la familia Collection. Los dos primeros devuelven Sets ya que estos tienen la propiedad de no poder repetirse ningún valor, y si recuerda bien, en los mapas no pueden existir claves iguales, que no se repiten. Por otra parte, los valores pueden repetirse así que se devuelve una colección genérica.

```

import java.util.Collection;
import java.util.HashMap;
import java.util.Map.Entry;
import java.util.Set;

public class Main{
    public static void main(String[] args) {

        //Definición del mapa
        HashMap<String, String> numerosEnIngles =
        new HashMap<String, String>();

        System.out.println("Introduciendo valores...");  

numerosEnIngles.put("Uno", "One");  

numerosEnIngles.put("Dos", "Two");  

numerosEnIngles.put("Tres", "Three");  

        System.out.println("numerosEnIngles: " +
        numerosEnIngles.toString() + "\n");

        //Método entrySet
        Set<Entry<String, String>> numerosEntrySet =
        numerosEnIngles.entrySet();
        System.out.println("Set de entradas: " +
        numerosEntrySet + "\n");

        //Método keySet
        Set<String> claves = numerosEnIngles.keySet();
        System.out.println("Claves: " + claves + "\n");

        //Método values
        Collection<String> valores =
        numerosEnIngles.values();
        System.out.println("Valores: " + valores +" \n");
    }
}

```

SALIDA

Introduciendo valores...

numerosEnIngles: {Uno=One, Dos=Two, Tres=Three}

Set de entradas: [Uno=One, Dos=Two, Tres=Three]

Claves: [Uno, Dos, Tres]

Valores: [One, Two, Three]

CONCLUSIÓN

El framework de las colecciones de datos de java son muy interesantes y espero que al lector también le haya parecido eso. Aún hay mucho de lo que hablar sobre colecciones en Java, Sets, Queues, sincronización, clases abstractas, métodos de hashing... pero eso deberá verse en otro libro.

¿De dónde viene la información?

La información viene de diferentes sitios web, estos son:

- Wikipedia.org
- geeksforgeeks.org
- docs.oracle.com
- akcoding.com

Además hay que mencionar el libro “Aprende Java en un fin de semana” que me introdujo a las colecciones de datos y me ha inspirado para escribir un libro.

Agradecimientos

“Como humanos, no hemos evolucionado haciendo descubrimientos, sino compartiéndolos con otros” - NoSeQuienLoDijo. Sin duda este libro no sería posible si no hubiese esas personas que comparten su conocimiento. Escribir este documento me ha hecho respetar más a los escritores de libros.

Y por supuesto, ¡gracias a usted por adquirir “Colecciones de Datos en Java”! Espero que le haya encontrado utilidad. Además, esto es solo el principio, las colecciones de datos le acompañarán cuando programe y descubrirá cada vez más sobre ellas.





CLIFFORD

Editorial

DATOS, DATOS y MÁS DATOS, pero, ¿dónde los almacenamos? La respuesta es, en Colecciones de Datos (Casualmente el título del libro). En Colecciones de Datos en Java se presenta una introducción al maravilloso mundo de las colecciones. El lector podrá aprender sobre las familias del "Collection Framework" y las nociones básicas sobre ArrayList y HashMap ¡EMOCIONANTE!



83,22 €

ISBN 99999999999998

1DAMPIGC128



99999999999998

