Colecciones

Colecciones en Java

Elizabeth León Guzmán, Ph.D.
eleonguz@unal.edu.co

Jonatan Gómez Perdomo, Ph. D.
jgomezpe@unal.edu.co

Arles Rodríguez, Ph.D.
aerodriguezp@unal.edu.co

Camilo Cubides, Ph.D. (c)
eccubidesg@unal.edu.co

Carlos Andres Sierra, M.Sc.
casierrav@unal.edu.co

Research Group on Data Mining – Grupo de Investigación en Minería de Datos – (Midas)
Research Group on Artificial Life – Grupo de Investigación en Vida Artificial – (Alife)
Computer and System Department
Engineering School
Universidad Nacional de Colombia

Agenda

1 Introducción

2 Listas

3 Map

MIDAS

UNIVERSIDAD NACIONAL DE COLOMBIA un

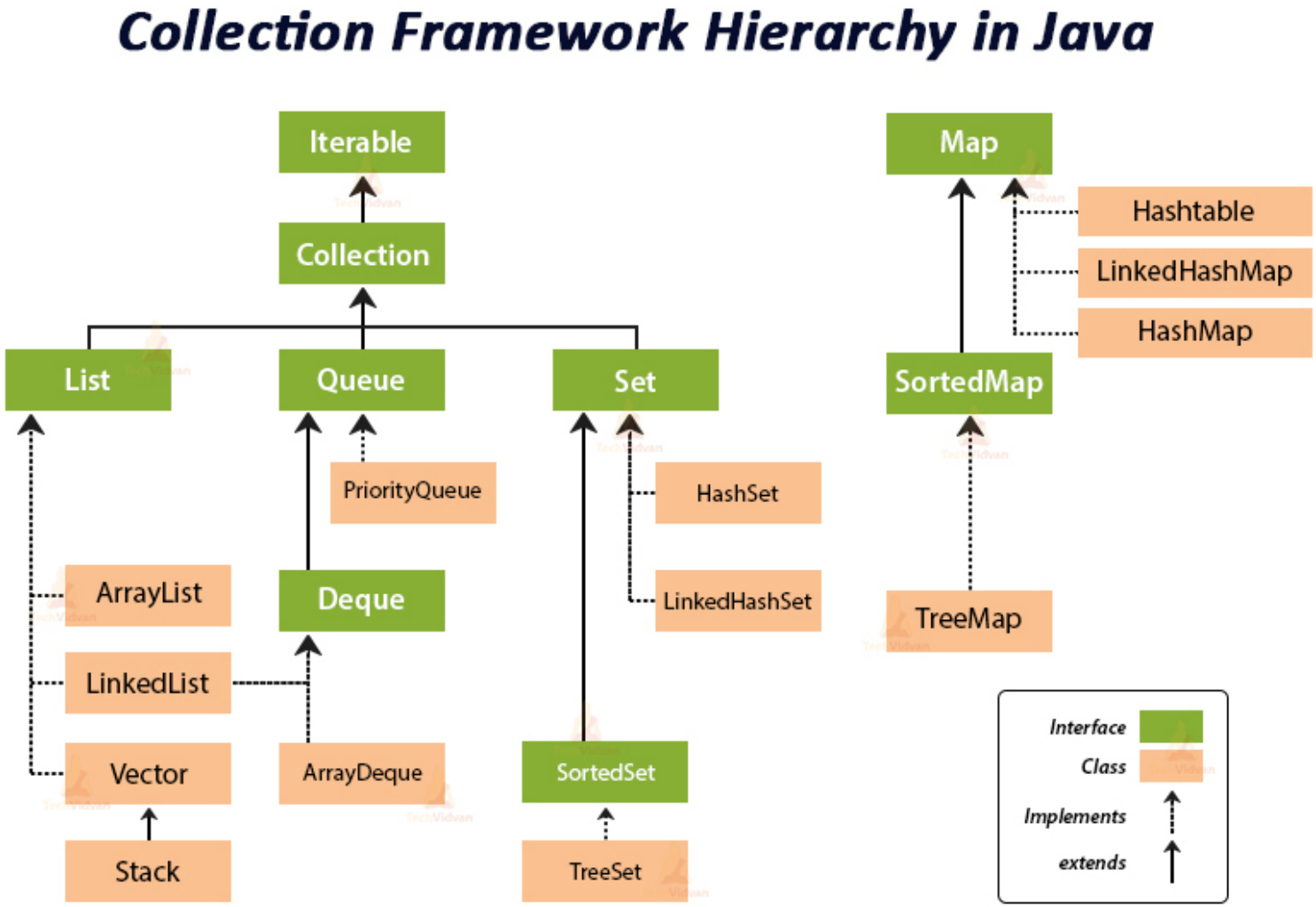
Colecciones en Java

Definición

Una **colección** en Java es una estructura que permite mantener y manipular, de manera dinámica, elementos de un mismo tipo (paramétrizado) de acuerdo a políticas de manejo diseñadas para la colección.

Las colecciones del estándar de Java extienden y/o implementan la interfaz genérica `Iterable<T>` o la interfaz genérica `Map<K,V>`.
En el primer caso la colección puede ser recorrida para consultar y/o manipular los elementos almacenados en ella. En el segundo caso, se puede iterar sobre la colección de llaves, de valores o de las parejas clave/valor.
Se debe tener cuidado con la manipulación de los elementos de una colección mientras se itera sobre ella pues se pueden generar efectos laterales indeseados.

Colecciones en Java



https://techvidvan.com/tutorials/java-collection-framework/

Agenda

1 Introducción

2 Listas

3 Map

Interfaz List - Definición

Un **List**, o simplemente lista, es una colección que mantiene los elementos en un orden lineal (secuencial), lo que le permite acceder a ellos (para modificarlos, eliminarlos o adicionarlos) mediante un índice entero mayor o igual a 0.

Existen tres implementaciones de `List`: ArrayList, LinkedList y Vector. Estas implementaciones varian en detalles que las hacen o más eficientes o menos eficientes en acceso, adición, y/o eliminación de elementos.

La descripción completa de la interfaz `List` se puede encontrar en:

`https://docs.oracle.com/javase/8/docs/api/java/util/List.html`

Interfaz List - Métodos de consulta

1. `get(indice)`: Retorna el elemento ubicado en la posición `indice` de la lista.
2. `contains(e)`: Retorna `true` si la lista contiene al elemento `e`, `false` en otro caso.
3. `indexOf(e)`: Retorna la primer posición en la que aparece el elemento `e`, `-1` si la lista no tiene dicho elemento.
4. `isEmpty()`: Retorna `true` si la lista está vacia, `false` en otro caso.
5. `size()`: Retorna el tamaño (número de elementos) de la lista.

Los métodos que requieren posición generan una excepción si la misma no es válida, es decir, si es negativa o mayor al tamaño de la lista.

MIDAS

UNIVERSIDAD NACIONAL DE COLOMBIA un

Interfaz List - Métodos de modificación

1. `add(e)`: Agrega el elemento `e` al final de la lista.
2. `add(indice, e)`: Inserta el elemento `e` en la posición `indice` de la lista.
3. `set(indice, e)`: Reemplaza el elemento que se encuentre en la posición `indice` con el elemento `e`.
4. `remove(indice)`: Elimina el elemento de la lista que se encuentre en la posición `indice`. Recibe un entero.
5. `remove(e)`: Elimina la primer ocurrencia del elemento `e` en la lista. Recibe un objeto del tipo parametrizado (genérico).
6. `clear()`: Elimina todos los elementos de la lista.

Los métodos que requieren posición generan una excepción si la misma no es válida, es decir, si es negativa o mayor al tamaño de la lista.

ArrayList I

La clase `ArrayList<T>` implementa la interfaz `List<T>` usando un arreglo para mantener los elementos (no entraremos en los detalles aquí). El tiempo promedio que gasta en los procesos de almacenamiento y acceso de elementos es menor que el tiempo promedio de las otras dos clases que implementa `List` (`Vector` y `LinkedList`). No sucede lo mismo con el tiempo promedio usado en procesos de eliminación.

La descripción completa de la clase `ArrayList` se puede encontrar en:

`https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html`

ArrayList II

Ejemplo

```
import java.util.ArrayList;
public class Main{
    public static void main(String[] args) {
       // Creando una lista de enteros
        ArrayList<Integer> lista = new ArrayList<Integer>();
        // Agregando elementos
        for (int i = 1; i <= 10; i++) lista.add(i);
        // Imprimiendo elementos
        System.out.println(lista);
```

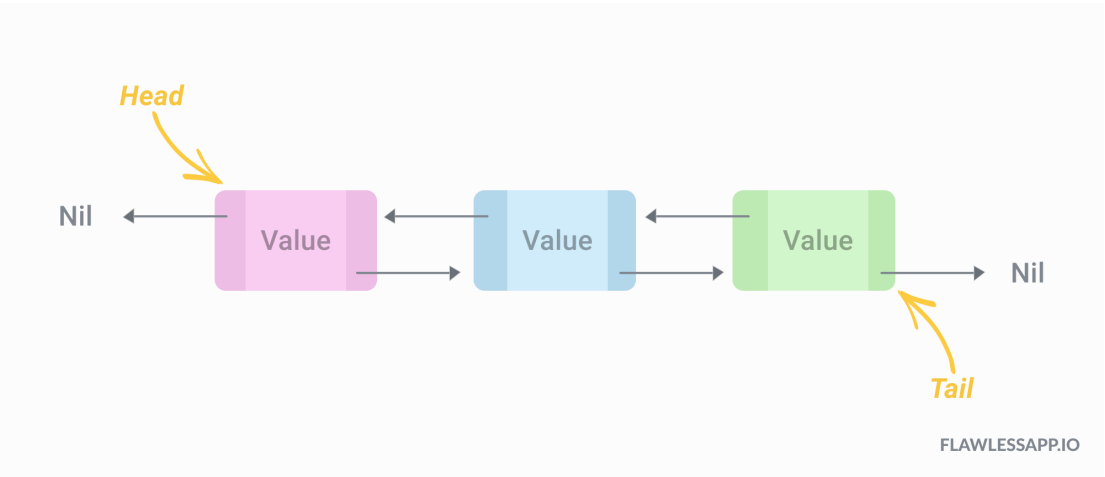
ArrayList III

Ejemplo (continuación)

```
...
        // Quitando el elemento en la posición 3
        lista.remove(3);
        // Imprimiendo el arreglo
        System.out.println(lista);
        for (int i = 0; i < lista.size(); i++)
            System.out.print(lista.get(i) + " ");
        System.out.println();
        // Usando el iterador de la lista
        for( Integer n:lista ) System.out.print(n + " ");
    }
}
```


LinkedList I

La clase `LinkedList<T>` implementa la interfaz `List<T>` manteniendo los elementos en una serie de nodos enlazados entre sí como eslabones de una cadena. Cada uno de estos nodos apunta tanto a su antecesor y al elemento que le sigue.



`https://miro.medium.com/max/4000/1*Rkn3q6HJoEkRO4T_SVlyuw.png`

La descripción completa de la clase `ArrayList` se puede encontrar en:

`https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html`

MIDAS

UNIVERSIDAD NACIONAL DE COLOMBIA un

LinkedList II

Algunos métodos adicionales, respecto a `List`, que son muy importantes de esta clase son:

1. `addFirst(e)`: Agrega el elemento e al inicio de la lista.
2. `addLast(e)`: Agrega el elemento e al final de la lista.
3. `getFirst()`: Obtiene el primer elemento de la lista.
4. `getLast()`: Obtiene el último elemento de la lista.
5. `removeFirst()`: Elimina el primer elemento de la lista.
6. `removeLast()`: Elimina el último elemento de la lista.

Existen muchos más métodos que operan sobre el primero y el último de la lista, que son facilmente accedidos por la estructura de lista encadenada usada. Estos se pueden consultar con más detalle en:

`https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html`

LinkedList III

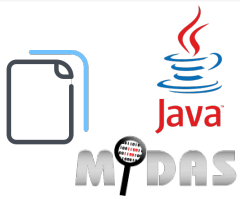
Ejemplo

```
import java.util.LinkedList;
public class Main{
     public static void main(String args[]){
        // LinkedList de cadenas de caracteres
        LinkedList<String> lista = new LinkedList<String>();

        // mecanismos para agregar elementos
        lista.add("A");
        lista.add("B");
        lista.addLast("C");
        lista.addFirst("D");
        lista.add(2, "E");
        System.out.println("Lista 1:" + lista);
```

LinkedList IV

Ejemplo (continuación)

```
...
        // modificación de un elemento
        lista.set(1, "a");
        System.out.println(lista);
        // impresión utilizando ciclos
        for (int i = 0; i < lista.size(); i++)
            System.out.print(lista.get(i) + " ");
        System.out.println();

        // mecanismos para eliminar elementos
        lista.remove("B");
        lista.remove(3);
        lista.removeFirst();
        lista.removeLast();
        System.out.println("Lista final: " + lista);
    }
}
```



Vector I

La clase `Vector<T>` es muy parecida a la clase `ArrayList<T>`, pues usa un arreglo de base. Esta implementación de lista varia un poco en el nombre de algunas operaciones, pero su comportamiento es similar al de otras inmplementaciones.

Vector es una implementación sincronizada, esto quiere decir que no admite múltiples procesos accediéndolo al mismo tiempo. Tiene buen desempeño en términos de la expandibilidad, pero suele ser más lento que otras implementaciones de listas.

La descripción completa de la clase `Vector` se puede encontrar en:

`https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html`

Vector II

Ejemplo

```
import java.util.Vector;

public class Main{

    public static void main(String[] args){
        // Vector para almacenar cualquier tipo
        Vector vector = new Vector();

        // agregar elementos de distinto tipo
        vector.add(1);
        vector.add(true);
        vector.add("Mision");
        vector.add("TIC");
        vector.add(2);
        System.out.println("Vector 1: " + vector);
```

Vector III

Ejemplo (continuación)

```
...
        // modificar el valor de uno de los elementos
        vector.set(0, 2021);
        System.out.println("Vector 2: " + vector);

        // eliminar elemento en el índice 4
        vector.remove(4);
        System.out.println("Vector 3: " + vector);

        // impresión de elementos con ciclo por elemento
        for(Object elemento : vector)
            System.out.print(elemento + " ; ");
    }
}
```

Agenda

1 Introducción

2 Listas

3 Map

MIDAS

UNIVERSIDAD NACIONAL DE COLOMBIA un

Pareja clave-valor I

En algunas ocasiones, determinar que un objeto es igual a otro (por su **valor**), resulta una tarea muy dispendiosa, ya sea por la complejidad de los objetos o por su tamaño. Por ejemplo, determinar si dos ciudadanos son el mismo sería difícil usando las características físicas (con dispositivos biométricos es cada vez más fácil), que si les asignáramos un número, etiqueta o similar (llamado **clave** o **llave**) con el que los pudiéramos reconocer. Por ejemplo, las cédulas y tarjetas de identidad ya no son sólo el número que sirve para identificarnos, sino que también llevan información biométrica de nosotros en ellas.

Una pareja clave-valor relaciona un par de objetos, la **clave** con que se identifica o localiza a un objeto, y el **valor** que es el objeto referenciado.

MIDAS

UNIVERSIDAD NACIONAL DE COLOMBIA un

Pareja clave-valor II

Las parejas clave-valor en Java son instancias de alguna clase que implemente la interfaz genérica `Entry<K,V>` que se encuentra encapsulada en la interfaz `Map<K,V>`. Conceptualmente, notaremos a las parejas clave-valor como `clave:valor`.
Los siguientes son ejemplos de parejas clave-valor (conceptualmente):

- `22:"SSH"`
- `80:"HTTP"`
- `"edad":18`
- `"Java":"Lenguaje de programación"`
- `12100100:"Juan María de los Santos"`
- `"primos":[2, 3, 5, 7, 11, 13, 17, 19]`

MIDAS

UNIVERSIDAD NACIONAL DE COLOMBIA un

Interfaz Map<K,V> y Diccionarios

Un **diccionario** es una colección de parejas clave-valor donde los valores pueden ser recuperados principalmente por su clave. Cada pareja clave-valor en un diccionario es considerada un **ítem** o **registro**. En Java, los diccionarios se representan de manera abstracta con la interfaz genérica `Map<K,V>` y de manera concreta con alguna de las clases genéricas que implementan esta interfaz: `TreeMap<K,V>`, `Hashtable<K,V>`, `LinkedHashMap<K,V>` o `HashMap<K,V>`.
Las siguientes, son representaciones conceptuales de un diccionario:

- `{22:"SSH", 23:"Telnet", 80:"HTTP", 3306:"MySQL"}`
- `{'A':65, 'B':66,..., 'Z':90}`
- `{"nombre":"Pepe", "Apellido":"Grillo", "tipo":"amigo"}`

La descripción completa de la interfaz Map se puede encontrar en:
`https://docs.oracle.com/javase/8/docs/api/java/util/Map.html`

Interfaz Map - Métodos de iteración y tamaño

1. `entrySet()`: Retorna una colección iterable (un `Set<Entry<K,V>>`) de las parejas clave-valor almacenadas en el diccionario.
2. `keySet(clave)`: Retorna una colección iterable (un `Set<K>`) de las claves almacenadas en el diccionario.
3. `values()`: Retorna una colección iterable (no específica) de los valores almacenados en el diccionario.
4. `isEmpty()`: Retorna `true` si el diccionario está vacio, `false` en otro caso.
5. `size()`: Retorna el número de items (parejas ordenadas) almacenadas en el diccionario.

Interfaz Map - Métodos de consulta y modificación

1. `get(clave)`: Retorna el elemento asociado a la `clave` dada.
2. `containsKey(clave)`: Retorna `true` si el diccionario tiene la `clave` dada, `false` en otro caso.
3. `containsValue(valor)`: Retorna `true` si el diccionario tiene el `valor` dado, `false` en otro caso.
4. `put(clave,valor)`: Agrega la pareja `clave:valor` al diccionario.
5. `remove(clave)`: Elimina del diccionario la pareja con la `clave` dada.
6. `remove(clave,valor)`: Elimina del diccionario la pareja `clave:valor`.
7. `clear()`: Elimina todos las parejas clave-valor del diccionario.

HashMap I

Esta implementación almacena las claves en una **tabla hash**. Es la implementación con mejor rendimiento de todas, pero no garantiza ningún orden a la hora de realizar iteraciones.

Esta implementación proporciona tiempos constantes en las operaciones básicas siempre y cuando la *función hash* disperse de forma correcta los elementos dentro de la tabla hash. Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación.

La descripción completa de la interfaz `HashMap` se puede encontrar en:
`https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html`

HashMap II

Ejemplo

```
import java.util.HashMap;
import java.util.Map;
public class Main {
    public static void main(String[] args){
        // HashMap con claves string y valores double
        HashMap<String, Double> mapa = new HashMap<>();

        // agregar elementos al HashMap
        mapa.put("nota1", 3.2);
        mapa.put("nota2", 4.3);
        mapa.put("nota3", 3.9);

        // cantidad de elementos en el HashMap
        System.out.println("Tamaño: " + mapa.size());
        // impresión del HashMap
        System.out.println("HashMap 1: " + mapa);
```

HashMap III

Ejemplo (continuación)

```
...
        // verificación de llave y extracción de valor
        if (mapa.containsKey("nota2")) {
            Double valor = mapa.get("nota2");
            System.out.println("nota 2 === " + valor);
        }

        mapa.remove("nota1");
        System.out.println("HashMapFinal:");
        // recorrer HashMap por llave y clave
        for (Map.Entry<String, Double> elem : mapa.entrySet())
            System.out.println(elem.getKey() + " : " + elem.getValue());
    }
}
```

TreeMap I

Esta implementación almacena las claves ordenándolas dentro de una estructura de datos de árbol. Es **más lento** que `HashMap` en el almacenamiento, pero es más rápido en recuperación de objetos.

Para ello, la clase de las claves almacenadas debe implementar la interfaz `Comparable<T>`. Esta implementación garantiza un rendimiento de $log(n)$ en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.

La descripción completa de la interfaz `TreeMap` se puede encontrar en:
`https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html`

TreeMap II

Ejemplo

```
import java.util.Map;
import java.util.TreeMap;
public class Main {
    public static void main(String args[]) {
        // Creación del TreeMap
        TreeMap<Integer, String> mapa
                = new TreeMap<Integer, String>();
        // Agregar elementos
        mapa.put(3, "MisionTIC");
        mapa.put(2, "estudiantes");
        mapa.put(1, "hola");
        for (Map.Entry pareja : mapa.entrySet()) {
            int clave = (int)pareja.getKey();
            String valor = (String)pareja.getValue();
            System.out.println(clave + " : " + valor);
        }
    }
}
```

Problemas varios I

Problemas

1. Dada una lista de números, eliminar los duplicados e imprimir la lista procesada.
2. Dadas dos listas de números, las cuales pueden ser de distinto tamaño, e imprimir los números se encuentran en ambas listas.
3. Dada una lista de objetos, imprimir la inversa de dicha lista.
4. Desarrollar un algoritmo que verifique si todas las parejas `clave:valor` de un HashMap se encuentran en otro HashMap.
5. Desarrollar una función que reciba dos TreeMap como parámetros y los mezcle, es decir, que se construya un nuevo TreeMap con las llaves de los dos TreeMap recibidos; si hay una `clave` repetida en ambos TreeMap, se debe asignar el `valor` que tenga la `clave` en el primer TreeMap.