INFORMATORIO

Taller de Programación Web

Arrays, Colecciones y Maps

INTRODUCCIÓN

Como hemos visto en módulos anteriores en ocasiones necesitamos almacenar un conjunto de datos (que pueden ser datos simples como números u objetos complejos como Alumnos) para poder manipularlos (buscar, filtrar, agrupar, y más). Para ellos necesitaremos estructuras del tipo colección o diccionarios (en JAVA, le diremos Map).

ARRAY

Los Array son objetos en Java que pueden almacenar múltiples variables al mismo tiempo. Pueden almacenar datos primitivos (como int, long, double, etc) o referencias de objetos.

Los Arrays son eficientes, pero la mayoría de las veces quizás utilices uno de los tipos de colecciones (ArrayList, HashSet, etc). Las clases de Coleccion ofrecen métodos más flexibles de acceso a objetos (insercion, eliminacion, etc) a diferencia de Arrays, pueden expandir o contraer su capacidad a medida que agreguemos y removamos objetos.

DECLARACIÓN DE ARRAYS, CREACION Y ASIGNACION

```
   //Declaración de un array de primitivos (ambos son correctos):
   int[] dnisChaco; //corchetes antes del nombre de la variable
(recomendado)

   //Declaración de Array de Referencias de Objetos
   Thread[] threads;

   //Declaración de Arrays multidimensional
   String [][] jefesYEmpleados;
   //Declarando y Construyendo un Array
   //Declaramos una variable de tipo Array y invocamos su constructor
con new
   //Dentro de los corchetes debemos definir el tamaño del Array
   int[] notas = new int[10];
  {1,2,3,4,5,6,7,8,9}
   //Asignar datos al Array
   notas[0] = 10;
   notas[1] = 4;

   //Ejemplo con Objetos
   //Alumnos[] alumnos = new Alumnos[4];
   //alumnos[0] = new Alumno();
   //alumnos[1] = new Alumno();
```

IMPORTANTE:
Si creamos un Array de 5 elementos (new Array[5]). Los índices para acceder a los elementos del array comienzan en 0 (o sea, que el primer índice para un array de 5 elementos será 0 y el último el 4).
En caso de querer acceder a un índice fuera del rango se producirá un excepción del tipo `ArrayIndexOutOfBoundsException`

COLECCIONES

Como hemos visto en el módulo de "Programación Web (Python/Django)" existen estructuras para almacenar objetos. En Java pueden que tengan otros nombres o implementaciones pero trataremos de hacer relaciones para que sea más fácil la interpretación.

Interfaces claves

`Collection`	`Set`	`SortedSet`
`List`	`Map`	`SortedMap`
`Queue`	`NavigableSet`	`NavigableMap`

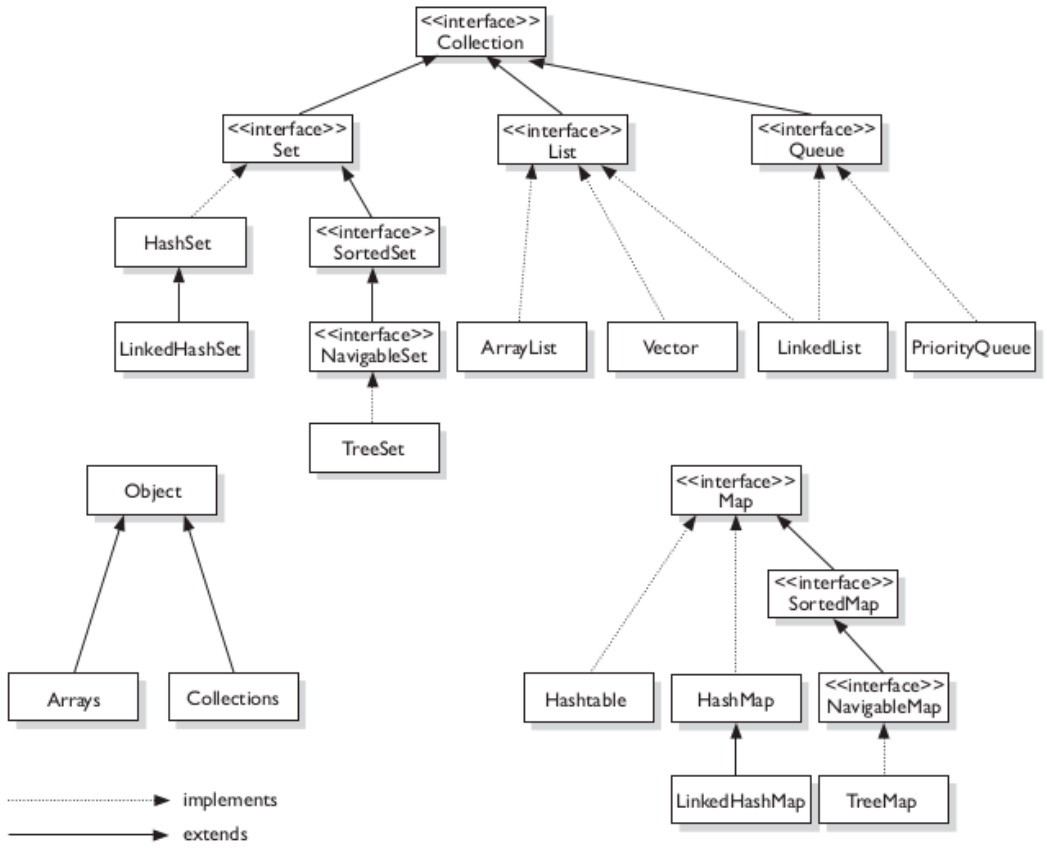
Clases de Implementación

Maps	Set	Lists	Queues	Utilities
HashMap	HashSet	ArrayList	PriorityQueue	Collections
HashTable	LinkedHashSet	Vector		Arrays
TreeMap	TreeSet	LinkedList		

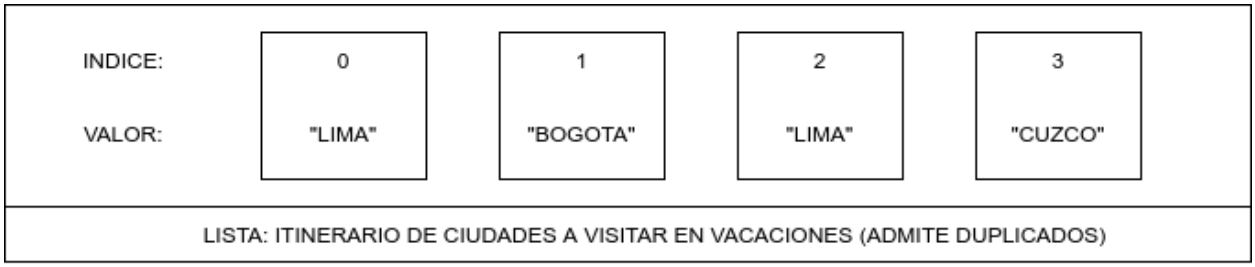
LinkedHashMap				

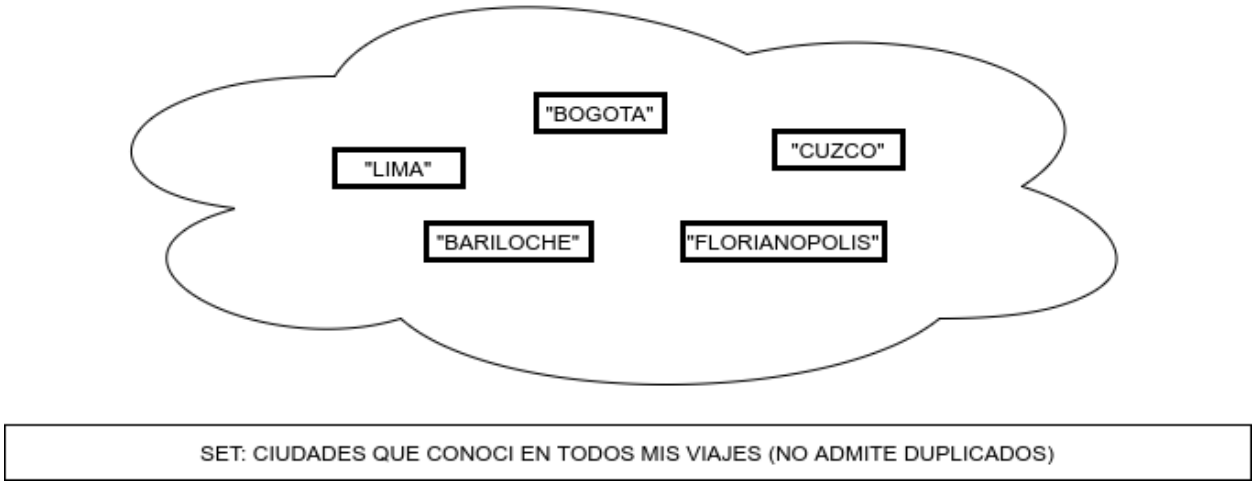
Observación 1:
La interface Map y sus implementaciones pueden ser relacionados con los Diccionarios en Python.
Observación 2:
JAVA Collection Framework contiene todas estas estructuras. Pero, no todas implementan la interface Collection. Específicamente, ninguna de las clases e interfaces relacionadas con Map extienden de una Collection.

Jerarquía de Interfaces y Clases para Colecciones



EJEMPLOS GRÁFICOS DE ESTRUCTURA SIMPLE DE LIST Y SET





LISTA (INTERFACE LIST)

En esta interface se hace enfoque en un elemento llamado índice (index) como se puede apreciar en el gráfico anterior y prácticamente es similar a la estructura de Lista en Python.

Características:

- Posee métodos relacionados a índices. Como lo son `getIndex(int index)`, `indexOf(Object o)`, `add(int index, Object obj)`, y más.

- Todas las implementaciones de la interfaz `List` s**on ordenadas de acuerdo a la posición del índice**.
- Se puede agregar un elemento a una posición específica (indicando el índice), o se puede agregar sin especificarlo. En el último caso, el elemento se coloca al final.

IMPLEMENTACIONES DE LA INTERFACE LIST:

- **ArrayList**: Una de las más usadas en este curso. Provee una rápida iteración y un rápido acceso aleatorio. Es una colección ordenada (por índice), esto quiere decir que respeta el orden de inserción pero no está ordenada por algún criterio (alfabético asc/desc, numérico asc/desc).
- **Vector**: Una de las primeras colecciones de JAVA (junto a `HashTable`). Básicamente, es lo mismo que un `ArrayList`, pero los métodos de la Clase `Vector` son sincronizados y thread-safe (estos términos son de concepto avanzado que los veremos más adelante en el curso. Pero pueden buscar e investigar si desean adelantar).
- **LinkedList**: Está ordenada por la posición del índice, como `ArrayList`, excepto que los elementos están doblemente enlazados entre ellos. Este enlace nos provee nuevos métodos para agregar elementos al principio y al final, lo que hace mucho más fácil implementar una pila o cola (investigar stack/queue y LIFO/FIFO). Hay que tener en cuenta que una `LinkedList` al iterar (recorrer) puede ser más lenta que un `ArrayList`, pero es una buena elección cuando necesitas una rápida inserción y borrado.

EJEMPLOS DE OPERACIONES BÁSICAS:

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

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

       //Inicializar ArrayLists (Ejemplo de Constructores)
       List<Integer> dnis = new ArrayList<Integer>(); //versión tradicional
       List<String> nombres = new ArrayList<>(); //versión abreviada

       // Operaciones de Inserción
       dnis.add(123456789);
       dnis.add(987654321);

       nombres.add("Sr. Calamardo");
       nombres.add("Bob Esponja");
       nombres.add("Patricio");

       System.out.println("Elementos en la Lista dnis: " + dnis.size());
       System.out.println("Elementos en la Lista nombres: " + nombres.size());
       System.out.println(dnis.size() == nombres.size());  //Preguntamos si ambas listas
tienen la misma cantidad de elementos

       //Operaciones de Acceso
       System.out.println("Elemento " + 1 + " de la Lista dni: " + dnis.get(1));
       System.out.println("Elemento " + (nombres.size() -1)
           + " de la Lista nombres: " + nombres.get(nombres.size() -1)); //acceder al último
elemento

       //Operaciones de Eliminar
       nombres.remove("Patricio");
       System.out.println("Elementos en la Lista: " + nombres.size()); //Comparar el tamaño
luego de remover
   }
}
```

SET (INTERFACE SET)

Un Set se enfoca en no admitir duplicados (solo únicos). Gracias al método equals() puede determinar si 2 objetos son idénticos.

Características:

- Sus elementos son únicos
- Para poder identificar si un objeto es único se hace uso del método `equals()`
- Cuando trabajamos con objetos propios debemos sobreescribir (Override) el método `hashCode()` que también es un método importante para las estructuras `HashSet` y `LinkedHashSet`.

IMPLEMENTACIONES DE LA INTERFACE SET:

- **HashSet**: Un `HashSet` es desordenado (no tiene orden ascendente/descendente por claves, números, alfabético, etc) y además es no ordenado (no reconoce un orden de inserción, si un elemento lo agrega no le asigna un índice). Hace uso del `hashCode()` del objeto a insertar (mientras más eficiente sea la implementación del `hashCode()` , mejor rendimiento tendrá en el acceso). Se debe usar esta clase cuando quieras una colección sin duplicados y no te interese el orden de los elementos al iterar.
- **LinkedHashSet**: Es la versión ordenada (con respecto a la inserción) del `HashSet`, que mantiene una lista doblemente enlazada a través de todos los elementos. Usa esta clase en vez de `HashSet` si te interesa iterar de acuerdo a un orden. Cuando se recorre un `HashSet` el orden es impredecible, mientras que `LinkedHashSet` te permite iterar los elementos de acuerdo al orden que fueron insertados.

- **TreeSet**: No profundizaremos en este tipo de colección. Pero podemos decir que junto a `TreeMap` son colecciones que están ordenadas (orden natural ascendente) y posee una estructura de Árbol (Red-Black Tree).

EJEMPLOS DE OPERACIONES BÁSICAS:

```
import java.util.HashSet;
import java.util.Set;

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

       //Inicializar HashSets (Ejemplo de Constructores)
       Set<Long> dnis = new HashSet<Long>(); //version tradicional
       Set<String> nombres = new HashSet<>(); //versión abreviada

       // Operaciones de Inserción
       dnis.add(Long.valueOf(2131231312));
       dnis.add(Long.valueOf(987654321));

       nombres.add("Sr. Calamardo");
       nombres.add("Bob Esponja");
       nombres.add("Patricio");
       nombres.add("Patricio"); //agregamos un repetido para comprobar luego el size()

       System.out.println("Elementos en la Lista dnis: " + dnis.size());
       System.out.println("Elementos en la Lista nombres: " + nombres.size());
       System.out.println(dnis.size() == nombres.size());  //Preguntamos si ambas listas tienen la
misma cantidad de elementos

       //Operaciones de Acceso - Al ser Set no tendremos índices. ¿Cómo accedemos? Iterando
       for (String nombre: nombres) { //Recorremos
           System.out.println(nombre); //nombre es la variable que tiene el valor asignado del
elemento en un instante dado
       }

       //Operaciones de Eliminar
       nombres.remove("Patricio");
       System.out.println("Elementos en la Lista: " + nombres.size()); //Comparar el tamaño luego de
remover
   }
}
```

MAP (INTERFACE MAP)

Un `Map` (O Diccionario como vimos en Python) se enfoca en identificadores únicos. Mapeas una clave única (un ID) a un valor específico.

Características:

- Es una estructura de clave/valor (key/value).
- Permite operaciones de búsqueda por clave, solicitar todos los valores, o solicitar todos los valores y más.
- Al igual que el `Set`, `Map` hace uso del método `equals()` para determinar si dos claves son idénticas o diferentes.

IMPLEMENTACIONES DE LA INTERFACE MAP:

- **HashMap**: Provee un `Map` desordenado (sin orden alfabético, numérico, etc) y no ordenado (con respecto a la inserción). Cuando no te interese el orden (al iterar), `HashMap` es la indicada; las demás agregan más complejidad. Al igual que el `HashSet`, una buena implementación del `hashCode()` ayudará a la performance de acceso. `HashMap` permite una clave null, y múltiples valores null.
- **Hashtable**: Como la clase `Vector` (`List`), `Hashtable` es la versión sincronizada de `HashMap`. A diferencia de `HashMap`, `Hashtable` no permite null (en claves o valores).
- **LinkedHashMap**: Mantiene el orden de inserción (idem a `LinkedHashSet`). Aunque será un poco más lenta que `HashMap` a la hora de inserción y borrado de elementos.
- **TreeMap**: Posee un ordenamiento natural(numérico ascendente, u otro configurable). Se puede definir un ordenamiento personalizado (con

Comparable o Comparator) al construir un `TreeMap`, que especifica cómo los elementos deben ser comparados.

EJEMPLOS DE OPERACIONES BÁSICAS:

```
import java.util.HashMap;
import java.util.Map;

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

       //Inicializar HashMaps (Ejemplo de Constructores)
       Map<Integer, String> dnis = new HashMap<Integer, String>(); //clave (Integer) y valores
(String)
       Map<String, String> personajesShow = new HashMap<>(); //clave (String) y valores (String)

       // Operaciones de Insercion
       dnis.put(2131231312, "Homero Simpson");
       dnis.put(987654321, "Bart Simpson");

       personajesShow.put("Sr. Calamardo", "Bob Esponja");
       personajesShow.put("Bob Esponja", "Bob Esponja");
       personajesShow.put("Patricio", "Bob Esponja");
       personajesShow.put("Patricio", "Bob Esponja - Temporada 2"); //agregamos una clave repetida
para comprobar luego el size()

       System.out.println("Elementos en la Lista dnis: " + dnis.size());
       System.out.println("Elementos en la Lista nombres: " + personajesShow.size());
       System.out.println(dnis.size() == personajesShow.size());  //Preguntamos si ambas maps tienen
la misma cantidad de elementos

       //Operaciones de Acceso
       System.out.println(personajesShow.get("Bob Esponja")); //Buscamos por clave, nos devuelve
valor

       //Operaciones de Eliminar
       personajesShow.remove("Patricio");
       System.out.println("Elementos en la Lista: " + personajesShow.size()); //Comparar el tamaño
luego de remover
   }
}
```

QUEUE (INTERFACE QUEUE)

Está diseñada para almacenar una lista de “quehaceres” (to-do list), o cosas que deben ser procesadas de alguna manera. Aunque otros órdenes pueden ser posibles, queues son típicamente FIFO (First-In First-Out, Primero-EnLlegar Primero-EnSalir).

En este curso no profundizaremos en implementaciones de `Queue`. Pero puedes investigarlo en tu tiempo libre.

CONCLUSIÓN

A la hora de trabajar con un conjunto de objetos y realizar operaciones, cálculos, estadísticas y más sobre ellos haremos uso de estructuras de colecciones.

Dependiendo del problema a resolver se debe analizar cuál es la colección que más se ajusta a nuestra necesidad. ¿Nos interesa el orden en que se insertan? Si es así, una de las interfaces de List puede sernos útil.
¿Admitiremos duplicados? Quizás una de las implementaciones de `Set` se ajuste mejor.
Quizás un `Map`, con su estructura de clave/valor nos permite indexar la información de manera más prolija.
Por último, deberemos considerar el volumen de objetos porque si utilizamos alguna implementación con algunas optimizaciones en las operaciones puede impactar en la performance (pero estaríamos hablando de un gran volumen de objetos).