Curso de Java SE Orientado a...

Entender la Programación...

1 Programación orientada a objetos en Java

2 ¿Qué es un Objeto?

3 Abstracción: ¿Qué es una Clase?

4 Modularidad

Definir Clases y sus componentes

5 Creando nuestra primera Clase

6 Método constructor

7 Static: Variables y Métodos Estáticos

8 Creando elementos estáticos

9 Final: Variables Constantes

10 ¡Reto!

11 Sobrecarga de métodos y constructores

12 Variable vs. Objeto: Un vistazo a la memoria

Collections 25/38

Curso de Java SE Orientado a Objetos

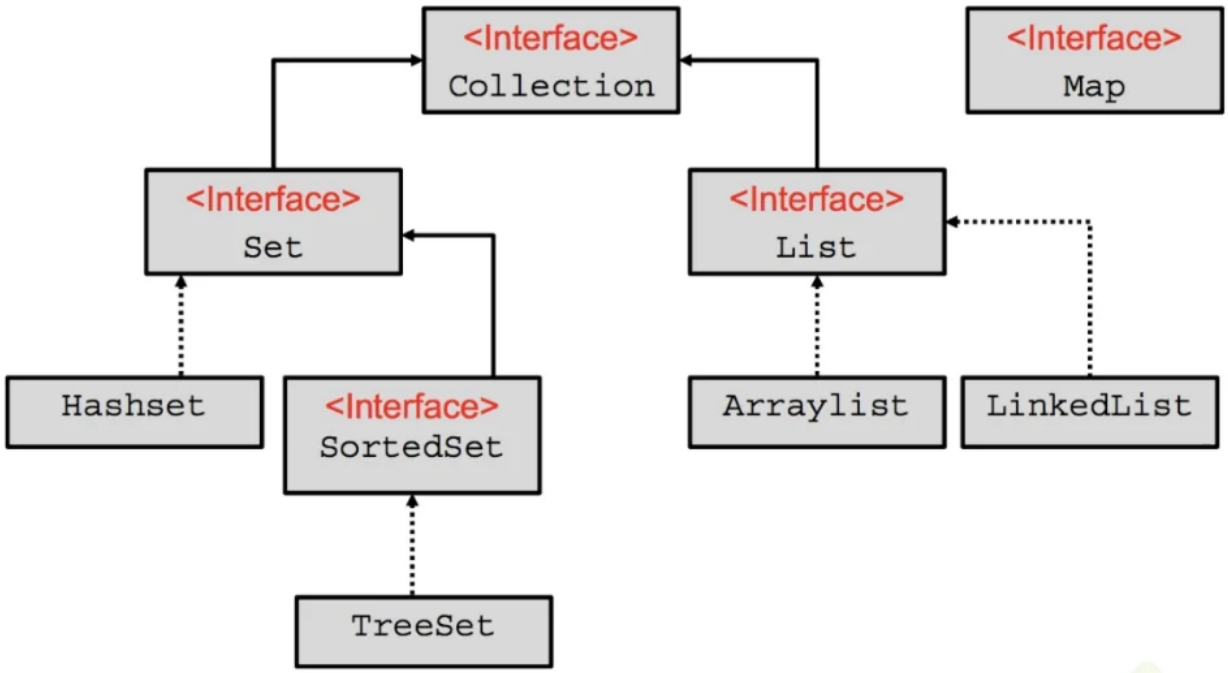
→|CLASES ABSTRAC...

Otras interfaces que son muy importantes en Java son los llamados **Collections**

Los Collections nos van a servir para trabajar con colecciones de datos, específicamente y **solamente con objetos**, para esto recuerda que tenemos disponibles nuestras clases Wrapper que nos ayudan a convertir datos primitivos a objetos.

Los collections se diferencian de los arrays en que su tamaño no es fijo y por el contrario es dinámico.

A continuación te muestro un diagrama de su composición:



Como podemos observar el elemento más alto es la interfaz **Collection**, para lo cual, partiendo de su naturalidad de interface, entendemos que tiene una serie de métodos

"básicos" dónde su comportamiento será definido a medida que se vaya implementando en más elementos. De ella se desprenden principalmente las interfaces **Set** y **List.**

La interface **Set** tendrá las siguientes características:

Almacena objetos únicos, no repetidos.
La mayoría de las veces los objetos se almacenarán en desorden.
No tenemos índice.

La interface **List** tiene éstas características:

Puede almacenar objetos repetidos.
Los objetos se almacenan en orden secuencial.
Tenemos acceso al índice.

Si seguimos analizando las familias tenemos que de Set se desprenden:

Clase HashSet
Interfaz SortedSet y de ella la clase TreeSet.

HashSet los elementos se guardan en **desorden** y gracias al mecanismo llamado hashing (obtiene un identificador del objeto) **permite almacenar objetos únicos.**

TreeSet almacena **objetos únicos**, y gracias a su estructura de árbol el **acceso* es sumamente **rápido.**

Ahora si analizamos la familia List, de ella se desprenden:

Clase **ArrayList** puede tener duplicados, no está sincronizada por lo tanto es más rápida
Clase **Vector** es sincronizada, los datos están más seguros pero es más lento.
Clase **LinkedList**, puede contener elementos duplicados, no está sincronizada (es más

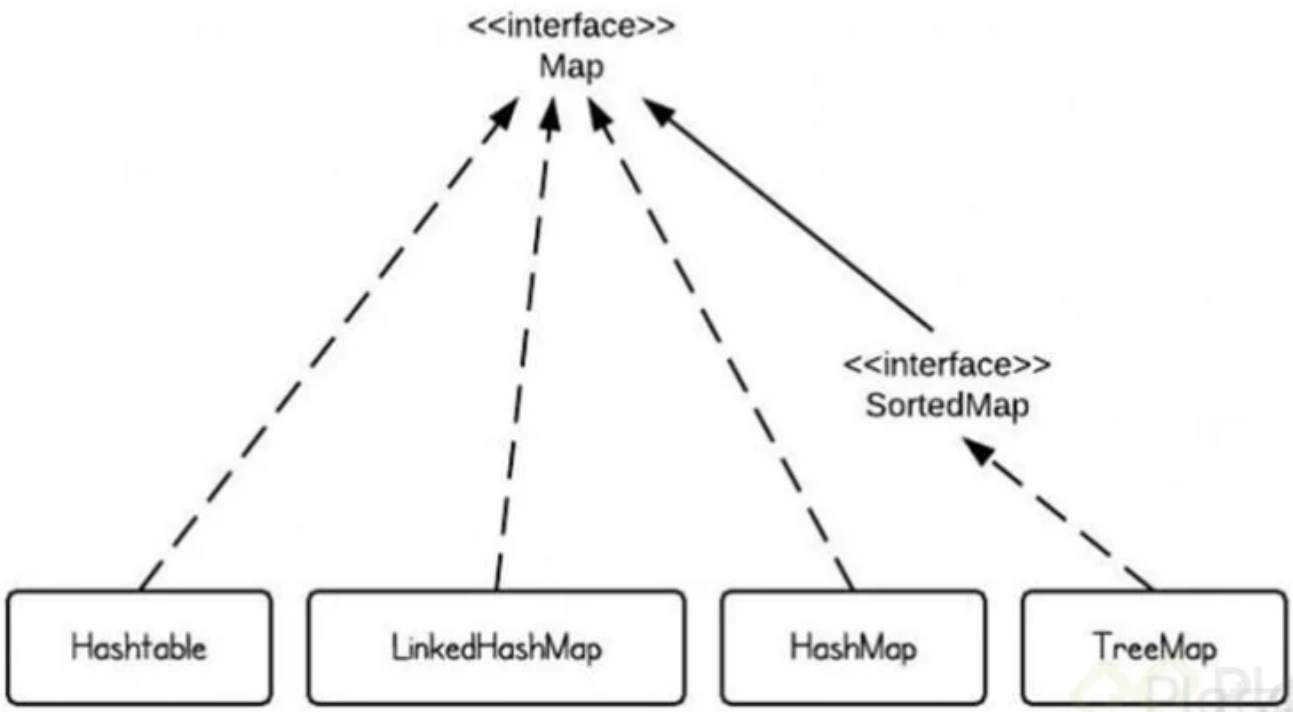
rápida) al ser una estructura de datos doblemente ligada podemos añadir datos por encima de la pila o por debajo.

fig- doubly linked list

Sigamos con Map

Lo primero que debes saber es que tiene tres implementaciones:

HashTable
LinkedHashMap
HashMap
SortedMap ➡️ TreeMap



La interfaz **Map** no hereda de la interfaz Collection porque representa una estructura de datos de Mapeo y no de colección simple de objetos. Esta estructura es más compleja, pues cada elemento deberá venir en pareja con otro dato que funcionará como la llave del elemento.

Map

Donde K es el key o clave
Donde V es el value o valor

Podemos declarar un map de la siguiente forma:

```
Map<Integer, String> map = new HashMap<Integer, String>();
Map<Integer, String> treeMap = new TreeMap<Integer, String>();
Map<Integer, String> linkedHashMap = new LinkedHashMap<Integer, String>();
```

Como observas solo se puede construir el objeto con tres elementos que implementan de ella: **HashMap**, **TreeMap** y **LinkedHashMap** dejando fuera HashTable y SortedMap. SortedMap estará fuera pues es una interfaz y HashTable ha quedado deprecada pues tiene métodos redundantes en otras clases. Mira la funcionalidad de cada uno.

Como te conté hace un momento Map tiene implementaciones:

HashMap: Los elementos no se ordenan. No aceptan claves duplicadas ni valores nulos.
LinkedHashMap: Ordena los elementos conforme se van insertando; provocando que las búsquedas sean más lentas que las demás clases.
TreeMap: El Mapa lo ordena de forma “natural”. Por ejemplo, si la clave son valores enteros (como luego veremos), los ordena de menos a mayor.

Para iterar alguno de estos será necesario utilizar la interface **Iterator** y para recorrerlo lo haremos un bucle while así como se muestra:

Para HashMap

```
// Imprimimos el Map con un Iterador
Iterator it = map.keySet().iterator();
while(it.hasNext()){
  Integer key = it.next();
  System.out.println("Clave: " + key + " -> Valor: " + map.get(key));
}
```

Para LinkedHashMap

```
// Imprimimos el Map con un Iterador
Iterator it = linkedHashMap.keySet().iterator();
while(it.hasNext()){
  Integer key = it.next();
  System.out.println("Clave: " + key + " -> Valor: " + linkedHashMap.get(key));
}
```

Para TreeMap

```
// Imprimimos el Map con un Iterador
Iterator it = treeMap.keySet().iterator();
while(it.hasNext()){
  Integer key = it.next();
  System.out.println("Clave: " + key + " -> Valor: " + treeMap.get(key));
}
```

Ahora lee esta lectura y en la sección de tutoriales cuéntanos en tus palabras cómo funciona **Deque.**