Contenedores

Programación I

Grado en Ingeniería Informática

MDR, JCRdP y JDGD

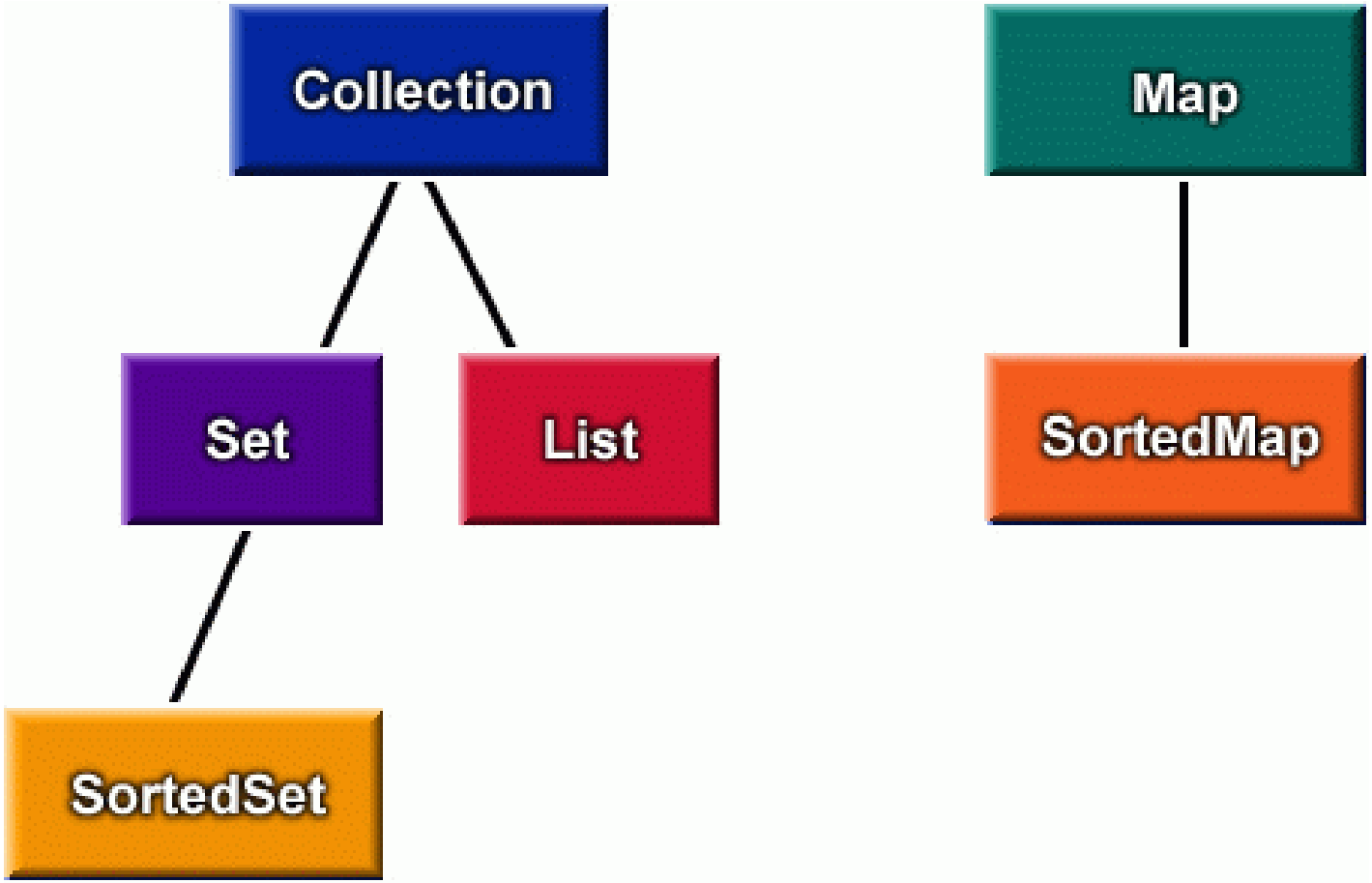
Contenedores

- Los contenedores o colecciones son objetos que almacenan otros objetos
- Los contenedores se usan para almacenar, recuperar, manipular datos y transmitirlos de un método a otro
- Los contenedores, generalmente, representan elementos que forman un grupo natural, como un conjunto de objetos gráficos, una carpeta de correo o un listín telefónico
- Un ejemplo de contenedor sencillo son los vectores
- Java 2 suministra un marco unificado para representar y manipular colecciones de datos en **java.util**

Contenedores genéricos

- Desde la versión 5 de Java las clases e interfaces contenedoras son genéricas
- Al definir el contenedor se establece qué tipo de dato va a almacenar
- Formato: NombreClase<T>, donde el nombre de la clase es el nombre del contenedor y T es el tipo de datos a almacenar
- El tipo de datos sólo puede ser una clase, no puede ser un tipo básico
- Tiene la ventaja de que no requiere conversiones como cuando se usaban contenedores de Object

Jerarquía de interfaces



Collection<T> (1/2)

Método	Descripción
`boolean add(T e)`	Añade un objeto
`boolean addAll(Collection<T> c)`	Añade los objetos contenidos en la colección
`boolean remove(T e)`	Extrae un objeto si existe
`boolean removeAll(Collection<T> c)`	Extrae los objetos contenidos en la colección
`void clear()`	Vacía la colección

Collection<T> (2/2)

Método	Descripción
`int size()`	Devuelve el número de elementos
`boolean contains(T e)`	Devuelve si contiene el objeto
`boolean containsAll(Collection<T> c)`	Devuelve si contiene todos los objetos de la colección
`Iterator<T> iterator()`	Devuelve un iterador sobre el contenedor

Ejemplo de Collection<T>

```
import java.util.*;
public class Coleccion {
  public static void main(String[] args) {
       Collection<String> c = new TreeSet<String>();
       c.add("lunes");
       c.add("martes");
       c.add("miércoles");
       System.out.println("El contenedor tiene " + c.size()
                          + " elementos");
       if (c.contains("martes"))
              System.out.println("El contenedor almacena martes");
       if (c.remove("martes"))
              System.out.println("Se ha eliminado martes");
       System.out.println("El contenedor tiene " + c.size()
                          + " elementos");
  }
}
```

Iteradores

- Los iteradores son objetos asociados a contenedores cuya finalidad es recorrer los elementos almacenados en el contenedor
- Según el caso, el iterador puede no solo acceder a los datos sino modificar, eliminar o añadir datos al contenedor
- No se permiten modificaciones externas durante el recorrido del contenedor por un iterador
- Los iteradores y contenedores con tipo pueden manejarse sin tipo.
- Equivalen a verlos como si almacenacen referencias a Object. Se consigue definiendo referencias sin el "<>".

Interface Iterator<T>

Método	Descripción
`boolean hasNext()`	Devuelve si existe siguiente
`T next()`	Devuelve el siguiente y avanza
`void remove()`	Borra del contenedor el último elemento accedido

Ejemplo de Iterator

```
import java.util.*;
public class Iterador {
    static void muestra(Iterator i) {
        while (i.hasNext()) System.out.print("=>" + i.next());
        System.out.println();
    }
    public static void main(String[] args) {
        Collection<String> c = new LinkedList<String>();
        c.add("lunes");
        c.add("martes");
        c.add("miércoles");
        System.out.println("El contenedor tiene "
                           + c.size() + " elementos");
        muestra(c.iterator());
        if (c.contains("martes"))
            System.out.println("El contenedor tiene almacenado martes");
        if (c.remove("martes"))
            System.out.println("Se ha eliminado martes");
        System.out.println("El contenedor tiene "
                           + c.size() + " elementos");
        muestra(c.iterator());
    }
}
```

Set y SortedSet

- Los **Set** tienen todas las características de las **Collection** pero, además, no contienen elementos repetidos
- Los **SortedSet** son **Set** que además mantienen los datos ordenados ascendentemente
- Para poder almacenar objetos en un **SortedSet** deben implementar la interfaz **Comparable** o añadir un comparador en el momento de creación
- Para implementar Comparable<T> se debe crear el método `int compareTo(T obj)` que
  - si actual > obj devuelve un número positivo
  - si actual = obj devuelve 0
  - si actual < obj devuelve un número negativo

Ejemplo de Comparable<T>

```
public class Pareja implements Comparable<Pareja>{
    int a, b;
    public Pareja(int a, int b){
        this.a = a;
        this.b = b;
    }
    @Override
    public int compareTo(Pareja o){
        if (a > o.a) return 1;
        if (a < o.a) return -1;
        if (b > o.b) return 1;
        if (b < o.b) return -1;
        return 0;
    }
}
```

Interface SortedSet<T>

Método	Descripción
`T first()`	Devuelve el primer elemento
`T last()`	Devuelve el último elemento
`SortedSet<T> headSet(T e)`	Devuelve un subconjunto ordenado con los menores que e
`SortedSet<T> tailSet(T e)`	Devuelve un subconjunto ordenado con los mayores que e
`SortedSet<T> subSet(T a, T b)`	Devuelve un subconjunto ordenado con los mayores o iguales que a y menores que b

Ejemplo de SortedSet<T>

```
import java.util.*;
public class Conjunto {
   static void muestra(Collection<String> c) {
         for(String v: c) System.out.print("=>" + v);
         System.out.println();
   }
   public static void main(String[] args) {
         SortedSet<String> s = new TreeSet<String>(); // Solo TreeSet
         s.add("lunes");s.add("martes");s.add("miércoles");
         s.add("jueves");s.add("viernes");s.add("sábado");
         muestra(s);
         System.out.println(s.contains("martes")?"martes está":"martes no está");
         System.out.println(s.remove("martes")?"martes eliminado":"error");
         s.add("domingo");
         System.out.println("El conjunto tiene " + s.size() + " elementos");
         muestra(s);
         muestra(s.tailSet("martes"));
   }
}
```

Interfaz Comparator<T>

- Requiere la implementación del método **`int`** `compare(T a, T b)` que tiene el mismo esquema de funcionamiento que `compareTo` pero recibiendo los dos datos a comparar en los parámetros
- La clase que implementa esta interfaz no tiene que ser T y, normalmente, sólo tiene la función de crear un objeto que sirva de comparador de T's.
- Se usa pasando un objeto de este tipo al constructor del contenedor ordenado o al método que ordena: (**Arrays.sort(T[] a, Comparator<T> c)**)

Ejemplo de Comparator<T>

```
import java.util.*;
public class Conjunto {
    public static class OrdenInversoString implements Comparator<String>{
       public int compare(String a, String b){ return -a.compareTo(b);}
    }
    ...
    public static void main(String[] args) {
          SortedSet<String> s = new TreeSet<String>(new OrdenInversoString());
          s.add("lunes");s.add("martes");s.add("miércoles");
          s.add("jueves");s.add("viernes");s.add("sábado");
          muestra(s);
          s.add("domingo");
          System.out.println("El conjunto tiene " + s.size() + " elementos");
          muestra(s);
          muestra(s.tailSet("martes"));
          ...
          List<String> l = new ArrayList<String>();
          l.add("enero"); l.add("febrero"); l.add("marzo");
          Collections.sort(l, new OrdenInversoString());
          muestra(l);
    }
}
```

List

- List añade una propiedad posicional a los elementos que contiene
- Los elementos se sitúan en posiciones que van desde **0 hasta size()-1**
- A las operaciones de añadir y eliminar elementos se les puede añadir un parámetro que indica en qué posición se realiza la operación. Por omisión se opera al final de la lista
- Se añaden operaciones de acceso a los elementos según posición
- Como iterador, se tiene la posibilidad de obtener un **ListIterator** que añade funcionalidad al Iterator normal

Interface List<T>

Método	Descripción
`boolean add(int i, T e)`	Añade e en la posición i desplazando a la derecha los elementos en la posición i y superiores
`boolean addAll(int i, Collection<T> c)`	Añade los objetos de c a partir de la posición i desplazando los elementos en las posiciones i y superiores
`boolean remove(int i)`	Extrae el elemento de la posición i desplazando los elementos de posiciones superiores a su izquierda

Interface List<T>

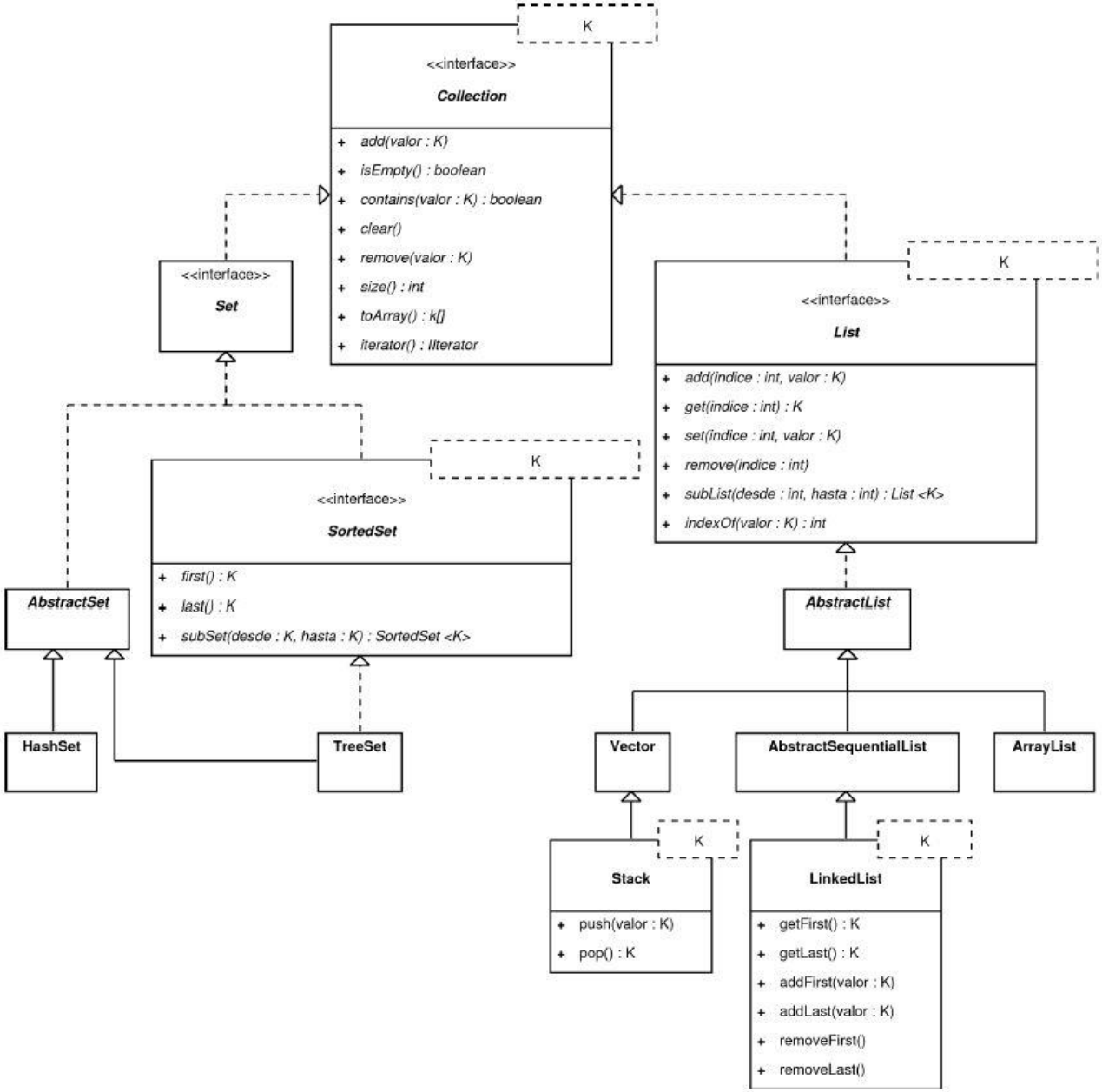
Método	Descripción
`T get(int i)`	Devuelve el objeto en la posición i
`T set(int i, T e)`	Sustituye el objeto en la posición i por e y devuelve el sustituido
`void clear()`	Vacía la colección
`int indexOf(T e)`	Da la posición de e ó -1, si no está
`ListIterator<T> listIterator()`	Devuelve un ListIterator sobre la lista

Ejemplo de List<T>

```
import java.util.*;
public class Lista {
   static void muestra(List lista) {
        for (int i = 0; i < lista.size(); i++)
                System.out.println(i + " => " + lista.get(i));
   }
   public static void main(String[] args) {
        List<String> l = new LinkedList<String>();
        l.add("lunes"); // Se puede hacer lo mismo que con Collection
        l.add("martes"); // add añade por el final
        l.add(1, "miércoles"); // Empiezan en 0, se añade antes de martes
        muestra(l);
        System.out.println("La list tiene " + l.size() + " elementos");
        if (l.contains("martes"))
           System.out.println("martes está en "+ l.indexOf("martes"));
        l.remove(0);
        muestra(l);
   }
}
```

ListIterator<T> (implementa Iterator<T>)

Método	Descripción
`boolean hasPrevious()`	Devuelve si existe anterior
`T previous()`	Devuelve el actual y retrocede
`int nextIndex()` `int previousIndex()`	Devuelve la posición del siguiente y anterior en la lista
`void set(T e)`	Sustituye el último objeto devuelto de la lista


K
<<interface>>
Collection
+ add(valor : K)
+ isEmpty() : boolean
+ contains(valor : K) : boolean
+ clear()
+ remove(valor : K)
+ size() : int
+ toArray() : k[]
+ iterator() : IIterator
<<interface>>
Set
K
<<interface>>
List
+ add(indice : int, valor : K)
+ get(indice : int) : K
+ set(indice : int, valor : K)
+ remove(indice : int)
+ subList(desde : int, hasta : int) : List <K>
+ indexOf(valor : K) : int
K
<<interface>>
SortedSet
+ first() : K
+ last() : K
+ subSet(desde : K, hasta : K) : SortedSet <K>
AbstractSet
AbstractList
HashSet
TreeSet
Vector
AbstractSequentialList
ArrayList
K
Stack
+ push(valor : K)
+ pop() : K
K
LinkedList
+ getFirst() : K
+ getLast() : K
+ addFirst(valor : K)
+ addLast(valor : K)
+ removeFirst()
+ removeLast()

Interfaces Map y SortedMap

- Un **Map** es una colección de parejas formadas por claves y valores, estableciendo una correspondencia entre cada clave y su valor
- Un mapa no puede contener claves repetidas
- El **Map** puede suministrar tres visiones de los datos: conjunto de claves, colección de valores, conjunto de pares clave-valor
- La idea principal de los mapas es poder acceder a los valores asociados a las claves por medio de éstas
- Un **SortedMap** tiene las mismas características que un Map pero, además, los datos están ordenados
- Este orden es el de las claves y se refleja en el recorrido con iteradores de las tres vistas del mapa

Interface Map<K,V> (1/2)

Método	Descripción
`V put(K clave, V valor)`	Añade un par clave-valor o actualiza uno existente. Devuelve el valor previo en la pareja existente previamente o null, si no existe
`V get(K clave)`	Devuelve el valor asociado con la clave o null, si no existe
`V remove(K clave)`	Elimina del mapa una pareja con clave y devuelve el valor de la pareja o null, si no existe
`int size()`	Devuelve el número de parejas en el mapa
`void clear()`	Borra todas las parejas del mapa

Interface Map<K,V> (2/2)

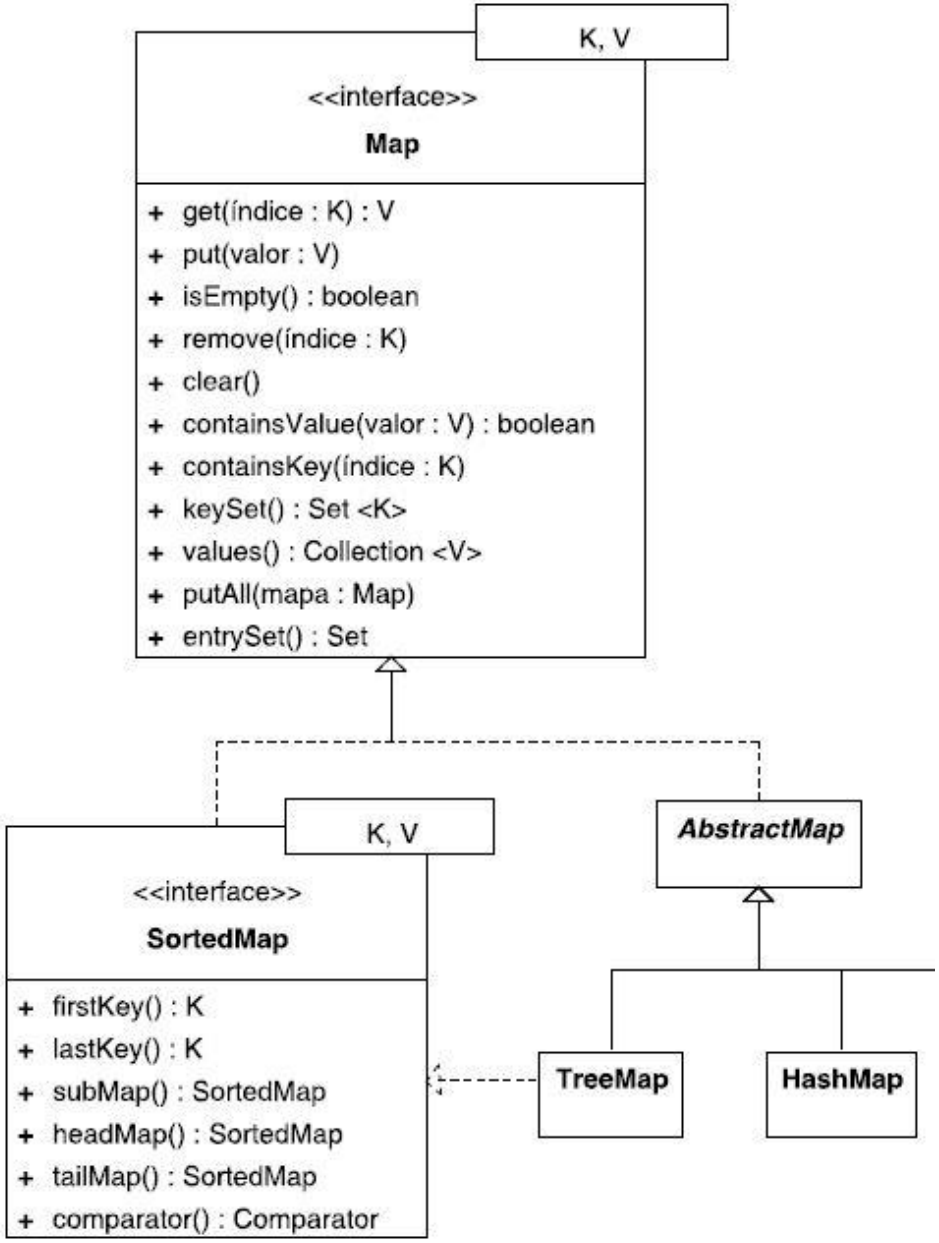
Método	Descripción
`Set<Entry<K,V>> entrySet()`	Devuelve el conjunto de parejas en el mapa en forma de Entry<K,V>
`Set<K> keySet()`	Devuelve el conjunto de claves en el mapa
`Collection<V> values()`	Devuelve la colección de valores en el mapa

Ejemplo de Map<K,V>

```
import java.util.*;
public class Correspondencia {
  static void muestra(Collection<Map.Entry<String, Integer>> c) {
        for (Map.Entry<String, Integer> e : c)
            System.out.println("clave: " + e.getKey() +
                               "\tvalor: " + e.getValue());
  }
  public static void main(String[] args) {
      Map<String, Integer> m = new HashMap<String, Integer>();
      m.put("enero", 31);m.put("febrero", 28);m.put("marzo", 31);
      m.put("abril", 30);m.put("mayo", 31);m.put("junio", 30);
      System.out.println("El map tiene " + m.size()+ " elementos");
      muestra(m.entrySet());
      if (m.get("abril") != null)
          System.out.println("abril está en la correspondencia");
      if (m.remove("abril") != null)
          System.out.println("Se ha eliminado abril");
      System.out.println("El map tiene " + m.size()+ " elementos");
      muestra(m.entrySet());
    }
}
```

Interface SortedMap<K,V>

Método	Descripción
`K firstKey()`	Devuelve la primera clave del mapa
`K lastKey()`	Devuelve la última clave del mapa
`SortedMap<K,V> headMap(K e)`	Devuelve los pares con claves menores que e
`SortedMap<K,V> tailMap(K e)`	Devuelve los pares con claves mayores que e
`SortedMap<K,V> subMap(K e1, K e2)`	Devuelve los pares con claves mayores o iguales que e1 y menores que e2


K, V
<<interface>>
Map
+ get(índice : K) : V
+ put(valor : V)
+ isEmpty() : boolean
+ remove(índice : K)
+ clear()
+ containsValue(valor : V) : boolean
+ containsKey(índice : K)
+ keySet() : Set <K>
+ values() : Collection <V>
+ putAll(mapa : Map)
+ entrySet() : Set
K, V
<<interface>>
SortedMap
+ firstKey() : K
+ lastKey() : K
+ subMap() : SortedMap
+ headMap() : SortedMap
+ tailMap() : SortedMap
+ comparator() : Comparator
AbstractMap
TreeMap
HashMap

Implementaciones de contenedores (1 /2)

- **LinkedList<T>**: lista doblemente enlazada que implementa List<T>. Utilizable cuando se requiere almacenar secuencias de datos donde las posiciones importan y se modifica por el principio o por el final o, mediante iterador
- **ArrayList<T>**: vector dinámico que implementa List<T>. Utilizable cuando se requiere almacenar secuencias de datos donde la posiciones importan. Se requiere acceso por posición rápido y se modifica sólo por el final, o se modifican elementos

Implementaciones de contenedores (2/2)

- **TreeSet<T>** y **TreeMap<K,V>**: árbol equilibrado que implementan SortedSet<T> y SortedMap<K,V>. Se usa cuando se requiere que los datos estén siempre ordenados, con frecuentes operaciones de inserción y extracción
- **HashSet<T>** y **HashMap<K,V>**: tabla de dispersión que implementan Set<T> y Map<K,V>. Usada cuando se requiere alto rendimiento en inserción, búsqueda y extracción pero que los datos no mantengan un orden
- **LinkedHashSet<T>** y **LinkedHashMap<K,V>**: igual que HashSet<T> y HashMap<K,V> pero con recorrido en el orden de inserción

Requisitos de las implementaciones

- Todos los contenedores requieren la correcta implementación de `boolean equals()` para el correcto funcionamiento de `contains(T e)` y `remove(T e)`
- `TreeSet<T>` y `TreeMap<K,V>` requieren que T y K implementen `Comparable<T>`,o disponer de un objeto de una clase que implemente `Comparator<T>` y pasarlo en el constructor para establecer el orden de T
- `HashSet<T>`, `HashMap<K,V>`, `LinkedHashSet<T>` y `LinkedHashMap<K,V>` requieren una redefinición adecuada de `int hashCode()`

int hashCode()

- Devuelve un entero que representa el objeto
- Está definido en la clase Object
- Las clases predefinidas del lenguaje lo implementan adecuadamente
- Las clases creadas por los programadores deben redefinirlo si desean que los contenedores "hash" funcionen adecuadamente con ellas
- Si dos objetos son iguales (según equals()) su hashCode() debe dar igual
- Una forma sencilla de implementar hasCode() es devolver la suma de los hashCode() de los atributos que intervienen en equals()

Rendimiento (con n=size())

	LinkedList	ArrayList	HashSet HashMap LinkedHash	TreeSet TreeMap
add(T)	O(1)	O(1)*	O(1)	O(log n)
remove(T) contains(T)	O(n)	O(n)	O(1)	O(log n)
remove() (con iterador)	O(1)	O(n)	O(1)	O(log n)
add(int i,T e) remove(int i)	O(n)	O(n)	N/A	N/A
set(int i, T e) get(int i)	O(n)	O(1)	N/A	N/A
put(K,V) get(K)	N/A	N/A	O(1)	O(log n)

* en coste amortizado

Combinaciones de interfaces e implementaciones posibles

		Implementaciones			
		Tabla de dispersión	Vector dinámico	Árbol equilibrado	Lista enlazada
Interfaces	Set	HashSet LinkedHashSet		TreeSet	
	List		ArrayList		LinkedList
	Map	HashMap LinkedHashMap		TreeMap	
	SortedSet			TreeSet	
	SortedMap			TreeMap	

Bibliografía

- **Trail: Collections** http://docs.oracle.com/javase/tutorial/collections/index.html
- Diseñar y programar todo es empezar: Una introducción a la Programación Orientada a Objetos usando UML y Java. (José F.Vélez y otros. Universidad Rey Juan Carlos)