

– Escuela Java

Colecciones Parte 2



TOGETHER. FREE YOUR ENERGIES

Objetivo

- Lograr que aprendamos a programar en Java, hasta el punto de construir aplicaciones reales en la tecnología.



Agenda de hoy




- ✓ Parte 1: Herramientas de base y J2EE

Módulo 2: Java SE

- ✓ Colecciones – Part II

Colecciones – Part II I

Collections CON duplicados

-  Implementan la interface List, que a su vez también hereda de Collection.
-  Posibilita el acceso a los elementos a través de índice
-  Algunas de las clases que implementan esta interface son:

ArrayList

Ofrece un tiempo de acceso óptimo cuando este tipo de acceso es aleatorio.

LinkedList

El tiempo de acceso es óptimo si se suele acceder al principio o al final de la lista.

Vector

Es igual que ArrayList sólo que se puede sincronizar en multitarea. Debido a esto baja notablemente su rendimiento.

Colecciones – Part II II

Collections CON duplicados

Métodos de acceso

Object get (int index);

Object set (int index, Object element)

void add (int index, Object element)

Object remove(int index)

boolean addAll (int index, Collection c)

Colecciones – Part II III

Collections CON duplicados

Métodos para búsquedas

```
int indexOf(Object o)
```

```
int lastIndexOf(Object o)
```

Métodos para obtener sublistas

```
List subList(int from, int to)
```

Colecciones – Part II IV

Ejemplo ArrayList

```
public static void main(String[] args) {  
    List<String> ciudades = new ArrayList<String>();  
  
    ciudades.add("Buenos Aires");  
    ciudades.add("Tucumán");  
    ciudades.add("Rosario");  
    ciudades.add(1, "Paraná");  
    ciudades.add("Buenos Aires");  
    ciudades.add("Bariloche");  
  
    Iterator<String> it = ciudades.iterator();  
  
    while (it.hasNext()) {  
        System.out.println("Ciudad:" + it.next());  
    }  
}
```

```
Ciudad:Buenos Aires  
Ciudad:Paraná  
Ciudad:Tucumán  
Ciudad:Rosario  
Ciudad:Buenos Aires  
Ciudad:Bariloche
```

Colecciones – Part II V

Ejemplo LinkedList

```
public static void main(String[] args) {  
    List<String> ciudades = new LinkedList<String>();  
  
    ciudades.add("Buenos Aires");  
    ciudades.add("Tucumán");  
    ciudades.add("Rosario");  
    ciudades.add(1, "Paraná");  
  
    ciudades.add("Buenos Aires");  
    ciudades.add("Bariloche");  
  
    Iterator<String> it = ciudades.iterator();  
  
    while (it.hasNext()) {  
        System.out.println("Ciudad:" + it.next());  
    }  
}
```

```
Ciudad:Buenos Aires  
Ciudad:Paraná  
Ciudad:Tucumán  
Ciudad:Rosario  
Ciudad:Buenos Aires  
Ciudad:Bariloche
```


Colecciones – Part II VI

La interface Map

Aunque no hereda de la interface Collection, también se utiliza para tratar colecciones de objetos.

Representa colecciones de objetos con parejas de elementos clave – valor.

Algunas de las clases que implementan esta interface:

HashMap

Ofrece un tiempo de acceso óptimo cuando dicho acceso es aleatorio.

Hashtable.

Es la versión sincronizada de HashMap.

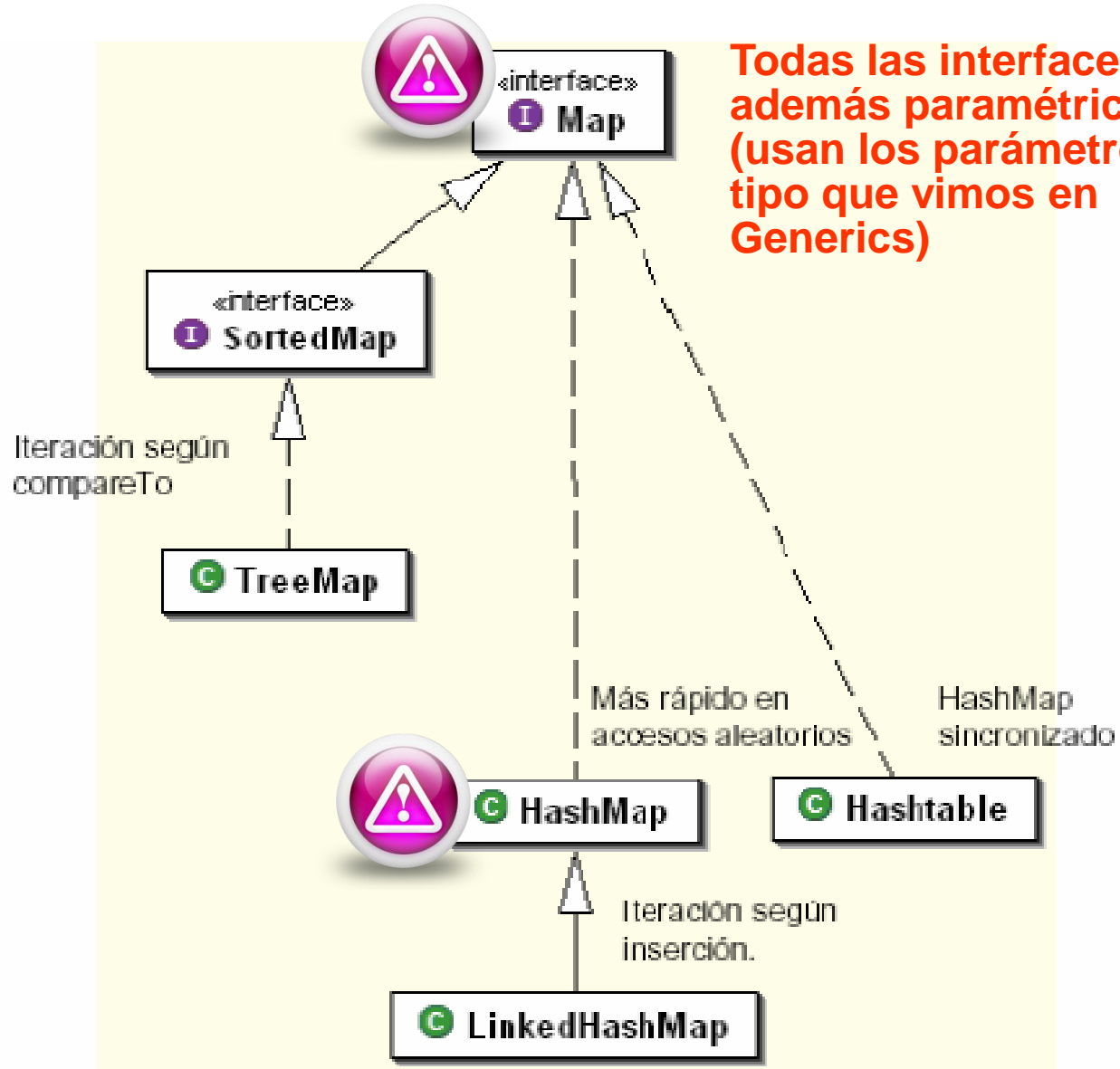
LinkedHashMap

Iteración según orden de inserción.

TreeMap.

Iteración según el método compareTo.

Colecciones – Part II VII



Todas las interfaces son además paramétricas (usan los parámetros de tipo que vimos en Generics)

Colecciones – Part II VIII

Métodos de la interface Map

Object put(Object clave, Object valor);

Object get(Object clave);

Object remove(Object clave);

boolean containsKey(Object key);

boolean containsValue(Object valor);

int size();

boolean isEmpty();

Colecciones – Part II IX

Ejemplo HashMap

```
public class PruebaPersona {
    public static void main(String[] args) {
        Persona persona1 = new Persona(' "Victor", "Fernandez ");

        Persona persona2 = new Persona(' "Ned"      , "Flanders ");

        Map<String, Persona> personas = new HashMap<String, Persona>();
        personas.put(persona1.getNombrePila(), persona1);
        personas.put(persona2.getNombrePila(), persona2);

        Set<String> claves = personas.keySet();
        Iterator<String> it = claves.iterator();
        while (it.hasNext()){
            String clave = it.next();
            Persona p = personas.get(clave);
            System.out.println(clave + ":" + p.getNombreApellidos());
        }
    }
}
```

Colecciones – Part II XIII

Métodos de la interface Map

```
void putAll(Map m);
```

```
void clear();
```

```
Set keySet();
```

```
Collection values();
```

Ejercicio




Ejercicio XIII de la guía



Colecciones – Part II XII

Nuevo bucle for versión 5.0

```
List<Persona> listaPersonas = new ArrayList<Persona>();  
listaPersonas.add(new Persona("Vity", "Victor Rodriguez", "viti@gmail.com"));  
listaPersonas.add(new Persona("Juan", "Juan Suárez", "jsuarez@gmail.com"));  
for (Persona p:listaPersonas){  
    System.out.println(p.getApellidos());  
}
```



En cada pasada del bucle for, la variable **p** es una persona de la colección **listaPersonas**

Colecciones – Part II XI

Qué colección utilizar?



ArrayList, HashMap y HashSet son las implementaciones “primarias” de List, Map y Set respectivamente.



Para que los elementos de la colección se puedan ordenar, es necesario que estos elementos implementen la interface Comparable.



Si utilizamos TreeMap, los objetos que utilicemos como clave deberán implementar Comparable.

Ingeniería Informática - Algoritmos

Puede haber múltiples algoritmos para un problema hay que saber cual conviene de acuerdo a la situación

	 <u>Insertion</u>	 <u>Selection</u>	 <u>Bubble</u>	 <u>Shell</u>	 <u>Merge</u>	 <u>Heap</u>	 <u>Quick</u>	 <u>Quick3</u>
 <u>Random</u>								
 <u>Nearly Sorted</u>								
 <u>Reversed</u>								
 <u>Few Unique</u>								

Ingeniería Informática - Complejidad

Cada algoritmo de ordenamiento tiene su complejidad temporal y espacial.

Que se utilizan para elegir cual es el mejor para el problema a resolver

Imágenes y gráficos de complejidad obtenidos de <https://www.bigocheatsheet.com/>

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
<u>Quicksort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
<u>Mergesort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Timsort</u>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Heapsort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
<u>Bubble Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Insertion Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Selection Sort</u>	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Tree Sort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
<u>Shell Sort</u>	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
<u>Bucket Sort</u>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
<u>Radix Sort</u>	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
<u>Counting Sort</u>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
<u>Cubesort</u>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$

Colecciones – Part II XIV

La interface








Comparable

- ➡ Cuando una clase implemente esta interface, estaremos indicando que sus objetos van a poder ser comparados y por lo tanto ordenados.
- ➡ Implementar esta interface (el método `compareTo`) nos permite:
 - ➡ Llamar a `Collections.sort()` y `Collections.binarySearch()`
 - ➡ Llamar a `Array.sort()` y `Array.binarySearch()`
 - ➡ Utilizar objetos como claves en los `TreeMap`
 - ➡ Utilizar objetos como elementos en los `TreeSet`

Colecciones – Part II XV

La interface Comparable

El método compareTo()

-  **x.compareTo(y)** Devuelve un entero que será:
 -  **-1** si x es menor que y.
 -  **1** si x es mayor que y.
 -  **0** si ambos objetos son iguales
-  **Conviene que el método equals sea consistente con compareTo.**

Colecciones – Part II XVI

La interface Comparable

Desde la versión 5.0 de Java podemos parametrizar la interface Comparable para indicarle qué objetos son los que vamos a comparar.

```
public class DVD implements Comparable<DVD> {

    String codigo;
    String titulo, director;
    List actores;

    public int compareTo(DVD o) {
        // Se ordenan atendiendo al valor numérico del código
        final int MENOR = -1;
        final int MAYOR = 1;
        final int IGUAL = 0;

        if (this == o) return IGUAL;

        int esteCodigo = new Integer(this.codigo);
        int otroCodigo = new Integer(o.codigo);
        if (esteCodigo < otroCodigo) return MENOR;
        if (esteCodigo > otroCodigo) return MAYOR;
        return IGUAL;
    }
}
```

Colecciones – Part II XVII

La interface Comparable

Ordenar colecciones

 **Implementamos, tal como vimos, la interface Comparable**

 **Esto nos permite insertar productos en un TreeSet y utilizar la clase Producto como clave en un TreeMap.**

 **Si utilizamos Collections.sort(listaProductos), se ordenará la colección listaProductos.**

 **Pero si queremos ordenar la lista de productos según otros criterios, esta solución no alcanza**

 **La solución es la interface Comparator.**

Colecciones – Part II XVIII

La interface Comparable

Ordenar colecciones



Para poder cambiar el criterio de ordenación, tendremos que crear ‘comparadores’.



Un ‘comparador’ es una clase que implementa la interface `Comparator` y que por tanto está obligada a implementar el método `compare()`.



Un ejemplo de comparador por edad para la clase `Persona`:

```
public class ComparadorEmail implements Comparator<Persona> {  
  
    public int compare(Persona p1, Persona p2) {  
        return p1.getEmail().compareTo(p2.getEmail());  
    }  
}
```

Colecciones – Part II XIX

La interface Comparable

Ordenar colecciones

Por cada distinto criterio de ordenación, crearemos una clase 'comparador'.

```
ArrayList<Persona> listaPersonas= new ArrayList<Persona>();  
//...  
//añadimos personas a la lista  
//...  
//Ordenamos la lista por email  
Collections.sort(listaPersonas,new ComparadorEmail());  
//Ordenamos la lista por apellidos  
Collections.sort(listaPersonas, new ComparadorApellidos());
```


Ejercicio



Ejercicio XIV de la guía



Muchas gracias!