Resumen Guia 5 parte II

Colecciones

Clases **Wrapper**

Las clases Wrapper proporcionan una forma de representar tipos de datos primitivos como objetos.

Son útiles cuando se necesita tratar los tipos primitivos como objetos, por ejemplo, cuando se desea almacenarlos en una colección o pasarlos como argumentos a métodos que requieren objetos en lugar de tipos primitivos.

También proporcionan métodos y utilidades para realizar conversiones entre tipos primitivos y objetos, así como operaciones y comparaciones específicas de cada tipo (definen compareTo).

Tipos de datos	
Primitivos	Objetos
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Algunas utilidades **Wrapper** valueOf()/parse() - Boxing/unboxing- compareTo()

valueOf() y parse() son métodos utilizados para convertir valores entre tipos primitivos y objetos Wrapper

Por ejemplo, Integer.valueOf("10") crea un objeto Integer con el valor 10.

Por ejemplo, Integer.parseInt("10") devuelve el valor 10 de tipo int. Este método es útil cuando se necesita convertir una cadena en un valor primitivo.

Boxing: Es el proceso automático de convertir un valor primitivo en su correspondiente objeto Wrapper.

Integer nro= 12; Es lo mismo que: Integer nro= new Integer(12);

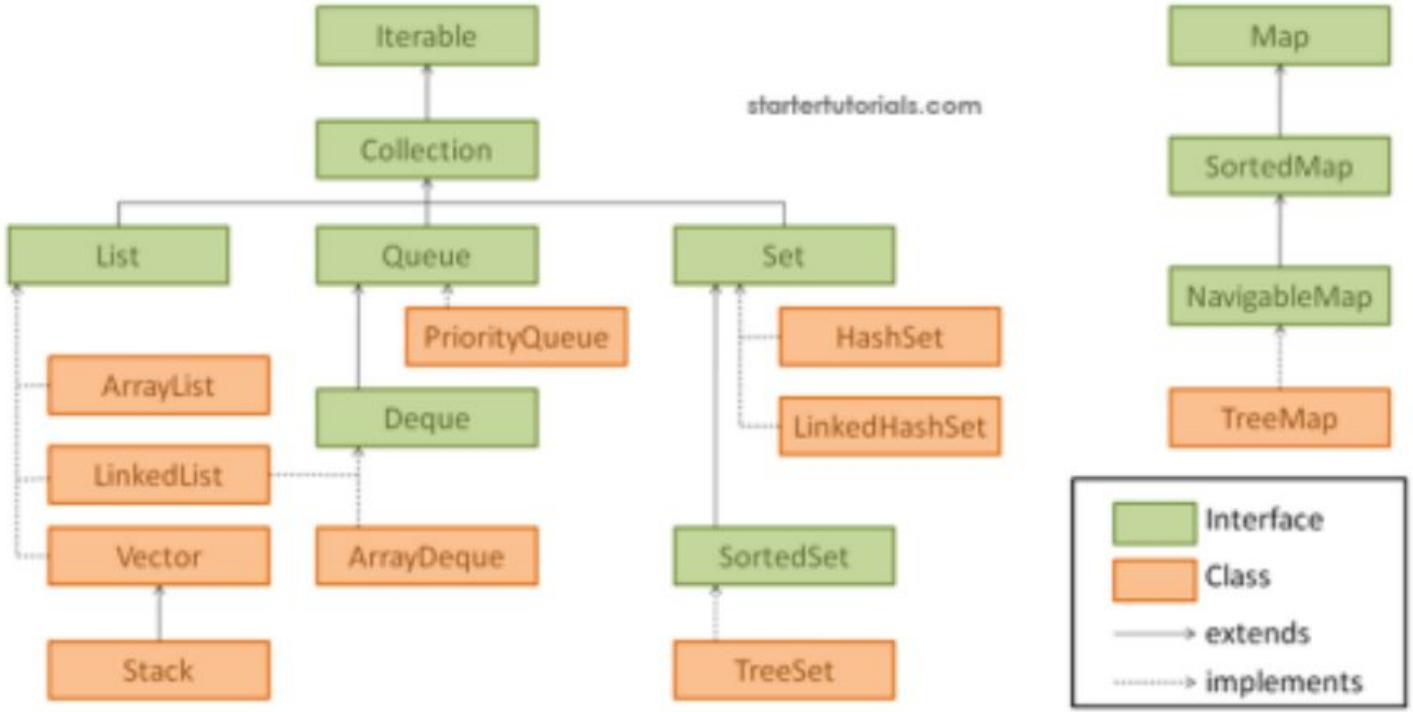
Unboxing: Es el proceso automático de convertir un objeto Wrapper en su correspondiente valor primitivo.

Integer nro=12; int otroNro= nro; Es lo mismo que: int otroNro= nro.intValue();

El método compareTo(). Este método se utiliza para comparar objetos de las clases envoltorio entre sí y determinar su orden relativo. **compareTo() está definido en la interfaz Comparable, que es implementada por las clases envoltorio.**

```
Integer num1 = 5;
Integer num2 = 10;
int resultado = num1.compareTo(anotherInteger: num2);

if (resultado < 0) {
    System.out.println(x: "num1 es menor que num2");
} else if (resultado == 0) {
    System.out.println(x: "num1 es igual a num2");
} else {
    System.out.println(x: "num1 es mayor que num2");
}
```

Jerarquía de Collection



Colecciones: Listas y Conjuntos

Ejemplo de un ArrayList de numeros:

```
ArrayList<Integer> numeros = new ArrayList();
```

Ejemplo de una LinkedList de números:

```
LinkedList<Integer> numeros = new LinkedList();
```

Ejemplo de una Vector de números:

```
Vector<Integer> numeros = new Vector();
```

Ejemplo de un HashSet de cadenas:

```
HashSet<String> nombres = new HashSet();
```

Ejemplo de un TreeSet de numeros:

```
TreeSet<Integer> numeros = new TreeSet();
```

Ejemplo de un LinkedHashSet de cadenas:

```
LinkedHashSet<String> frases = new LinkedHashSet();
```

ArrayList vs LinkedList vs Vector

ArrayList: Proporciona un acceso rápido a elementos por índice, ya que *se almacenan de manera contigua* en la memoria. Es menos eficiente para inserciones y eliminaciones en el medio de la lista, ya que puede requerir el desplazamiento de elementos contiguos.

LinkedList: Utiliza una lista enlazada para almacenar los elementos. Cada elemento contiene una referencia al siguiente elemento de la lista. Es *eficiente para operaciones de inserción y eliminación en el medio de la lista*, pero no es eficiente para acceder a elementos por índice.

Vector: Similar a ArrayList, utiliza un arreglo dinámico para almacenar elementos. Sin embargo, a diferencia de ArrayList, *los métodos de Vector están sincronizados, lo que lo hace seguro para operaciones concurrentes*, pero puede afectar su rendimiento en comparación con ArrayList.

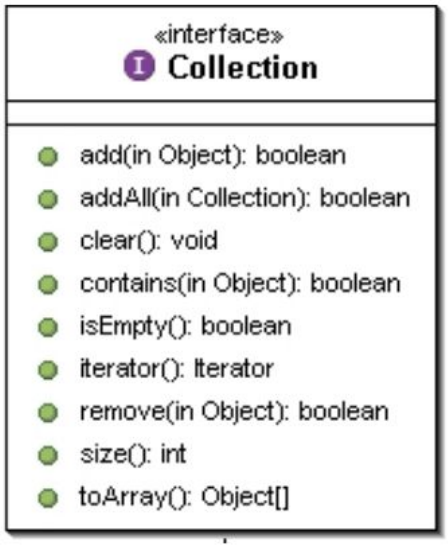
Listas y conjuntos dependen de Collection

- Añadir elementos a una lista o conjunto:

  numeros.**add(**elemento)

- Remover elementos de una lista o conjunto

  numeros.**remove**()



List vs set

Una lista permite agregar elementos en una posición específica. Tanto ArrayList, LinkedList, como Vector pueden almacenar elementos en una posición específica.

```
lista.add(1, "Elemento insertado");
```

```
vector.insertElementAt("Elemento insertado", 1);
```

Set Un conjunto es una estructura de datos que no conserva un orden específico de los elementos y no permite elementos duplicados. Los conjuntos implementan la interfaz Set y no proporcionan métodos para agregar elementos en posiciones específicas.

```
Set<String> conjunto = new HashSet<>();
```

```
conjunto.add("Elemento 1");
```

HashSet

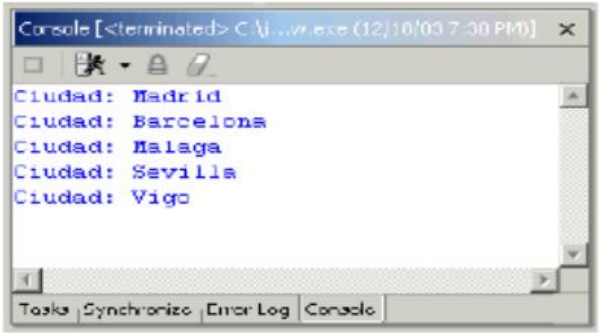
Un set implementado mediante una tabla hash; ofreciendo un rápido acceso aleatorio a sus posiciones, acelerando las búsquedas.

- Su orden de iteración es impredecible.

EJEMPLO CON HashSet

```
import java.util.*;

public class TestHashSet
{
  public static void main(String[] args)
  {
    HashSet ciudades = new HashSet();

    ciudades.add("Madrid");
    ciudades.add("Barcelona");
    ciudades.add("Malaga");
    ciudades.add("Vigo");
    ciudades.add("Sevilla");
    ciudades.add("Madrid"); // Repetido.

    Iterator it = ciudades.iterator();

    while(it.hasNext())
      System.out.println("Ciudad: " + it.next());
  }
}
```

Ejemplo



ES NECESARIO SOBREESCRIBIR EQUALS Y HASHCODE

Si no se sobrescribe se heredará la versión definida en la clase object, en la cual la Comparación de dos referencias devolverá true solamente si ambas apuntan al mismo objeto.

Si se sobrescribe equals() debemos sobrescribir hashcode también.

Teniendo en cuenta que dos objetos son iguales según equals() deben tener idénticos valores de hasCode().

TreeSet

Su orden de iteración depende de la implementación que los elementos hagan

de la interfaz: java.lang.Comparable.

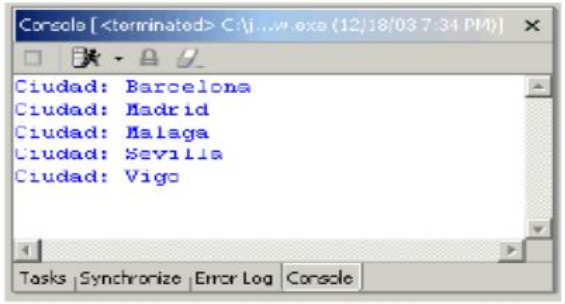
• Es más lento para buscar o modificar que un HashSet, pero mantiene los elementos ordenados.

• Asume que los elementos son comparables si no se les ha pasado un

comparator en el constructor.

EJEMPLO CON TreeSet

```
public class TestTreeSet
{
  public static void main(String[] args)
  {
    TreeSet ciudades = new TreeSet();

    ciudades.add("Madrid");
    ciudades.add("Barcelona");
    ciudades.add("Malaga");
    ciudades.add("Vigo");
    ciudades.add("Sevilla");
    ciudades.add("Madrid"); // Repetido.

    Iterator it = ciudades.iterator();

    while(it.hasNext())
      System.out.println("Ciudad: " + it.next());
  }
}
```



Nota: funciona porque java.lang.String implementa java.lang.Comparable.

Interfaz **Map**

- El interfaz Map no hereda del interfaz Collection.
- Representa colecciones con parejas de elementos: clave y valor.
- No permite tener claves duplicadas. Pero si valores duplicados.
- Para calcular la colocación de un elemento se basa en el uso del método:

```
public int hashCode();
```

Recorrer un mapa Map

```
HashMap<Integer, String> alumnos = new HashMap();
// Recorrer las dos partes del mapa
for (Map.Entry<Integer, String> entry : alumnos.entrySet()) {

        System.out.println("documento="+entry.getKey()+"nombre="+entry.getValue());

        // entry.getKey trae la llave y entry.getValue trae los valores del mapa

}
```

Sin Map.Entry:

```
// mostrar solo las llaves
for (Integer dni : alumnos.keySet()) {
        System.out.println("Documento: " + dni);
}
// mostrar solo los valores
for (String nombres : alumnos.values()) {
        System.out.println("Nombre: " + nombres);
}
```

Comparación entre colecciones

	Permiten repetidos	Ordenación por orden natural	Ordenación por introducción
ArrayList	si	no	si
LinkedList	si	no	si
HashSet	no	no	no
LinkedHashSet	no	no	si
TreeSet	no	si	no
HashMap	no	no	no
LinkedHashMap	no	no	si
TreeMap	no	si	no

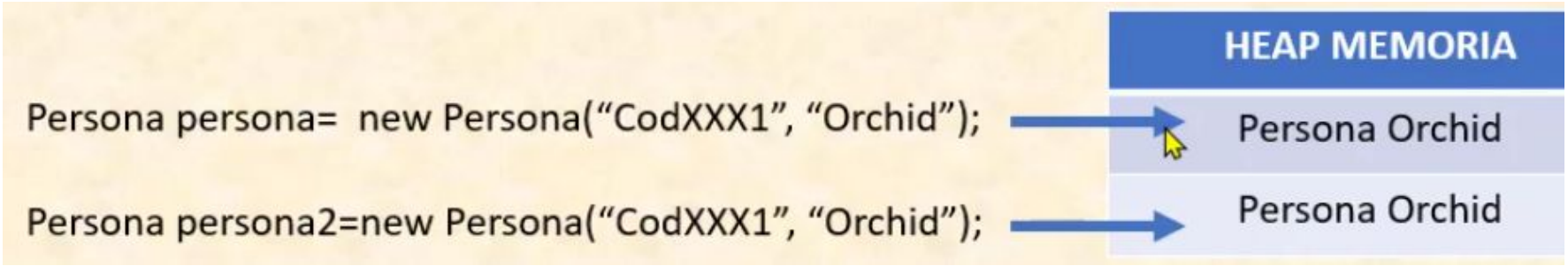
Iterator

Una vez que tienes un iterador, puedes utilizar los métodos proporcionados por la interfaz Iterator para recorrer la colección. Los métodos clave incluyen:

- **hasNext():** Comprueba si hay más elementos disponibles para recorrer en la colección.
- **next():** Devuelve el siguiente elemento de la colección y avanza el iterador al siguiente elemento.
- **remove():** Elimina el último elemento devuelto por el iterador de la colección. Esta operación es opcional y no todos los iteradores admiten la eliminación

```
Iterator<String> iterator = lista.iterator();
while (iterator.hasNext()) {
        String elemento = iterator.next();
        if (elemento.startsWith(prefix: "a")) {
        iterator.remove();
    }
}
```

Equals - Verificar si dos objetos son equivalentes



Debemos entonces sobreescribir el método equals en la clase Persona.

```
// Código que añadimos a la clase Persona. Sobreescritura del método equals ejemplo aprenderaprogramar.com
   public boolean equals (Object obj) {
      if (obj instanceof Persona) {
      Persona tmpPersona = (Persona) obj;
        if (this.nombre.equals(tmpPersona.nombre) && this.apellidos.equals(tmpPersona.apellidos) &&
 this.edad == tmpPersona.edad) { return true; } else { return false; }
   } else { return false; }
} //Cierre del método equals
```

Comparable vs Compare

Ordenamos Colecciones

COMPARABLE	COMPARATOR
La interfaz comparable permite solo una secuencia de clasificación.	La interfaz del comparador permite múltiples secuencias de clasificación.
paquete java.lang	paquete java.util
La interfaz comparable contiene solo un método public int compareTo (objeto obj);	Contiene dos métodos. public int compare (Objeto obj1, Objeto obj2) boolean equals (Object obj)
Collections.sort (List lst)	Colecciones.sort (Lista, Comparador)
La clase debe implementar a la Interfaz	No es necesario que la clase implemente la Interfaz

Comparable vs Comparator

- Retorna un < 0 Si el obj 1 antes que el obj2
- Retorna un = 0 Si están en la mismo orden
- Retorna > 0 Si el obj 1 después que el obj2

Obj1	Obj2	Retorno
1	10	-1
100	10	1
5	5	0
“A”	“Z”	-25
“J”	“B”	8
“Hola”	“Hola”	0
“Hola”	“Mundo”	-5

Implementando la Interfaz Comparable

```
public class Persona implements Comparable{
    private int edad;
    private String nombre, apellido;

    public Persona(int edad, String nombre, String apellido) {
        this.edad = edad;
        this.nombre = nombre;
        this.apellido = apellido;
    }
    public int getEdad() {...3 lines }
    public String getNombre() {...3 lines }
    public String getApellido() {...3 lines }

    @Override
    public String toString() {...3 lines }

    @Override
    public int compareTo(Object t) {
        Persona p= (Persona)t;
        return p.apellido.compareTo(p.getApellido());
    }
}
```

Desde *main* ordenando la colección

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class Ejemplo {
    public static void main(String arg[]){
        List<Persona> lista=new ArrayList();
        Persona p1=new Persona(10,"Ana","Sosa");
        Persona p2=new Persona(20,"Analia","Sosa");
        Persona p3=new Persona(30,"Carola","Fernandez");
        Persona p4=new Persona(40,"Juan","Perez");
        Persona p5=new Persona(50,"Pedro","Sosa");
        lista.add(p1);
        lista.add(p2);
        lista.add(p3);
        lista.add(p4);
        lista.add(p5);
        //foreach imprime lista sin orden
        System.out.println("--Lista sin ordenar--");
        for (Persona i: lista){
            System.out.println("Persona: "+i);
        }
        //lista ordenanda segun Comparable
        Collections.sort(lista);
        //foreach abreviado imprime lista ordenada
         System.out.println("--Lista ordenada--");
        lista.forEach(System.out::println);
    }
}
```

Implementando la Interfaz Comparator

```
package Listas;
import java.util.Comparator;
public class Ordenamientos {
    public static Comparator<Persona> porNombre=new Comparator<Persona>(){
        @Override
        public int compare(Persona t, Persona t1) {
          int orden= t.getNombre().compareTo(t1.getNombre());
          return orden;
        }
    };
    public static Comparator<Persona> porApellido=new Comparator<Persona>(){
        @Override
        public int compare(Persona t, Persona t1) {
          int orden= t.getApellido().compareTo(t1.getApellido());
          return orden;
        }
    };
    public static Comparator<Persona> porEdad=new Comparator<Persona>(){
        @Override
        public int compare(Persona t, Persona t1) {
          int orden= t.getEdad()-t1.getEdad();
          return orden;
        }
    };
}
```

Ordenando la colección desde *main* con Comparator

```
public class Ejemplo {
    public static void main(String arg[]){
        List<Persona> lista=new ArrayList();
        Persona p1=new Persona(10,"Ana","Sosa");
        Persona p2=new Persona(20,"Analia","Sosa");
        Persona p3=new Persona(30,"Carola","Fernandez");
        Persona p4=new Persona(40,"Juan","Perez");
        Persona p5=new Persona(50,"Pedro","Sosa");
        lista.add(p3);
        lista.add(p4);
        lista.add(p2);
        lista.add(p5);
        lista.add(p1);
        //foreach imprime lista sin orden
        System.out.println("--Lista sin ordenar--");
        lista.forEach(System.out::println);
        //lista ordenanda segun Comparable
        Collections.sort(lista,Ordenamientos.porNombre);
        //foreach abreviado imprime lista ordenada
        System.out.println("--Lista ordenada por nombre--");
        lista.forEach(System.out::println);
        Collections.sort(lista,Ordenamientos.porApellido);
        //foreach abreviado imprime lista ordenada
        System.out.println("--Lista ordenada por Apellido--");
        lista.forEach(System.out::println);
        Collections.sort(lista,Ordenamientos.porEdad);
        //foreach abreviado imprime lista ordenada
        System.out.println("--Lista ordenada por edad--");
        lista.forEach(System.out::println);
    }
```