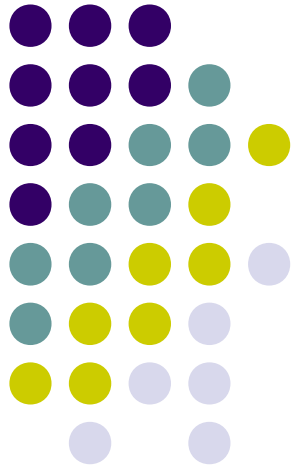
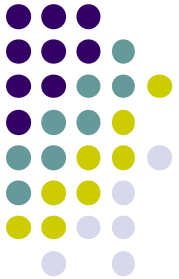
Colecciones de datos

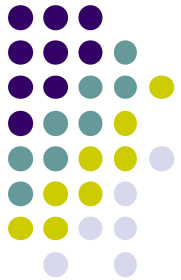




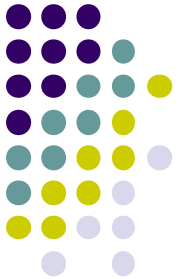
ÍNDICE

1. Colecciones
2. Arrays vs Colecciones
3. Interface java.util.Collection
4. Interface java.util.Iterator
5. Interface java.util.Set
6. Interface java.util.List
7. Interface java.util.Map
8. Importancia de equals y hashcode
9. Ordenación

COLECCIONES



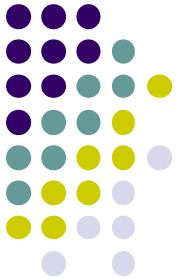
- Colección: Objeto que agrupa varios elementos en uno solo.
- Se utilizan para guardar y manipular datos así como transmitir información entre métodos.
- En Java tenemos un framework de colecciones:
  - Interfaces: representaciones abstractas de las colecciones que permiten usarlas sin conocer sus detalles.
  - Implementaciones: colecciones concretas.
  - Algoritmos: métodos que permiten realizar operaciones como búsquedas, ordenaciones, etc…



COLECCIONES

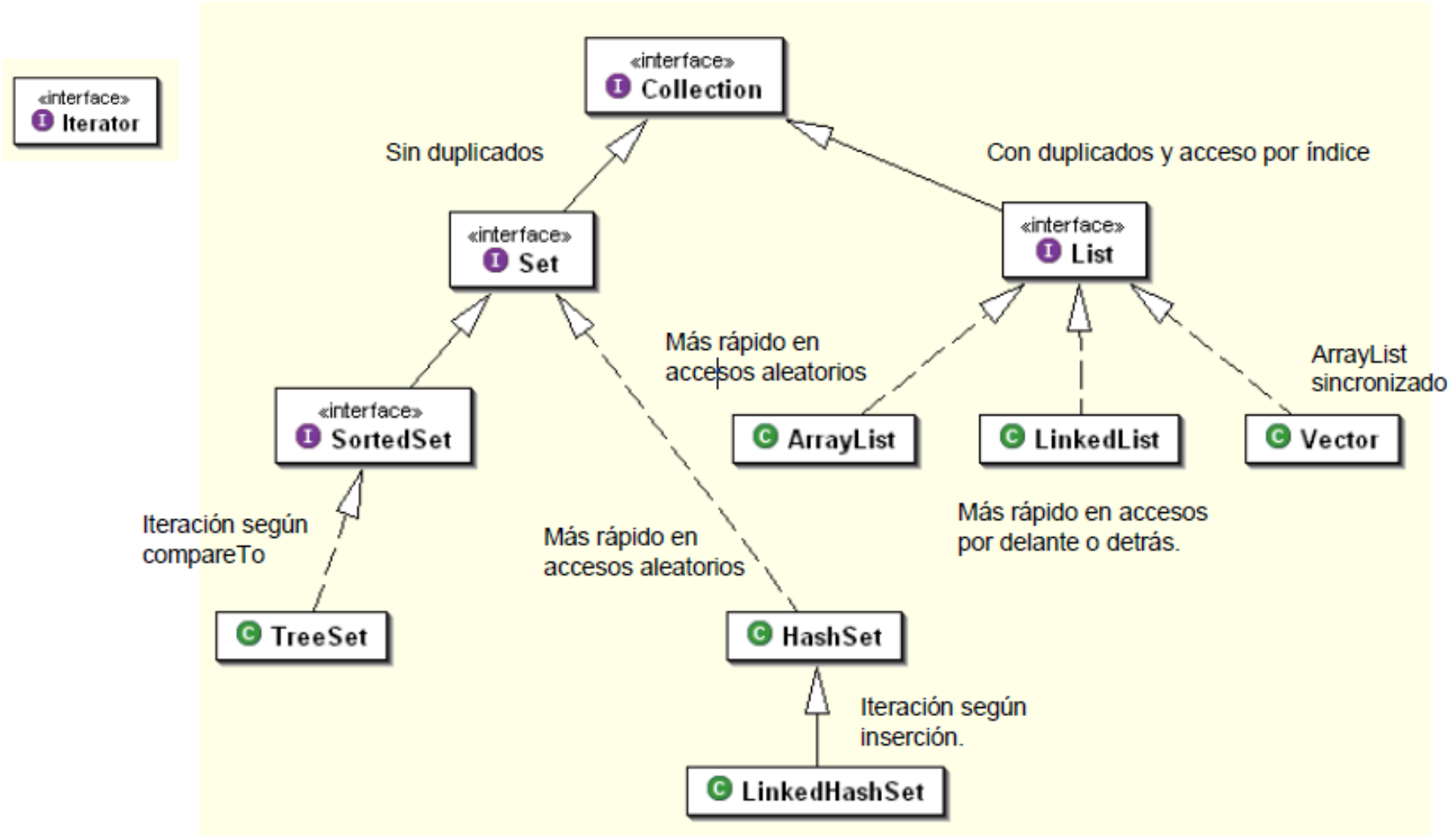
- Todas las colecciones se encuentran en el paquete java.util.*;
- java.util.Collection es la raíz de la jerarquía de las colecciones.
- Existen especializaciones que permiten:
  - Elementos duplicados o no
  - Ordenar los elementos o no
  - …
- Esta clase contiene la definición de todos los métodos genéricos que deben implementar las colecciones.

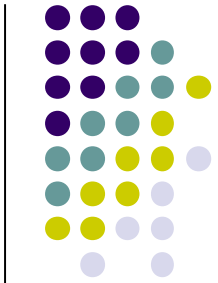
COLECCIONES



- Interfaces:
  - java.util.Collection
  - java.util.Iterator (Permite iterar sobre elementos de una colección)
  - java.util.Set (No permite duplicados)
  - java.util.List (Permite duplicados)
  - java.util.Map (Colecciones con parejas de elementos: clave y valor)

COLECCIONES

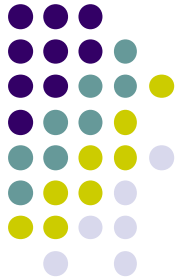




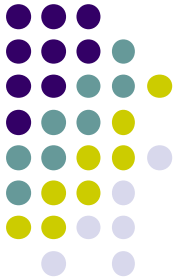
ÍNDICE

1. Colecciones
2. Arrays vs Colecciones
3. Interface java.util.Collection
4. Interface java.util.Iterator
5. Interface java.util.Set
6. Interface java.util.List
7. Interface java.util.Map
8. Importancia de equals y hashcode
9. Ordenación

ARRAYS VS COLECCIONES



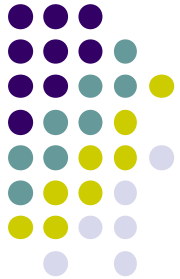
Arrays	Colecciones
Tamaño estático.	Tamaño dinámico.
Su tamaño se conoce mediante el atributo length.	Su tamaño se conoce mediante el método size().
Puede almacenar tanto tipos primitivos como objetos .	Solo puede almacenar tipos complejos.
	.



ÍNDICE

1. Colecciones
2. Arrays vs Colecciones
3. Interface java.util.Collection
4. Interface java.util.Iterator
5. Interface java.util.Set
6. Interface java.util.List
7. Interface java.util.Map
8. Importancia de equals y hashcode
9. Ordenación

INTERFAZ JAVA.UTIL.COLLECTION



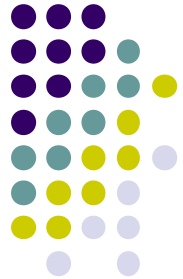
- Los métodos de este interfaz son:
  - Operaciones básicas
  - Operaciones masivas
  - Operaciones con arrays

- Nota: las colecciones no permiten el uso de tipos primitivos.
  - Si necesitamos trabajar con ellos habrá que hacer uso de los Wrappers de Tipos Primitivos

INTERFAZ JAVA.UTIL.COLLECTION

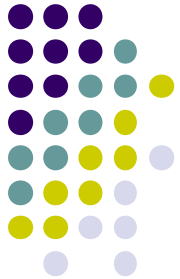
- Operaciones básicas:
  - int size(); // Número de elementos que contiene.
  - boolean isEmpty(); // Si no contiene ningún elemento.
  - boolean contains(Object element); // Si contiene ese elemento.
  - boolean add(Object element); // Añadir un elemento.
  - boolean remove(Object element); // Borrar un elemento.
  - Iterator iterator(); // Devuelve una instancia de Iterator.

INTERFAZ JAVA.UTIL.COLLECTION

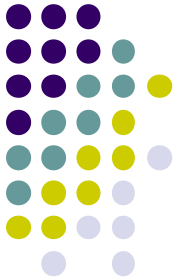


- Operaciones masivas:
  - boolean containsAll(Collection c); // Si contiene todos esos elementos.
  - boolean addAll(Collection c); // Añadir todos esos elementos.
  - boolean removeAll(Collection c); // Borrar todos esos elementos.
  - boolean retainAll(Collection c); // Borrar todos los elementos menos esos concretos.
  - void clear(); // Borrar todos los elementos.

INTERFAZ JAVA.UTIL.COLLECTION



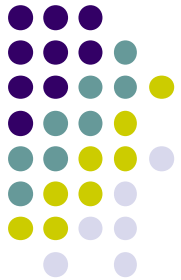
- Operaciones con arrays:
  - Object[] toArray(); // Devuelve un array con todos los elementos.
  - Object[] toArray(Object a[]); // Devuleve un array con todos los elementos. El tipo será el del array enviado.



ÍNDICE

- Colecciones
- Arrays vs Colecciones
- Interface java.util.Collection
- Interface java.util.Iterator
- Interface java.util.Set
- Interface java.util.List
- Interface java.util.Map
- Importancia de equals y hashcode
- Ordenación

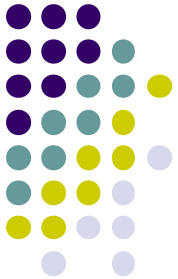
INTERFAZ JAVA.UTIL.ITERATOR



- El interfaz Iterator representa un componente que permite iterar sobre los elementos de una colección.

- Todas las colecciones ofrecen una implementación de Iterator por medio del método:
  - public Iterator iterator();

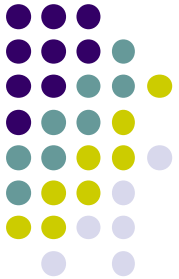
- Métodos:
  - boolean hasNext(); // Si tiene mas elementos.
  - Object next(); // Devuelve el primer elemento y se queda apuntando al siguiente.
  - void remove(); // Elimina el primer elemento y se queda apuntando al siguiente.



ÍNDICE

1. Colecciones
2. Arrays vs Colecciones
3. Interface java.util.Collection
4. Interface java.util.Iterator
5. Interface java.util.Set
6. Interface java.util.List
7. Interface java.util.Map
8. Importancia de equals y hashcode
9. Ordenación

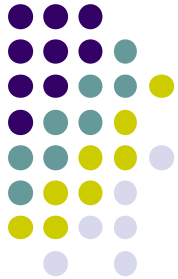
INTERFAZ JAVA.UTIL.SET



- Métodos:
  - De búsqueda
  - De subcolecciones

- Clases que implementan esta interfaz:
  - java.util.HashSet
  - java.util.LinkedHashSet
  - java.util.TreeSet

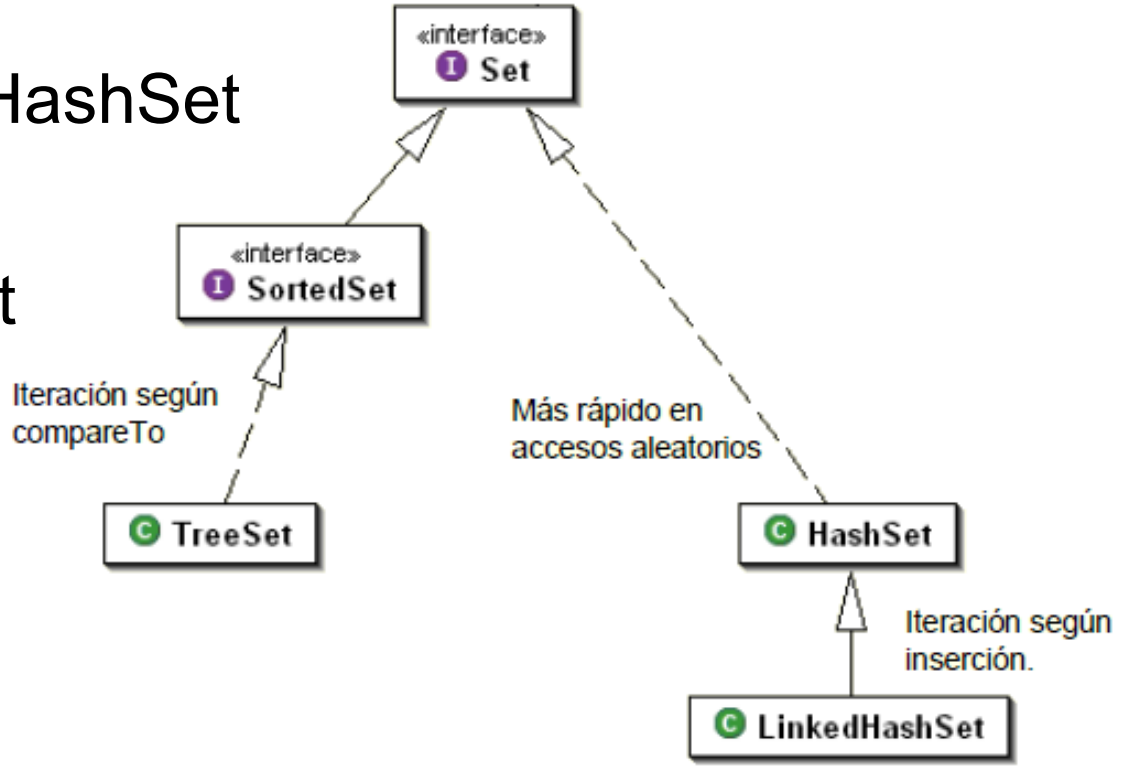
INTERFAZ JAVA.UTIL.SET



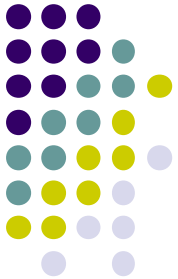
- El interfaz Set hereda del interfaz Collection. Pero no añade la definición de ningún método nuevo.
- Representa colecciones que no permiten tener elementos duplicados.
- Para saber si un elemento está duplicado, hace uso del método:
  - public boolean equals(Object o);
  - public int compareTo(Object o); en el caso de TreeSet
- Existen distintas implementaciones de este interfaz.

INTERFAZ JAVA.UTIL.SET

- Clases que implementan esta interfaz:
  - java.util.HashSet
  - java.util.LinkedHashSet
  - java.util.TreeSet

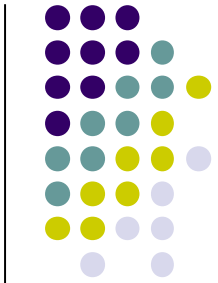


INTERFAZ JAVA.UTIL.SET



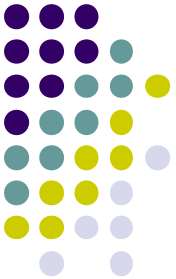
- Clase java.util.HashSet:
  - Ofrece el acceso mas rápido cuando dicho acceso es aleatorio.
  - Su orden de iteración es impredecible
- Clase java.util.LinkedHashSet:
  - Su orden de iteración es el orden de inserción
- Clase java.util.TreeSet:
  - Su orden de iteración depende de la implementación que los elementos hagan del interfaz java.lang.Comparable:
    - public int compareTo(Object o);

EJEMPLO HASHSET



```
import java.util.*;

public class Main {
    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 LINKEDHASHSET

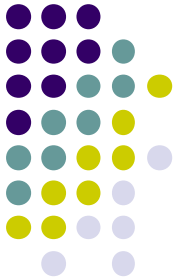


```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        LinkedHashSet ciudades = new LinkedHashSet();
        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 TREESET

```
import java.util.*;

public class Main {
    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());
        }
    }
}
```

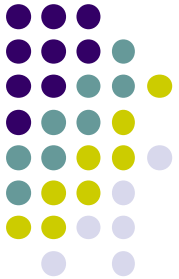
Nota: funciona porque java.lang.String implementa java.lang.Comparable



ÍNDICE

1. Colecciones
2. Arrays vs Colecciones
3. Interface java.util.Collection
4. Interface java.util.Iterator
5. Interface java.util.Set
6. Interface java.util.List
7. Interface java.util.Map
8. Importancia de equals y hashcode
9. Ordenación

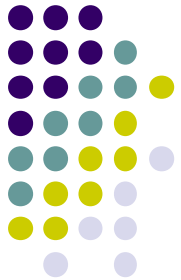
INTERFAZ JAVA.UTIL.LIST



- Métodos:
  - De acceso posicional
  - De búsqueda
  - De subcolecciones

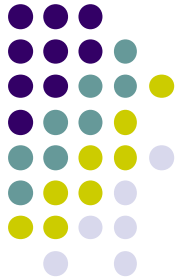
- Clases que implementan esta interfaz:
  - java.util.ArrayList
  - java.util.LinkedList
  - java.util.Vector

INTERFAZ JAVA.UTIL.LIST



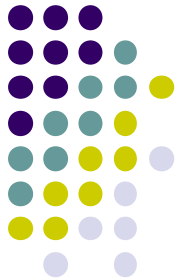
- El interfaz List hereda del interfaz Collection.
- Representa colecciones con elementos en secuencia. Es decir, con orden.
- Permite tener duplicados.
- Es accesible mediante índice, de manera que se puede:
  - Acceder a un elemento concreto de una posición.
  - Insertar un elemento en una posición concreta.
  - Eliminar el elemento que ocupa determinada posición

INTERFAZ JAVA.UTIL.LIST



- Acceso posicional:
  - Object get(int index); // Devuelve el elemento de esa posición.
  - Object set(int index, Object element); // Reemplaza el elemento de esa posición con ese elemento.
  - void add(int index, Object element); // Inserta ese elemento en esa posición.
  - Object remove(int index); // Elimina el elemento de esa posición
  - boolean addAll(int index, Collection c); // Inserta todos esos elementos en esa posición.

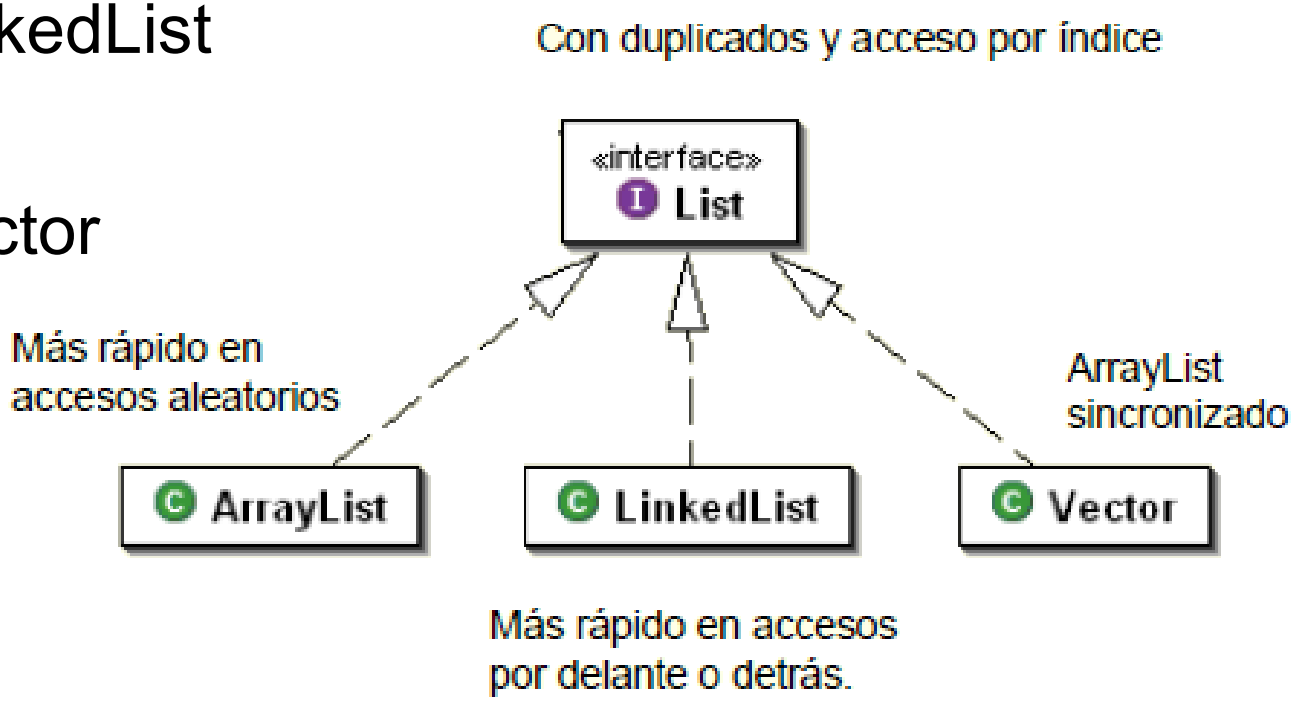
INTERFAZ JAVA.UTIL.LIST



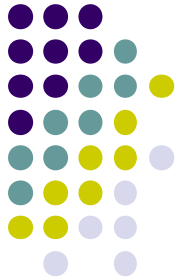
- Búsqueda:
  - int indexOf(Object o); // Devuleve la posición de la primera ocurrencia de ese elemento.
  - int lastIndexOf(Object o); // Devuleve la posición de la última ocurrencia de ese elemento
- Subcolecciones:
  - List subList(int from, int to); // Devuelve una lista con los elementos comprendidos entre ambas posiciones.

INTERFAZ JAVA.UTIL.LIST

- Clases que implementan esta interfaz:
  - java.util.ArrayList
  - java.util.LinkedList
  - java.util.Vector

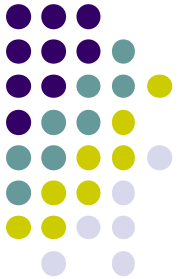


INTERFAZ JAVA.UTIL.LIST



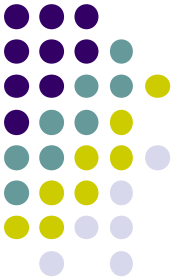
- Clase java.util.ArrayList
  - Ofrece un tiempo de acceso óptimo cuando dicho acceso es aleatorio.
- Clase java.util.LinkedList
  - Ofrece un tiempo de acceso óptimo cuando dicho acceso es para añadir o eliminar elementos del comienzo y final de la lista (típico para pilas).
- Clase java.util.Vector
  - Es como el ArrayList pero sincronizado lo que penaliza notablemente el rendimiento.
  - La sincronización es importante cuando más de un thread (hilo de ejecución) va a acceder a la colección.

EJEMPLO ARRAYLIST



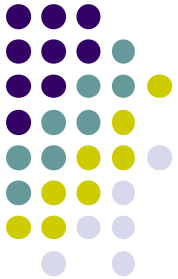
```
public class Main {

    public static void main(String[] args) {
        ArrayList ciudades = new ArrayList();
        ciudades.add("Madrid");
        ciudades.add("Barcelona");
        ciudades.add("Malaga");
        ciudades.add("Vigo");
        ciudades.add(1, "Sevilla");
        ciudades.add("Madrid"); // Repetido.
        Iterator it = ciudades.iterator();
        while (it.hasNext()) {
            System.out.println("Ciudad: " + it.next());
        }
    }
}
```

EJEMPLO ARRAYLIST (II)



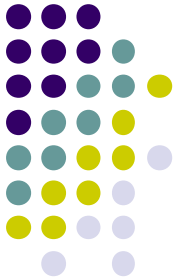
```
public class Main {

    public static void main(String[] args) {
        ArrayList ciudades = new ArrayList();
        ciudades.add("Madrid");
        ciudades.add("Barcelona");
        ciudades.add("Malaga");
        ciudades.add("Vigo");
        ciudades.add(1, "Sevilla");
        ciudades.add("Madrid"); // Repetido.
        for (int i = ciudades.size() - 1; i >= 0; i--) {
            System.out.println("Ciudad: " + i + " es: " + ciudades.get(i));
        }
    }
}
```

EJEMPLO LINKEDLIST

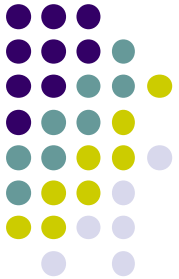


```
public class Main {

    public static void main(String[] args) {
        LinkedList ciudades = new LinkedList();
        ciudades.add("Madrid");
        ciudades.add("Barcelona");
        ciudades.add("Malaga");
        ciudades.add("Vigo");
        ciudades.add(1, "Sevilla");
        ciudades.add("Madrid"); // Repetido.
        Iterator it = ciudades.iterator();
        while (it.hasNext()) {
            System.out.println("Ciudad: " + it.next());
        }
    }
}
```

EJEMPLO VECTOR



```
public class Main {

    public static void main(String[] args) {
        Vector ciudades = new Vector();
        ciudades.add("Madrid");
        ciudades.add("Barcelona");
        ciudades.add("Malaga");
        ciudades.add("Vigo");
        ciudades.add(1, "Sevilla");
        ciudades.add("Madrid"); // Repetido.
        for (int i = ciudades.size() - 1; i >= 0; i--) {
            System.out.println("Ciudad: " + i + " es: " + ciudades.get(i));
        }
    }
}
```



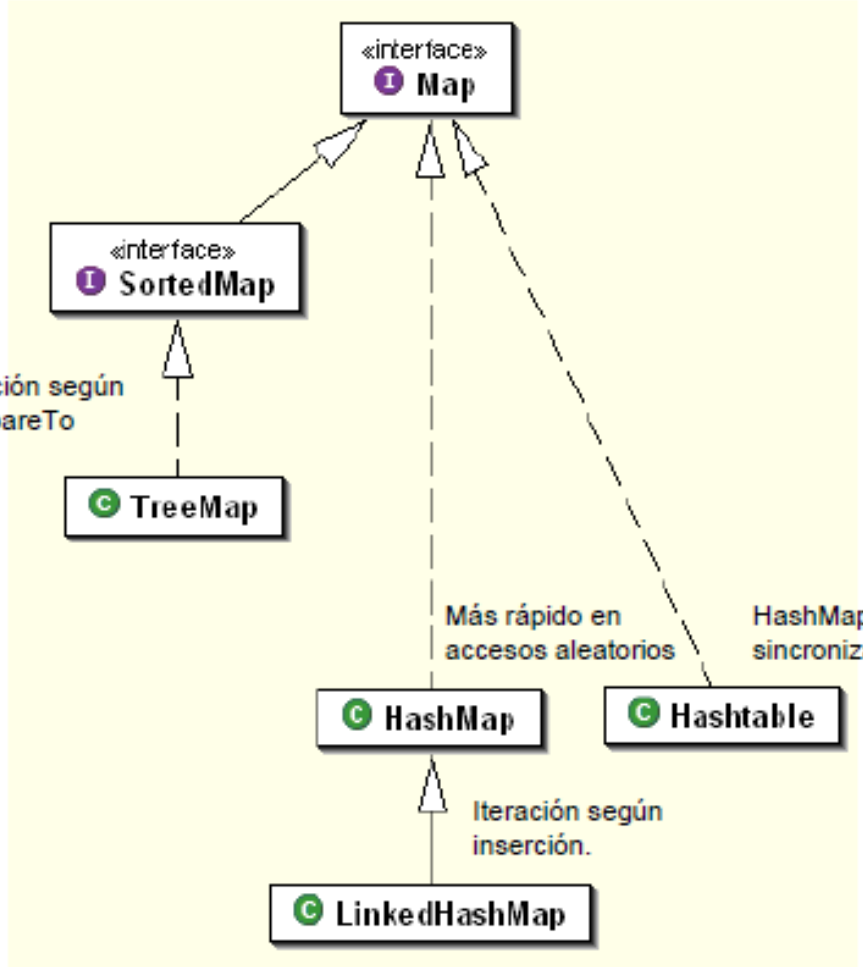
ÍNDICE

1. Colecciones
2. Arrays vs Colecciones
3. Interface java.util.Collection
4. Interface java.util.Iterator
5. Interface java.util.Set
6. Interface java.util.List
7. Interface java.util.Map
8. Importancia de equals y hashcode
9. Ordenación

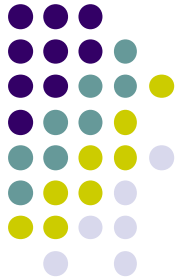
INTERFAZ JAVA.UTIL.MAP

- Métodos:
  - Operaciones básicas
  - Operaciones masivas
  - Operaciones de colecciones

- Clases que implementan esta interfaz:
  - java.util.HashMap
  - java.util.Hashtable
  - java.util.LinbkedHashMap
  - java.util.TreeMap

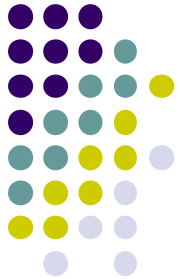


INTERFAZ JAVA.UTIL.MAP



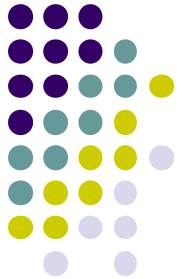
- El interfaz Map no hereda del interfaz Collection.
- Representa colecciones con parejas de elementos: clave y valor.
- No permite tener claves duplicadas. Pero si valores duplicados.
- Para calcular la colocación de un elemento se basa en el uso del método public int hashCode(); (excepto TreeMap)

INTERFAZ JAVA.UTIL.MAP



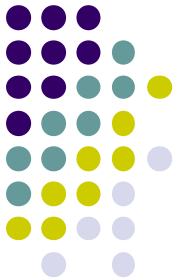
- Operaciones básicas:
  - Object put(Object key, Object value); // Inserta una pareja.
  - Object get(Object key); // Accede al valor de una clave.
  - Object remove(Object key); // Elimina una pareja.
  - boolean containsKey(Object key); // Comprueba si existe una clave.
  - boolean containsValue(Object value); // Comprueba si existe un valor
  - int size(); // Número de parejas.
  - boolean isEmpty(); // Si no contiene ninguna pareja

INTERFAZ JAVA.UTIL.MAP



- Operaciones masivas
  - void putAll(Map t); // Añade todas las parejas.
  - void clear(); // Elimina todas las parejas
- Obtención de colecciones
  - public Set keySet(); // Devuelve las claves en un Set.
  - public Collection values(); // Devuelve los valores en una Collection.

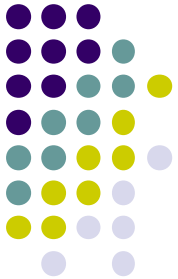
INTERFAZ JAVA.UTIL.MAP



- Clases que implementan esta interfaz:
  - java.util.HashMap
    - Ofrece un tiempo de acceso óptimo cuando dicho acceso es aleatorio.
    - Orden de iteración imprevisible
  - java.util.Hashtable
    - Versión sincronizada de HashMap
  - java.util.LinkedHashMap
    - Orden de iteración es el de inserción
  - java.util.TreeMap
    - Su orden de iteración depende de la implementación que los elementos hagan del interfaz java.lang.Comparable

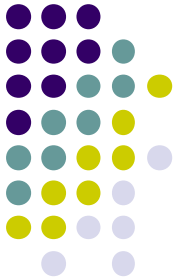
      public int compareTo(Object o);

EJEMPLO HASHMAP



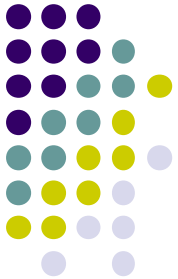
```
public class Main {

    public static void main(String[] args) {
        HashMap codigos = new HashMap();
        codigos.put("01", "Urgente");
        codigos.put("02", "Importante");
        codigos.put("03", "Normal");
        codigos.put("04", "Baja prioridad");
        System.out.println("Aleatorio 03: " + codigos.get("03"));
        System.out.println("Aleatorio 01: " + codigos.get("01"));
        Set s = codigos.keySet();
        Iterator it = s.iterator();
        while (it.hasNext()) {
            String aux = (String) it.next();
            System.out.println(aux + ": " + codigos.get(aux));
        }
    }
}
```

EJEMPLO HASHTABLE



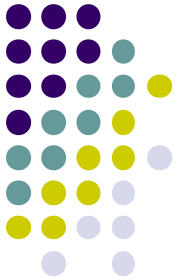
```
public class Main {

    public static void main(String[] args) {
        Hashtable codigos = new Hashtable();
        codigos.put("01", "Urgente");
        codigos.put("02", "Importante");
        codigos.put("03", "Normal");
        codigos.put("04", "Baja prioridad");
        System.out.println("Aleatorio 03: " + codigos.get("03"));
        System.out.println("Aleatorio 01: " + codigos.get("01"));
        Set s = codigos.keySet();
        Iterator it = s.iterator();
        while (it.hasNext()) {
            String aux = (String) it.next();
            System.out.println(aux + ": " + codigos.get(aux));
        }
    }
}
```

EJEMPLO LINKEDHASHMAP

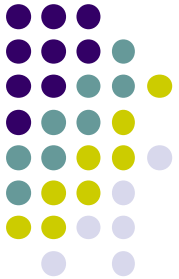


```
public class Main {

    public static void main(String[] args) {
        LinkedHashMap codigos = new LinkedHashMap();
        codigos.put("01", "Urgente");
        codigos.put("02", "Importante");
        codigos.put("03", "Normal");
        codigos.put("04", "Baja prioridad");
        System.out.println("Aleatorio 03: " + codigos.get("03"));
        System.out.println("Aleatorio 01: " + codigos.get("01"));
        Set s = codigos.keySet();
        Iterator it = s.iterator();
        while (it.hasNext()) {
            String aux = (String) it.next();
            System.out.println(aux + ": " + codigos.get(aux));
        }
    }
}
```

EJEMPLO TREEMAP



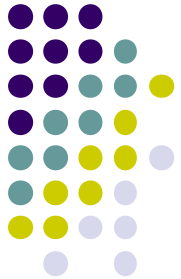
```
public class Main {

    public static void main(String[] args) {
        TreeMap codigos = new TreeMap();
        codigos.put("04", "Baja prioridad");
        codigos.put("01", "Urgente");
        codigos.put("03", "Normal");
        codigos.put("02", "Importante");
        System.out.println("Aleatorio 03: " + codigos.get("03"));
        System.out.println("Aleatorio 01: " + codigos.get("01"));
        Set s = codigos.keySet();
        Iterator it = s.iterator();
        while (it.hasNext()) {
            String aux = (String) it.next();
            System.out.println(aux + ": " + codigos.get(aux));
        }
    }
}
```



ÍNDICE

1. Colecciones
2. Arrays vs Colecciones
3. Interface java.util.Collection
4. Interface java.util.Iterator
5. Interface java.util.Set
6. Interface java.util.List
7. Interface java.util.Map
8. Importancia de equals y hashcode
9. Ordenación

IMPORTANCIA DE EQUALS Y HASHCODE

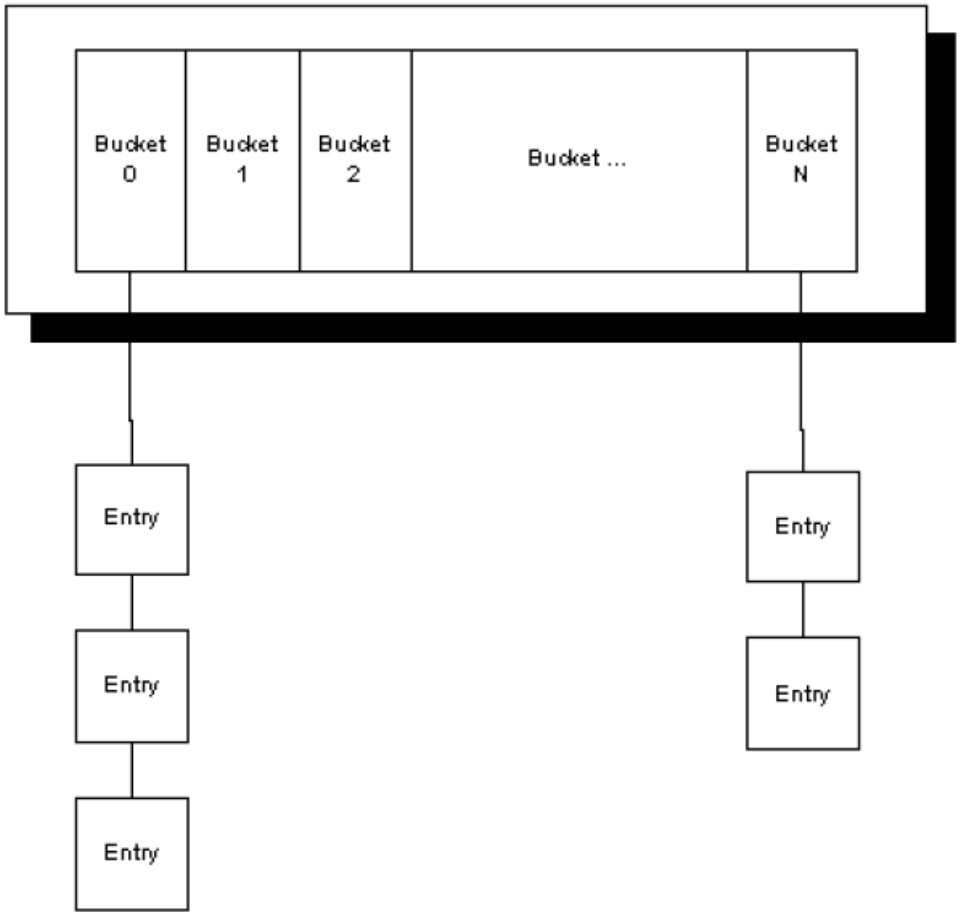


- Siempre que creemos nuestras propias claves para el uso de los Map, debemos sobreescribir los métodos equals() y hasCode()

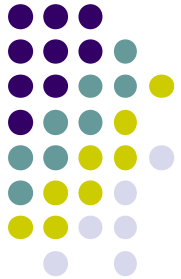
- Los Map utilizan estos métodos para realizar inserciones y extracciones de valores

- Para entender mejor el uso de estos dos métodos, veamos un poco mas en detalle la estructura interna de este tipo de colección

IMPORTANCIA DE EQUALS Y HASHCODE



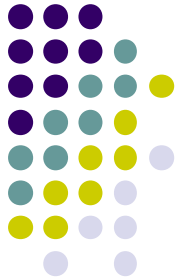
IMPORTANCIA DE EQUALS Y HASHCODE



- Un Map internamente contiene una secuencia de compartimentos (buckets) donde se van almacenando todos los valores (clave/valor).

- Para decidir en qué compartimento se almacena un valor, se llama al método hashCode() del objeto utilizado como clave.

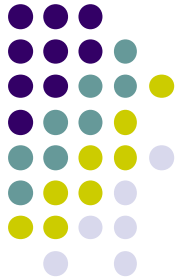
IMPORTANCIA DE EQUALS Y HASHCODE



- Pueden ocurrir colisiones, es decir, que un compartimento ya esté utilizado por una pareja clave/valor.

- Esto puede ser debido a que:
  - Dos objetos distintos devolvieron el mismo código hash.
  - Dos códigos hash distintos correspondieron al mismo compartimento.

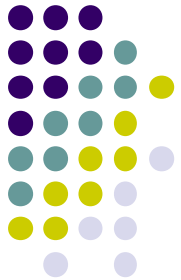
IMPORTANCIA DE EQUALS Y HASHCODE



- Imaginemos que hacemos un get y el compartimento correspondiente tiene colisiones.
  - ¿Qué valor nos devuelve?
    - Lo sabe mediante el uso del método equals() de la clave.
    - Va iterando por todas las claves de ese compartimento para encontrar la que se ha pedido.

- Imaginemos que hacemos un put (añadir) con una clave ya existente.
  - ¿Cómo sabe que ya existe y que hay que machacar el valor anterior?
    - Lo sabe mediante el uso del método equals() de la clave. Itera para comprobar si ya existe.

IMPORTANCIA DE EQUALS Y HASHCODE



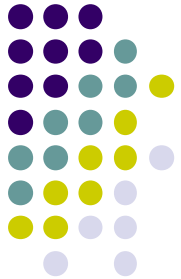
- La implementación del método equals() debe cumplir las siguientes normas:
  - Reflexiva: x.equals(x) debe devolver true.
  - Simétrica: Si x.equals(y) devuelve true, y.equals(x) debe devolver también true.
  - Transitiva: Si x.equals(y) devuelve true, e y.equals(z) devuelve true, x.equals(z) debe devolver también true.
  - Consistente: Si x.equals(y) devuelve true, entonces las sucesivas invocaciones de x.equals(y) sin haber modificado el estado de x o y deben seguir devolviendo true.
  - Null: x.equals(null) siempre debe devolver false.

EJEMPLO

```
public class TestEquals {

   private int valor1;
   private Integer valor2;

   public boolean equals(Object o) {
      if (this == o) // Primer paso.
      {
         return true;
      }
      if (!(o instanceof TestEquals)) // Segundo paso.
      {
         return false;
      }
      TestEquals param = (TestEquals) o; // Tercer paso.
      return param.valor1 == valor1 && param.valor2.equals(valor2);
   }
}
```

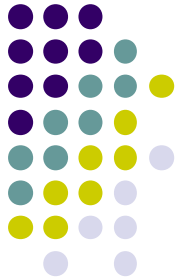
Se ha omitido el constructor de la clase

EJEMPLO (cont)



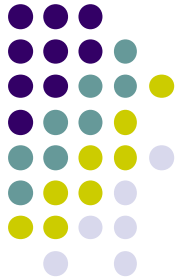
```
public static void main(String[] args) {
        TestEquals test1 = new TestEquals(1, new Integer(2));
        TestEquals test2 = new TestEquals(1, new Integer(2));
        System.out.println(test1.equals(test2));
    }
```

IMPORTANCIA DE EQUALS Y HASHCODE



- La implementación del método hashCode() debe cumplir las siguientes normas:
  - La ejecución sucesiva del método hashCode() sobre un mismo objeto sin haber modificado su estado interno entre medias, debe devolver siempre el mismo código hash.
  - Si x.equals(y) devuelve true, entonces tanto x como y deben generar el mismo código hash.
  - Si x.equals(y) devuelve false, no es obligatorio que tanto x como y deban generar un código hash distinto.
    - Pero es deseable para evitar colisiones y ofrecer un mejor rendimiento.

IMPORTANCIA DE EQUALS Y HASHCODE



- La implementación del hashCode() no es trivial.

- Aquí proponemos dos sugerencias sencillas:
  1. Utilizar hashCode() de la clase String:
     - Convertir a String los valores de los atributos de la clase.
     - Concatenarlos y delegar la generación del código hash en el método hashCode() del String resultante
       - String posee una implementación eficaz del método hashCode().
  2. Utilizar hashCode() de los wrappers:
     - Sumar el código hash de cada uno de los atributos de la clase
       - Los wrappers de tipos primitivos también tienen sobreescrito el método hashcode().

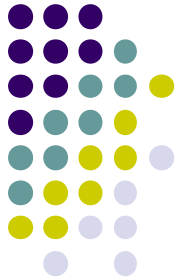
EJEMPLO

- Utilizando el hashCode() de la clase String:

```
public class TestHashCode {

    private int valor1;
    private Integer valor2;

    public int hashCode() {
        StringBuffer buffer = new StringBuffer();
        buffer.append(Integer.toString(valor1));
        buffer.append(valor2.toString());
        return buffer.toString().hashCode();
    }
}
```

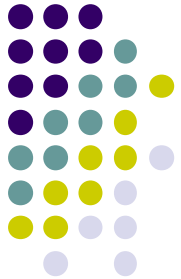
Se ha omitido el constructor de la clase

EJEMPLO (cont)



```
public class Main {

    public static void main(String[] args) {
        TestHashCode test1 = new TestHashCode(1, new Integer(2));
        TestHashCode test2 = new TestHashCode(1, new Integer(2));
        System.out.println(test1.hashCode());
        System.out.println(test2.hashCode());
    }
}
```

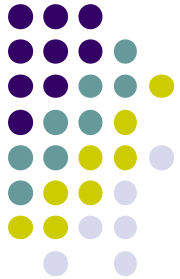
IMPORTANCIA DE EQUALS Y HASHCODE



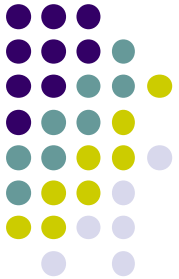
- El método equals() también es importante para el resto de colecciones.

- Por ejemplo, ¿cómo funcionan los métodos contains(), add() y remove() de las colecciones?
  - Para saber si un objeto está contenido en una colección se va llamando al método equals() de todos los objetos de la colección.
  - Para borrarlo de una colección, se le busca de igual forma.
  - Y para añadirlo en un Set que no permite duplicados, lo mismo.

IMPORTANCIA DE EQUALS Y HASHCODE



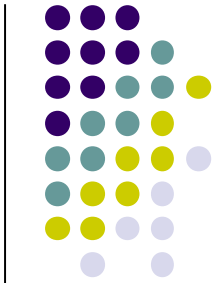
```
public class Main {

    public static void main(String[] args) {
        TestEquals test1 = new TestEquals(1, new Integer(2));
        List list = new ArrayList();
        list.add(test1);
        TestEquals test2 = new TestEquals(1, new Integer(2));
        System.out.println(list.contains(test2));
    }
}
```



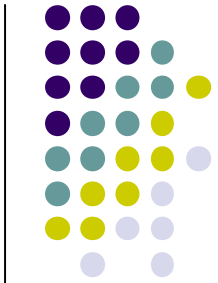
ÍNDICE

1. Colecciones
2. Arrays vs Colecciones
3. Interface java.util.Collection
4. Interface java.util.Iterator
5. Interface java.util.Set
6. Interface java.util.List
7. Interface java.util.Map
8. Importancia de equals y hashcode
9. Ordenación

ORDENACIÓN



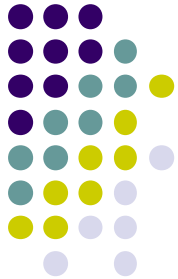
- Algunas de las clases con las que se ha trabajado en este tema mantienen sus elementos ordenados o, dicho de otra manera, cuando se itera sobre sus elementos, estos aparecen ordenados:
  - TreeSet
  - TreeMap
  - (En otros, el orden era el de inserción o impredecible).
- Por otra parte, existen clases capaces de realizar una ordenación de colecciones o estructuras de datos. Por ejemplo:
  - Un List se puede ordenar de la siguiente forma
    - List <String> l = new ….
    - Collections.**sort**(l)
  - Un array se puede ordenar de la siguiente forma
    - String v[] = new String […];
    - Arrays.**sort**(v);



ORDENACIÓN

- Si la lista o el array están formados por String, se ordenarán por orden alfabético. Si están formados por objetos de tipo Date (fechas), se ordenarán cronológicamente.
- ¿Por qué ocurre esto? ¿Por qué se ordenan así?
  - Tanto String como Date implementan el interface **Comparable**.
  - Implementar Comparable obliga a las clases String y Date a implementar el método **compareTo**
  - Es en el método compareTo donde se establece el orden de los objetos de la clase.
  - El que determina el método compareTo es lo que se denomina ***orden natural (natural ordering)***

ORDENACIÓN



- Algunas clases que implementan Comparable:

Clase	Orden natural
Byte, Long, Integer, Sort, Double, Float	Numérico con signo
Character	Numérico sin signo
Boolean	false < true
String	Lexicográfico
Date	Cronológico

ORDENACIÓN

- Si se intenta la ordenación de objetos que no implementan Comparable se producirá error

```
class Alumno{ …} //No implementa comparable
```

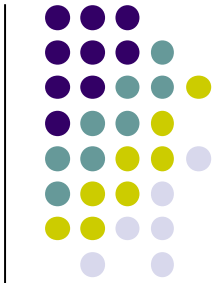
  - Se producirá **error de compilación** cuando se utilicen genéricos

```
List<Alumno> l = new ArrayList<Alumno> ();
…
Collections.sort(l);
…
TreeSet<Alumno> s = new TreeSet<Alumno> ();
```

  - Se producirá **excepción** cuando no se usen genéricos.

```
List l = new ArrayList();
l.add(new Alumno(…));
Collections.sort(l); //excepción

TreeSet s = new TreeSet();
s.add(new Alumno()); //excepción
```

ORDENACIÓN



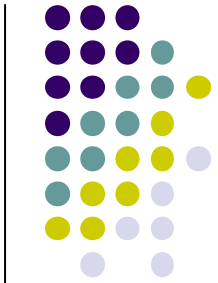
- Para crear nuestras **propias ordenaciones naturales**
  - La clase debe implementar Comparable

```
public interface Comparable{
        int compareTo(Object o);
}
```

  - Ejemplo

```
public class Alumno implements Comparable {
  ...
  public int compareTo(Object o){
        Alumno a = (Alumno) o;
        return numExpediente – a.numExpediente;
  }
}
```

ORDENACIÓN

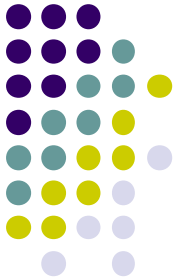


- Para crear nuestras **propias ordenaciones naturales**
  - ó, la clase debe implementar Comparable <T>, en el caso de clases genéricas.

```
public interface Comparable<T>{
  int compareTo(T o);
}
```

  - Ejemplo

```
public class Alumno implements Comparable<Alumno> {
  ...
  public int compareTo(Alumno o){
         return numExpediente – o.numExpediente;
  }
}
```



ORDENACIÓN

- ¿Qué ocurre si …
  - … queremos ordenar por un orden distinto al natural?
  - … queremos ordenar objetos que no implementan comparable?
- En ambos casos tendremos que utilizar un **Comparator**
- Comparator es un interface:
  - (versión no genérica)

```
public interface Comparator{
        int compare(Object o1, Object o2);
}
```

  - (versión genérica)

```
public interface Comparator<T>{
        int compare(T o1, T o2);
}
```

- La implementación debe devolver un valor menor, igual o mayor que cero dependiendo de si o1 es menor, igual o mayor que o2, respectivamente.

ORDENACIÓN

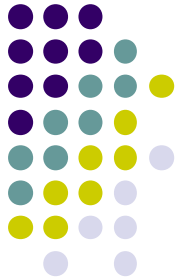
- Supongamos que queremos ordenar una lista de alumnos por su edad.
- Crearemos un Comparator …

```
public class OrdenAlumnoEdad implements Comparator {
   public int compare(Object o1, Object o2){
   Alumno a1 = (Alumno) o1;        Alumno a2 = (Alumno) o2;
   if(a1.edad < a2.edad) return -1;
   else if(a1.edad > a2.edad) return 1;
   else return 0;
   }
   }
```

  - (versión genérica)

```
public class OrdenAlumnoEdad implements Comparator<Alumno> {
   public int compare(Alumno a1, Alumno a2){
   if(a1.edad < a2.edad) return -1;
   else if(a1.edad > a2.edad) return 1;
   else return 0;
   }
   }
```

ORDENACIÓN



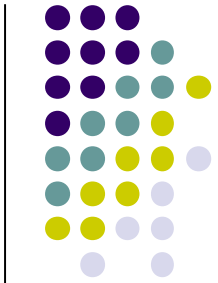
- … y lo utilizaremos a la hora de ordenar

```
List l = new ArrayList();
l.add(new Alumno(…));
...
...
Collections.sort(l,new OrdenAlumnoEdad());
```

  - (versión genérica)

```
List<Alumno> l = new ArrayList<Alumno>();
l.add(new Alumno(…));
...
...
Collections.sort(l,new OrdenAlumnoEdad<Alumno>());
```

ORDENACIÓN



- CUIDADO
  - Podemos utilizar un comparator cualquiera para ordenar un List pero hay que tener cuidado si utilizamos un Comparator para una colección ordenada (por ejemplo un TreeSet)

```
Set s = new TreeSet(new OrdenAlumnoEdad());
...
...
```

  - En las colecciones ordenadas, el método equals debe ser coherente con el criterio de ordenación.
    - Si la comparación devuelve cero ENTONCES el método equals debe devolver true y viceversa.
    - De lo contrario, la colección ordenada no funcionará correctamente.
    - Para que esto se diera en el ejemplo de alumnos, dado que el método compare devuelve cero cuando dos alumnos tienen la misma edad, el método equals de la clase Alumno debería devolver true cuando dos alumnos tienen la misma edad.