# Java Básico

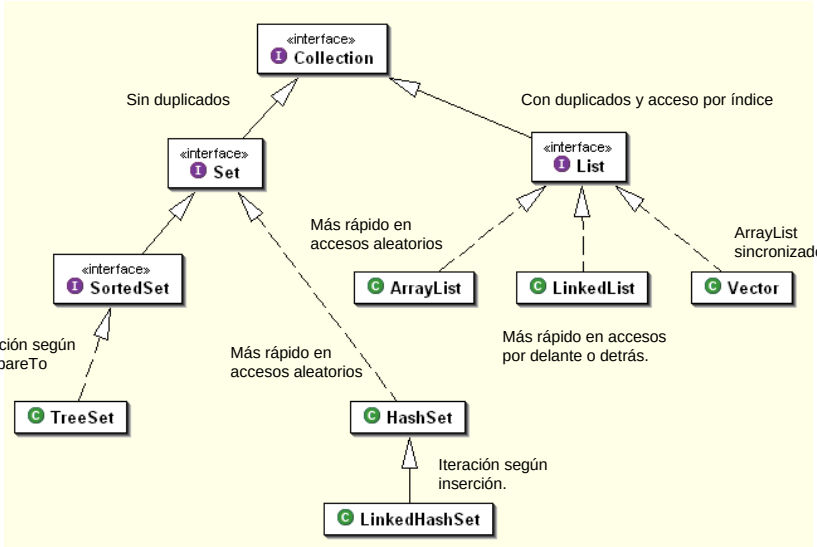
## Estructuras de Datos 2

## Copyright

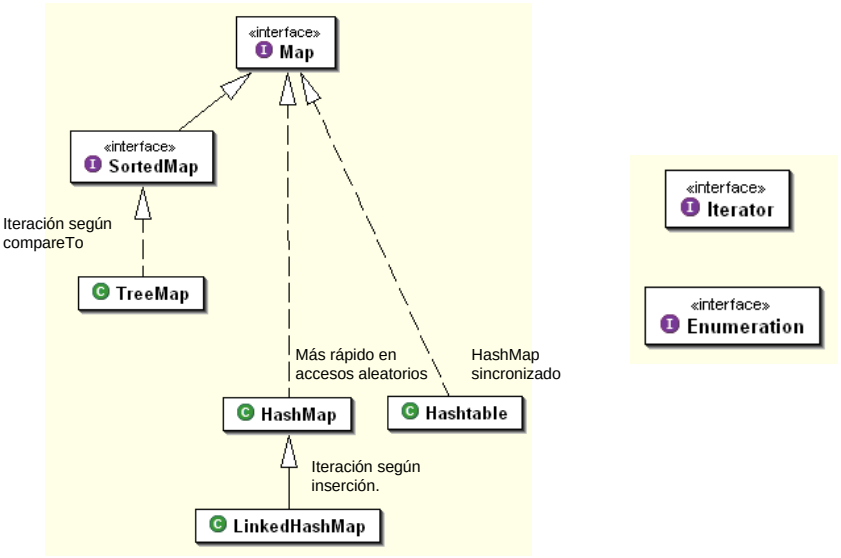
# Colecciones

- Una colección es simplemente un 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



# Colecciones




# Colecciones

- Todas las colecciones se encuentran en el paquete java.util.*;
- java.util.Collection es la raíz de la jerarquía de las colecciones.
- Existirán especializaciones que permitan elementos duplicados o no, que permitan ordenar los elementos o no, etc.
- Esta clase contiene la definición de todos los métodos genéricos que deben implementar las colecciones.

# java.util.Collection

- Los métodos de este interfaz son:
- 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.
  - remove(Object element); // Borrar un elemento.
  - Iterator iterator(); // Devuelve una instancia de Iterator.

# 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.

# 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.
- Nota: las colecciones no permiten el uso de tipos primitivos. Por tanto, siempre que necesitemos trabajar con ellos habrá que hacer uso de los Wrappers de Tipos Primitivos.

# 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();
- Sus métodos son:
  - boolean hasNext(); // Si tiene mas elementos.
  - Object next(); // Devuelve el primer elemento y se queda apuntando al siguiente.

# java.util.Iterator

- void remove(); // Elimina el primer alemento y se queda apuntando al siguiente.
- En el SDK 1.1.x existía otro interfaz:
- java.util.Enumeration
- Pero ya no se usa.

# 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);
- Existen distintas implementaciones de este interfaz.

# java.util.Set

- java.util.HashSet:
  - Es nueva en el SDK 1.2.x.
  - Ofrece el acceso mas rápido cuando dicho acceso es aleatorio.
  - Su orden de iteración es impredecible.
- java.util.LinkedHashSet:
  - Es nueva en el SDK 1.4.x.
  - Su orden de iteración es el orden de inserción.

# java.util.Set

- java.util.TreeSet:
  - Es nueva en el SDK 1.2.x.
  - Su orden de iteración depende de la implementación que los elementos hagan del interfaz java.lang.Comparable:

    public int compareTo(Object o);

# Ejemplo

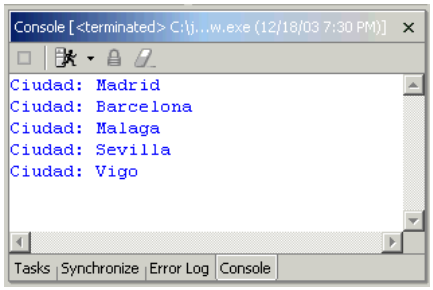
```
import java.util.*;

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

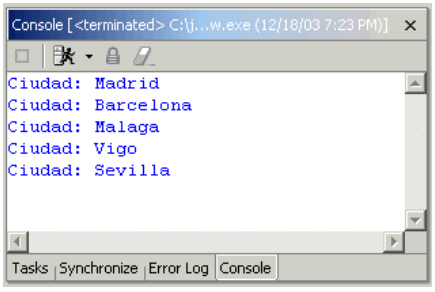
```
import java.util.*;

public class TestLinkedHashSet
{
  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

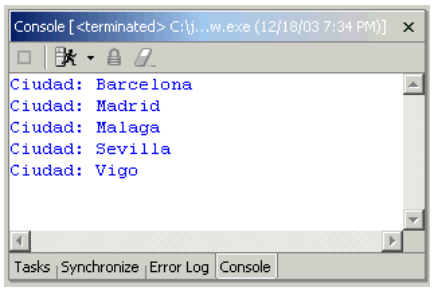
```
import java.util.*;

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.

# 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.

# java.util.List

- Los métodos que añade de este interfaz son:
- 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.

# java.util.List

- boolean addAll(int index, Collection c); // Inserta todos esos elementos en esa posición.
- 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.

# java.util.List

- Existen distintas implementaciones de este interfaz.
- java.util.ArrayList:
  - Es nueva en el SDK 1.2.x.
  - Ofrece un tiempo de acceso óptimo cuando dicho acceso es aleatorio.
- java.util.LinkedList:
  - Es nueva en el SDK 1.2.x.
  - 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).

# java.util.List

- java.util.Vector:
  - Es como el ArrayList pero sincronizado lo que penaliza notablemente el rendimiento.
  - La sincronización es importante cuando mas de un thread (hilo de ejecución) va a acceder a la colección.

# Ejemplo

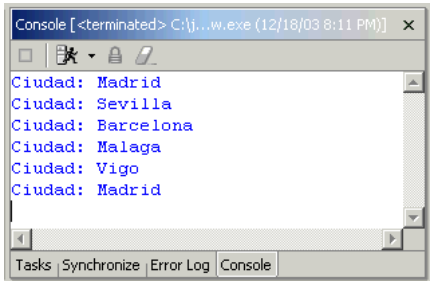
```
import java.util.*;

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



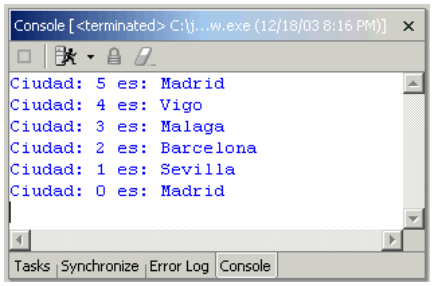

# Ejemplo

```
import java.util.*;

public class TestArrayList2
{
  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

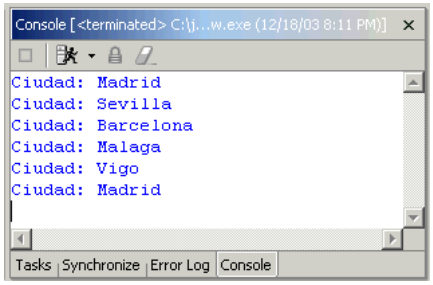
```
import java.util.*;

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



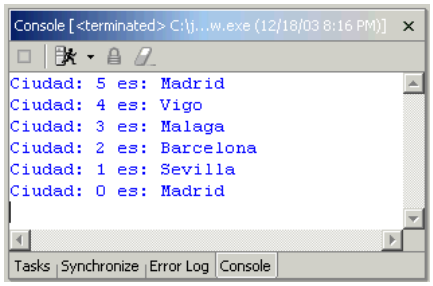

# Ejemplo

```
import java.util.*;

public class TestVector
{
  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));
  }
}
```



# 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();

# java.util.Map

- Los métodos de este interfaz son:
- 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 la existencia de una clave.
  - boolean containsValue(Object value); // Comprueba la existencia de un valor.

# java.util.Map

- int size(); // Número de parejas.
- boolean isEmpty(); // Si no contiene ninguna pareja.
- 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.

# java.util.Map

- Existen distintas implementaciones de este interfaz.
- java.util.HashMap:
  - Es nueva en el SDK 1.2.x.
  - Ofrece un tiempo de acceso óptimo cuando dicho acceso es aleatorio.
  - Su orden de iteración es imprevisible.
- java.util.Hashtable:
  - Es la versión sincronizada de HashMap.

# java.util.Map

- java.util.LinbkedHashMap:
  - Es nueva en el SDK 1.4.x.
  - Su 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);

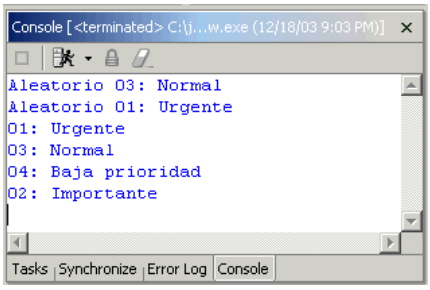
# Ejemplo

```
import java.util.*;

public class TestHashMap
{
  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));
    }
  }
}
```



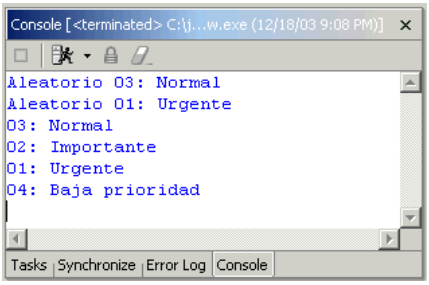
# Ejemplo

```
import java.util.*;

public class TestHashtable
{
  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));
    }
  }
}
```



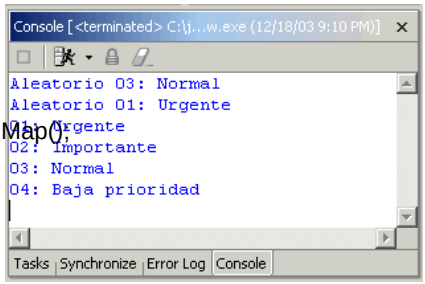

# Ejemplo

```
import java.util.*;

public class TestLinkedHashMap
{
  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));
    }
  }
}
```



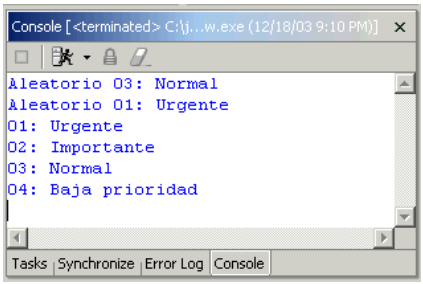
# Ejemplo

```
import java.util.*;

public class TestTreeMap
{
  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));
    }
  }
}
```



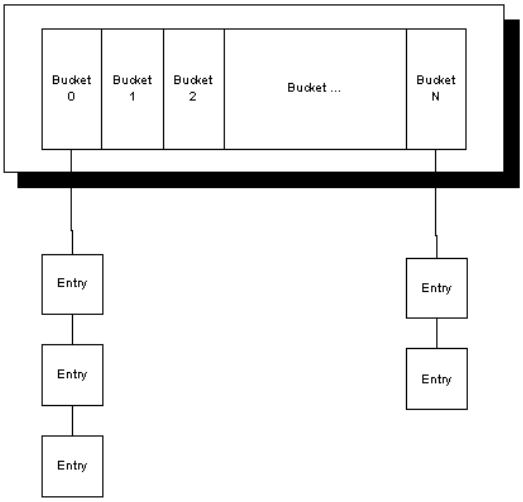

**Nota**: funciona porque java.lang.String implementa java.lang.Comparable.

# Importancia de equals() y hashCode()

- Siempre que creemos nuestras propias claves para el uso de los Map, debemos sobreescribir los métodos equals() y hasCode().
- El motivo es que los Map utilizan estos dos métodos para llevar a cabo tanto las inserciones como las extracciones de valores.
- Para entender mejor el uso de estos dos métodos por parte de los Map, veamos un poco mas en detalle la estructura interna de este tipo de colección.

# Importancia de equals() y hashCode()

- Estructura interna de un Map:




# 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.
- Pero 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 del Map 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 en el Map 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.

# Importancia de equals() y hashCode()

Ejemplo:

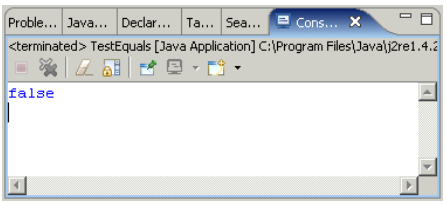
```
public class TestEquals
{
  private int valor1;
  private Integer valor2;

  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);
  }
}
```

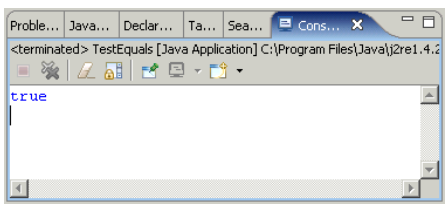
# Importancia de equals() y hashCode()

Ejemplo:

```
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));
}
```




Sin sobreescribir equals()




Sobreescribiendo equals()

## 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.
  - Sin embargo, si x.equals(y) devuelve false, no es obligatorio que tanto x como y deban generar un código hash distinto. No obstante es lo deseable para evitar en la medida de lo posible las colisiones en los Map y por tanto ofrecer un mejor rendimiento.

## Importancia de equals() y hashCode()

- La implementación del método hashCode() no es una tarea tan trivial.
- No obstante, aquí proponemos dos sugerencias bastante sencillas:
  - Convertir a String los valores de los distintos atributos de la clase. Concatenarlos y delegar la generación del código hash en el método hashCode() del String resultante (la clase String posee una implementación bastante eficaz del método hashCode()).
  - Sumar el código hash de cada uno de los atributos de la clase (los wrappers de tipos primitivos también tienen sobreescrito el método hashcode()).

# Importancia de equals() y hashCode()

Ejemplo:

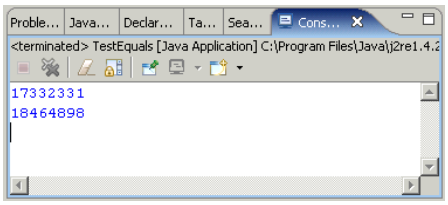
```
public class TestHashCode
{
  private int valor1;
  private Integer valor2;

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

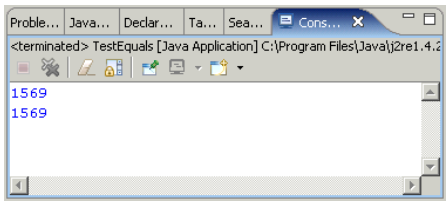
# Importancia de equals() y hashCode()

Ejemplo:

```
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());
}
```




Sin sobreescribir hashCode()




Sobreescribiendo hashCode()

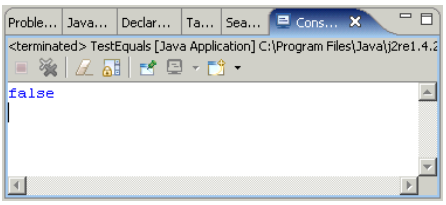
# Importancia de equals() y hashCode()

- 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.

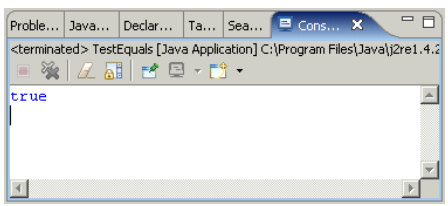
# Importancia de equals() y hashCode()

- Ejemplo:

```
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));
}
```




Sin sobreescribir equals()




Sobreescribiendo equals()

# Sentencia for/in 5.0

- Esta nueva sentencia del J2SE 5.0 nos facilita la iteración por los elementos de cualquier tipo de colección: arrays, listas, etc… (excepto Map).

```
for(inicialización: colección)      Nota: Se usa ":" en vez de ";".
{
   sentencias;
}
```

- Ejemplo:

```
public void listar(ArrayList param)
{
  for(Object o: param)
    System.out.println(o);
}
```

# Sentencia for/in 5.0

- Básicamente, se trata de una simplificación a la hora de codificar.
- Es decir, al final, el compilador convierte el código en una sentencia *for* convencional:

```
public void listar(ArrayList param)
{
  Iterator it = param.iterator();
  while(it.hasNext())
    System.out.println(it.next());
}
```

- Veremos como trabajar con este nuevo tipo de sentencia en el capítulo dedicado a J2SE 5.0

# Arrays vs. Collections

- Arrays:
  - Tamaño estático.
  - Su tamaño se conoce mediante el atributo length.
  - Puede almacenar tanto tipos primitivos como tipos complejos.
  - Solo pueden albergar elementos de un tipo.

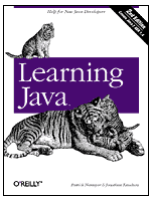
# Arrays vs. Collections

- Collections:
  - Tamaño dinámico.
  - Su tamaño se conoce mediante el método size().
  - Solo puede almacenar tipos complejos.
  - Puede albergar elementos de distinto tipo.

# Bibliografía

- Head First Java
  Kathy Sierra y Bert Bates.
  O'Reilly


- Learning Java (2nd edition)
  Patrick Niemeyer y Jonathan Knudsen.
  O'Reilly.




- Thinking in Java (3rd edition)
  Bruce Eckel.
  Prentice Hall.


- The Java tutorial
  http://java.sun.com/docs/books/tutorial/