

## Colecciones en Java

# La interfaz Iterator



La interfaz `java.util.Iterator` se usa para recorrer las colecciones posicionales

```
Interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

## Ejemplo

```
Iterator it = puntos.iterator();
while (it.hasNext()) {
    Point2D p = (Point2D) it.next();
    canvas.draw(p);
}
```

# Métodos equals y hashCode



- La comparación entre objetos Java se realiza con el operador `==` y el método `equals`.
- La clase `java.lang.Object` define un método `equals` por defecto que utiliza la igualdad de referencia. Si queremos implementar la igualdad de contenidos en los objetos de nuestras clases, hay que sobrescribir este método.
- El método `hashCode` también se define en la superclase `Object` y devuelve un `int` que define la clave hash de un objeto. Debe ser compatible con `equals` (si `equals` devuelve `true`, `hashCode` debe devolver el mismo `int` para dos objetos).

# La interfaz Comparable



- La interfaz `java.lang.Comparable` define un único método

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

- El método `compareTo` debe devolver un entero negativo, 0, o positivo cuando el objeto que recibe el mensaje es menor, igual o mayor que el `Object o` pasado como parámetro.
- Es aconsejable que el método sea consistente con `equals`, esto es, devuelva 0 si y sólo si los dos objetos que se comparan con `equals` son iguales.
- Tanto `equals` como `compareTo` no deben cambiar mientras que los objetos están dentro de colecciones.

# Ejemplo de Comparable



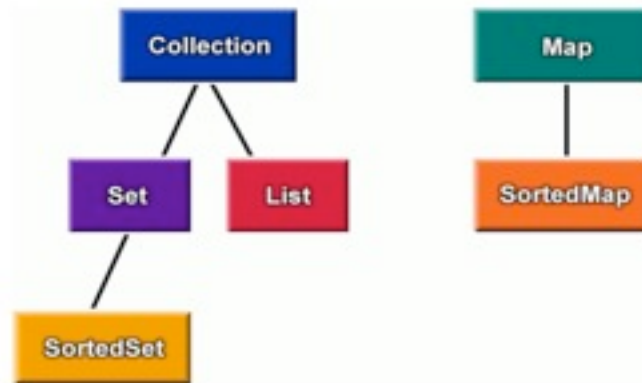
```
public boolean equals(Object o) {
    if (!(o instanceof Punto2D))
        return false;
    Punto2D punto = (Punto2D) o;
    return (x == punto.x) && (y == punto.y);
}

public int compareTo(Object o) {
    if (this.equals(o))
        return 0; // consistente con equals
    Punto2D punto = (Punto2D) o;
    if (x >= punto.x && y > punto.y)
        return 1;
    return -1;
}
}
```

# Interfaces de las colecciones



- Las colecciones se definen por la siguiente jerarquía de interfaces



- Ventajas de definir interfaces:
  - Se separa la especificación de la implementación
  - Es más sencillo reemplazar el funcionamiento de una clase por otra que implementa la misma interfaz y que es más eficiente

# La interfaz Collection



```
public interface Collection {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element); // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c); // Optional
    boolean removeAll(Collection c); // Optional
    boolean retainAll(Collection c); // Optional
    void clear(); // Optional

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

# Implementaciones de Set



- Es la misma que `Collection`, con la particularidad de que no se permiten objetos iguales en la colección.
- Implementación `HashSet`
  - Usa una tabla hash
  - No introduce orden en sus elementos
  - Los métodos `add`, `remove` y `contains` son muy eficientes:  $O(c)$
- Implementación `TreeSet`
  - Usa un árbol
  - Introduce un orden en los elementos, útil para recorrer el conjunto
  - Los métodos `add`, `remove` y `contains` son  $O(\log(n))$

# Ejemplo



```
import java.util.*;

public class FindDups {
    public static void main(String args[]) {
        Set s = new HashSet();
        for (int i=0; i<args.length; i++)
            if (!s.add(args[i]))
                System.out.println("Duplicate detected: "+args[i]);
        System.out.println(s.size()+" distinct words detected: "+s);
    }
}
```

```
% java FindDups i came i saw i left
Duplicate detected: i
Duplicate detected: i
4 distinct words detected: [came, left, saw, i]
```

# La interfaz List



```
public interface List extends Collection {
    // Positional Access
    Object get(int index);
    Object set(int index, Object element);           // Optional
    void add(int index, Object element);           // Optional
    Object remove(int index);                       // Optional
    abstract boolean addAll(int index, Collection c); // Optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator listIterator();
    ListIterator listIterator(int index);

    // Range-view
    List subList(int from, int to);
}
```

# Implementaciones de List



- `ArrayList`
  - Usa un array extensible como implementación
  - Operaciones `get` y `set` en tiempo constante
- `LinkedList`
  - Basada en una lista doblemente enlazada
  - Operaciones `add` y `remove` en tiempo constante

# Ejemplo de ArrayList



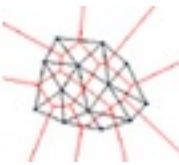
```
import java.util.*;
public class Shuffle {
    public static void main(String args[]) {
        List l = new ArrayList();
        for (int i = 0; i < args.length; i++)
            l.add(args[i]);
        Collections.shuffle(l, new Random());
        System.out.println(l);
    }
}
```

Aún más corto:

```
import java.util.*;

public class Shuffle {
    public static void main(String args[]) {
        List l = Arrays.asList(args);
        Collections.shuffle(l);
        System.out.println(l);
    }
}
```

# Ejemplo de LinkedList



```
import java.util.*;
public class MyStack {
    private LinkedList list = new LinkedList();
    public void push(Object o){
        list.addFirst(o);
    }
    public Object top(){
        return list.getFirst();
    }
    public Object pop(){
        return list.removeFirst();
    }

    public static void main(String args[]) {
        Point2D punto;
        MyStack s = new MyStack();
        s.push (new Point2D());
        punto = (Point2D)s.pop();
    }
}
```

# El iterador ListIterator



```
public interface ListIterator extends Iterator {
    boolean hasNext();
    Object next();

    boolean hasPrevious();
    Object previous();

    int nextIndex();
    int previousIndex();

    void remove(); // Optional
    void set(Object o); // Optional
    void add(Object o); // Optional
}
```

# La interfaz Map



- Un Map (correspondencia) es un objeto que asocia una clave a un valor. También se denomina Diccionario.
- Métodos para añadir y borrar:
  - `put(Object key, Object value)`
  - `remove(Object key)`
- Métodos para la extracción de objetos
  - `get(Object key)`
- Métodos para obtener las claves, los valores y las parejas (clave, valor) como un conjunto
  - `keySet()` // devuelve un set
  - `values()` // devuelve un collection
  - `entrySet()` // devuelve un set

# La interfaz Map



```
public interface Map {
    // Basic Operations
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk Operations
    void putAll(Map t);
    void clear();

    // Collection Views
    public Set keySet();
    public Collection values();
    public Set entrySet();

    // Interface for entrySet elements
    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```

# Implementación de Map



- **HashMap**
  - Basado en una tabla hash
  - No se realiza ninguna ordenación en las parejas (clave, valor)
- **TreeMap**
  - Basado en un árbol rojo-negro
  - Las parejas (clave, valor) se ordenan sobre la clave

# Ejemplo



```
import java.util.*;
public class Freq {
    private static final Integer ONE = new Integer(1);
    public static void main(String args[]) {
        Map m = new HashMap();

        // Initialize frequency table from command line
        for (int i=0; i < args.length; i++) {
            Integer freq = (Integer) m.get(args[i]);
            m.put(args[i], (freq==null ? ONE :
                new Integer(freq.intValue() + 1)));
        }

        System.out.println(m.size()+ " distinct words detected:");
        System.out.println(m);
    }
}
```