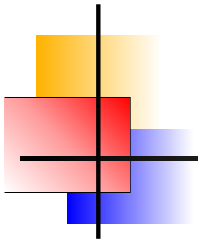


# Algoritmos y Estructuras de Datos:

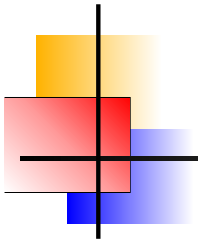
## *Ordenación y Colas con Prioridad*

**Guillermo Román Díez**

**groman@fi.upm.es**

Universidad Politécnica de Madrid

Curso 2015-2016



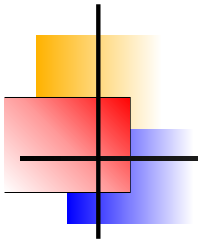
# Ordenación

- Es muy frecuente necesitar estructuras de datos que se encuentren ordenadas
  - Cuando se trata de tipos básicos hay un **orden total** entre sus elementos

*Pregunta*

¿qué ocurre con dos objetos? ¿cómo sabemos si o1 $>$ o2?

  - La ordenación entre dos objetos distintos tiene que establecerla el programador
  - La ordenación puede depender de los valores de un atributo, o de varios, o de una combinación de sus valores
  - pero ¿cómo hacemos esto en Java?



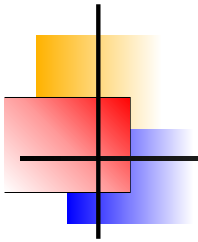
# Iterfaces Comparable<T> y Comparator<T>

- Mediante los interfaces `Comparable<T>` y `Comparator<T>` definimos ordenes totales entre objetos
- Interfaz 'java.lang.Comparable':

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

- Interfaz 'java.util.Comparator':

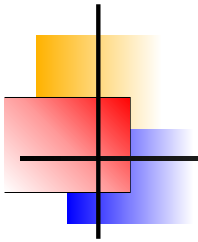
```
public interface Comparator<T> {
   public int compare(T t1, T t2);
}
```



# Interfaz Comparable<T>

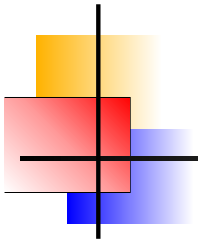
- La implementación de `Comparable<T>` por la clase `T` hace que ésta tenga que implementar el método `compareTo` para comparar dos objetos de tipo `T`
- La implementación del método fija el **orden natural** de los objetos de tipo `T`
- La llamada `t1.compareTo(t2)` donde `t1` y `t2` son de tipo `T`, `compareTo` devuelve un entero `i` tal que:
  - $i < 0$ si t1 es menor que t2.
  - $i == 0$ si t1 es igual que t2.
  - $i > 0$ si t1 es mayor que t2.
- Debe ser consistente con `equals`, es decir:

  t1.equals(t2) $\Leftrightarrow$ `t1.compareTo(t2) == 0`
- Normalmente este interfaz es implementado en los elementos contenidos en la estructura de datos



# Interaz Comparable<T>

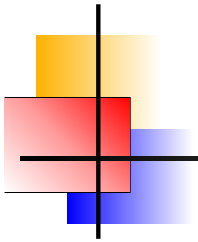
```
public class Alumno implements Comparable<Alumno> {
  int dni;
  String nombre, apellidos;
  ...
  public int compareTo(Alumno a) {
    return dni - a.getDni();
  }
  ...
}
```



# Interfaz Comparator<T>

- La implementaciones del interfaz `Comparator<T>` implementan un método de comparación `compare` para objetos de la clase `T`
- La llamada `c.compareTo(t1,t2)` donde `t1` y `t2` son de tipo `T`, y `c` es de tipo `Comparator<T>`, `compareTo` devuelve un entero `i` tal que:
  - $i < 0$ si t1 es menor que t2.
  - $i == 0$ si t1 es igual que t2.
  - $i > 0$ si t1 es mayor que t2.
- Debe ser consistente con `equals`, es decir:

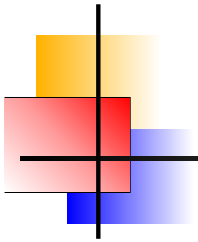
  t1.equals(t2) $\Leftrightarrow$ `t1.compareTo(t2) == 0`



## Comparator<T> a partir de Comparable<T>

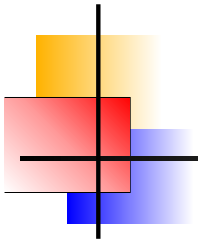
- Si un objeto es de una clase que implementa el interfaz `Comparable<T>` entonces se puede crear un objeto `Comparator<T>` por defecto para dicha clase:

```
class DefaultComparator<T> implements Comparator<T> {
  public int compare(T a, T b) {
    if (a instanceof Comparable<?>)
      return ((Comparable<T>) a).compareTo(b);
    else throw new UnsupportedOperationException();
  }
}

class DefaultComparator<T extends Comparable<T>>
              implements Comparator<T> {
  public int compare(T o1, T o2) {
    return o1.compareTo(o2);
  }
}
```



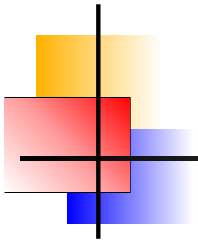
# Colas con Prioridad

- En una cola FIFO el primer elemento en entrar es el primero en salir
- En las *colas con prioridad* el orden de salida viene determinado por la **prioridad** del elemento
- Se puede ver una cola con prioridad como una estructura de datos en la que los elementos se almacenan en orden de prioridad
  - A nivel de implementación no es necesario que esto ocurra así, lo importante es que la cola devuelva el elemento con mayor prioridad
- Las **entradas** de una cola con prioridad tienen
  - Una **clave** que indica la prioridad del elemento
  - Un **valor** que indica el elemento a insertar
  - Al par *clave-valor* lo llamamos **entrada** (entry)



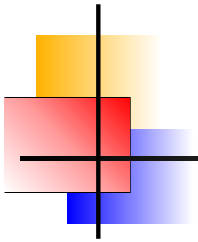
# Interfaz Entry<K,V

- En el interfaz `Entry<K,V>` tenemos:
  - **K** es la clave (key) de la entrada que vamos a insertar
  - **V** es el valor (value) que vamos a insertar
- Por convención: *la clave establece la prioridad inversamente: cuanto menor es la clave mayor es la prioridad*
  - También se conocen como min-max queue porque al desencolar se devuelve el elemento con la menor clave
  - Se utilizará el orden total. Los objetos que se usen para la clave deben ser ordenables
- Nos podemos encontrar con:
  - Dos o más entradas con la misma clave pero distintos valores
  - Dos o más entradas con el mismo valor pero distintas claves
  - Puede modificarse la clave o el valor de una entrada
- Se puede usar un atributo del valor como clave



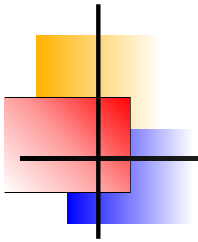
# Interfaz PriorityQueue<K,V>

```
public interface PriorityQueue<K,V> {

  public int size();

  public boolean isEmpty();

  public Entry<K,V> min() throws
      EmptyPriorityQueueException;

  public Entry<K,V> insert(K key, V value) throws
      InvalidKeyException;

  public Entry<K,V> removeMin() throws
      EmptyPriorityQueueException;
}
```



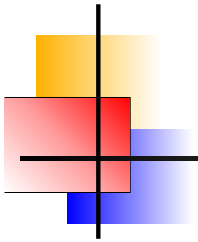
# Interfaz PriorityQueue<K,V>

- Los métodos `min`, `insert` y `removeMin` devuelven objetos que implementan el interfaz `Entry<K,V>`
- `insert`: recibe por separado una clave `key` y un valor `value` y devuelve el objeto `Entry<K,V>` insertado en la cola
- `min`: Es simplemente un método observador para recoger el elemento con mayor prioridad (con la clave con menor valor)
- `removeMin`: recoge el elemento con mayor prioridad (con la clave con menor valor) y lo borra de la cola
- `EmptyPriorityQueueException` se lanza cuando se intenta obtener o borrar la entrada de clave mínima en una cola vacía.
- `InvalidKeyException]` se lanza cuando la clave es `null` o no tiene definido un orden para los elementos de su clase



# Ejemplo de Colas con Prioridad

```
PriorityQueue<Integer,String> cola = new
    SortedListPriorityQueue<Integer,String>();

cola.insert(1, "Programacion II");
cola.insert(4, "Algor tmica Numerica");
cola.insert(3, "Lenguajes y Aut matas");
cola.insert(0, "AED");

while (cola.size() > 0)
  ...println(cola.removeMin());
```



# Implementación de Colas con Prioridad

- Con una **lista desordenada**
  - `insert` tiene complejidad $O(1)$
  - `min` tiene complejidad $O(n)$
  - `removeMin` tiene complejidad $O(n)$
- Con una **lista desordenada con caché mínimo**
  - `insert` tiene complejidad $O(1)$
  - `min` tiene complejidad $O(1)$
  - `removeMin` tiene complejidad $O(n)$
- Con una **lista ordenada**
  - `insert` tiene complejidad $O(n)$
  - `min` tiene complejidad $O(1)$
  - `removeMin` tiene complejidad $O(1)$