DAM/DAW

PROGRAMACIÓN

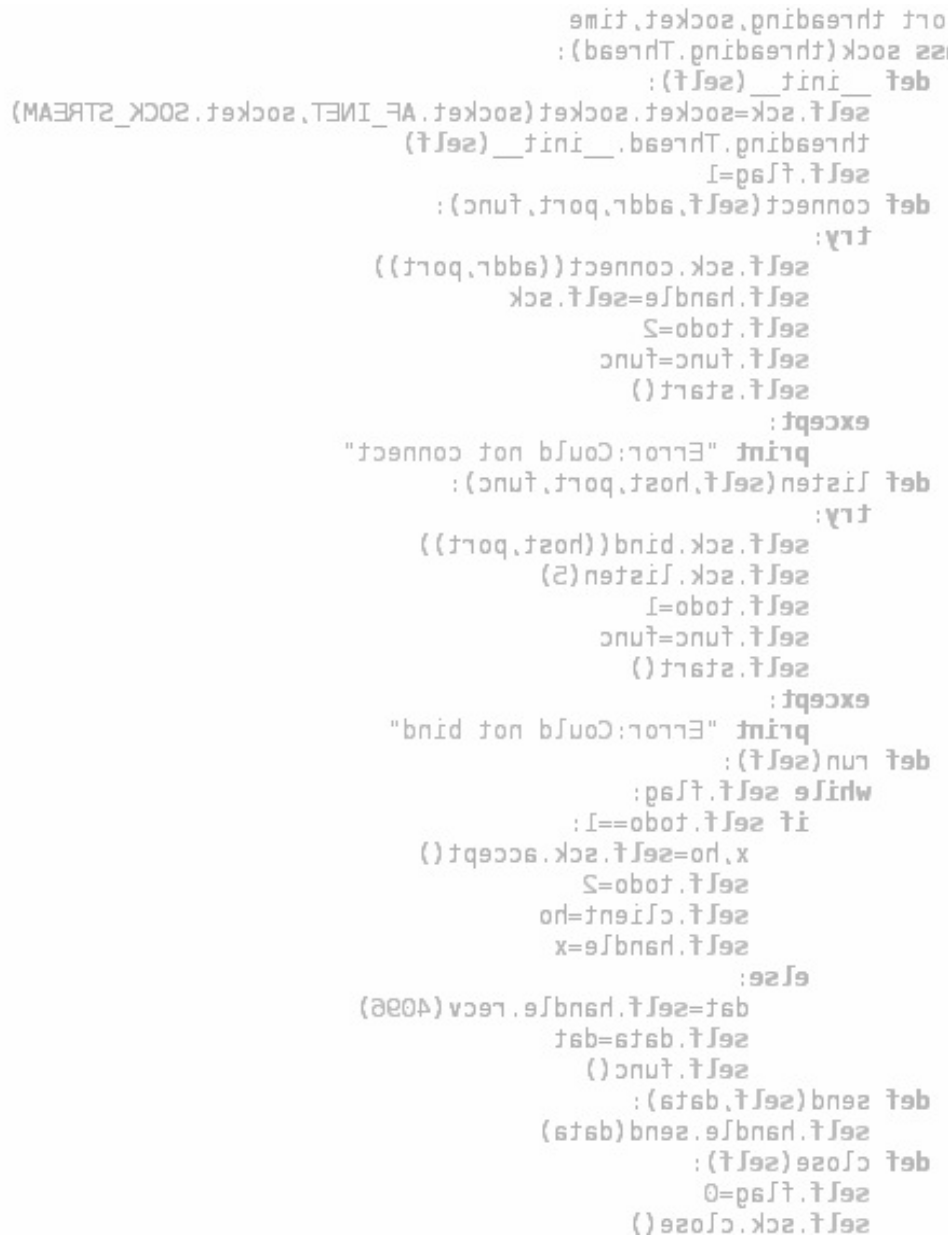
08.3

Java Collections Framework II

Rev: 3.1

fran@iessanclemente.net

Indice



El interfaz *Set*

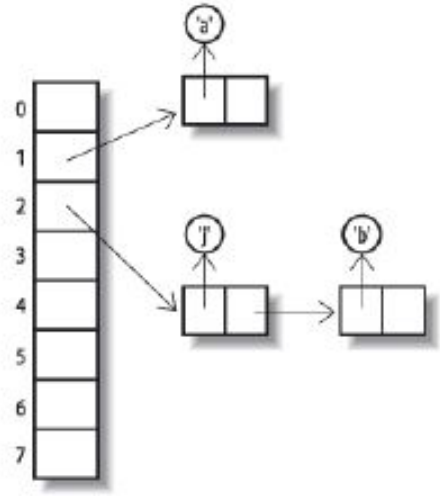
- Un conjunto (*set*) es una colección de objetos que no puede contener duplicados. Añadir un elemento ya presente en la colección no tendrá ningún efecto
- El interfaz *Set* define los mismos métodos que el interfaz *Collection*
- El JFC proporciona múltiples implementaciones del interfaz *Set*: *HashSet*, *EnumSet*, *AbstractSet*, *LinkedHashSet*, *TreeSet*, *CopyOnWriteArraySet*,...
- La diferencia fundamental entre ellas estriba en su rendimiento en relación a operaciones de lectura y/o búsquedas, así como en el orden en que sus iteradores devuelven los elementos. Estas diferencias de rendimiento son debidas al tipo de estructura interna que emplean para el almacenamiento de los datos
- Si bien no las veremos todas, es conveniente tener una idea general de las mismas con objeto de seleccionar la más adecuada para nuestras intereses

La clase *HashSet* (I)

- Es la implementación más usada de *Set*
- Como su nombre indica, se implementa mediante un *tabla hash*
- Una *tabla hash* básicamente es un *array* donde cada uno de sus elementos se asocia a una posición calculada a partir del propio contenido que se almacenará en dicha posición
- Tradicionalmente, para determinar la posición de la tabla (*array*) que ocupará un elemento, se aplica un función *hash* al contenido de dicho elemento. A continuación, se usa el resto de la división entera del valor *hash* entre el tamaño del *array* como *índice* (posición) en el array
- En el caso concreto de Java, se emplea el valor *hash* devuelto por cada objeto (método *hashCode( )* de *Object)* y se aplica una máscara sobre los *bits* menos significativos de dicho valor

La clase *HashSet* (II)

- Evidentemente, la limitación del tamaño del *array* subyacente va a hacer que diferentes elementos (aún on *hash* diferentes) puedan "caer" en la misma posición. Para solucionarlo se suelen emplear *listas enlazadas*



- Por tanto, aún con buenas funciones *hash* que amplíen y separen el espacio de claves, a mayor ocupación del *array*, mayor probabilidad de colisión, lo que redunda en una pérdida de rendimiento al tener que crear nuevas estructuras y operar sobre ellas
- Para evitar este problema, cuando el *array* alcanza un determinado nivel de ocupación (*load factor*), las implementaciones de este tipo de estructuras suelen redimensionar automáticamente el *array* creando una nueva tabla de mayor capacidad sobre la que se copia la actual

La clase *HashSet* (III)

- Iterar sobre una tabla *hash* supone examinar cada una de las posiciones del *array* para determinar si está ocupada o no. Y, en el caso de que esté ocupada por más de un elemento, recorrer en orden la lista enlazada asociada a dicha posición
- Dado que la posición que ocupa un elemento en la tabla *hash* depende de su propio contenido, y no del orden o instante en que se añadió a la tabla, no podemos garantizar el orden en que un iterador sobre la tabla nos devolverá su contenido (como veremos en el próximo ejemplo)
- La principal ventaja de este tipo de estructuras de almacenamiento es que, idealmente, el acceso de lectura/escritura de un elemento cualquiera de la colección sería constante (no depende del tamaño del conjunto). Como contrapartida, presentan un rendimiento pobre en las iteraciones, pues depende del tamaño del *array* (espacio), no de la colección (datos)

La clase *HashSet* (y IV)

- La ordenación interna del *HashSet* no tiene que coincidir necesariamente con el orden de entrada de los elementos en el conjunto:

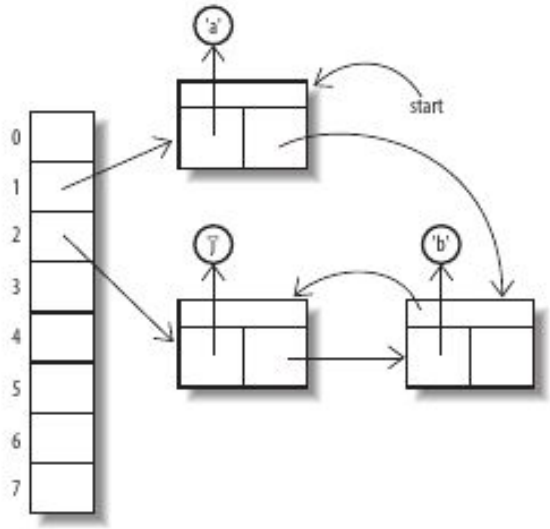
```
import java.util.HashSet;
public class SetDemo1 {
   public static void main(String[] args) {
       HashSet<Character> set = new HashSet<>();
       set.add('B'); // Devuelve True
       set.add('A'); // Devuelve True
       set.add('1'); // Devuelve True
       set.add('0'); // Devuelve True
       set.add('a'); // Devuelve True
       set.add('1'); // Añadir más de una vez el mismo elemento no tiene ningún efecto
                     // en la colección. El método devolverá False
       System.out.println(set); // El orden no tiene que ser el de inserción
   }
}
```

```
[0, A, 1, a, B]
```

La clase *LinkedHashSet* (I)

- Esta clase, que hereda de *HashSet*, tiene como finalidad la de garantizar que el orden en que su iterador devuelve los elementos, es el mismo en el que fueron introducidos

- Para ello mantiene, paralelamente a la *tabla hash*, una lista enlazada que referencia los elementos del conjunto según haya sido su orden de inserción



- La sobrecarga debida al mantenimiento de la nueva estructura, se compensa con un mejor rendimiento al iterar sobre la colección

- Por tanto, será preferible un *LinkedHashSet* sobre un *HashSet* cuando el orden de inserción o la eficiencia en la iteración sobre la colección sean importantes para la aplicación

La clase *LinkedHashSet* (II)

- El *LinkedHashSet* recorre la colección manteniendo el orden de la inserción:

```
import java.util.LinkedHashSet;
public class SetDemo2 {
   public static void main(String[] args) {
       LinkedHashSet<Character> set = new LinkedHashSet<>();
       set.add('B'); // Devuelve True
       set.add('A'); // Devuelve True
       set.add('1'); // Devuelve True
       set.add('0'); // Devuelve True
       set.add('a'); // Devuelve True
       set.add('1'); // Añadir más de una vez el mismo elemento no tiene ningún efecto
                     // en la colección. El método devolverá False
       System.out.println(set); // Mismo orden que el de inserción
   }
}
```

```
[B, A, 1, 0, a]
```

La clase TreeSet (I)

- El interfaz *Set* tiene un subinterfaz, *SortedSet*, que garantiza que el iterador sobre la colección la recorrerá en orden ascendente (orden natural), con independencia de cuál haya sido el orden concreto de inserción de los elementos
- En JDK6, se añadió un subinterfaz de *SortedSet*, *NavigableSet*, que añade nuevos métodos para encontrar elementos cercanos a uno dado. Métodos como *lower*, *floor*, *ceiling*, y *higher* devuelven respectivamente el elemento *menor que*, *menor o igual que*, *mayor o igual que*, y *mayor que* uno dado
- JFC proporciona una implementación de estos interfaces, la clase *TreeSet*
- Internamente, con objeto de mantener ordenados los elementos de la colección, esta clase emplea una estructura de árbol binario de búsqueda balanceado o *red-black tree*. A medida que se van insertando elementos en la colección, estos se irán "colocando" de forma ordenada sobre el árbol

La clase TreeSet (II)

- Algunos métodos adicionales de la clase *TreeSet* (*SortedSet*/*NavigableSet*)

Collection<E>	**Descripción**
E **first**()	Devuelve el primer elemento de el conjunto
E **last**()	Devuelve el último elemento de el conjunto
E **pollFirst**()	Devuelve y elimina el primer elemento del conjunto (menor)
E **pollLast**()	Devuelve el último elemento del conjunto (mayor)
E **ceiling**(E e)	Devuelve el menor elemento del conjunto mayor o igual a e
E **floor**(E e)	Devuelve el mayor elemento del conjunto menor o igual a e
E **higher**(E e)	Devuelve el menor elemento del conjunto mayor que e
E **lower**(E e)	Devuelve el mayor elemento del conjunto menor que e
NavigableSet<E> **descendingSet**()	Devuelve un conjunto con los elementos en orden inverso
Iterator<E> **descendingIterator()**	Devuelve un iterador sobre la conjunto de orden inverso
NavigableSet<E> **subSet**(E from, boolean incFrom, E to, boolean incTo)	Devuelve un subconjunto desde los elementos from y to (que se incluirán o no en función de los valores incFrom y incTo)
NavigableSet<E> **headSet**(E e, boolean inc)	Devuelve el subconjunto hasta el elemento e
NavigableSet<E> **tailSet**(E e, boolean inc)	Devuelve el subconjunto desde el elemento e

La clase TreeSet (y III)

- Ejemplos de uso de *TreeSet:*

```
import java.util.*;
public class SetDemo3 {
   public static void main(String[] args) {
       TreeSet<Integer> ts = new TreeSet<>();
       ts.add(4); ts.add(-1); ts.add(0);
       ts.add(2); ts.add(7); ts.add(5);
       ts.add(3); ts.add(3); ts.add(3);  // Sólo almacenará uno
       System.out.println("ts: " + ts);  // Estarán ordenados de menor a mayor
       System.out.println("Mayor: " + ts.last() + ", Menor: " + ts.first());
       System.out.println("Primer elemento mayor que 0: " + ts.higher(0));
       System.out.println("Primer elemento menor o igual que 6: " + ts.floor(6));
       NavigableSet<Integer> range = ts.subSet(-5, true, 3, false);
       System.out.println("Elementos en el rango [-5, 3): " + range);
   }
}
```

```
ts: [-1, 0, 2, 3, 4, 5, 7]
Mayor: 6, Menor: -1
Primer elemento mayor que 0: 2
Primer elemento menor o igual que 6: 5
Elementos en el rango [-5, 3): [-1, 0, 2]
```

Comparación y ordenación (I)

- Como acabamos de comprobar, la colección *TreeSet* mantiene los elementos de colección ordenados según su valor
- Esto no nos supone ninguna consideración adicional cuando creamos conjuntos de elementos de las diferentes clases *wrapper* de los tipos primitivos (*Integer, Double,...*) u otras clases de Java (*String,...*)
- Sin embargo, cuando almacenamos instancias de nuestras propias clases (Cliente, Libro, Producto, Usuario,...), ¿qué criterio emplea *TreeSet* para ordenar esos objetos?
- La realidad es que no va a poder ordenarlos, lo que derivará en un error de compilación. Va a necesitar cierta "colaboración" por nuestra parte. Tedremos que "decirle" cómo comparar las instancias de nuestras clases
- Esto lo podemos realizar de dos maneras diferentes. Una es que nuestras clases implementen el interfaz *Comparable*, de forma que "sepan" ordenarse. La otra será mediante la creación de un *Comparator* externo

Comparación y ordenación (II)

❖ Implementando el interfaz *Comparable*

- Supongamos que creamos una aplicación de registro de usuarios que define la siguiente clase:

```
package pruebas.jcf;
import java.time.LocalDateTime;
public class User {
    private int id;                     // id del usuario
    private String loginName;           // login de inicio de sesión
    private LocalDateTime lastLogin;    // último inicio de sesión

    User(int id, String loginName) {
       this.id = id;
       this.loginName = loginName;
    }
    public LocalDateTime getlLastLogin() { return this.lastLogin; }
    public void regSystemLogin() { this.lastLogin = LocalDateTime.now(); }
    @Override
    public String toString() {
        return "{" + this.id + ": " + this.loginName + ": " + this.lastLogin + "}";
    }
}
```

Comparación y ordenación (III)

- Si dejáramos la definición de la clase *User* tal como está, cualquier intento de crear un *TreeSet* que contuviera elementos de la misma derivaría en un error de compilación:

```
Exception in thread "main" java.lang.ClassCastException:
class User cannot be cast to class java.lang.Comparable
```

- Para solucionar este problema, implementaremos el interfaz *Comparable*, proporcionándole así al *TreeSet* un mecanismo para la ordenación de los objetos de la clase *User*
- El interfaz *Comparable* sólo define un método, *compareTo( )*, que deberemos implementar en nuestra clase *User*

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

Comparación y ordenación (IV)

- El método *compareTo* recibirá una instancia de la misma clase (en este caso *User*) y devolverá un valor entero negativo, cero, o un valor entero positivo si el propio objecto (*this*) es menor que, igual a, o mayor que el objeto pasado como argumento.
- Aunque no estrictamente requerido, es altamente recomendable que se verifique:

```
(x.compareTo(y)==0) == (x.equals(y))
```

  Es decir, que si dos objetos son iguales según *compareTo( )*, también lo sean según *equals( )*

- Además, tal como vimos en su momento al estudiar las comparaciones y búsquedas en listas, sobreescribir el método *equals*( ) nos obliga a sobreeescribir el método *hashCode( )*, de forma que dos objetos iguales según *equals( )* devuelvan el mismo valor de *hashCode*

Comparación y ordenación (V)

- Así, nuestra clase quedaría:

```
package pruebas.jcf;
import java.time.LocalDateTime;
public class User implements Comparable<User> {
    //...
    @Override
    public int compareTo(User other) {
        // Comparamos los id de ambos usuarios. Podemos aprovechamos del propio método compareTo()
        // de la clase Integer pues la clase Integer implementa el interfaz Comparable
        return Integer.valueOf(this.id).compareTo(other.id);
    }
    @Override
    public boolean equals(User other) {
        return this.id == other.id;
    }
    @Override
    public int hashCode() {
        int result = Integer.hashCode(this.id);
        result = 31*result + this.loginName.hashCode();
        result = 31*result + ((this.lastLogin==null)? 0: this.lastLogin.hashCode());
        return result;
    }
}
```

Comparación y ordenación (VI)

- Ahora ya podemos añadir instancias de *User* a un *TreeSet*:

```
package pruebas.jcf;
import java.util.TreeSet;

public class SetDemo4 {
   public static void main(String[] args) {
       TreeSet<User> users = new TreeSet<>();

       users.add(new User(55, "Poole"));
       users.add(new User(3, "Bowman"));
       users.add(new User(0, "HAL"));

       // Estarán ordenados de menor a mayor
       System.out.println("users list: " + users);
   }
}
```

```
users list: [{0: HAL: null}, {3: Bowman: null}, {55: Poole: null}]
```

Comparación y ordenación (VII)

❖ Crendo un *Comparator*

- Alternativamente, podemos optar por crear una clase externa que implemente el interfaz *Comparator* y se encargue de realizar la comparación de los objetos *User*
- De los diferentes métodos de *Comparator* (muchos estáticos o con implementaciones por defecto) nos interesa uno en concreto, *compare( )*, que debe mantener la coherencia con el método *equals( )* del objeto

```
public interface Comparator<T> {
    // negativo si o1<o2; 0 si o1==o2, positivo si o1>02
    int compare(T o1, To2);
}
```

- Los objetos *Comparator* pueden pasarse al método *sort( )*, como el de *Collections* o *Arrays*, para controlar el orden en la ordenación

Comparación y ordenación (VIII)

- Las instacias de *Comparator* nos van a permitir, por un lado, proporcionar un mecanismo de ordenación a aquellas clases que no implementen *Comparable* y, por tanto, no tienen capacidad para "autoordenarse". Por otro, para aquellas que si lo hacen, nos da la posibilidad de establecer criterios diferentes de comparación y ordenación
- Vamos a probar este segundo supuesto con nuestra clase *User*. Para ello, definiremos una nueva clase, *UserComparator*, que implemente el interfaz *Comparator* y nos permita ordenar las instancias de *User* en base al último inicio de sesión en el sistema (*lastLogin*) en lugar del id de usuario
- Esta nueva clase deberá implementar el método *compare( )*. El método recibirá dos instancias de *User* como argumento y devolverán el resultado correspondiente a la comparación entre ellos
- Finalmente, pasaremos una instancia de este *Comparator* en el constructor del *TreeSet* que contendrá las instancias de *User*

Comparación y ordenación (IX)

- Nuestro *Comparator* para *User* quedaría:

```
package pruebas.jcf;
import java.time.LocalDateTime;

class UserComparator implements Comparator<User> {
    @Override
    public int compare(User u1, user u2) {
        // Comparamos el lastLogin de ambos usuarios.
        // Podemos aprovechamos del propio método compareTo() de la clase LocalDateTime
        // ya que implementa el interfaz Comparable
        return u1.getLastLogin().compareTo(u2.getLastLogin());
    }
}

public class User implements Comparable<User> { //... }
```

Comparación y ordenación (y X)

- Probemos ahora un nuevo *TreeSet* de *User* con una ordenación diferente

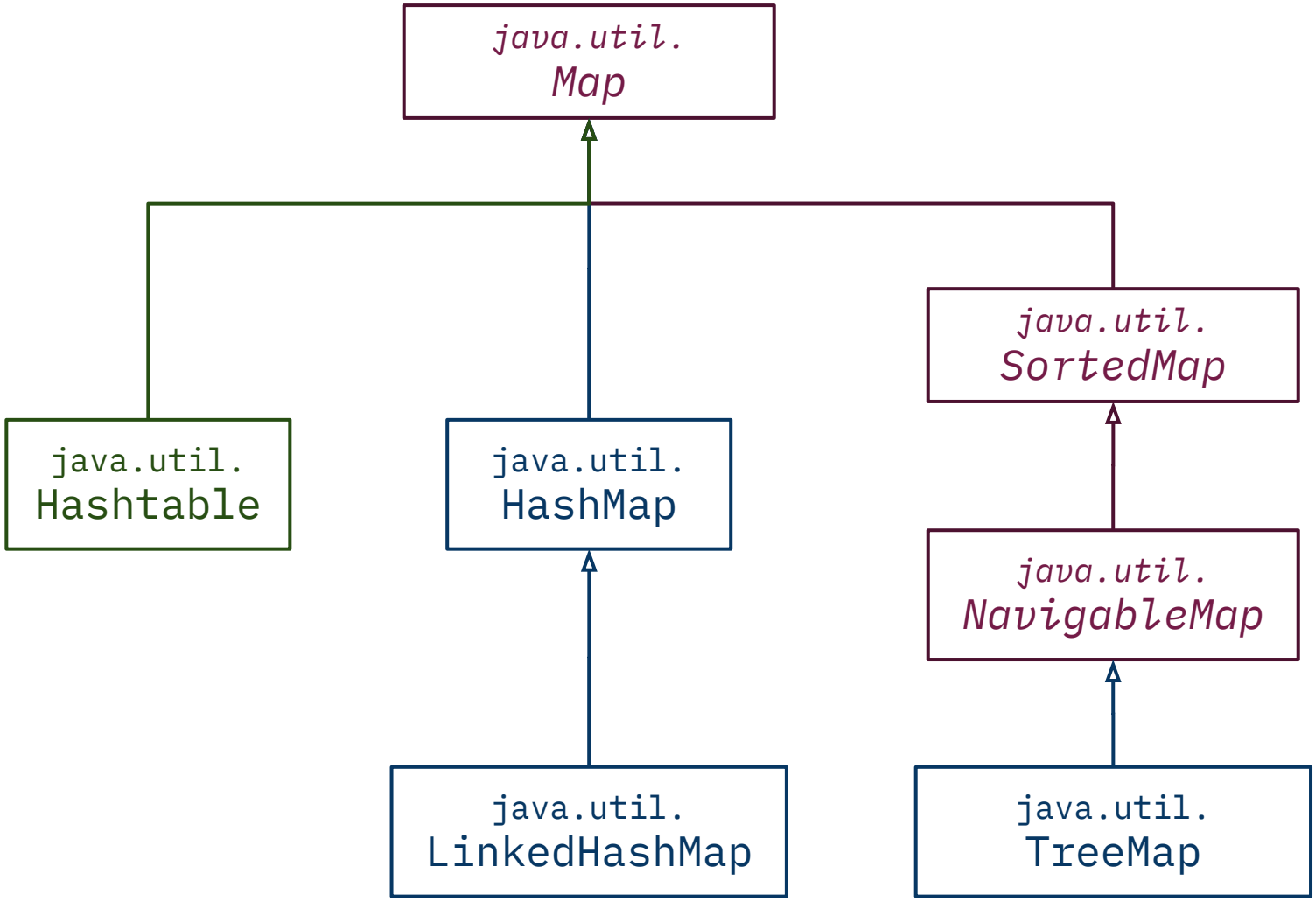
```
package pruebas.jcf;
import java.util.*;
public class SetDemo5 {
   public static void main(String[] args) {
       TreeSet<User> users2 = new TreeSet<>(new UserComparator()); // Pasamos una instancia del comparador

       User poole = new User(55, "Poole");
       poole.regSystemLogin();
       User bowman = new User(3, "Bowman");
       bowman.regSystemLogin();
       User hal = new User(0, "HAL");
       hal.regSystemLogin();

       users2.add(hal);
       users2.add(bowman);
       users2.add(poole);

       System.out.println("login: " + users2);  // Estarán ordenados por el timestamp del login, no por id
   }
}
```

```
login: [{55: Poole: 2001-01-01T00:00:00.771308}, {3: Bowman: 2001-01-01T00:00:00.773578}, {0: HAL: 2001-01-01T00:00:00.773593}]
```

El interfaz *Map* (I)

- El interfaz *Map* es otro de los principales interfaces del JCF y el único que no deriva de *Collection*
- Define las operaciones relacionadas con conjuntos de parejas *clave:valor* donde las claves son únicas
- Se basa en una estructura dinámica de tipo *diccionario* cuyos nodos, o entradas, son instancias de la clase interna *Map.Entry<K, V>*
- Su función principal es la de almacenar elementos asociados a una clave. Emplearemos esas claves para acceder a los elementos del mapa
- Existen tres implementaciones principales de *Map*: *HashMap*, *LinkedHashMap* y *TreeMap*. Como en el caso de las implementaciones de *Set*, estas tres variantes de *Map* emplean estructuras internas diferentes y, por tanto, comportamientos diferentes en relación al orden y acceso secuencial a los elementos contenidos

El interfaz *Map* (II)



El interfaz *Map* (III)

❖ Añadir elementos:

V **put**(K key, V value)	Añade o reemplaza la entrada del mapa de clave key Devuelve el valor antiguo si la entrada existe o *null*
void **putAll**(Map<? extends K, ? extends V> m)	Añade todas las entradas del mapa m

❖ Eliminar elementos:

void **clear**()	Elimina todas las entradas del mapa
V **remove**(Object k)	Elimina la entrada de clave k Devuelve el valor antiguo si la entrada existe o *null*

❖ Consultar contenido:

V **get**(Object k)	Devuelve el valor asociado a la clave k o null si la clave no existe
boolean **containsKey**(Object k)	Devuelve *true* si existe una entrada de clave k o *false* si no existe
boolean **containsValue**(Object v)	Devuelve *true* si existe una entrada de valor v o *false* si no existe
int **size**()	Devuelve el número de entradas en el mapa
boolean **isEmpty**()	Devuelve *true* si no existe ninguna entrada en el mapa

El interfaz *Map* (IV)

❖ Obtener colecciones (vistas) a partir de las claves, valores o entradas:

`Set<Map.Entry<K, V>> entrySet()`	Devuelve un *set* con las entradas del mapa
`Set<K> keySet()`	Devuelve un *set* con las claves del mapa
`Collection<V> values()`	Devuelve una colección con los valores del mapa

- Las colecciones devueltas por estos métodos contienen referencias a los elementos del mapa. Cualquier modificación en una entrada del mapa se reflejará en la entrada correspondiente de la colección (y viceversa)
- De hecho, sólo se pueden realizar un conjunto limitado de operaciones sobre las vistas (*sets*). Están permitidas las eliminaciones, directamente o a través de un iterador sobre la vista, pero no se podrán añadir nuevos elementos. Eliminar una clave supone la eliminación de la entrada del mapa. Eliminar un valor, supone la eliminación de una de las entradas del mapa que contengan ese valor (puede haber más de una)

El interfaz *Map* (y V)

❖ El interfaz *Map.Entry*

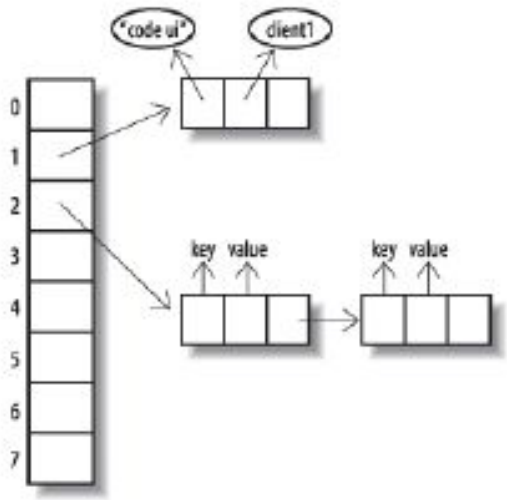
- Representa una entrada del mapa, formada por una pareja clave:valor
- El método *entrySet( )* del interfaz *Map* nos proporciona una colección (vista limitada) de *Map.Entry* con todas las entradas del mapa
- Podemos obtener referencias de tipo *Map.Entry* mediante el iterador del mapa (por ejemplo, al hacer un *for-each*). Sin embargo, estas referencias sólo serán válidas durante la iteración
- Algunos de los métodos proporcionados por este interfaz son:

`K` **`getKey`**`()`	Devuelve la clave correspondiente a esta entrada
`V` **`getValue`**`()`	Devuelve el valor correspondiente a esta entrada
`V` **`setValue`**`(V value)`	Reemplaza el valor correspondiente a esta entrada por `value` Devuelve el valor antiguo de esta entrada

Implementaciones de *Map* (I)

❖ *HashMap*

- Como en el caso de *HashSet* emplea una tabla *hash* para referenciar las diferentes entradas, lo que deriva en buenos tiempos de acceso aleatorio (en tiempo constante, si no hay colisiones)



- Su mayor limitación viene dada por su bajo rendimiento de iteración, pues depende del tamaño de la estructura interna y no por el número de entradas en el mapa

- Además, no es posible determinar el orden en que serán devueltas sus entradas por un iterador sobre el mapa, pues la posición de cada una depende del valor *hash* de su clave
- Aconsejable cuando hay una mayoría de operaciones *put( )* y *get( )*

Implementaciones de *Map* (II)

❖ *LinkedHashMap*

- Como en el caso de *LinkedHashSet*, la clase *LinkedHashMap* "refina" el comportamiento de su superclase *HashMap* garantizando el orden en que los iteradores devuelven las entradas del mapa, que será el de inserción
- Presenta mejor rendimiento que *HashMap* al iterar sobre el mapa

❖ *TreeMap*

- Como su contraparte *TreeSet* para los conjuntos, emplea una estructura de árbol para almacenar los elementos por orden natural de la clave
- Buen rendimiento en operaciones de búsqueda, acceso y eliminación
- Tanto para *LinkedHashMap* como para *TreeMap* se aplican los mismos requisitos para las igualdades y comparaciones de las claves que los ya vistos para los elementos de *LinkedHashSet* y *TreeSet*

Implementaciones de *Map* (III)

- Algunas operaciones sobre mapas:

```
import java.util.*;
class MapaDemo1 {
   public static void main(String[] args) {
        LinkedHashMap<Integer, String> map = new LinkedHashMap<>();
        TreeMap<Integer, String> tree = new TreeMap<>();
        map.put(3, "tres"); tree.put(3, "tres");
        map.put(1, "uno"); tree.put(1, "uno");
        map.put(2, "dos"); tree.put(2, "dos");
        System.out.println("HashMap = " + map);    // mantiene el orden de inserción
        System.out.println("TreeMap = " + tree);   // orden natural (creciente) de clave
        map.replace(2, "two");
        System.out.println("map[2] = " + map.get(2));
        if(!tree.containsKey(0)) { tree.put(0, "zero"); }
        System.out.println("TreeMap = " + tree);
   }
}
```

```
HashMap = {3=tres, 1=uno, 2=dos}
TreeMap = {1=uno, 2=dos, 3=tres}
map[2] = two
TreeMap = {0=zero, 1=uno, 2=dos, 3=tres}
```

Implementaciones de *Map* (y IV)

❖ Iterando sobre el mapa:

```
import java.util.*;
class MapaDemo2 {
   public static void main(String[] args) {
       LinkedHashMap<Integer, String> map = new LinkedHashMap<>();
       map.put(1, "uno"); map.put(2, "dos"); map.put(3, "tres");

       System.out.println("Claves = " + map.keySet());     // set-vista de claves
       System.out.println("Valores = " + map.values());    // set-vista de valores
       System.out.println("Entradas = " + map.entrySet()); // set-vista de entradas

       Set<Integer> claves = map.keySet();
       for(Integer k: claves) { System.out.println("clave: " + k + "-> val: ") + map.get(k); }

       Set<Map.Entry<Integer, String>> entradas = map.entrySet();
       for(Map.Entry<Integer, String> e: entradas) {
           System.out.println("clave: " + e.getKey() + "-> val: ") + e.getValue();
       }
   }
}
```

```
Claves: [1, 2, 3]
Valores: [uno, dos, tres]
Entradas: [1=uno, 2=dos, 3=tres]
```

```
clave: 1 -> valor: uno
clave: 2 -> valor: dos
clave: 3 -> valor: tres
```

```
clave: 1 -> valor: uno
clave: 2 -> valor: dos
clave: 3 -> valor: tres
```