UNIDAD DIDÁCTICA 5: COLECCIONES

Índice

1) BUCLE FOR MEJORADO

A partir de la versión Java 5, existe otra forma de la sentencia "for" diseñada para iterar en los arrays y colecciones que ya hemos visto por encima en temas anteriores. Puede usarse para hacer los bucles más compactos y fáciles de leer y suele usarse mucho en colecciones.

La sintaxis empleada es:

```
for (TipoARecorrer nombreVariableTemporal : nombreDeLaColección ) {
        Instrucciones;
}
```

Fíjate que en ningún momento se usa la palabra clave **each** que se usa en otros lenguajes, aunque al for muchas veces se le nombre como for each. Para saber si un for es un for extendido o un for normal hemos de fijarnos en la sintaxis que se emplea. La interpretación que podemos hacer de la sintaxis del for extendido es: ***"Para cada elemento del tipo TipoARecorrer que se encuentre dentro de la colección nombreDeLaColección ejecuta las instrucciones que se indican"***. La variable local-temporal del ciclo almacena en cada paso el objeto que se visita y sólo existe durante la ejecución del ciclo y desaparece después. Debe ser del mismo tipo que los elementos a recorrer. Observa los siguientes ejemplos:

Nota: No te preocupes por no conocer qué es eso de ArrayList, lo veremos enseguida.

```
List <String> listaDeNombres= new ArrayList <String>();

 public void listarTodosLosNombres () {

        for (String nombre: listaDeNombres) {

            System.out.println (nombre); //Muestra cada uno de los
            nombres dentro de listaDeNombres, siempre llega hasta el final

        }

 }
```

```
public class Ppal {

    public static void main (String [ ] args) {

        int [ ] numeros = {1,3,5,7,9,11,13,15,17,19};

        int suma = 0;

        for (int valor : numeros ) {

            suma =suma + valor;

            System.out.println (valor); //Para ver el valor del elemento

        }

        System.out.println ("La suma es:  " + suma);

    }

}
```

//valor se refiere al valor guardado en el array y no al índice como en los bucles normales. números es el nombre del array

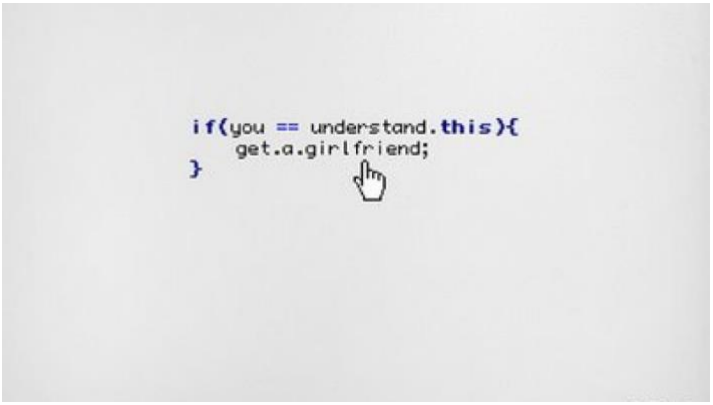
El for extendido tiene algunas ventajas y algunos inconvenientes. No se debe usar siempre. Su uso no es obligatorio, de hecho, como hemos indicado, en versiones anteriores ni siquiera existía en el lenguaje. En vez de un for extendido podemos preferir usar los bucles normales.

El uso del bucle for-each tiene algunos inconvenientes. Uno de ellos, que para recorrer la colección nos basamos en la propia colección y por tanto no podemos (o al menos no debemos) manipularla durante su recorrido.

El ciclo for-each es una herramienta muy útil cuando tenemos que realizar recorridos completos de colecciones, por lo que lo usaremos en numerosas ocasiones antes que los bucles for o while, que nos obligan a estar pendientes de más cuestiones, por ejemplo, de llevar un contador, llamar en cada iteración a un método, etc. Un for extendido, en principio, recorre todos y cada uno de los elementos de una colección. Sin embargo, para que no llegue al final, podemos introducir un condicional asociado a una sentencia break; que aborte el recorrido una vez se cumpla una determinada condición, aunque nosotros no lo haremos.

```
ArrayList <String> jugadoresDeBaloncesto = new ArrayList<String> ();
jugadoresDeBaloncesto.add ("Michael Jordan");
jugadoresDeBaloncesto.add ("Kobe Briant");
jugadoresDeBaloncesto.add ("Pau Gasol");
jugadoresDeBaloncesto.add ("Ángel Naranjo");

int i = 0;
System.out.println ("Los jugadores de baloncesto en la lista son: ");

for (String nombre : jugadoresDeBaloncesto) {
       System.out.println ((i+1) + ".- " +nombre);
       i++;
}
```

NOTA: Esta forma de for es muy útil para las colecciones.



2) API COLLECTIONS

a) Introducción

Una *colección* es un solo objeto que administra un grupo o conjunto de elementos de un tipo en una sola unidad. Los objetos de la colección se llaman *elementos* y se utilizan para almacenamiento, recuperación y manipulación de datos.

Suelen representar elementos que forman grupos naturales, por ejemplo, una colección de mensajes (buzón de correo electrónico), colección de productos (carrito de la compra), colección de parejas nombre-datos (agenda de contactos).

Java dispone de clases e interfaces para trabajar con colecciones de objetos.

En la versión 1.2 del JDK se introdujo el ***Java Framework Collections*** o "estructura de colecciones de Java" (en adelante JCF). Se trata de un conjunto de clases e interfaces que mejoran notablemente las capacidades del lenguaje respecto a estructuras de datos. Además, constituyen un excelente ejemplo de aplicación de los conceptos propios de la *programación orientada a objetos*.

Nota: Puedes echar un vistazo a toda la documentación sobre esto en el api de java versión 11.

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/doc-files/coll-overview.html

Ha habido novedades a lo largo de las diversas versiones de Java que veremos más adelante, por ejemplo, en java 5 aparece el uso de genéricos, el operador diamond en java 7, en Java 8 aparecen las expresiones Lambda y los Streams, factorías en Java 9...

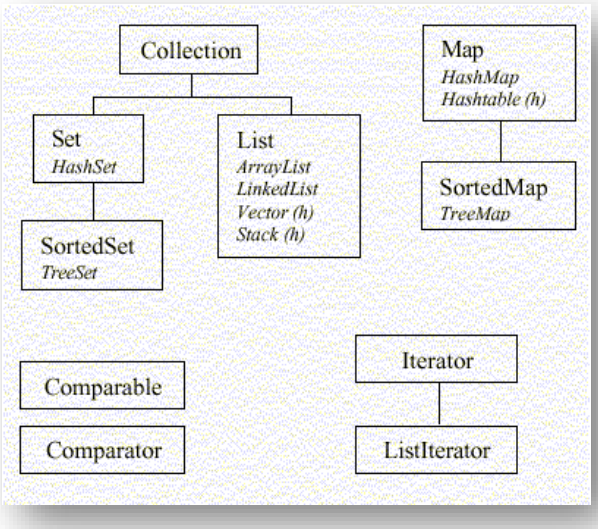
La Figura de la izquierda siguiente muestra la jerarquía de interfaces de la ***Java Collection Framework*** (JCF). En *letra cursiva* se indican las clases que implementan las correspondientes interfaces. Por ejemplo, hay dos clases que implementan la interface ***Map***: ***HashMap*** y ***Hashtable***.

Las clases que aparecen con la letra "h" se denominan clases "históricas", es decir, clases que existían antes de la versión JDK 1.2. Aunque dichas clases se han mantenido por motivos de compatibilidad, sus métodos no siguen las reglas del diseño general del JCF; en la medida de lo posible se recomienda utilizar las nuevas clases.

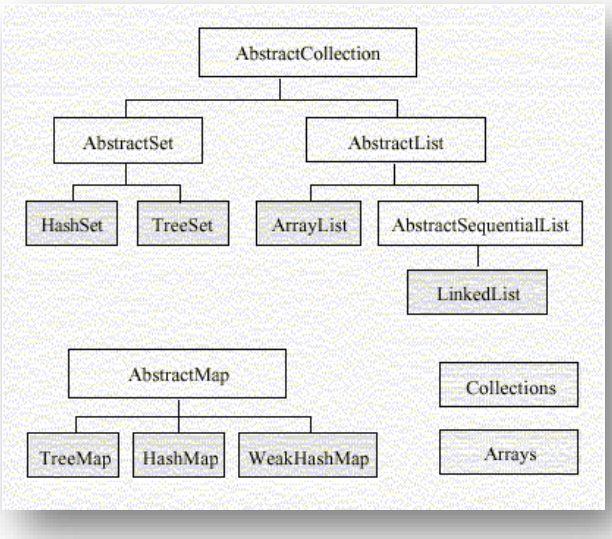
La Figura de la derecha muestra la jerarquía de clases de la JCF. En dicha figura se muestran con fondo blanco las clases abstractas, y con fondo gris claro las clases de las que se pueden crear objetos (no abstractas).

Las ventajas del framework de colecciones pueden ser:

- Reduce el esfuerzo de programación.
- Aumenta la calidad y velocidad del programa.
- Permite la interoperabilidad con librerías de terceros.
- Reduce el esfuerzo para aprender y usar otras librerías.
- Reduce el esfuerzo para diseñar nuevas librerías.
- Fomenta la reutilización de software



Interfaces de la Collection Framework



Jerarquía de clases de la Colletion Framework

En el diseño de la JCF las ***interfaces*** son muy importantes porque son ellas las que determinan las capacidades de las clases que las implementan. Dos clases que implementan la misma interface se pueden utilizar exactamente de la misma forma. Por ejemplo, las clases ***ArrayList*** y ***LinkedList*** disponen exactamente de los mismos métodos y se pueden utilizar de la misma forma. La diferencia está en la implementación: mientras que ***ArrayList*** almacena los objetos en un array, la clase ***LinkedList*** los almacena en una lista vinculada. La primera será más eficiente para acceder a un elemento arbitrario, mientras que la segunda será más flexible si se desea borrar e insertar elementos.

Las clases ***Collections*** y ***Arrays*** son un poco especiales: no son ***abstract***, pero no tienen constructores públicos con los que se puedan crear objetos. Fundamentalmente contienen métodos ***static*** para realizar ciertas operaciones de utilidad: ordenar, buscar, introducir ciertas características en objetos de otras clases, etc.

b) Elementos del Java Collections Framework

Interfaces de la JCF: Constituyen el elemento central de la JCF. Las más comunes son:

- ***Iterable:*** Define el concepto de iterador que sirve para recorrer y eliminar elementos (lo veremos más adelante).
- ***Collection***: define métodos para tratar una colección genérica de elementos. Extiende a Iterable (hereda su funcionalidad). Las implementaciones determinan si hay un orden específico y si se permiten los duplicados. Permite tener una serie de métodos comunes a (casi) todos los tipos de colecciones (salvo Map y derivados). Sirve para manipular colecciones de la forma más general posible.
- ***Set***: colección que **no admite elementos repetidos** y **sin orden específico**.
- ***SortedSet***: *set* cuyos elementos se mantienen ordenados según el criterio establecido.
- ***List***: admite elementos repetidos y mantiene un orden inicial.
- ***Map***: conjunto de pares clave/valor, sin repetición de claves.
- ***SortedMap***: *map* cuyos elementos se mantienen ordenados según el criterio establecido.

Como hemos dicho, cada tipo de colección tiene unas características. Las más destacadas son:

a) **Una lista** es una colección de objetos donde cada uno de ellos lleva un índice asociado. Así, podríamos tener una lista con los nombres de las personas que han utilizado un servicio de acceso a internet que podría ser: usuarios --> (Juan R.R., Sara G.B., Rodolfo M.N., Pedro S.T., Claudio R.S., Juan R.R.). Donde cada contenido va asociado a un índice, usuario (0) sería Juan R.R., usuario (1) sería Sara G.B, usuario (2) sería Rodolfo M.N. y así sucesivamente. **En una lista podemos insertar y eliminar objetos de posiciones intermedias**. Ejemplos de listas son la clase ArrayList y LinkedList del API de Java.

b) **Un conjunto o set** sería una colección de objetos que **no admite duplicados**. Siguiendo el ejemplo anterior, un conjunto nos serviría para saber los usuarios distintos que han utilizado el servicio de acceso a internet, pero no tendríamos información sobre el orden y una misma persona no aparecería más de una vez ni siquiera, aunque hubiera utilizado el servicio varias veces. Ejemplo de conjunto sería la clase HashSet del API de Java.

c) **Una cola** sería una colección de objetos que se comportan como lo haría un grupo de personas en la cola de una caja de un supermercado. **Los objetos se van poniendo en cola y el primero en salir es el primero que llegó.**

Interfaces de soporte:

- ***Iterator***: sustituye a la interface ***Enumeration*** *(desde Java 5)*. Dispone de métodos para recorrer una colección y para borrar elementos.
- ***ListIterator***: deriva de ***Iterator*** y permite recorrer *lists* en ambos sentidos.
- ***Comparable***: declara el método ***compareTo ( )*** que permite ordenar las distintas colecciones según un orden natural (***String***, ***Date***, ***Integer***, ***Double***…).
- ***Comparator***: declara el método ***compare ( )*** y se utiliza en lugar de ***Comparable*** cuando se desea ordenar objetos no estándar o sustituir a dicha interface.

Clases de propósito general: Son las implementaciones de las interfaces de la JFC.

- ***HashSet***: Interface ***Set*** implementada mediante una hash table.
- ***TreeSet***: Interface ***SortedSet*** implementada mediante un árbol binario ordenado.
- ***ArrayList***: Interface ***List*** implementada mediante un array.
- ***LinkedList***: Interface ***List*** implementada mediante una lista vinculada.
- ***HashMap***: Interface ***Map*** implementada mediante una hashtable.
- ***WeakHashMap***: Interface ***Map*** implementada de modo que la memoria de los pares clave/valor pueda ser liberada cuando las claves no tengan referencia desde el exterior de la ***WeakHashMap***.
- ***TreeMap***: Interface ***SortedMap*** implementada mediante un árbol binario.

Clases Wrapper: Colecciones con características adicionales, como no poder ser modificadas o estar sincronizadas. No se accede a ellas mediante constructores, sino mediante métodos "factory" de la clase ***Collections***.

Clases de utilidad: Son mini-implementaciones que permiten obtener ***sets*** especializados, como por ejemplo ***sets*** constantes de un sólo elemento (***singleton***) o ***lists*** con ***n*** copias del mismo elemento (***nCopies***). Definen las constantes EMPTY_SET y EMPTY_LIST. Se accede a través de la clase ***Collections***.

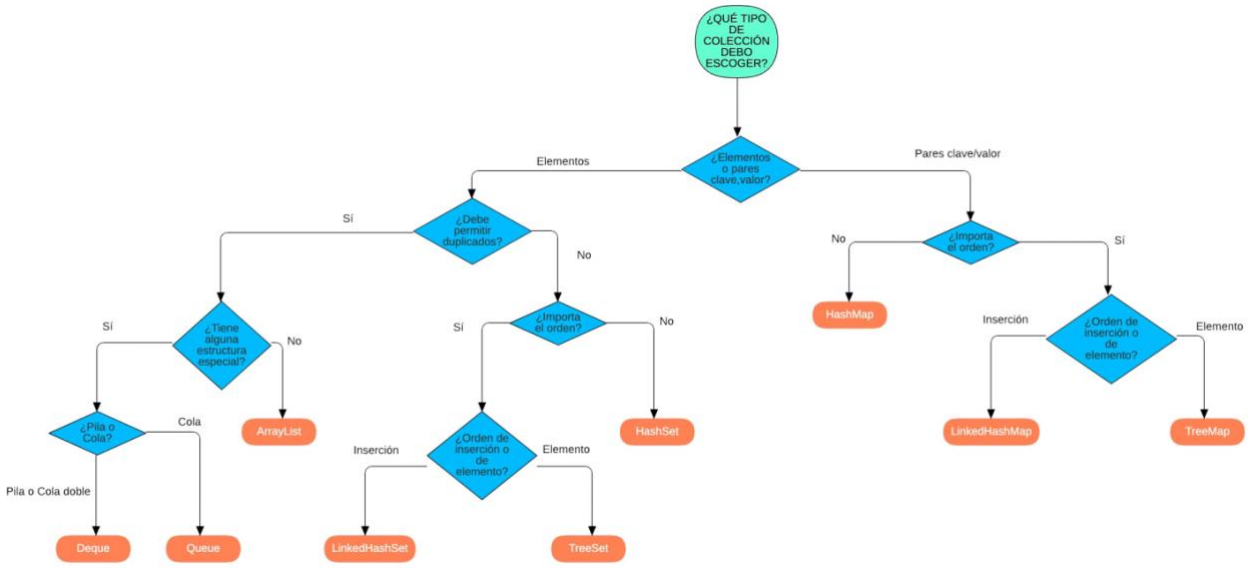
Clases históricas: Son las clases ***Vector*** y ***Hashtable*** presentes desde las primeras versiones de ***Java***. En las versiones actuales, implementan respectivamente las interfaces ***List*** y ***Map***, aunque conservan también los métodos anteriores.

Clases abstractas: Son las clases abstract de la figura anterior. Tienen total o parcialmente implementados los métodos de la interface correspondiente. Sirven para que los usuarios deriven de ellas sus propias clases con un mínimo de esfuerzo de programación.

Algoritmos: La clase ***Collections*** dispone de métodos ***static*** para ordenar, desordenar, invertir orden, realizar búsquedas, llenar, copiar, hallar el mínimo y hallar el máximo.

Clase Arrays: Es una clase de utilidad introducida en el JDK 1.2 que contiene métodos ***static*** para ordenar, llenar, realizar búsquedas y comparar los arrays clásicos del lenguaje. Permite también ver los ***arrays*** como ***lists***.

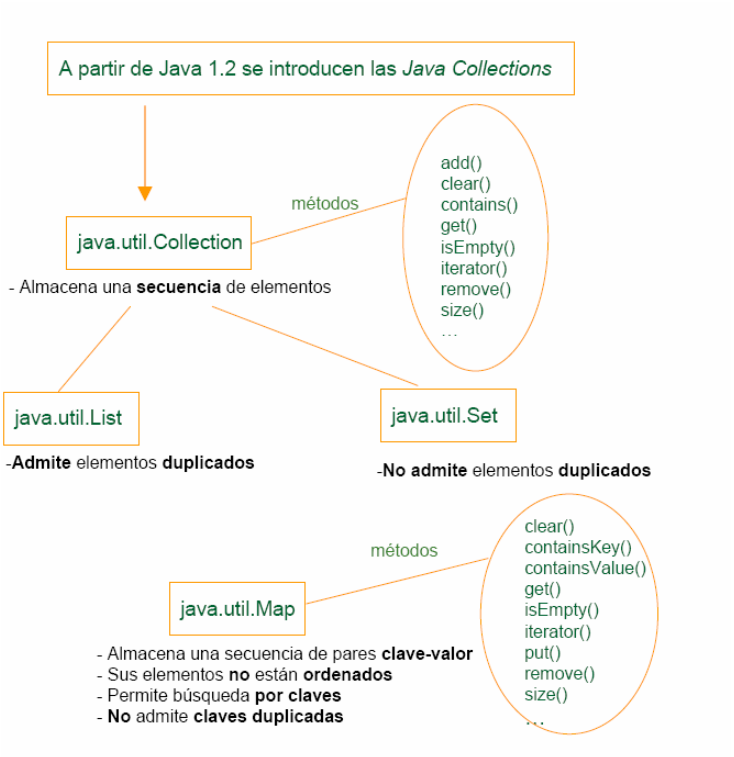
¿Cómo saber qué clase elegir? Hay varios factores a tener en cuenta, entre ellos el número de datos que tenemos que gestionar (no es lo mismo trabajar con una colección de 50 objetos que con una colección de 50.000 objetos) y el tipo de procesos que tenemos que realizar con ellos (no es lo mismo una lista en que los nuevos elementos se añaden casi siempre al final de la lista que una lista donde los nuevos elementos se añaden frecuentemente en posiciones intermedias). Cada clase resulta más eficiente que otra para realizar determinados procesos. Esto es de especial interés cuando hay que gestionar muchos datos. Si hablamos de sólo unas decenas de datos no vamos a ser capaces de apreciar diferencias de rendimientos.



NOTA: Evidentemente, no vamos a estudiar en profundidad todas las clases e interfaces, ya tendréis tiempo en vuestro brillante futuro como programadores de verlas todas con detenimiento (si os hacen falta). Veremos ejemplos de las más usadas.

La mayoría de estas clases e interfaces se encuentran en el paquete java.util del API de Java.

Un esquema concreto:



c) La interfaz Collection <E>

La interfaz más importante es Collection. Una Collection es todo aquello que se puede recorrer (o "iterar") y de lo que se puede saber el tamaño. Muchas otras clases extenderán Collection imponiendo más restricciones y dando más funcionalidades.

No puedo construir una Collection. No se puede hacer "new" de una Collection, sino que todas las clases que realmente manejan colecciones "son" Collection, y admiten sus operaciones.

La interface ***Collection*** es implementada por los ***conjuntos*** (*sets*) y las ***listas*** (*lists*). Esta interface declara una serie de métodos generales utilizables con ***Sets*** y ***Lists***.

También cuenta con métodos para hacer operaciones Bulk (a granel o en masa) como containsAll, addAll...

Método	Uso
boolean add (Object **o**)	Añade el objeto a la colección. Devuelve **true** si se pudo completar la operación. Si no cambió la colección como resultado de la operación devuelve **false**
boolean remove (Object **o**)	Elimina al objeto indicado de la colección.
int size ( )	Devuelve el número de objetos almacenados en la colección
boolean isEmpty ( )	Indica si la colección está vacía
boolean contains (Object **o**)	Devuelve **true** si la colección contiene a *o*
void clear ( )	Elimina todos los elementos de la colección
boolean addAll (Collection **otra**)	Añade todos los elementos de la colección *otra* a la colección actual
boolean removeAll (Collection **otra**)	Elimina todos los objetos de la colección actual que estén en la colección *otra*
boolean retainAll (Collection **otra**)	Elimina todos los elementos de la colección que no estén en la otra
boolean containsAll (Collection **otra**)	Indica si la colección contiene todos los elementos de otra
Object[] toArray ( )	Convierte la colección en un array de objetos.
Iterator iterator ( )	Obtiene el objeto iterador de la colección

d) La interfaz Set <E>

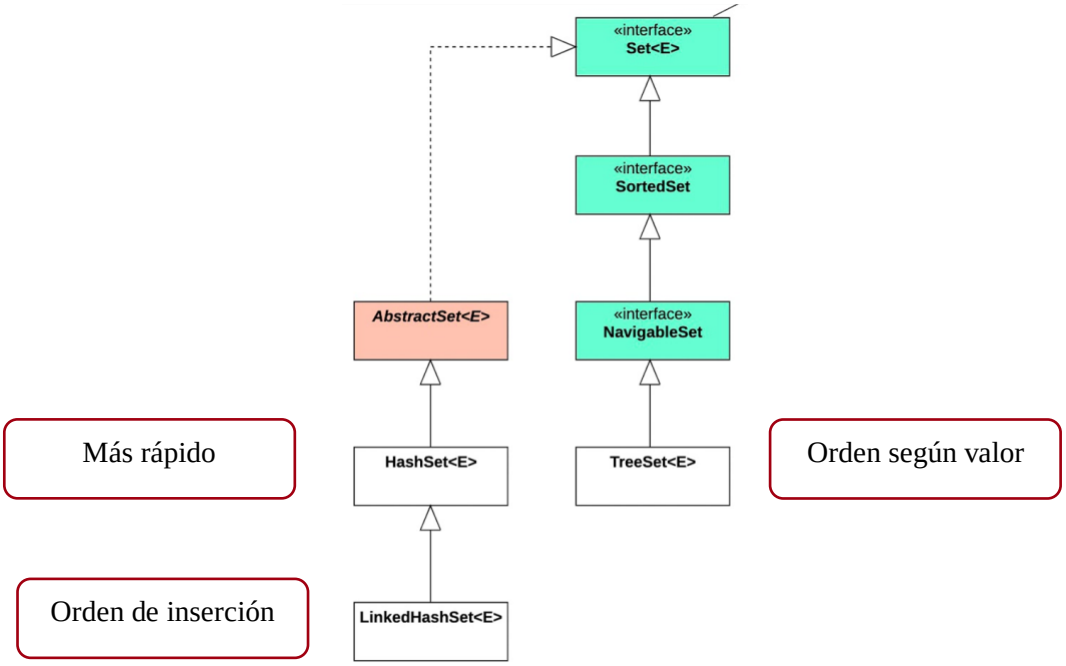
Como ya hemos dicho, se trata de una Colletion que no permite duplicados y es la abstracción del concepto matemático de "conjunto".

No hay acceso posicional, por ejemplo, decir "el tercer elemento" no tiene sentido y mejora la implementación de los métodos equals y hashCode con respecto a Colletion <E>:

> Dos instancias de set <E> son iguales si contienen los mismos elementos.

Las implementaciones de Set <E> son:

AbstractSet es una clase abstracta que implementa la parte común de las funcionalidades de una implementación de Set.



Implementaciones de Set <E>

*** HashSet <E>**

https://download.java.net/java/GA/jdk14/docs/api/java.base/java/util/HashSet.html

Características:

- No podemos predecir nada sobre el orden.
- Mejor rendimiento de todas.
- Proporciona tiempo constante (*O(1)*) en las operaciones básicas.
- Permite insertar valores nulos.
- No sincronizada.
- Se mejora el rendimiento si se establece una capacidad inicial no muy elevada.

Almacena sus valores en una tabla hash.

Una función hash tiene como entrada un conjunto de elementos, que suelen ser cadenas, y los convierte en un rango de salida finito, normalmente cadenas de longitud fija.

Una **tabla hash**, **matriz asociativa**, **hashing**, **mapa hash**, **tabla de dispersión** o **tabla fragmentada** es una estructura de datos que asocia llaves o claves con valores. La operación principal que soporta de manera eficiente es la búsqueda: permite el acceso a los elementos (teléfono y dirección, por ejemplo) almacenados a partir de una clave generada (usando el nombre o número de cuenta, por ejemplo). Funciona transformando la clave con una función hash en un hash, un número que identifica la posición (casilla o cubeta) donde la tabla hash localiza el valor deseado.

Las **tablas hash** se suelen implementar sobre vectores de una dimensión, aunque se pueden hacer implementaciones multi-dimensionales basadas en varias claves. Como en el caso de los arrays, las tablas hash proveen tiempo constante de búsqueda promedio O(1),[2] sin importar el número de elementos en la tabla. Sin

embargo, en casos particularmente malos, el tiempo de búsqueda puede llegar a O(*n*), es decir, en función del número de elementos.

Comparada con otras estructuras de arrays asociadas, las tablas hash son más útiles cuando se almacenan grandes cantidades de información.

Las tablas hash almacenan la información en posiciones pseudo-aleatorias, así que el acceso ordenado a su contenido es bastante lento. Otras estructuras como árboles binarios auto-balanceables tienen un tiempo promedio de búsqueda mayor (tiempo de búsqueda O(log *n*)), pero la información está ordenada en todo momento.

O(1) significa tiempo constante, independientemente del número de elementos
O(n) significa que el tiempo va en función del número de elementos.

*** *LinkedHashSet<E>***

https://download.java.net/java/GA/jdk14/docs/api/java.base/java/util/LinkedHashSet.html

- Almacena sus valores en una tabla hash con una lista doblemente enlazada.
- Mantiene el orden de inserción.
- Posibilidad de almacenar un valor nulo.
- No sincronizada.
- Rendimiento mejor que *TreeSet<E>* pero peor que *HashSet<E>*.

*** *TreeSet<E>***

Se dará el material para su estudio en el trabajo de uno de los compañeros.

Ejemplo: Interfaz Set

```
/*
* La variable primerSet de tipo SET se inicializa como objeto HashSet. Después
* agrega algunos elementos e imprime la variable primerSet c salida.
* Si se intentan agregar valores duplicados a set, al no ser posible los
* métodos add devuelven false y no se agrega el valor
*/

import java.util.*;
public class Tema6EjemploSet {
        public static void main(String[] args) {
                Set<String> primerSet = new HashSet<String>();
```

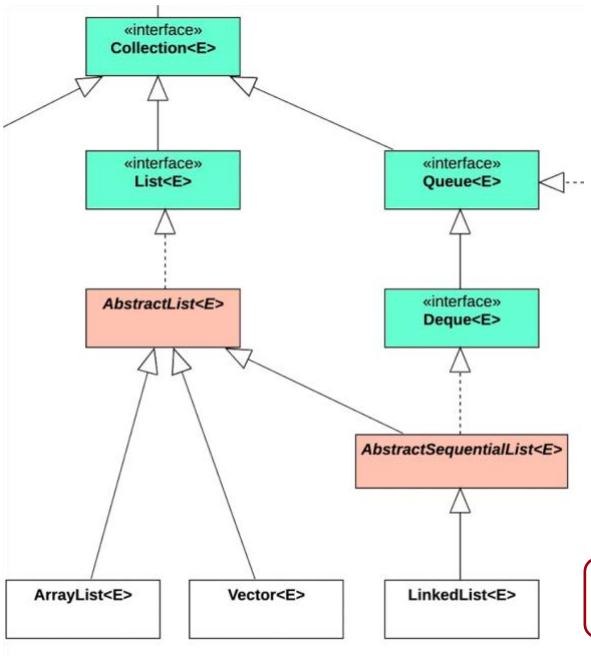
```
                primerSet.add("one");
                primerSet.add("second");
                primerSet.add("3rd");
                primerSet.add("second"); // duplicado, no se agrega

                System.out.println(primerSet.add("second"));

/* En la siguiente línea, se imprime el objeto set como salida estándar. Esto funciona porque la clase HashSet anula el método
heredado toString y crea una secuencia separada por comas con los elementos delimitados por los corchetes inicial y final*/

                System.out.println(primerSet);
        }

}

//Ojo, pues los elementos no aparecen en el mismo orden en el que se agregan
```

e) La interfaz List <E>

Como se ha dicho anteriormente, se trata de una Collection <E> que permite duplicados y añade a dicha interfaz, funcionalidades como:

- Acceso posicional
- Búsqueda
- Iteración extendida
- Operaciones sobre un rango de elementos

Implementaciones de List <E>



Más usual

Eficiente en algunas situaciones

Vector, es sincronizada, pero con métodos legacy (antiguos, por lo que es menos recomendable)

*** ArrayList <E>**

Más adecuada en la mayoría de las situaciones.

Acceso por índice en O(1)

Inserción, en media, en O(1)

Ocupa menos espacio que LinkedList.

*** LinkedList <E>**

Suele tener peor rendimiento.

Acceso por índice en O(n).

Inserción/borrado: O(1) extremos, O(n) por índice, O(1) en iteración.

Necesita más espacio (debe incluir dos referencias)

Operaciones con List <E>

* Acceso posicional, parecido al uso de un array.

* Búsqueda, para obtener el índice de un elemento con indexof, lastIndexOf.

* Iteración extendida, con listIterator:

    Se puede avanzar en cualquier posición

    Se puede añadir, modificar y eliminar

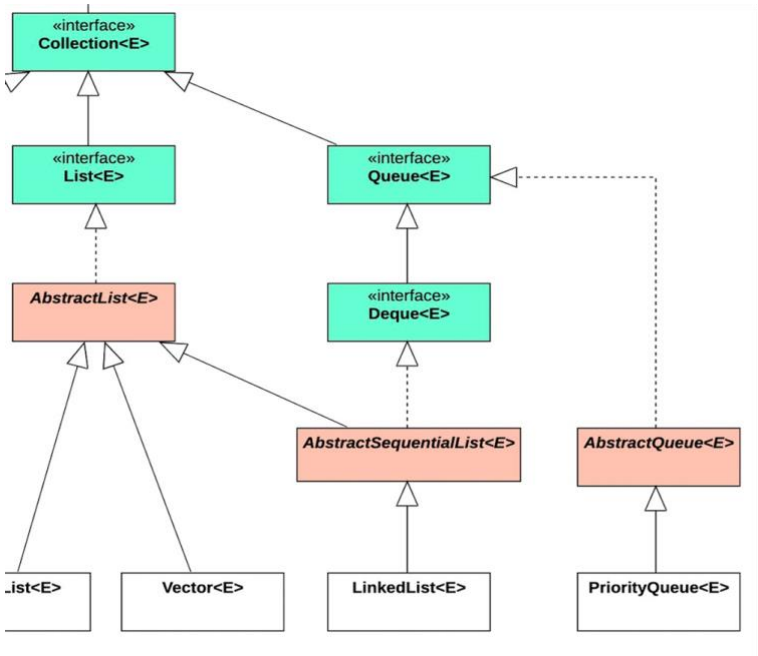
* Operaciones sobre un rango de elementos

    Se puede obtener un view de una sublista

    Las operaciones de inserción, actualización y borrado modifican la lista subyacente.

(Las colecciones de tipo View, son colecciones que no se almacenan directamente en memoria, sino a través de otra colección). No las veremos este curso.

Otras interfaces relacionadas con List <E>



Queue y Deque serán preparadas por un compañero.

Ejemplo: interfaz List

```
/*Se declara una variable de tipo Lista que se asigna a un objeto ArrayList nuevo.
Se agregan algunos elementos e imprimimos la lista como salida estándar. Como en
este caso, las listas sí admiten duplicados, los métodos add que añaden elementos
duplicados devuelven true*/


import java.util.*;
public class Tema6EjemploList {
	public static void main(String[] args) {

		List <String>lista = new ArrayList<String>();
		lista.add("one");
		lista.add("second");
		lista.add("3rd");
		lista.add("second"); // duplicado, pero sí se añade

		//Ahora los elementos sí aparecen en el orden en que
		//se agregaron
```

```
                System.out.println(lista);
        }
}
```

Como métodos para operar con listas podemos señalar: añadir un objeto en una posición determinada, añadir un objeto al final de la lista, recuperar un objeto situado en determinada posición, etc. Los objetos de un ArrayList tienen un orden, que es el orden en el que se insertaron en la lista.

Otro ejemplo con List

```
import java.util.ArrayList;

//Los import deben ir siempre al principio antes de declarar la clase y después del package

//Esta clase representa una lista de nombres manejada con la clase ArrayList de Java

public class ListaNombres {

  private String nombreDeLaLista; //Establecemos un atributo nombre de la lista

  private List<String> listadenombres; //Declaramos un ArrayList que contiene objetos String, lo de
<string> se explican más adelante


  public ListaNombres (String nombreDeLaLista) {     //Constructor: crea una lista de nombres vacía

      this.nombreDeLaLista = nombreDeLaLista;

      listadenombres = new ArrayList<String>(); //Instanciamos el objeto como ArrayList

  }

  public void addNombre (String valor_nombre) {

          listadenombres.add (valor_nombre);

  }

  public String buscarNombre (int posicion) {

      if (posicion >= 0 && posicion < listadenombres.size() ) {

          return listadenombres.get(posicion);

      }

      else {

          return "No existe nombre para la posición solicitada";

      }

  }

  public int getTamaño () {

          return listadenombres.size();

  }

  public void removeNombre (int posicion) {

      if (posicion >= 0 && posicion < listadenombres.size() ) {

          listadenombres.remove(posicion); }

      else {

          System.out.println("No existe nombre para la posición solicitada");
```

```
        }
    }
}
```

3) GENÉRICOS

a) Introducción

Las clases colección utilizan los tipos ***Object*** para admitir diferentes tipos de entradas y salidas. Por tanto, es necesario convertir el tipo de objeto explícitamente para poder recuperar un objeto.

Aunque la infraestructura de colecciones existente admite las colecciones homogéneas (es decir, colecciones de un tipo de objeto, por ejemplo, tipo Date), no había ningún mecanismo para evitar la inserción de otros tipos de objetos en la colección. Para poder recuperar un objeto, casi siempre había que convertirlo.

Este problema se resolvió con la funcionalidad de los genéricos. Dicha funcionalidad se introdujo con la plataforma Java SE 5.0. Proporciona información para el compilador sobre el tipo de colección utilizado. Así, la comprobación del tipo se resuelve de forma automática en el momento de la ejecución. Esto elimina la conversión explícita de los tipos de datos para su uso en la colección. Gracias al autoboxing de los tipos primitivos, es posible utilizar tipos genéricos para escribir código más sencillo y comprensible.

Antes de los genéricos:

ArrayList lista = new ArrayList ( );

lista.add (0, new Integer (42));

int total = ((Integer) lista.get (0)).intValue ( );

En este ejemplo, se necesita la clase envoltorio Integer para la conversión del tipo mientras se recupera el valor del número entero de lista. En el momento de la ejecución, el programa debe controlar el tipo para lista.

Al aplicar los tipos genéricos, ArrayList debe declararse como ArrayList <Integer> para informar al compilador sobre el tipo de colección que se va a usar. Al recuperar el valor, no se necesita una clase envoltorio Integer.

Se pueden parametrizar clases con uno, varios tipos, con herencia entre ellos, etc. *(Ver enlaces en la plataforma)*

Usando colecciones genéricas. COMO NOSOTROS LO HAREMOS SIEMPRE.

List<Integer> lista = new ArrayList<Integer> ( );

lista.add (0, new Integer (42));

int total = lista.get (0) .intValue ( );

Usando la función autoboxing (el propio compilador hace el casteo automáticamente, de int a Integer, por ejemplo):

```
List<Integer> lista = new ArrayList<Integer> ( );
lista.add (0, 42);
int total = lista.get (0);
```

b) Algunas diferencias entre clases genéricas y no genéricas

- En las no genéricas, se declaraban las clases como "***public class ArrayList extends AbstractList implements List***" por ejemplo, y usando genéricas se declara como "***public class ArrayList <E> extends AbstractList <E> implements List <E>***", donde el parámetro de tipo E representa el tipo de los elementos incluidos en la colección.
- En los constructores no hay diferencia de sintaxis.
- En la declaración de los métodos, antes era "***public void add (Object o)***" y ahora "***public void add (E o)***".
- En las declaraciones de variables: antes era "**ArrayList lista1**"; Con genéricas es: "***ArrayList <String> lista1***".
- En la instanciación: Antes era "***lista1= new ArrayList (10);***" y usando genéricas "***lista1= new ArrayList <String> (10);***"

c) Garantía de seguridad

Imaginemos que tenemos las clases necesarias para escribir el siguiente código, donde Cuenta es la clase madre y CCorriente y CAhorro extienden de ella:

```
import java.util.*;
public class PruebaBanco {
        public static void main (String [ ] args ) {
                List <Cuenta> micuenta;
                List <CCorriente> miCCorriente = new ArrayList <CCorriente> ( );
                List <CAhorro> miCAhorro = new ArrayList <CAhorro> ( );
//Si lo siguiente fuera posible
                miCuenta= miCCorriente;
                miCuenta.add(new CCorriente ("Ángel"));
//Lo siguiente también debería ser posible
                miCuenta= miCAhorro;
                miCuenta.add (new CCorriente ("Ángel"));
        }
```

}

OJO: miCuenta=miCCorriente es ilegal. Por tanto, a pesar de que CCorriente es una Cuenta (extiende de ella), ArrayList <CCorriente> no equivale a ArrayList <Cuenta>.

Para que la garantía de seguridad de tipo siempre sea válida, debe ser imposible asignar una colección de un tipo a una colección de un tipo distinto, incluso si el segundo tipo es una subclase del primero.

Esto va en contra del polimorfismo tradicional y, a primera vista, parece restar flexibilidad a las colecciones genéricas. Para solucionar esto, existen los tipos "covariantes" o "comodines" que nosotros NO estudiaremos, pero por si os lo encontráis algún día.

4. LA INTERFAZ ITERATOR

Una capacidad de un objeto Collection es la de poder ser recorrido. Como a este nivel no está definido un orden, la única manera es proveyendo un iterador, mediante el método **iterator ( )**. Un iterador es un objeto "paseador" que nos permite ir obteniendo todos los objetos al ir invocando progresivamente su método **next ( )**. También, si la colección es modificable, podemos remover un objeto durante el recorrido mediante el método **remove ( )** del iterador.

Supongamos que vamos recorriendo una lista de 20 objetos y que durante el recorrido borramos 5 de ellos. Probablemente nos saltará un error porque Java no sabrá qué hacer ante esta modificación concurrente. Sería como seguir un camino marcado sobre unas baldosas y que durante el camino nos movieran las baldosas de sitio: no podríamos seguir el camino previsto. Este tipo de problemas se suelen presentar con un mensaje de error del tipo java.util.ConcurrentModificationException.

El uso de operaciones sobre colecciones usando un for tradicional también puede dar lugar a resultados no deseados. En general, operar sobre una colección al mismo tiempo que la recorremos puede ser problemático. Este problema queda salvado mediante un recurso del API de Java: los objetos tipo Iterator. **Un objeto de tipo Iterator funciona a modo de copia para recorrer una colección,** es decir, el recorrido no se basa en la colección "real" sino en una copia. De este modo, al mismo tiempo que hacemos el recorrido (sustentado en la copia) podemos manipular la colección real, añadiendo, eliminando o modificando elementos de la misma. Para poder usar objetos de tipo Iterator hemos de declarar en cabecera import java.util.Iterator; La sintaxis es la siguiente:

```
Iterator <TipoARecorrer> it = nombreDeLaColección.iterator ();
```

Esta sintaxis resulta un tanto confusa. Por ello vamos a tratar de analizarla con detenimiento. En primer lugar, declaramos un objeto de tipo Iterator. La sentencia para ello es Iterator <TipoARecorrer> it; donde it es el nombre del objeto que estamos declarando. El siguiente paso es inicializar la variable para que efectivamente pase a contener un objeto. Pero Iterator carece de constructor, así que no podemos usar una sentencia de tipo *n*ew Iterator<TipoARecorrer> ( ); porque no resulta válida. Iterator es una construcción un tanto especial. De momento nos quedamos con la idea de que es algo parecido a una clase. Dado que no disponemos de constructor, usamos un método del que disponen todas las colecciones denominado iterator () y que cuando es invocado devuelve un objeto de tipo Iterator con una copia de la colección. Por tanto, hemos de distinguir:

a) Iterator con mayúsculas, que define un tipo.

b) El método iterator (con minúsculas) que está disponible para todas las colecciones y que cuando es invocado devuelve un objeto de tipo Iterator con una copia de la colección.

¿Cómo sabemos cuándo nos referimos al tipo Iterator y cuándo al método iterator ( )?

Prestando atención a las mayúsculas/minúsculas y al contexto de uso. Si el nombre de una colección va seguido de .iterator () sabremos que estamos utilizando un método para obtener el iterador de la colección. Por las minúsculas, por los paréntesis, y por ir a continuación del nombre de la colección. Iterator es un tipo genérico o parametrizado, porque requiere que definamos un tipo complementario cuando declaramos objetos de tipo Iterator. Los objetos de tipo Iterator tienen como métodos principales:

Método	Uso
Object next ( )	Obtiene el siguiente objeto de la colección. Si se ha llegado al final de la colección y se intenta seguir, da lugar a una excepción de tipo: **NoSuchElementException** (que deriva a su vez de **RunTimeException**)
boolean hasNext ( )	Indica si hay un elemento siguiente (y así evita la excepción).
void remove ( )	Elimina el último elemento devuelto por **next**

Uno de los métodos disponibles para la clase String es contains. La sintaxis para su uso es "Cadena".contains ("textoabuscar"), que devuelve true si Cadena contiene textoabuscar. Consideremos el siguiente ejemplo:

```
if (coleccion.next().contains(cadena) ) {
    System.out.println ("Cadena encontrada en " + coleccion.next() );
}
```

Este código presenta un problema. ¿Cuál es? Que con la primera referencia a coleccion.next() se evalúa si un ítem de la colección contiene la variable *cadena*, y cuando le decimos que imprima que se ha encontrado la cadena en coleccion.next() no nos devuelve el ítem deseado. ¿Por qué? Porque cada vez que aparezca el método next() se devuelve el siguiente objeto dentro de la colección.

Por eso, en el siguiente ejemplo, donde queremos evaluar y mostrar el String que se analiza, utilizamos otro String temporal para almacenar el que nos devuelve cada llamada al método next ().

Ejemplo uso Iterator:

```
import java.util.*

public class TestPrueba {

   public static void main(String[] args) {

        List <String> listaDeNombres = new ArrayList <String> ();

        listaDeNombres.add("Juan Pérez Sánchez");

        listaDeNombres.add("José Alberto García Montes");

        String cadenaBuscar = "Alberto";

        System.out.println ("La cadena que buscamos es " + cadenaBuscar);


        Iterator<String> it = listaDeNombres.iterator();//Creamos el objeto it de tipo
        Iterator con String

        String tmpAnalizando;

        while (it.hasNext()) {//Utilizamos el método hasNext de los objetos tipo
              Iterator

              tmpAnalizando = it.next();//Utilizamos el método next de los      objetos
              tipo Iterator

              System.out.println ("Analizando elemento... " + tmpAnalizando);


              if (tmpAnalizando.contains(cadenaBuscar)) {

                    System.out.println ("Cadena encontrada!!!");

              } else { }//else vacío. No hay acciones a ejecutar.

        }

   }

}
```

Además de las ventajas propias de trabajar con una copia en vez de con la colección original, otro aspecto de interés de la clase Iterator y el método iterator radica en que, no todas las colecciones de objetos en Java tienen un índice entero asociado a cada objeto, y por tanto no se pueden recorrer basándonos en un índice. En cambio, siempre se podrán recorrer usando un iterador.

5. LA INTERFAZ MAP

https://download.java.net/java/GA/jdk14/docs/api/java.base/java/util/Map.html

Un Map es una estructura de datos agrupados en parejas clave/valor. Pueden ser considerados como una tabla de dos columnas.

La clave debe ser única y se utiliza para acceder al valor.

Aunque la interface Map no deriva de Collection, es posible ver los Maps como colecciones de claves, de valores o de parejas clave/valor.

Por definición, los objetos map no admiten claves duplicadas y una clave sólo puede asignarse a un valor.

Puede haber una clave nula y múltiples valores nulos.

En otros lenguajes de programación, se les conocen como diccionarios.

No puede almacenar tipos primitivos, hay que usar los tipos wrapper en su lugar (int ---> Integer)

Operaciones con Map<K,V>

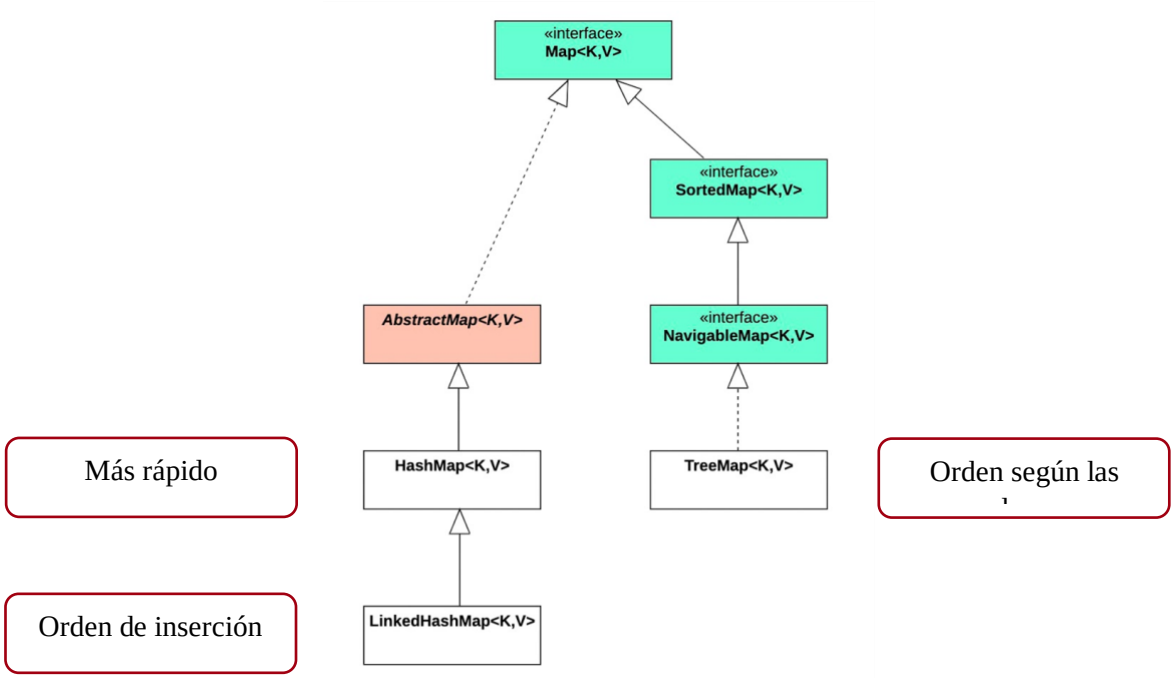
Método	Uso
Object get (Object **clave**)	Devuelve el objeto que posee la clave indicada
Object put (Object **clave,** Object **valor**)	Coloca el par clave-valor en el mapa. Si la clave ya existiera, sobrescribe el anterior valor y devuelve el objeto antiguo. Si esa clave no aparecía en la lista, devuelve **null.**
Object remove (Object **clave**)	Elimina de la lista el valor asociado a esa clave.
boolean containsKey (Object **clave**)	Indica si el mapa posee la clave señalada.
boolean containsValue (Object **valor**)	Indica si el mapa posee el valor señalado.
void putAll (Map **mapa**)	Añade todo el mapa al mapa actual.
Set keySet ( )	Obtiene un objeto *Set* creado a partir de las claves del mapa.
Collection values ( )	Obtiene la colección de valores del mapa, es decir, devuelve una variable Colletion con todos los valores contenidos en el mapa.
Set entrySet ( )	Devuelve una lista formada por objetos **Map.Entry,** es decir, devuelve una variable Set que contiene todos los pares formados por una clave y un valor.

NOTA: La interfaz SortedMap amplía la interfaz Map. Algunas de las clases que implementan la interfaz Map son: HashMap, TreeMap, IdentityHashMap y WeakHashMap. El "programador" tendrá que acudir a ellas cuando sea necesario.

Para recorrer un mapa

- Opción 1: forEach
  - Obtener un Set con las claves.
  - Para cada clave, obtener los valores.
- Opción 2: Usando Lambdas (se explicará más adelante)
  - Método forEach.
  - Expresión lambda (Biconsumer)

Implementaciones Map<K,V>



*** HashMap <K,V>**

Es la más utilizada, por tener mejor rendimiento.

No podemso suponer nada sobre el orden de los pares.

El tiempo de ejecución de inserción y consulta es constante O(1).

No es sincronizada.

**** LinkedHashMap<K,V>***

Rendimiento un poco peor que HashMap.

Ordena los pares clave-valor según su inserción

No es sincronizada.

*** *TreeMap<K,V>*** La explicará un compañero.

Peor rendimiento de las 3 implementaciones.

Mantiene las claves en orden (natural)

No puede tener ninguna clave nula, aunque sí puede almacenar valores nulos.

Map.Entry<K,V>

Esta clase permite consultar un par clave-valor de un Map.

No se pueden utilizar para insertar valores.

Se puede obtener un Set <Map.Entry <K, V>> a través del método Map.entrySet ( ).

Ejemplo con Map

En este ejemplo, se declara una variable mapa de tipo Map y y la asigna a un objeto HashMap nuevo. Se agregan algunos elementos al mapa mediante la operación *put*. Para demostrar que los mapas no admiten claves duplicadas, el programa intenta agregar un valor nuevo con una clave ya existente.

El resultado es que el valor nuevo sustituye al valor agregado anteriormente para la clave. A continuación, el programa utiliza las operaciones de visualización de colecciones keySet, values y entrySet para recuperar el contenido del mapa.

```
import java.util.*;
public class Tema6EjemploMap {
	public static void main(String args[]) {
		Map <String, String> mapa = new HashMap <String, String>();
		mapa.put("one","1st");
		mapa.put("second", "2nd");
		mapa.put("third","3rd");
		//Sobrescribe la asignación anterior porque no admite claves repetidas
		mapa.put("third","III");
		//Devuelve el conjunto de las claves y se guarda en set1
		Set <String>set1 = mapa.keySet();
		//Devuelve la vista Colletion de los valores y se guarda en collection
		Collection collection = mapa.values();
		//Devuelve el conjunto de las asignaciones (clave,valor)
		Set<Entry<String, String>>set2 = mapa.entrySet();
		System.out.println(set1 + "\n" + collection + "\n" + set2);
```

```
        }
}
//No aparecen ordenados al impirmir
```

Salida:

```
[third, one, second]
[III, 1st, 2nd]
[third=III, one=1st, second=2nd]
```

NOTA: Existen otros tipos de estructuras que no vamos a ver como son los "árboles" en treeSet, lo harán los compañeros que preparen ese tema en el trabajo de esta unidad.

Aunque es antiguo, os dejo un enlace con otro ejemplo completo para el uso de Map bastante claro.

http://jarroba.com/map-en-java-con-ejemplos/

6. LOS MÉTODOS EQUALS ( ) Y HASHCODE ()

Los genéricos fueron una de las características principales en el lanzamiento de Java 5. Con la salida de los genéricos, cambió la forma en cómo podemos manejar colecciones en Java. Una de las características que tienen las colecciones, es el poder ordenar y buscar entre los elementos que forman parte del contenido. Podemos establecer el criterio de búsqueda u ordenamiento para nuestras colecciones, pero para esto debemos sobrescribir los métodos equals y hashCode.

Sobrescribiendo el método equals ( )

Indagando en la javadoc acerca del método equals (Object o), podemos encontrar lo siguiente:

Método	Valor de retorno	Descripción
equals (Object o)	boolean	Indica si un objeto es igual a este.

Este método lo utilizamos para saber si un objeto es igual que otro. Este método utiliza el operador == para comparar a dos objetos y decidir si son iguales, por ejemplo, podemos tener la siguiente clase:

```
public class Demo {
        private int billete;
        public Demo (int billete){
                this.billete = billete;
        }
```

```
	public int getBillete(){
		return this.billete;
	}
}
public class Prueba {
	public static void main(String[] args) {
	Demo demoA = new Demo(20);
	Demo demoB = new Demo(20);
	System.out.println(demoA.equals(demoB));
	}
}
```

La ejecución del anterior método main, da como resultado en consola false. Esto es porque el método equals sin sobrescribir usa el operador = = para comparar a dos objetos. Pero supongamos que queremos diferenciar a los objetos mediante su atributo **billete**, ya que es el identificador de un Pasajero para una aplicación. Para esto debemos reescribir el método equals, diciéndole qué propiedad del objeto debe ser comparada para determinar si un objeto es igual a otro. El método sobrescrito se vería de esta forma:

(No es necesario trabajar con Object, se hace con el objeto que sea, en este caso, Demo y no hace falta casteo, solo se hace para que se vea con la definición de equals del API, académicamente).

```
@Override
public boolean equals(Object o){
	if(( o instanceof Demo) && (((Demo)o).getBillete() == this.billete))
	{
		return true;
	} else {
		return false;
	}
}
```

La tercera línea es la que tienen la magia. En ella hacemos un par de validaciones, la primera tiene que ver con estar seguros de que el objeto **o** es una instancia de la clase Demo, y la segunda es verificar si la propiedad **billete** del objeto **o** es igual a la propiedad **billete** del objeto que invocó el método equals. Si los valores de **billete** son los mismos, el método equals que sobrescribimos devolverá true. Con esto, nosotros tomamos la decisión sobre qué criterio se debe tomar para comparar a nuestros objetos.

```
public static void main(String[] args) {
	Demo demoA = new Demo(20);
```

```
    Demo demoB = new Demo(20);
    System.out.println(demoA.equals(demoB));
}
```

Por lo anterior, esta ejecución del método main, nos dará como resultado un true en la salida por consola.

Hay tipos de colecciones, como los Sets, que no nos permiten agregar objetos duplicados a la colección. La forma en cómo los Sets van a decidir si un objeto está duplicado o no, la especificamos nosotros cuando sobrescribimos el método equals. Lo mismo ocurre, cuando en una colección de tipo Hash queremos buscar un elemento en concreto, la colección sabe qué objeto devolver, dado el criterio que nosotros definimos, al sobrescribir el método equals y hashCode. En esto radica la importancia de sobrescribir estos métodos, siempre y cuando este dentro de nuestras intenciones contar con este tipo de prestaciones.

Reglas que sigue el método equals ( )

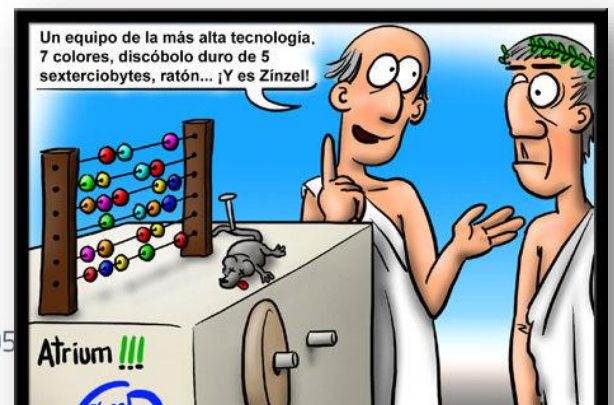
- **Reflexivo**: Para cualquier referencia al valor **x**, x.equals(x) debe devolver true.
- **Simetrico**: Para cualquier referencia a los valores **x** y **z**, x.equals(z) debe devolver true si y solo si z.equals(x) es true.
- **Transitivo**. Para cualquier referencia a los valores **w**, **x** y **z**, si w.equals(x) regresa true y x.equals(z) regresa true, entonces w.equals(z) debe devolver true.
- **Consistente**: Para cualquier referencia a los valores **x** y **z**, múltiples invocaciones a x.equals(z) consistentemente devolverán true o false, si es que los valores utilizados para la comparación de los objetos no ha sido modificada.
- Para cualquier referencia no nula al valor **x**, x.equals(null), debe devolver false.

Sobrescribiendo el método hashCode ( )

¿Qué implica el hashcode?

Algunas colecciones, usan el valor hashcode para ordenar y localizar a los objetos que están contenidos dentro de ellas. El hashcode es un número entero, sin signo, que sirve en colecciones de tipo Hash para un mejor funcionamiento en cuanto a rendimiento. Este método debe ser sobrescrito en todas las clases que sobrescriban el método equals, si no se quiere tener un comportamiento extraño al utilizar las colecciones de tipo Hash y otras clases. **Si dos objetos son iguales según el método equals sobrescrito, estos deberían devolver el mismo hashcode**. Véase el siguiente ejemplo:

```
public class Demo {
    private int billete;
    public Demo(int billete){
      this.billete = billete;
    }
```



```
    public int getBillete(){
      return this.billete;
    }
    @Override
    public boolean equals(Object o){
        if((o instanceof Demo) && (((Demo)o).getBillete()== this.billete))
        {
            return true;
        } else{
            return false;
        }
    }
}
```

Si escribimos una clase de prueba, en el main podemos escribir:

```
public static void main(String[] args) {

    Demo demoA = new Demo(20);
    Demo demoB = new Demo(20);

    System.out.println(demoA.equals(demoB));
    System.out.println(demoA.hashCode());
    System.out.println(demoB.hashCode());
}
```

Esto nos da como resultado:

true

1414159026

1569228633

Devuelve true porque estos objetos son iguales, debido a que se sobrescribió el método equals. Si son considerados iguales por equals, esto se reflejará al utilizarse en las colecciones de tipo Hash. Lo que podemos ver en la salida de este código es que a pesar de que ambos objetos son iguales, el hashcode no es igual. **Esto es porque no hemos sobrescrito el método hashCode.**

El uso principal del hashcode se da, como mencionamos arriba, cuando se manejan colecciones de tipo Hash. La forma en cómo operan las colecciones de este tipo es a grandes rasgos la siguiente:

Las colecciones de tipo Hash almacenan los objetos en lugares llamados "celdas", de acuerdo al número obtenido por el método hashCode. Si el método hashCode devuelve un 150, el objeto será guardado en la celda número 150. Puede llegar a pasar que haya más de un objeto de diferente tipo en la misma celda. Esto no ocasiona ningún problema en el momento de recuperar el objeto de la celda ya que, al buscarlo, este tipo de colecciones necesita como parámetro un objeto con el mismo valor hashcode, el cual utilizará para buscar el número de celda que contiene el objeto en cuestión. Si hay más de un objeto, el siguiente criterio para determinar cuál es el objeto buscado, es la utilización del método equals. Así es como este tipo de colecciones para buscar un objeto, ya sea para devolverlo y para ordenarlo. Todo esto falla si no sobrescribimos el método hashCode.

Sobrescribiendo el método hashCode ( )

A continuación, veremos cómo podremos sobrescribirlo. En internet se pueden encontrar muchas implementaciones de este método, por ejemplo:

```
@Override
public int hashCode() {
        int hash = 7;
        hash = 97 * hash + this.billete;//El 97 no es por nada en especial, hay gente que usa el 33 o 31.
        return hash;
}
```

La salida que obtenemos, una vez que se rescribe este método en el ejemplo de arriba es:

true

699

699

Lo que nos dice que estos métodos son iguales según equals, y que además tienen el mismo número hashcode.

Este método hashCode devolverá consistentemente el mismo valor, siempre y cuando el campo **billete** no cambie. Cada desarrollador puede implementar de diferente manera igualmente validas, correctas y eficientes este método. Lo que debemos de tener en mente al sobrescribir este método es que, si para sobrescribir el método utilizamos variables de instancia (en nuestro ejemplo **billete**), también debemos utilizar variables de instancia para generar un hashcode correcto. En el caso de las constantes que se utilizan en el ejemplo de arriba, se recomienda utilizar números primos, para una mejor distribución del hashcode generado.

Reglas que sigue el método hashCode ( )

- Si el método hashCode es invocado en múltiples ocasiones durante la ejecución de una aplicación, debe devolver consistentemente el mismo valor entero, esto si la información utilizada para calcular el hashcode no ha cambiado entre invocación e invocación del método hashCode.
- Si dos objetos son iguales según el método equals, entonces la llamada al método hashCode debe devolver el mismo hashcode.
- No es requerido que, si dos objetos no son iguales según el método equals, tengan diferentes valores hashcode.

Ojo 

- Si dos objetos son iguales según el método equals, el método hashCode debe devolver el mismo entero para ambos métodos. Sin embargo, si el método equals dice que dos métodos no son iguales, el método hashCode puede o no devolver el mismo entero.
- Devolver un valor fijo en un método hashCode es una mala idea, ya que tendremos múltiples objetos con el mismo valor hashcode, lo cual no ayuda en nada a la hora de trabajar con colecciones de tipo Hash.

Hashcode, otra forma de explicarlo

Sobrescritura de hashCode()

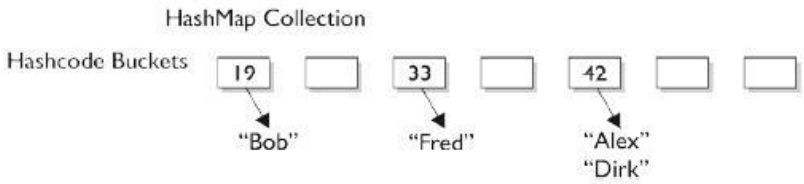
Los hashCode se usan normalmente para incrementar el rendimiento de grandes colecciones de datos. El valor hashCode de un objeto, se usa por algunas clases de colecciones. Aunque puedas pensar que esto es como el ID del objeto, no es necesariamente, único. Las colecciones tales como HashMap y HashSet usan el valor hashCode del objeto para determinar cómo se debe almacenar el objeto en la coleccion, y el hashCode se usa otra vez para ayudar a localizar el objeto en la colección. Es perfectamente “legal” tener un método ineficiente de hashCode en tu clase, mientras no violes el contrato especificado en la clase Object del API de Java.

Entender el hashCode()

Para entender qué es apropiado y correcto, tenemos que echar un vistazo a como usan las colecciones los hashCode.

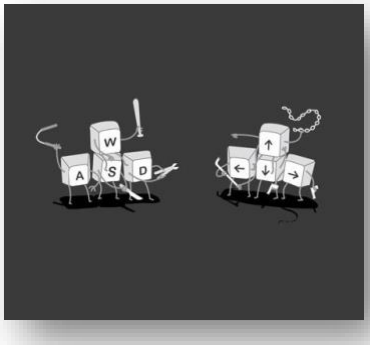
Imagina una serie de cubos alineados. Alguien coge un trozo de papel con un nombre. Coge el nombre y calcula un número entero usando, A es 1, B es 2…, y añade el valor numérico a todas las letras del nombre. El mismo nombre dado siempre dará el mismo código.

Key	Hashcode Algorithm	Hashcode
Alex	A(1) + L(12) + E(5) + X(24)	= 42
Bob	B(2) + O(15) + B(2)	= 19
Dirk	D(4) +I(9) + R(18) + K(11)	= 42
Fred	F(6) + R(18) + E(5) + (D)	= 33



No introducimos nada al azar, simplemente tenemos un algoritmo que se ejecutará siempre de la misma manera dada una entrada específica, así que la salida será la misma para dos entradas iguales. ¿Todo bien hasta aquí? Ahora la manera de usar ese código (al que llamaremos hashCode desde ahora) es determinar en qué cubo se va a poner el trozo de papel (imagina que cada cubo representa un código distinto). Ahora imagina que alguien viene, te muestra un nombre y dice "Por favor, saque un trozo de papel que tenga este nombre". Así que miras el nombre que te enseñan, y ejecutas el algoritmo. El hashCode te dice en qué cubo debería estar ese nombre.

Seguro que te habrás dado cuenta que puede haber un fallo en este sistema. Dos nombres diferentes pueden tener el mismo valor, por ejemplo, Amy y May. Son nombres distintos con el mismo hashCode. Esto es aceptable, pero significa que cuando alguien te pida la pieza de papel de Amy, aún tienes que buscar los nombres que haya en el cubo hasta que encuentres el nombre. El hashCode te dice en que cubo está el papel, pero no te dice como localizar el nombre en el cubo.



Ojo

En la vida real, no es normal que en el hashing tengas más de una entrada en el cubo. La obtención del objeto mediante hashing tiene 2 pasos:

1. Encontrar el cubo correcto (usando **hashCode()**)
2. Buscar el elemento correcto (usando **equals()**)

Así que, para más eficiencia, tu meta es tener los papeles distribuidos de manera igualitaria a través de todos los cubos. Lo ideal es que tuvieras un nombre por cubo, así que cuando alguien te pregunte por el papel, solo tuvieras que realizar el cálculo y coger el papel del cubo correcto (sin tener que estar mirando todos los papeles del cubo).

El menos eficiente (pero funcional) generador de hashcode devolverá el mismo hashcode (digamos 42) sin importar el nombre, así que todos los papeles estarán en el mismo cubo mientras los otros cubos están vacíos. Una vez en el cubo adecuado, hay que rebuscar entre todos los papeles hasta dar con el papel adecuado. Y así es como funciona, puedes pensar en no usar el hashCode, pero entonces el cubo seria aún más grande (en realidad, un cubo).

Esta distribución en cubos es una manera similar a la que el hashCode se usa en las colecciones. Cuando pones un objeto en una colección que usa hashCodes, la colección usa el hashCode del objeto para decidir en qué cubo debería ir. Luego, cuando quieras coger el objeto (o, el objeto asociado al hashCode), tienes que darle a la colección una referencia del objeto que tiene en la colección. Hasta ahora, el objeto (almacenado en la colección, como el papel en el cubo) que estás intentando buscar se hará por su hashCode.

Pero... imagina que pasaría si, volviendo a nuestro ejemplo de los nombres en el cubo, te muestran un nombre y calculas el código basándote solo en la mitad de las letras en lugar de en todas ellas. ¡Nunca encontrarías el nombre en el cubo!

¿Ahora puedes ver por qué si dos objetos son considerados iguales, sus hashCodes deben ser iguales? De otra manera, no podrías encontrar el objeto ya que el método hashCode por defecto de la clase Object siempre va a otorgar un único hashCode a cada objeto, aunque se sobrescriba equals() de tal manera que dos o más objetos sean considerados iguales. No importa como de iguales sean esos objetos si sus hashCodes no dice que lo son. Así que una vez más:

Si dos objetos son iguales, sus hashcodes tienen que serlo también

Implementado el hashCode()

Pensemos lo que hace el método equals (). Comparamos atributos. La comparación casi siempre involucra a variables de instancia. La implementación de tu hashCode () debería usar las mismas variables de instancia que se usan en equals (). Por ejemplo:

```
1  public class HasHash {
2      public int x;
3      HasHash (int xVal) {
4          x = xVal;
5      }
6      public boolean equals(Object o) {
7          HasHash h = (HasHash) o; // No lo intentes sin usar instanceof
8          if (h.x == this.x) {
9              return true;
10         } else {
```

```
11          return false;
12      }
13  }
14  public int hashCode() {
        return (x * 17);
    }
}
```

Este método equals() dice que dos objetos son equivalentes si ellos tienen el mismo valor x, así que los objetos con el mismo valor x tendrán el mismo hashcode.

NOTA: Atención, un hashCode() que devuelve el mismo valor para todas las instancias si son iguales o no, es legal, incluso apropiado. Por ejemplo:

```
public int hashCode() {
        return 1482;
}
```

Esto no violaría lo que el API de Java dice. Dos objetos con un valor x de 8 tendrán el mismo hashCode. Pero nuevamente, dos objetos serán distintos si llevan el valor de 12 y -920 (por decir algo). Este método es realmente ineficiente, recuerda, que esto hace que un objeto entre en un cubo u otro, pero incluso así, los objetos aún pueden ser encontrados mientras la colección pueda buscar dentro de ese cubo usando equals (). Es un intento desesperado por encontrar y localizar el objeto. En otras palabras, el hashCode no ayuda en absoluto a acelerar la búsqueda, aunque el propósito del hashCode sea este. Este método de meterlo todo en el mismo cubo es considerado apropiado e incluso correcto porque no viola el contrato del API, pero nuevamente, correcto no significa necesariamente que sea bueno.

Normalmente, vas a ver los métodos de hashCode en combinación con sus variables de instancia, haciendo girar sus bits o quizás multiplicándolos por un número primo. En cualquier caso, mientras la meta sea conseguir una amplia cantidad de cubos, el API de Java (no importa si el objeto se encuentra o no) exige solo que la igualdad del objeto tenga el mismo hashCode.

Ahora que sabemos que dos objetos iguales tienen que tener el mismo hashCode, ¿es lo contrario cierto? Dos objetos con idéntico hashCode, ¿tienen que ser considerados igual? Piensa en ello, tienes un motón de objetos que entran en el mismo cubo porque sus hashCode son idénticos, pero a menos que pasen el test de equals (), no se mostrarán en la búsqueda de la colección. Esto es exactamente lo que deberías conseguir con nuestro ineficiente "todo el mundo consigue el mismo hashcode". Es legal y correcto, pero muy lento.

Así que para que un objeto sea encontrado, la búsqueda del objeto y el objeto en la coleccion deben tener idénticos hashCode y devolver true a la igualdad (equal()).

El contrato de hashCode ()

Ahora directo desde los fabulosos docs de API de Java para la clase Object, presentamos (redoble de tambor) el contrato de hashCode():

- Si es invocado en el mismo objeto más de una vez durante una ejecución de una aplicación de Java, el método hashCode() debe devolver siempre el mismo entero, sin proveer información que modifique el resultado de equals(). Este entero no necesita permanecer constante en la ejecución de una aplicación a otra ejecución de la misma aplicación.
- Si dos objetos son iguales de acuerdo al método equals(), entonces el método hashCode() en cada uno de los dos objetos debe producir el mismo resultado de número entero.
- Si no se requiere que dos objetos sean no equals () (de acuerdo con equals de java.lang.Object), entonces la llamada a hashCode() en cada uno de los objetos debe producir un resultado distinto. Sin embargo el programador debería hacer que se produjeran distintos resultados para un objeto distinto, de manera que mejorase el rendimiento de la colección hash.

Y esto quiere decir...

Condición	Requisito	No requerido pero permitido
x.equals(y) == true	x.hasCode()== y.hashCode()	
x.hashCode() == y.hashCode()		x.equals(y) == true
x.equals(y) == false		Sin requerimientos en hashCode.
x.hashCode() != y.hashCode()	x.equals(y) == false	

7) ORDENAR COLECCIONES

Lo veremos con un ejemplo en clase y el siguiente enlace. Dejo también algunos tutoriales en la plataforma para diferentes tipos de colecciones.

http://jarroba.com/ordenar-un-arraylist-en-java/

8) COLECCIONES PARA SITUACIONES ESPECIALES

Colecciones no modificables

- Son colecciones que, una vez creadas, no se pueden modificar.
- Si se trata de modificar, se lanza *UnsupportedOperationException*
- Se pueden usar, por ejemplo, como resultado de una operación.
- Hasta Java 8:

  Collections.unmodifiableXXX(coleccion): versión no modificable de la colección.

  Collections.emptyXXX. Colección vacía no modificable.

- Desde Java 9:

  Métodos factoría *.of(...)*

  Diferentes versiones: desde 0 elementos hasta un número variable.

  Disponibles en los diferentes interfaces: *List, Set, Map, ...*

Colecciones sincronizadas

- Aptas para el uso de diferentes hilos de ejecución.
- Varios procesos/hilos que compiten por el uso de la colección
- Versión sincronizada de una colección no sincronizada
- Métodos *Collections.synchronizedXXX*

Otras implementaciones menos usadas

- *Set<E>*
  - *EnumSet*

Set pensado para contener valores de enumeraciones.

  - *CopyOnWriteArraySet*
    - *Thread-safe* pero no sincronizado
    - Cada operación de modificación provoca la creación de un nuevo array subyacente.
    - Adecuado en contextos concurrentes para iterar de una manera segura, pero sin el coste de la sincronización.

- *List<E>*
  - *CopyOnWriteArrayList*
    - *Thread-safe* pero no sincronizado
    - Cada operación de modificación provoca la creación de un nuevo array subyacente.
    - Adecuado en contextos concurrentes para iterar de una manera segura, pero sin el coste de la sincronización.

- *Map<K,V>*
  - *EnumMap*
    - *Implementación de un Map de alto rendimiento, donde las claves son valores de una enumeración.*
  - *WeakHashMap*
    - Implementación de un Map que tiene referencias débiles a sus claves.
    - Si una clave no se va a usar más, el *garbage collector* puede eliminar el par del Map.

9) ALGORITMOS PARA COLECCIONES

Disponibles en la clase Colletions (ojo en plural)

Ordenar/desordenar

- Collections.sort(list)

  Recibe una colección de tipo List<E>.

  Modifica la colección que recibe.

  Los elementos deben implementar Comparable.

- Collections.sort(list, comparator)

  Implementación que ordena según el orden inducido por la instancia de comparator.

- Collections.shuffle(list)

  Realiza la operación opuesta a sort

Desordena los elementos de una colección

- Collections.shuffle(list, random)

  Versión del anterior, donde el componente aleatorio se le puede proporcionar.

Buscar

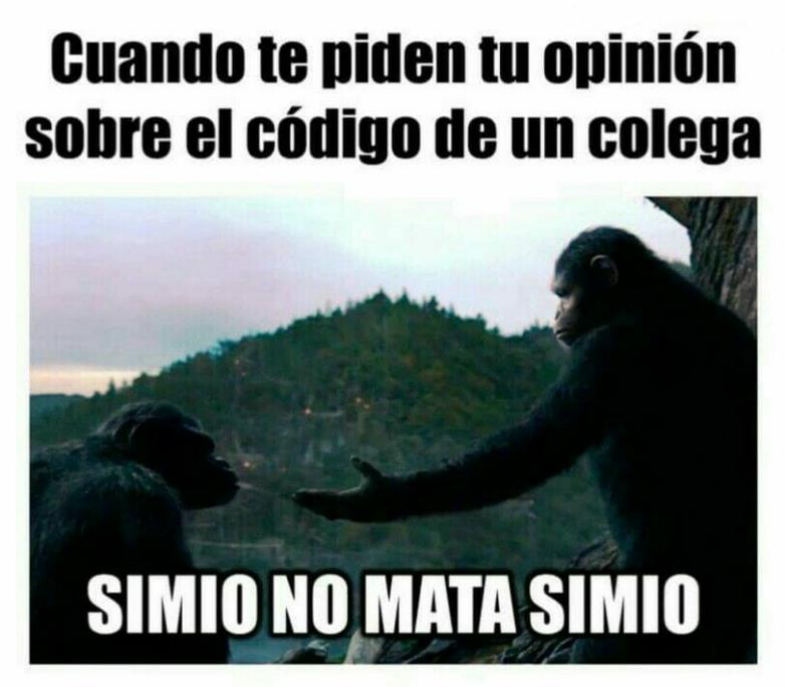
- Búsqueda binaria.
- La lista debe estar ordenada previamente.
- Si hay varias ocurrencias, no se garantiza cual se va a encontrar.
- Devuelve
  - el índice del elemento si lo encuentra
  - si no, (-(punto de inserción)-1).
    - Esto garantiza que se devuelve un valor mayor que cero sólo si se encuentra

Operaciones: max/min, frecuencia...

- Máximo y mínimo
  - Acorde al orden natural
  - Otra implementación con un Comparator.
  - Aplicable a Collection.
- Frecuencia
  - Devuelve el número de ocurrencias de un objeto en una colección.

- Intersección vacía
  - Compara dos colecciones
  - Devuelve true si no hay ningún elemento común entre ambas.

NOTA: Existen múltiples librerías de colecciones de terceros para poder usar pero que no estudiaremos en este curso, como son Guava (implementada por Google, disponibles para java y Android), Eclipse Collections, Apache Commons Collections...


Cuando te piden tu opinión sobre el código de un colega
SIMIO NO MATA SIMIO