DAM/DAW

PROGRAMACIÓN

08.2

Java Collections Framework I

fran@iessanclemente.net

Rev: 3.2

Indice

Introducción (I)

- El *Java Collections Framework* es un conjunto de clases e interfaces que implementa estructuras de datos de uso habitual para colecciones de objetos: listas, conjuntos, mapas,...
- Aunque su nombre oficial incluye el término *"framework"*, su funcionamiento es el de una librería. Basicamente, pone a nuestra disposición su conjunto de interfaces y las clases que los implementan para reutilizarlas libremente en nuestras aplicaciones.
- Tanto colecciones como *arrays* nos permiten almacenar un conjunto de referencias a objetos y manejarlas como un único grupo. La principal diferencia radica en que, a diferencia de los arrays, las colecciones son dinámicas. No necesitan que especifiquemos cuál va a ser su capacidad en el momento de la instanciación y pueden "crecer" y "encoger" de forma automática a medida que añadimos o eliminamos objetos de la misma.

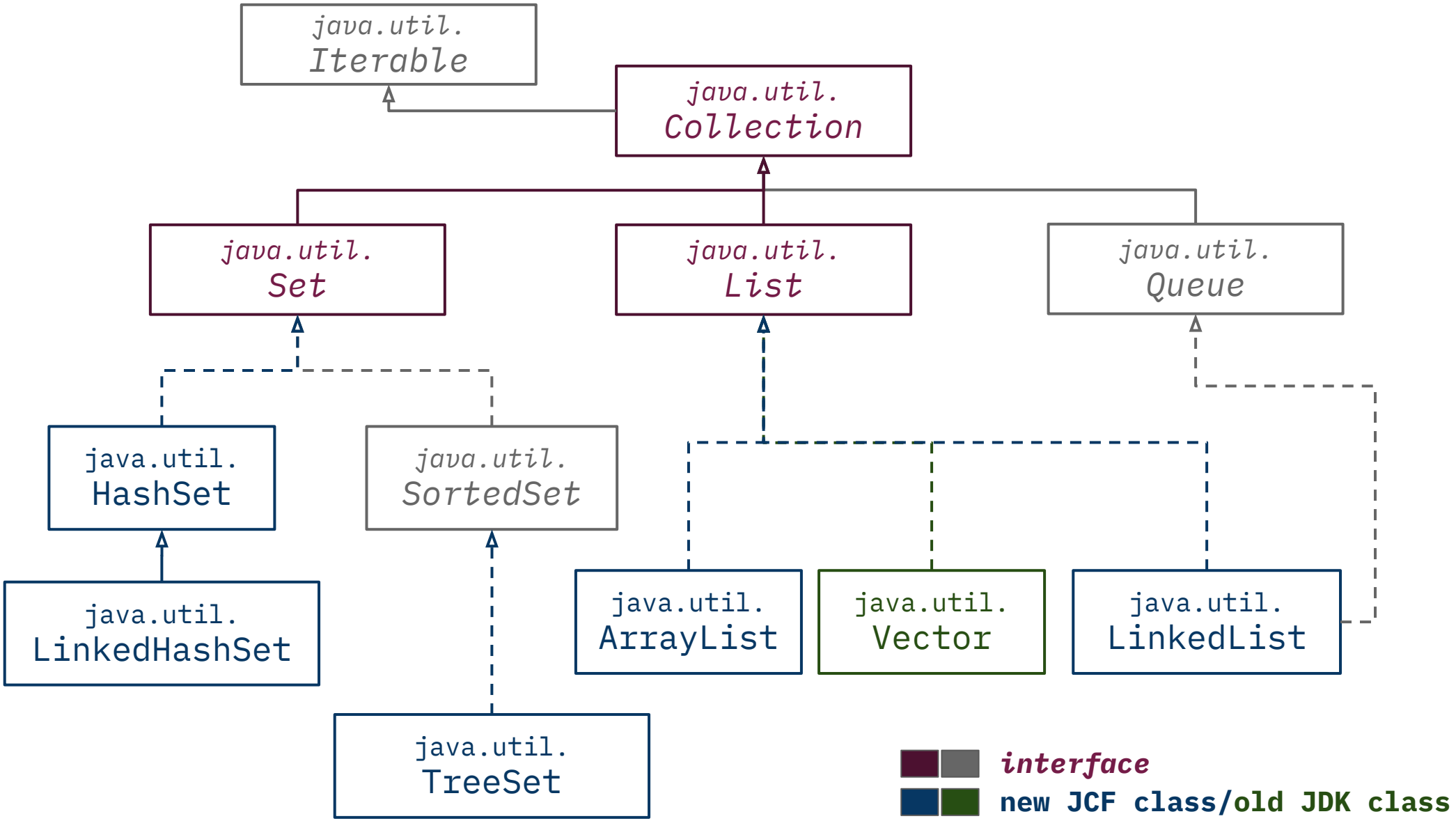
Introducción (y II)

- Las colecciones de Java sólo pueden almacenar tipos referenciados, por tanto, no podemos crear colecciones de tipos primitivos (int, double,...). Para almacenar valores de estos tipos, deberemos hacerlos a través de sus clases *wrapper* (envoltorio): *Integer*, *Long*, *Double*,...
- Introducido en el JDK1.2, la incorporación del soporte de genéricos en JSE 5 supuso la adaptación del JCF a la nueva sintaxis. Esto trajo consigo un gran avance, al añadir a las clases del JCF seguridad de tipos (*type safety*) en tiempo de compilación y eliminar la necesidad de *cast* (desde *Object*) al acceder a los miembros de la colección
- En resumen, el JCF es una amplia librería que proporciona numerosos "modelos organizativos" de datos, lo que se denominan *tipos de datos abstractos*, junto con la implementación de diversos algoritmos que nos permitirán operar sobre los objetos almacenados (buscar, ordenar,...)

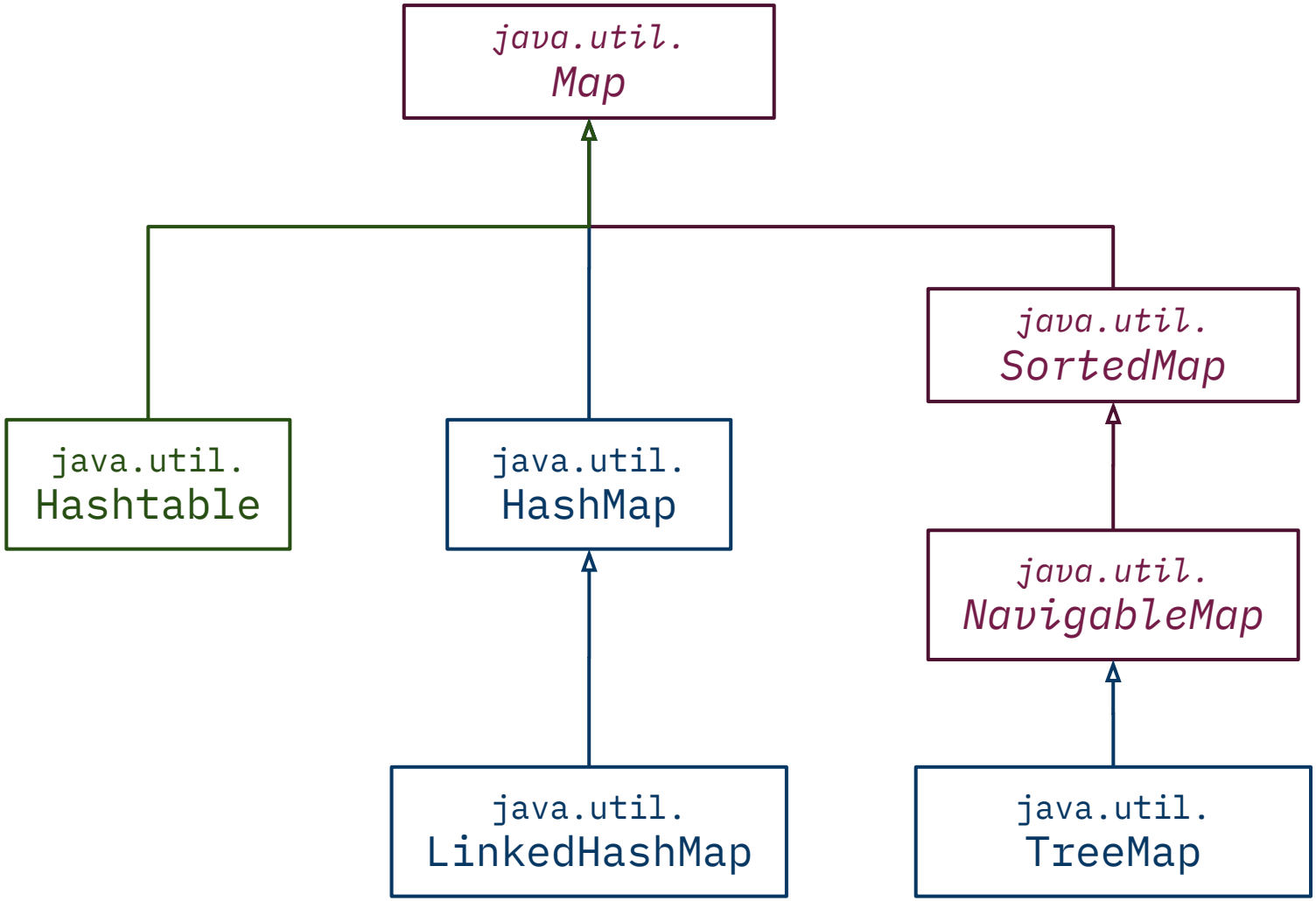
Arquitectura (I)

- Casi todas las colecciones en Java derivan del interface *java.util.Collection*
- El interfaz *Collection* define un conjunto de métodos que serán comunes en todas las colecciones que lo implementan. Así, este interfaz común nos permitirá, entre otras cosas, añadir, eliminar y buscar elementos de forma homogénea transparente al tipo concreto de colección
- Existen tres tipos principales de colecciones:
  - *listas ordenadas*, donde los elementos son almacenados y recuperados en base a un determinado orden de inserción en la colección
  - *conjuntos (sets)*, donde los elementos son almacenados sin un orden específico y lo único relevante es la "pertenencia" al conjunto
  - *mapas*, donde los datos se almacenan junto a una clave que los identifica. En lugar del interfaz *Collection*, implementan *java.util.Map* y conforman una jerarquía de clases separada

Arquitectura (III)



Arquitectura (y IIV)



El interfaz *Collection* (I)

- El interfaz *java.util.Collection* define la funcionalidad central que esperamos de cualquier colección que no sea un mapa
- Sus métodos (*ver figura de la página siguiente*) se agrupan en cuatro grupos principales:
  - añadir elementos,
  - eliminar elementos,
  - consultar el contenido de la colección,
  - hacer accesible el contenido de la colección para procesamiento fuera de la colección
- Al ser *Collection* un subinterfaz de *Java.lang.Iterable*, cualquier colección que lo implemente podrá ser recorrida mediante bucles *for-each*

El interfaz *Collection* (II)

- Métodos del interfaz *Collection*:

Collection<E>	**Descripción**
boolean **add**(E e)	Añade el nuevo objeto `e` a la colección
boolean **addAll**(Collection<? extends E> c)	Añade todos los elementos de la colección `c`
void **clear**()	Elimina todos los elementos de la colección
boolean **remove**(Object o)	Elimina el objeto `o` de la colección
boolean **removeAll**(Collection<?> c)	Elimina todos los elementos que se encuentren también en `c`
boolean **retainAll**(Collection<?> c)	Mantiene **sólo** los elementos que se encuentren también `c`
boolean **contains**(Object o)	*true* si la colección contiene al objeto; *false* en otro caso
boolean **containsAll**(Collection<?> c)	*true* si la colección contiene a todos los elementos de `c`
boolean **isEmpty**()	*true* si la colección está vacía
int **size**()	Devuelve el número de elementos de la colección
Iterator<E> **iterator()**	Devuelve un iterador sobre la colección
Object[] **toArray**()	Devuelve un array con todos los objetos de la colección
T[] **toArray**(T[] a)	Devuelve un array con todos los objetos del tipo del array `a`

El interfaz *Collection* (III)

❖ Añadir elementos:

`boolean add(E e)`	Añade el nuevo objeto `e` a la colección
`boolean addAll(Collection<? extends E> c)`	Añade todos los elementos de la colección `c`

- El argumento *boolean* devuelto por este método indica si la colección ha sido modificada en la ejecución.
- Puede ser *false* en colecciones, como los conjuntos, que no se modifican si se trata de añadir un elemento que ya está presente en la colección. Si, por alguna otra razón, la colección rechazara el elemento (algunas colecciones no aceptan *null's*), el método lanzará una excepción
- De la firma de los métodos observamos que, como es de esperar, sólo podemos añadir elementos del tipo parametrizado (E) especificado por la colección en el momento de sus creación (o subclases)

El interfaz *Collection* (IV)

❖ Eliminar elementos:

`void clear()`	Elimina todos los elementos de la colección
`boolean remove(Object o)`	Elimina el objeto `o` de la colección
`boolean removeAll(Collection<?> c)`	Elimina todos los elementos que se encuentren también en `c`
`boolean retainAll(Collection<?> c)`	Mantiene **sólo** los elementos que se encuentren también `c`

- *remove* eliminará el primer elemento e de la colección para el que se cumpla que *o.equals(e)* es verdadero. Si no encuentra ningún elemento que verifique la condición anterior, la colección permanecerá sin cambios.
- El argumento *boolean* devuelto por este método indica si la colección ha sido modificada en la ejecución.
- A diferencia de los métodos para añadir, los métodos para eliminar elementos aceptan objetos de cualquier tipo

El interfaz *Collection* (V)

❖ Consultar el contenido de la colección:

boolean **contains**(Object o)	*true* si la colección contiene al objeto o; *false* en otro caso
boolean **containsAll**(Collection<?> c)	*true* si la colección contiene a todos los elementos de c
boolean **isEmpty**()	*true* si la colección está vacía
int **size**()	Devuelve el número de elementos de la colección

- En el caso de que la colección contuviera más elementos que el valor máximo del tipo *int* (*Integer.MAX_VALUE*, más de dos mil millones de elementos) se devolverá el valor *Integer.MAX_VALUE*
- A diferencia de los métodos para añadir, los métodos para consultar el contenido de la colección, aceptan objetos de cualquier tipo

El interfaz *Collection* (y VI)

❖ Habilitar acceso al contenido de la colección:

`Iterator<E> iterator()`	Devuelve un iterador sobre la colección
`Object[] toArray()`	Devuelve un array con todos los objetos de la colección
`T[] toArray(T[] a)`	Devuelve un array con todos los objetos del tipo del array `a`

- Existen dos versiones sobrecargadas del método *toArray*. Ambas nos devolverán una array con todos los miembros de la colección
- La primera de ellas nos devolverá un array de *Object*
- La segunda versión emplea el tipo del array pasado como argumento para determinar en tiempo de ejecución el tipo del array devuelto. En caso de que el *array* pasado como argumento tenga suficiente espacio para almacenar los elementos de la colección, se empleará esta para hacer el volcado en lugar de crear un nuevo *array*

El interfaz *List* (I)

- La lista es probablemente el tipo de colección más usada
- Las listas nos permitirán almacenar datos ordenados por su orden de inserción (*index*), lo que facilitará el acceso aleatorio a los elementos
- A diferencia de los conjuntos, permiten la repetición de elementos
- La especificación original de JCF (JDK1.2), introdujo dos nuevas clases que implementaban la interfaz *List*: *ArrayList* y *LinkedList*
- La diferencia fundamental entre ambas clases radica en el tipo de estructura de datos interna que da soporte al almacenamiento de los elementos: *array* y lista enlazada.
- Esta diferencia de estructura de almacenamiento hace que el *ArrayList* tenga mejor desempeño cuando realizamos numerosas operaciones de lectura sobre la colección (acceso aleatorio) mientras que *LinkedList* se comporta mejor cuando las inserciones y borrados son habituales

El interfaz *List* (II)

- La clase *Vector*, introducida en JDK1.0, es similar a *ArrayList* en el sentido de que emplea igualmente un *array* como estructura interna. Con la aparición de JCF en JDK1.2 fue reescrita para implementar List
- La diferencia fundamental entre *Vector* y *ArrayList* es que la primera soporta operaciones concurrentes (*thread-safe*) de forma que sólo un hilo de ejecución (*thread*) pueda realizar cambios estructurales (add, remove,...) sobre la lista (*thread synchronization*). Dado el coste en rendimiento que supone esta sincronización, *ArrayList* se diseño no *thread-safe*
- De forma general, debido a la diferencia de rendimiento, optaremos siempre por emplear la clase *ArrayList*. Si dicha lista va a operar en un entorno *multithread*, deberemos encargarnos de la sincronización. Para ello, aplicaremos el modificador *synchronized* al método o bloque que actúe sobre la lista, o usaremos *java.util.Collections.synchronizedList()* para obtener una versión sincronizada de nuestro *ArrayList*

El interfaz *List* (III)

- Adicionalmente a los heredados de *Collection*, el interfaz *List* nos aporta nuevos métodos relacionados con el acceso aleatorio (posicional) a los elementos de la lista

List<E>	**Descripción**
`boolean` **`add`**`(int i, E e)`	Añade el objeto `e` a la colección en la posición `i`
`boolean` **`addAll`**`(int i, Collection<? extends E> c)`	Añade todos los elementos de la colección `c` a partir de la posición `i`
`E` **`get`**`(int i)`	Devuelve el elemento en la posición `i`
`E` **`remove`**`(int i)`	Elimina el objeto en la posición `i`
`E` **`set`**`(int i, E e)`	Reemplaza el elemento en la posición `i` por el objeto `e`
`int` **`indexOf`**`(Object o)`	Devuelve la posición de la **primera** ocurrencia del objeto `o` en la colección; `-1` si no se encuentra
`int` **`lastIndexOf`**`(Object o)`	Devuelve la posición de la **última** ocurrencia del objeto `o` en la colección; `-1` si no se encuentra
`List<E>` **`subList`**`(int from, int to)`	Devuelve una lista con los elementos de la colección desde la posición `from` hasta la posición (`to`-1)
`ListIterator<E>` **`listIterator`**`()` `ListIterator<E>` **`listIterator`**`(int i)`	Amplían las capacidades del Iterator devuelto por *Collection* con nuevos métodos para obtener la posición de los elementos de la colección y recorrerla en ambos sentidos

El interfaz *List* (IV)

- Veamos algún ejemplo simple de creación y uso listas...

```
import java.util.ArrayList;
public class ListaDemo1 {
   public static void main(String[] args) {
       ArrayList<Integer> lista = new ArrayList<>();
       lista.add(3); lista.add(5); lista.add(1); lista.add(4); lista.add(5);
       System.out.println(lista);
       System.out.println("lista[2] = " + lista.get(2));
       lista.set(2, 8);
       System.out.println("lista[2] = " + lista.get(2));
       lista.remove(2);
       System.out.println(lista);
       System.out.println("lista[2] = " + lista.get(2));
       System.out.print("lista = ");
       for(int i:lista) { System.out.print(i + " ");}
   }
}
```

```
[3, 5, 1, 4, 5]
lista[2] = 1
lista[2] = 8
[3, 5, 4, 5]
lista[2] = 4
lista = 3 5 4 5
```

El interfaz *List* (y V)

- Podemos crear listas para contener nuestros propios objetos...

```
import java.util.*;
class Libro {
    String isbn;
    String tit;
    String aut;
    public Libro(String isbn, String tit, String aut) {
        this.isbn = isbn; this.tit = tit; this.aut = aut;
    }
    public String toString() { return "\"" + this.tit + "\""; }
}
public class ListaDemo2 {
   public static void main(String[] args) {
        List<Libro> biblioteca = new ArrayList<>();
        Libro libro1 = new Libro("0451457998", "2001: A Space Odissey", "Arthur C. Clarke");
        biblioteca.add(libro1);
        biblioteca.add(new Libro("0345404475", "Do Androids Dream of Electric Sheep?", "Philip K. Dick"));
        System.out.println(biblioteca);
   }
}
```

```
["2001: A Space Odissey", "Do Androids Dream of Electric Sheep?"]
```

Comparaciones y búsquedas (I)

- Tanto el interfaz *Collections* como el subinterfaz *List*, nos proporcionan métodos que nos permiten comprobar si un objeto se encuentra en la colección (*contains*, *indexOf*,...)

```
import java.util.*;
public class ListaDemo3 {
   public static void main(String[] args) {
       List<String> lista = new ArrayList<>();
       String s = "bye";
       lista.add("hi");
       lista.add(s);
       System.out.println("hello está en la lista? " + lista.contains("hello"));
       System.out.println("bye está en la lista? " + lista.contains("bye"));
       System.out.println(s + " está en la lista? " + lista.contains(s));
       System.out.println("posición de " + s + " en la lista? " + lista.indexOf(s));
   }
}
```

```
hello está en la lista? false
bye está en la lista? true
bye está en la lista? true
posición de bye en la lista? 1
```

Comparaciones y búsquedas (II)

- Al comparar instancias de una clase, debemos distinguir entre aquellas clases donde cada instancia representa un "objeto del problema" único en si mismo, de las llamadas "clases-valor", donde las instancias representan "valores" y, por tanto, dos instancias con el mismo valor deben ser vistas como iguales (tal es el caso de las instancias de String, donde dos intancias con la misma cadena, aunque para Java son dos objetos diferentes (con su espacio de memoria diferenciado), a efectos de comparación son iguales)

- Volviendo a nuestro ejemplo anterior, dadas dos instancias de la clase Libro con los mismos valores de *ISBN*, *Título* y *Autor* (es decir, que hacen referencia a una misma publicación), ¿cómo deben ser tratadas por nuestra aplicación a efectos de comparación? ¿deben ser vistas como iguales o diferentes? Pues... depende!

Comparaciones y búsquedas (III)

- *Situación A)*
  - Consideramos que cada instancia de Libro representa un ejemplar único de la publicación. Piensa en una biblioteca donde puede haber múltiples "copias" (ejemplares) del mismo libro y nos interesa tratarlos como objetos "físicamente" diferentes (préstamos)
- En este caso, el comportamiento por defecto de Java al comparar objetos (compara referencias no el contenido) nos vale para determinar si dos objetos Libro son iguales o no

```
Libro libro1 = new Libro("0345404475", "Do Androids Dream of Electric Sheep?", "Philip K.Dick");
Libro libro2 = libro1;
Libro libro3 = new Libro("0345404475", "Do Androids Dream of Electric Sheep?", "Philip K.Dick");

libro1 == libro2 // ¿? --> true (ambas variables contienen la misma referencia de instancia,
                 //              es decir, ambas variables "apuntan" al mismo objeto en memoria)
libro1 == libro3 // ¿? --> false (son dos objetos diferentes y, por tanto, sus referencias
                 //               también lo son)
```

Comparaciones y búsquedas (IV)

- Por supuesto, ese comportamiento afectará a las operaciones de búsqueda que hagamos con nuestras colecciones de Libros...

```
Libro libro1 = new Libro("0345404475", "Do Androids Dream of Electric Sheep?", "Philip K.Dick");
Libro libro2 = libro1;
Libro libro3 = new Libro("0345404475", "Do Androids Dream of Electric Sheep?", "Philip K.Dick");

List<Libro> lista = new ArrayList<>()
lista.add(libro1);
lista.add(libro2);

System.out.println(lista.contains(libro1)); // ¿? --> true
System.out.println(lista.contains(libro2)); // ¿? --> true
System.out.println(lista.contains(libro3)); // ¿? --> false
System.out.println(lista.contains(
        new Libro("0345404475", "Do Androids Dream of Electric Sheep?", "Philip K.Dick")
    )); // ¿? --> false
```

- Este comportamiento puede que no nos interes en otras situaciones...

Comparaciones y búsquedas (V)

- *Situación B)*
  - Consideramos que cada instancia de Libro representa una publicación (no un ejemplar). Este sería un caso de *clase-valor* donde las instancias representan "valores" y no objetos independientes. Piensa en una librería *on-line* donde sólo necesitamos la información de la publicación (producto) pero no de los ejemplares físicos individuales (basta con un atributo que nos indique el *stock* actual de unidades)
- En este caso, la comparación basada en referencias ya no nos servirá. Necesitamos poder comparar los objetos en base a los "valores" que representan, no en base a sus referencias
- Volviendo al ejemplo anterior, necesitamos que Java nos diga que dos objetos Libro con el mismo ISBN son iguales a efectos de comparción (aún siendo realmente instancias u objetos diferentes)

Comparaciones y búsquedas (VI)

❖ El método *equals( )*

- Para poder realizar estas comprobaciones de forma adecuada, Java nos ofrece una solución simple consistente en el reescritura del método *equals()* que toda clase hereda de *Object*

```
public boolean equals(Object obj)
```

- Sobreescribiendo este método en nuestras clases, podremos redefinir la manera en que nuestros objetos son comparados entre sí, adecuando los "criterios" de comparación a cada caso particular
- El método deberá devolver un valor *booleano* indicando si el objeto invocante del método es igual (o no) al objeto pasado como parámetro
- Fíjate que el método define como parámetro una variable de tipo Object. Para poder hacer las operaciones de comprobación correspondientes, deberemos hacer un *cast* de ese Object a nuestra clase particular

Comparaciones y búsquedas (VII)

- Nuestra clase Libro quedaría así...

```
class Libro {
    // ...
    public boolean equals(Object obj) {
        return this.isbn.equals(((Libro)obj).isbn);  // comparamos los atributos isbn de ambos objetos
    }
}
```

- Y las comparaciones con *equals()*...

```
Libro libro1 = new Libro("0345404475", "Do Androids Dream of Electric Sheep?", "Philip K.Dick");
Libro libro2 = libro1;
Libro libro3 = new Libro("0345404475", "Do Androids Dream of Electric Sheep?", "Philip K.Dick");
Libro libro4 = new Libro("0451457998", "2001: A Space Odissey", "Arthur C. Clarke");

libro1.equals(libro2) // ¿? --> true
libro1.equals(libro3) // ¿? --> true
libro1.equals(libro4) // ¿? --> false
```

Comparaciones y búsquedas (VIII)

- El resultado de las operaciones de búsqueda que hagamos con nuestras colecciones de Libros va a ser diferente ahora...

```
Libro libro1 = new Libro("0345404475", "Do Androids Dream of Electric Sheep?", "Philip K.Dick");
Libro libro2 = libro1;
Libro libro3 = new Libro("0345404475", "Do Androids Dream of Electric Sheep?", "Philip K.Dick");

List<Libro> lista = new ArrayList<>()
lista.add(libro1);
lista.add(libro2);
System.out.println(lista.contains(libro1)); // ¿? --> true
System.out.println(lista.contains(libro2)); // ¿? --> true
System.out.println(lista.contains(libro3)); // ¿? --> true (antes era false)
System.out.println(lista.contains(
        new Libro("0345404475", "Do Androids Dream of Electric Sheep?", "Philip K.Dick")
    )); // ¿? --> true (antes era false)
```

- Fíjate que ahora indica que hay *Libros* en la colección que no fueron "añadidos"!! (porque ahora buscamos "valores", no objetos)

Comparaciones y búsquedas (IX)

❖ El método *hasCode( )*

- Cada vez que sobreescribimos el método *equals( )*, debemos sobreescribir el método *hashCode( )*
- El método *hashCode( )* devuelve un identificador del objeto que es empleado en colecciones del tipo HashMap o HashSet donde las búsquedas u ordenaciones se realizan en base al valor de *hash* devuelto por dicho método.
- La especificación de *Object* establece el siguiente contrato respecto al método *hashCode( )*:
  - ➢ Durante la ejecución de una aplicación, la invocación de *hashCode( )* sobre un objeto devolverá siempre el mismo valor, siempre y cuando ninguno de los atributos empleados en *equals( )* se hayan modificado. Dicho valor podrá variar entre diferentes ejecuciones del programa

Comparaciones y búsquedas (X)

- Si dos objetos son iguales según el método *equals( )*, ambos objetos deben devolver el mismo valor al invocar su método *hashCode( )*
- Si dos objetos son diferentes según el método *equals( )*, no están obligados a devolver un valor diferente al invocar su método *hashCode()*

- De los puntos indicados, nos interesa especialmente la segunda:

  "objetos iguales deben tener código hash iguales"

- Sin embargo, la implementación por defecto de *hashCode( )* devuelve valores diferentes para cada objeto, lo cual no es válido en nuestro caso *B)*:

```
Libro libro1 = new Libro("0345404475", "Do Androids Dream of Electric Sheep?", "Philip K.Dick");
Libro libro2 = new Libro("0345404475", "Do Androids Dream of Electric Sheep?", "Philip K.Dick");
libro1.equals(libro2) // ¿? --> true
libro1.hashCode();    // ==> 225534817
libro2.hashCode();    // ==> 1878246837
```

Comparaciones y búsquedas (XI)

- La solución pasa, como en el caso de *equals()*, por reescribir el método *hashCode()* para que genere el mismo número *hash* para instancias de la clase que representen el mismo "objeto lógico"

```
public int hashCode()
```

- No existe ningún requerimiento específico para la implementación de dicho método, más allá de que debe devolver un número entero y que ese número debería ser igual para aquellas instancias consideradas iguales por el método *equals()*.

- No estamos obligados a generar un valor único para cada instancia diferente de la clase, pero es muy conveniente si no queremos tener problemas con colecciones (u otras construcciones) que se basen en el empleo del valor *hash* del objeto para su correcto funcionamiento

Comparaciones y búsquedas (XII)

- Existen diferentes métodos para generar estos valores *hash* únicos. En general, se trata de "calcular" dicho valor a partir de los atributos significativos de la clase, es decir aquellos atributos que representan "unívocamente" un objeto de la clase (por ejemplo, el atributo *isbn* de *Libro*) y aplicarles una *función hash* (MD5, SHA1, CRC,...)

- Por ejemplo, dado que el atributo *isbn* de *Libro* es único para cada uno de ellos y lo empleamos en la comparación, podríamos hacer (usando los propios métodos *hashCode()* de las clases "wrapper" de los tipos Java):

```
class Libro {
    // ...
    public int hashCode() {
        return String.hashCode(this.isbn);
    }
}
```

Comparaciones y búsquedas (y XIII)

- Normalmente, se emplean métodos "más elaborados" que amplíen el espacio de claves y eviten colisiones (mismo *hash* para objetos diferentes)
- La siguiente es una implementación típica en la que se calcula el *hash* para uno de los campos significativos y se van acumulando los *hashes* del resto de campos (normalmente, significativos):

```
public int hashCode() {
    int result = Tipo.hashCode(atributo1);
    result += 31*result + Tipo.hashCode(atributo2);
    result += 31*result + Tipo.hashCode(atributo3);
    // ...
    return result
}
```

- Como curiosidad, usar el número 31 tiene que ver con que se trata de un número primo impar (aumenta la varianza en la distribución resultante) y que multiplicar x31 se puede implementar con un simple desplazamiento de bits (`(num<<5)-num`) (lo que redunda en el rendimiento)

Recorriendo la colección (I)

- Como vimos al principio de la unidad, el interfaz *Collection* implementa la interfaz *Iterable*
- Por tanto, toda clase que implemente *Collection* dispondrá de un método *iterator()* que nos proporcionará una instancia de *Iterator*

```
Iterator<E> iterator()
```

- *Iterator*, a su vez, es un interfaz que define una serie de métodos que nos permiten recorrer la colección

```
boolean hasNext() // Devuelve true si la colección tiene más elementos
E next()          // Devuelve el siguiente elemento de la colección
```

- Este *Iterator* que devuelve la colección es el que solicita la construcción *for-each* para recorrerla. Así, mediante sus métodos, se puede ir extrayendo cada uno de sus elementos desde el primero al último

Recorriendo la colección (II)

- Nosotros, utilizando dicho Iterator, también podemos acceder secuencialmente a los elementos de la colección...

```
Libro libro1 = new Libro("0345404475", "Do Androids Dream of Electric Sheep?", "Philip K.Dick");
Libro libro2 = new Libro("0451457998", "2001: A Space Odissey", "Arthur C. Clarke");

List<Libro> biblio = new ArrayList<>()
lista.add(libro1);
lista.add(libro2);

Iterator<Libro> it = biblio.iterator(); // Iterator que apuntará al inicio de la colección
while(it.hasNext()) {            // mientras haya "siguiente" elemento...
    Libro libro = it.next();  // leemos el "siguiente" elemento
    System.out.println("Título: " + libro));
}
```

```
Título: "Do Androids Dream of Electric Sheep?"
Título: "2001: A Space Odissey"
```

Recorriendo la colección (III)

- Adicionalmente, el interfaz *List* proporciona un iterador especial, denominado *ListIterator*, que extiende las capacidades de *Iterator* permitiendo la inserción, borrado y modificación de elementos, acceso bidireccional y la obtención de la posición (*índex*) del elemento en la lista
- Además, pondremos inicializar el *ListIterator* desde cualquier posición

`ListIterator<E> listIterator()`	Devuelve un iterador sobre la colección
`ListIterator<E> listIterator(int p)`	Devuelve un iterador sobre la colección empezando en la posición `p`
`boolean hasNext()`	Devuelve `true` si hay más elementos avanzando hacia adelante
`boolean hasPrevious()`	Devuelve `true` si hay más elementos avanzando hacia atrás
`E next()`	Devuelve el siguiente elemento avanzando hacia adelante
`E previous()`	Devuelve el siguiente elementos avanzando hacia atrás
`int nextIndex()`	Devuelve el índice del siguiente elemento hacia adelante
`int previousIndex()`	Devuelve el índice del siguiente elemento hacia atrás

Recorriendo la colección (IV)

- Vamos a reescribir el código anterior utilizando el *ListIterator* para extraer las posiciones de los elementos de la lista...

```
Libro libro1 = new Libro("0345404475", "Do Androids Dream of Electric Sheep?", "Philip K.Dick");
Libro libro2 = new Libro("0451457998", "2001: A Space Odissey", "Arthur C. Clarke");

List<Libro> biblio = new ArrayList<>()
lista.add(libro1);
lista.add(libro2);

ListIterator<Libro> it = biblio.listIterator(); //devuelve un iterador al inicio de la colección
while(it.hasNext()) {            // mientras haya "siguiente" elemento...
    int p = it.nextIndex();  // índice del "siguiente" elemento (llamar antes de next())
    Libro libro = it.next();  // leemos el "siguiente" elemento
    System.out.println("idx: " + p + " -> Título: " + libro));
}
```

```
idx: 0 -> Título: "Do Androids Dream of Electric Sheep?"
idx: 1 -> Título: "2001: A Space Odissey"
```

Recorriendo la colección (V)

- *ListIterator* nos proporciona métodos que nos permitirán alterar tanto la estructura (número de elementos) como el contenido (valores) de la lista al mismo tiempo que la recorremos

`void add(E e)`	Inserta el elemento e en la lista El nuevo elemento es insertado inmediatamente antes del elemento que sería devuelto por next(), si lo hubiera, y después del elemento que sería devuelto por previous(), si lo hubiera
`void remove()`	Elimina de la lista el último elemento que fue devuelto por next() o previous(). Este método sólo puede ser invocado una vez por cada llamada a next() o previous() y si no se llamó a add(E) tras ellos
`void set(E e)`	Reemplaza el último elemento devuelto por next() o previous() con el elemento e especificado Este método sólo puede ser invocado si no hubo una llamada a remove() o add(E) después de la última llamada a next() o previous()

Recorriendo la colección (VI)

- El siguiente ejemplo usa los métodos de *ListIterator* para eliminar los números pares y sustituir los impares por su valor al cuadrado mientras recorremos la lista

```
Integer[] nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
List<Integer> lista_nums = new ArrayList<>(Arrays.asList(nums));
System.out.println(lista_nums);

ListIterator<Integer> it = lista_nums.listIterator();
while(it.hasNext()) {
    Integer i = it.next();
    if(i%2==0)
        it.remove();
    else
        it.set(i*i);
}
System.out.println(lista_nums);
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 9, 25, 49, 81]
```

Recorriendo la colección (VII)

- Debemos tener presente que el intefaz *Listener* (y sus subinterfaces como *ListListener*) son interfaces genéricos
- Cuando los declaremos, es conveniente que les asignemos un tipo parametrizado (normalmente será el propio de la colección desde la cual lo creamos). Por ejemplo:

```
List<Integer> lista_nums = new ArrayList<>();

ListIterator<Integer> it1 = lista_nums.listIterator(); // tipo parametrizado Integer

ListIterator it2 = lista_nums.listIterator(); // sin indicar tipo; usará Object
```

- Si no lo hacemos (como en el segundo caso), Java empleará *Object* como argumento tipo, lo que va a suponer ciertas "divergencias" a la hora de operar con dichos iteradores, tal como veremos a continuación

Recorriendo la colección (VIII)

- En primer lugar, nos obligará a realizar un *cast* explícito al iterar sobre la colección...

```
Integer i1 = it1.next(); // sin cast
Integer i2 = (Integer)it2.next(); // tenemos que hacer un cast: Object -> Integer
```

- Esta operación no es *type-safe* pues, en tiempo de compilación, no se puede garantizar que el objeto devuelto por la colección sea *Integer*
- En general, cuando trabajemos con iteradores sin especificación de tipo, operaciones como *remove( )* pueden considerarse seguras. Sin embargo, aquellas operaciones que alteren el contenido de la colección (*add*, *set*,...) no lo serán. De hecho, el compilador nos informará de este hecho mediante algún *warning* similar a los siguientes:

```
unchecked call to add(E) as a member of the raw type java.util.ListIterator
XXX...X.java uses unchecked or unsafe operations
...
```

Recorriendo la colección (y IX)

- En ocasiones, se tiende a pasar por alto estos *warnings*, incluso a suprimirlos añadiendo a nuestro código una anotación como la siguiente:

```
@SuppressWarnings("unchecked")
```

- Como vemos a continuación, las consecuencias podrían ser desastrosas...

```
List<Integer> lista_nums = new ArrayList<>();
ListIterator it2 = lista_nums.listIterator(); // sin indicar tipo; usará Object
it2.add(3);  // OK
it2.add(5);  // OK
it2.add("ups!!");  // La colección "traga" porque, tras compilar, almacena Object's
System.out.println(lista_nums);  // imprime la colección (no hay problema)
for (Integer i : lista_nums) System.out.println(i + "\u00B2 = " + i*i);  // Uuuy!!!
```

```
[3, 5, ups!!]
3² = 9
5² = 25
Exception in thread "main" java.lang.ClassCastException: class java.lang.String cannot be cast to
class java.lang.Integer
```

Tenemos un *String* en nuestra colección de *Integer!!!*