Introducción a la Programación

Tema 6. Colecciones I

1. Introducción

Cuando se plantea el desarrollo de programas, resulta muy conveniente establecer los criterios que permitan la reutilización de código, bien usando el que ya existe, o diseñando el nuestro para que pueda ser mantenible y reusable en el futuro. Una estrategía muy útil para este fin de reutilización es disponer de un conjunto amplio de tipos que datos que permitan hacer nuestros programas más rápidamente, más fáciles de comprender, de probar y de mantener.

Los tipos de datos ofrecidos por distintos lenguajes han ido evolucionando tanto en su diseño como en los nombres que han se les han asignado. De hecho, no existe todavía un conjunto de tipos de datos estándar común a los lenguajes disponibles. En este tema vamos a ver un conjunto de tipos de datos básico que son de los más usados en la programación.

Si bien el núcleo fundamental de los tipos que se estudiarán son los formados por los tipos de datos ofrecidos en el API de Java, además de estos estudiaremos otros que, aunque no aparecen en el API de Java, y por lo tanto tienen versiones menos estandarizadas, es importante conocer. Se estudian, tambien en algunos casos, versiones alternativas a las ofrecidas en el citado API.

En general trabajar con un conjunto bien definido y amplio de tipos de datos tiene bastantes ventajas:

- Reduce esfuerzos de programación pues proporciona tipos de datos cuya implementación puede ser reutilizada.
- Incrementa la velocidad y calidad de los programas pues proporciona implementaciones optimizadas y con garantías de no contener errores.
- Ayuda a la interoperabilidad entre aplicaciones pues facilita tipos de datos comunes entre ellas.
- Reduce esfuerzos de aprendizaje y diseño.
- Si el tipo de dato está bien diseñado, suele ser más sencillo establecer estrategias de pruebas sobre ellos.

2. Vistas, tipos no modificables y tipos inmutables

Introducimos ahora un conjunto de conceptos que usaremos en el resto del capítulo: vistas, tipos no modificables y tipos inmutables.

Un tipo decimos que es un **tipo inmutable** si los objetos de este tipo no pueden cambiar su estado. Un tipo inmutable puede ser reconocido porque no tiene métodos set ni en general métodos mofiicadores.

Un tipo decimos que es un **tipo no modificable** si los objetos de este tipo no disponen de métodos de modificadores pero pueden cambiar su estado.

Algunos tipos se diseñan específicamente para que sus **métodos sean sincronizados**. A estos tipos los denominaremos **tipos concurrentes**. En estos tipos se garantiza que cada método está siendo invocado en cada momento por una única hebra. Estos tipos y en general la programación concurrente se verá en otra asignatura pero aquí presentaremos algunos tipos concurrentes del API de Java.

Una variable *v* de tipo *V* es una **vista** de otra variable *a* de tipo *A* si los valores (en general los estados) de ambas variables están ligados y al cambiar el estado de una de ellas cambia también el de la otra.

Una variable *v* de tipo *A* es una **copia** de otra variable *a* de tipo *A* si los valores (en general los estados) de ambas variables son iguales en el momento de la copia pero posteriormente sus estados evolucionan de forma independiente.

Como podemos ver una copia es algo muy diferente a una vista. Hay distintos tipos de vistas: vistas con invariante, vistas no modificables, vistas sincronizadas.

Las vistas son variables cuyos valores están ligados a los valores que en cada momento tienen otra variable. Veremos diferentes tipos de vistas:

2.1 Vistas con invariante

Una variable *v* de tipo *V* es una **vista con invariante** de otra *a* de tipo *A* cuando los estados de ambas están ligadas en el sentido que cuando uno de ellos cambia también cambia el otro. Es decir si modificamos la variable *a* queda actualizada la vista v y al revés. Además los valores de v deben cumplir un invariante.

Un ejemplo de este tipo de vistas nos la proporciona el método subList del tipo List cuya cabera es: *List<E> subList(int fromIndex,int toIndex).*

Por ejemplo en el código:

```
List<Integer> a = …;
List<Integer> b = a.subList(2,5);
```

La variable *b* es una vista con invariante de *a.* La variable *b* es una lista que contiene los elementos contenidos en las casillas 2, 3 y 4 de la variable *a*. El tamaño de *b* es 3 (mientras que el de *a* es mayor) por lo tanto la operación *b.get(0)* devolverá un elemento idéntico al devuelto por *a.get(2).*

2.2 Vista no modificable

Una variable *v* de tipo *V* es una **vista no modificable** de otra *a* de tipo *A* cuando los estados de ambas están ligadas en el sentido que cuando el de a cambia también cambia el de v. Es decir si modificamos la variable *a* queda actualizada la vista v. Pero la variable v no puede ser modificada directamente poque todos sus métodos modificadores disparan la excepción *UnsupportedOperationException*.

Ejemplos de vistas no modificables *(unmodifiable)* lo proporcionan los métodos de la clase *Collections*.

```
static <E> List<E> unmodifiableList(List<? extends E> list)
static <E> Set<E> unmodifiableSet(Set<? extends E> s)
```

2.3 Vista sincronizada

Decimos que una variable *s* de un tipo *S* es una **vista sincronizada** de otra *t* de un tipo *T* si s y t toman los mismos valores, el tipo S ofrece los mismos métodos que T y los métodos de observadores de S tienen una semántica adicional a los T. Todos los métodos de S están sincronizados en el sentido de la sentencia *syncronized* de Java. No explicaremos aquí todos los detalles de esta sentencia per e forma muy sintética podemos decir en una vista sincronizada sólo puede haber en cada momento un posible usuario (una hebra) accediendo al estado del objeto. Es decir en una vista sincronizada sólo en cada momento sólo puede haber una un usuario invocando alguno de los métodos de la misma.

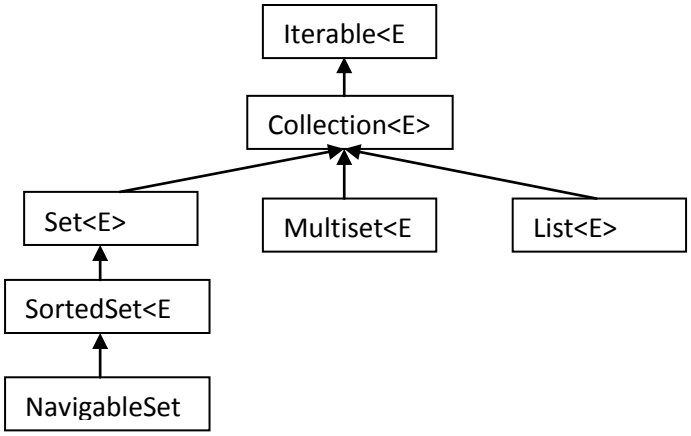
Ejemplos de vistas concurrentes *(synchronized)* lo proporcionan los métodos de la clase *Collections*.

```
static <E> List<E> synchronizedList(List<? extends E> list)
static <E> Set<E> synchronizedSet(Set<? extends E> s)
```

3. Colecciones

Las colecciones son tipos de datos adecuados para modelar agregados de elementos que se estructuran tal y como se presenta e la figura 1. En esa figura se representan un conjunto de tipos y sus relaciones de subtipado. Están disponible en la API de Java.

Uno de los aspectos importantes de las colecciones es que todos extienden el tipo *Iterable<T>*. De manera práctica esto nos va a permitir recorrerlas mediante iteradores generales y más concretamente, en el entorno de Java mediante un *for* extendido.



3.1 El Tipo Collection

El tipo *Collection* modela un agregado de objetos con la máxima generalidad. Por esta razón, en la colección no se especifica si los objetos pertenecientes a la colección pueden estar repetidos a no, si están ordenados a no. La interfaz del tipo es el que se muestra en la figura 2.

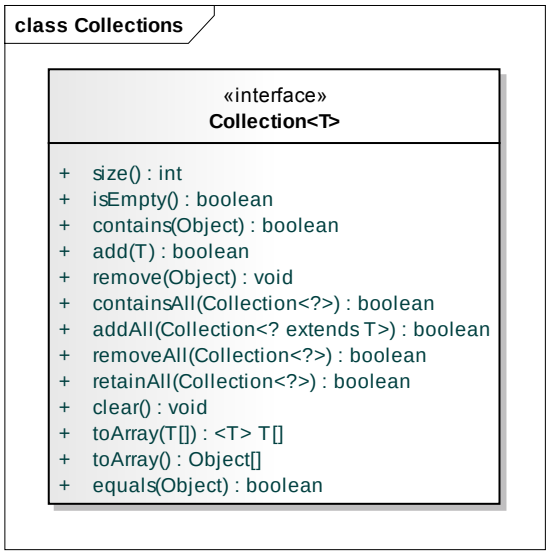


Figura 2. Interfaz de Collection

Como todos los tipos en Java *Collection* extiende al tipo *Object*. Esto implica que dispone de los métodos *equals*, *hashCode* y *toString*.

En la tabla siguiente mostramos los métodos, si son observadores o modificadores, su precondición, postcondición y resultado que devuelve en su caso. Igualmente mostramos los invariantes.

Collection<E>				
Invariante				
`size() >= 0, Se pueden añadir otros invariantes en el momento de la creación.`				
Métodos	Devuelve	O/M	Pre.	Post.

boolean equals(Object c)	Verdadero si *this* es igual a c	O	true	
int size()	Número de elementos de *this*	O	true	
boolean contains(Object e)	Verdadero si *this* contiene a e	O	true	
boolean isEmpty()	size() > 0	O	true	
void clear()		M	true	*isEmpty()*
boolean add(T o)	Verdadero si *this* ha cambiado	M	true	*contains(e)*
boolean remove(Object e)	Verdadero si *this* ha cambiado	M	true	*this* tiene una instancia menos de e si tenía una o más.
boolean addAll(Collection<? super E> c)	Verdadero si *this* ha cambiado	M	true	*c.contains(c) => this.contains(c)*
boolean removeAll(Collection<? super E> c)	Verdadero si *this* ha cambiado	M	true	*c.contains(c) => !contains(c)* *!c.contains(c)&& contains(c)@pre => contains(c)*
boolean containsAll(Collection<? super E> c)	c.contains(e) => contains(e)	O	true	

Métodos de Collection

En el momento de la creación de una colección se pueden añadir nuevos invariantes y restricciones del tipo: la colección no puede contener elementos *null*, todos los elementos de una colección deben cumplir una propiedad, la colección es no modificable, etc. Hay previsto excepciones para ser disparadas si se pretende violar alguno de estos invariantes.

- *UnsupportedOperationException:* Disparada por los métodos modificadores si la colección es no modificable.
- *ClassCastException:* Disparada por los métodos *add* y *addAll* si existen restricciones sobre el tipo de los elementos que puede contener la colección.
- *NullPointerException:* Disparada por los métodos *add* y *addAll* si no se permiten *null* como elemento de la colección
- *IllegalArgumentException:* Disparada por los métodos *add* y *addAll* si existe un invariante que restrinja las propiedades de los objetos que pertenecen a la colección.
- *IllegalStateException:* Disparada si elemento no puede ser añadido por existir restricciones de inserción en ese momento.

3.2 Notación para Colecciones

Dadas dos colecciones *a* y *b* y un elemento *e*, las operaciones sobtre ellos podemos representarlas como:

Método	*Operación Equivalente*	*Significado*
a.add(e)	a+e	Añadir elemento
a.remove(e)	a-e	Eliminar elemento

a.addAll(b)	a = a + b	Unión
a.removeAll(b)	a = a – b	Diferencia
a.retainAll(b)	a = a | b	Filtro (Intersección en Set)
a.containsAll(b)	b ≤ a	Inclusión
a.isEmpty( )	a == ∅	Está vacío?
a.size()	|a|	Cardinal
a.contains(e)	$e \in a$	Pertenencia

Para expresar de forma compacta operaciones con coleccioes y sus subtipos usaremos la notación anterior. En general a los conjuntos los representaremos por s, s_1, s_2, s_3, etc. Las listas por l, l_1, l_2, l_3 y los conjuntos ordenados por ss, ss_1, ss_2, ss_3. Los elementos susceptibles de pertenecer a una colección por e, e_1, e_2, etc.

Mas adelante veremos la forma de añadir invariantes en el momento de crear una colección o alguno de sus subtipos.

3.3 Tipo Set

Un conjunto es una colección con un invariante añadido: un conjunto tiene todos sus elementos distintos. O dicho de otra forma un conjunto no tiene elementos repetidos. El concepto de repetido coincide con la definición que marque el método equals del tipo <T>. De esta forma, el conjunto también puede definirse como una colección de elementos de tipo T en los que no existen dos elementos *e1* y *e2* que cumplan que *t1.equals(t2) == 0.*

El tipo conjunto (*Set*) hereda todos los métodos de Collection sin añadir ningún nuevo método pero si añade semántica nueva a alguno de ellos. Incluimos la semántica nueva añadida.

Set<E> extends Collection<E>		
Invariante		
`No existen dos elementos e1, e2 tal que Objects.equals(e1,e2)`		
Métodos	Devuelve	Post.
`boolean equals(Object c)`	`Verdadero si this tiene los mismos elementos que c`	

Propiedades específicas de Set respecto a Collection

Recordamos que el método *Object.equals(e1,e2)* es verdadero si *e1* y *e2* son *null* o si son distintos de *null* y es verdadero *e1.equals(e2).*

3.4 Tipo SortedSet

Un conjunto ordenado es un conjunto dotado de un orden total entre sus elementos. Ofrece algunas operaciones nuevas que permiten preguntar por las posiciones respectivas de los elementos según el orden proporcionado. Las operaciones que añade al tipo *Set<E>* son todas observadoras y se muestran en su interfaz de la Figura 4.

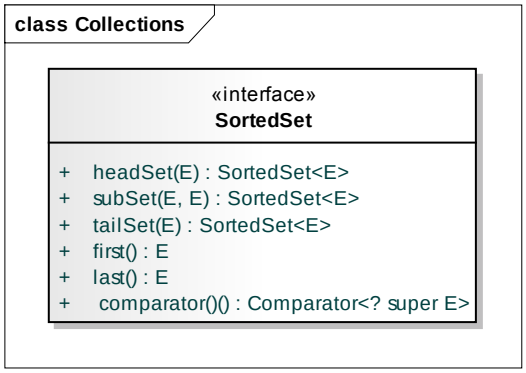


Figura 4. Interfaz de Set

El método *comparator* devuelve el orden con respecto al cual se pregunta por las posiciones de los objetos o *null* si el orden es el natural del tipo *E*.

SortedSet<E> extends Set<E>			
Invariante			
Todos los elementos pueden ser comparados según el orden del tipo No existen dos elementos e1, e2 tal que comparator().compare(e1,e2)==0			
Métodos	Devuelve	O/M	Pre.
SortedSet<E> headSet(E toElement)	Devuelve una vista con invariante e < toElement	O	true
SortedSet<E> tailSet(E fromElement)	Devuelve una vista con invariante fromElement <= e	O	true
SortedSet<E> subSet(E fromElement, E toElement)	Devuelve una vista con el invariante fromElement <= e < toElement	O	true
E first()	El primer elemento: first() <= e	O	!isEmpty()
E last()	El ultimo elemento: last() >= e	O	!isEmpty()
Comparator<? super E> comparator()	El orden del conjunto ordenado o null si está ordenado por el orden natural de E	O	true

Los métodos *headSet*, *tailSet* y *subSet* devuelven una **vista** del conjunto original formada por los elementos que cumple el invariante especificado para cada uno de ellos (hemos representado por *e* un elemento cualquiera del *SortedSet*). Es decir si modificamos el conjunto original los valores de la vista quedan actualizados a aquellos del conjunto original que cumplan el invariante. Igualmente ocurre si modificamos la vista pero a esta no podemos añadir valores que no cumplan el invariante. Eso quiere decir que los métodos *add* y *addAll,* de la vista resultante, pueden disparar la excepción *IllegalArgumentException* si se intenta añadir un elemento que no esté dentro del rango especificado por el invariante.

Si no se cumplen las precondiciones de los métodos *first* y *last* se dispara la excepción *NoSuchElementException.*

Posibles inconsistencias del contrato *SortedSet* en el API de Java

Como hemos visto en el contrato de *SortedSet* incluye dos nuevos invariantes. El primero que todos los elementos puedan ser comparados entre sí mediante el orden del tipo. Eso quiere decir que cuando se intenten comparar no se dispare una excepción. El segundo es que noe puede haber dos elementos iguales con la igualdad definida por el orden (es decir *compare(e1,e2) == 0*). Como *SortedSet* es un subtipo de *Set* todos los invariantes de éste deberían estar vigentes y así ocurre si el orden del tipo es compatible con el igual. Es decir si:

```
a.equals(b) == (comparator().compare(a, b) == 0)
```

Tabla 10. Restricción de compatibilidad con igual

Pero si la igualdad no es compatible con el orden entonces *SortedSet* no sea estrictamente un subtipo de *Set* puesto que no se respeta su invariante. El invariante añadido en el *SortedSet* es redundante con el que tenía el *Set* solamente si el orden es consistente con la igualdad. Hay dos tipos de posibles inconsistencias entre la igualdad y el orden. En efecto según la especificación si tenemos dos objetos de tipo *T* que cumplan:

```
a.equals(b) && comparator().compare(a, b) != 0
```

Tabla 11. Inconsistencia de tipo 1

Llamaremos inconsistencia de **tipo uno** a este tipo de inconsistencia entre el orden y la igualdad. En este caso ambos *a* y *b* pueden estar en el *SortedSet*. O lo que es lo mismo si se añade *a* y luego *b* el segundo *add* devolverá *true*. Es decir el *SortedSet* no respeta el invariante del tipo *Set* salvo que el orden sea compatible con el igual. Otra inconsistencia se presenta para el caso:

```
!a.equals(b) && comparator().compare(a, b) == 0
```

Tabla 12. Inconsistencia de tipo 2

Llamaremos inconsistencia de **tipo dos** a este tipo de inconsistencia entre el orden y la igualdad. En ese caso al añadir a y luego b a un *SortedSet* ambos deberían estar en el conjunto, según el invariante de Set pero no lo están porque devuelven 0 al ser comparados.

Es una inconsistencia en la definición del API que produce frecuentes problemas. Por lo tanto lo recomendable es diseñar el orden de forma consistente con la igualdad. Esto puede ser conseguido siempre que el tipo tenga un orden natural consistente conla igualdad. En efecto dado un orden *ord* (de tipo Ordering en general) sobre *E* entonces el orden siguiente Es compatible con la igualdad.

```
Ordering<E> ord = ...;
Ordering<E> orc = ord.compound(Ordering<E>.natural());
```

Notación para Conjuntos Ordenados

Para los conjuntos ordenados están disponibles las operaciones de conjuntos y algunas nuevas como hemos visto. Dado un conjunto ordenado *a* las nuevas operaciones las representaremos de forma sintética de la forma:

Método	*Operación Equivalente*	*Significado*
a.first()	a^f	Primer Elemento
a.last()	a^l	Último Elemento
a.head(e)	$a^h(e)$	Cabeza del conjunto
a.tail(e)	$a^t(e)$	Cola del conjunto
a.subSet(e1,e2)	$a^s(e_1, e_2)$	Subconjunto

Tabla 15. Notación para conjuntos ordenados

Los conjuntos ordenados los representaremos por ss, ss_1, ss_2, etc. Esta notación, con el significado adecuado en el resto de tipos.

3.5 El tipo NavigableSet

El tipo NavigableSet<E> extiende el tipo SortedSet<E>. No añade nuevos invariantes per sí algunas operaciones nuevas y otra que tienen semántica similar a la del SortedSet<E> pero más parámetros.

NavigableSet<E> extends SortedSet<E>				
Invariante				
Métodos	Devuelve	O/M	Pre.	Post.
`NavigableSet<E> headSet(E toElement, boolean in)`	`Devuelve una vista con invariante e < toElement o e <= toElement según in`	`O`	`true`	
`NavigableSet<E> tailSet(E fromElement, boolean in)`	`Devuelve una vista con invariante fromElement <= e o fronElement < e según in`	`O`	`true`	
`NavigableSet<E> subSet(E fromElement, boolean fromIn, E toElement, boolean toIn)`	`Devuelve una vista con el invariante fromElement < e < toElement incluyendo lo límites según fromIn y toIn`	`O`	`true`	
`E pollFirst()`	`El primer elemento o null`	`M`	`true`	`!contains(first()@pre)`
`E pollLast()`	`El ultimo elemento o null`	`M`	`true`	`!contains(last()@pre)`
`E ceiling(E e)`	`El menor` *el* `en` *this* `tal` *el* `>= e o null`	`O`	`true`	
`E heigher(E e)`	`El menor` *el* `en` *this* `tal` *el* `> e o null`	`O`	`true`	
`E floor(E e)`	`El mayor` *el* `en` *this* `tal` *el* `<= e o null`	`O`	`true`	
`E lower(E e)`	`El mayor` *el* `en` *this* `tal` *el* `< e o null`	`O`	`true`	

3.6 Tipo MultiSet. Multiconjuntos

El tipo *Multiconjunto* es una colección donde se permiten elementos repetidos y como los conjuntos no impiorte el orden de lso mismos. No se ofrece en el API de Java. El tipo ofrece nuevos métodos observadores y modificadores y refina la semántica de los métodos heredados de *Collection*.

Métodos	Devuelve	O/M	Pre.	Post.
`int size()`	Número de elementos	O	true	
`boolean isEmpty()`	`size() > 0`	O	true	
`boolean contains(Object elem)`	`count(elem) > 0`	O	true	
`int count(E elem)`	Número de veces que contiene a elem	O	true	
`boolean equals(Object obj)`	Verdadero si this y obj tienen los mismos elementos y el mismo número de copias de cada uno	O	true	
`int setCount(E elem, int nc)`	`count(elem)@pre`	M		`count(elem) == nc`
`boolean setCount(E elem, int old, int nc)`	Verdadero si this ha cambiado	M	`nc, old >=0`	`count(elem) == (count(elem))@pre == old ? nc : count(elem))@pre)`
`boolean add(E elem)`	Verdadero si this ha cambiado	M	true	`count(elem) == (count(elem))@pre +1`
`int add(T elem, int nc)`	`count(elem)@pre`	M	`nc >= 0`	`count(elem) == (count(elem))@pre + nc`
`boolean remove(E elem)`	Verdadero si this ha cambiado	M	true	`count(elem) == max(0,(count(elem))@pre - 1)`
`int remove(E elem, int nc)`	`count(elem)@pre`	M	`nc >= 0`	`count(elem) == max(0,(count(elem))@pre - nc)`
`Set<E> elementSet()`	Vista formada por el conjunto de elementos	O	true	
`Set<Multiset.Entry<E>> entrySet()`	Vista formada por el conjunto pares elemento numero de copias	O	true	
`boolean containsAll( Collection<? super T> c)`	`c.contains(e) => contains(e)`	O	true	
`boolean addAll(Collection <? super T> c)`	Verdadero si this ha cambiado	M	true	`count(e ) == count(e)@pre+c.count(e)`
`boolean retainAll(Collection <? super T> c)`	Verdadero si this ha cambiado	M	true	`count(e ) == (e ∈ c ? count(e )@pre : 0)`

Si no se respetan las precondiciones de los métodos se dispara la excepción *IllegalArgumentException.*

Notación para Multiconjuntos

Método	*Operación Equivalente*	*Significado*
a.add(e)	*a+e*	Añadir elemento
a.add(e,n)	*a+ nxe*	Añadir copias
a.remove(e)	*a-e*	Eliminar elemento
a.remove(e,n)	*a-nxe*	Eliminar copias
a.addAll(b)	$a = a + b$	Unión
a.removeAll(b)	$a = a - b$	Diferencia
a.retainAll(b)	$a = a \mid b$	Filtro (Intersección en Set)
a.containsAll(b)	$b \leq a$	Inclusión
a.isEmpty( )	$a == \emptyset$	Está vacío?
a.size()	$\lvert a \rvert$	Cardinal
a.contains(e)	$e \in a$	Pertenencia
a.count(e)	*a(e)*	Número de copias
a.setCount(e,n)	*a(e) = n*	Cambiar el número de copias

3.7 Factorías de Conjuntos, Conjuntos Ordenados, Conjuntos Navegables y Multiset

Para crear conjuntos y conjuntos ordenados podemos usar las clases que implementan los tipos porporcionadas en el API de Java. Además podemos usar la clase de factoría *Sets* proporcionada por *Guava.* Esta clase nos permite crear conjuntos con distintas implementaciones y a partir de diferentes datos de partida. También nos permite crear conjuntos de tipos enumerados con una implementación muy eficiente y copias inmutables de un conjunto. También proporciona impletaciones funcionales de operaciones sobre conjuntos. Veamos algunos métodos de esa factoría.

- *static <E> HashSet<E> newHashSet()*
- *static <E> HashSet<E> newHashSet(E… elements)*
- *static <E> HashSet<E> newHashSet(Iterable<? extends E> elements)*
- *static <E> LinKedHashSet<E> newLinkedHashSet()*
- *static <E> LinkedHashSet<E> newLinkedHashSet(Iterable<? extends E> elements)*
- *static <E> TreeSet<E> newTreeSet()*
- *static <E> TreeSet<E> newTreeSet* (*Comparator<? super E> comparator)*
- *static <E> TreeSet<E> newTreeSet(Iterable<? extends E> elements)*
- *static <E extends Enum<E>>*
  ImmutableSet<E> immutableEnumSet(E anElement, E... otherElements)
- *static <E extends Enum<E>>*
  ImmutableSet<E> immutableEnumSet(Iterable<E> elements)
- *static <E extends Enum<E>>*
  EnumSet<E> complementOf(Collection<E> elements)
- *static <E extends Enum<E>>*
  EnumSet<E> complementOf(Collection<E> elements, Class<E> type)
- *static <E> Set.SetView<E> difference(Set<E> set1, Set<?> set2):*

- *static <E> Set.SetView<E> symmetricDifference(Set<E> set1, Set<?> set2):*
- *static <E> Set.SetView<E> union(Set<E> set1, Set<?> set2):*
- *static <E> Set.SetView<E> intersection(Set<E> set1, Set<?> set2):*
- *static <E> Set<E> powerSet(Set<E> set)*
- *static <E> Set<E> filter(Set<E> set, Predicate<? super E> predicate)*
- *static <B> Set<List<B>> cartesianProduct(List<? extends Set<? extends B>> sets)*

Los métodos anteriores constuyen objetos de tipo *Set<T>* y *SortedSet<T>* mediante diferentes implementaciones. El tipo *Set<T>* puede ser implementado por los tipos *HashSet<E>, LinkedHashSet<E> y TreeSet<E>.* Los dos primeros no necesitan un orden para cosntruir el conjunto. El último si usa el orden natural o un orden que se proporciona como parámetro. Este último tiene el invariante adicional de no mantener dos objetos que al ser comparados con el orden devuelva 0. Se pueden crear, también, conjuntos de tipos enumerados y conjuntos inmutables. La elección de una implementación u otra vendrá dada por las exigencias sobre la complejidad de cada una de las operaciones. Estos detalles los veremos más adelante.

El tipo *SortedSet<E>* y el *NavigableSet<E>* se implementa con el tipo *TreeSet<E>.*

Se proporcionan, también, métodos funcionales para costruir la unión, intersección, diferencia, diferencia simétrica, subconjunto filtrado, conjunto potencia o el producto cartesiano.

Los multiconjuntos se pueden crear con métodos estáticos de las clases *HashMultiset, LinkedHashMultiset, TreeMultiset, InmutableMultiset.*

- *static <E> HashMultisetset<E> create()*
- *static <E> HashMultiset<E> create(Iterable<? extends E> elements)*
- *static <E> LinKedHashMultiset<E>create()*
- *static <E> LinkedHashMultiset<E>create(Iterable<? extends E> elements)*
- *static <E> TreeMultiset<E> create()*
- *static <E> TreeMultiset<E> create(Comparator<? super E> comparator)*
- *static <E> TreeMultiset<E> create(Iterable<? extends E> elements)*
- *static <E> InmutableMultiset<E> copyOf(E[] elememts)*
- *static <E> InmutableMultiset<E> copyOf(Iterable<? extends E> elements)*

Los métodos anteriores constuyen objetos de tipo Multi*set<T>* mediante diferentes implementaciones. El tipo *Multiset<T>* puede ser implementado por los tipos *HashMultiset<E>, LinkedHashMultiset<E> y TreeMultiset<E>.* Los dos primeros no necesitan un orden para construir el multiconjunto. El último si usa el orden natural o un orden que se proporciona como parámetro. Este último tiene el invariante adicional de no mantener dos objetos que al ser comparados con el orden devuelvan 0. La elección de una implementación u otra vendrá dada por las exigencias sobre la complejidad de cada una de las operaciones. Estos detalles los veremos más adelante.

3.8 Listas

Una lista es una colección en la que los elementos pueden ser accedidos mediante un índice de tipo entero que representa su posición en la lista. El índice que sirve para acceder a los elementos puede tomar valores desde *0* a *size() -1*, siendo size() el número de elementos de la lista.

El tipo lista enriquece a la interfaz de Collection añadiendo una serie de métodos tal y como puede observarse en la figura 8.

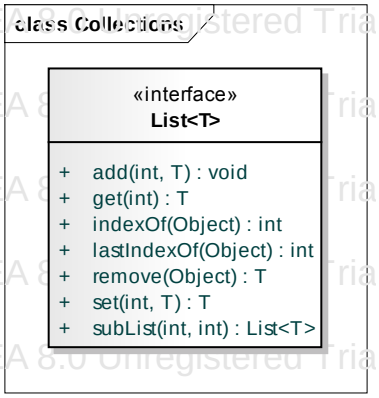


Figura 8. Interfaz de List

En el tipo *List<E>* las nuevas operaciones observadoras son: *get, indexOf, lastIndexOf, subList*. Las nuevas operaciones modificadoras son: *set, add, remove*. La funcionalidad de cada uno de estos métodos es la que se muestra en la tabla junto a la funcionalidad añadida a los métodos de Collection<E>.

List<E> extends Collection<E>				
Invariante				
Métodos	Devuelve	O/M	Pre.	Post.
`boolean equals(Object c)`	Verdadero si *this* tiene el mismo tamaño que *c* y en cada celda contiene un elemento igual al que contiene *c*	O	true	
`int size()`	Número de elementos de *this*	O	true	
`boolean contains(Object e)`	Verdadero si *this* contiene a e	O	true	
`boolean isEmpty()`	`size() > 0`	O	true	
`void clear()`		M	true	*isEmpty()*
`boolean add(E e)`	Verdadero si *this* ha cambiado	M	true	*get(size()-1).equals(e)*
`void add(int index, E e)`		M	`0 <= index < size()`	*get(index).equals(e).* Todos los elementos que estaban en la posición i con i > index están ahora en

				i+1.
E get(int index)	El elemento que había en la posición index.	O	0 <= index < size()	
E set(int index, E e)	El elemento que había en la posición index.	M	0 <= index < size()	*get(index).equals(e)*
boolean remove(Object e)	Verdadero si *this* ha cambiado	M	true	La primera ocurrencia de e ha sido eliminada. Los elementos a su derecha ocupan una posición menos.
E remove(int index)	El elemento que estaba en la posición index.	M	0 <= index < size()	Todos los elementos que estaban en la posición i con i > index están ahora en i-1.
int indexOf(Object e)	La primera posición de un elemento igual a e o -1 si no hay	O	true	
int lastIndexOf(Object e)	La última de un elemento igual a e o -1 si no hay	O	0 <= index < size()	
List<E> subList(int fromIndex, index toIndex)	Devuelve una vista de la porción de lista delimitada por las posiciones fromIndex <= i < toIndex	O	0 <= fromIndex ,toIndex < size()	
boolean addAll(Collection<? super E> c)	Verdadero si *this* ha cambiado	M	true	Añade a this (con add(e)) cada uno de los elementos en c
boolean removeAll(Collection<? super E> c)	Verdadero si *this* ha cambiado	M	true	Elimina de this (con remove(e)) cada uno de los elementos en c
boolean containsAll(Collection<? super E> c)	Verdadero si *this* contiene al menos una instancia de cada elemento en c	O	true	

Métodos de List<E>

Si no se cumplen las precondiciones de los métodos que tienen se dispara la excepción *IndexOutOfBoundsException.*

Notación para listas

Para las listas están disponibles las operaciones de conjuntos extendidas que usaremos de la misma forma y algunas nuevas como hemos visto. Dadas dos listas *a* y *b*, un elemento *e* y un entero *i*, las operaciones las representaremos de forma sintética:

Método	*Operación Equivalente*	*Significado*
a.add(e)	a = a+ e	Añadir un elemento por la derecha
a.set(i,e)	a(i) = e	Modifica el elemento i por e
a.get(i)	a(i)	Obtener el elemento en posición i
a.add(0,e)	e+a	Añadir e por la izquierda
a.addAll(b)	a = a + b	Unión

a.removeAll(b)	a = a – b	Diferencia
a.retainAll(b)	a = a | b	Filtrado
a.containsAll(b)	b ≤ a	Inclusión
a.isEmpty( )	a == ∅	Está vacío?
a.size()	|a|	Cardinal
a.contains(e)	$e \in a$	Pertenencia
a.first()	a^f	Primer Elemento
a.last()	a^l	Último Elemento
a.subList(i1,i2)	$a^s(i_1, i_2)$	SubLista

Tabla 21. Notación para las listas

Las listas las representaremos por l, l_1, l_2, l_3, etc.

3.9 Factorías de Listas

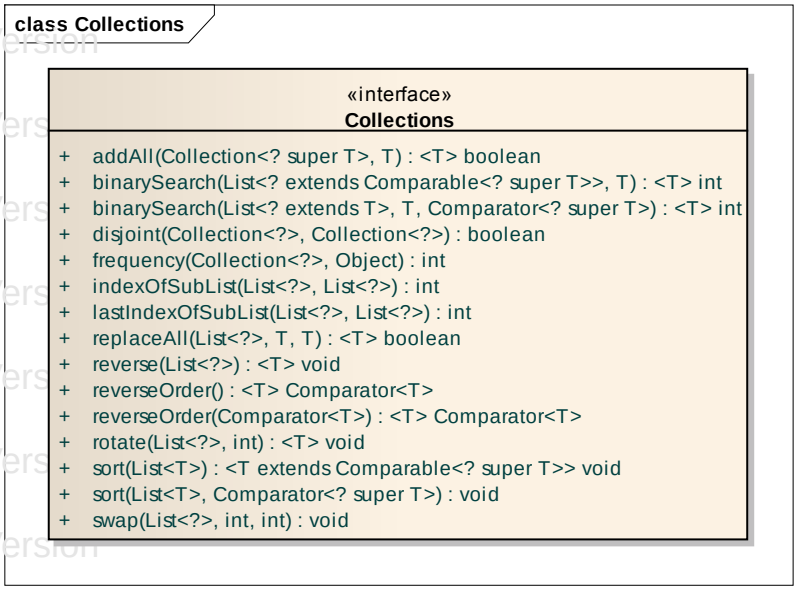
Para crear listas podemos usar las clases que implementan los tipos porporcionadas en el API de Java. Además podemos usar la clase de factoría *Lists* proporcionada por *Guava.* Esta clase nos permite crear listas con distintas implementaciones y a partir de diferentes datos de partida. Veamos algunos métodos de esa factoría.

- *static <E> List<E> asList (E first, E[] rest)*
- *static <E> ArrayList<E> newArrayList()*
- *static <E> ArrayList<E> newArrayList(E… elements)*
- *static <E> LinkedList<E> newLinkedList()*
- *static <E> LinkedList<E> newLinkedList(E… elements)*
- *static <E> List<E> reverse(List<E> list)*
- *static <F,T> List<T> transform(List<F> fromList, Function<? super F,? extends T> function)*

Los métodos anteriores constuyen objetos de tipo *List<T>* mediante diferentes implementaciones. El tipo *List<T>* puede ser implementado por los tipos *ArrayList<E> y LinkedList<E>.* La elección de una implementación u otra vendrá dada por las exigencias sobre la complejidad de cada una de las operaciones. Estos detalles los veremos más adelante. También se proporcionan métodos para invertir una lista o transformarla.

3.10 Funciones sobre Colecciones

La clase *Collections* de Java, ofrece, además, un conjunto de métodos para trabajar con colecciones. Algunos de ellos son:



Interfaz de Collections

La funcionalidad ofrecida por los diversos métodos de Collections es:

Métodos	**Descripción**
`addAll(Collection<? super T> c, T... elements)`	Añade a la colección los elementos indicados en elements.
`binarySearch(List<? extends Comparable<? super T>> list,T key)`	Devuelve la posición del objeto key en la lista ordenada según su orden natural o -1 si no lo encuentra.
`binarySearch(List<? extends T> list,T key, Comparator<? super T> c)`	Devuelve la posición del objeto key en la lista ordenada según el orden c o -1 si no lo encuentra.
`disjoint(Collection<?> c1, Collection<?> c2)`	Devuelve verdadero si no hay elementos comunes a las dos colecciones.
`frequency(Collection<?> c, Object o)`	Devuelve el número de elementos iguales a o en la colección c.
`indexOfSubList(List<?> source, List<?> target)`	Devuelve la primera posición de la primera ocurrencia de target dentro de source o -1 si no encuentra ninguna.
`lastIndexOfSubList(List<?> source, List<?> target)`	Devuelve la primera posición de la última ocurrencia de target dentro de source o -1 si no encuentra ninguna.
`replaceAll(List<T> list, T oldVal, T newVal)`	Reemplaza en list el objeto oldVal por newVal.
`reverse(List<?> list)`	Devuelve la lista invertida
`reverseOrder()`	Devuelve un *Comparator* que representa el orden inverso del orden natural de un tipo dado.
`rotate(List<?> list, int distance)`	Rota la lista en un número *distance* de casillas hacia la derecha si es positivo o hacia la izquierda si es negativo.

`sort(List<T> list)`	Ordena la lista según el orden natural del tipo.

Propiedades específicas de Collections

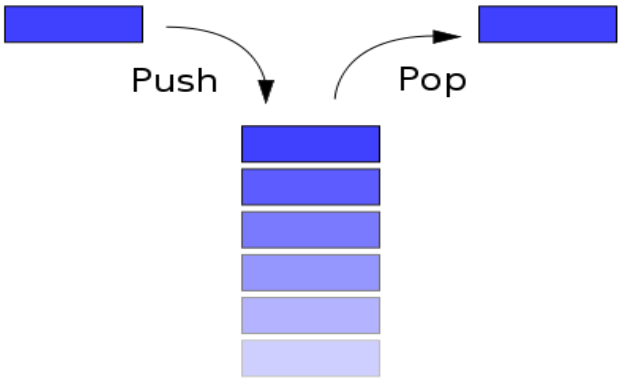
4. Otras colecciones Lineales

Las colecciones lineales son secuencias de elementos con una determina disciplina para añadir elemento y retirarlos. Junto con la lista, que ya se ha visto en capítulos anteriores, el API de java ofrece *Stack, Queue, PriorityQueue.* Todos son subtipos de *Collection<T>*. Para escribir el contrato de cada uno de los tipos usaremos un modelo para cada tipo. Como modelo para pilas y colas usaremos una lista y para las colas de prioridad un conjunto ordenado. Como en las colecciones los métodos observadores son: *size, isEmpty*. Los métodos modificadores: *add, remove*. Estos métodos añaden semántica a la que ya heredan de *Collection<E> y* añaden otros métodos nuevos.

Para cada tipo incluimos los métodos nuevos (con respecto a *Collection<E>)* y los que añaden semántica nueva.

Tipo Stack

La Pila *(Stack)* es un tipo de datos con la disciplina último en entrar primero en salir. Es lo que se denomina disciplina LIFO. La pila también la podemos pensar como una secuencia de elementos donde cada vez que se añade un elemento se añade al principio y al eliminar uno se elimina el del principio.



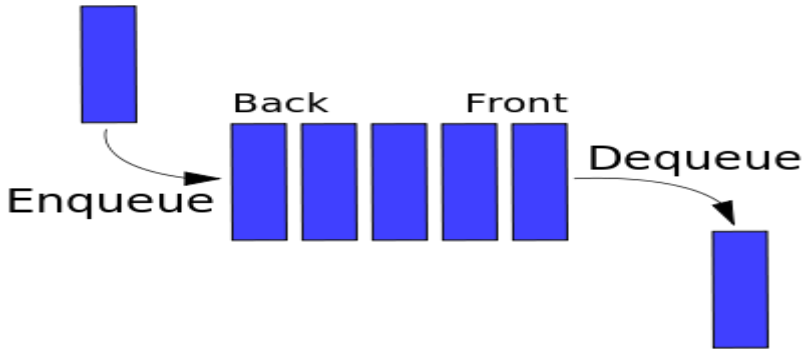
Stack<E> extends Collection<E>	
Métodos	**Descripción**
`boolean add(E e)`	Añadir un elemento a la pila. Dispara *IllegalStateException* si no hay sitio en la cola para ubicar el elemento.
`E push(E e)`	Añadir un elemento a la pila (similar a *add*).
`E peek()`	Devuelve el elemento que se añadió en último lugar con el método *add* o *push*. Dispara *EmptyStackException* si la pila está vacía
`E pop()`	Devuelve el elemento que se añadió en último lugar con el método *add*

	o *push* y lo elimina. Dispara *EmptyStackException* si la pila está vacía

Para comprender la semántica de los métodos de una pila podemos usar como modelo una lista. En ese caso el método *add* y *push* son similares a *add* de la lista (es decir añadir al principio de la lista). El método *peek* es equivalente a devolver el primer elemento de la lista. Por último *pop()* es equivalente a obtener el primer elemento de la lista, eliminarlo de la misma y devolverlo.

Tipo Queue

La Cola *(Queue)* es un tipo de datos con la disciplina primero en entrar primero en salir. Es lo que se denomina disciplina FIFO. La cola podemos pensarla como una secuencia de elementos donde cada vez que se añade un elemento se añade al principio y al eliminar uno se elimina el último.



Queue<E> extends Collection<E>	
Métodos	**Descripción**
`boolean add(E e)`	Añadir un elemento a la cola. Dispara *IllegalStateException* si no hay sitio para ubicar el elemento.
`boolean offer(E e)`	Añadir un elemento a la cola (similar a *add*) pero si no hay espacio disponible simplemente devuelve *false* y no dispara excepción.
`E element()`	Devuelve el elemento que se añadió en primer lugar con el método *add* o *offer*. Dispara *NoSuchElementException* si la cola está vacía
`E peek()`	Devuelve el elemento que se añadió en primer lugar con el método *add* o *push*. Si la cola está vacía devuelve *null*. Similar a *element().*
`E remove()`	Devuelve el elemento que se añadió en primer lugar con el método *add* o *push* y lo elimina. Dispara *NoSuchElementException* si la cola está vacía
`E poll()`	Devuelve el elemento que se añadió en primer lugar con el método *add* o *push* y lo elimina. Devuelve *null* si la cola está vacía

Para comprender la semántica de los métodos de una pila podemos usar como modelo una lista. En ese caso el método *add* y *offer* son similares a *add* de la lista (es decir añadir al principio de la lista). El método *peek* y *element* son equivalentes a devolver el último elemento

de la lista. Por último *remove y poll* son equivalentes a obtener el último elemento de la lista, eliminarlo de la misma y devolverlo.

Hay otros tipos de colas como *Deque<E>* (cola doble), que permite añadir y eliminar tanto al principio como al final.

Tipo PriorityQueue

La Cola de Prioridad *(PriorityQueue)* es un tipo de datos con la disciplina distinta a las anteriores. La cola de prioridad podemos pensarla una secuencia de elementos que se mantienen ordenados con respecto a un orden. Existe, por lo tanto, un primer elemento. Cuando se añade un elemento se coloca en el lugar que le corresponda según el orden y al eliminar uno se elimina el más pequeño según el orden. Los métodos tienen la misma signatura que los de la Cola. La funcionalidad de los mismos en la Cola de prioridad:

Queue<E> extends Collection<E>	
Métodos	**Descripción**
`boolean add(E e)`	Añadir un elemento a la cola. Dispara *IllegalStateException* si no hay sitio para ubicar el elemento.
`boolean offer(E e)`	Añadir un elemento a la cola (similar a *add*) pero si no hay espacio disponible simplemente devuelve *false* y no dispara excepción.
`E element()`	Devuelve el elemento el primer elemento de la Cola. Dispara *NoSuchElementException* si la pila está vacía
`E peek()`	Devuelve el primer elemento de la cola. Si está vacía devuelve *null*. Similar a *element().*
`E remove()`	Devuelve el primer elemento y lo elimina. Dispara *NoSuchElementException* si la cola está vacía
`E poll()`	Devuelve el primer elemento y lo elimina. Devuelve *null* si la cola está vacía

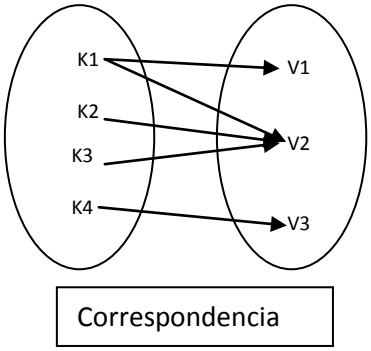
La Cola de Prioridad Modificable es un subtipo de una cola de prioridad que, junto con los métodos de la cola de prioridad, añade un nuevo método que permite modificar las propiedades de un elemento de la cola y reordenar la misma posteriormente.

Las pilas, colas y colas de prioridad pueden añadir invariantes que restrinjan el número de elementos. En este caso se denominan **pilas, colas o colas de prioridad acotadas**.

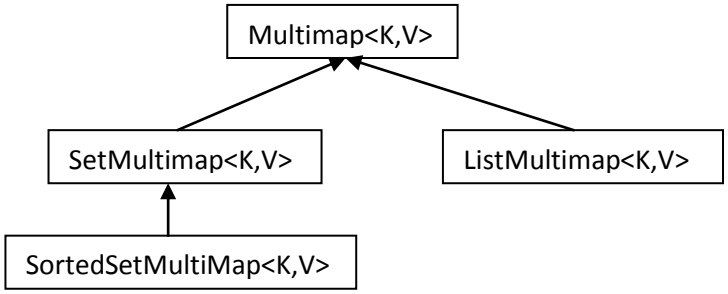
5. Correspondencias y Funciones

Una **correspondencia** es un conjunto de pares de valores. La primera componente del par se toma de un cojunto que llamaremos dominio o conjunto de las claves y la segunda componente de de otro conjunto que llamaremos imagen o conjunto de valores. Las

coorespondencias, a su vez, pueden ser en particular **aplicaciones** o **funciones** si tienen la restricción de que a cada clave le corresponden una sola imagen. Las funciones pueden ser biunívocas si además a cada imagen le corresponde una sola clave. Es muy importante conocer los tipos que modelan las ideas anteriores en el API de Java ampliado con Guava. Gráficamente una correspondencia la podemos respresentar como:



Las **correspondencias** generales las modelaremos mediante el tipo *Multimap<K,V>* donde *K* es el tipo del conjunto de las claves y *V* el tipo de los valores. A los objetos de este tipo los llamaremos correspondencias o **multifunciones**.

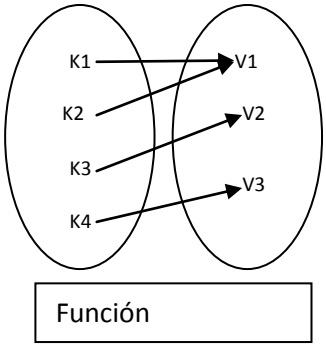


Jerarquía de interfaces de Multifunciones.

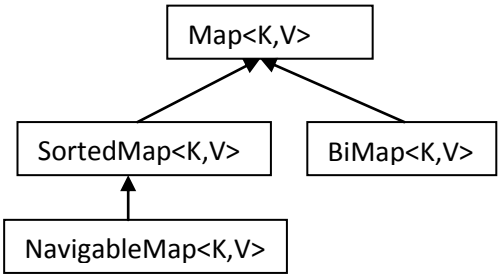
Los subtipos son coorespondencias que permiten organizar las imágenes de una clave mediante una lista, un cojunto o un cojunto ordenado.

Una **función**, tal como la usaremos aquí (funciones con dominio finito), es una coorespondencia a la que le imponemos una restricción: *dos pares distintos no pueden tener la misma clave*. La restricción anterior implica que cada clave tiene asociado un único valor.

El tipo que representa una función lo denominamos *Map<K,V>*. Donde *K* es el tipo de las claves y *V* el de los valores. Abajo vemos el conjunto de tipos relacionados con el tipo *Map*<K,V> y una imagen gráfica:



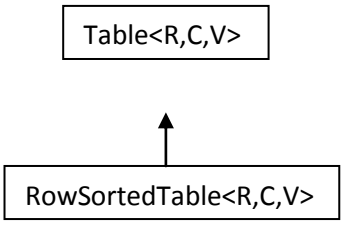
Si para cada clave se le asocia una sola imagen entonces la fuución es biyectiva.



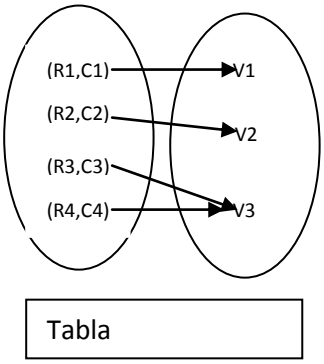
Jerarquía de interfaces de Funciones

Los subtipos representan **funciones biyectivas** (*BiMap<K,V>*), funciones cuyas claves forman un cojunto ordenado (*SortedMap<K,V>*) o aquellas cuyas claves forman un conjunto navegable(*NavigableMap<K,V>).*

Junto a las anteriotres también usaremos el tipo *Table<R,C,V>* que es una función que toma un par de claves que ahcen el papel de filas y columnas.



El subtipo es una tabla ordenada según sus filas. Una idea gráfica de los objetos de tipo *Table<R,C,V>* es de la forma:



5.1 Tipo Map

La interfaz que define el tipo *Map* junto con tipo interno *Map.Entry* es.

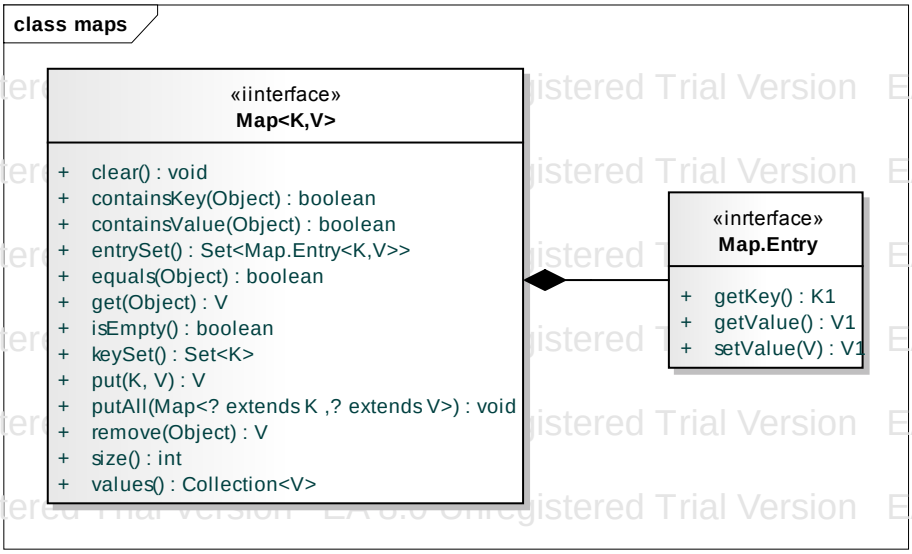


Figura 13. Interfaz de Map

La funcionalidad ofrecida por cada uno de los métodos es:

Map<K,V>			
Invariante			
Una clave tiene una y solo una imagen			
Métodos	**Devuelve**	**O/M**	**Post.**
`void clear()`		M	*isEmpty()*
`boolean containsKey(Object key)`	`keySet().contains(key)`	O	
`boolean containsValue(Object value)`	`values().contains(value)`	O	
`Set<Map.Entry<K,V>> entrySet()`	Conjunto de pares que definen la función. Vista con las operaciones add y addAll restringidas.	0	
`boolean equals(Object m)`	`entrySet().equals(m.entrySet())`	O	
`V get(K key)`	`get(key)@pre`	O	
`boolean isEmpty()`	`keySet().isEmpty()`	O	
`Set<K> keySet()`	Conjunto de las claves. Vista con las operaciones add y addAll restringidas.	O	
`V put(K key, V value)`	`get(key)@pre`	M	`get(key).equals(value)`
`void putAll(Map<? extends K,? extends V> m)`		M	Añade a *this* (con el método put) cada uno de pares los pares de m.
`V remove(Object key)`	`get(key)@pre`	M	`!containsKey(key)`

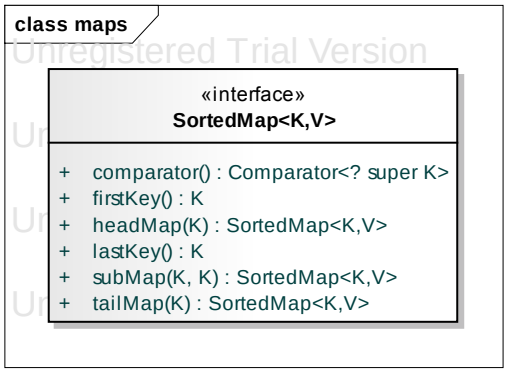
`int size()`	`keySet().size()`		
`Collection<V> values()`	`Colección de los valores. Vista con las operaciones add y addAll restringidas.`	`O`	

Propiedades específicas de Map

Todos los métodos tienen true como precondición. El conjunto de métodos observadores es: *get, containsKey, containsValue, size, isEmpty, keySet, values.* Los demás son modificadores.

5.2 Tipo SortedMap

Es el tipo de las funciones que ofrecen un conjunto de métodos relacionados a una vista del conjunto de claves como conjunto ordenado. Los nuevos métodos ofrecidos tienen una relación directa con los métodos ofrecidos por el tipo *SortedSet*. Son los siguientes:



Interfaz de SortedMap

La funcionalidad de cada uno de estos métodos es la que se muestra en la tabla 21.

SortedMap<K,V> extends Map<K,V>		
Métodos	**Devuelve**	**Pre.**
`Comparator<? super K> comparator()`	`Devuelve el comparator asociado al SortedMap, o null si se usa el orden natural`	`true`
`K firstKey()`	`Devuelve primera clave`	`!isEmpty()`
`Map<K,V> headMap(K tokey)`	`Devuelve una vista con el invariante: k < toKey`	`true`
`K lastKey()`	`Devuelve la última clave`	`!isEmpty()`
`Map<K,V> subMap(K fromKey, K toKey)`	`Devuelve una vista con el invariante: fromKey <= k < toKey`	`true`
`Map<K,V> tailMap(K fromKey)`	`Devuelve una vista con el invariante: k >= fromKey`	`true`

Propiedades específicas de SortedMap respecto a Map.

Si no se cumplen las precondiciones de los métodos *firstKey* y *lastKey* se dispara la excepción *NoSuchElementException.*

5.3 Funciones Navegables

El tipo *NavigableMap<K,V>* extiende el tipo *SortedMap<K,V>.* No añade nuevos invariantes per sí algunas operaciones nuevas y otra que tienen semántica similar a la del *SortedMap<K,V>* pero más parámetros.

NavigableMap<K,V> extends SortedMap<K,V>				
Invariante				
Métodos	Devuelve	O/M	Pre.	Post.
`NavigableMap<K,V> headMap(K toKey, boolean in)`	Devuelve una vista con invariante k < toKey o k <= toKey según in	O	true	
`NavigableSet<K,V> tailMap(K fromKey, boolean in)`	Devuelve una vista con invariante fromKey <= k o fronKey < e según in	O	true	
`NavigableMap<K,V> subMap(K fromKey, boolean fromIn, K toKey, boolean toIn)`	Devuelve una vista con el invariante fromKey < k < toKey incluyendo los límites según fromIn y toIn	O	true	
`NavigableSet<K> navigableKeySet()`	Devuelve una vista del conjunto de las claves que no soporta la operación *addAll*.	O	true	
`Map.Entry<K,V> pollFirstEntry()`	El primer par clave valor o null	M	true	`!contains(firstKey()@pre)`
`Map.Entry<K,V> pollLastEntry()`	El ultimo par clave valor o null	M	true	`!contains(lastKey()@pre)`
`K ceilingKey(K key)`	La menor *K* en *this* tal *k >= key* o null	O	true	
`K heigherKey(K key)`	La menor *k* en *this* tal *k > key* o null	O	true	
`K floorKey(K key)`	La mayor *k* en *this* tal *k <= key* o null	O	true	
`K lower(K key)`	La mayor *k* en *this* tal *k < key* o null	O	true	

5.4 El tipo función Biyectiva

Es un subtipo del tipo *Map<K,V>* que añade el invariante adicional de que cada imagen tiene una sola clave. Es decir cada clave tiene una sola imagen y cada imagen un solo origen. Añade, además algunas operaciones.

BiMap<K,V> extends Map<K,V>				
Invariante				
Dadas dos valores v1, v2 distintos sus claves también son distintas.				
Métodos	Devuelve	O/M	Pre.	Post.
`V put(K key, V value)`	`get(key)@pre`	M	`!get(key)@pre.equals(value)`	`get(key).equals(value)`
`V forcePut(K key, V value)`	`get(key)@pre`	M	true	`(para cualquier k) get(key).equals(value) && !containsKey(k) &&`

				`!key.equals(key)`
`BiMap<V,K> inverse()`	`Devuelve una vista con una función biyectiva que es la inversa de this.`	`O`	`true`	
`Set<V> values()`				
`void putAll(Map<? extends K,? extends V> map)`		`M`	`map no puede contener imágenes para las claves en this`	`Contiene los pares clave valor de map`

Notación para funciones, funciones ordenadas y funciones biyectivas

Dadas las funciones *a* y *b*, el elemento e, la clave *k* y el par *(k,e)* las operaciones sobre funciones la representaremos por:

Método	*Operación Equivalente*	*Significado*
a.get(k)	a(k)	Obtener la imagen de k
a.put(k,e)	a = a+ (k,e)	Modificar la imagen de k
a.forcePut(k,e)	a = a+ (k,e)	Modificar la imagen de k
a.remove(k)	a = a – k	Eliminar la clave k
a.putAll(b)	a = a + b	Combinación de funciones
a.isEmpty( )	a == ∅	Está vacía?
a.size()	|a|	Cardinal de la función
a.containsKey(k)	$k \in a$	Pertenencia de la clave k
a.containsValue(e)	$e \in a$	Pertenencia del valor
a.firstKey()	a^f	Primera Clave
a.lastKey()	a^l	Última Clave
a.keySet()	a^k	Conjunto de las claves
a.values()	a^v	Colección de los valores
a.head(e)	$a^h(e)$	Cabecera del Map
a.tail(e)	$a^t(e)$	Cola del Map
a.subMap(e1,e2)	$a^s(e_1, e_2)$	SubMap
a.inverse()	a^{-1}	Fucnión inversa

Propiedades específicas para funciones, funciones ordenadas y biyectivas.

5.5 Factorías de Funciones y Funciones Ordenadas

Para crear funciones y funciones ordenadas podemos usar las clases que implementan los tipos porporcionadas en el API de Java. Además podemos usar la clase de factoría *Maps* proporcionada por *Guava.* Esta clase nos permite crear funciones con distintas implementaciones y a partir de diferentes datos de partida. Veamos algunos métodos de esa factoría.

- *static <K,V> HashMap<K,V> newHashMap()*
- *static <K,V> HashMap<K,V> newHashMap(Map<? extends K,? extends V> map)*
- *static <K,V> LinKedHashMap<K,V> newLinkedHashMap()*

- *static <K,V> LinkedHashMap<K,V> newLinkedHashMap(Map<? extends K,? extends V> map)*
- *static <K,V> TreeMap<K,V> newTreeMap()*
- *static <K,V> TreeMap<K,V> newTreeMap(Comparator<C> comparator)*
- *static <K,V> TreeMap<K,V> newTreeMap(SortedMap<? extends K,? extends V> map)*
- *static <K extends Enum<K>, V> EnumMap<K,V> newEnumMap(Class<K> type)*
- *static <K extends Enum<K>, V> EnumMap<K,V> newEnumMap(Map<K,? extends V> map)*
- *static <K,V> Map<K,V> filterEntries(Map<K,V> unfiltered, Predicate<? super Map.Entry<K,V>> entryPredicate):*
- *static <K,V> Map<K,V> filterKeys(Map<K,V> unfiltered, Predicate<? super K> keyPredicate):*
- *static <K,V> Map<K,V> filterValues(Map<K,V> unfiltered, Predicate<? super V> valuePredicate):*
- *static <K,V1,V2> Map<K,V2> transformValues(Map<K,V1> fromMap, Function<? super V1,V2> function)*

Los métodos anteriores constuyen objetos de tipo *Map<K,V>* y *SortedMap<K,V>* mediante diferentes implementaciones. El tipo *Map<K,V>* puede ser implementado por los tipos *HashMap<K,V>, LinkedHashMap<K,V> y TreeMap<K,V>.* Los dos primeros no necesitan un orden sobre las claves para construir el conjunto. El último si usa el orden natural o un orden que se proporciona como parámetro. Este último tiene el invariante adicional de no mantener dos claves que al ser comparados con el orden devuelva 0. Se pueden crear, también, funciones de tipos enumerados. La elección de una implementación u otra vendrá dada por las exigencias sobre la complejidad de cada una de las operaciones. Estos detalles los veremos más adelante.

El tipo *SortedMap<K,V>* y el *NavigableMap<K,E>* se implementa mediante el tipo *TreeMap<K,V>.*

Se proporciona, además, métodos para filtrar las funciones por sus claves, sus valores o los pares que las componen y transformarlas.

Las funciones biyectivas se crean con los métodos estáticos de la clase *HashBiMap<K,V>:*

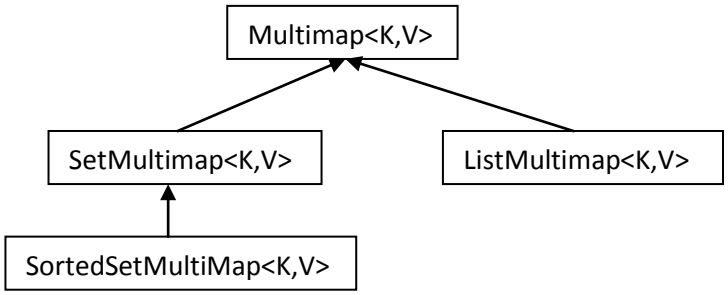
- *static <K,V> HashBiMap<K,V> create()*
- *static <K,V> HashBiMap<K,V> create(Map<? extends K,? extends V> map)*

También se pueden crear funciones biyectivas entre tipos enumerados con los métodos de la clase *EnumBiMap<K,V>:*

- *static <K extends Enum<K>,V extends Enum<V>>*
  EnumBiMap<K,V> EnumBiMap<K,V> create(Class<K> keyType, Class<K> keyValue)
- *static <K extends Enum<K>,V extends Enum<V>>*
  EnumBiMap<K,V> EnumBiMap<K,V> create(Map<K,V> map)

5.6 Multifunciones

El tipo *Multimap<K,V>* es una generalización del tipo *Map<K,V>* donde un clave puede tener más de una imagen. Tiene varios subtipos según que pueda ofrecervistas de todas las imágenes d ede unlave en forma de conjunto, conjunto oredenado o lista. Los métodos son muy similares a los de *Map<K,V>* pero algunos tienen una semántica diferente.



Jerarquía de interfaces de Multifunciones

Multimap<K,V>			
Invariante			
Métodos	**Devuelve**	**O/M**	**Post.**
`void clear()`		M	*isEmpty()*
`boolean containsKey(Object key)`	`keySet().contains(key)`	O	
`boolean containsValue(Object value)`	`values().contains(value)`	O	
`Set<Map.Entry<K,V>> entrySet()`	Conjunto de pares que definen la función. Vista con las operaciones add y addAll restringidas.	O	
`boolean equals(Object m)`	Dos Multimap son iguales cuando sus vistas com Maps son iguales	O	
`Collection<V> get(K key)`	Una vista de las imágenes de key	O	
`boolean isEmpty()`	`keySet().isEmpty()`	O	
`Multiset<K> keys()`	Multiconjunto de las claves.Cada clave se repite tantas veces como imágenes tenga	O	
`Set<K> keySet()`	Conjunto de las claves.	O	
`Map<K,Collection<V>> asMap()`	Devuelve una vista en forma de Map que asociada cada clave a sus imágenes.	O	
`boolean put(K key, V`	Si this cambia	M	`get(key).con`

value)			tains(value)
void putAll(Map<? extends K,? extends V> m)		M	Añade a *this* (con el método put) cada uno de pares los pares de m.
boolean remove(Object key)	Si this cambia	M	Elimina un par clave valor.
Collection<V> removeAll(Object key)	get(key)@pre	M	get(key).isEmpty()
int size()	keys().size()		
Collection<V> values()	Colección de los valores. Vista con las operaciones add y addAll restringidas.	O	

Propiedades específicas de Multimap

Los subtipos tienen una semántica similar pero ofrecen las colecciones formadas por las imágenes de una clave en forma de conjuntos, conjuntos ordenados o listas. Solo presentamos lo nuevos métodos relevantes para cada subtipo.

SetMultimap<K,V> extends Multimap<K,V>			
Invariante			
Métodos	**Devuelve**	**O/M**	**Post.**
Set<V> get(K key)	Una vista de las imágenes de key	O	

SortedSetMultimap<K,V> extends SetMultimap<K,V>			
Invariante			
Métodos	**Devuelve**	**O/M**	**Post.**
SotedSet<V> get(K key)	Una vista de las imágenes de key	O	

ListMultimap<K,V> extends Multimap<K,V>			
Invariante			
Métodos	**Devuelve**	**O/M**	**Post.**
List<V> get(K key)	Una vista de las imágenes de key	O	

5.7 Factorías de Multifunciones

Para crear multifunciones podemos las diferentes implementaciones ofrecidas por Guava y la clase de factoría *Multimaps.* Las implementaciones posibles están en als clases

ArrayListMultimap, HashMultimap, LinkedHashMultimap, LinkedListMultimap, TreeMultimap, etc. Todas tienen dos métodos create: uno que crea un *Multimap* vacío y otro que crea una copia.

Esta clase nos permite crear multifunciones con distintas implementaciones y a partir de diferentes datos de partida. Veamos algunos métodos de esa factoría.

- *static <K,V> HashMultimap<K,V> create()*
- *static <K,V> HashMultimap<K,V> create(Multimap<? extends K,? extends V> map)*
- *…*
- *static <K,V> TreeMultimap<K,V> create()*
- *static <K,V> TreeMultimap<K,V> create(Comparator<C> comparator)*
- *static <K,V> TreeMultimap<K,V> create(SortedSetMultimap<? extends K,? extends V> map)*
- *…*

El tipo *ListMultiMap* puede ser implemetado por *ArrayListMultimap* y *LinkedListMultimap*. El tipo *SetMultimap* puede ser implementado por *HashMultimap, LinkedHasmultimap y TreeMultimap*. Y el tipo *SortedsetMultimap* por *TreeMultimap*.

5.8 Tablas (tables)

El tipo Table*<R,C,V>* es un agregado de datos que asocia un valor a un par de claves que podemos pensar como filas y columnas.

Table<R,C,V>			
Invariante			
Por cada par de valores en fila y columna hay un solo valor			
Métodos	**Devuelve**	**O/M**	**Post.**
`void clear()`		M	*isEmpty()*
`boolean contains(Object rowKey, Object columnKey)`	Verdadero si contiene el par de claves rowKey, columKey	O	
`boolean containsColum(Object columnKey)`	Verdadero si contiene la columKey	O	
`boolean containsRow(Object rowKey)`	Verdadero si contiene la claves rowKey	O	
`boolean containsValue(Object value)`	`values().contains(value)`	O	
`Set<Table.Cell<R,C,V>> cellSet()`	Conjunto de tripletes que definen la tabla. Vista con las operaciones add y addAll restringidas.	0	
`boolean equals(Object m)`	Dos tabls son iguales cuando sus vistas como conjunto de celdas son iguales	O	

V get(Object rowKey, Object columnKey)	El valor asociado a las claves o null	O	
boolean isEmpty()	cellSet().isEmpty()	O	
Multiset<K> keys()	Multiconjunto de las claves.Cada clave se repite tantas veces como imágenes tenga	O	
Set<C> columKeySet()	Conjunto de las claves de las columnas. Es una vista.	O	
Set<R> rowKeySet()	Conjunto de las claves de las filas. Es una vista.	O	
Map<R,Map<C,V>> rowMap()	Devuelve una vista en forma de Map que asociada a cada clave de una fila un Map.	O	
Map<C,V> row(R rowKey)	Devuelve una vista en forma de Map para cada clave de una fila.	O	
Map<C,Map<R,V>> columnMap()	Devuelve una vista en forma de Map que asociada a cada clave de una columna un Map.	O	
Map<R,V> column(C columnKey)	Devuelve una vista en forma de Map para cada clave de una columna.	O	
V put(R rowkey, C columnKey, V value)	El antiguo valor asociado al par de claves	M	get(rowKey,columnKey).equals(value)
void putAll(Table<? extends R,? extends C,? extends V> m)		M	Añade a *this* (con el método put) cada uno de pares los pares de m.
V remove(Object rowkey, Object columnKey)	El antiguo valor, si existe, asociado al par de claves.	M	Elimina el par de claves.
int size()	cellSet().size()		
Collection<V> values()	Colección de los valores. Vista con las operaciones add y addAll restringidas.	O	

5.9 Factorías de Tablas

Para crear tablas podemos usar las diferentes implementaciones ofrecidas por Guava y la clase de factoría *Multimaps.* Las implementaciones posibles están en als clases *HashBasedTable, TreeBasedTable*. Ambas tienen dos métodos create: uno que crea una *Table* vacía y otra que crea una copia.

Esta clase nos permite crear multifunciones con distintas implementaciones y a partir de diferentes datos de partida. Veamos algunos métodos de esa factoría.

- *static <R,C,V> HashBasedTable<R,C,V> create()*

- *static <R,C,V> HashBasedTable<R,C,V> create(Table<? extends R, ? extends C, ? extends V> table)*
- *static <R,C,V> TreeBasedTable<R,C,V> create()*
- *static <R,C,V> TreeBasedTable<R,C,V> create(Comparator<? super R> rowComparator, Comparator<? super C> columnComparator)*
- *static <R,C,V> TreBasedTable<K,V> create(TreeBasedTable<R,C, ? extends V> table)*

El tipo *Table* puede ser implemetado por *HashBasedTable* y por *TreeBasedTable*. Con TreBasedTable podemos implementar también el tipo *RowSortedTable<R,C,V>* que es unsubtipo de *Table<R,C,V>* con la filas ordenadas.

5.10 Colecciones y funciones restringidas

Tanto las colecciones como las funciones pueden ser creadas con invariantes añadidos. Estos invariantes son predicados que deben ser verdad para cada elemento en la colección o cada par clave-valor en la función. Para trabajar con este tipo de agregados usamos objetos capaces de comprobar que un elemento cumple el invariante en el momento de ser añadido al agregado. Estos tipos (tomados de Guava) son *Constraint<E>* y *MapConstraint<K,V>* que podemos representarlos por las interfaces:

```
interface Constraint<E> {
	E checkElement(E element);
	String toString();
}
interface MapConstraint<K,V> {
	void checkKeyValue(K key, V value);
	String toString();

}
```

Los predicados *check…* de ambos tipos comprueban que se cumple un predicado sobre el parámetro *element* o el par de parámetros *key*, *value* y si no se cumplen se dispara una excpeción hija de *RuntimeException*. Podemos ver que ambos tipos podemos considerarlos como expresiones lógicas que disparan una excepción si evalúan a *false.* Por lo tanto cualquier objeto de tipo *Predicate<E>* puede ser convertido automáticamente a un objeto de tipo *Constraint<E>.* Alternativamente podemos implementar un objeto de tipo *Constraint<E> o MapConstraint<K,V>* siguiendo las mismas ideas que en el caso de *Predicate<E>* o *Function<T,S>.*

Disponiendo de objetos de los tipos anteriores podeos construir agregados de datos con invariantes con los métodos de la clase *Constraints.*

- *static <E> Collection<E> constrainedCollection(Collection<E> collection, Constraint<? super E> constraint)*
- *static <E> Set<E> constrainedSet(Set<E> set, Constraint<? super E> constraint)*
- *static <E> List<E> constrainedList(List<E> list, Constraint<? super E> constraint)*

- *static <E> SortedSet<E> constrainedSortedSet(SortedSet<E> collection, Constraint<? super E> constraint)*
- *static <E> Multiset<E> constrainedMultiset(Multiset<E> collection, Constraint<? super E> constraint)*

Como podemos comprobar a partir de un predicado podemos construir una restricción que dispare la excepción *RuntimeException.*

- *public static <E> Constraint<E> fromPredicate(Predicate<E> p)*

El método anterior, que se propone como ejercicio se puede ubicar el la clase *Constraints2.*

Todos los métodos anteriores devuelven vistas del agregado de partida con el invariante añadido. Para el caso de las funciones tenemos algo similar en al clase *MapContraints.*

- *static <K,V> Map<K,V> constrainedMap(Map<K,V> map, MapConstraint<? super K, ? super V > constraint)*
- *static <K,V> SortedMap<K,V> constrainedSortedMap(SortedMap<K,V> map, MapConstraint<? super K, ? super V > constraint)*

Como ejemplo de agregado de datos con restrcciones veamos la forma de construir un conjunto de enteros que solo acepte pares.

```
Set<Integer> sep = Constraints.constrainedSet(Sets.<Integer>newHashSet(),
                   Constraints2.fromPredicate(#{e -> e%2==0));
```

En el código anterior hemos usado lambda expresiones para escribir predicados.

5.11 Comprobación de parámetros y precondiciones

Junto a los predicados y restricciones vistos antes hay otros usos de expresiones lógicas que pueden ser sistematizados. Son maneras sistemáticas de comprobar que se cumple una precondición, que los parámetros de entrada de un método cumplen un predicado, que el índice sobre una lista cuple los requisitos o que el estado de un objeto permite determaindas operaciones. Esta sistematización puede coseguirse usando los métodos de la clase *Preconditions.*

- *static void checkArgument(boolean expression)*
- *static void checkArgument(boolean expression, String errorMessageTemplate, Object... errorMessageArgs)*
- *static void checkState(boolean expression)*
- *static void checkState(boolean expression, String errorMessageTemplate, Object... errorMessageArgs)*
- *static int checkElementIndex(int index, int size)*
- *static int checkElementIndex(int index, int size, String desc)*
- *static int checkPositionIndex(int index, int size)*
- *static int checkPositionIndex(int index, int size, String desc)*

- *static <T> T checkNotNull(T reference)*
- *static <T> T checkNotNull(T reference, String errorMessageTemplate, Object... errorMessageArgs)*

El método *checkArgument* comprueba que un parámetro o variable cumple un predicado en otro caso dispara *IllegalArgumentException.*

El método *checkState* comprueba que el estado de un objeto cumple un predicado en otro caso dispara *IllegalStateException.*

El método *checkElementIndex* comprueba que *0 <= index < size* en otro caso dispara *IndexOutOfBoundsException.*

El método *checkPositionIndex* comprueba que *0 <= index <= size* en otro caso dispara *IndexOutOfBoundsException.*

El método *checkNotNull* comprueba que reference es dintinto de *null* en otro caso dispara *NullPointerException.*

El uso de estos métodos no permite hacer una programación más sistemática y legible.

6. Problemas propuestos

Resuelva los problemas siguientes usando los tipos explicados en este tema junto con los de los temas anteriores: *Iterables, Ordering, Query, …*

Utilice la clase *Preconditions* para comprobar los parámetros de entrada de los métodos, las precondiciones, el estado de los objetos, etc.

1. Implemente el método public *iterableToMap* en la clase *Maps2* cuya signatura es:

   static <K,E> Map<K,List<E>> iterableToMap(Iterable<E> it, Function<E,K> f)

   que toma un iterable y una expresión y construye una función cuyas claves son los valores devueltos por la función y para cada clave la imagen es una lista con los objetos del iterable que dieron ese valor al ser evaluados con *f.*

2. Implemente el método public *iterableToIntegerMap* en la clase *Maps2* cuya signatura es:

   static <K,E> Map<K,Integer> iterableToIntegerMap(Iterable<E> it, Function<E,K> f)

   que toma un iterable y una expresión y construye una función cuyas claves son los objetos y para cada clave la imagen es el número de veces que ese objeto aparece en el iterable.

3. Implemente el método public *iterableToMultiset* en la clase *Multisets2* cuya signatura es:

static <E> Multiset<E> iterableToMultiset(Iterable<E> it)

que toma un iterable y constuye un Multiset.

4. Implemente el método *groupBy* del tipo *Query<T>* visto en el capítulo anterior cuya signatura es:

   <K> GroupQuery<K, T> groupBy(Function<T, K> e);

5. Implemente el método *join* del tipo *Query<T>* visto en el capítulo anterior cuya signatura es:

   <U, K, R> Query<R> join(Query<U> q, Function<T, K> e1, Function<U, K> e2,
   Function2<T,U,R> e3);

6. Implemente el método *creaEnterosPrimos,* que sólo acepte enteros que sean primos, en la clase *Sets2* cuya signatura es:

   static Set<Integer> creaEnterosPrimos()

7. Implemente el método public *fromPredicate* en la clase *Constraints2* cuya signatura es:

   static <E> Constraint<E> fromPredicate(Predicate<E> predicate)

   que toma una predicado y lo transforma en restricción.

8. Implemente las siguientes operaciones de diferencia entre diferentes tipo de agregados de datos.

   $s_1 - s_2,\ l_1 - l_2,\ m_1 - m_2$

   Recordatorio: *s: Set, l: List, m: Map.*

9. Implemente las siguientes operaciones que corresponde a la eliminación de una colección de los elementos contenidos en un iterable.

   $s - it, l - it$

10. Implemente la siguiente operación que corresponde a la operación inversa de un *Map* (es decir, los valores serán las claves y las claves los valores).

    $m^{-1}(v) = \{a : m^k \mid m(a) = v\}$

    Observación: Decida qué hacer con aquellos valores que aparezcan repetidos al hacer la operación inversa.

11. Implemente distintos métodos que lleven a cabo la expresión

    $argMin_o\{a : A | f(a); g(a)\}$

    Representa el agregado de valores tal que el valor del correspondiente *g(a)*, con respecto al orden *o*, es el mínimo de los valores calculados sobre el agregado *a* que cumplen el filtro

f. El resultado es del mismo tipo que: es decir conjunto, lista, iterable,… En el caso del conjunto será siempre un conjunto con un solo elemento por lo que asumimos que devuelve el elemento directamente.

Implemente métodos que devuelvan conjuntos, listas, conjuntos oredenados y multiconjuntos

12. Escriba un método llamado *contarNumRepetidos* en una clase de utilidad llamada Utiles y su correspondiente test en una clase TestUtiles. El método debe, a partir de una lista de enteros, devolver un *Map<Integer, Integer>* en el que el conjunto de claves hace referencia a los números que aparecen en la lista y el valor es el número de veces que ese número aparece en la lista. La signatura del método debe ser:

```
public static Map<Integer, Integer> contarNumRepetidos(List<Integer> lEnteros)
```

13. Escriba un método llamado *contadorTexto* en una clase de utilidad llamada Textos y su correspondiente test en una clase TestTextos. El método debe, a partir de un nombre de fichero de texto, devolver un *Map<String, Integer>* en el que el conjunto de claves hace referencia a las palabras que aparecen en el fichero y el valor es el número de veces que esa palabra aparece en el fichero. La signatura del método debe ser:

```
public static Map<String, Integer> contadorTexto(String nomFich)
```

NOTA: Implemente un método auxiliar para cargar en la aplicación una línea del fichero de texto que tenga la signatura:

```
public static void cargarLineaMap(String linea, Map<String, Integer> ap)
```

14. Escriba un método llamado *contadorTexto2* en una clase de utilidad llamada Textos y su correspondiente test en una clase TestTextos. El método debe, a partir de un nombre de fichero de texto, devolver un *Multiset<String>* con las palabras que aparecen en el fichero. La signatura del método debe ser:

```
public static Multiset<String> contadorTexto2(String nomFich)
```

15. Escriba un método llamado *crearMapInterpreteCanciones* en la clase de utilidad *Canciones* que dado un *Iterable<Cancion>* devuelva un *Map<String, List<String>>* en el que el conjunto de claves hace referencia a los nombres de los intérpretes de las canciones del Iterable y el valor una lista con las canciones de ese intérprete. La signatura del método debe ser:

```
public static Map<String, List<String>> crearMapInterpreteCancion(
        Iterable<Cancion> itCancion)
```

16. Se dispone de un fichero con los datos de una serie de personas y se desea obtener un conjunto ordenado alfabéticamente con los nombres de las personas que sean menores de una edad determinada y su apellido contenga una determinada letra.

17. Diseñe un método estático en la clase Aeropuertos que dado un Aeropuerto y un Pasajero devuelva una lista con los números de los vuelos en los que aparece el pasajero. El método retornará un conjunto que contenga los números de los vuelos encontrados. Dos pasajeros son iguales si tienen su DNI igual. Implemente las interfaces.

```
public interface Aeropuerto {
   List<Vuelo> getVuelos();
```

```
}
public interface Vuelo {
    Integer getNumeroVuelo();
    List<Pasajero> getPasajeros();
}
public interface Pasajero {
    String getDni();
    String getNombre();
}
```

18. Disponemos de dos ficheros de texto que almacenan dos listas de personas. Para cada persona almacenada en las listas conocemos su DNI, nombre, apellidos y edad. Se pide calcular:

    a) El conjunto de todas las personas que aparecen en las dos listas.
    b) El conjunto de todas las personas que aparecen en alguna de las dos listas.
    c) El conjunto de todas las personas que aparecen en las dos listas y con edad mayor de 17.

    Los ficheros constarán de empleados, cada uno de ellos representado en una línea de la siguiente forma:

    12345-D, Luis, Narváez, 23

    NOTA: Dos personas serán las mismas si tiene el mismo DNI.

19. Disponemos de un fichero con las transacciones bancarias que se realizan en un banco. Cada transacción se compone del número de cuenta del ordenante, del número de cuenta del beneficiario y de la cantidad transferida. También se dispone de otro fichero con las cantidades iniciales de cada cuenta. Se pide:

    a) Diseñar las clases Transaccion y Cuenta.
    b) Implementar un método estático en la clase Cuentas que calcule las cuentas que se encuentran en números rojos después de realizar todas las transacciones bancarias.

20. La compañía aérea EtsiiAir desea habilitar una web para la reserva de vuelos a toda Europa. Cada uno de dichos vuelos posee las siguientes propiedades:

    Identificador, Nombre, Origen, Destino, Capacidad

    Implementar una la clase *Vuelos*, que permitirá gestionar las reservas para los vuelos. Dicha clase tiene todos sus métodos estáticos y se inicializará sin ningún vuelo. Sus métodos son:

    static boolean reservaPlaza(Vuelo vuelo, Pasajero p)
    static Set<Vuelo> getVuelos(Pasajero p)
    static Set<Vuelo> getVuelos()
    static boolena addVuelo(Vuelo v);

    El método boolean *reservaPlaza* será el encargado de reservar la plaza para un pasajero. Si el vuelo no está entre la lista de vuelos disponibles, deberá devolver false, al igual que si el vuelo está completo.

El método *getVuelos* devuelve los vuelos reservados por el pasajero y el *getVuelos* todos los disponibles.
El método *addVuelo* añade un nuevo vuelo vacío a los disponibles. Devuelve false si el vuelo ya estaba o no se ha podido añadir.

21. Se desea realizar el modelado del nuevo sistema de evaluaciones de la asignatura EDA. En este sistema se decide tener en cuenta tanto la nota del examen (o de los exámenes) de cada uno de los alumnos, así como la nota de prácticas.

    Para ello se cuenta con un fichero de todos los alumnos de la asignatura en el que aparecen el nombre, los apellidos, el DNI y la dirección separadas por comas.

    Se almacena en un fichero información relativa a cada examen parcial. Las líneas de este fichero son DNI, número de examen parcial y nota. Las notas de las prácticas se almacenan en otro fichero cuyas líneas contienen el DNI el número de la práctica y nota.

    Realice un modelado del problema anterior e identifique el tipo de colecciones que deben ser usadas en cada caso.

    Modele el problema anterior para poder leer la información de los ficheros y poder calcular lo siguiente:

    - Número de alumnos en la asignatura
    - Nota de en alumno dado suponiendo que las prácticas valen el 40%, la teoría el 60% y que las notas de prácticas y teoría son respectivamente las medias de las prácticas y los exámenes parciales.

22. Una nueva cadena de hipermercados desea crear un sistema de gestión que le permita almacenar, consultar y eliminar los productos que se comercializan en un *Hipermercado.* Para ello se ha diseñado un tipo de datos *Producto* cuyas propiedades son nombre, cantidad y precio, el cual representará a cada uno de los productos ofrecidos.

    Se desea dotar al sistema de la siguiente funcionalidad establecida en forma de funciones:

    - *boolean alta (Producto p):* da de alta el producto que se pasa como parámetro en la colección de productos comercializados y devuelve true o false si lo consigue o no.
    - *boolean baja (Producto p):* da de baja el producto que se pasa como parámetro en la colección de productos comercializados y devuelve true o false si lo consigue o no.
    - *boolean existe (Producto p):* comprueba si existe el producto que se pasa como parámetro en la colección de productos comercializados
    - *Set<Producto> preciosMayorA(Double precio):* devuelve los productos cuyo precio es mayor que el que se pasa por parámetro
    - *Set<Producto> preciosMenorA(Double precio):* devuelve los productos cuyo precio es menor que el que se pasa por parámetro
    - *Set<Producto> preciosIgualA(Double precio):* devuelve los productos cuyo precio es igual que el que se pasa por parámetro
    - *Set<Producto> preciosEntre(Double precioMenor, Double precioMayor):* devuelve los productos cuyo precio está comprendido entre los precios que se pasan como parámetros

- *Set<Producto> preciosEntre(Producto p1, Producto p2):* devuelve los productos cuyo precio está comprendido entre los precios de los productos que se pasan como parámetros
- *Set<Producto> getPorPrecioCercanoMax(Double precio):* devuelve los productos cuyo precio es el más caro pero menor o igual al que se pasa como parámetro
- *Set<Producto> getPorPrecioCercanoMin(Double precio):* devuelve los productos cuyo precio es el más barato pero mayor o igual al que se pasa como parámetro
- *Set<Producto> getMasBaratos():* devuelve los productos cuyo precio es el más barato
- *Set<Producto> getMasCaros():*devuelve los productos cuyo precio es el más caro

Se pide modelar el tipo *Hipermercado* dotándolo de los métodos anteriores e implementarlo.

23. En una clase de utilidad *EjerciciosIterablesUD21* escriba los siguientes métodos SIN UTILIZAR NINGÚN BUCLE:

    - Un método que devuelva un *Iterable<Integer>* a partir de un fichero de texto que contiene en cada línea una lista de números enteros separados por comas. La signatura del método será:

```
public static Iterable<Integer> obtenerIterableEnteros(String nomFich)
```

24. Los buscadores como Google, para prestar el servicio de búsqueda de información realizan dos procesos distintos. El primer proceso se conoce como indexación y consiste en generar una serie de índices de búsqueda, como por ejemplo, un índice invertido. El segundo proceso es el de búsqueda, que se basará en los índices de búsqueda generados en la etapa anterior. Se quiere generar un índice invertido, tal y como lo hacen los buscadores, para ayudar en la búsqueda de información dentro de una serie de archivos. Un índice invertido es una colección de todas las palabras existentes en un conjunto de documentos, de tal forma que cada palabra tiene asociado un conjunto con todos los documentos en los que aparece (Figura 1). Algunos libros incluyen este tipo de índice al final de sus páginas, de forma que se listan todas las palabras relevantes y las páginas donde aparece cada una de ellas.

    Como ejemplo, suponga que tiene tres archivos, cada uno de ellos conteniendo uno de los textos que se muestran a continuación:

    - Texto 1: “Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems”
    - Texto 2: “Las aplicaciones Java están compiladas en un bytecode”
    - Texto 3: “Sun Microsystems proporciona una implementación GNU General Public License de un compilador Java y una máquina virtual Java”

    La estructura del índice invertido para un conjunto de búsqueda formado por estos tres archivos será (NOTA: Puede mirar también la Figura 1):

Palabra	**Aparece en**
GNU	Texto3.txt

```
Java            Texto1.txt, Texto2.txt, Texto3.txt
Microsystems    Texto1.txt, Texto3.txt
...
```

El tipo IndiceInvertido tiene las siguientes propiedades, que se reflejan en la interfaz que aparece en la Figura 2:

- *Textos*, de tipo *Set<String>*, solo consultable, que para una palabra dada nos devuelve un conjunto con losarchivos donde aparece.
- *Lineas, Set<Integer>,* solo consultable, que dada una palabra y un archivo donde aparece nos devuelve las líneas donde aparece.
- *NumeroDePalabras,* de tipo Integer no negativo, solo consultable, que nos indica e número de palabras distintas que aparecen.
- *Opcion*, de tipo *Integer* no negativo, consultable y modificable. Esta propiedad representa las opciones de configuración del iterador. Si la opción es cero se iterará sobre todas las palabras del conjunto de claves y si es mayor que cero se iterará sobre aquellas palabras del cojunto de las claves que aparecen en más documentos que el valor dado por *Opcion.*

Además tiene el método

```
boolean add(String p, String f, Integer lin);
```

Que añade la palabra *p* que está en la línea *lin* del texto *f*.

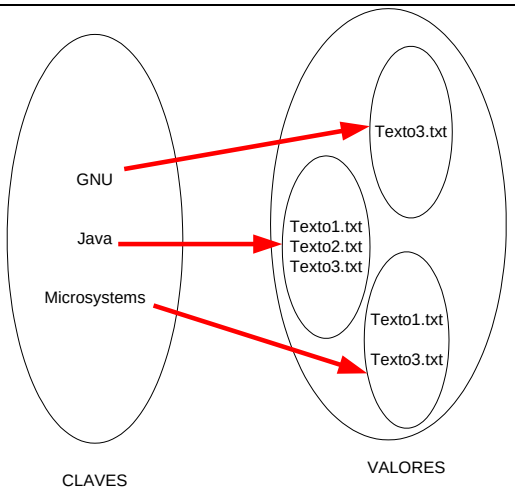


Figura 1. Índice invertido.

```
public interface IndiceInvertido
                extends Iterable<String>{
    Set<String> getTextos(String s);
    Integer getNumeroDePalabras();
    Set<Integer> getLineas(String s, String f);
    Integer getOpcion();
    void setOpcion(Integer op);
    boolean add(String p,String f,Integer lin);
}
```

Figura 2. Tipo IndiceInvertido

Se pide implementar la clase *IndiceInvertidoImpl* y una clase de utilidad *IndicesInvertidos* en la que se definan una serie de métodos para ayudar en la fase de búsqueda de los buscadores, tal como se indica en los siguientes apartados.

Para la clase *IndiceInvertidoImpl*

- Escriba los atributos y los métodos consultores y modificadores.
- Añada un constructor por defecto.

En la clase *IndicesInvertidos* escriba los métodos:

- Un método que lea una lista de ficheros y construya un objeto de tipo IndiceInvertido.

```
IndiceInvertido indexaArchivo(List<String> ficheros);
```

- Un método para responder a la pregunta ¿Cuál es la palabra del índice que comienza por *"Ja"* y se encuentra en más archivos? El método debe tener la siguiente cabecera:

```
public static String palabraPrefijoMasDocs(IndiceInvertido ii,
       String prefijo);
```

- Un método que dado un índice invertido, y un conjunto con palabras, devuelva un conjunto con los nombres de aquellos ficheros en los que aparece alguna de la palabra del conjunto. La signatura del método ha de ser la que se muestra más abajo. Implemente dos versiones de este método una en la que no use NINGÚN BUCLE, y otra en la que no use ninguna artillería de guava.

```
public Set<String> ficherosConAlgunaDeLasPalabras(IndiceInvertido ii,
       Set<String> conjPals);
```