PROGRAMACIÓN UD-12

COLECCIONES

ÍNDICE

[INTRODUCCIÓN 3](#_Toc134469309)

[TIPOS PARAMETRIZADOS O GENÉRICOS 4](#_Toc134469310)

[TIPOS PARAMETRIZADOS LIMITADOS 5](#_Toc134469311)

[METODOS GENÉRICOS 5](#_Toc134469312)

[TIPOS GENÉRICOS: COMODINES 6](#_Toc134469313)

[INTERFAZ COLLECTION 6](#_Toc134469314)

[ITERADORES 7](#_Toc134469315)

[MÉTODOS GLOBALES DE COLLECTION 7](#_Toc134469316)

[MÉTODOS GLOBALES DE TABLA 7](#_Toc134469317)

[MÉTODOS DE LA INTERFAZ LIST 8](#_Toc134469318)

[INTERFAZ SET 8](#_Toc134469319)

[CONVERSIONES ENTRE COLECCIONES 8](#_Toc134469320)

[MÉTODOS DE LA CLASE COLLECTIONS 9](#_Toc134469321)

[INTERFAZ MAP 12](#_Toc134469322)

[MÉTODOS INTEFAZ MAP 13](#_Toc134469323)

[RECORRIDO DE UN MAP 13](#_Toc134469324)

[METODOS INTERFAZ MAP.ENTRY 13](#_Toc134469325)

# INTRODUCCIÓN

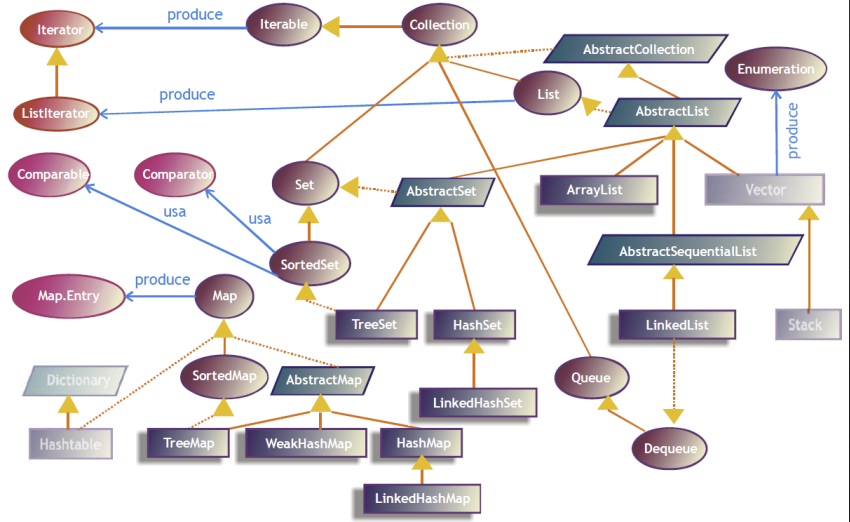
Hasta ahora, cuando teníamos un conjunto de datos del mismo tipo usábamos tablas, pero si el número de elementos varía, debemos redimensionar la tabla y copiar todos los datos a una nueva ubicación en la memoria, lo que lleva mucho tiempo.

Lo lógico para estos casos, de número de elementos cambiantes, es utilizar estructuras dinámicas de memoria, que en Java se encuentra bajo la interfaz Collection.

Definimos por tanto colección, como un conjunto de elementos que tienen una relación entre sí, y que se puede llevar a cabo una serie de operaciones sobre los mismos como son inserción, eliminación y búsqueda.

Diagrama

Descripción generada automáticamente



Existen tres tipos fundamentales de estructuras de tipo Collection:

* **Listas** (ArrayList, LinkedList): Similar a un array, pero añade más funcionalidades.
* **Conjuntos** (HashSet, LinkedHashSet, TreeSet): Es un conjunto de datos sin repeticiones.
* **Mapas o diccionarios** (HashMap, LinkedHashMap, TreeMap): No implementan la interfaz Collection, pero están muy relacionadas. Sirven para guardar datos con claves que no se repiten.

# TIPOS PARAMETRIZADOS O GENÉRICOS

Si una clase usa un tipo genérico, se podrá utilizar diferentes objetos, que se comprobarán en tiempo de ejecución.

Los métodos de comparación compareTo (Object o) y compare (Object o), utilizan tipos de datos genéricos. Para eso al poner el interfaz, debo indicar la clase “implements Comparable<Persona>”. Con esto nos quedará “public int compareTo(Persona Persona)”.

Al declarar una clase podemos definir que la definimos para tipos genéricos con un tipo por defecto “<T>”, si tenemos objetos de varios tipos usaremos varios tipos genéricos “<T, U…>”.

Texto

Descripción generada automáticamenteInterfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Aunque no sea significativo el nombre que le demos al tipo genérico, se suele usar E para colecciones, K para claves, N para números y T para tipos genéricos.



Con esto tendríamos creado un atributo llamado objeto de la clase Prueba, al que le indicaríamos que el tipo genéricos es Integer.

En la segunda sentencia, vemos que hemos dejado vacío el tipo de datos en el new, ya que será el mismo que en la definición. Estas dos maneras de crear un objeto son correctas.

Con interfaces es similar:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

# TIPOS PARAMETRIZADOS LIMITADOS

Vamos a limitar las clases que se pueden asignar a T. Se pueden limitar a la clase indicada y las subclases con extends o a la clase indicada y a las superclases de ta con super.

En teste caso T puede ser claseLimite o cualquiera de sus subclases.

Texto

Descripción generada automáticamente con confianza media

En este caso T puede ser claseLimite o cualquiera de sus superclases.

Imagen que contiene Texto

Descripción generada automáticamente

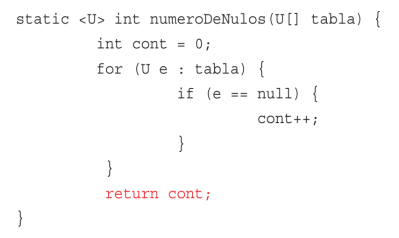
Diagrama

Descripción generada automáticamente

# METODOS GENÉRICOS

Si una clase tiene tipos parametrizados, es normal que tenga también métodos que usen esos tipos. Sin embargo, si tenemos dentro de cualquier clase métodos con sus propios parámetros genéricos distintos de los de la clase, se llamarán métodos genéricos.

En este caso contamos cuantos elementos de tipo U son nulos.



# TIPOS GENÉRICOS: COMODINES

Los comodines o *wildcards*, permiten declarar variables sin saber su tipo.



Son muy útiles con las clases límite (super y extendes).

Hay que tener cuidado, porque una clase con un tipo genérico, no va a heredar de la misma clase con el tipo porque herede de otra.

Diagrama

Descripción generada automáticamente

La relación de herencia entre clases usadas como tipos genéricos en otras clases no implica que estas últimas mantengan la misma relación de herencia.

Los tipos genéricos no pueden ser primitivos, ni se pueden crear instancias de tipo genérico (new T()), ni se pueden crear arrays de clases parametrizadas (new NombreClase<Integer>[9]), ni se pueden usar excepciones genéricas.

# INTERFAZ COLLECTION

Al ser Collection una interfaz, no se pueden crear directamente objetos de la misma, poro va a disponer de una serie de métodos que es interesante conocer, ya que serán utilizables en todas las listas y conjuntos.

* boolean add(T elemento): Inserta un elemento en la colección, devolviendo true si ha podido insertarlo y false si no ha podido.
* boolean remove(Object elemento): Igual que add, pero en este caso elimina el elemento. El método equals() determina que elementos son iguales, sino no borra elementos de la colección.
* void clear(): Vacía la colección dejándola sin elementos.
* int size(): Indica el número de elementos que tiene la colección.
* boolean isEmpty(): Indica si la colección está sin elementos.
* Boolean contains(Object elemento): Indica si el elemento está en la colección. Para usar este método, es necesario que la clase que usemos para los elementos tenga implementado el método equals. Lo que equals determine que es igual, determinara contains que existe.

# ITERADORES

Diagrama

Descripción generada automáticamente

Sirven para recorrer los elementos de las colecciones.

Al definir un iterador, apuntaremos al inicio de la colección, antes del primer elemento.

Tenemos varias funciones para usar iteradores:

* iterator(): genera un iterador del tipo de los elementos guardados en la colección.
* hasNext(): indica si existe el elemento siguiente.
* next(): devuelve el elemento siguiente y se mueve a el siguiente.
* remove(): elimina el último elemento que ha devuelto next().

# MÉTODOS GLOBALES DE COLLECTION

* boolean containsAll (Collection<?> c): Indica si todos los elementos, de la colección pasado como parámetro se encuentran en la principal.
* boolean removeAll (Collection <?> c): Se eliminan los elementos de la colección pasada como parámetro que se encuentran en la que ha invocado al método. Solo elimina todas las ocurrencias, con remove() solo la 1ª. Devuelve true si ha podido eliminar algo.
* boolean retainAll (Collection <?> c): Se elimina los elementos de la colección invocante, que se encuentren en la que se ha pasado como parámetro. Devuelve true si ha podido eliminar algo.

# MÉTODOS GLOBALES DE TABLA

* Object[] toArray(): Devuelve un array de objetos que contiene los elementos de la colección.
* <T> T[] toArray(T[] arrT): Igual que el anterior, pero el array es del tipo de los elementos. El array que pasamos como parámetro, se suele crear en ese momento con tamaño 0.
* <T> List<T> asList(T[] arrayT): Este es un método de Arrays, no de Collection. Pero hace lo contrario del anterior, a partir de un array, devuelve una lista con sus elementos.

# MÉTODOS DE LA INTERFAZ LIST

Es importante la ubicación de los elementos, permitiendo insertar o sacar elementos de esa ubicación.

* E get(int intIndice): Devuelve el elemento que ocupa la posición intIndice, siendo 0 el primero elemento y size() – 1 el último.
* E set(int intIndice, E elemento): Inserta en la posición intIndice el elemento y devuelve el valor actual de esa posición.
* void add(int intIndice, E elemento): Inserta en la posición intIndice el elemento y los elementos desde esa posición se mueven hacia delante.
* boolean addAll(int intIndice, Collection<? extends E> colec2): Inserta la lista colec2 en la posición indicada en el índice.
* E remove(int intIndice): Elimina un elemento y lo devuelve.
* int indexOf(E elemento): Indica el índice de la primera ubicación del elemento.
* int lastIndex(E elemento): Indica el índice de la última ubicación del elemento.
* Boolean equals(Object lista2): Indica si las listas son iguales, da igual que no sean de la misma clase.
* void sort(Comparator <? super E> colec2): Ordena la lista según el comparador.

# INTERFAZ SET

No tiene repeticiones de los elementos y deben definir el método equals y el hashCode.

Puede usar todos los métodos de Collection y puede recorrerse con un iterador y con un for-each.

Existen tres tipos de set:

* HashSet: funciona rápido, pero no tiene orden.
* LinkedHashSet: los elementos se guardan según el orden de inserción.
* TreeSet: es un árbol de elementos ordenados. Debe implementar Compatable (método compareTo).

# CONVERSIONES ENTRE COLECCIONES

Existen dos maneras de pasar una colección a otra:

* Se utiliza el método addAll(otra), indicando la colección desde la que se desea convertir.
* En el constructor se le pasa como parámetro la colección.

# MÉTODOS DE LA CLASE COLLECTIONS

Hay una clase llamada Collections (no confundir con la interfaz Collection) que contiene numerosos métodos estáticos para usar con todo tipo de colecciones.

* static <T> boolean addAll(Collection<? Super T> c, T… elementos): Añade la lista de elementos de tipo T (que irán separados por comas) a la colección indicada por c.
* static <T> int binarySearch(List<? extends T> l, T valor, Comparator<? Super T> c): Busca de forma binaria el objeto en la lista que deberá estar ordenada según el comparador indicado.
* static <T> int binarySearch(List<? extends Comparable <? super T>> l, T valor): Busca de forma binaria el objeto en la lista que deberá estar odenada (por ello los elementos de la lista deben ser de una clase que implemente la interfaz Comparable).
* static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> tipo): Devuelve una vista de tipo seguro en el tiempo de ejecución de una colección. Un intento de añadir un elemento de clase incompatible provoca la excepción ClassCastException.
* static <E> List<E> checkedList(List<E> l, Class<E> tipo): Igual que la anterior pero orientado a listas doblemente enlazadas.
* static <K, V> Map<K, V> checkedMap(Map<K, V> m, Class<K> tipoClave, Class<V> tipoValor): Funciona como las anteriores pero pensada para mapas.
* static <K, V> SortedMap<K, V> checkedSortedMap(SortedMap<K, V> m, Class<L> tipoClave, Class <V> tipoValor): Para mapas ordenados.
* static <K, V> SortedMap<K, V> checkedSortedSet(SortedSet<E> s, Class<E> tipo): Para hash ordenados.
* static <T> void copy(List<? super T> lista1, List<? super T> lista2): Copia los elemento de la segunda lista en la primera. Esa primera parte tiene que ser tan larga como la segunda, ya que los elementos copiados reemplazan a los originales. Si no es suficientemente grande ocurre una excepción tipo IndexOutOfBoundsException.
* static boolean disjoint(Collection<?> c1, Collection<?> c2): Devuelve verdadero si ambas colecciones contienen elementos distintos.
* static <T> List<T> emptyList(): Devuelve un objeto del tipo List indicado vacía.
* static <K, V> Map<K, V> emptyMap(): Devuelve un objeto del tipo Map indicado vacío.
* static <T> Set<T> emptySet(): Devuelve un objeto del tipo List indicado vacío.
* static <T> Enumeration<T> enumeration(Collection<T> c): Obtiene una enumeración a partir de la colección.
* static <T> void fill(List<? extends T> lista, T obj): Reemplaza los elementos de la lista con el objeto indicado. Si la lista tiene 10 elementos, ahora los 10 serán una copia del objeto.
* static int frequency(Collection<?> c, Object obj): Cuenta el número de veces que aparece el objeto indicado en la colección.
* static int indexOf(Lista<?> lista, Lista<?> sublista: Busca todos los elementos de la sublista en la lista indicada. Devuelve la posición en la que empieza la sublista. Si no hay esa sublista, devuelve -1.
* static int lastIndexOf(Lista<?>, Lista<?> sublista): Idéntica a la anterior pero empieza a buscar desde el final de la lista, por lo que devuelve la posición inicial en la que comienza la última vez que aparece la sublista en la lista (o -1 si no se encuentra).
* static <T> ArrayList<T> list(Enumeration<T> enum): Obtiene una lista de tipo ArrayList a partir de la enumeración indicada.
* static <T> T max(Collection<? extends T> c, Comparator<? Super T> comp): Devuelve el mayor valor de la colección, comparando sus elementos según el comparador indicado.
* static <T extendes Object & Comparable <? Super T>> T max max(Collection<? extends T> c): Devuelve el mayor valor de la colección, que debe implementar la interfaz Comparable e indicar en el método compareTo la forma de ordenar.
* static <T> T min(Collection<? extendes T> c, Comparator<? Super T> comp): Devuelve el menor valor de la colección, comparando sus elementos según el comparador indicado.
* static <T extends Object & Comparable <? Super T>> T max min(Collection<? extends T> c): Devuelve el menor valor de la colección, que debe implementar la interfaz Comparable e indicar en el método compareTo la forma de ordenar.
* static <T> List<T> nCopies(int num, T obj): Obtiene una lista formada por tantas copias como indique el parámetro num, del objeto indicado.
* static <T>boolean replace All(List< T> lista, T antiguo, T nuevo): Reemplaza en la lista todas las apariciones del elemento antiguo por el nuevo. Devuelve false si no pudo realizar ni un reemplazo (por que el objeto antiguo no está en la lista)
* static <T>boolean reverse(List <T> lista: invierte el orden de la lista.
* static <T>Comparator<T> reverseOrder(Comparator<IT> comparador): Obtiene un comparador que invierte el orden en el que ordena el comparador que recibe como parámetro.
* static <T>Comparator<T> reverseOrder(Comparator<T> comparador): Obtiene un comparador que invierte el orden en el que ordena normalmente la clase T (que deberá implementar la interfaz Comparable). Por ejemplo: Arrays.sort(a, Collections.reverseOrder()) ordenaría el array a al revés del orden normal en el que se solería ordenar.
* static void rotate(List<T> lista, int n): Rota la lista el número de elementos indicados por n. Si la lista 1 es [1,2,3,4], esta instrucción Collections.rotate(I,1) dejaría la lista con los valores [2,3,4,1]. El número n puede ser negativo y entonces la rotación es la izquierda.
* static void shuffle(List<T> lista): Mezcla aleatoriamente los elementos de la lista.
* static void shuffle(List<T> lista, Random r): Mezcla aleatoriamente la lista utilizando como semilla de rotación el objeto r.
* static <T> Set <T> singleton(T obj): Devuelve obj como una lista de tipo Set inmutable y serializable.
* static <T> List <T> singletonList(T obj): Devuelve obj como una lista de tipo List inmutable y
* static <K,V> Map <K,V> singletonMap(K clave, V valor): Devuelve el elemento de clave y valor indicados como un mapa inmutable y serializable.
* static <T extends Comparable<? super T> void sort(List<T> lista): Ordena la lista según el orden natural de los elementos (cuya clase debe implementar la interfaz Comparable) de la
* static <T> void sort(List<T> lista, <Comparator <? super t> comp): Ordena la lista en base al comparador indicado. Los elementos iguales no se reordenan entre sí.
* static <T> void swap(List<T> lista, int il, int 12): Intercambia en la lista los elementos cuyos índices son los indicados. La lista debe poseer elementos en dichos índices son pena de provocar la excepción: IndexOutOfBoundsException.
* static <T> Collection <T> synchronizedCollection(Collection <T> c): Obtiene una colección sincronizada para hilos seguros a partir de la colección indicada
* static <T> List <T> synchronizedList(List <T> lista): Obtiene una lista sincronizada para hilos seguros a partir de la lista indicada.
* static <K,V> SortedMap <K,V> synchronizedMap(Map <K,V> mapa): Obtiene un mapa sincronizado para hilos seguros a partir del mapa indicado
* static <T> Set <T> synchronizedSet(Set <T> set): Obtiene una lista sin duplicados sincronizada para hilos seguros a partir de la lista indicada
* static <K,V> SortedMap <K,V> synchronizedSortedMap(SortedMap <K,V> mapa): Obtiene un mapa ordenado sincronizado para hilos seguros a partir del mapa indicado
* static <T> SortedSet <T> synchronizedSortedSet(Set <T> set): Obtiene una lista ordenada sin duplicados sincronizada para hilos seguros a partir de la lista indicada
* static <T>Collection<T> unmodifiableCollection( Collection<? extends T> c): Obtiene una colección de sólo lectura a partir de la colección indicada
* static <T>List<T> unmodifiableList(List<? extends T> lista): Obtiene una lista de sólo lectura a partir de la lista indicada
* static <K, V>Map<K,V> unmodifiableMap( Map<? extends K, ? extends V> mapa): Obtiene un mapa de sólo lectura a partir del indicado
* static <T>Set<T> unmodifiableSet(Set<? extends T> lista): Obtiene una lista de valores
* únicos, de sólo lectura a partir de la lista indicada
* static <K,V>SortedMap<K,V> unmodifiableSortedMap( SortedMap<? extends K, ? extends V> mapa): Obtiene un mapa ordenado de sólo lectura a partir del indicado
* static <T>Set<T> unmodifiableSortedSet(Set<? extends T> lista): Obtiene una lista de valores únicos ordenados, de sólo lectura a partir de la lista indicada

# INTERFAZ MAP

Son similares a las colecciones y almacenan unos elementos llamados Map.Entry que contienen una clave y un valor.

Existen tres tipos de map:

* HashMap: funciona rápido, pero no tiene orden.
* LinkedHashMap: los elementos se guardan según el orden de inserción.
* TreeMap: es un árbol elementos ordenados.

Al crear un mapa, tendremos que indicar dos parámetros genéricos, que indicarán a la clave y al valor. En el ejemplo metemos clientes con un número como clave.

# MÉTODOS INTEFAZ MAP

* V put(K clave, V valor): Similar a add de colecciones. Inserta un valor con la clave indicada. Si esta existe, modifica el valor. Devuelve el valor antiguo.
* V remove(Object clave): Elimina la entrada con la clave indicada. Si esta existe, devuelve el valor, sino devuelve null.
* void clear(): Elimina todas las entradas del mapa, quedando vacío.
* V get(Object clave): Devuelve el valor si existe la clave, sino devuelve null.
* boolean containsKey(Object clave): Devuelve true si existe la clave, sino devuelve false.
* boolean containsValue(Object valor): Devuelve el true si existe alguna entrada con el valor indicado, sino devuelve false.
* Set<K> keySet(): Nos devuelve una colección Set con el conjunto de claves del mapa.
* Collection<V> values(): devuelve una vista Collection, con los valores del mapa.
* Set<MapEntry <K, V> entrySet(): Devuelve una vista del conjunto de entradas clave:valor, que tuviera almacenado el mapa.

# RECORRIDO DE UN MAP

Existen diversas formas de recorrer un mapa, dependiendo de cual sea nuestro interés.

Lo normal es que nos interese ver de cada Map.Entry tanto su clave como su valor, por tanto lo podríamos hacer con un for-each de entrySet().

Puede que nos interése mirar sólo las claves por lo que podríamos hacer un for-each de keySet().

Puede interersarnos recorrer sólo los valores con un for-each de values().

También es posible que nos interese recorrer el mapa con un iterador de las claves.

# METODOS INTERFAZ MAP.ENTRY

* K getKey(): Devuelve la clave del Map.Entry.
* V getValue(): Devuelve el valor del Map.Entry.
* V setValue(V nuevoValor): Modifica el valor del Map.Entry, devuelve el antiguo.