8. Colecciones de datos

Autor	Xerach Casanova
Clase	Programación
Fecha	@Mar 14, 2021 9:39 PM

1. Introducción a las colecciones

Una colección a nivel software es un grupo de elementos almacenados de forma conjunta en una misma estructura. Una colección o contenedor es un objeto que agrupa elementos múltiples en un objeto simple. Las colecciones se usan para almacenar, recuperar y manipular datos. Un array o vector no está incluido en el framework Colletions.

Las colecciones definen un conjunto de interfaces, clases genéricas y algoritmos que permite manejar los grupos de objetos que a priori están relacionados entre sí (aunque no es obligatorio) y pueden trabajar con cualquier tipo de objeto (por eso se emplean genéricos).

Además permiten realizar algunas operaciones útiles sobre elementos almacenados, como búsqueda u ordenación. En algunos casos, los objetos almacenados deben cumplir

algunas condiciones implementando algunas interfaces para hacer uso de esos algoritmos.

En java, el uso de las colecciones es bastante más sencillo que en otros lenguajes. Parten de una serie de interfaces básicas que definen un modelo de colección y las operaciones que se pueden llevar a cabo sobre los datos almacenados. La interfaz inicial, a partir de la cual están creadas todas las colecciones es java.util.Collection y define todas las operaciones comunes de las colecciones derivadas.

Hay que tener en cuenta que una colección es una interfaz genérica donde "<E>" es el parámetro de tipo.

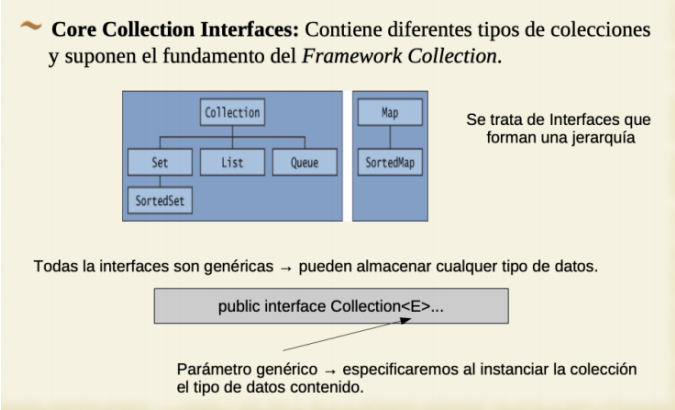
Las operaciones más importantes de cualquier colección son:

- Método **int size():** retorna el número de elementos de la colección.
- Método **boolean isEmpty():** retornará verdadero si la colección está vacía.
- Método **boolean contains (Object element):** retornará verdadero si la colección tiene el elemento pasado como parámetro.
- Método **boolean add(E element):** permitirá añadir elementos a la colección.
- Método **boolean remove (Object element):** permitirá eliminar elementos de la colección.
- Método **Iterator <E> iterator():** permitirá crear un iterador para recorrer los elementos de la colección.
- Método **Object[] toArray():** permite pasar la colección a un array de objetos tipo Object.
- Método **containsAll(Collection<?> c):** permite comprobar si una colección contiene los elementos existentes en otra colección, si es así, retorna verdadero.
- Método **addAll (Collection<? extends E> c):** permite añadir todos los elementos de una colección a otra colección, siempre que sean del mismo tipo (o deriven del mismo tipo base).
- Método **boolean removeAll(Collection<?> c):** si los elementos de la colección pasada como parámetro están en nuestra colección, se eliminan, el resto se quedan.
- Método **boolean retainAll(Collection<?> c):** si los elementos de la colección pasada como parámetro están en nuestra colección, se dejan, el resto se eliminan.
- Método **void clear():** vacíar la colección.

Las ventajas de utilizar las colecciones son:

1. Reducen el esfuerzo de programación.
2. Incrementan la calidad de las aplicaciones.
3. Incrementan la interoperabilidad, muchas APis utilizan clases o interfaces de colecciones como argumentos en sus métodos.

4. Reducen el esfuerzo de aprender nuevas APIs
5. Contribuyen a la reusabilidad del código.



2. Clases y métodos genéricos (I)

Las clases y métodos genéricos son un recurso de programación disponible en muchos lenguajes, con el objetivo de facilitar la reutilización de software, creando distintos métodos y clase que pueden trabajar con diferentes tipos de objetos, evitando la conversión de tipos.

Ejemplo de versión genérica y no genérica de un método compararTamano.

Versión no genérica.

```
public class util {

  public class util {public static int compararTamano(Object[] a,Object[] b) {
     return a.length-b.length;
  }

}
```

Versión genérica.

```
public class util {
    public static <T> int compararTamano (T[] a, T[] b) {
    return a.length-b.length;
  }
}
```

Los dos métodos permiten y comprueban si un array es mayor que otro. Retornan 0 si son iguales, mayor de cero si el array b es mayor y menor de cero si el array es mayor. La versión genérica incluye la expresión <T> antes del tipo retornado por el método. Esta es la definición de una variable o parámetro formal de tipo de la clase o método genérico (parámetro de tipo o parámetro genérico) y se puede utilizar a lo largo de todo el método

o clase haciendo referencia a cualquier clase con la que nuestro algoritmo tiene que trabajar.

Para invocar un método genérico, solo hay que realizar una invocación de tipo genérico, olvidándonos de de las conversiones de tipo. Consiste en indicar qué clases o interfaces concretas se utilizarán en lugar de cada parámetro genérico "<t>" para después ejecutar ese algoritmo pasándole los argumentos correspondientes.

Cada clase o interfaz la podemos denominar tipo o tipo base y se da por sentado que los argumentos pasados al método genérico serán también de dicho tipo base.

Invocación genérica y no genérica de un método Integer.

Invocación del método no genérico.	Invocación del método genérico.

`Integer []a={0,1,2,3,4};`
`Integer []b={0,1,2,3,4,5};`
`util.compararTamano ((Object[])a, (Object[])b);`
	`Integer []a={0,1,2,3,4};`
	`Integer []b={0,1,2,3,4,5};`
	`util.<Integer>compararTamano (a, b);`

2.1. Clases y métodos genéricos (II)

Las clases genéricas permiten definir un parámetro de tipo o genérico que se puede utilizar a lo largo de toda la clase, facilitando crear clases genéricas que son capaces de trabajar con diferentes tipos de datos base. Para crear una clase genérica se especifican los parámetros de tipo al lado del nombre de la clase.

```
public class Util<T> {
   T t1;
   public void invertir(T[] array) {
     for (int i = 0; i < array.length / 2; i++) {
       t1 = array[i];
       array[i] = array[array.length - i - 1];
       array[array.length - i - 1] = t1;
     }
   }
}
```

En el ejemplo anterior, la clase Util contiene el método invertir cuya función es invertir el orden de los elementos de cualquier array, sea del tipo que sea. Para usar esa clase genérica hay que crear un objeto o instancia de dicha clase especificando el tipo base entre <>, justo detrás del nombre de la clase. Veamos un ejemplo:

```
Integer[] numeros={0,1,2,3,4,5,6,7,8,9};
Util<Integer> u= new Util<Integer>();
u.invertir(numeros);
for (int i=0;i<numeros.length;i++) System.out.println(numeros[i]);
```

Simplemente a la hora de crear una instancia genérica se especifica el tipo, tanto en la definición (Util<integer> u) como en la creación (new Util<Integer>()).

Los parámetros de tipo de las clases genéricas no pueden ser datos primitivos: int, short, double, etc...

3. Conjuntos (I)

Los conjuntos son un tipo de colección que no admite duplicados.

La interfaz java.util.Set define cómo deben ser los conjuntos y extiende la interfaz Collection, aunque no añade ninguna operación nueva. Las implementaciones más usadas son las siguientes:

- **java.util.HashSet.** Conjunto que almacena los objetos usando tablas hash.
  - Ventajas: rapidez en el acceso a objetos.
  - Desventajas: no almacenan de forma ordenada y necesitan bastante memoria.
- **java.util.LinkedHashSet.** Conjunto que almacena objetos combinando tablas hash y listas enlazadas. El orden de almacenamiento es el de la inserción.
  - Ventajas: rapidez en acceso a objetos y almacena de forma ordenada.
  - Desventajas: necesitan bastante memoria y es algo más lenta que HashSet.
- **java.util.TreeSet.** Conjunto que almacena objetos usando estructuras conocidas como árboles rojo-negro.
  - Ventajas: los datos almacenados se ordenan por valor aunque se almacenen de manera desordenada.
  - Desventajas: son más lentas que las dos anteriores.

Para crear un conjunto se crea el HashSet se debe importar el paquete java.util.Hashset indicando el tipo de objeto que se va a almacenar, dado que es una clase genérica que puede trabajar cualquier tipo de datos.

```
HashSet<Integer> conjunto=new HashSet<Integer>();
```

Después podemos almacenar objetos dentro del conjunto con el método add (definido por la interfaz Set). Los objetos deben ser siempre del tipo especificado al crear el conjunto:

```
Integer n=new Integer(10);
if (!conjunto.add(n)) System.out.println("Número ya en la lista.");
```

Si el elemento ya está en el conjunto, el método add retorna false y podemos indicar que no se pueden insertar duplicados.

3.1. Conjuntos (II)

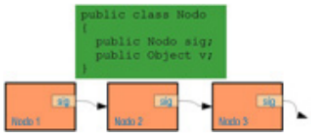
Para obtener los elementos almacenados en un conjunto se utilizan iteradores que permiten obtener elementos del conjunto uno a uno de forma secuencial. la forma más transparente de usarlo es a través de un for-each:

```
for (Integer i: conjunto) {
 System.out.println("Elemento almacenado:"+i);
}
```

3.2. Conjuntos (III)

La diferencia entre LinkedHasSet, TreeSet y HashSet radica en su funcionamiento interno.

LinkedHashSet es una estructura que funciona como lista enlazada, pero también usa tablas ash para acceder rápidamente a los elementos. Una lista enlazada es una estructura compuesta por nodos o elementos que forman la lista y que se van enlazando entre sí.



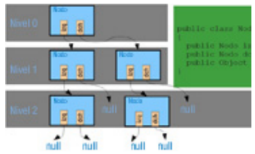
Un nodo contiene el dato almacenado y el siguiente nodo de la lista. Si no hay nodo, la variable que contiene el siguiente nodo es nula.

Las listas enlazadas tienen un montón de operaciones asociadas como son: inserción de un nodo al final, al principio, entre dos nodos, etc.

```
LinkedHashSet <Integer> t;
t=new LinkedHashSet<Integer>();
t.add(new Integer(4));
t.add(new Integer(3));
t.add(new Integer(1));
t.add(new Integer(99));
for (Integer i:t) System.out.println(i);
```

Salida por pantalla: 4 3 1 99 (los valores salen ordenados según el momento de inserción en el conjunto)

Treeset utiliza internamente árboles, que son como listas pero más complejos, en vez de tener un único elemento siguiente, puede tener dos o más, formando estructuras organizadas y jerárquicas.



Los nodos se diferencian en nodos padre y nodos hijos. Un nodo padre puede tener varios hijos asociados, dando lugar a una estructura de árbol invertido.

Puesto que un nodo hijo puede ser padre a su vez, los árboles se suelen visualizar para su estudio por niveles para entenderlos mejor, donde cada nivel contiene hijos de los nodos del nivel anterior, excepto el del primer nivel que no tiene padre.

Los árboles son estructuras complejas de manejar con operaciones muy sofisticados. Los árboles utilizados en Treeset son árboles rojo-negro, que son auto-ordenados. Al insertar un elemento, se queda ordenador por su valor y al recorrer el árbol los elementos salen ordenados. También tiene operaciones a nivel interno como son inserción de nodos, eliminación, búsqueda de valor, etc.

La creación de un TreeSet es similar a la de un HashSet, solo sustituyendo su nombre. Tampoco admite duplicados y se utilizan los métodos vistos antes, existentes en la interfaz Set.

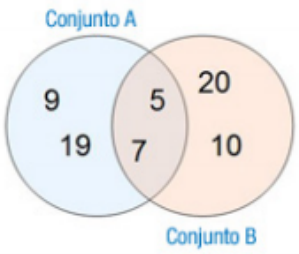
```
TreeSet <Integer> t;
t=new TreeSet<Integer>();
t.add(new Integer(4));
t.add(new Integer(3));
t.add(new Integer(1));
t.add(new Integer(99));
for (Integer i:t) System.out.println(i);
```

Salida por pantalla: 1 3 4 99 (el resultado sale ordenado por valor).

3.3. Conjuntos (IV)

Los conjuntos y las colecciones facilitan operaciones para poder combinar datos de varias colecciones.

Ejemplo:

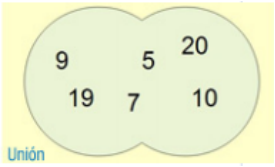



```
TreeSet<Integer> A= new TreeSet<Integer>();

A.add(9); A.add(19); A.add(5); A.add(7); // Elementos del conjunto A: 9, 19, 5 y 7

LinkedHashSet<Integer> B= new LinkedHashSet<Integer>();

B.add(10); B.add(20); B.add(5); B.add(7); // Elementos del conjunto B: 10, 20, 5 y 7
```

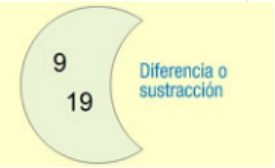
En el ejemplo anterior, el literal de número se convierte automáticamente a la clase envoltorio Integer sin tener que hacer nada.

Tipos de combinaciones:

- **Unión. A.add(B).** Añadir todos los elementos del conjunto B con el A. Todos los del conjunto A, añadiendo los del B, pero sin
  repetir los que ya están: 5, 7, 9, 10, 19 y 20.



- **Diferencia. A.removeAll(B).** Eliminar los elementos del conjunto B que puedan estar en el conjunto A. Todos los elementos del
  conjunto A, que no estén en el conjunto B: 9, 19.



- **Intersección. A.RetainAll(B).** Retiene los elementos comunes de ambos conjuntos. Todos los elementos del conjunto A, que también están en el conjunto B: 5 y 7.

3.5. Conjuntos (V)

Los TreeSet ordenan sus elementos de forma ascendente, pero tienen un conjunto de operaciones adicionales además de las que incluye por el hecho de ser conjunto, que permite, por ejemplo, cambiar la forma de ordenar los elementos. Es muy útil para tipos de objetos complejos. TreeSet es capaz de ordenar tipos básicos, como números, cadenas o fechas, pero otros tipo de objetos no se puede con tanta facilidad.

Para indicar a un TreeSet cómo tiene que ordenar los elementos debemos decirle cuándo un elemento va antes o después que otro, y cuándo son iguales. Para ello podemos utilizar la interfaz genérica java.util.Comparator, la cual se usa en general en algoritmos de ordenación, creando una clase que implementa dicha interfaz. Esta interfaz solo requiere un método que debe calcular si un objeto pasado por parámetro es mayor, menor o igual que otro del mismo tipo.

```
class ComparadorDeObjetos implements Comparator<Objeto> {
 public int compare(Objeto o1, Objeto o2) { ... }
}
```

La interfaz Comparator obliga a implementar un método (compare), el cual tiene dos parámetros, que son los dos elementos a comparar.

- Si el objeto (o1) es menor que el objeto (o2), debe retornar un número entero negativo. otra forma de verlo: Si el primer objeto (o1) debe ir antes que el segundo objeto (o2).
- Si el objeto (o1) es mayor que el objeto (o2), debe retornar un número positivo. otra forma de verlo: Si el primer objeto (o1) debe ir después que el segundo objeto (o2).
- Si ambos son iguales, debe retornar 0.

Una vez creado el comparador solo tenemos que pasarlo como parámetro en el momento de la creación al TreeSet y los datos internamente mantendrán dicha ordenación.

Ejemplo:

Tenemos un objeto en una clase como la siguiente:

```
class Objeto {
 public int a;
 public int b;
}
```

Al añadirlos en un TreeSet, estos se tienen que ordenar de forma que la suma de sus atributos (a y b) sea descendente, ¿como sería el comparador?

```
class ComparadorDeObjetos implements Comparador<Objeto> {
 @Override
   public int compare(Objeto o1, Objeto o2) {

     int sumao1=o1.a+o1.b;
     int sumao2=o2.a+o2.b;

     if (sumao1<sumao2) return 1;
     else if (sumao1>sumao2) return -1;
     else return 0;
   }
}
```

4. Listas (I)

Las listas son elementos de programación algo más avanzados que los conjuntos. Amplían el conjunto de operaciones de colecciones añadiendo operaciones extra:

- Pueden almacenar duplicados, si no queremos duplicados se debe verificar manualmente antes de la inserción.
- Acceso posicional. Se puede acceder a un elemento indicando su posición en la lista.
- Búsqueda. Se puede buscar y obtener posición. En las colecciones solo se puede saber si un conjunto contenía o no un elemento, devolviendo true o false.

- Extracción de sublistas. Podemos obtener una lista que contenga solo una parte de los elementos.

Java, para las listas dispone de la interfaz java.util.list y dos implementaciones: java.util.LinkedList y java.util.ArrayList. Todos los métodos de la interfaz List están presentes en ambas interfaces y son:

- **E get(int index).** Permite obtener un elemento partiendo de su posición.
- **E set(int index, E element).** Permite cambiar el elemento almacenado en una posición de la lista por otro que se le pase por parámetro.
- **void add (int index, E Element).** Se añade el elemento en la lista en una posición concreta, desplazando los siguientes.
- **E remove (int index).** Permite eliminar un elemento indicando su posición.
- **boolean addAll(int index, Collection <?extends E> c).** Permite insertar una colección pasada por parámetro en una posición de la lista desplazando al resto de elementos siguientes.
- **int indexOf(Object o).** Permite conocer la posición de un elemento y si no está retorna -1.
- **int lastIndexOf(Object o).** Permite obtener la última ocurrencia del objeto en la lista.
- **List<E> subList(int from, int to).** Genera una sublista coin elementos correspondidos entre la posición inicial incluida y la posición final no incluida.

Todos los elementos de una lista comienzan a enumerarse por 0.

4.1. Listas (II)

Hay que recordar que para hacer uso de LinkedList y ArrayList hay que importar los paquetes correspondientes.

En el siguiente ejemplo creamos un LinkedList con números enteros. El contenido de la lista al final será: 2, 3 y 5

```
LinkedList<Integer> t=new LinkedList<Integer>(); // Declaración y creación del LinkedList de enteros.

t.add(1); // Añade un elemento al final de la lista.
t.add(3); // Añade otro elemento al final de la lista.
t.add(1,2); // Añade en la posición 1 el elemento 2.
t.add(t.get(1)+t.get(2)); // Suma los valores contenidos en la posición 1 y 2, y lo agrega al final.
t.remove(0); // Elimina el primer elementos de la lista.

for (Integer i: t) System.out.println("Elemento:" + i); // Muestra la lista.
```

En este otro ejemplo creamos un ArrayList, el resultado final será 10 y 12.

```
ArrayList<Integer> al=new ArrayList<Integer>(); // Declaración y creación del ArrayList de enteros.
```

```
al.add(10); al.add(11); // Añadimos dos elementos a la lista.

al.set(al.indexOf(11), 12); // Sustituimos el 11 por el 12, primero lo buscamos y luego lo reemplazam
```

En el ejemplo siguiente utilizamos sublistas. Con el método size se obtiene el tamaño de la lista. El método subList extrae una sublista de la lista desde la posición indicada por parámetro hasta la posición indicada por parámetro. El método addAll añade todos los elementos de la sublista al arraylist anterior.

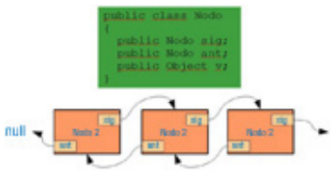
```
al.addAll(0, t.subList(1, t.size()));
```

Se debe tener en cuenta que si se realiza una operación sobre una sublista, también repercute sobre la lista original. En el siguiente ejemplo, borrando de la sublista también se borra de la lista.

```
al.subList(0, 2).clear();
```

4.2. Listas (III)

La diferencia entre LinkedList y ArrayList es que los LinkedList utilizan listas doblemente enlazadas, que son listas enlazadas que permiten ir hacia atrás en la lista de elementos. Los elementos de la lista se encapsulan en los llamados nodos, estos nodos van enlazados unos a otros para no perder el orden y no limitar el tamaño de almacenamiento.



Al tener un doble enlace significa que cada nodo almacena la información del siguiente y del anterior. Si un nodo no tiene siguiente o anterior se almacena null.

En el caso de los ArrayList se implementan utilizando arrays que se van redimensionando de forma transparente conforme se necesita más o menos espacio. Los ArrayList son más rápidos en cuanto a acceso a los elementos, ya que en una lista doblemente enlazada hay que recorrer la lista. En cambio eliminar un elemento implica muchas más operaciones en un array que en una lista enlazada.

Sabiendo lo anterior, llegamos a la conclusión que si se van a realizar muchas operaciones de eliminación sobre una lista, se debe utilizar LinkedList, y si la mayoría de operaciones consisten en insertar o consultar por posición, conviene usar ArrayList.

LinkedList implementa las interfaces java.util.Queue y java.util.Deque, las cuales permiten utilizar las listas como si uferan una cola de prioridad o una pila respectivamente.

Las colas son conocidas como colas de prioridad (FIFO, primero que llega, primero en ser atendido).

En una LinkedList normal tenemos estos tres métodos:

- **boolean add(E e) y boolean offer(E e),** retornarán true si se ha podido insertar el elemento al final de la LinkedList.
- **E poll()** retornará el primer elemento de la LinkedList y lo eliminará de la misma. Al insertar al final, los elementos más antiguos siempre están al principio. Retornará null si la lista está vacía.
- **E peek()** retornará el primer elemento de la LinkedList pero no lo eliminará, permite examinarlo. Retornará null si la lista está vacía.

Las pilas son todo lo contrario a las listas. Mientras que en una lista se añade y se elimina al final, en una pila se añade y se elimina al principio. Para ello se utilizan tres métodos:

- push: meter al principio de la pila.
- pop: sacar y eliminar al principio de la pila.
- peek: igual que s i se usara la lista pero en una cola.

4.3. Listas (IV)

Cuando usamos colecciones, no es lo mismo usarlos con objetos inmutables como Strings, Integer, etc... que con objetos mutables.

Los objetos mutables no se pueden modificar después de su creación, cuando se incorporan a la lista a través del método add se pasan por copia. En cambio los objetos mutables (objetos propios) no se copian y puede producir efectos no deseados.

Ejemplo:

```
class Test
  {
   public Integer num;
   Test (int num) {
        this.num=new Integer(num);
       }
  }
```

Esta clase es mutable, no se pasa copia a la lista.

Si en el siguiente código se crea una lista que utilice este tipo de objetos:

```
Test p1=new Test(11); // Se crea un objeto Test donde el entero que contiene vale 11.

Test p2=new Test(12); // Se crea otro objeto Test donde el entero que contiene vale 12.

LinkedList<Test> lista=new LinkedList<Test>(); // Creamos una lista enlazada para objetos tipo Test.

lista.add(p1); // Añadimos el primero objeto test.

lista.add(p2); // Añadimos el segundo objeto test.

for (Test p:lista) System.out.println(p.num); // Mostramos la lista de objetos
```

En el código anterior se mostrarían los números 10 y 11.

Si modificamos el valor de uno de los números de los objetos test:

```
p1.num=44;
for (Test p:lista) System.out.println(p.num);
```

El resultado de imprimir la lista será 44 y 12.

El número ha sido modificado sin tener que volver a insertarlo en la lista, ya que en la lista está almacenado el apuntador al objeto original. Solo existe un objeto al que se hace referencia desde distintos lugares.

5. Conjuntos de pares clave/valor

Los conjuntos de pares son arrays asociativos, los cuales permiten almacenar pares de valores conocidos como clave/valor. La clave se utiliza para acceder al valor, como una entrada de un diccionario permite acceder a su definición.

En java existe java.util.Map, que define los métodos que deben tener los mapas. Existen tres implementaciones principales: java.util.HashMap, java.util.TreeMap y java.util.LinkedHashMap. Cada una de las tiene características similares a HashSet, TreeSet y LinkedHashset, tanto en funcionamiento como en rendimiento.

Los mapas utilizan clases genéricas para dar extensibilidad y flexibilidad y permiten definir un tipo base para la clave y otro para el valor.

```
HashMap<String,Integer> t=new HashMap<String,Integer>();
```

Métodos principales de los mapas:

- **V put(K key, V value);** Inserta un par de objetos llave (key) y valor (value) en el mapa. Si la llave ya existe en el mapa, entonces retornará el valor asociado que tenía antes, si la llave no existía, entonces retornará null.
- **V get(Object key);** Obtiene el valor asociado a una llave ya almacenada en el mapa. Si no existe la llave, retornará null.
- **V remove(Object key);** Elimina la llave y el valor asociado. Retorna el valor asociado a la llave, por si lo queremos utilizar para algo, o null, si la llave no existe.
- **boolean containsKey(Object key);** Retornará true si el mapa tiene almacenada la llave pasada por parámetro, false en cualquier otro caso.
- **boolean containsValue(Object value);** Retornará true si el mapa tiene almacenado el valor pasado por parámetro, false en cualquier otro caso.
- **int size();** Retornará el número de pares llave y valor almacenado en el mapa.
- **boolean isEmpty();** Retornará true si el mapa está vacío, false en cualquier otro caso.

6. Iteradores (I)

Los iteradores nos permiten recorrer las colecciones de dos formas: bucles for-each y a través de un bucle normal creando un iterador.

Para crear un iterador se invoca el método iterator() de cualquier colección. Por ejemplo:

```
Iterator<Integer> it=t.iterator();
```

Se especifica un parámetro para el tipo de dato genérico en el iterador poniendo <Integer> después de Iterator, ya que también son clases genéricas. Si no se especifica el tipo base también se permite recorrer la colección pero retornará objetos tipo Object y nos vemos forzados a conversión de tipo.

Para recorrer y gestionar la colección el iterador ofrece tres métodos:

- **boolean hasNext();** Devuelve true si quedan más elementos a la colección por visitar.
- **E next().** Devuelve el siguiente elemento de la colección, si no existe lanza una excepción (NoSuchElementException). Conviene chequear primero si existe el elemento.
- **remove().** Elimina de la colección el último elemento retornado de la invocación next. Si next no ha sido invocado lanza una excepción.

Por tanto para recorrer una colección con estos métodos se utiliza un while:

```
while (it.hasNext()) // Mientras que haya un siguiente elemento, seguiremos en el bucle.
  {
     Integer t=it.next(); // Escogemos el siguiente elemento.
     if (t%2==0) it.remove(); //Si es necesario, podemos eliminar el elemento extraído de la lista.
  }
```

6.1. Iteradores (II)

Los inconvenientes de usar los iteradores sin especificar el tipo de objeto es la obligatoriedad de tener que usar la operación de conversión de tipos. Usando genéricos se aportan grandes ventajas.

Ejemplo indicando el tipo de objeto del iterador:

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

for (int i=0;i<10;i++) lista.add(i);

Iterator<Integer> it=lista.iterator();

while (it.hasNext()) {
   Integer t=it.next();
   if (t%2==0) it.remove();
}
```

Ejemplo sin indicar el tipo de objeto del iterador:

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

for (int i=0;i<10;i++) lista.add(i);

Iterator it=lista.iterator();

while (it.hasNext()) {
   Integer t=(Integer)it.next();
   if (t%2==0) it.remove();
}
```

Para recorrer mapas con iteradores se utiliza el método entrySet que ofrecen los mapas para generar un conjunto con las entradas (pares de llave-valor), o bien el método keySet para generar un conjunto con ls llaves existentes en el mapa:

```
HashMap<Integer,Integer> mapa=new HashMap<Integer,Integer>test();

for (int i=0;i<10;i++) mapa.put(i, i); // Insertamos datos de prueba en el mapa.

for (Integer llave:mapa.keySet()) // Recorremos el conjunto generado por keySet, contendrá las llaves
  {
   Integer valor=mapa.get(llave); //Para cada llave, accedemos a su valor si es necesario.
  }
```

El conjunto generado por KeySet no tiene el método add para añadir elementos al mismo, eso se tiene que hacer a través del mapa.

7. Algoritmos

Los algoritmos basados en colecciones nos sirven para:

- Ordenar listas y arrays
- Desordenar listas y array
- Búsqueda binaria en listas y arrays
- Conversión de arrays a listas y de listas a arrays
- Partir cadenas y almacenar el resultado en un array.

La mayoría de estos algoritmos están recogidos como métodos estáticos en java.util.Collections y java.util.Arrays, salvo los referentes a cadenas.

Los algoritmos de ordenación lo hacen ordenando de manera natural siempre que java sepa como hacerlo. Si no, hay que facilitar el mecanismo para producir la ordenación.

Los tipos ordenables son enteros, cadenas y fechas en orden ascendente.

La clase Collections y la clase Arrays facilitan el método sort que permite ordenar listas y arrays. Los siguientes ejemplos ordenan números de forma ascendente.

Ejemplo de ordenación de array de números:

```
Integer[] array={10,9,99,3,5};
Arrays.sort(array);
```

Ejemplo de ordenación de una lista con números

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

lista.add(10);
lista.add(9);
lista.add(99);
lista.add(3);
lista.add(5);

Collections.sort(lista)
```

7.1. Algoritmos (II)

En java hay dos maneras para cambiar la forma en que se ordenan los elementos.

Imagina que hay artículos almacenados en una lista llamada "articulos" y que cada artículo se almacena en la siguiente clase, teniendo en cuenta que el código es un String.

```
class Articulo {

   public String codArticulo; // Código de artículo
   public String descripcion; // Descripción del artículo.
   public int cantidad; // Cantidad a proveer del artículo.
}
```

La primera forma es crear una clase que implemente la interfaz java.util.Comparator e implementar el método compare definido en esa interfaz.

```
class comparadorArticulos implements Comparator<Articulo> {
 @Override
   public int compare( Articulo o1, Articulo o2) {
      return o1.codArticulo.compareTo(o2.codArticulo);
  }
```

Una vez creada la clase, simplemente se pasa como segundo parámetro una instancia del comparador creado.

```
Collections.sort(articulos, new comparadorArticulos());
```

La segunda forma consiste en hacer que los objetos que se meten en la lista o array implementen la interfaz java.util.Comparable. Todos los objetos que implementan esta interfaz son ordenables y se puede invocar el método sort sin indicar un comparador para ordenarlos. La interfaz comparable solo requiere implementar el método compareTo.

```
class Articulo implements Comparable<Articulo>{
    public String codArticulo;
    public String descripcion;
    public int cantidad;
    @Override

    public int compareTo(Articulo o) {
        return codArticulo.compareTo(o.codArticulo);
    }
}
```

La interfaz Comparable es genérica y para que funcione es conveniente indicar el tipo de base sobre el que se permite la comparación. El método compareTo solo admite un parámetro y es el que comparará el objeto con el que se pasa por parámetro.

El funcionamiento del método compareTo es el mismo que el método compare de Comparator (-1, 0, 1).

```
Collections.sort(articulos);
```

7.2. Algoritmos (III)

java.util.Collections y java.util.Arrays ofrecen, además los siguientes métodos.

- **Desordenar una lista. Collections.shuffle(lista);** Desordena una lista, este método no está disponible para arrays.
- **Rellenar una lista o array. Collections.fill(lista, elemento); Array.fill(array, elemento);** Rellena una lista o array copiando el mismo valor en todos los elementos del array ,o lista. Útil para reiniciar una lista o array.
- **Búsqueda binaria: Collections. binarySearch(lista, elemento); Arrays.binarySearch(array, elemento);** Permite realizar búsquedas rápidas en un una lista o array ordenados. Es necesario que la lista o array estén ordenados, si no lo están, la búsqueda no tendrá éxito.
- **Convertir un array a lista List lista = Arrays.asList(array);** Si el tipo de dato almacenado en el array es conocido es conveniente especificarlo: List<Integer>lista = Arrays.asList(array); Permite rápidamente convertir un array a una lista de elementos, extremadamente útil. No se especifica el tipo de lista retornado (no es ArrayList ni LinkedList), solo se especifica que retorna una lista que implementa la interfaz java.util.List.

- **Convertir una lista a array Integer[] array = new Integer[lista.size()]; lista.toArray(array);** Permite convertir una lista a array. Esto se puede realizar en todas las colecciones, y no es un método de la clase Collections, sino propio de la interfaz Collection. Es conveniente que sepas de su existencia.
- **Dar la vuelta. Collections.reverse(lista);** Da la vuelta a una lista, poniéndola en orden inverso al que tiene.

Otra operación es la de dividir texto por partes. Cuando una cadena está formada por trozos de texto delimitados por un separador que se repite es posible obtener cada uno de los trozos de texto por separado en un array de cadenas. Para poder realizar esta operación se utiliza split de la clase String.

```
String texto="Z,B,A,X,M,O,P,U";

String []partes=texto.split(",");

Arrays.sort(partes);
```

Mapa conceptual