**COLECCIONES**

**Qué son las colecciones**

**Definición**

Las colecciones (Collections) son el modo de agrupar objetos. También llamados contenedores, representa a un conjunto de ítems, un conjunto de objetos, que pueden ser homogéneos o no. Por ejemplo, una agenda es una colección de datos de personas.

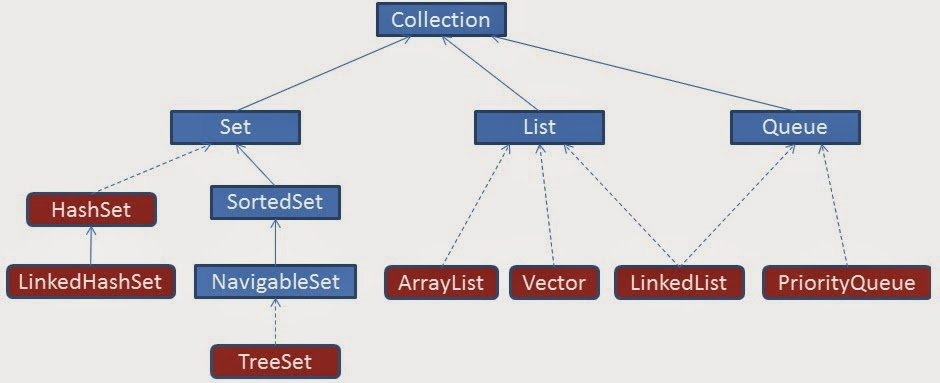
**La interfaz Collection**

La interfaz collection es la super interfaz de donde heredan las mayorías de las interfaces utilizadas para el manejo de las colecciones. Es la interfaz raíz de la jerarquía de interfaces.

La interfaz Collection forma parte del Collection Framework, un conjunto de interfaces y clases que representan distintos modos de agrupar objetos, según distintas políticas de manejo de memoria o acceso a ellos.

Representan un conjunto de objetos, también llamados elementos. Una clase que quiera comportarse como una Collection deberá implementar esta interfaz, por lo tanto, sus métodos.

A continuación, observamos un gráfico con la jerarquía de Interfaces que componen el Collection Framework, como dato adicional las interfaces correspondientes a Queue están disponibles desde la versión 1.5 de Java.



En el gráfico se observan tres grandes grupos de interfaces que representan tres tipos de contenedores distintos en su comportamiento.

**Interfaz Set**

Un set es una interfaz que no contiene valores duplicados, tiene tres implementaciones de propósitos generales en Java:

1) [HashSet](https://www.google.com/url?q=http://www.beingjavaguys.com/2013/03/hash-set-in-java.html&sa=D&ust=1503404578571000&usg=AFQjCNGKB11PwXI5VeetV6n6sCLcNPL-aw" \t "_blank)

2) TreeSet

3) LinkedHashSet

**HashSet**

HashSet es la implementación más performante de la interfaz Set. Almacena sus elementos en una tabla Hash sin garantizar ningún tipo de orden durante la iteración.

public static void main(String args[]){

        HashSet<String> hashSet = new HashSet<String>();

        hashSet.add("element1");

        hashSet.add("element2");

        System.out.println(hashSet);

        hashSet.remove("element1");

        System.out.println(hashSet);

    }

**TreeSet**

TreeSet es un poco más lenta que HashSet, almacena sus elementos en una estructura de árbol rojo-negro. TreeSet ordena sus elementos en base a sus valores.

public static void main(String args[]){

         TreeSet<String> treeSet = new TreeSet<String>();

         treeSet.add("element1");

        treeSet.add("element2");

        treeSet.add("element3");

        System.out.println(treeSet);

         treeSet.remove("element2");

        System.out.println(treeSet);

    }

**LinkedHashSet**

LinkedHashSet está implementada como una HashTable con una LinkedList corriendo sobre esta. Ordena sus elementos en base al orden en el que fueron insertados en el Set.

  public static void main(String[] args) {

        LinkedHashSet<String> linkedHashSet = new LinkedHashSet<String>();

        linkedHashSet.add("element1");

        linkedHashSet.add("element2");

        linkedHashSet.add("element3");

        System.out.println(linkedHashSet);

        linkedHashSet.remove("element1");

        System.out.println(linkedHashSet);

    }

**Interfaz List**

Una Lista es una colección ordenada que puede contener valores duplicados. Puede contener elementos Nulos. Soporta manipulación de elementos vía índices a través del método: Object get(int indice), que permite obtener el elemento en la posición índice de la lista.

También puede obtenerse el índice en el cual está almacenado un determinado objeto, mediante el método: int indexOf(unObjeto).

También permite trabajar con un subconjunto de elementos de la lista con el método: List subList(int indiceDesde, int indiceHasta).

Existen las siguientes tres implementaciones en Java:

1) [ArrayList](https://www.google.com/url?q=http://www.beingjavaguys.com/2013/04/arraylist-in-java.html&sa=D&ust=1503404578577000&usg=AFQjCNFhYY_IlWM-qFPx1RmGomcJBkWK7w" \t "_blank)

2) LinkedList

**ArrayList**

[ArrayList](https://www.google.com/url?q=http://www.beingjavaguys.com/2013/04/arraylist-in-java.html&sa=D&ust=1503404578578000&usg=AFQjCNGbAuusKyFFBkTO0HdWSZ00zAGF5w" \t "_blank) es la implementación más performante en condiciones normales. Para definirlo de una forma simple es un array de valores u objetos expandible en tamaño.

  public static void main(String args[]){

        ArrayList<String> arrayList = new ArrayList<String>();

        arrayList.add("element1");

        arrayList.add("element2");

        arrayList.add("element3");

        System.out.println(arrayList);

        arrayList.remove("element3");

        System.out.println(arrayList);

    }

**LinkedList**

En la mayoría de los casos LinkedList es un poco más lenta que [ArrayList](https://www.google.com/url?q=http://www.beingjavaguys.com/2013/04/arraylist-in-java.html&sa=D&ust=1503404578579000&usg=AFQjCNGAnSC_Mbbzab6eFkLqA36CD7XRMQ" \t "_blank)  pero puede funcionar mejor en ciertas condiciones.  LinkedList está implementada con una lista doblemente linkeada mientras que ArrayList está implementada con un array que se redimensiona dinámicamente.

LinkedList permite realizar inserción y eliminaciones de forma más performante, pero el acceso a los elementos siempre es secuencial, solo se puede iterar sobre la lista para adelante y para atrás, para acceder un elemento en el medio toma tiempo proporcionalmente al tamaño de la lista.

ArrayList permite el acceso aleatorio a sus elementos, por lo que se puede acceder rápidamente a cualquier elemento a un tiempo constante, pero las inserciones y eliminaciones de cualquier lado (excepto al final) requiere mover todos los elementos, ya sea para hacer un hueco para el nuevo elemento o llenar un espacio dejado por uno que se eliminó. Si se añaden más elementos que la capacidad del arreglo, uno nuevo con el doble de tamaño es creado, y el arreglo anterior es copiado al nuevo, con todo el tiempo de cómputo que eso significa en arreglos grandes.

  public static void main(String args[]){

        LinkedList<String> linkedList = new LinkedList<String>();

        linkedList.add("element1");

        linkedList.add("element2");

        linkedList.add("element3");

        System.out.println(linkedList);

        linkedList.remove("element3");

        System.out.println(linkedList);

    }

**Interfaz Queue**

Es una colección diseñada para contener elementos anteriormente a que se procesen. Además de las operaciones básicas de una colección, provee operaciones adicionales de inserción, extracción e inspección. Cada uno de estos métodos existen en dos formas: una arroja una Exception en caso de que la operación falle, la otra devuelve un valor especial (null o False, dependiendo de la operación).

Las colas típicamente, pero no necesariamente, ordenan sus elementos de la forma FIFO (First In First Out). También se encuentran las colas LIFO (Last In First Out, también llamadas Stacks o Pilas).

Más allá del orden que se utilice, en todas las colas la cabeza o head de la cola es el elemento que se remueve llamando al método remove() o  poll().

Podemos destacar dos implementaciones para la interfaz Queue:

1)LinkedList

2) PriorityQueue

**LinkedList**

Como vimos anteriormente LinkedList implementa la interfaz List, pero si observamos la imagen con la jerarquía de interfaces observaremos que también implementa Queue.

         public static void main(String args[]){

                Queue queueA = new LinkedList();

           // Metodos que arrojan execpciones ante una falla

               queueA.add("element 1"); // inserta un elemento en la cola

         queueA.add("element 2");

         queueA.add("element 3");

         queueA.remove(); // remueve y devuelve el head de una cola

         queueA.element(); //  devuelve, pero no remueve, el head de una cola

         // Metodos que devuelven null o false ante una falla.

queueA.offer("element 4"); // inserta un elemento en la cola,  puede fallar en el caso de una cola con capacidad limitada

         queueA.poll(); // remueve y devuelve el head de una cola

        queueA.peek(); //  devuelve, pero no remueve, el head de una cola

    }

**PriorityQueue**

Las priority queues ordenan sus elementos acorde a un comparador provisto, o el orden natural de los elementos.

public static void main(String[] args){

        Comparator<String> comparator = new StringLengthComparator();

        PriorityQueue<String> queue = new PriorityQueue<String>(10, comparator);

        queue.add("short");

        queue.add("very long indeed");

        queue.add("medium");

        while (queue.size() != 0)

        {

            System.out.println(queue.remove());

        }

    }

**Interfaz Deque**

Una Deque interfaz también conocida como double-queue provee inserción y extracción de elementos tanto del final como del principio de la cola.

**Iteradores**

**Definición**

Es un patrón de diseño utilizado para recorrer las colecciones, abstrayendo al usuario de la implementación de la colección. En Java es una interfaz denominada Iterator. Está formado por tres métodos:

**boolean hasNext()**, retorna true en caso de haber más elementos y false en caso de llegar al final del iterador.

**Object next()**, retorna el siguiente elemento en la iteración.

**void remove()**, remueve un objeto de la colección.

En las clases Vector, ArrayList, HashSet y TreeSet un iterador se consigue a través del método: Iterator iterator()

**Utilización**

Construye un contenedor, en este caso un ArrayList

Ejemplo:

ArrayList lasPersonas = new ArrayList();

  lasPersonas.add("Pepe");

  lasPersonas.add("Juan");

  lasPersonas.add("Sabrina");

  lasPersonas.add("Cecilia");

  Iterator it = lasPersonas.iterator();

  while(it.hasNext()){

        String unaPersona = (String)it.next();

  }

**La interfaz Map**

Map es un conjunto de interfaces basadas en la interfaz java.util.Map que no hereda de Collection pero que igualmente contiene operaciones del tipo *collection-view*, lo que permite que sean manipuladas como colecciones.

La interfaz Map provee tuplas del tipo Key-value, los objetos Map contienen claves o Keys asociadas con valores o values. Los Maps no pueden contener claves duplicadas y una clave puede ser asociada como máximo a un elemento. Existen tres implementaciones destacables:

1. HashMap
2. TreeMap
3. LinkedHashMap

**HashMap**

Una HashMap es una implementación del tipo HashTable de la interfaz MAP, al contrario de una HashTable puede contener claves y valores null. HashMap no garantiza que el orden de los objetos será el mismo a través del tiempo.

  public static void main(String args[]){

    HashMap<Integer, String> hashMap = new HashMap<Integer, String>();

    hashMap.put(0,"value1");

    hashMap.put(1,"value2");

    hashMap.put(2,"value3");

    System.out.println(hashMap);

    hashMap.remove(0);

    System.out.println(hashMap);

  }

**TreeMap**

TreeMap provee una implementación de Map basada en una estructura de datos del tipo arbol red-black.

 public static void main(String args[]){

        TreeMap<String, Integer> treeMap = new TreeMap<String, Integer>();

        treeMap.put("key1",123);

        treeMap.put("key2",435);

        treeMap.put("key3",654);

        System.out.println(treeMap);

        treeMap.remove("key1");

        System.out.println(treeMap);

    }

**LinkedHashMap**

Es una implementación de Map con una HashTable y una LinkedList. LinkedHashMap tiene una doble lista linkeada a través de sus elementos.

public static void main(String args[]){

        HashMap<Integer, String> linkedHashMap = new HashMap<Integer, String>();

        linkedHashMap.put(0,"value1");

        linkedHashMap.put(1,"value2");

        linkedHashMap.put(2,"value3");

        System.out.println(linkedHashMap);

        linkedHashMap.remove(0);

        System.out.println(linkedHashMap);

   }

**Collection Implementations**

Las clases que implementan la interfaz Collection por lo general tienen nombres que indican el tipo de implementación a bajo nivel, este dato es muy importante porque la performance de cada una de las clases varía de acuerdo a como se la vaya a utilizar.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Interface | Hash Table | Resizable Array | Balanced Tree | Linked List | Hash Table + Linked List |
| Set | [HashSet](https://www.google.com/url?q=http://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html&sa=D&ust=1503404578595000&usg=AFQjCNErrppF4GfQnC0phaGZEkMsTEjumQ) |  | [TreeSet](https://www.google.com/url?q=http://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html&sa=D&ust=1503404578596000&usg=AFQjCNFPLf_V38ic-FFeFMdyzlSD8Iew-w) |  | [LinkedHashSet](https://www.google.com/url?q=http://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html&sa=D&ust=1503404578597000&usg=AFQjCNFM-JK3izzXhVJjz4H7Ssj0WQyiEA) |
| List |  | [ArrayList](https://www.google.com/url?q=http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html&sa=D&ust=1503404578599000&usg=AFQjCNHYqxR2qtNn2ncJ6eclmAdrryEFtg) |  | [LinkedList](https://www.google.com/url?q=http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html&sa=D&ust=1503404578599000&usg=AFQjCNFwZDUgoEeR1mgUSv7u0oel0rFWpw) |  |
| Deque |  | [ArrayDeque](https://www.google.com/url?q=http://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html&sa=D&ust=1503404578601000&usg=AFQjCNHPPbam0ocbtubH_ccO0ZjuPx7kBA) |  | [LinkedList](https://www.google.com/url?q=http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html&sa=D&ust=1503404578602000&usg=AFQjCNGQ8LhthwC0YRwLsWUoXzMPG5pjYQ) |  |
| Map | [HashMap](https://www.google.com/url?q=http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html&sa=D&ust=1503404578603000&usg=AFQjCNHTsIFRPBp_jVSV0UoN_Sglr7xaqQ) |  | [TreeMap](https://www.google.com/url?q=http://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html&sa=D&ust=1503404578604000&usg=AFQjCNFbx7wKsT6eEmLtPFaxjoBi3dBTOA) |  | [LinkedHashMap](https://www.google.com/url?q=http://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html&sa=D&ust=1503404578605000&usg=AFQjCNGevl7skXYnztIDCoEdklzNOMPlVA) |

**GENERICS**

Este concepto permite que un tipo o método operen con objetos de varios tipos mientras se provee una seguridad en tiempo de compilación. Esta funcionalidad especifica el tipo de objetos almacenados en una colección Java.

Generics suma estabilidad al código haciendo más detectables errores de programación en tiempo de compilación. El siguiente bloque de código ilustra el problema que se puede dar al no utilizar generics. Primero, se declara un objeto del tipo ArrayList. Luego, se inserta un String al ArrayList. Finalmente, se intenta obtener el String insertado y castearlo a un Integer.

List v = new ArrayList();  
v.add("test");  
Integer i = (Integer)v.get(0); // Run time error

Aunque el código es compilado sin error, este tira una runtime exception (java.lang.ClassCastException) al ejecutar la tercer línea de código. Este tipo de problemas pueden ser evitados utilizando generics, esta es una de las principales motivaciones para utilizar generics. Utilizando generics, el código anterior puede ser reescrito de la siguiente forma:

List<String> v = new ArrayList<>();  
v.add("test");  
Integer i = v.get(0); // (type error)  compilation-time error

El parámetro de tipo String dentro de los brackets mayor y menor declara que el ArrayList será constituido por objetos String. Con generics, ya no es necesario castear la tercer línea a un tipo particular, porque el resultado de **v.get(0)**es definido como String por el código generado por el compilador.

Compilando la tercer línea de este fragmento con J2SE 5.0 (o posterior) arrojara un error en tiempo de compilación porque el compilador detectará que **v.get(0)**devuelve un String en lugar de un Integer.