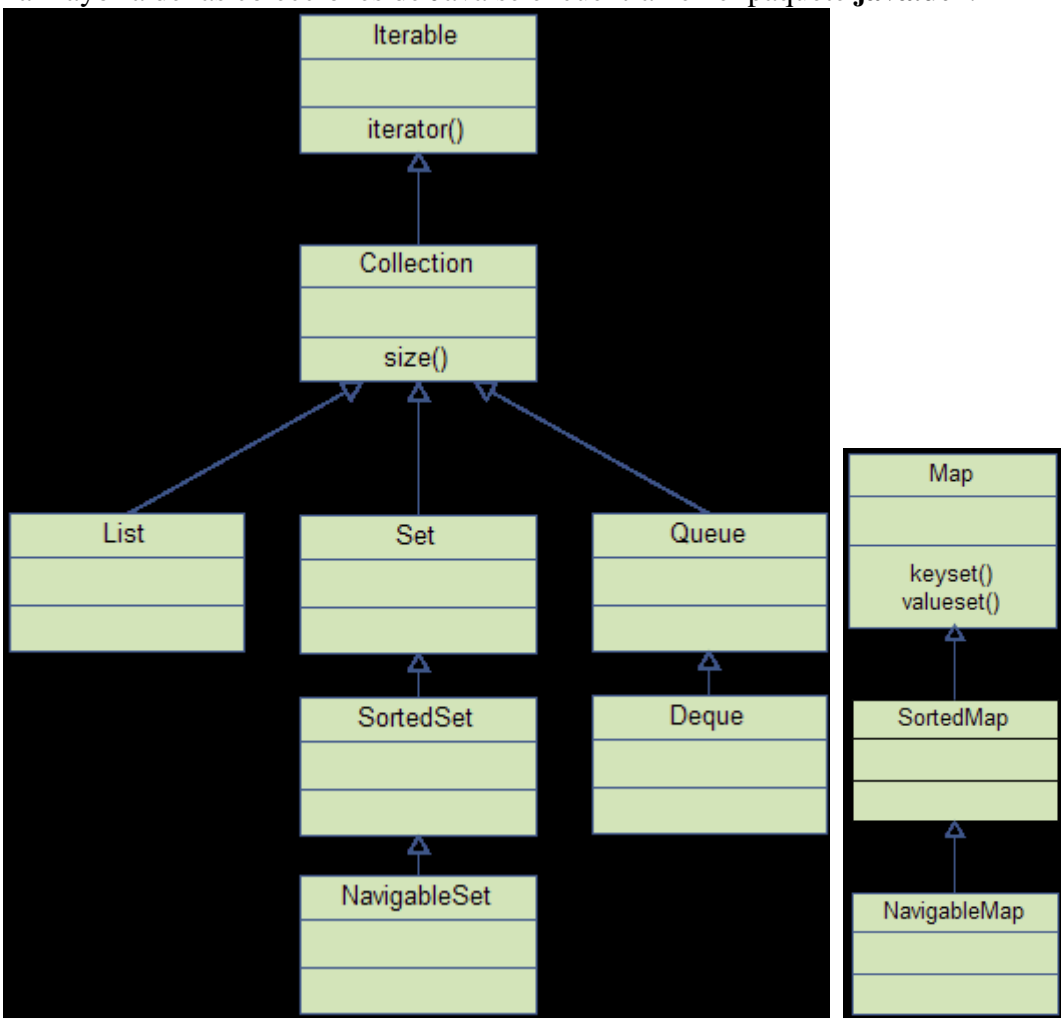
Colecciones en Java
(Java Collections)

La API de Colecciones *(Collections) de* Java provee a los desarrolladores con un conjunto de clases e interfaces que facilitan la tarea de manejar colecciones de objetos. En cierto sentido, las colecciones trabajan como las tablas, la diferencia es que su tamaño puede cambiar dinámicamente y cuentan con un comportamiento (métodos) más avanzado que las tablas.

La mayoría de las colecciones de Java se encuentran en el paquete **java.util**.



Existen una serie de **convenciones para nombrar a los genéricos**:
E – Element (usado bastante por Java Collections Framework)
K – Key (Llave, usado en mapas)
N – Number (para números)
T – Type (Representa un tipo, es decir, una clase)
V – Value (representa el valor, también se usa en mapas)
S,U,V etc. – usado para representar otros tipos.

Iterable

La interfaz Iterable es una de las interfaces raíz de las clases de colecciones. La interfaz Collection hereda de Iterable, así que todos los *subtipos* de Collection también implementan la interfaz **Iterable**.

Una clase que implementa la interfaz Iterable puede ser usada con el nuevo ciclo for. A continuación se muestra un ejemplo de dicho ciclo:

```
//Entorno:
List lista;
//Algoritmo:
lista = new ArrayList();
for(Object obj : lista){
   //hacer algo con obj
}//Fin Para
```

La interfaz Iterable únicamente tiene un método:

```
public interface Iterable<T> {
          public Iterator<T> iterator();
}
```

Collection

La interfaz Collection es otra de las interfaces raíz de las clases de colecciones de Java. Aunque no es posible instanciar de manera directa una Colección, en lugar de eso se instancia un subtipo de Colección (una lista por ejemplo), de manera recurrente querremos tratar a estos subtipos de manera uniforme como una Colección.

Las siguientes interfaces (tipos de colección) heredan de la interfaz Collection:

- List
- Set
  - SortedSet
    - NavigableSet
- Queue
  - Deque

Java no incluye una implementación utilizable de la interfaz Collection, así que se tendrá que usar alguno de los subtipos mencionados. La interfaz Collection (como cualquier otra interfaz) únicamente define un grupo de métodos (comportamiento) que compartirán cada uno de los subtipos de dicha interfaz. Lo anterior, hace posible el ignorar el tipo específico de colección que se está usando y se puede tratar como una colección genérica (Collection).

Aquí se muestra un método que opera sobre un Collection:

```
public class EjemploCollection{
   public static void hacerAlgo(Collection collection) {
   //Entorno:
      Iterator iterador;
      Object obj;
   //Algoritmo:
      iterador = collection.iterator();
      while(iterador.hasNext()){
         obj = iterador.next();
         //hacer algo con obj aqui...
      }//Fin Mientras
   }
}
```

A continuación, se muestran algunas formas de llamar este método con diferentes subtipos de Collection:

```
//Entorno:
Set conjunto;
List lista;
//Algoritmo:
conjunto = new HashSet();
lista = new ArrayList();
EjemploCollection.hacerAlgo(conjunto);
EjemploCollection.hacerAlgo(lista);
```

Sin importar el subtipo de Collection que se esté utilizando, existen algunos métodos comunes para agregar y quitar elementos de una colección:

```
//Entorno:
String cadena;
Collection coleccion;
boolean huboUnCambio,seEliminoElemento;
//Algoritmo:
cadena = "Esto es una cadena de texto";
coleccion = new HashSet();
huboUnCambio = coleccion.add(cadena);
seEliminoElemento = coleccion.remove(cadena);
```

- **add()** agrega el elemento dado a la colección y regresa un valor verdadero (*true*) si la colección cambió como resultado del llamado al método add(). Un conjunto (*set*) por ejemplo, podría no haber cambiado ya que, si el conjunto ya contenía el elemento, dicho elemento no se agrega de nuevo. Por otro lado, si el método add() fue llamado con una lista (*List*) y la lista ya contenía el elemento, dicho elemento se agrega de nuevo y ahora existirán 2 elementos iguales dentro de la lista.
- **remove()** quita el elemento dado y regresa un valor verdadero si el elemento estaba presente en la colección y fue removido. Si el elemento no estaba presente, el método remove() regresa falso (*false*).

También se pueden agregar y remover colecciones de objetos. Aquí hay algunos ejemplos:

```
//Entorno:
   Set unConjunto = ... // se agregan los elementos del conjunto aquí
   List unaLista = ... // se agregan los elementos de la lista aquí
   Collection coleccion;
//Algoritmo:
   coleccion = new HashSet();
   coleccion.addAll(unConjunto); //aquí también devuelve un booleano pero lo
   coleccion.addAll(unaLista); //estamos ignorando
   coleccion.removeAll(unaLista); //también devuelve booleano...
   coleccion.retainAll(unConjunto); //también devuelve booleano...
```

- **addAll()** agrega todos los elementos encontrados en la colección pasada como argumento al método. El objeto colección no es agregado, solamente sus elementos. Si en lugar de utilizar addAll() se hubiera utilizado add(), entonces el objeto colección se habría agregado y no sus elementos individualmente. Exactamente cómo se comporta el método addAll() depende del subtipo de colección. Algunos subtipos de colección permiten que el mismo elemento sea agregado varias veces (listas), otros no (conjuntos).
- **removeAll()** elimina todos los elementos encontrados en la colección pasada como argumento al método. Si la colección pasada como argumento incluye algunos elementos que no se encuentran en la colección objetivo, simplemente se ignoran.
- **retainAll()** hace lo contrario que removeAll(). En lugar de quitar los elementos encontrados en la colección argumento, retiene todos estos elementos y quita cualquier otro elemento que no se encuentre en la colección argumento. Hay

que tener en cuenta que, si los elementos ya se encontraban en la colección objetivo, estos son retenidos. Cualquier nuevo elemento encontrado en la colección argumento que no se encuentre en la colección objetivo no se agrega automáticamente, simplemente se ignora.

Veamos un ejemplo utilizando pseudo-código:

```
//Entorno:
Collection colA,colB,objetivo;
//Algoritmo:
colA = [A,B,C]
colB = [1,2,3]
objetivo = [];
objetivo.addAll(colA); //objetivo ahora contiene [A,B,C]
objetivo.addAll(colB); //objetivo ahora contiene [A,B,C,1,2,3]
objetivo.retainAll(colB); //objetivo ahora contiene [1,2,3]
objetivo.removeAll(colA); //no pasa nada, los elementos ya se habían quitado
objetivo.removeAll(colB); //objetivo ahora está vacía
```

La interfaz Collection tiene dos métodos para revisar si una colección contiene uno o más elementos. Estos métodos son cantains() y containsAll(). A continuación se ilustran:

```
//Entorno:
Collection coleccion,elementos;
boolean contieneElemento,contienTodos;
//Algoritmo:
coleccion = new HashSet();
contieneElemento = coleccion.contains("un elemento");
elementos = new HashSet();
contieneTodos = collection.containsAll(elementos);
```

- **contains()** regresa verdadero si la colección incluye el elemento y falso si no es así.
- **containsAll()** regresa verdadero si la colección contiene todos los elementos de la colección argumento y falso si no es así.

Se puede revisar el tamaño de una colección utilizando el método size(). El tamaño de una colección indica el número de elementos que contiene. Un ejemplo:

```
//Entorno:
int numeroElemntos;
//Algoritmo:
numeroElementos = coleccion.size();
```

También se puede utilizar el nuevo ciclo for:

```
//Entorno:
Collection coleccion;
//Algoritmo:
coleccion = new HashSet();
//... agrega elementos a la colección
for(Object objecto : coleccion) {
   //hacer algo con objecto;
}//Fin Para
```

Es posible generalizar los varios tipos y subtipos de Colecciones (Collection) y Mapas (Map) de la siguiente manera:

```
//Entorno:
Collection<String> coleccionCadenas;
//Algoritmo:
coleccionCadenas = new HashSet<String>();
```

Esta “coleccionCadenas” ahora únicamente puede contener instancias u objetos del tipo String.

Si se trata de agregar cualquier otra cosa, o hacer un cast de los elementos en la colección a cualquier otro tipo que no sea String, se arrojará un error de compilación. Se puede iterar la colección mostrada anteriormente utilizando el nuevo ciclo for:

```
//Entorno:
Collection<String> coleccionCadenas;
//Algoritmo:
coleccionCadenas = new HashSet<String>();
for(String cadena : coleccionCadenas) {
   //hacer algo con cada "cadena"
}//Fin Para
```

Listas (*List*)

Las listas representan una lista ordenada de objetos, lo cual quiere decir que se puede acceder a los elementos en un orden específico o mediante un índice (*index*). En una lista se puede agregar el mismo elemento más de una vez.

Implementaciones de Listas

Al ser un subtipo de la interfaz Collection, todos los métodos de Collection también se encuentran disponibles en la interfaz Lista (*List*).

Como List es una interfaz, es necesario instanciar una implementación concreta de la interfaz para poder utilizarla. Existen varias implementaciones de la interfaz List en el API de Java:

- **java.util.ArrayList**: vector dinámico, no sincronizable (hay que hacerlo a mano) crece el 50% de overflow.
- **java.util.LinkedList**: lista doblemente enlazada.
- **java.util.Vector**: vector dinámico, sincronizable crece el DOBLE de overflow.
- **java.util.Stack**: hereda de Vector.

```
//Entorno:
List listaA,listaB,listaC,listaD;
//Algoritmo:
listaA = new ArrayList();
listaB = new LinkedList();
listaC = new Vector();
listaD = new Stack();
```

Para agregar elementos a un lista se llama su método *add()*. Este método es heredado de la interfaz Collection. Algunos ejemplos:

```
//Entorno:
List lista;
//Algoritmo:
listaA = new ArrayList();
listaA.add("elemento 1");
listaA.add("elemento 2");
listaA.add("elemento 3");
listaA.add(0, "elemento 0");
```

El orden en el que se agregan los elementos a la lista es almacenado, de manera que se puede acceder a dichos elementos en el mismo orden. Para esto, se puede utilizar el método *get(índice)* o a través de un iterador devuelto por el método *iterator()*:

```
//Entorno:
      List<String> miArray;
      Iterator<String> iterador;
      String cadena;
      int i;
//Algoritmo:
      miArray = new ArrayList(); //crea lista
```

```
   //carga lista
      miArray.add("primero");
      miArray.add("segundo");
      miArray.add(1, "tercero");
      miArray.add(3, "cuarto");
   //recorre con interador
      iterador = miArray.iterator();
      while (iterador.hasNext()) {
         cadena = iterador.next();
         System.out.println(cadena);
      }//Fin Mientras
   //recorre con for extendido
      System.out.println("Otra forma");
      for (String elemento : miArray) {
         System.out.println(elemento);
      }//Fin para
   //recorre con for
      for (i = 0; i < miArray.size(); i++) {
         System.out.println(miArray.get(i));
      }//Fin Para
```

Se pueden quitar elementos de dos formas:

- **remove(Object elemento)** elimina el elemento de la lista si está presente. Todos los elementos de la lista se mueven un lugar hacia arriba de manera que, todos sus índices se reducen en 1.
- **remove(int indice)** elimina el elemento indicado por el índice dado. Todos los elementos de la lista se mueven un lugar hacia arriba de manera que, todos sus índices se reducen en 1.

Listas Genéricas

Por defecto se puede poner cualquier tipo de objeto en una lista, pero a partir de Java 5 (la versión actual es la 7), "Java Generics" hace posible limitar los tipos de objeto que se pueden insertar en una lista, por ejemplo:

```
//Entorno:
List<MyObject> lista;
//Algoritmo:
lista = new ArrayList<MyObject>();
```

Esta lista ahora sólo puede tener instancias de la clase MyObject dentro de ella. Se puede entonces acceder e iterar estos elementos sin necesidad de hacer un casting:

```
//Entorno:
MyObject unObjeto;
//Algoritmo:
unObjeto = lista.get(0);
for(MyObject unObjeto : lista){
   //hacer algo con anObjeto...
}
```

Pilas (Stack)

La clase Stack (java.util.Stack) merece una breve explicación propia. El uso típico de una Pila o Stack no es como el de una lista común.

Una pila es una estrucutra de datos en la cual se agregan elementos a la "cima" de la pila, y también se quitan elementos de la "cima". A este enfoque se le llama "Último que entra, primero que sale" o LIFO *(Last in, First out).* Al contrario que una cola (queue) donde "Primero que entra, primero que sale" o FIFO ("First in, First out").

Se muestra un ejemplo para crear un Stack:

```
//Entorno:
Stack pila;
Object objTop,obj3,obj2,obj1;
//Algoritmo:
pila = new Stack();
pila.push("1");
pila.push("2");
pila.push("3");
//observar el elemento en el tope de la pila sin sacarlo.
objTop = pila.peek();
obj3 = pila.pop(); //la cadena "3" está en la cima de la pila. La saca.
obj2 = pila.pop(); //la cadena "2" está en la cima de la pila. La saca.
obj1 = pila.pop(); //la cadena "1" está en la cima de la pila. La saca.
```

- **push()** "empuja" un objeto al tope del stack.
- **peek()** devuelve el objeto en el tope del stack pero sin sacarlo.
- **pop()** devuelve el objeto en el tope del stack, eliminándolo.

Se puede buscar un objeto dentro de la pila para obtener su índice utilizando el método search(). Se llama al método equals() de cada objeto de la pila para determinar si el objeto buscado está presente en la misma. El índice que se obtiene es el índice a partir de la cima de la pila, lo que significa que un elemento con índice 1 se encuentra en la cima:

```
//Entorno:
Stack pila;
int index;
//Algoritmo:
pila = new Stack();
pila.push("1");
pila.push("2");
pila.push("3");
index = pila.search("3"); //index = 3
```

Queue (Colas)

La interfaz Queue (java.util.Queue) es un subtipo de la interfaz Collection, representa una lista ordenada de objetos justo como List, pero su uso es ligeramente distinto. Una cola está diseñada para que sus elementos sean insertados al final de la cola y eliminados por el principio.

Al ser un subtipo de Collection, todos los métodos de Collection también se encuentran disponibles en la interfaz Queue.

Como Queue es una interfaz, es necesario instanciar una implementación concreta para poder utilizarla. Existen 2 clases en el API de Java que implementan la interfaz Queue:

- java.util.LinkedList
- java.util.PriorityQueue

- **LinkedList** es una implementación estándar de una cola.
- **PriorityQueue** guarda sus elementos internamente de acuerdo a su orden natural (si implementan la interfaz Comparable), o de acuerdo a un Comparador (Comparator) pasado a PriorityQueue.

Aquí hay algunos ejemplos de cómo crear una instancia de Queue:

```
//Entorno:
Queue colaA,colaB;
//Algoritmo:
colaA = new LinkedList();
colaB = new PriorityQueue();
```

Para agregar elementos a una cola, se llama su método *add()*. Este método se hereda de la interfaz Collection:

```
colaA.add("elemento 1");
colaA.add("elemento 2");
colaA.add("elemento 3");
```

El orden en el que los elementos se agregan a Queue es almacenado internamente y depende de la implementación. Esto mismo es cierto para el orden en el cual los elementos son obtenidos (eliminados) de la cola.
Se puede observar cuál es el elemento que se encuentra a la "cabeza" de la cola sin quitarlo utilizando el método element():

```
//Entorno:
Object primerElemento;
//Algoritmo:
firstElement = colaA.element();
```

Para quitar el primer elemento de la cola, se utiliza el método *remove()*. También es posible iterar todos los elementos de la cola, en lugar de procesarlos uno a la vez:

```
//Entorno:
Queue colaA;
Iterator iterador;
String elemento;
//Algoritmo:
colaA = new LinkedList();
colaA.add("elemento 0");
colaA.add("elemento 1");
colaA.add("elemento 2");
//acceso con iterador
iterador = colaA.iterator();
while(iterador.hasNext(){
   elemento = (String) iterador.next();
}
//acceso con ciclo for
for(Object objecto : colaA) {
   elemento = (String) objecto;
}
```

Para eliminar (quitar) elementos de la cola, se llama el método *remove()*. Éste método quita el elemento que se encuentra a la "cabeza" de la cola:

```
primerElemento = colaA.remove();
```

Colas Genéricas

Por defecto, se puede poner cualquier tipo de objeto dentro de una cola, pero a partir de Java 5, es posible limitar el tipo de objetos que se pueden insertar en Queue:

```
//Entorno:
Queue<MyObject> cola;
//Algoritmo:
cola = new LinkedList<MyObject>();
```

Esta cola únicamente podrá tener instancias MyObject dentro de ella. Se puede acceder e iterar los elementos sin realizar casting:

```
//Entorno:
MyObject unObjecto;
```

```
//Algoritmo:
unObjecto = cola.remove();
for(MyObject unObjeto : cola){
   //hacer algo con unObjecto...
}
```

Deque

La interfaz Deque (java.util.Deque) es un subtipo de la interfaz Queue. Representa un tipo de cola en la cual se pueden insertar y eliminar elementos de ambos lados. Por lo tanto, "Deque" es la versión corta de "Double Ended Queue" o Cola de 2 lados.
Al ser un subtipo de Queue, todos los métodos de Queue y Collection se encuentran disponibles en Deque.
Como Deque es una interfaz, es necesario instanciar una implementación concreta de la interfaz para poder utilizarla. Existen 2 clases en el API de Java que implementan la interfaz Deque:

- java.util.ArrayDeque
- java.util.LinkedList

- **LinkedList** es una implementación estándar de Queue y Deque.
- **ArrayDeque** almacena sus elementos internamente en un arreglo. Si el número de elementos excede el espacio en el array, se crea uno nuevo y todos los elementos se copian de uno al otro.

Ejemplos sobre cómo instanciar un Deque:

```
//Entorno:
Deque dequeA,dequeB;
//Algoritmo:
dequeA = new LinkedList();
dequeB = new ArrayDeque();
```

Para agregar elementos a la "cola" del Deque se utiliza el método add(). También se pueden utilizar los métodos addFirst() y addLast() para agregar elementos a la "cabeza" y la "cola" del Deque respectivamente.

```
dequeA = new LinkedList();
dequeA.add ("elemento 1"); //agregar elemento al final
dequeA.addFirst("elemento 2"); //agregar elemento a la cabeza
dequeA.addLast ("elemento 3"); //agregar elemento al filnal
```

El orden en el cual se agregan los elementos al Deque se almacena internamente y depende de la implementación. Las dos implementaciones mencionadas anteriormente almacenan sus elementos en el orden en el que fueron insertados.
Se puede observar el elemento que se encuentra a la cabeza del Deque sin quitarlo utilizando el método element(). Adicionalmente, se pueden utilizar los métodos getFirst() y getLast(), que devuelven el primer y el último elemento del Deque:

```
//Entorno:
Object primerElemento,ultimoElemento;
//Algoritmo:
primerElemento = dequeA.element();
primerElemento = dequeA.getFirst();
ultimoElemento = dequeA.getLast();
```

También se pueden iterar los elementos del Deque, en lugar de procesarlos uno a la vez:

```
//Entorno:
Deque dequeA;
Iterator iterador;
String elemento;
//Algoritmo:
```

```
dequeA = new LinkedList();
dequeA.add("elemento 0");
dequeA.add("elemento 1");
dequeA.add("elemento 2");
//acceso con iterador
iterador = dequeA.iterator();
while(iterador.hasNext(){
   elemento = (String) iterador.next();
}
//acceso con ciclo for
for(Object objecto : dequeA) {
   elemento = (String) objecto;
}
```

Cuando se itera el Deque mediante su iterador o mediante el ciclo for, la secuencia en la que se iteran los elementos depende de la implementación.
Para eliminar elementos de un Deque, se utilizan los métodos remove(), removeFirst() y removeLast():

```
//Entorno:
Object primerElemento,ultimoElemento;
//Algoritmo:
primerElement = dequeA.remove();
primerElement = dequeA.removeFirst();
ultimoElement = dequeA.removeLast();
```

Deque Genérico

Por defecto, se puede poner cualquier tipo de objeto dentro de un Deque, pero a partir de Java 5, es posible limitar el tipo de objetos que se pueden insertar en Deque:

```
//Entorno:
Deque<MyObject> deque;
//Algoritmo:
deque = new LinkedList<MyObject>();
```

Este Deque únicamente podrá tener instancias MyObject dentro de él. Se puede acceder e iterar los elementos sin realizar casting:

```
//Entorno:
Object unObjecto;
//Algoritmo:
unObjecto = deque.remove();
for(MyObject unObjeto : deque){
   //hacer algo con unObjecto...
}
```

Set (Conjuntos)

La interface Set representa un conjunto de objetos, lo que significa que cada elemento puede existir solamente una vez en el Set.

Implementaciones de Set

Al ser un subtipo de Collection, todos los métodos de Collection se encuentran disponibles en Set.
Como Set es una interface, es necesario instanciar una implementación concreta de la interfaz para poder usarla. Existen varias clases en el API de Java que implementan la interfaz Set:

- java.util.EnumSet
- java.util.HashSet
- java.util.LinkedHashSet

- java.util.TreeSet

```
//Entorno:
Set setA,setB,setC,setD;
//Algoritmo:
setA = new EnumSet();
setB = new HashSet();
setC = new LinkedHashSet();
setD = new TreeSet();
```

Cada una de estas implementaciones se comporta de manera ligeramente distinta respecto al orden de los elementos cuando se itera el Set, y en el tiempo que toma el insertar y agregar elementos a los sets.

- **HashSet** es respaldado por un HashMap. No garantiza la secuencia de los elementos cuando éstos son iterados.
- **LinkedHashSet** difiere de un HashSet en que garantiza el orden de los elementos durante la iteración, es el mismo orden en el cual fueron insertados. Reinsertar un elemento que ya se encontraba en el LinkedHashSet no cambia su orden.
- **TreeSet** también garantiza el orden de los elementos al iterarlos, pero el orden es el orden de ordenamiento de los elementos. En otras palabras, el orden en el cual dichos elementos se almacenarían si se utilizara el método Collections.sort() en una lista o arreglo que contenga dichos elementos. Este orden es determinado por su orden natural (si implementan la interfaz Comparable) o mediante un comparador (Comparator) específico para la implementación.

Cuando se iteran los elementos de un Set el orden de los elementos depende de cuál implementación de Set se utilice como se mencionó anteriormente. Un ejemplo de uso:

```
//Entorno:
Set setA;
Iterator iterador;
String elemento;
//Algoritmo:
setA = new HashSet();
setA.add("elemento 0");
setA.add("elemento 1");
setA.add("elemento 2");
//acceso mediante iterador
iterador = setA.iterator();
while(iterador.hasNext(){
   elemento = (String) iterador.next();
}
//acceso mediante ciclo for
for(Object objecto : setA) {
   elemento = (String) objecto;
}
```

Los elementos se eliminan llamando al método remove(Object o). No hay manera de eliminar un objeto mediante un índice en un Set ya que el orden de los elementos depende de la implementación.

Sets Genéricos

Por defecto se puede almacenar cualquier tipo de objeto en un Set, sin embargo, es posible limitar el tipo de objetos que se pueden insertar mediante el uso de genéricos.

Un ejemplo:

```
//Entorno:
Set<MyObject> set;
//Algoritmo:
set = new HashSet<MyObject>();
```

Este Set ahora únicamente puede tener instancias MyObject dentro de él. Se puede acceder e iterar sus elementos sin realizar casting:

```
for(MyObject anObject : set){
   //do someting to anObject...
}
```

Sorted Set (Sets ordenados)

La intefaz SortedSet (java.util.SortedSeet) es un subtipo de la interfaz Set. Se comporta como un set normal con la excepción de que los elementos se ordenan internamente. Esto significa que cuando se iteran los elementos de un SortedSet los elementos se devuelven de manera ordenada.

La ordenación es la ordenación natural de los elementos (si implementan java.lang.Comparable), o el orden determinado por un Comparator que se le puede proporcionar al SortedSet.

Por defecto, los elementos son iterados en orden ascendente, empezando con el "más chico" y moviéndose hacia el "más grande". Pero también es posible iterar los elementos en orden descendente utilizando el método descendingIterator().

En las API de Java existe únicamente una clase que implementa SortedSet: java.util.TreeSet.

Un par de ejemplos sobre cómo instanciar un SortedSet:

```
//Entorno:
SortedSet setOrdenadoA,setOrdenadoB;
Comparator comparador;
//Algoritmo:
setOrdenadoA = new TreeSet();
comparador = new MyComparator();
setOrdenadoB = new TreeSet(comparador);
```

Clase Collections

La clase Collections (no confundir con la interfaz Collection) consiste exclusivamente en métodos estáticos que operan sobre colecciones. Contiene algoritmos polimórficos que operan sobre colecciones, los cuales regresan una nueva colección respaldada por la colección original.

Los métodos de esta clase arrojan NullPointerException si las colecciones proporcionadas a los métodos como argumentos son null.

Los algoritmos "destructivos" contenidos en esta clase, es decir, los algoritmos que modifican la colección sobre la cual operan, arrojan UnsuportedOperationException si la colección no soporta la mutación apropiada de los primitivos.

Algunos ejemplos de métodos de la clase Collections:

- void copy(List, List)
- boolean disjoint(Collection, Collection)
- Object max(Collection)
- void reverse(List)
- void sort(List)

Algunos ejemplos de la utilización de los métodos de la clase Collections:

```
//Entorno:
List lista,lista2;
//Algoritmo:
lista = new LinkedList();
lista.add("Juan");
lista.add("Luis");
lista.add("Adrian");
lista.add("Cheko");
lista.add("Rodolfo");
Collections.sort(lista);
lista2 = new LinkedList();
Collections.copy(lista, lista2);
```