**2014 Big Java**

Contenido

[SET 1](#_Toc507841565)

[Eligiendo una implementación de Set 1](#_Toc507841566)

[HashSet (Hash Table, Hash Code) 1](#_Toc507841567)

[TreeSet (Binary Search Trees) 2](#_Toc507841568)

[Trabajando con Sets 3](#_Toc507841569)

# SET

* Una colección set organiza sus valores/elementos en un orden que esta optimizado para eficiencia, dicho orden puede no ser en el cual agregamos los elementos. **Insertar y remover elementos es más eficiente con un set que con una lista.**

## Eligiendo una implementación de Set

* La interface **Set** de la librería estándar de Java tiene los mismos métodos que la interface **Collection**. Sin embargo, existe una diferencia esencial entre colecciones arbitrarias y sets. Una colección set no permite duplicados. Si se agrega un elemento a la colección set que ya existe, la inserción es ignorada.
* Las clases **HashSet** y **TreeSet** implementan la interface **Set**.
* Estás dos clases proporcionan implementaciones set basadas en dos diferentes mecanismos, llamados **hash tables** y **árboles de búsqueda binaria.** Ambas implementaciones organizan los elementos set (establecidos) de tal manera que la búsqueda, agregación y eliminación de elementos sea eficiente, pero utilizan diferentes estrategias.
* Las implementaciones de Set organizan los elementos de manera que puedan localizarlos rápidamente.

### HashSet (Hash Table, Hash Code)

* La idea básica de un **Hash Table** es simple. Los elementos de un Set son agrupados en pequeñas colecciones de elementos que comparten mismas características. Podemos imaginar un Hash Set de libros como tener un grupo para cada color, de tal manera que los libros del mismo color están en el mismo grupo. Para saber si un libro ya está presente, solo verificarlo con los libros en el mismo grupo de colores. Las Hash Tables no utilizan colores, utilizan valores enteros (llamados **hash codes**) que pueden ser computados/calculados a partir de los elementos.
* Para usar un Hash Table, los elementos deben tener un método para calcular esos valores enteros. Este método se llama **hasCode.** Los elementos deben pertenecer a una clase con un método **equals** correctamente definido.
* Varias clases en la librería estándar implementan estos métodos, por ejemplo, String, Integer, Double, Punto, Rectangulo, Color y todas las clases pertenecientes a Colecciones. Por lo tanto, podemos formar un HashSet<String>, HashSet<Rectangulo> o incluso HashSet<HashSet<Integer>>>
* Supongamos que queremos formar un set de elementos pertenecientes a una clase que nosotros declaramos, como HashSet<Libro>. Entonces necesitamos proporcionar el método **hashCode** y **equals** a la clase Libro. No hay excepción para esta regla. Si todos los elementos son distintos (por ejemplo, si nuestro programa nunca tiene dos objetos Libro con el mismo título y autor), entonces podemos simplemente heredar los métodos **hashCode** y **equals** de la clase **Object**.
* En este estante, se agrupan libros del mismo color. Del mismo modo, en una Hash Table, los objetos con el mismo hash code se colocan en el mismo grupo.

|  |
| --- |
|  |

### TreeSet (Binary Search Trees)

* Un Tree Set mantiene sus elementos ordenados.

|  |
| --- |
|  |

* La clase **TreeSet** utiliza una estrategia diferente para organizar sus elementos. Los elementos se mantienen ordenados. Por ejemplo, un conjunto (set) de libros puede estar organizado por peso, alfabéticamente por el nombre del autor y título. Los elementos no son guardados en un array/arreglo, eso haría que la agregar y eliminar elementos sea demasiado ineficiente. En lugar de eso, son almacenados en nodos, como en una Linked List. Sin embargo, los nodos no están organizados en una sequencia lineal sino en forma de árbol.
* Para usar un TreeSet, debe ser posible comparar los elementos y determinar cuál es “más grande”. Podemos utilizar un TreeSet con clases como String e Integer que implementan la interface **Comparable.**
* Podemos formar TreeSet’s con cualquier clase que implemente la interface **Comparable**, tales como String e Integer.
* Como regla general, se elige un TreeSet si queremos visitar los elementos del conjunto (set) de manera ordenada. De lo contrario, se elige un HashSet, siempre que la función hash sea bien elegida, es un poco más eficiente.
* Cuando se construye un HashSet o TreeSet, se guarda la referencia en una variable Set, pudiendo ser:

|  |
| --- |
| Set<String> nombres = **new** HashSet<>();  Set<String> nombres = **new** TreeSet<>(); |

* Después de que construimos el objeto de la colección, la implementación ya no importa, solo la interfaz es importante.

## Trabajando con Sets

* Los Sets no tienen duplicados. Si agregamos un elemento que ya está presente en el conjunto, es ignorado.
* Agregar y eliminar elementos se logra con los métodos **add** y **remove.**

|  |
| --- |
| nombres.add(“Romeo”);  nombres.remove(“Juliet”); |

* Una colección Set en Java rechaza los duplicados. Agregar un elemento no tiene efecto si el elemento ya está en el conjunto. Del mismo modo, se ignora el intento de eliminar un elemento que no está en el conjunto.
* El método **contains** verifica si un elemento está o no en el conjunto (set)

|  |
| --- |
| If (nombres.contains(“Juliet”)) … |

* El método **contains** utiliza el método **equals** del tipo de elemento. Si nuestro conjunto colecciona objetos String o Integer, no tenemos que preocuparnos. Esas clases proporcionan el método **equals.** Sin embargo, si implementamos el tipo de elemento nosotros mismos con alguna clase que declaremos, entonces tenemos que definir el método **equals**.
* Finalmente, para listar todos los elementos en el conjunto (set), tenemos que obtener un iterador (**iterator**). Así como los iteradores de las listas, usamos los métodos **next** y **hasNext** para atravesar el conjunto (set).

|  |
| --- |
| Iterator<String> iterador = nombres.iterator();    **while** (iterador.hasNext()) {  String nombre = iterador.next();  System.***out***.println(nombre);  } |

* También podemos utilizar el bucle **for each** en lugar de usar un iterador.

|  |
| --- |
| **for** (String nombre : nombres) {  System.***out***.println(nombre);  } |

* Un iterador de conjunto, visita los elementos en el orden en que la implementación del conjunto los mantiene/guarda. Este no es necesariamente el orden en el cual insertamos los elementos. El orden de los elementos en un **Hash Set** parece bastante aleatorio porque el hash code propaga los elementos en diferentes grupos. Cuando visitamos los elementos de un **Tree Set,** estos siempre aparecen ordenados, incluso si los insertamos en diferente orden.
* Existe una importante diferencia entre **Iterator** que obtenemos de un **Set** y **ListIterator** que da una lista. El **ListIterator** tiene un método **add** para agregar un elemento en la posición del iterador de la lista. La interface **Iterator** no tiene dicho método. No tiene sentido agregar un elemento en una posición en particular en un conjunto (set), porque el conjunto ordena los elementos de la forma que quiera. Por lo tanto, siempre se agregan elementos directamente al conjunto (set), nunca a un iterador del conjunto.
* No puedes agregar un elemento a un cojunto en una posición del iterador.
* Sin embargo, podemos remover un elemento de un conjunto en una posición del iterador, justo como lo hacemos con los iteradores de lista (List Iterators).
* Además, la interface **Iterator** no tiene el método **previous** para retroceder a través de los elementos.
* Ya que los elementos no están ordenados, no es significativo distinguir entre “ir hacia adelante” y “ir hacia atrás”