Java Collections Framework

Colecciones

- **Colección**. Un objeto que almacena datos; también conocido como "estructura de datos."
- Los objetos almacenados se llaman *elementos*.
- Algunas colecciones mantienen un orden (índice); algunas permiten duplicados.
- Operaciones típicas: agregar un elemento, eliminar un elemento, buscar un elemento, vaciar la colección, tamaño de la colección.
- Ejemplos de colecciones en Java: List, Set, Map, Stack, Queue.

Colecciones

- Las colecciones están en el paquete java.util.

```
import java.util.*;
```

- Información oficial: https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/doc-files/coll-overview.html

Java Collections Framework

General purpose implementations

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

- https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/doc-files/coll-overview.html

Listas

- **Lista**. Una colección que almacena una secuencia de elementos con un orden.
- Cada elemento es accesible por un índice que comienza en 0.
- Las listas tienen un tamaño (número de elementos).
- Los elementos se pueden agregar al frente, atrás o en cualquier otro lugar.
- Por default, el elemento se agrega al final de la lista.
- Los elementos se pueden eliminar al frente, atrás o en cualquier otro lugar.

Listas

- Las lista se pueden pensar como arreglos dinámicos (cambian automáticamente de tamaño).

Métodos de List

add(valor)	Agrega el valor al final de la lista
add(índice, valor)	Agrega el valor antes del índice, moviendo los valores siguientes a la derecha
clear()	Borra todos los elementos de la lista
indexOf(value)	Regresa el índice donde está el valor o -1 si no está en la lista
get(índice)	Regresa el valor en el índice indicado
remove(índice)	Borra el valor en el índice indicado
set(índice, valor)	Reemplaza el valor en el índice indicado con el valor dado
size()	Regresa el número de elementos en la lista
toString()	Regresa una representación de la lista como string

Métodos de List

addAll(lista) addAll(índice, lista)	Agrega todos los elementos de la lista dada a esta lista (al final de la lista, o los inserta en el índice dado)
contains(valor)	Devuelve true si el valor dado se encuentra en algún lugar de esta lista
containsAll(lista)	Devuelve true si esta lista contiene todos los elementos de la lista dada
equals(lista)	Devuelve true si esta lista contiene los mismos elementos que la lista dada
iterator() listIterator()	Devuelve un objeto iterador para examinar el contenido de la lista
lastIndexOf(valor)	Devuelve el último índice donde se encuentra el valor o -1 si no se encuentra
remove(valor)	Encuentra y elimina el valor dado de esta lista
removeAll(lista)	Elimina los elementos de esta lista que se encuentran en la lista dada
retainAll(lista)	Elimina los elementos de esta lista que *no* se encuentran en la lista dada
subList(desde, hasta)	Devuelve los elementos entre los índices desde (inclusive) y hasta (exclusivo)
toArray()	Devuelve los elementos de esta lista como un arreglo

Implementación de una lista

- List es una interface.
- Por lo tanto, no se puede crear una lista mediante new List().
- Se tiene que buscar una clase que implemente la interface List.
- En esta curso se ven dos clases: ArrayList y LinkedList.
- Las diferencias se analizan un poco más adelante.

Parámetros de tipo (genéricos)

- Ejemplo:

```
List<Tipo> nombre = new ArrayList<>();
```

- Al construir una lista se debe especificar el tipo de los elementos.
- Esto se denomina *parámetro de tipo* o clase *genérica*.
- Permite tener listas de tipos distintos, por ejemplo:

```
List<String> nombres = new ArrayList<>();
nombres.add("Ana");
nombres.add("Blanca");
```

Ordenamiento de una lista

- Una lista de strings se puede ordenar usando el método Collections.sort().

```
List<String> p = Arrays.asList("Mercurio", "Venus", "Tierra", "Marte",
"Júpiter", "Saturno", "Urano", "Neptuno");
Collections.sort(p);
System.out.println(p);  // [Júpiter, Marte, Mercurio, Neptuno, Saturno,
Tierra, Urano, Venus]
Collections.sort(p, Collections.reverseOrder());
System.out.println(p);  // [Venus, Urano, Tierra, Saturno, Neptuno,
Mercurio, Marte, Júpiter]
```

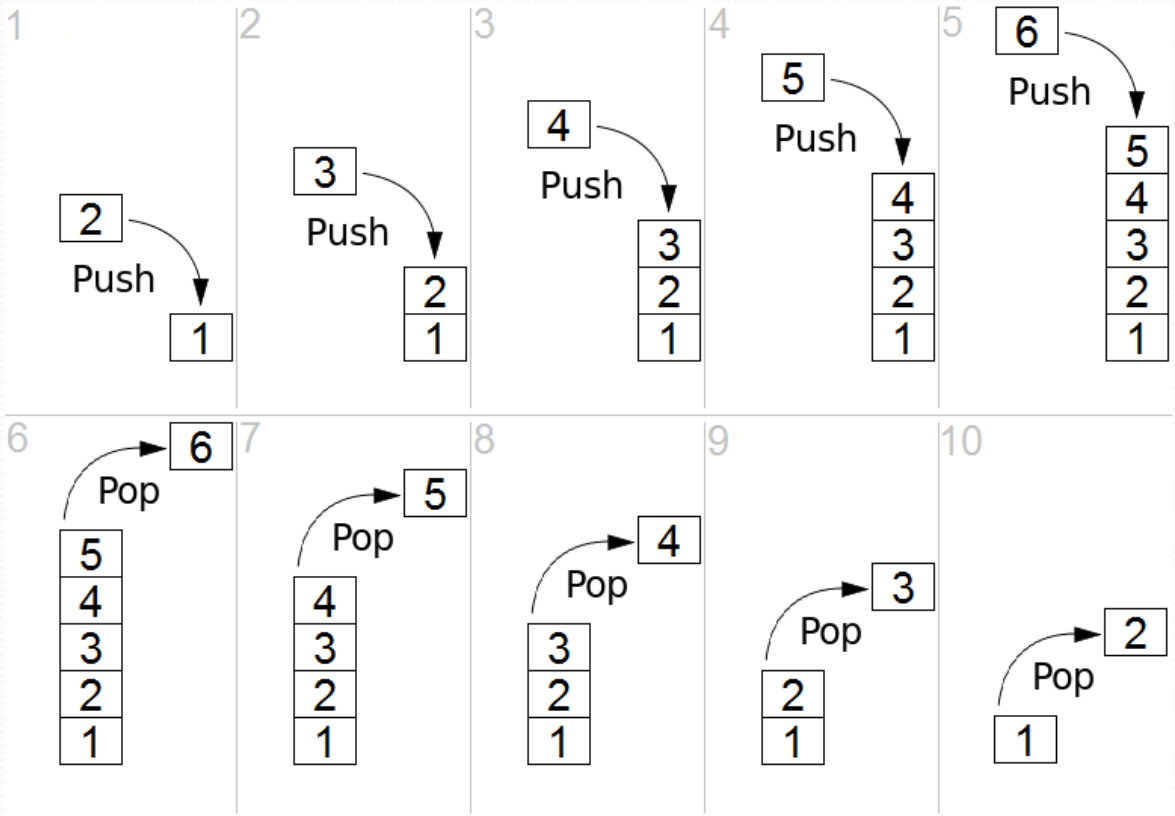
Ordenamiento de una lista

- Una lista de objetos de cierta clase se puede ordenar siempre y cuando la clase implemente la interface Comparable.

Pilas y colas

- A veces es bueno tener una colección que sea menos poderosa, pero que esté optimizada para realizar ciertas operaciones rápidamente.
- Dos colecciones especializadas:
- **Pila**. Elimina elementos en el orden inverso al que se agregaron.
- **Cola**. Elimina elementos en el mismo orden en que se agregaron.

Pila



By Maxtremus - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=44458752

Explicación

- Último en entrar, primero en salir (LIFO – last in, first out).
- Los elementos se almacenan en orden de inserción.
- El cliente solo puede agregar / quitar / examinar el último elemento agregado (el tope de la pila).
- Operaciones básicas de pila:
  - push: agrega un elemento en la parte superior.
  - pop: elimina el elemento superior.
  - peek: examina el elemento superior.

Clase Stack

Stack<E>()	Construye una nueva pila con elementos de tipo E
push(valor)	Agrega el valor dado en el tope de la pila
pop()	Elimina el valor en el tope de la pila y lo devuelve; lanza EmptyStackException si la pila está vacía
peek()	Devuelve el valor en el tope de la pila sin eliminarlo; lanza EmptyStackException si la pila está vacía
size()	Devuelve el número de elementos en la pila
isEmpty()	Devuelve true si la pila está vacía

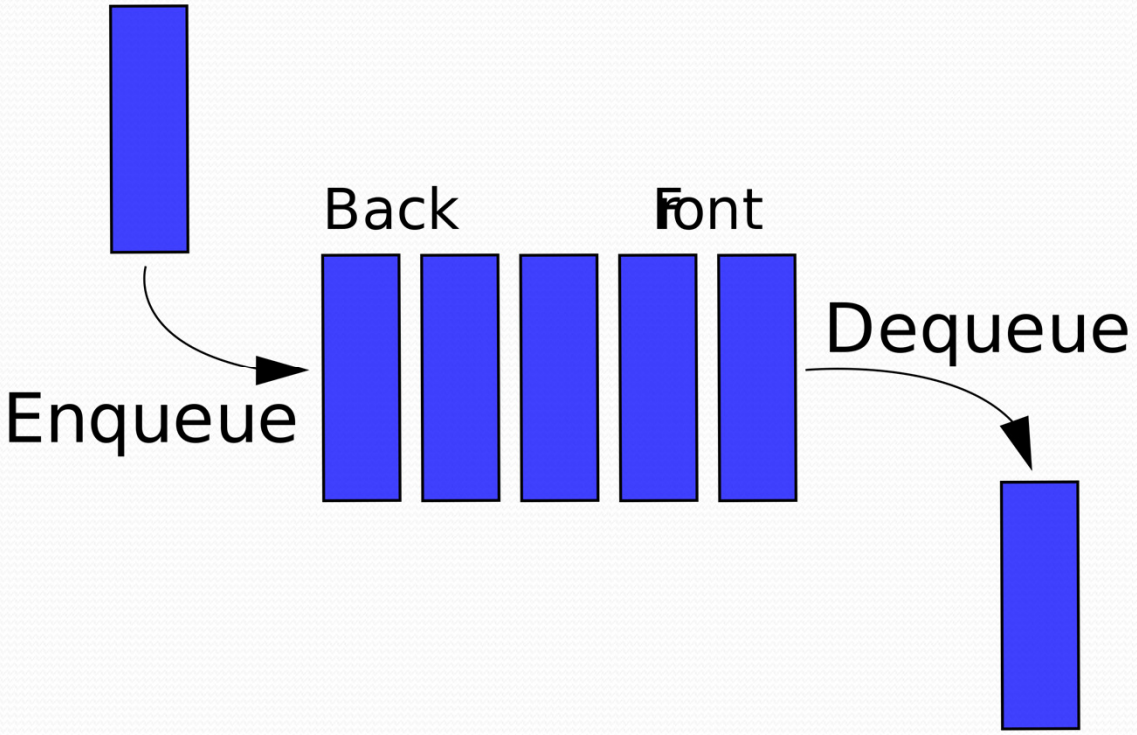
Ejemplo 1

```
Stack<Integer> s = new Stack<>();
s.push(42);          // bottom [42] top
s.push(-3);          // bottom [42, -3] top
s.push(17);          // bottom [42, -3, 17] top
System.out.println(s.pop()); // 17 bottom [42, -3] top
```

Ejemplo 2

```
// Ordenar un arreglo de números de mayor a menor
int[] numbers = {…}
Arrays.sort(numbers);    // de menor a mayor
Stack<Integer> stack = new Stack<>();
for (int n: numbers) {    // entran de cabeza
    stack.push(n);
}
for (int i = 0; i < numbers.length; i++) {    // salen al revés
        numbers[i] = stack.pop();
}
```

Cola



By User:Vegpuff. Own work, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=7586271

Explicación

- Primero en entrar, primero en salir (FIFO – first in, first out).
- Los elementos se almacenan en el orden de inserción.
- El cliente solo puede agregar elementos al final de la cola y solo puede examinar / eliminar el elemento en el frente de la cola.
- Operaciones de cola básicas:
  - add (poner en cola): añade un elemento al final.
  - remove (sacar de la cola): retira el elemento al frente.
  - peek: examina el elemento frontal.

Interface Queue

add(valor)	Inserta el valor al final de la cola
remove()	Elimina el elemento del frente de la cola y lo devuelve; si la cola está vacía lanza NoSuchElementException
peek()	Devuelve el elemento del frente de la cola sin eliminarlo; devuelve null si la cola está vacía
size()	Devuelve el número de elementos en la cola
isEmpty()	Devuelve true si la cola está vacía

Ejemplo

```
Queue<Integer> q = new LinkedList<>();
q.add(42);          // front [42] back
q.add(-3);          // front [42, -3] back
q.add(17);          // front [42, -3, 17] back
System.out.println(q.remove()); // 42 front [-3, 17] back
```

- Queue es una interface y para crear una cola se debe buscar una clase que implemente esta interface.
- LinkedList es una clase que implementa la interface Queue.

Tipos de datos abstractos

- **Tipo de datos abstractos (ADT)**. Es una especificación de una colección de datos y las operaciones que se pueden realizar en ella.
- Describe *que* hace una colección, no *cómo* lo hace.
- No se sabe exactamente cómo se implementa una pila o una cola y no es necesario.
- Solo se necesita entender la idea de la colección y qué operaciones puede realizar.
- Ejemplo: el tipo de datos abstracto pila.
- Es una secuencia de objetos con comportamiento LIFO.
- Operaciones: push, pop, seek, size, etc.

ADTs como interfaces

- El framework de colecciones de Java utiliza interfaces para describir ADTs:
- Collection, Deque, List, Map, Queue, Set.
- Un ADT se puede implementar de varias formas por clases:
- ArrayList y LinkedList implementan List.
- HashSet y TreeSet implementan Set.
- LinkedList, ArrayDeque, etc. implementan Queue.
- Confusamente, Stack no es una interface, es una clase.

ADTs como interfaces

- Se considera una buena práctica declarar las variables de colección utilizando el tipo de interface ADT correspondiente:

```
List<String> lista = new ArrayList<>();
```

- Los métodos que aceptan una colección como parámetro también deben declarar el parámetro utilizando el tipo de interface ADT:

```
public void foo(List<String> list)
{
    ...
}
```

Implementaciones de ADTs

- ¿Por qué hay más de un tipo de lista, cola, etc.?
- Respuesta: cada implementación es más eficiente en determinadas tareas.
- ArrayList es más rápido para agregar / eliminar al final.
- LinkedList es más rápido para agregar / eliminar en el frente / en el medio.
- Uno elige la mejor implementación para su tarea y, si el resto del código está escrito para usar las interfaces, funcionará.

ADTs en Java

- Listas (interface List).
- Pilas (clase Stack).
- Colas (interface Queue).
- Conjuntos (interface Set).
- Mapas (interface Map).

Conjuntos

- **Conjunto**. Una colección de valores sin duplicados.
- Operaciones:
- Elementos: agregar, eliminar, buscar.
- Conjuntos: tamaño, unión, intersección, diferencia.
- Los conjuntos no tienen índices.
- Los elementos se agregan a un conjunto y no existe un orden.

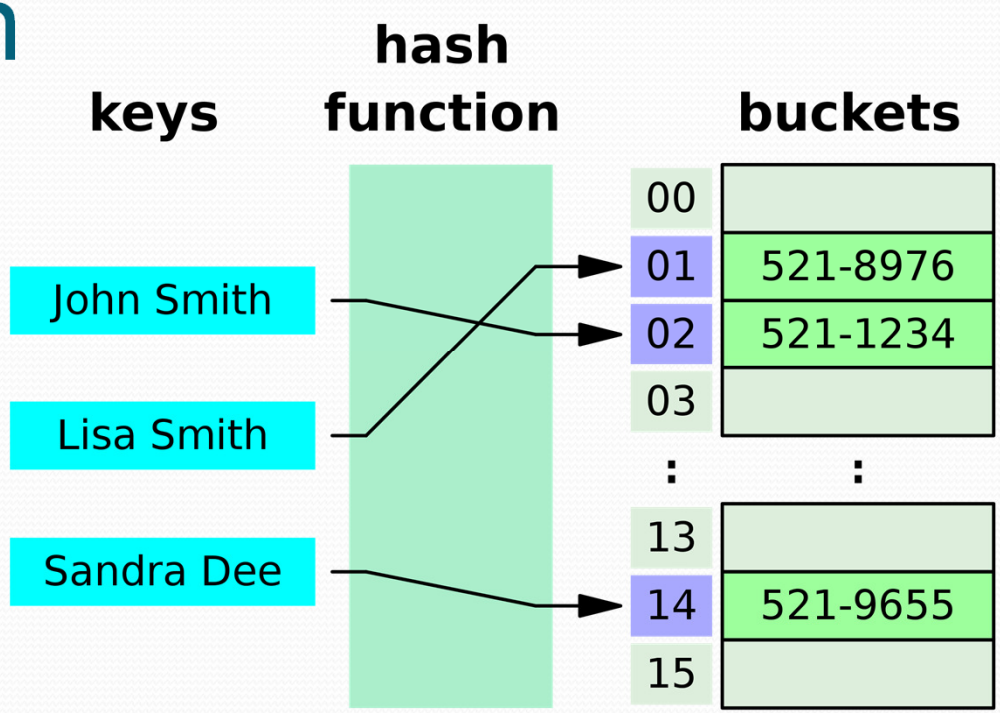
Interface Set

- En Java, los conjuntos están representados por la interface Set.
- Set es implementado por las clases HashSet, TreeSet y LinkedHashSet.
- HashSet:
  - Implementado mediante una tabla hash.
  - Muy rápido: O(1) para todas las operaciones.
  - Los elementos se almacenan en un orden impredecible.

Interface Set

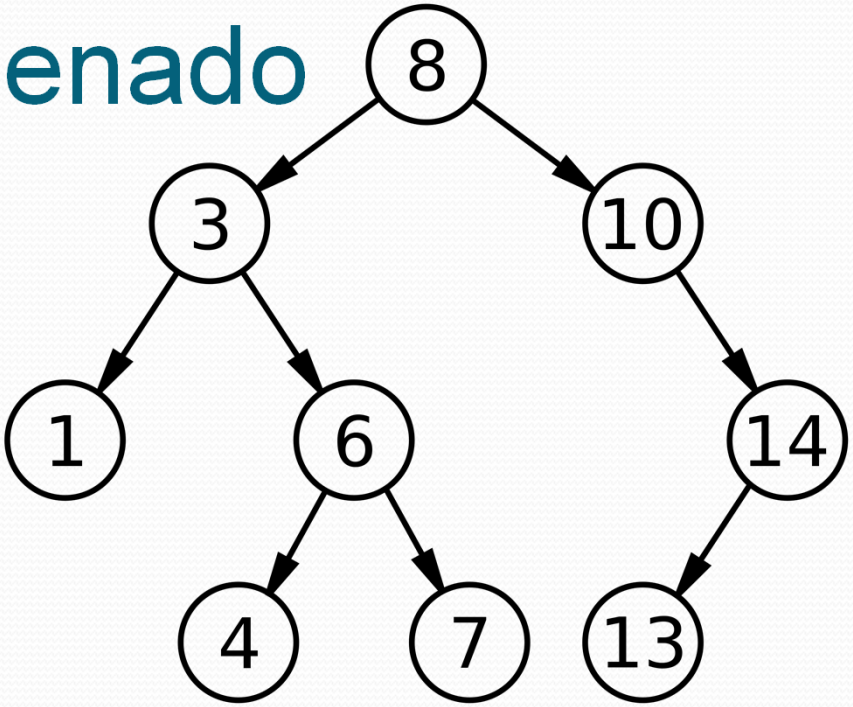
- TreeSet:
  - Implementado mediante un árbol de búsqueda binario.
  - Bastante rápido: O(log N) para todas las operaciones.
  - Los elementos se almacenan en orden.
- LinkedHashSet:
  - Implementado mediante una lista ligada y una tabla hash.
  - Muy rápido: O(1).
  - Los elementos se almacenan en el orden de inserción.

Tabla hash



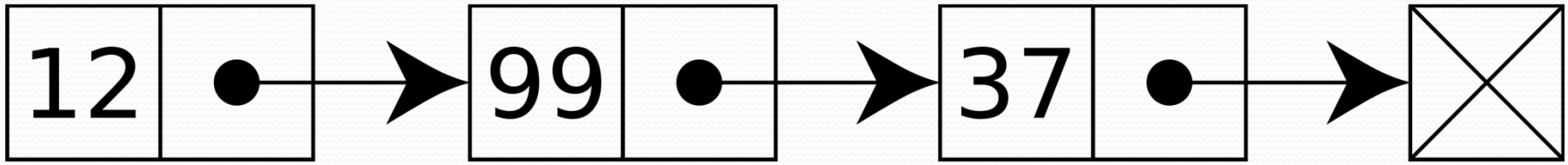
Fuente: By Jorge Stolfi - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6471238

Árbol binario ordenado



Fuente: By No machine-readable author provided. Dcoetzee assumed (based on copyright claims). - No machine-readable source provided. Own work assumed (based on copyright claims)., Public Domain, https://commons.wikimedia.org/w/index.php?curid=488330

Lista ligada



Fuente: By Vectorization: Lasindi - Own work based on: Singly linked list.png by Derrick Coetzee, Public Domain, https://commons.wikimedia.org/w/index.php?curid=2245162

Métodos de Set

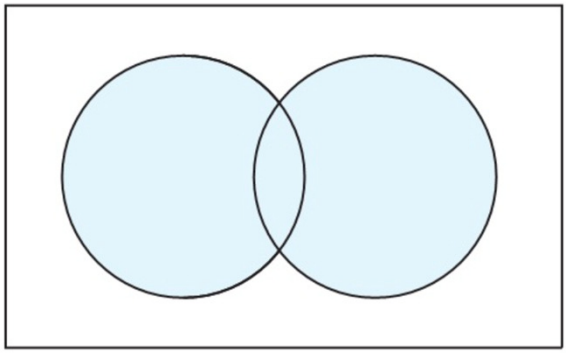
add(valor)	Agrega el valor al conjunto
contains(valor)	Devuelve true si el valor dado pertenece al conjunto
remove(valor)	Elimina el valor dado del conjunto
clear()	Elimina todos los elementos del conjunto
size()	Devuelve el número de elementos en el conjunto
isEmpty()	Devuelve true si el tamaño del conjunto es 0
toString()	Devuelve una representación del conjunto como string

Operaciones de Set

addAll(colección)	Agrega al conjunto todos los elementos de la colección
containsAll(colección)	Devuelve true si el conjunto tiene todos los elementos de la colección
equals(conjunto)	Devuelve true si este conjunto y el conjunto dado tienen los mismos elementos
iterator()	Devuelve un iterador para examinar el contenido del conjunto
removeAll(colección)	Borra del conjunto todos los elementos de la colección
retainAll(colección)	Borra del conjunto todos los elementos que *no* están en la colección
toArray()	Regresa un arreglo de los elementos de este conjunto

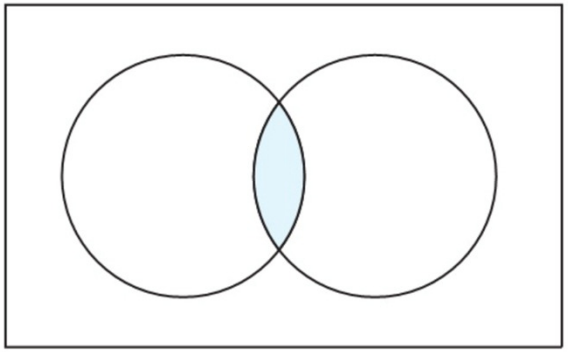
Operaciones de conjuntos

A ∪ B Union



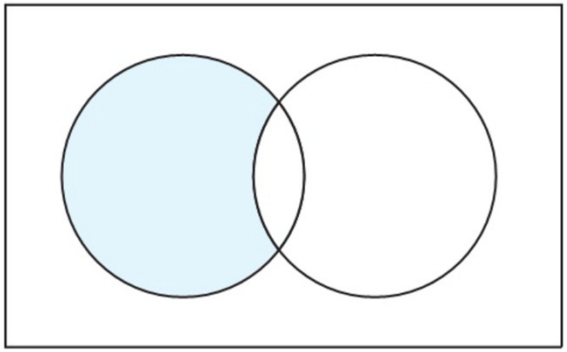
`addAll`

A ∩ B Intersection



`retainAll`

A - B Difference



`removeAll`

Ejemplos

```
List<String> lista = new ArrayList<>();
...
Set<Integer> set1 = new TreeSet<>();     // vacío
Set<String> set2 = new HashSet<>(lista);
```

- Se puede construir un conjunto vacío o uno basado en una colección ya existente.

Conjuntos y ordenamiento

- HashSet: los elementos se almacenan en orden impredecible.

```
Set<String> nombres_1 = new HashSet<>();
nombres_1.add("Juan");
nombres_1.add("Roberto");
nombres_1.add("Marissa");
nombres_1.add("Karina");
System.out.println(nombres_1);
// [Karina, Roberto, Juan, Marissa]
```

Conjuntos y ordenamiento

- TreeSet: los elementos se almacenan en su orden natural.

```
Set<String> nombres_2 = new TreeSet<>();
...
// [Juan, Karina, Marissa, Roberto]
```

- LinkedHashSet: los elementos se almacenan en orden de inserción.

```
Set<String> nombres_3 = new LinkedHashSet<>();
...
// [Juan, Roberto, Marissa, Karina]
```

Conjuntos de objetos

- Se pueden crear sin problemas conjuntos de objetos con HashSet y LinkedHashSet.
- Para utilizar TreeSet, la clase debe implementar la interface Comparable.
- Recordar que algunas clases, como String, ya implementan Comparable.

Recorrer un conjunto

- Se utiliza un foreach porque los conjuntos no tienen índices.

```
Set<Double> calificaciones = new HashSet<>();
...
for (double c : calificaciones) {
    System.out.println("Calificación: " + c);
}
```

Conjuntos con tipos enumerados

- **EnumSet**. Implementación especializada de la interface Set para usarse con tipos enumerados.
- EnumSet es una clase abstracta, no se puede instanciar.

Métodos para crear un EnumSet

Método	Descripción
allOf(Class)	Crea un conjunto con todos los elementos de la clase
complementOf(EnumSet)	Crea un conjunto con el complemento del conjunto dado
copyOf(Collection)	Crea un conjunto a partir de la colección dada
copyOf(EnumSet)	Crea un conjunto conteniendo los mismos elementos que el conjunto dado
noneOf(Class)	Crea un conjunto vacío
of(Element…)	Crea un conjunto conteniendo los elementos dados
range(From, To)	Crea un conjunto conteniendo los elementos especificados en el rango

Métodos de EnumSet

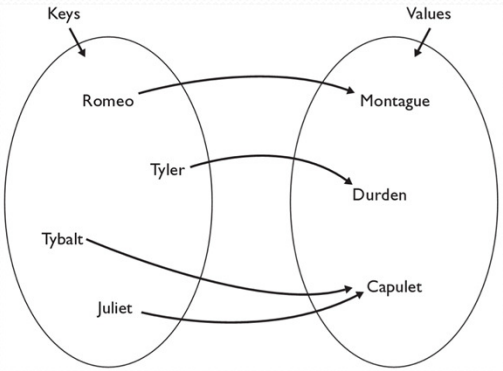
- La clase EnumSet hereda de Set los métodos para realizar operaciones con conjuntos: unión (addAll), intersección (retainAll) y resta (removeAll).

Ejemplo

```
public enum Months {JANUARY, FEBRUARY, MARCH, APRIL, MAY,
    JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER,
    DECEMBER}
EnumSet<Months> allMonths = EnumSet.allOf(Months.class);
EnumSet<Months> noMonths = EnumSet.noneOf(Months.class);
EnumSet<Months> springMonths = EnumSet.of(Months.MARCH,
Months.APRIL, Months.MAY, Months.JUNE);
EnumSet<Months> summerMonths = EnumSet.range(Months.JUNE,
Months.SEPTEMBER);
```

ADT Mapas

- **Mapa**. Contiene un conjunto de claves y una colección de valores.
- Cada clave está asociada con un valor.
- También conocido como “diccionario” o “arreglo asociativo”



ADT Mapas

- Operaciones básicas:
- put(clave, valor): agrega un mapeo de una clave a un valor.
- get(clave): recupera el valor asociado a la clave.
- remove(clave): elimina la clave y su valor asociado.

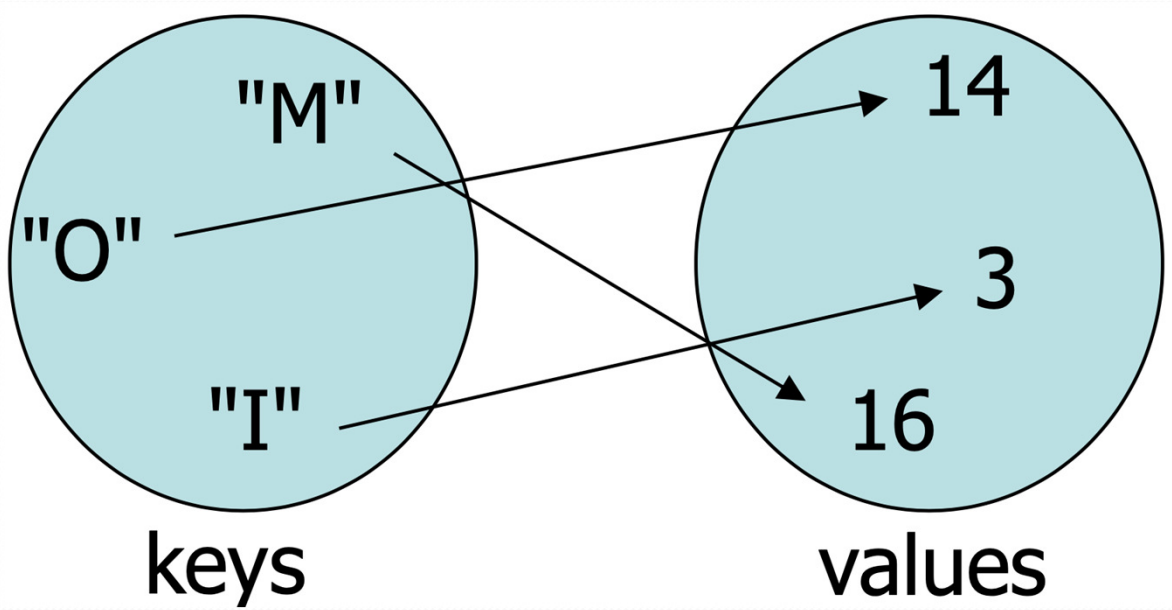
ADT Mapas

- Un mapa se puede utilizar como un arreglo de conteo.
- Ejemplo: hay tres candidatos: “I”, “M” y “O”.
- Votos: “MOOOOOOMMMMMOOOOOOMMMIMOMMIMOMMIM”.
- Conteo de votos:

Clave	“I”	“M”	“O”
Valor	3	16	14

ADT Mapas

- Mapa:



Implementación de mapas

- En Java, los mapas están representados por la interface Map.
- La interface Map es implementada por las clases HashMap y TreeMap.
- HashMap:
  - Implementado mediante una tabla hash.
  - Extremadamente rápido: O(1).
  - Las claves se almacenan en un orden impredecible.

Implementación de mapas

- TreeMap:
  - Implementado mediante un árbol binario.
  - Rápido: O(log N).
  - Las claves se almacenan en orden.

Declaración

- Un mapa requiere 2 parámetros de tipo: uno para las claves, otro para los valores.

```
// Mapa de claves String a valores Integer
Map<String, Integer> votos = new HashMap<>();
```

Métodos de mapas

put(clave, valor)	Agrega un mapeo de la clave dada al valor dado; si la clave ya existe, reemplaza su valor con el valor dado
get(clave)	Devuelve el valor asignado a la clave dada (null si no se encuentra)
containsKey(clave)	Devuelve true si el mapa contiene un mapeo para la clave dada
remove(clave)	Elimina cualquier mapeo existente para la clave dada
clear()	Elimina todos los pares clave/valor de este mapa
size()	Devuelve el número de pares clave/valor en este mapa
isEmpty()	Devuelve true si el tamaño de este mapa es 0
toString()	Devuelve una representación de este mapa como string
keySet()	Regresa un conjunto con todas las claves en este mapa
values()	Regresa una colección con todos los valores en este mapa

Métodos de mapas

putAll(mapa)	Agrega todos los pares clave/valor del mapa dado a este mapa
equals(mapa)	Devuelve true si el mapa dado tiene los mismos mapeos que este mapa

Keyset y values

- El método keySet devuelve un conjunto con todas las claves en el mapa.
- El conjunto se puede recorrer con un ciclo foreach.
- Se puede obtener el valor asociado de cada clave llamando a get en el mapa.

Keyset y values

- El método values devuelve una colección de todos los valores en el mapa.
- La colección se puede recorrer con un ciclo foreach.
- No hay una manera fácil de pasar de un valor a su(s) clave(s) asociada(s).

Ejemplo

```
Map<String, Integer> edades = new TreeMap<>();
edades.put("Juan", 19);
edades.put("Gaby", 2);
edades.put("Vicki", 57);
// edades.keySet() regresa Set<String>
for (String nombre : edades.keySet()) {                  // Gaby -> 2
    int edad = edades.get(nombre);                       // Juan -> 19
    System.out.println(nombre + " -> " + edad);          // Vicki -> 57
}
```

Interface Map.Entry

- Contiene una entrada (llave – valor) del mapa.
- Se puede usar el método entrySet para recorrer el mapa.
- Ejemplo:

```
Map<String, String> map = ...
for (Map.Entry<String, String> entry : map.entrySet()) {
    System.out.println(entry.getKey() + "/" + entry.getValue());
}
```

Iteradores

- No se pueden accesar los elementos de un Set o de un Map mediante un índice.
- Se debe usar un ciclo foreach.

```
Set<Integer> scores = new HashSet<>();
for (int score : scores) {
    System.out.println("El score es " + score);
}
```

- Problema: foreach es de solo lectura; no se puede modificar el conjunto durante el ciclo.

Iteradores

```
for (int score : scores) {
    if (score < 60) {
        // lanza un ConcurrentModificationException
        scores.remove(score);
    }
}
```

- La solución es usar un *iterador*.

Iterador

- **Iterador**. Un objeto que permite recorrer los elementos de una colección.
- Recuerda la posición y permite:
  - Obtener el elemento en esa posición.
  - Avanzar a la siguiente posición.
  - Remover el elemento en esa posición.

Métodos de Iterator

hasNext()	Devuelve true si hay más elementos para examinar
next()	Devuelve el siguiente elemento de la colección (lanza una NoSuchElementException si no queda ninguno para examinar)
remove()	Elimina el último valor devuelto por next() (lanza una IllegalStateException si aún no se ha llamado a next())

- La interface Iterator está en java.util.
- Cada colección tiene un método iterator() que devuelve un iterador sobre sus elementos

```
Set<String> set = new HashSet<>();
Iterator<String> iterator = set.iterator();
```

Uso de Iterator en un conjunto

```
Set<Integer> scores = new TreeSet<>();
scores.add(94);
scores.add(38);
scores.add(87);
scores.add(43);
scores.add(72);
...
```

Uso de Iterator en un conjunto

```
Iterator<Integer> iterator = scores.iterator();
while (iterator.hasNext()) {
    int score = iterator.next();
    System.out.println("El score es " + score);
    if (score < 60) {          // elimina los scores menores a 60
        iterator.remove();
    }
}
System.out.println(scores);       // [72, 87, 94]
```

Uso de Iterator en un mapa

```
Map<String, Integer> scores = new TreeMap<>();
scores.put(“Kim”, 38);
scores.put(“Lisa”, 94);
scores.put(“Roy”, 87);
scores.put(“Marty”, 43);
scores.put(“Marissa”, 72);
...
```

Uso de Iterator en un mapa

```
Iterator<String> iterator = scores.keySet().iterator();
while (iterator.hasNext()) {
    String name = iterator.next();
    int score = scores.get(name);
    System.out.println(name + " obtuvo " + score);
    if (score < 60) {            // elimina los scores menores a 60
        iterator.remove();       // elimina nombre y score
    }
}
System.out.println(scores);          // {Lisa=94, Marissa=72, Roy=87}
```

Conclusión

Main collection classes	D	O	S	TS
ArrayList	Yes	Yes	No	No
LinkedList	Yes	Yes	No	No
Vector	Yes	Yes	No	Yes
HashSet	No	No	No	No
LinkedHashSet	No	Yes	No	No
TreeSet	No	Yes	Yes	No
HashMap	No	No	No	No
LinkedHashMap	No	Yes	No	No
Hashtable	No	No	No	Yes
TreeMap	No	Yes	Yes	No

- **D:** Duplicate elements is allowed?
- **O:** Elements are ordered?
- **S:** Elements are sorted?
- **TS:** The collection is thread-safe?

Fuente: https://www.codejava.net/java-core/collections/java-collections-framework-summary-table

Explicación

- All lists allow duplicate elements which are ordered by index.
- All sets and maps do not allow duplicate elements.
- All list elements are not sorted.
- Generally, sets and maps do not sort its elements, except TreeSet and TreeMap – which sort elements by natural order or by a comparator.
- Generally, elements within sets and maps are not ordered, except for:
  - LinkedHashSet and LinkedHashMap have elements ordered by insertion order.
  - TreeSet and TreeMap have elements ordered by natural order or by a comparator.

Explicación

- There are only two collections are thread-safe: Vector and Hashtable. The rest is not thread-safe.