# Collections

# Introducción

Una *collection* es un objeto que agrupa múltiples elementos en una única unidad. Las Collections se utilizan para almacenar, recuperar, manipular y comunicar datos agregados.

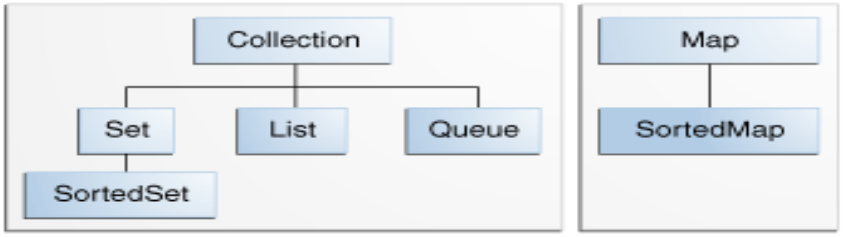
## Collections Framework

*Collections framework* es una arquitectura unificada para representar y manipular colecciones. Contiene:

- **Interfaces:** Tipos abstractos que representan colecciones. Permiten manipular las colecciones independientemente de los detalles de su representación. Forman una jerarquía.
- **Implementaciones:** Implementaciones concretas de los interfaces. En esencia, son estructuras de datos reusables.
- **Algoritmos:** Son métodos que realizan operaciones comunes, tal como ordenar o buscar, sobre objetos que implementan las interfaces colección. Estos algoritmos son polimórficos, esto es, el mismo método puede ser utilizado sobre diferentes implementaciones de las interfaces colección.

# Interfaces

Encapsulan diferentes tipos de colecciones. Permiten manipular las colecciones independientemente de los detalles de su representación. Tenemos la siguiente jerarquía de interfaces:




Un `Set` es una clase especial de `Collection`, un `SortedSet` es una clase especial de `Set`, etc. La jerarquía está compuesta por 2 árboles distintos — un `Map` no es una auténtica `Collection`.

Todas las interfaces son genéricas. Por ejemplo, la siguiente sería la declaración de la interface `Collection`.

```
public interface Collection<E>...
```

La sintaxis `<E>` indica que la interface es genérica. Al declarar una instancia de una `Collection` se puede *y debe* especificar el tipo de objeto contenido en la colección. El especificar el tipo permite al compilador verificar (en tiempo de compilación) que el objeto a introducir en la colección es correcto, reduciendo así los errores en tiempo de.

- `Collection` — raíz de la jerarquía. Una colección representa un grupo de objetos conocidos como sus *elementos*. El interface `Collection` es el mínimo común denominador que todas las colecciones implementan. Algunos tipos de colecciones permiten elementos duplicados, otros no. Algunos están ordenados, otros desordenados. La plataforma Java no provee ninguna implementación de esta interface, pero sí de subinterfaces más específicas como `Set` y `List`.

- `Set` — Una colección que no contiene elementos duplicados. Esta interface modela la abstracción matemática de conjunto.

- `List` — Una colección ordenada (llamada a veces *sequence*). `List` puede contener elementos duplicados. El usuario de una `List` generalmente necesita control preciso sobre donde se inserta cada elemento en la lista y accede a los elementos por su índice (posición numérica)

- `Queue` — una colección que mantiene múltiples elementos previamente a su proceso. A mayores de las operaciones básicas de una `Collection`, una `Queue` proporciona operaciones adicionales de inserción, extracción e inspección.

  Las colas normalmente, aunque no necesariamente, ordenan sus elementos en modo FIFO (First-In, First-Out)

- `Map` — un objeto que mapea clave a valores. Un `Map` no puede tener claves duplicadas; cada clave debe referenciar a al menos un valor.

Las últimas dos interfaces son versions ordenadas de `Set` y `Map`:

- `SortedSet` — Un `Set` que mantiene a sus elementos en orden ascendente.
- `SortedMap` — Un `Map` que mantiene sus mapeos en orden ascendente de clave.

Interface `Collection`:

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

# Recorrido de Collections

Hay dos maneras principales de recorrer colecciones: (1) con `for-each` (2) usando `Iterator`s.

### for-each

La sentencia `for-each` permite recorrer colecciones o arrays usando un bucle `for`. Ejemplo de recorrido de una colección imprimiendo cada elemento en una línea.

```
for (Object o : collection)
    System.out.println(o);
```

### Iterators

Un `Iterator` es un objeto que permite recorrer una colección y eliminar elementos de la colección selectivamente. Se obtiene un `Iterator` llamando al método `iterator`.

Interface `Iterator`:

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
```

```
}
```

El método `hasNext` devuelve `true` si la iteración tiene más elementos, y el método `next` devuelve el siguiente elemento de la iteración. El método `remove` elimina el último elemento devuelto por `next`. El método `remove` solo puede ser llamado una vez por cada llamada a `next` y lanza una excepción si se viola esta regla.

`Iterator.remove` es la *única* forma segura de modificar una colección durante la iteración; el comportamiento es impredecible si la colección se modifica de cualquier otra manera mientras la iteración está en progreso.

Usar `Iterator` en lugar de `for-each` cuando se necesite:

- Eliminar el element actual. El `for-each` oculta el iterador, con lo cual no se puede llamar a `remove`.
- Iterar sobre multiples colecciones en paralelo.

El siguiente ejemplo muestra cómo usar un `Iterator` para filtrar una `Collection` — esto es, recorrer la colección eliminando elementos específicos.

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

Este código es polimórfico, funciona para cualquier `Collection` independientemente de su implementación.

# Collection Interface Bulk Operations

*Bulk operations* realizan una operación sobre una `Collection` completa.

- `containsAll` — devuelve `true` si la `Collection` destino contiene todos los elementos de la `Collection` especificada.
- `addAll` — añade todos los elementos de la `Collection` especificada a la `Collection` destino.
- `removeAll` — elimina de la `Collection` destino todos los elementos que están también contenidos en la `Collection` especificada.
- `retainAll` —elimina de la `Collection` destino los elementos que no están contendidos en la `Collection` especificada.
- `clear` — elimina todos los elementos de la `Collection`.

Los métodos `addAll`, `removeAll`, y `retainAll` devuelven `true` si la `Collection` destino fue modificada al ejecutar la operación.

Ejemplo. Eliminar todos los elementos `null` de una `Collection`.

```
c.removeAll(Collections.singleton(null));
```

El ejemplo utiliza `Collections.singleton`, que devuelve un `Set` inmutable conteniendo solo el element específicado.

# Collection Interface Array Operations

El método `toArray` se proporciona como un Puente entre las colecciones y las APIs antiguas que esperan arrays como entrada. Las operaciones array permiten trasladar el contenido de una `Collection` en un array. La forma más simple, sin argumentos, crea un Nuevo array de `Object`. La forma compleja permite especificar un array destino o escoger el tipo en tiempo de ejecución del array de salida.

Ejemplo, suponiendo que `c` es una `Collection`. El siguiente código vuelca el contenido de `c` en un Nuevo array de `Object` cuya longitud es idéntica al número de elemento tos en `c`.

```
Object[] a = c.toArray();
```

Suponiendo que sabemos que `c` solo contiene strings (quizá porque `c` es de tipo `Collection<String>`), el siguiente código volcaría el contenido de `c` en un Nuevo array de `String` cuya longitud es idéntica al número de elementos en `c`.

```
String[] a = c.toArray(new String[0]);
```