Colecciones en JAVA

Colecciones

Una **colección o contenedor** es un objeto que agrupa múltiples elementos ú objetos.

Las colecciones se usan para almacenar, recuperar, manipular y comunicar datos agregados.

Las colecciones representan elementos agrupados naturalmente, por ej. una carpeta de emails, un catálogo, el conjunto de registros retornados al ejecutar una consulta a una BD, un diccionario, etc.

Un **framework de colecciones** es una arquitectura que permite representar y manipular colecciones de datos de manera estándar. Todos los *frameworks de colecciones* están compuestos por:

Interfaces: son **tipos de datos abstractos** que representan colecciones. Las interfaces permiten que las colecciones sean manipuladas independientemente de los detalles de implementación. Forman una jerarquía.

Implementaciones: son las implementaciones concretas de las interfaces. Son **estructuras de datos** reusables.

Algoritmos: son **métodos** que realizan operaciones útiles (búsquedas y ordenamientos) sobre objetos que implementan alguna de las interfaces de colecciones. Son métodos **polimórficos** es decir el mismo método se usa sobre diferentes implementaciones de las interfaces de colecciones. Son unidades funcionales reusables.

El **framework de colecciones** forma parte de JAVA a partir de la versión 1.2

Colecciones en JAVA

Reduce la programación: provee estructuras de datos y algoritmos útiles. Facilita la interoperabilidad entre APIs no relacionadas evitando escribir adaptadores o código de conversión para conectar APIs.

Provee estructuras de datos de tamaño no-limitado: es posible almacenar la cantidad de objetos que se desee.

Aumenta la velocidad y calidad de los programas: provee implementaciones de estructuras de datos y algoritmos de alta *performance y calidad*. Las diferentes implementaciones de las interfaces son intercambiables pudiendo los programas adaptarse a diferentes implementaciones.

Permite interoperabilidad entre APIs no relacionadas: establece un lenguaje común para pasar colecciones de elementos.

Promueve la reusabilidad de software: las interfaces del framework de colecciones y los algoritmos que manipulan las implementaciones de las interfaces son reusables.

El *framework* de colecciones de JAVA está formado por un conjunto de clases e interfaces ubicadas mayoritariamente en el paquete **java.util.**

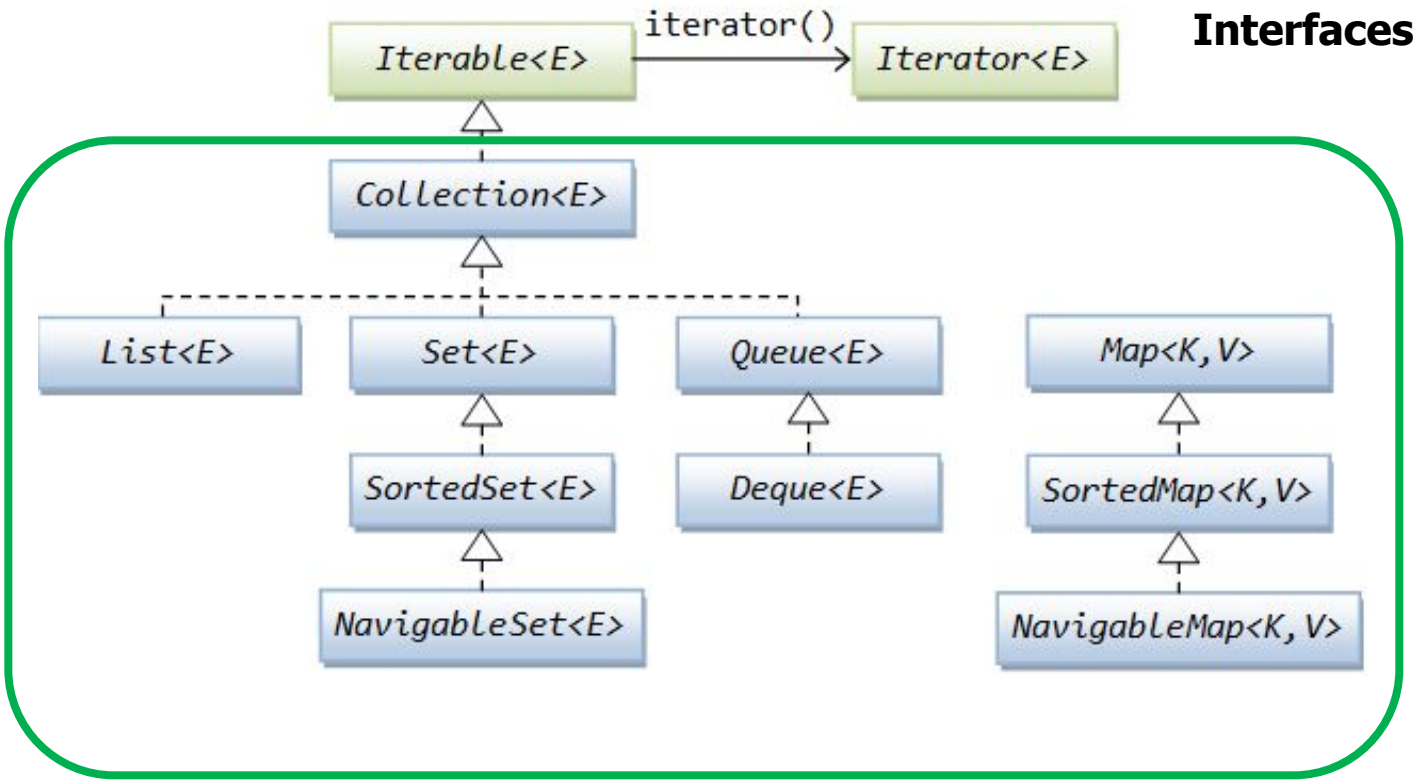
A partir de **JAVA 5** soporta **tipos genéricos.** El **compilador inserta castings** automáticamente y realiza comprobación de tipos cuando se agregan elementos, evitando errores que sólo podían detectarse en ejecución. Como resultado los **programas son más seguros y claros**.

Colecciones y Genéricos en JAVA

Los **tipos genéricos en JAVA** fueron **incorporados** fundamentalmente para **implementar colecciones genéricas.**

- El **framework de colecciones no-genérico en JAVA** no ofrecía ninguna forma de colecciones homogéneas. Todas las colecciones contenían elementos de tipo **Object** y por esa razón eran de **naturaleza heterogénea**, mezcla de objetos de diferente tipo. Esto se puede observar desde la API de colecciones (jse 4): las colecciones no-genéricas aceptan objetos de cualquier tipo para insertar en la colección y retornan una referencia a un **Object** cuando un elemento es recuperado de una colección.
- El **framework de colecciones genérico de JAVA** permite implementar **colecciones homogéneas**. Este framework se define a través de **interfaces genéricas y clases genéricas** que pueden ser instanciadas por una gran variedad de tipos. Por ejemplo, la interface genérica **List<E>** puede ser parametrizada como una **List<String>**, **List<Integer>**, cada una de ellas es una lista homogénea de valores strings, enteros, etc. A partir de la clase genérica **ArrayList<E>** puede obtenerse la clase parametrizada **ArrayList<String>, ArrayList<Double>,** etc.

Interfaces Centrales y Secundarias



Interfaces secundarias

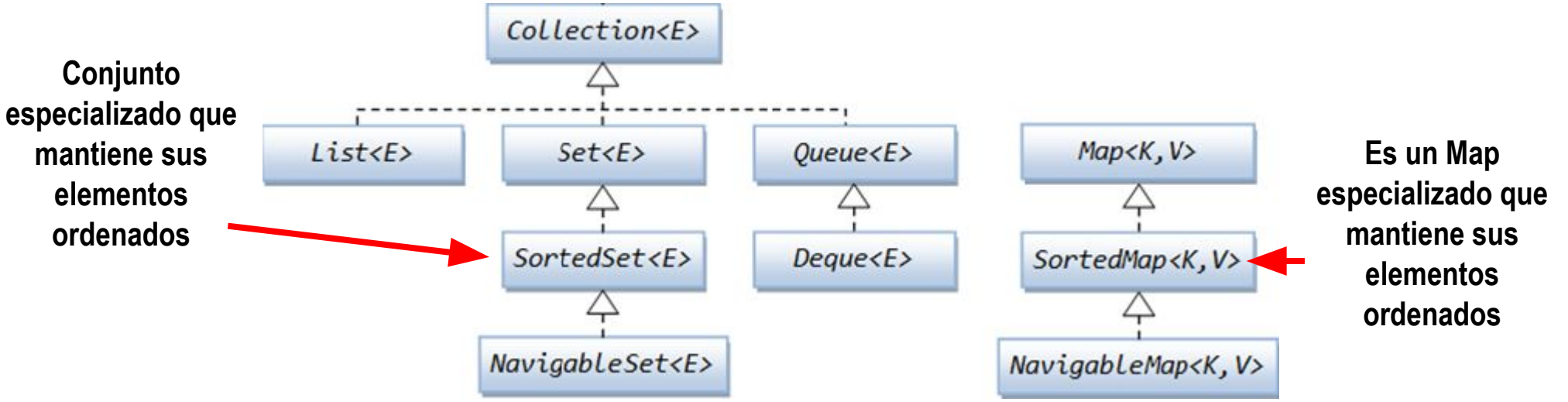
Interfaces centrales

Las **interfaces centrales especifican los múltiples contenedores** de elementos y las **interfaces secundarias especifican las formas de recorrido** de las colecciones.

Las **interfaces centrales** permiten a las colecciones ser manipuladas independientemente de los detalles de implementación.

A partir de Collection y Map se definen dos jerarquías de interfaces que constituyen el fundamento del framework de colecciones de JAVA

Interfaces Centrales



Todas **las interfaces de colecciones son genéricas**. Encapsulan distintos tipos de colecciones de objetos y son el **fundamento del framework de colecciones de Java**; pertenecen al paquete **java.util**. Las **interfaces centrales** son:

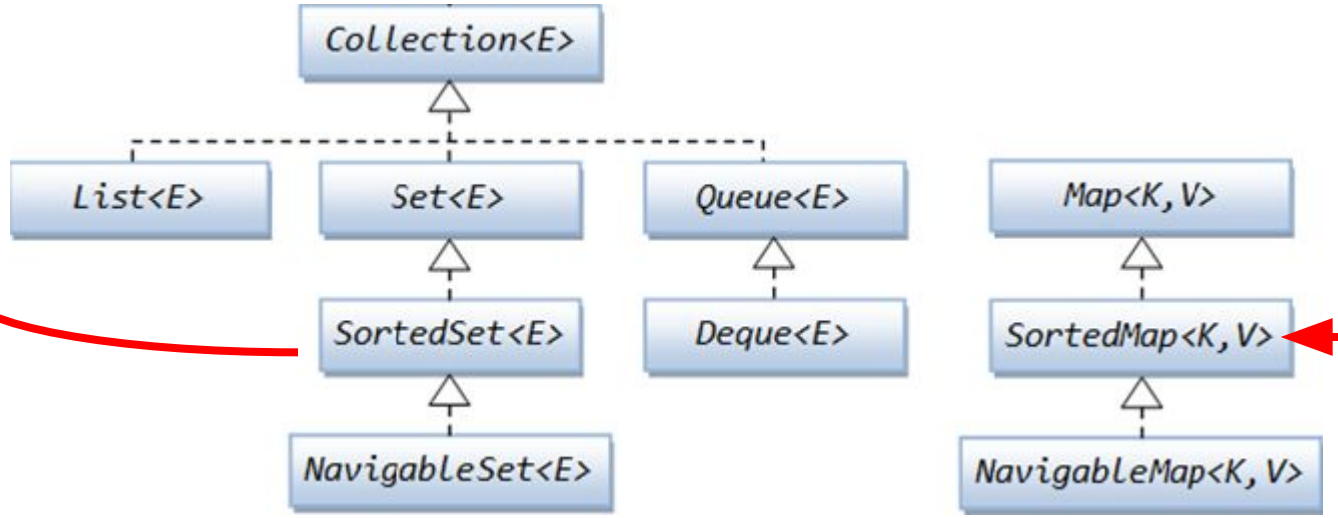
Collection: generaliza el concepto de **grupos de objetos** llamados elementos. Es la raíz de la jerarquía de colecciones. La plataforma Java no provee una implementación directa para la interface Collection pero sí para sus subinterfaces **Set**, **List** y **Queue**.

Set: es una colección que **no contiene duplicados**. Modela el concepto de conjunto matemático y es usado para representar conjuntos por ej. los materias que cursa un estudiante, los procesos en una computadora.

List: guarda los elementos en el mismo orden en que fueron insertados, permite elementos duplicados. También llamada **secuencia**. Provee acceso indexado.

Interfaces Centrales

Conjunto especializado que mantiene sus elementos ordenados



Es un Map especializado que mantiene sus elementos ordenados

Queue: mantiene los elementos previo a ser procesados. Agrega **operaciones adicionales para inserción, extracción e inspección** de elementos. Típicamente los elementos de una Queue están **ordenados** mediante una **estrategia FIFO** (First In First Out). Existen implementaciones de colas de prioridades.

Deque: puede ser usada con estrategia FIFO (First In First Out) y como una LIFO (Last In First Out). Todos los elementos pueden ser insertados, recuperados y removidos de ambos extremos.

Map: representa asociaciones entre objetos de la forma clave-valor. **No permite claves duplicadas**. Cada clave está asociada a lo sumo con un valor. También llamada **diccionario**. Tiene similitudes con **Hashtable**.

Versiones ordenadas de Set y Map:

SortedSet: es un Set que mantiene todos los elementos ordenados en orden ascendente. Se agregan métodos adicionales para sacar ventaja del orden. Se usan para conjuntos ordenados naturalmente.

SortedMap: es un Map que mantiene sus asociaciones ordenadas ascendentemente por clave. Son usados para colecciones de pares clave-valor naturalmente ordenados (diccionarios, directorios telefónicos).

Implementaciones de propósito general

Implementación de SortedSet

Interfaces	Implementaciones				
	Tabla de Hashing	Arreglo de tamaño variable	Árbol	Lista Encadenada	Tabla de Hashing + Lista Encadenada
Set	HashSet	EnumSet	TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue			PriorityQueue		
Map	HashMap	EnumMap	TreeMap		LinkedHashMap

Implementación de SortedMap

Las implementaciones de propósito general definen todos los métodos opcionales. Permiten elementos, claves y valores **null**. No son *thread-safe*. Todas son serializables. La regla es pensar en términos de interfaces y no de implementaciones.

HashMap: es una implementación de un Map basada en una tabla de hashing. Las operaciones básicas (get() y put()) tienen tiempo de ejecución constante O(1).

HashSet: es una implementación de Set basada en una tabla de hashing. No hay garantías acerca del orden de ejecución en las distintas iteraciones sobre el conjunto. Las operaciones básicas (add(), remove(), size(), contains()) tienen tiempo de ejecución constante O(1).

ArrayList: es una implementación de List basada en arreglos de longitud variable, es similar a Vector. Las operaciones size(), isEmpty(), get(), set(), iterator(), y listIterator() tienen tiempo de ejecución constante O(1). La operación add() tiene tiempo de ejecución lineal O(n).

Hay dos implementaciones de **Queue**: **PriorityQueue** y **LinkedList** (también implementa List)

Interface Colletion

```
public interface Collection<E> extends Iterable<E> {
    //Operaciones para Agregar Elementos
    boolean add(E element);
    boolean addAll(Collection<? extends E> c);
    //Operaciones para Eliminar Elementos
    boolean remove(Object element);
    void clear();
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    //Operaciones de Consultas
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean containsAll(Collection<?> c);
    //Operaciones que facilitan el Procesamiento
    Iterator iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    //Operaciones para obtener Stream
    default   Stream<E> parallelStream();
    default   Stream<E> stream();
}
```

Es una interface genérica que representa un agrupamiento de objetos de tipo E

Cuando se declara un objeto Collection es posible establecer el tipo de dato que se almacenará en la colección y de esta manera evitar *castear* cuando se leen los elementos. Se evitan errores de *casting* en ejecución y se le da al compilador información sobre el tipo usado para hacer un chequeo fuerte de tipos.

```
import java.util.*;
public class ColeccionSimple {
  public static void main(String [] args){
    Collection<Integer> c=new ArrayList<>();
     for (int i=0; i < 10; i++)
         c.add(i);
      for(int i: c)
         System.out.println(i);
   }
}
```

Auto-Boxing

El framework de colecciones de JAVA NO provee ninguna implementación de Collection, sin embargo es una interface muy importante pues establece un comportamiento común para todas las subinterfaces. Permite convertir el tipo de las colecciones.

Recorrer una Colección

1) Usando el constructor *for-each.*

El constructor **for-each** provee una manera concisa de recorrer colecciones y arreglos. Se incorporó en la versión JSE 5. Es la manera preferida

Autoboxing/Unboxing

Convierte automáticamente datos de tipos primitivos (como int) a objetos de clases *wrappers* (como Integer) cuando se inserta en la colección y, convierte instancias de clases *wrappers* en primitivos cuando se leen elementos de la colección.

Es una característica soportada a partir de **JSE 5**

El for-each funciona bien con cualquier cosa que produzca un iterador. En Java 5 la interface Collection extiende Iterable, entonces con cualquier implementación de Set, List y Queue se puede usar el for-each

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

```
public class DemoColeccion {
  public static void main(String[] args) {
     List<Integer> lista= new ArrayList<>();
     lista.add(1);
     lista.add(2);
     lista.add(190);
     lista.add(90);
     lista.add(7);
     for (int i: lista)
     System.out.println(i);
  }
}
```

Recorrer una Colección

2) Usando la interface Iterator

Un objeto **Iterator** permite recorrer secuencialmente una colección y remover elementos selectivamente si se desea. Es posible obtener un iterador para una colección invocando al método **iterator()**. Una colección es un objeto **Iterable.**

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

Interfaces secundarias del *framework* de colecciones de java

```
public interface ListIterator<E> extends Iterator<E>
{
    boolean hasPrevious();
    int nextIndex();
    E previous();
    int previousIndex();
    void add(E elemento); //opcional
    void set(E elemento);  //opcional
    void remove();          //opcional
}
```

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

```
public class DemoIterador{
 public static void main(String[] args) {
    List<Integer> lista= new ArrayList<>();
    lista.add(1);
    lista.add(2);
    lista.add(190);
    lista.add(90);
    lista.add(7);
    Iterator it<Integer>= lista.iterator();
    while (it.hasNext())
        System.out.println(it.next());
 }
}
```

NO es la manera preferida de escribir código para recorrer secuencias en JAVA 5

Interface List

Un objeto **List** es una **secuencia de elementos**, cada uno tiene una posición, permite duplicados y los elementos se almacenan en el mismo orden en que son insertados. La interface **List** define métodos para recuperar y agregar elementos en una posición determinada o índice.

```
public interface List<E> extends Collection<E> {
// Métodos de Accesso Posicional
   void add(int index, E element);
   boolean addAll(int index, Collection<? extends E> c);
   E get(int index);
   E remove(int index);
   E set(int index, E element);
// Métodos de Búsqueda
    int indexOf(Object o);
    int lastIndexOf(Object o);
//Métodos de Iteración
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);
//Método de Rangos de Vistas
    List<E> subList(int from, int to);
}
```

ListIterator es un iterador que aprovecha las ventajas de la naturaleza secuencial de la lista: permite recorrer la lista en cualquier dirección (hacia adelante y hacia atrás), modificarla durante el recorrido y obtener la posición del elemento actual

```
public class DemoIterador{
  public static void main(String[] args) {
   List <Number> lista=new ArrayList<>();
   lista.add(10);
   lista.add(200.0f);
   lista.add(100.0);
   Set<Float> setDecimales= new HashSet<>();
   setDecimales.add(100.0F);
   setDecimales.add(200.0F);
   setDecimales.add(300.0F);
   lista.addAll(setDecimales);
   ListIterator it<Number>= lista.listIterator(lista.size());
     while (it.hasPrevious())
          System.out.println(it.previous());
  }
}
```

Interface Set

Un objeto **Set** es una colección de objetos **sin duplicados**: no contiene 2 referencias al mismo objeto, 2 referencias a **null** o referencias a 2 objetos a y b tal que a.equals(b). El propósito general de las implementaciones de **Set** son colecciones de objetos sin orden. Sin embargo existen conjuntos con orden, SortedSet.

```
public interface Set<E> extends Collection<E> {
    boolean add(E o);
    boolean addAll(Collection<? extends E> c);
    void clear( );
    boolean contains(Object o);
    boolean containsAll(Collection<?> c);
    boolean equals(Object o);
    int hashCode( );
    boolean isEmpty( );
    Iterator<E> iterator( );
    boolean remove(Object o);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    int size( );
    Object[ ] toArray( );
    <T> T[ ] toArray(T[ ] a);
}
```

```
public class TestSortedSet {
   public static void main (String[] args) {
     SortedSet<String> s = new TreeSet<>(Arrays.asList(args));
     // Itera: los elementos son ordenados automáticamente
     for(String palabra : s)
         System.out.println(palabra);
     // Recuperate elementos especiales
     String primero = s.first(); // Primer Elemento
     String ultimo = s.last(); // Último Elemento
   }
}
```

java TestSortedSet hola chau adios quetal elena hola chau elena luis

¿Cuál es la salida?

Interface Map

Es la única interface que no hereda de Collection. Un objeto **Map** es un conjunto de asociaciones clave-valor. Las claves son únicas y cada clave está asociada con a lo sumo un valor. Es un tipo genérico con 2 parámetros que representan tipos de datos: **K** es el tipo de las claves y **V** el tipo de los valores.

```
public interface Map <K,V> {
// Agregar Asociaciones
   V put(K clave, V valor);
   void putAll(Map<? extends K, ? extends V> t);
// Eliminar Asociaciones
    void clear();
    V remove(K clave);
// Consultar el Contenido
   V get(K clave);
   boolean containsKey(K clave);
   boolean containsValue(V valor);
   int size();
   boolean isEmpty();
//Proveer Vistas de Claves, Valores o Asociaciones
    Set<Map.Entry<K, V>> entrySet();
    Set<K> keySet();
    Collection<V> values();
}
```

Implementación de la interface Map

```
public class DemoMap {
public static void main(String[] args) {
   Map<String, Persona> tablaPersona = new HashMap<>();
   Persona[] arregloPer = {
         new Persona("Luis", 27), new Persona("Elena", 20),
         new Persona("Claudia", 30), new Persona("Claudia", 40),
         new Persona("Elena", 22), new Persona("Manuel", 28),
         new Persona("Luis", 44) };
    for (Persona unaPer : arregloPer)
         tablaPersona.put(unaPer.getNombre(), unaPer);
    Collection<Persona> listPer = tablaPersona.values();
     for (Persona unaPer : listPer)
          System.out.println(unaPer);
 }
}
```

Cambiando la instanciación por un objeto **TreeMap()** obtenemos una colección ordenada:

```
Map<String, Persona> numeros=new TreeMap<>();
```

Aunque no es una Collection sus claves pueden ser recuperadas como un Set, sus valores como Collection y sus asociaciones como un Set de Map.Entry(es una interface anidada en Map que representa pares clave-valor)

Interface Queue

Un objeto **Queue** es una colección diseñada para mantener elementos que esperan por procesamiento. Además de las operaciones de **Collection** provee operaciones para insertar, eliminar e inspeccionar elementos. No permite elementos nulos. La plataforma java provee una implementación de **Queue** en el paquete **java.util**, **PriorityQueue**, que es una cola con prioridades y varias en el paquete **java.util.concurrent** como **DelayQueue** y **BlockingQueue** que implementan diferentes tipos de colas, ordenadas o no, de tamaño limitado o ilimitado, etc.

```
public interface Queue<E> extends Collection<E>{
 boolean add(E e);
 E element();
 E remove();
 E peek();
 boolean offer(E e);
 E poll();
}
```

Recupera, pero no elimina la cabeza de la cola.

Inserta el elemento en la cola si es posible

Recupera y elimina la cabeza de la cola.

PriorityQueue chequea que los objetos que se insertan sean **Comparables!!**

```
public class DemoQueue{
 public static void main(String[] args) {
    Queue<String> pQueue = new PriorityQueue<>();
    pQueue.offer("Buenos Aires");
    pQueue.offer("Montevideo");
    pQueue.offer("La Paz");
    pQueue.offer("Santiago");
    System.out.println(pQueue.peek());
    System.out.println(pQueue.poll());
    System.out.println(pQueue.peek());
 }
}
```

¿Cuál es la salida?

¿Cómo las interfaces SortedSet y SortedMap mantienen el orden de sus elementos?

De acuerdo al ordenamiento natural de sus elementos (en el caso de SortedMap de sus claves) o a un comparador de ordenación provisto en el momento de la creación (**Comparator**).

Las interfaces **Comparable** y **Comparator** permiten comparar objetos y de esta manera es posible ordenarlos.

Múltiples clases de la plataforma Java implementan la interface **Comparable**, entre ellas: String, Integer, Double, Float, Boolean, Long, Byte, Character, Short, Date, etc.

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

Interfaces secundarias del framework de colecciones de java

Retorna: cero (0), un valor negativo o uno positivo

```
public interface Comparator<T> {
    int compare(T o1, T o2);
    boolean equals(Object obj)
}
```

```
public class Alumno implements Comparable<Alumno>  {
  private int legajo;

   public int compareTo(Alumno otro) {
        int resultado=0;
        if (this.getLegajo() < otro.getLegajo())
                resultado = -1;
        if (this.getLegajo() > otro.getLegajo())
                resultado = 1;
        return resultado;
    }

}
```

Ejemplos de Set con Orden y sin Orden

```
public class DemoOrdenado{
    public static void main(String[] args) {
    Set<String> instrumentos= new HashSet<>();
    instrumentos.add("Piano");
    instrumentos.add("Saxo");
    instrumentos.add("Violin");
    instrumentos.add("Flauta");
    instrumentos.add("Flauta");
    System.out.println(instrumentos.toString());
}
```

La interface Set es útil para crear colecciones sin duplicados desde una colección **c** con duplicados:

```
Set<String> sinDup=new TreeSet<>(c);
```

salida **[Violin, Piano, Saxo, Flauta]**

Implementa **SortedSet**

Cambiando únicamente la instanciación por un objeto **TreeSet()** obtenemos una colección ordenada:

```
Set<String> instrumentos= new TreeSet<String>();
```

En este caso el compilador chequea que los objetos que se insertan con el método add() sean **Comparables!!**

salida **[Flauta, Piano, Saxo, Violin]**

API de Stream

La **API de Streams** se introduce en Java 8 y su propósito es manipular **colecciones de datos** de forma **declarativa** y **funcional**, es decir permite expresar qué se quiere hacer en lugar de codificar una implementación. Es un conjunto de clases e interfaces del paquete java.util.stream

```
List<String> instrumentos = List.of("Piano", "Saxo", "Violin", "Flauta", "Guitarra", "Saxo");
// Mostrar los instrumentos ordenados
instrumentos.stream().sorted()
                     .forEach(System.out::println);
```

Ordena los instrumentos alfabéticamente.
Recorre el stream secuencialmente.

```
int cant_instrumentos_con_a = (int) instrumentos.stream()
                                .filter(instr -> instr.contains("a"))
                                .count();
```

Filtra: selecciona los instrumentos que contienen una letra a.
Cuenta los elementos del stream filtrado.

Un **stream** es una **secuencia** de elementos **creado** a partir de una **fuente** (colección) que soporta operaciones de procesamiento de datos.

Favorece un estilo de código declarativo: se especifica qué se desea hacer, por ej. filtrar, ordenar, contar, recorrer, sin escribir el código de cómo se hace.

API de Stream

```
List<Dish> menu = List.of(
        new Dish(Dish.Type.PASTA, "pasta", true, 350),.......);
```

```
List<Dish> vegetarianDishes = menu.stream()
                .filter(Dish::isVegetarian)
                .collect(toList());
```

Filtra: selecciona las comidas vegetarianas.
Guarda en una lista las comidas filtradas.

```
List<String> lowCaloricDishesName = menu. stream()
                .filter(d -> d.getCalories() < 400)
                .sorted(comparing(Dish::getCalories))
                .map(Dish::getName)
                .collect(toList());
```

1. **Filtra**: selecciona las comidas que contienen menos de 400 calorías.
2. **Ordena** las comidas seleccionadas de acuerdo a las calorías.
3. **Extrae** los nombres de dichas comidas.
4. **Guarda** en una lista nombres de las comidas.

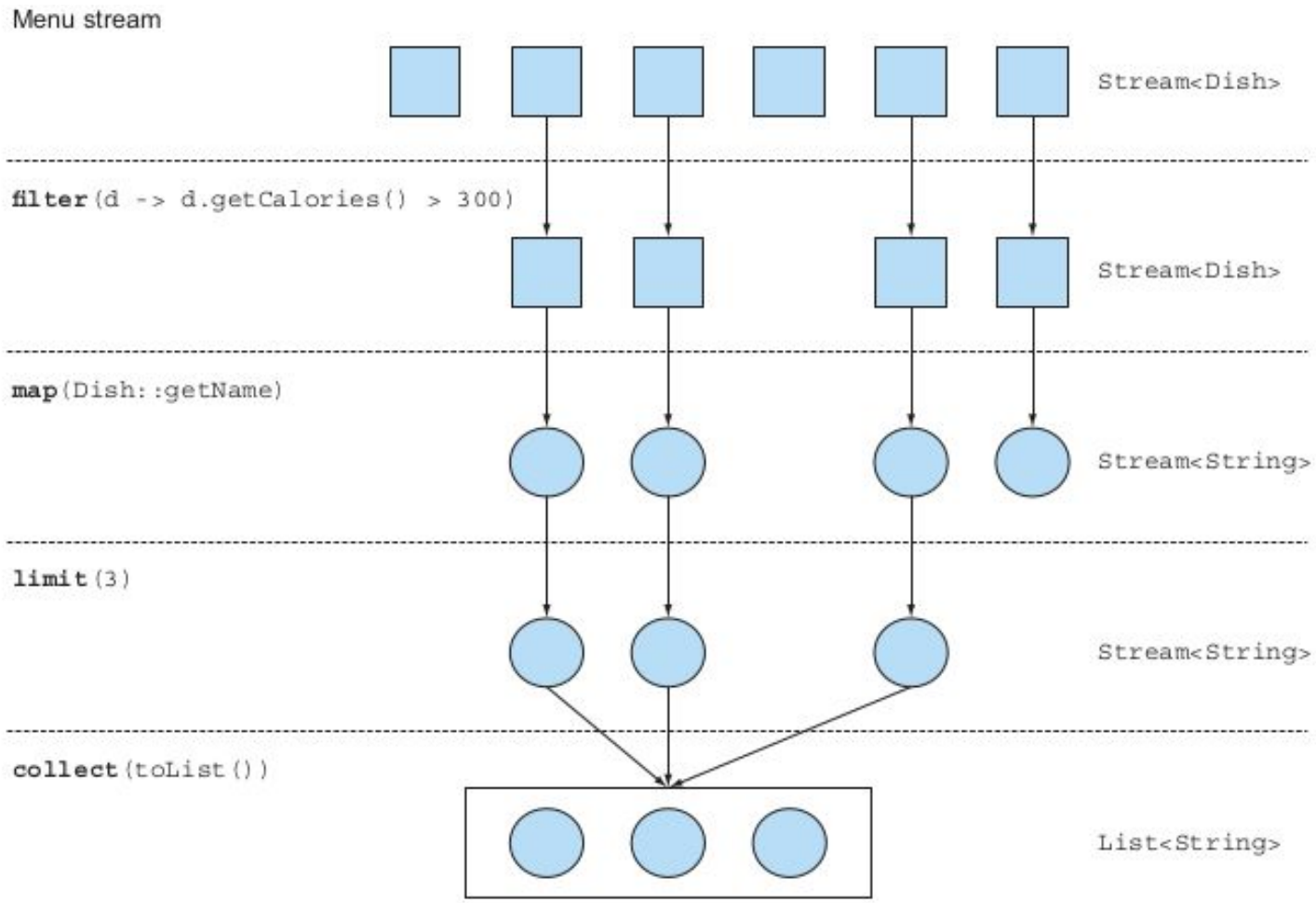
Operaciones intermedias: devuelven otro stream (ej: filter, map, sorted).
Operaciones terminales: consumen el stream y producen un resultado concreto (ej: collect, forEach, reduce). El stream queda cerrado y no se puede reutilizar.
Los **streams** son **inmutables**: no modifican la colección original sobre el que se crea y cada operación intermedia devuelve un nuevo stream.

API de Stream - *Pipelining*

Operaciones de procesamiento de datos

Operaciones intermedias

Operación terminal



Pipelining: las operaciones sobre un stream se encadenan en un pipeline o tubería, no se ejecutan de inmediato..
Evaluación perezosa (Lazy): cuando se invoca una operación terminal se recorre la secuencia y ejecutan todas las operaciones intermedias.