IPOO

- Docentes
  - Profesores Adjuntos:
    - Carlos Di Cicco.
  - JTPs:
    - Federico Naso (Junín) .
      Nelson Di Grazia (Pergamino) .

- **JAVA COLLECTIONS FRAMEWORK**

Collection

Una collección (Collection) es un objeto que agrupa múltiples elementos dentro de una unidad.

Las colecciones son usadas para guardar, recuperar y manipular datos agregados.

Normalmente, representan los elementos de un grupo natural, tales como una mano de poker (una colección de cartas), un buzón de correo (una colección de mails), o una guía telefónica (un mapeo de nombres a los números de teléfono).

Collection. ¿Qué es un framework de colecciones?

Un framework de colecciones o estructura de colecciones es una arquitectura unificada para representar y manipular colecciones. Todos los framework de colecciones contienen:

- Interfaces: Son los tipos de datos abstractos que representan las colecciones. Las interfaces permiten a las colecciones ser manipuladas de manera independiente a los detalles de su representación. Generalmente las interfaces forman una jerarquía.

- Implementaciones: Son las implementaciones concretas de la colección de interfaces.

- Algoritmos: Estos son los métodos que realizan los cálculos de utilidad, tales como búsqueda y clasificación, sobre los objetos que implementan interfaces de colección. Los algoritmos se dice que son polimórficos, es decir, el mismo método puede ser utilizado en diferentes implementaciones de la interfaz.

Collection

Además del Java Collections Framework, los ejemplos más conocidos de Frameworks de colecciones son la Standard Template Library (STL) de C++ y la jerarquía de colecciones de Smalltalk.

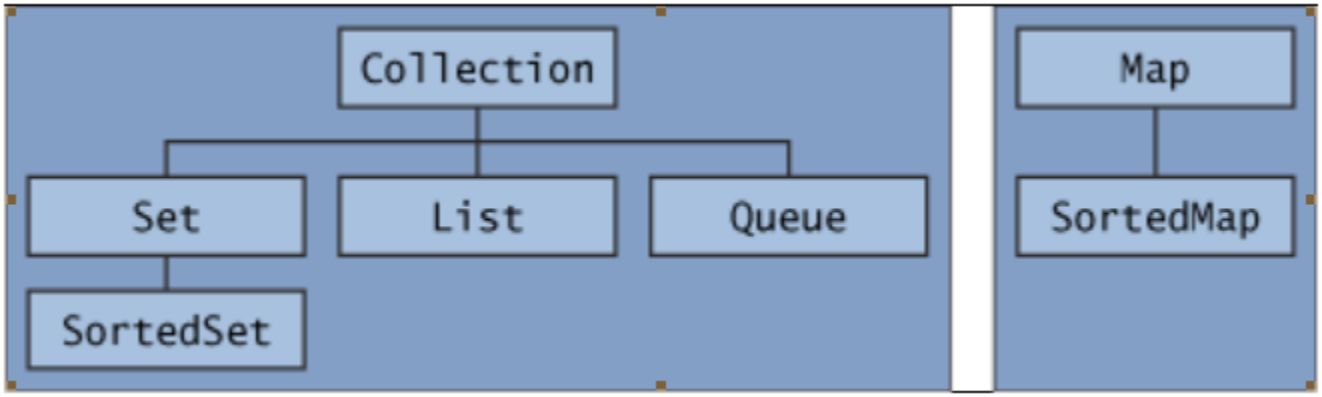
Java Collections Framework (JCF)

Se introdujo a partir de la versión 1.2 del JDK se trata de un conjunto de clases e interfaces para mejorar la capacidad del lenguaje respecto a las estructura de datos. Provee los siguientes beneficios:

- Reduce el esfuerzo de programación: Al proporcionar estructuras de datos y algoritmos útiles permitiendo al programador concentrarse en las partes más importantes de su programa.
- Calidad y velocidad del programa: Proporciona un alto rendimiento y alta calidad en las implementaciones de las estructuras, datos y algoritmos.
- Diseñadores y programadores no tienen que reinventar la rueda cada vez que tengan que crear una aplicación basada en colecciones.

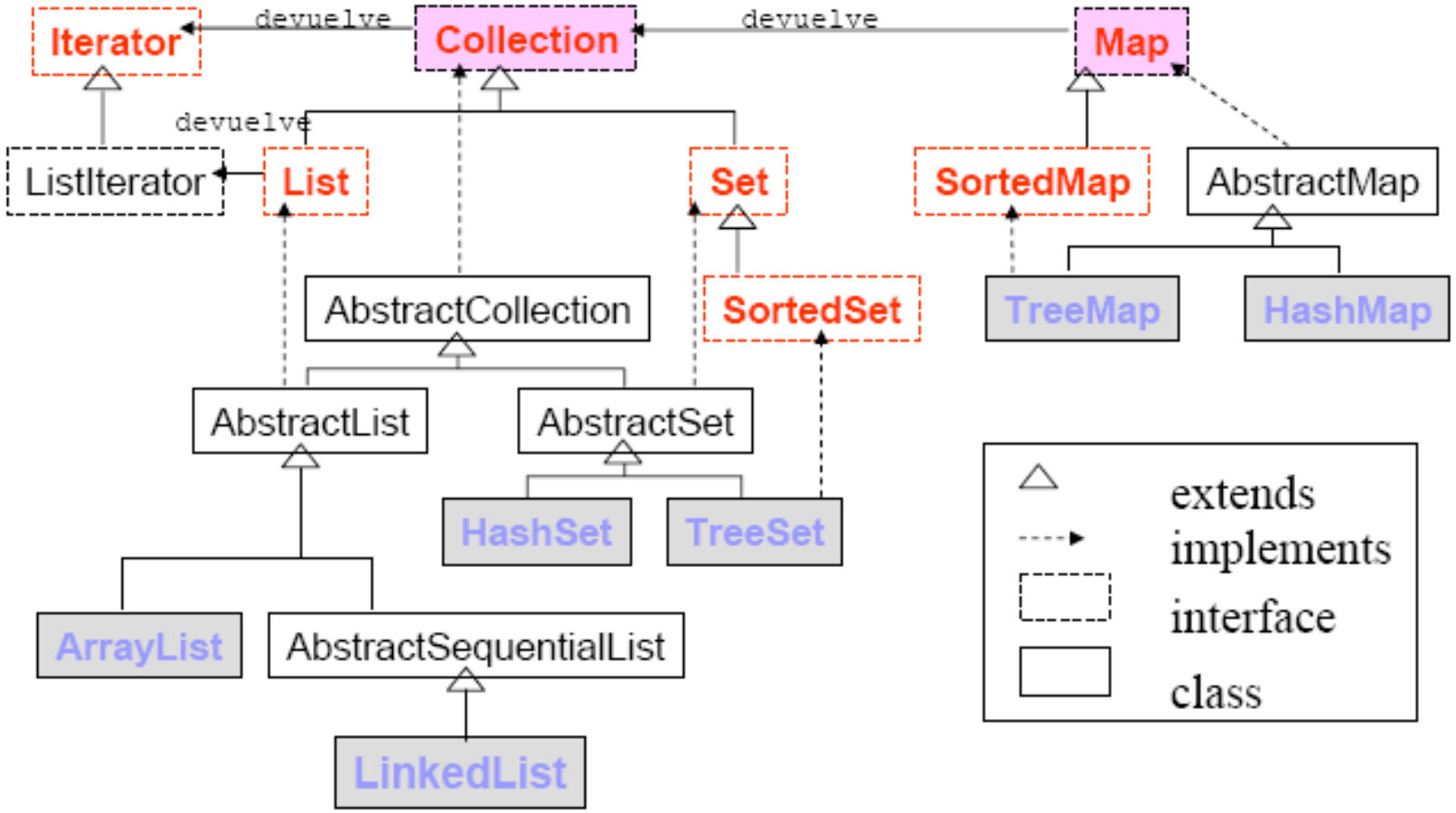
Collection. Interfaces

La Figura muestra la jerarquía del núcleo de interfaces de la JCF. Estas interfaces permiten manipular de forma independiente los detalles de su representación.



Set en un tipo especial de Colecction, SortedSet en un tipo especial de Set, y así sucesivamente. Notar que la jerarquía se compone de dos árboles.

Collections. JCF



Collection

Todas las interfaces son genéricas. A continuación se muestra un ejemplo de declaración de la interfaz Collection:

public interface Collection<E> …

<E> nos dice que la interfaz es genérica. Cuando se declara una instancia de Collection se puede y debe especificar el tipo de objeto que será contenido por la colección. Especificar el tipo permitirá al compilador verificar (en tiempo de compilación) que el tipo de objeto que coloquemos en la colección sea correcto, reduciendo por lo tanto, errores en tiempo de ejecución. Por ejemplo:

public Collection curso<Alumnos>; //La colección curso contendrá objetos de tipo Alumno.

Collection

A continuación se hará una descripción de las interfaces que conforman el núcleo de la JCF.

Collection: Representa un grupo de objetos denominados como elementos de la colección. La interfaz Collection es el denominador común que todas las colecciones implementan. Algunos tipos de colecciones permiten duplicar elementos, y otras no. Algunas son ordenadas y otras desordenadas. La plataforma Java no provee ninguna implementación directa de esta interface pero provee implementaciones de subinterfaces más especificas, tales como Set y List.

Collection

A continuación se muestra la **interfaz Collection**

```
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(E element);
    boolean remove(Object element);
    Iterator<E> iterator();
     // Bulk operations--------------------------------
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
     // Array operations-------------------------------
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Collection

La interfaz contiene métodos que nos dicen:

- cuantos elementos tiene la colección (size,isEmpty),
- si la colección contiene un determinado objeto (contains),
- agregar o eliminar elementos de la colección (add, remove),
- y provee un iterador de la colección (iterator).
- containsAll – retorna true si la colección contiene todos lo elementos de la colección pasado por parámetro
- addAll – agrega todos lo elementos de la colección pasada por parámetro a la colección
- removeAll – elimina de la colección todos los elementos que estén contenidos en la colección pasada por parámetro.
- retainAll – elimina de la colección todos lo elementos que no contenga la colección pasada por parámetro. Es decir, se queda sólo con los elementos que están en la colección y también están contenidos en la colección pasada por parámetro.
- clear – elimina todos los elementos contenidos en la colección.

Collection

El método toArray proporciona un puente entre colecciones y las viejas APIs que esperan arrays de entrada. Permite que el contenido de una colección pueda llevarse a un array de objetos. Por ejemplo, supongamos que c es una colección que solo contiene Strings; el siguente código muestra como volcar el contenido de c en un nuevo array de Strings.

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

Collection. Set

Set: No puede contener elementos duplicados. Esta interfaz modela la abstracción matemática de un conjunto y es usada para representar conjuntos.

Ejemplo: las materias a las que se inscribió un estudiante, los procesos que se ejecutan en una máquina, etc.

La interfaz Set contiene solo los métodos heredados de Collection y agrega como restricción la prohibición de agregar elementos duplicados.

Collection. Set

La plataforma Java contiene tres implementaciones de propósito general de Set: HashSet, TreeSet y LinkedHashSet.

- **HashSet**: Almacena sus elementos en una tabla hash, es la implementación con mejor desempeño, sin embargo no garantiza relación en el orden de iteración.
- **TreeSet**: Almacena sus elementos en un árbol, ordena los elementos en base a sus valores, es más lento que HashSet. Verdaderamente TreeSet implementa la interfaz SortedSet, la misma será descripta posteriormente.
- **LinkedHashSet**: Almacena sus elementos en una tabla hash con una lista enlazada que permite recorrerla. Ordena sus elementos en el orden en que fueron ingresados.

Collection. Set. Ejemplos

Ejemplo 1: Supongamos que se dispone de una colección, c, y se quiere crear otra a partir de sus elementos pero eliminando todos los duplicados:

```
Collection<Type> noDups = new HashSet<Type>(c);
```

Ejemplo 2: Lo mismo que lo anterior pero preservando el orden de la colección original:

```
Collection<Type> noDups = new
LinkedHashSet<Type>(c);
```

Collection. Set

La interfaz Set es la siguiente:

```
public interface Set<E> extends Collection<E> {
    // Basic operations
   int size();
   boolean isEmpty();
   boolean contains(Object element);
   boolean add(E element);   //optional
   boolean remove(Object element); //optional
   Iterator<E> iterator();
   // Bulk operations
   boolean containsAll(Collection<?> c);
   boolean addAll(Collection<? extends E> c); //optional
   boolean removeAll(Collection<?> c);        //optional
   boolean retainAll(Collection<?> c);        //optional
   void clear();                              //optional
   // Array Operations
    Object[] toArray();    <T> T[] toArray(T[] a); }
```

Collection. Set

Contiene los siguientes métodos:

- **size** devuelve el número de elementos del conjunto (su cardinalidad).
- **isEmpty** retorna true si Set está vacío o false si contiene elementos.
- **add** agrega el elemento especificado si no está presente y devuelve un valor booleano que indica si el elemento se añadió.
- Del mismo modo, el método eliminar elimina el elemento especificado del conjunto, devuelve un valor booleano si el elemento estaba presente.
- **iterador** retorna un Iterador.

Suponiendo que s1 y s2 son Sets:

- s1.**containsAll**(s2) – retorna true si s2 es un subconjunto de s1. (s2 es un subconjunto de s1 si s1 contiene todos los elementos de s2.)
- s1.**addAll**(s2) – transforma s1 en la unión de s1 and s2. (La unión entre dos conjuntos es el conjunto que contiene todos los elementos contenidos en cada conjunto.)
- s1.**retainAll**(s2) – transforma s1 en la intersección de s1 and s2. (La inserción entre dos conjuntos es el conjunto que contiene solo los elementos comunes a los dos conjuntos)
- s1.**removeAll**(s2) – transforma s1 en la diferencia asimétrica entre los conjuntos s1 y s2.(Por ejemplo, la diferencia de s1 menos s2 es el conjunto que contiene todos los elementos que están en s1 y no en s2.)

Collection. Set. Ejemplo

Ejemplo: El código que continúa toma las palabras de una lista de argumentos e imprime todas las palabras que se repiten, el número de palabras distintas y una lista de palabras con duplicados eliminados.

```
import java.util.*;


public class FindDups {
  public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            if (!s.add(a))
            System.out.println("Duplicate detected: " + a);
            System.out.println(s.size() + " distinct words: " + s);
  }
}
```

Collection. Set. Ejemplo

Notar que en el código anterior siempre refiere a la colección por su tipo de interfaz y no por su implementación. Se recomienda esta práctica de programación porque da la flexibilidad para cambiar las implementaciones por el mero hecho de cambiar el constructor. Si cada variable usada para almacenar una colección es declarada por el tipo de implementación de la colección en lugar del tipo interfaz, todas esas variables deben cambiarse en el orden que el tipo de implementación cambie.

El tipo de implementación de Set en el ejemplo anterior es HashSet, lo que hace que no haya garantías en cuanto al orden de los elementos del conjunto. Si se desea que el programa al imprimir la lista de palabras lo haga en orden alfabético, simplemente se debe cambiar la implementación al tipo TreeSet.

Collection. Set. Ejemplo

Ejemplo ilustrativo:

Si la entrada del código anterior es:

i came i saw i left

Si se implementa con HashSet la salida sería:

Duplicate detected: i

Duplicate detected: i

4 distinct words: **[i, left, saw, came]**

Si la implementación es TreeSet la salida por consola es la siguiente:

Duplicate detected: i

Duplicate detected: i

4 distinct words: **[came, i, left, saw]**

Collection. SortedSet

Mantiene sus elementos ordenados en orden ascendente. Provee operaciones adicionales que permiten tomar ventaja de la ordenación. Sorted sets son usados para conjuntos ordenados, tales como listas de palabras.

El código de la interfaz SortedSet en el siguiente:

```
public interface SortedSet<E> extends Set<E> {
// Range-view ----------
  SortedSet<E> subSet(E fromElement, E toElement);
  SortedSet<E> headSet(E toElement);
  SortedSet<E> tailSet(E fromElement);
// Endpoints ----------
  E first();
  E last();
// Comparator access ----------
  Comparator<? super E> comparator();
}
```

Collection. SortedSet

- El método comparator permite obtener el objeto pasado al constructor para establecer el orden. Si se ha utilizado el orden natural definido por la interface Comparable, este método retorna null.

- Los métodos first y last retornan el primer y último elemento del conjunto.

- Los métodos headSet, subSet y tailSet sirven para obtener subconjuntos al principio, medio, y al final del conjunto original (los dos primeros no incluyen el límite superior especificado).

Collection. List

List: es una colección ordenada, también llamada secuencia. Las listas pueden contener elementos duplicados. Generalmente el usuario de una lista desea tener un control preciso sobre el lugar en donde la lista ingresa cada elemento, pudiendo acceder a estos por su index (posición entera).

Collection. List

En adición a las operaciones heredadas desde Collection, la interfaz List incluye las siguientes operaciones:

- Acceso posicional – permite manipular elementos basados en su posición numérica en la lista
- Búsqueda – búsquedas de objetos específicos en la lista y retorna su posición numérica.
- Iteración – extiende la semántica de Iterator para aprovechar la secuencia natural de la lista.
- Rango – realiza diversas operaciones arbitrarias de rango sobre la lista.

Collection. List

A continuación se muestra la interfaz List:

```
public interface List<E> extends Collection<E> {
      // Positional access
        E get(int index);
        E set(int index, E element);    //optional
        boolean add(E element);         //optional
        void add(int index, E element); //optional
        E remove(int index);            //optional
        boolean addAll(int index, Collection<? extends E> c);//optional
   // Search
         int indexOf(Object o);
        int lastIndexOf(Object o);
      // Iteration
        ListIterator<E> listIterator();
        ListIterator<E> listIterator(int index);
      // Range-view
        List<E> subList(int from, int to);
}
```

Collection. List

La plataforma Java contiene dos implementaciones de propósito general de List.

ArrayList, que es generalmente la implementación de mejor desempeño, y **LinkedList** que ofrece un mejor rendimiento en determinadas circunstancias.

La diferencia está en que la primera almacena los elementos de la colección en un array de Objects, mientras que la segunda los almacena en una lista enlazada.

Los arrays proporcionan una forma de acceder a los elementos mucho más eficiente que las listas enlazadas. Sin embargo tienen dificultades para crecer (hay que reservar memoria nueva, copiar los elementos del array antiguo y liberar memoria) y para insertar y/o borrar elementos (hay que desplazar en un sentido u en otro los elementos que están detrás del elemento borrado o insertado).

Las listas enlazadas sólo permiten acceso secuencial, pero tienen una gran flexibilidad para crecer, para borrar y para insertar elementos.

El optar por una implementación u otra depende del caso concreto de que se trate.

Collection. List

Métodos y operaciones:

- Las operaciones add y addAll siempre agregan el o los nuevos elementos en el final de la lista. La operación remove siempre elimina la primera ocurrencia del elemento especificado de la lista.
- Los nuevos métodos add y addAll con un argumento adicional permiten insertar elementos en una posición determinada, desplazando el elemento que estaba en esa posición y los siguientes.
- Los métodos get y set permiten obtener y cambiar el elemento en una posición dada.
- Los métodos set y remove retornan el valor viejo antes de sobrescribirlo o eliminarlo.
- Los métodos indexOf y lastIndexOf permiten saber la posición de la primera o la última vez que un elemento aparece en la lista; si el elemento no se encuentra devuelve -1.
- El método subList devuelve una “vista” de la lista, desde el elemento fromIndex inclusive hasta el toIndex exclusive.

Collection. List

List proporciona un iterador más rico, llamado ListIterator, que le permite recorrer la lista en ambos sentidos, modificar la lista durante la iteración, y obtener la posición actual del iterador. La interfaz ListIterator es la siguiente:

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

Collection. ArrayList

• Un ArrayList es un *array dinámico.* No tiene restricciones de capacidad. Su tamaño se ajusta de forma dinámica.

• Constructor por defecto: new ArrayList(). Inicialmente, la capacidad de un ArrayList creado así es 0.

• Los elementos dentro de un ArrayList son Objetos. No pueden ser de tipo básico, pero pueden ser de cualquier tipo de objeto.

• La clase ArrayList forma parte del paquete java.util

• Para poner un elemento dentro de esta estructura, usamos el método *add* y para recoger un elemento usamos el método *get.*

Collection. ArrayList

- int size() -- El tamaño **actual** (puede ser 0)

- void add(obj) -- Añade un objeto al final del ArrayList, incrementando su tamaño en 1. obj es un objeto.

- Object get(N) -- Devuelve el elemento almacenado en la posición N. N debe estar entre 0 y size()-1.

- void set(index, obj) – Sustituye el elemento en la posición index por el objeto obj (index tiene que ser entre 0 y size()-1) dentro del ArrayList, sustituyendo el elemento previamente almacenado a la posición N. Es equivalente a A[N] = obj para un array A.

- Object remove(index) -- Elimina el elemento a la posición index (index entre 0 y size()-1).

Devuelve el objeto eliminado Los elementos después de este objeto están rebajados de una posición. El tamaño del ArrayList disminuye de 1.

- int indexOf(obj) -- Busca el objeto obj dentro del ArrayList, y si lo encuentra, devuelve la posición donde lo ha encontrado. Si no, devuelve -1.

Collection. ArrayList

Iterador

```
public static void main(String[ ] args) {
      ArrayList thisArrayList = new ArrayList();
      thisArrayList.add("hello");
      thisArrayList.add(",");
      thisArrayList.add("are");
      thisArrayList.add("you");
      thisArrayList.add("?");

      Iterator it = thisArrayList.iterator();
        while (it.hasNext())
              System.out.print(it.next()+" ");
}
```

Collection. Queue

Se conoce como Queue a una colección especialmente diseñada para ser usada como almacenamiento temporario de objetos a procesar. Además de las operaciones básicas de Collection, provee operaciones adicionales de inserción, extracción e inspección.

Normalmente, pero no necesariamente, las colas ordenan sus elementos como FIFO (first-in, first-out/primero en entrar, primero en salir). Entre las excepciones están las colas de prioridad, que ordenan sus elementos de acuerdo a una comparación o en su orden natural. Sea cual sea el orden utilizado, la cabeza (head) de la cola es el elemento que será removido si se invoca remove o poll. En una cola FIFO, los nuevos elementos son insertados al final de la cola.

Otros tipos de colas pueden utilizar diferentes formas de colocación. Cada implementación debe especificar sus propiedades de ordenamiento.

Collection. Queue

La interfaz Queue es la siguiente:

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

Collection. Queue

Métodos:

- El método add, que Queue hereda de Collection, inserta un elemento al menos que viole su capacidad, en ese caso arroja IllegalStateException.
- El método offer, usado exclusivamente en colas delimitadas, difiere del método add sólo en la medida que indica la falla de insertar un elemento retornando falso.
- Los métodos remove and poll que eliminan o retornan la cabeza de la cola, difieren sólo cuando la cola está vacía. En ese caso, remove lanza NoSuchElementException, mientras que poll retorna null.
- Los métodos element y peek retornan, pero no eliminan, la cabeza de la cola. Se diferencian entre sí precisamente en el mismo caso que remove y poll: si la cola está vacía, element lanza NoSuchElementException, mientras que peek retorna null.

Collection. Map

Es un objeto que mapea claves (keys) con valores. Map no puede contener claves duplicadas, cada clave puede mapear a un solo valor.

Modela la abstracción matemática de una función.

Collection. Map

La interfaz Map es la siguiente:

```
public interface Map<K,V> {
       // Basic operations
         V put(K key, V value);
         V get(Object key);
         V remove(Object key);
         boolean containsKey(Object key);
         boolean containsValue(Object value);
         int size();
         boolean isEmpty();
       // Bulk operations
         void putAll(Map<? extends K, ? extends V> m);
         void clear();
       // Collection Views
         public Set<K> keySet();
         public Collection<V> values();
         public Set<Map.Entry<K,V>> entrySet();
       // Interface for entrySet elements
         public interface Entry {
                          K getKey();
                  V getValue();
                  V setValue(V value);    }
```

Collection. Map

Métodos:

- El método entrySet devuelve una "vista" del Map como Set. Los elementos de este Set son referencias de la interface Map.
- getKey permite obtener el valor a partir de la clave.
- keySet devuelve un Set que contiene todas las claves.
- values devuelve una Collection que contiene todos los valores.
- put permite añadir una pareja clave-valor, mientras que putAll agrega todos los elementos de un Map a otro (los pares con clave nueva se añaden. Los pares con clave ya existente los valores nuevos sustituyen a los antiguos).
- remove elimina una pareja clave- valor a partir de la clave.

Collection. SortedMap

Es un Map que contiene sus claves ordenadas en orden ascendente.

Provee operaciones adicionales que permiten aprovechar la ordenación.

Sorted maps son usados para colecciones de clave/valor tales como, diccionarios y agendas de teléfono.

Collection. SortedMap

La interfaz SortedMap es la siguiente:

```
public interface SortedMap<K, V> extends Map<K, V>
{
   Comparator<? super K> comparator();
   SortedMap<K, V> subMap(K fromKey, K toKey);
   SortedMap<K, V> headMap(K toKey);
   SortedMap<K, V> tailMap(K fromKey);
   K firstKey();
   K lastKey();
}
```

Collection. Map

Las tres implementaciones de propósito general de Map son:

- HashMap, TreeMap y LinkedHashMap.

Si se requieren las operaciones de SortedMap o iterar sobre colecciones ordenadas por su clave, usar **TreeMap**; si en cambio, se busca máxima velocidad y no preocupa el orden de iteración, usar **HashMap**.

Respecto a esto, la situación de Map es análoga a Set, todo lo aplicado en las implementaciones de Set también se aplica.

Collection. HashMap

Un HashMap es un array asociativo (o hash table).

Contiene asociaciones <clave,valor>, donde la clave es única y permite acceder al valor.

Ejemplos de asociaciones:

- La lista de alumnos, cada alumno es accesible por su NIA.
- El directorio telefónico, cada número se accede por los apellidos y la dirección.
- Un diccionario, cada definición se accede por lemma.

Collection. HashMap

- int size() -- El tamaño actual = número de mappings <clave,valor>
- boolean isEmpty() - Devuelve true si el map es vacío
- void put(clave,valor) - Añade el mapping <clave,valor> a map, donde clave y valor son objetos. Si clave ya existe, sustituye su valor por valor.
- Object get(clave) - Devuelve el valor asociado con clave, o null si clave no existe dentro de map.
- Set keySet( ) - Devuelve un Set (conjunto de elementos individuales únicos) correspondiente a las claves del map. Podemos iterar sobre este
- Set (con un Iterator)
- boolean containsKey(clave) - Devuelve true si clave existe dentro de map como clave
- boolean containsValue(valor) - Devuelve true si valor existe dentro de map como valor
- Object remove(obj) - Elimina la asociación cuya clave es obj. Devuelve el valor asociado con esta clave antes de eliminar.
- Collection values( ) - Devuelve una Collection (conjunto de elementos individuales no únicos) de los valores que hay dentro del hashmap.
- Podemos iterar sobre esta Collection.
- void clear() - elimina todos los mappings <clave,valor> del array.

Collection. HashMap

Iterador

```
System.out.println("The elements of HashMap are");
Set set= hashmap.keySet();
Iterator iter = set.iterator();

while(iter.hasNext()){
  Object clave = iter.next();
  Object valor = hashmap.get(clave);
  System.out.println("Key: " + clave + " Value: " + valor);
}
```

Para recorrer un hashmap:

– Conseguimos el conjunto de claves

– Recorremos este conjunto, y para cada clave, sacamos su valor correspondiente en el hashmap

Collection

Ordenamiento de objetos

Si la lista se compone de elementos de tipo String, será ordenada en orden alfabético. En caso de que se componga de elementos de tipo Date, será ordenada en orden cronológico. Esto se debe a que tanto Date como String implementan la interfaz Comparable. Implementar Comparable permite proporcionar un ordenamiento natural para una clase, permitiendo que los objetos de esa clase se clasifiquen automáticamente.

Collection

El siguiente cuadro resume algunas de las clases más importantes de la plataforma Java que implementan Comparable.

Classes Implementing Comparable

Class	Natural Ordering
`Byte`	Signed numerical
`Character`	Unsigned numerical
`Long`	Signed numerical
`Integer`	Signed numerical
`Short`	Signed numerical
`Double`	Signed numerical
`Float`	Signed numerical
`BigInteger`	Signed numerical
`BigDecimal`	Signed numerical
`Boolean`	`Boolean.FALSE < Boolean.TRUE`
`File`	System-dependent lexicographic on path name
`String`	Lexicographic
`Date`	Chronological

Collection

La clase siguiente representa el nombre de una persona e implementa Comparable.

```
import java.util.*;
public class Name implements Comparable<Name> {
   private final String firstName, lastName;
   public Name(String firstName, String lastName) {
      if (firstName == null || lastName == null)
         throw new NullPointerException();
       this.firstName = firstName;
      this.lastName = lastName;
   }
   public String firstName() { return firstName; }
   public String lastName()  { return lastName;  }

     public int compareTo(Name n) {
      int lastCmp = lastName.compareTo(n.lastName);
      return (lastCmp != 0 ? lastCmp :
              firstName.compareTo(n.firstName));
   }}
```

El método *comapareTo* devuelve un entero menor que cero si el objeto string es menor (en orden alfabético) que el string dado, cero si son iguales, y mayor que cero si el objeto string es mayor que el string dado

Collection. Recorrer Colecciones

Existen dos maneras de recorrer colecciones:

1_ usando for-each Construct (por cada uno construir)

2_ usando Iterators.

1 . for-each Construct

Permite recorrer una colección o un array mediante un for loop. El siguiente código usa for-each construct para imprimir cada elemento de una colección.

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

Collection. Recorrer Colecciones

2 . Un iterador es un objeto que permite recorrer una colección y eliminar elementos, si es deseado. Para obtener un iterador para una colección debemos utilizar el método iterator. La Interfaz Iterator es la siguiente:

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

- El método hasNext retorna true si el iterador contiene más elementos
- El método next retorna el próximo elemento del iterador.
- El método remove borra el último elemento retornado por next. El método remove debe ser llamado solo una vez por cada llamado al método next, de lo contrario se dispara una exception.

Collection. Recorrer Colecciones

Utilizar Iterator en lugar de for-each construct cuando se necesite:

- Eliminar el elemento actual. For-each construct oculta el iterator, por lo que no permite borrar. Por lo tanto, for-each construct no debe utilizarse para filtrado.

- Iterar a través de múltiples colecciones en paralelo.

Collection. Recorrer Colecciones

En el siguiente método se muestra cómo utilizar un iterador para filtrar una colección determinada, es decir, recorrer una colección eliminando elementos específicos.

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

Este fragmento de código es polimórfico, lo que significa que funciona para cualquier colección, independientemente de la aplicación. Este ejemplo demuestra lo fácil que es escribir un algoritmo polimórfico usando JCF.

Collection. Sumario de implementaciones

La JCF provee diferentes implementaciones de propósito general

General-purpose Implementations

Interfaces	Implementations				
	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

Collection. Sumario de implementaciones

- Para la interfaz Set, HashSet es la implementación usada normalmente.
- Para la interfaz List, ArrayList es la implementación usada normalmente.
- Para la interfaz Map, HashMap es la implementación usada normalmente
- Para la interfaz Queue, LinkedList es la implementación usada normalmente.

Cada una de las implementaciones provee todas las operaciones opcionales contenidas en su interfaz.

Collections. Jerarquía