Programación 2

Java Collections Framework

Profesores:
Ismael Figueroa - ifigueroap@gmail.com
Eduardo Godoy - eduardo.gl@gmail.com

¿Qué es una Collection ?

Definición

Una *Collection* es simplemente ***un objeto que agrupa múltiples objetos en una sola unidad***.

- Se usan para **almacenar**, **manipular**, y **comunicar** datos agregados.
- Representan datos asociados de forma natural en el dominio del problema: una mano de cartas, un listado teléfonico, etc.
- Ya hemos usado una collection: `List` y `ArrayList`!

¿Qué es el Java Collections Framework?

Es una **arquitectura unificada** para representar y manipular collections, que consiste de:[1]

- **Interfaces**: son representaciones abstractas de las colecciones. Permiten la manipulación de colecciones sin importar su implementación actual, explotando los mecanismos de polimorfismo.
- **Implementaciones**: son implementaciones concretas de las colecciones, que implementan las interfaces. Sirven como código general y reutilizable que podemos aprovechar en todos nuestros programas.
- **Algoritmos**: son métodos para realizar operaciones útiles con las colecciones: buscar, ordenar, etc. Los algoritmos también aprovechan el polimorfismo para realizar *operaciones genéricas*.

[1] `https://docs.oracle.com/javase/tutorial/collections/intro/index.html`

¿Qué beneficios tiene el Java Collections Framework?

- **Reducir el esfuerzo de programación:** al usar colecciones pre-existentes podemos enfocarnos en las operaciones *interesantes y relevantes* de nuestro programa, más que en los detalles de cómo almacenar y obtener los datos.

- **Aumentar la velocidad y calidad de programación:** las implementaciones estándar son de alta calidad y performance. En caso de necesidad pueden ser cambiadas sin alterar el resto de los programas.

¿Qué beneficios tiene el Java Collections Framework?

- **Permite interoperabilidad entre programas:** distintas aplicaciones, incluso desarrolladas de manera independiente, pueden *intercambiar datos de manera transparente* mediante el paso de colecciones como parámetros.
- **Reduce el costo de aprendizajey diseño de APIs:** cada vez que desarrollamos un sistema, y sus respectivas interfaces públicas, podemos simplemente utilizar collecciones para manipular agregaciones de datos.
- **Fomenta la reutilización del software:** además de las implementaciones estándar, nuevas implementaciones se pueden agregar de manera transparente, siempre que cumplan con las interfaces requeridas.

Interfaz java.util.Collection

La primera interfaz es `java.util.Collection`[2].

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

Esta interfaz define **la raíz de la jerarquía de interfaces de colecciones**. Define los métodos **obligatorios** que toda colección **debe** implementar. También define métodos **opcionales** que toda colección **puede** implementar.

- Los métodos opcionales son aquellos que indican `throws UnsupportedOperationExpression`.
- Un método opcional podría o no arrojar esa excepción. Depende exclusivamente de la implementación.

[2] https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

En Java las Colecciones son Homogéneas

El código:

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

Indica que el tipo de dato `Collection` **toma como parámetro otro tipo de dato** `E`. Esto es necesario porque la colección necesita saber el tipo de los elementos que va a contener.

Siempre que definamos una variable que sea una colección **debemos especificar el tipo de elemento que contiene**

Interfaz java.util.Collection

```
class Ejemplo {

  public static void main(String[] args) {
    /** La variable 'c1' es de tipo Collection<String>
    Esto significa que es una agrupación de String */
    Collection<String> c1;

    /** La variable 'c2' es de tipo Collection<Double>
    Esto significa que es una agrupación de Double */
    Collection<Double> c2;

    /** La variable 'c3' es de tipo Collection<Persona>
    Esto significa que es una agrupación de Persona */
    Collection<Persona> c3;
  }
}
```

Métodos más relevantes de java.util.Collection

Toda implementación de Collection<E> **debe** implementar los siguientes métodos:

- `boolean contains(Object o)`: retorna `true` si existe un elemento en la colección que sea igual a `o`. Si no, retorna `false`. La igualdad se compara con el método `equals`.
- `boolean isEmpty()`: retorna `true` si la colección no tiene elementos. En otro caso retorna `false`.
- `int size()`: retorna la cantidad de elementos almacenados en la colección.

Métodos más relevantes de java.util.Collection

Una implementación de `Collection<E>` **puede** implementar los siguientes métodos opcionales. Se dice que la colección es **mutable**.

- `boolean add(E e)`: este método asegura que la colección contendrá el objeto `e` luego de ser invocado. El valor booleano indica si la colección fue cambiada internamente o no. Si la colección no acepta la inclusión de `e` **debe arrojar una excepción**.

- `boolean addAll(Collection<? extends E> c)`: similar a `add`, asegura que todos los elementos del argumento `c`—que es una colección de elementos de tipo `E` o de subtipos de `E`—estarán contenidos en la colección.

Métodos más relevantes de java.util.Collection

Una implementación de `Collection<E>` **puede** implementar los siguientes métodos opcionales. Se dice que la colección es **mutable**.

- `boolean remove(Object o)`: remueve un elemento de la colección, si es que éste es igual a `o`. La igualdad se realiza utilizando `equals`. Si el elemento aparece varias veces en la colección sólo se borrará 1 copia. El booleano indica si el elemento efectivamente fue encontrado y borrado.

- `boolean removeAll(Collection<?> c)`: similar a `remove`, elimina todos los elementos que también están contenidos en colección `c`.

- `void clear()`: Remueve todos los elementos de la colección. La colección quedará vacía luego de invocar `clear`.

Iterando sobre Colecciones

La interfaz `Collection` hereda a `java.util.Iterable`, esto en la práctica significa que todas las colecciones se pueden iterar con el `for` especial:

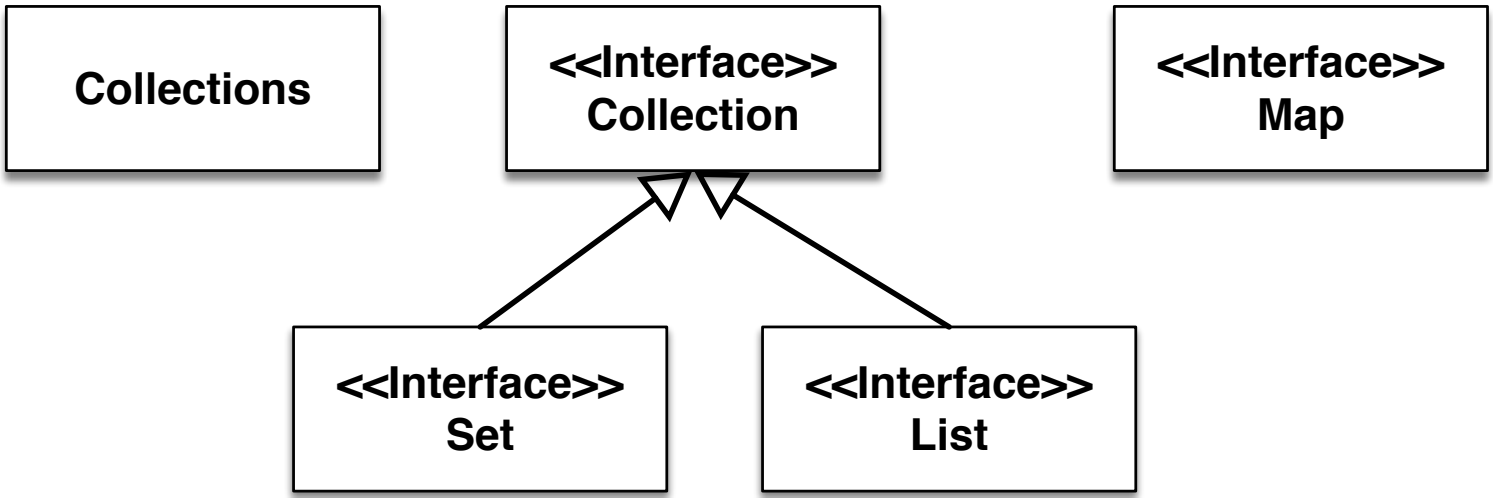
```
Collection<Double> c = ...
for(Double d : c) {
  System.out.println(d);
}
```

Más interfaces y colecciones

Además de `Collection`, tenemos las siguientes interfaces para colecciones esenciales:

- `Set`: es una colección que no puede tener elementos duplicados. Sirve para modelar la idea matemática de **conjunto**. Los elementos no tienen orden ni secuencia específica.
- `List`: es una colección **secuencial** de elementos. Pueden tener elementos duplicados. Los elementos tienen una **posición específica**.
- `Map`: es un objeto que asocia **llaves** con **valores**. También se le conoce como **diccionario**. No puede contener llaves duplicadas. Cada llave está asociada a un solo objeto.

Jerarquía de Interfaces



- `Collection`: es la interfaz general que ya fue descrita.
- `Set`: es la interfaz para colecciones `Set`, que hereda a `Collection`.
- `List`: es la interfaz para colecciones `List`, que hereda a `Collection`.
- `Collections`: es la clase utilitaria `java.util.Collections` con métodos estáticos para manipulación de colecciones.
- `Map`: es la interfaz para definir estructuras `Map`. **Esta interfaz no hereda desde** `Collection`.

Interfaz java.util.Set

```
public interface Set<E> extends Collection<E>
```

Esta interfaz modela el concepto matemático de **conjunto**. Es una colección de elementos donde no existen duplicados. Dos `Set` son iguales solamente si contienen exactamente los mismos elementos.

Protip

A veces queremos eliminar los elementos duplicados de una colección, por ejemplo una `List`. Podemos transformar esa colección en un `Set`, y luego de vuelta en una `List`!

Operaciones sobre Set

Esta interfaz no agrega nuevas operaciones respecto a las heredadas desde `Collection`. Consideremos las variables `Set<E> s1, s2`. Es decir dos conjuntos de elementos de tipo `E`. En base a las operaciones heredadas desde `Collection` tenemos:

- **Unión**: si ejecutamos `s1.addAll(s2)` tendremos que `s1` ahora contiene la **unión** de ambos conjuntos.
- **Intersección**: si ejecutamos `s1.retainAll(s2)` tendremos que `s1` ahora contiene la **intersección** de ambos conjuntos.
- **Diferencia**: si ejecutamos `s1.removeAll(s2)` tendremos que `s1` ahora contiene la **diferencia** entre `s1` y `s2`.
- **Subconjunto**: si ejecutamos `s1.containsAll(s2)` sabremos si `s2` es un **subconjunto** de `s1`.

Ejemplos de Set

```
class EjemploSet {
  public static void main(String[] args) {
    /** Conjunto de asignaturas de la carrera */
    Set<Asignatura> s1;

    /** Conjunto de estudiantes de la carrera */
    Set<Persona> s2;
  }
}
```

Interfaz java.util.List

```
public interface List<E> extends Collection<E>
```

Esta interfaz representa una colección secuencial de elementos. Una lista puede contener elementos duplicados. Además de las operaciones normales de `Collection`, una lista agrega:

- **Acceso basado en posiciones**: los elementos se pueden manipular en base a su posición en la lista.
- **Búsqueda**: al buscar un elemento en la lista se obtiene su posición en ella, si es que está contenido.

Interfaz java.util.List :: Acceso basado en posiciones

Los siguientes métodos permiten la manipulación de los elementos de la lista según su posición:

- `E get(int index)`: retorna el elemento en la posición `index`. Si `index` excede el tamaño de la lista se arroja una excepción.
- `E set(int index, E elem)`: inserta el elemento `elem` en la posición `index`. Retorna el elemento que había antes en esa posición. Arroja excepción si el índice está fuera de rango.
- `boolean add(E e)`: agrega el elemento `e` **al final de la lista**. Si no hay una excepción, siempre retorna `true`.
- `boolean addAll(Collection<? extends E> c)`: agrega todos los elementos de `c` al final de la lista. Los agrega en el orden en que son devueltos por `c`.

Interfaz java.util.List :: Búsqueda

Los siguientes métodos permiten la búsqueda de elementos en la lista:

- `int indexOf(Object o)`: retorna el índice de la primera ocurrencia de o en la lista, o bien retorna -1. Se usa el método `equals` para comparar. En otras palabras, el índice del elemento "más a la izquierda o hacia el comienzo" de la lista.

- `int lastIndexOf(Object o)`: retorna el índice de la última ocurrencia de o en la lista, o bien retorna -1. Retorna el elemento "más a la derecha o hacia el final" de la lista.

Ejemplos de List

```
class EjemploList {
  public static void main(String[] args) {
    /** Listado de asignaturas de la carrera */
    List<Asignatura> l1;

    /** Listado de alumnos de la carrera */
    List<Persona>l2;
  }
}
```

Interfaz java.util.Map

Un Map—conocido simplemente como "mapa"—es un objeto que mapea o asocia **llaves** con **valores**. Se refiere al concepto matemático de "mapeo", como una función que toma una llave como argumento y retorna un valor. Un mapa no puede tener llaves repetidas, y cada llave apunta a un solo valor. Los valores sí pueden ser repetidos para distintas llaves. También se le conoce como **diccionario**.

Su definición como interfaz es:

```
public interface Map<K, V>
```

A diferencia de las interfaces anteriores, Map esta parametrizado por dos tipos, K y V:

- K es el tipo de dato para las llaves
- V es el tipo de dato para los valores

Ejemplos de Map

```
class EjemploMap {
  public static void main(String[] args) {
    /** Mapa que asocia enteros con Strings,
    por ejemplo codigos postales con comunas */
    Map<Integer, String> m1;

    /** Mapa que asocia Strings con Integer,
    por ejemplo nombres de ciudades con su código */
    Map<String, Integer>m2;

    /** Mapa que asocia alumnos con las asignaturas
    que tienen inscritas actualmente */
    Map<Persona, List<Asignatura>>m3;
  }
}
```

Operaciones sobre Map

Los siguientes métodos definen las operaciones fundamentales sobre Map, considerando un mapa Map<K,V>:

- V put(K key, V value): asocia el valor value con la llave key en el mapa. Retorna el valor anterior asociado a esa llave, o bien retorna null si no existía ninguna asociación.
- V get(Object key): retorna el valor del mapa que está asociado a la llave key. Se usa el método equals para comparar key con las llaves existentes. Si el elemento no está en el mapa se retorna null.
- boolean containsKey(Object key): retorna true si el mapa contiene una asociación para la llave key; si no, retorna false.
- boolean containsValue(Object value): retorna true si existe una llave que está asociada a un objeto igual a value. Si no, retorna false. Si existe, no nos dice qué llaves son...

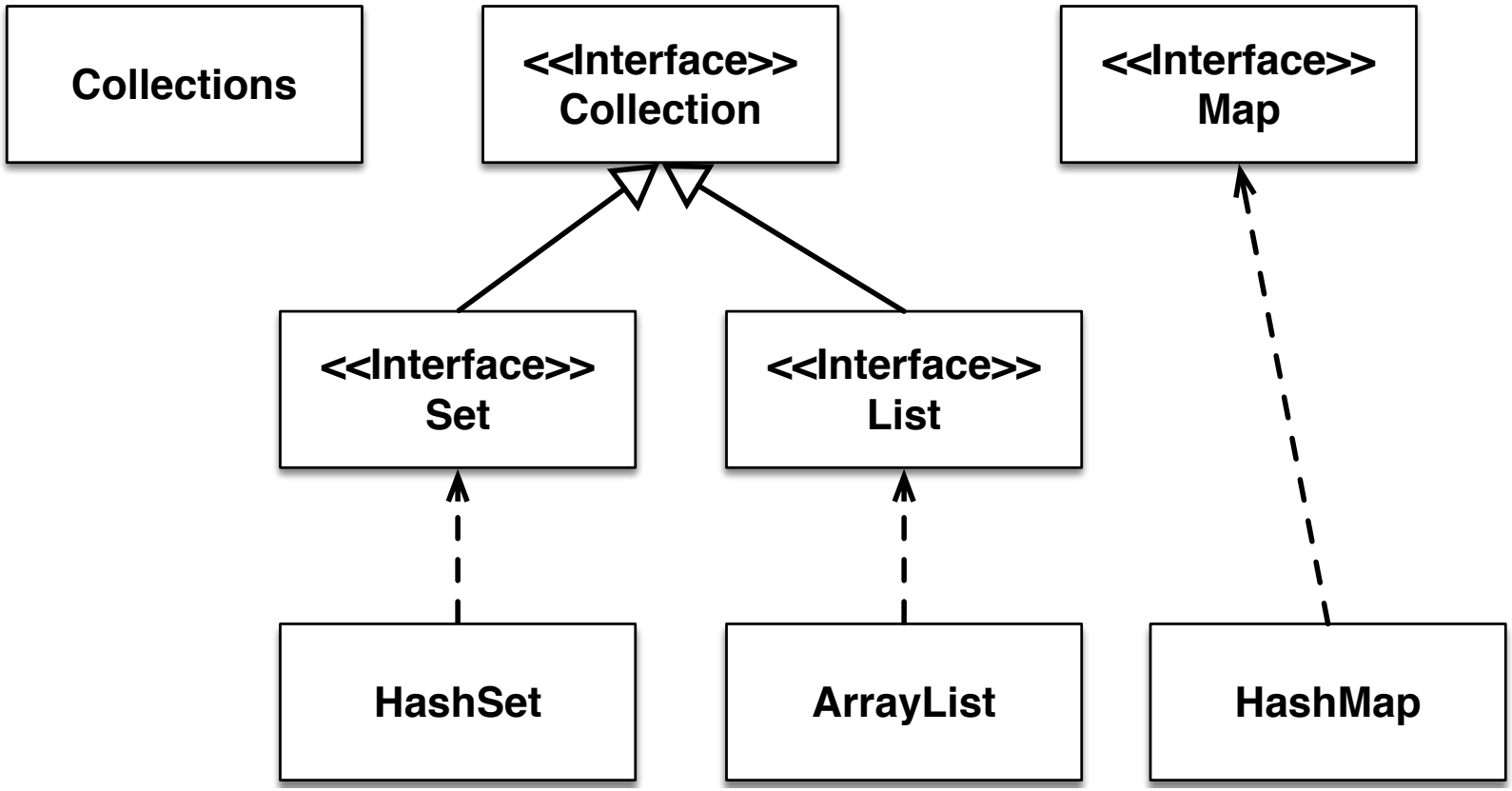
Mapas como Colecciones

Un mapa `Map<K, V>` se puede examinar como colecciones gracias a los siguientes métodos:

- `Set<K> keySet()`: retorna un conjunto con las llaves del mapa.
- `Collection<V> values()`: retorna una colección con los valores del mapa
- `Set<Map.Entry<K,V>> entrySet()`: retorna un conjunto con las asociaciones del mapa. Las asociaciones son instancias de `Map.Entry`; se usan los métodos `getKey` y `getValue` para acceder a la llave y valor.

Implementaciones Estándar

Una de los mayores beneficios del Java Collections Framework es que **incorpora implementaciones estándar para cada una de las colecciones** que hemos descrito anteriormente.



Implementacion de Set: HashSet

```
class EjemploSet {
  public static void main(String[] args) {
    /** Conjunto de asignaturas de la ca,rrera */
    Set<Asignatura> s1 = new HashSet<Asignatura>();
    Asignatura a1 = new Asignatura("Progra2");
    s1.add(a1);
    s1.add(a1);
    System.out.println(s1.size());

    /** Conjunto de estudiantes de la carrera */
    Set<Persona> s2 = new HashSet<Persona>();
    s2.add(new Persona(...));
  }
}
```

Implementacion de List: ArrayList

```
class EjemploList {
  public static void main(String[] args) {
    /** Listado de asignaturas de la carrera */
    List<Asignatura> l1 = new ArrayList<Asignatura>();
    Asignatura a1 = new Asignatura("Progra2");
    l1.add(a1);
    l1.add(a1);
    System.out.println(l1.size());

    /** Listado de alumnos de la carrera */
    List<Persona>l2 = new ArrayList<Asignatura>();
    l2.add(new Persona(...));
  }
}
```

Implementacion de Map: HashMap

```
class EjemploMap {
  public static void main(String[] args) {
    Map<Integer, String> m1 = new HashMap<Integer, String>();
    Map<String, Integer>m2 = new HashMap<String, Integer>();

    /** Mapa que asocia alumnos con las
    asignaturas que tienen inscritas actualmente */
    Map<Persona, List<Asignatura>>m3 =
    new HashMap<Persona, List<Asignatura>>();

    Persona p1 = new Persona("Juanito");
    Asignatura a1 = new Asignatura("Progra2");

    m3.put(p1, new ArrayList<Asignatura>(a1));

    Asignatura a2 = new Asignatura("Calculo");
    m3.get(p1).add(a2);
  }
}
```

¿Interface o Clase Concreta?

Siempre es mejor **programar para satisfacer interfaces generales** más que para clases o implementaciones específicas. **Así podremos aprovechar el polimorfismo basado en interfaces!**

¿Interface o Clase Concreta? —- Variables

```
public static void main(String[] args) {
  Set<Persona> alumnos = new HashSet<Persona>();
  // Mantiene abierta la opcion sobre cual implementación
  // ... de Set utilizar, promueve polimorfismo

}
```

```
public static void main(String[] args) {
  HashSet<Persona> alumnos = new HashSet<Persona>();
  // Deja cerrada la opción sobre la implementación
  // ... no permite polimorfismo, puede ser dificil después
  // ... cambiar por otra implementación
}
```

¿Interface o Clase Concreta? — Métodos

```java
public class Ejemplo {

  public Double promedioNotas(List<Double> notas) {
    // Implementacion usando interfaz List
    // ... Se puede invocar pasando cualquier objeto que implemente List
  }
}
```

```java
public class Ejemplo {

  public Double promedioNotas(ArrayList<Double> notas) {
    // Implementacion usando ArrayList
    // ... Solo se puede invocar para objetos de tipo ArrayList
  }
}
```

Preguntas

Preguntas?