Ciencias de la Computación
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

Curso de **Programación** en **Java**

Juan Francisco Maldonado León
Arquitecto de Software

Ciencias de la Computación
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

Java

Collections

Juan Francisco Maldonado León
Arquitecto de Software

Collections Frameworks

Una colección es simplemente un grupo de objetos de varios elementos representado en una sola unidad.

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

Colección de Contactos Telefonicos.

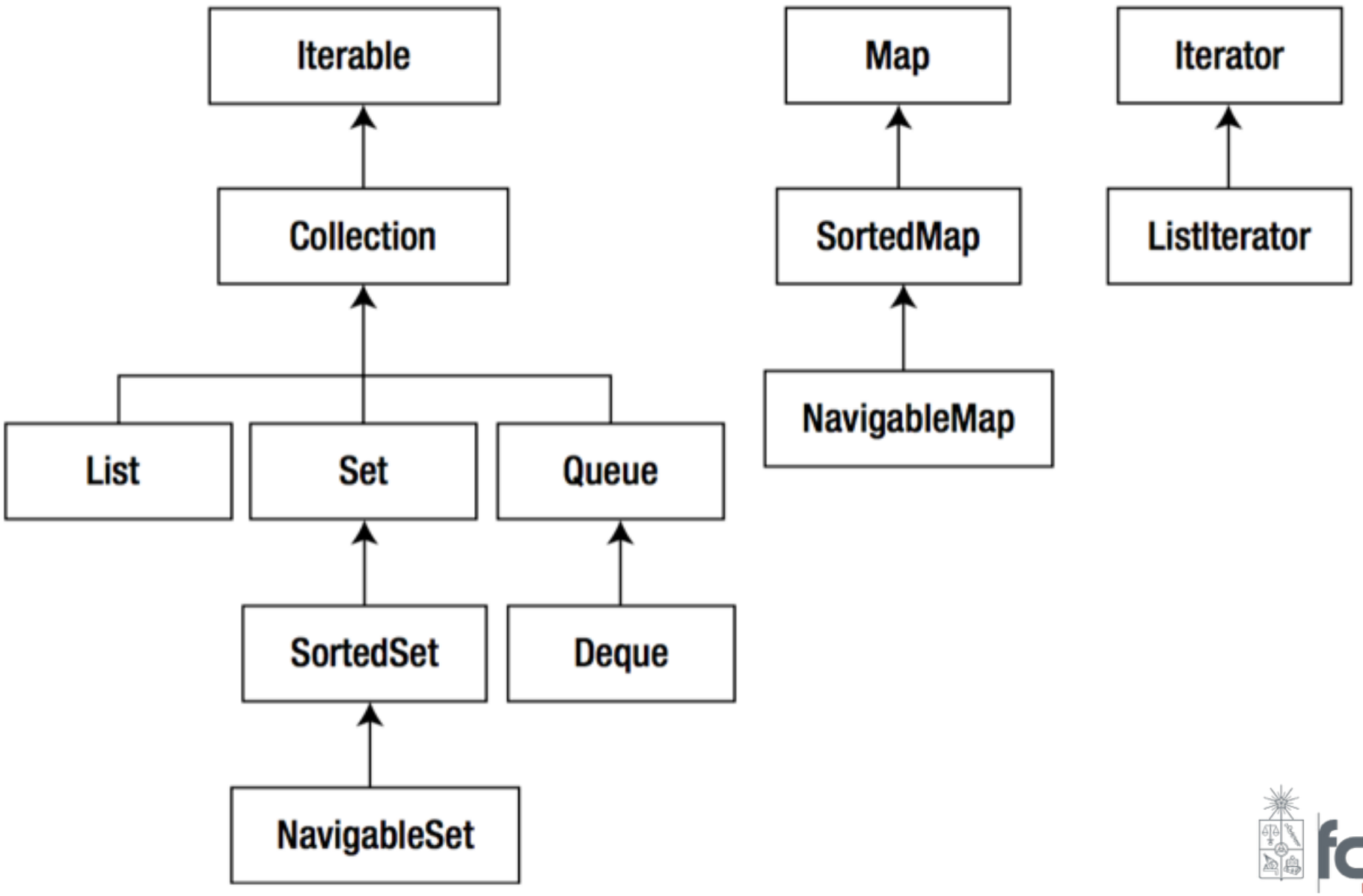
Componentes Principales del Framework

- **Clases abstractas e interfaces**: El Framework de la colecciones tiene muchas clases abstractas e interfaces que proporcionan funcionalidad general. (List, Set, Map, etc)

- **Clases concretas**: Estos son los casos reales de contenedores que va a utilizar en los programas.

- **Algoritmos**: funcionalidades como la clasificación, búsqueda, etc.

Interfaces


Iterable
Collection
List
Set
Queue
SortedSet
Deque
NavigableSet
Map
SortedMap
NavigableMap
Iterator
ListIterator

Introducción a Java

Collections Framework

Abstract Class/Interface	Short Description
`Iterable`	A class implementing this interface can be used for iterating with a `for each` statement.
`Collection`	Common base interface for classes in the collection hierarchy. When you want to write methods that are very general, you can pass the `Collection` interface. For example, `max()` method in `java.util.Collections` takes a `Collection` and returns an object.
`List`	Base interface for containers that store a sequence of elements. You can access the elements using an index, and retrieve the same element later (so that it maintains the insertion order). You can store duplicate elements in a `List`.
`Set`, `SortedSet`, `NavigableSet`	Interfaces for containers that don't allow duplicate elements. `SortedSet` maintains the set elements in a sorted order. `NavigableSet` allows searching the set for the closest matches.
`Queue`, `Deque`	`Queue` is a base interface for containers that holds a sequence of elements for processing. For example, the classes implementing `Queue` can be LIFO (last in, first out— as in stack data structure) or FIFO (first in, first out—as in queue data structure). In a `Deque` you can insert or remove elements from *both* the ends.
`Map`, `SortedMap`, `NavigableMap`	Base class for containers that map keys to values. In `SortedMap`, the keys are in a sorted order. A `NavigableMap` allows you to search and return the closest match for given search criteria. Note that `Map` hierarchy does *not* extend the `Collection` interface.

Interfaz Collection

Method	Short description
`boolean add(Element elem)`	Adds `elem` into the underlying container.
`void clear()`	Removes all elements from the container.
`boolean isEmpty()`	Checks whether the container has any elements or not.
`Iterator<Element> iterator()`	Returns an `Iterator<Element>` object for iterating over the container.
`boolean remove(Object obj)`	Removes the element if `obj` is present in the container.
`int size()`	Returns the number of elements in the container.
`Object[] toArray()`	Returns an array that has all elements in the container.

Clases Concretas

Concrete Class	Short Description
`ArrayList`	Internally implemented as a resizable array. This is one of the most widely used concrete classes. Fast to search, but slow to insert or delete. Allows duplicates.
`LinkedList`	Internally implements a doubly-linked list data structure. Fast to insert or delete elements, but slow for searching elements. Additionally, `LinkedList` can be used when you need a stack (LIFO) or queue (FIFO) data structure. Allows duplicates.
`HashSet`	Internally implemented as a hash-table data structure. Used for storing a set of elements—it does not allow storing duplicate elements. Fast for searching and retrieving elements. It does *not* maintain any order for stored elements.
`TreeSet`	Internally implements a red-black tree data structure. Like `HashSet`, `TreeSet` does not allow storing duplicates. However, unlike `HashSet`, it stores the elements in a sorted order. It uses a tree data structure to decide where to store or search the elements, and the position is decided by the sorting order.
`HashMap`	Internally implemented as a hash-table data structure. Stores key and value pairs. Uses hashing for finding a place to search or store a pair. Searching or inserting is very fast. It does *not* store the elements in any order.
`TreeMap`	Internally implemented using a red-black tree data structure. Unlike `HashMap`, `TreeMap` stores the elements in a sorted order. It uses a tree data structure to decide where to store or search for keys, and the position is decided by the sorting order.

Introducción a Java

Collections Framework

List

Collections

Un objeto List (conocido como secuencia) es una Colección ordenada que puede contener elementos duplicados.

Al igual que los índices de arreglos, los índices de objetos List empiezan desde cero

ArrayList

Collections

ArrayList es una colección de tipo lista. Un ArrayList es una colección de tamaño dinámico y pertenece al package java.util
Las posiciones de un ArrayList van desde el 0 hasta la cantidad de elementos - 1

Inicializar un ArrayList

Collections

Inicializar por defecto

```
ArrayList enteros = new ArrayList();
```

Inicializar por defecto indicando el tipo

```
ArrayList<Integer> enteros = new ArrayList<Integer>();
```

Inicializar indicando la cantidad de elementos

```
ArrayList<Integer> enteros = new ArrayList<Integer>(3);
```

Inicializar un ArrayList

Collections

Un ArrayList solo puede contener tipo de datos complejos, no primitivos.

```
List<int> enteros = new ArrayList<int>();
```

Inicializar un ArrayList

Collections

```
ArrayList<Integer> enteros = new ArrayList<Integer>();
```

Indica el tipo de objeto que contendrá cada posición de la lista

Agregar un elemento

Collections

```
List<String> enteros = new ArrayList<String>();
enteros.add("Diez");
enteros.add("Once");
```

Posición 0

Posición 1

Si no indicamos la posición en el método add, añadirá el elemento en la siguiente posición disponible.

Agregar un elemento

Collections

Utilizando metodo add sobrecargado

```
List<String> enteros = new ArrayList<String>();
enteros.add("Diez");
enteros.add("Once");
enteros.add(2, "Doce");
```

Indica la posición

Indica el elemento ha agregar

Agregar un elemento

Collections

```
List<String> enteros = new ArrayList<String>();
enteros.add("Diez");          posición 1
enteros.add("Once");          posición 2
enteros.add(0, "Doce");       posición 0
```

Obtener un elemento ArrayList

Collections

```
List<String> enteros = new ArrayList<String>();
enteros.add("Diez");
enteros.add("Once");
enteros.get(0);
```

Obtiene el elemento según la posición indicada

Obtener un elemento ArrayList

Collections

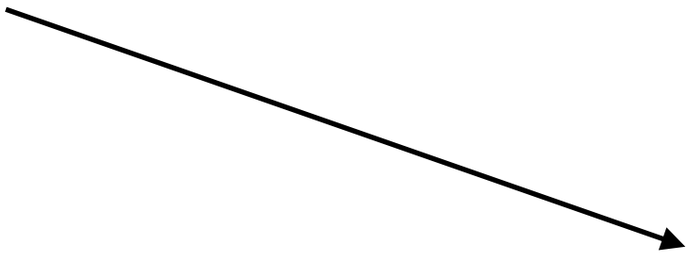
```
List<String> enteros = new ArrayList<String>();
enteros.add("Diez");
enteros.add("Once");
enteros.get(4);
```

java.lang.IndexOutOfBoundsException

Quitar un elemento ArrayList

Collections

```
ArrayList<String> cadenas = new ArrayList<String>();
cadenas.add("Hola");
cadenas.add("Chao");
cadenas.remove(0);
```



Elimina el elemento según la posición indicada

Iterar un ArrayList

Collections

El método size indica la cantidad de elemento de un ArrayList

```
for( int i = 0; i < cadenas.size() ; i++ )
{
    System.out.println( cadenas.get(i) );
}
```

Iterar un ArrayList

Collections

```
for( String cadena : cadenas )
{
    System.out.println( cadena );
}
```

Iterar un ArrayList

Collections

```
Iterator<String> iterador = cadenas.iterator();
while( iterador.hasNext() )
{
    System.out.println( iterador.next() );
}
```