Java Additional Concepts

# Java Collections Framework

The term **Java Collections Framework** refers primarily to a set of interfaces, implementations, and algorithms provided by the Java API (Java Standard Library) for handling collections of objects. It spans multiple packages, though it is mostly concentrated in the `java.util` package. The term "Java Collections Framework" is more of a general term rather than something explicitly defined with clear boundaries within the Java API.

## Core Components of the Java Collections Framework

* **Interfaces**: The main interfaces in the Java Collections Framework include:

**`Collection<E>`**

The root interface of the collection hierarchy.

**`List<E>`**

A subinterface extending `Collection<E>`.

**`Set<E>`**

A subinterface extending `Collection<E>`.

**`Queue<E>`**

A subinterface extending `Collection<E>`.

**`Map<K, V>`**

The foundational interface for all map-based data structures in Java.

These interfaces define the basic operations and behaviours expected from different types of collections.

* **Implementations**: The following classes are the most commonly used implementations of these interfaces:

**`ArrayList<E>`**

Implements `List<E>`.

**`LinkedList<E>`**

Implements `List<E>`.

**`HashSet<E>`**

Implements `Set<E>`.

**`HashMap<K, V>`**

Implements `Map<K, V>`.

`**TreeSet<E>`**

Implements `NavigableSet<E>` and `SortedSet<E>`.

**`TreeMap<K, V>`**

Implements `NavigableMap<K, V>` and `SortedMap<K, V>`.

Most collection classes (including all of those above) also implement the `Cloneable` and `Serializable` interfaces.

* The `Cloneable` interface is used to indicate that a class allows its objects to be cloned, meaning to create a copy of the object.
* The `Serializable` interface is used to indicate that a class can be serialised, meaning its objects can be converted into a **byte stream** and later reconstructed (deserialised) from that byte stream.

As can be seen, most of the interfaces and classes in the Java Collections Framework are generics, which means they use type parameters to ensure type safety at compile time. This feature was introduced in Java 5 to allow developers to specify the type of elements that a collection can hold, reducing the need for type casting and preventing runtime `ClassCastException` errors.

### `ArrayList`

An `ArrayList` in Java is a part of the Java Collections Framework and is a resizable array implementation of the `List` interface. Unlike regular arrays in Java, which have a fixed (static) size, an `ArrayList` can dynamically grow and shrink in size when elements are added or removed.

* **Resizable Array**: The size of an `ArrayList` can grow or shrink dynamically. When elements are added or removed, the internal array is resized automatically.
* **Indexed Access**: Elements in an `ArrayList` can be accessed using an index, similar to arrays, providing fast access to elements.
* **Ordered Collection**: `ArrayList` maintains the order of insertion, meaning the order in which elements are added is the order in which they are stored.
* **Allows Duplicates and Nulls**: `ArrayList` allows duplicate elements and also permits `null` values.

Common methods of the `ArrayList` class include:

* **`add(E element)`**: Adds an element to the end of the list.
* **`add(int index, E element)`**: Inserts an element at the specified position in the list.
* **`remove(int index)`**: Removes an element at the specified position in the list.
* **`remove(Object o)`**: Removes the first occurrence of the specified element from the list.
* **`get(int index)`**: Returns the element at the specified index in the list (the square bracket notation, i.e., `array[0]`, cannot be used with ArrayLists, only with regular arrays.
* **`size()`**: Returns the number of elements in the list.
* **`contains(Object o)`**: Returns `true` if the list contains the specified element.
* **`clear()`**: Removes all elements from the list.

### `HashMap`

A `HashMap` in Java is a part of the Java Collections Framework and is used to store key-value pairs. It implements the `Map` interface and is based on a hash table (hash map).

*\*The terms "hash table", "hash map", simply "map", or "associative array" are general, language-independent terms that refer to the type of data structure used in computer science to store key-value pairs.*

* **Key-Value Storage**: `HashMap` allows you to store data in key-value pairs, where each key is unique. You can retrieve, update, or delete the value associated with a specific key.
* **Hashing Mechanism**: `HashMap` uses a technique called hashing to efficiently store and retrieve objects. The key's `hashCode()` method is used to compute an index in the underlying array (bucket) where the value is stored. If two keys have the same hash code, they are stored in the same bucket, and a linked list or binary tree is used to resolve collisions.
* **Null Keys and Values**: `HashMap` allows one null key and multiple null values. This means you can store up to one key-value pair where the key is `null`, and you can store multiple pairs where the value is `null`.
* **Order**: The `HashMap` does not guarantee any specific order of the elements. The order of the keys may change as elements are added or removed.
* **Synchronisation**: `HashMap` is not synchronised, which means it is not thread-safe. `ConcurrentHashMap` is a thread-safe alternative to `HashMap`.

Common methods of the `HashMap` class include:

* **`put(K key, V value)`**: Associates the specified value with the specified key in this map.
* **`get(Object key)`**: Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
* **`remove(Object key)`**: Removes the mapping for a key from this map if it is present.
* **`containsKey(Object key)`**: Returns `true` if this map contains a mapping for the specified key.
* **`size()`**: Returns the number of key-value mappings in this map.
* **`isEmpty()`**: Returns `true` if this map contains no key-value mappings.

# Additional Notes on Interfaces

## Variables in Interfaces

All variables declared in an interface are **`public`, `static`, and `final`**. This cannot be changed. Instance variables (fields that belong to an object of a class) are specific to classes and cannot be declared in an interface. This is because an interface is meant to define a contract for behaviour, not to maintain state.

## Methods in Interfaces

Most methods in interfaces are **abstract**.

A **default method** is a method that has a body (implementation) and is declared in an interface using the `default` keyword. Default interfaces were introduced in Java 8. Before this, adding a new method to an interface would break all classes that implement the interface because they would be required to implement the new method. Default methods solve this problem by providing a default implementation that existing classes can use, so they don't need to be modified.

Implementing classes can optionally choose to override the default method if they want to provide a different implementation.

*To sum up the above, in interfaces variables must be* ***`public`, `static`, and `final`,*** *and methods must be* ***`abstract` or `default`*** *(abstract methods to not need to be explicitly specified in such in interfaces as this is assumed if there is no implementation).*