The Collection Interface

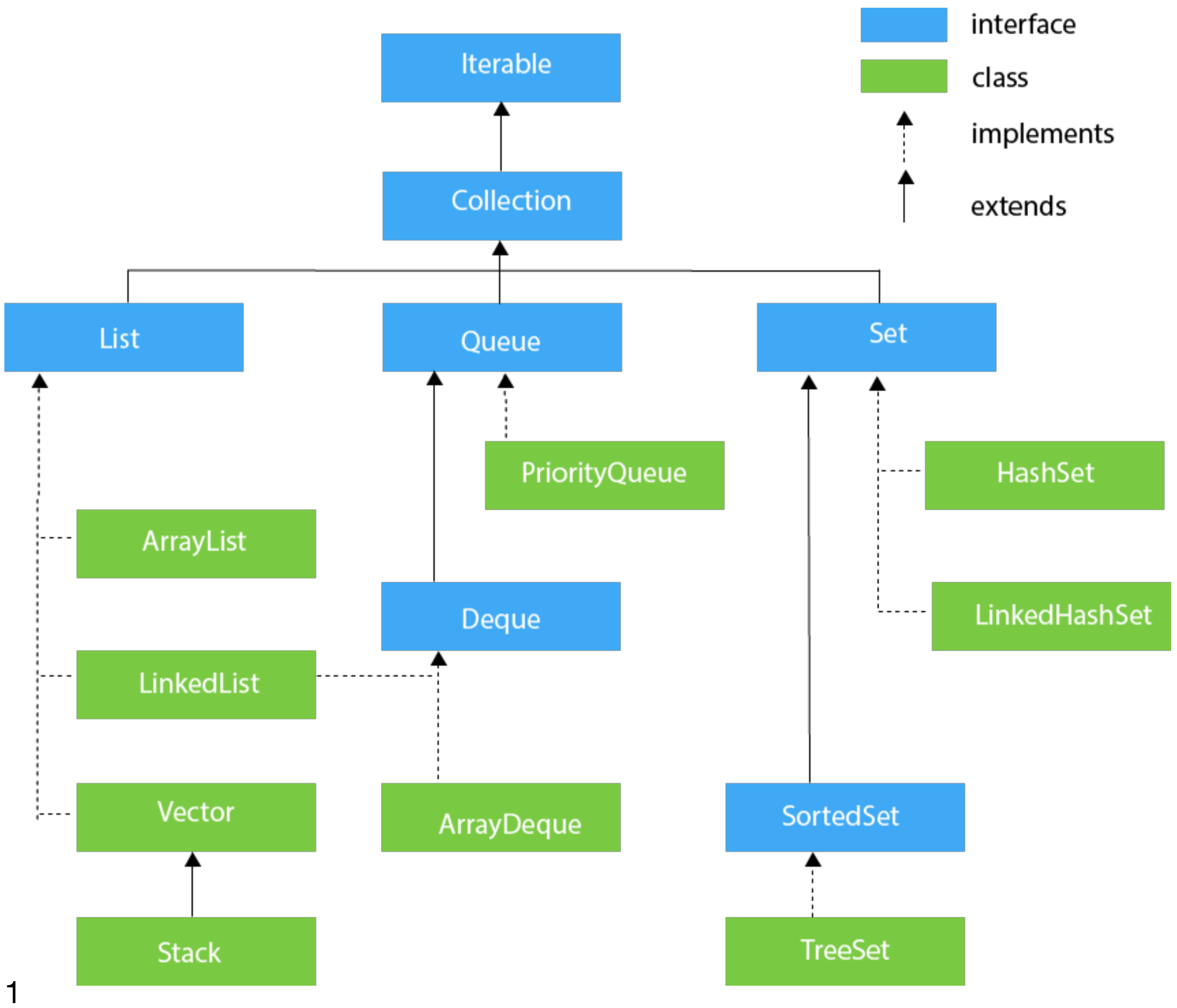
- **Collection** is the root interface of the Java Collections Framework (JCF).
- It provides a set of methods for managing groups of objects known as collections.
- Common methods defined in the **Collection** interface:
  - **add**(E element): Adds an element to the collection.
  - **remove**(Object element): Removes a specific element from the collection.
  - **size**(): Returns the number of elements in the collection.
  - **clear**(): Removes all elements from the collection.
  - **contains**(Object element): Checks if a collection contains a specific element.

Other interfaces in JCF (such as **List**, **Set**, **Queue**) extend the **Collection** interface, providing specialised functionalities.

Collection Architecture in Java

The Collection Framework provides a unified architecture for representing and manipulating collections in Java. The key components of this architecture include:

1. **Interfaces**: These define the abstract data types that collections must follow.
   - **Examples**: **Collection**, **List**, **Set**, **Queue**, **Map** (though **Map** is not part of the **Collection** hierarchy).



2. **Implementations** (**Classes**): These are concrete implementations of the collection interfaces.
- **Examples**: **ArrayList**, **LinkedList**, **HashSet**, **TreeSet**, **HashMap**.

3. **Algorithms**: Methods that perform useful computations (like sorting or searching) on collections.
- Defined as static methods in the **Collections** utility class.
- **Examples**: sort(List<T> list), binarySearch(List<T> list, T key).

Key Collection Interfaces:

- **List**: Ordered collection (sequence), allows duplicates. Implements **Collection**.
- **Set**: Unordered collection that doesn't allow duplicates. Implements **Collection**.
- **Queue**: Typically ordered in a FIFO (First-In-First-Out) manner. Implements **Collection**.
- **Map**: A collection of key-value pairs. Does not extend **Collection**.

Collection Classes

Java provides several concrete classes that implement the Collection interfaces. Some of the most common are:

1. ArrayList (Implements List):
- A resizable array implementation of the List interface.
- Allows random access to elements (constant time complexity for get).
- **Example**:

```
List<String> list = new ArrayList<>();
 list.add("Java");
 list.add("Python");
```

2. LinkedList (Implements List, Deque):
- A doubly-linked list implementation.
- Ideal for insertions and deletions from the list (constant time for adding/removing at the start/end).
- **Example**:

```
LinkedList<Integer> linkedList = new LinkedList<>();
linkedList.add(10);
linkedList.addFirst(5);
```

3. HashSet (Implements Set):
- Backed by a **HashMap**, this implementation provides constant-time performance for basic operations (add, remove, contains).
- Does not guarantee any specific order of elements.

- **Example**:
```
  Set<String> set = new HashSet<>();
  set.add("Apple");
  set.add("Banana");
```

4. TreeSet (Implements NavigableSet):
- A **Set** implementation that maintains the elements in a sorted (ascending) order.
- Uses a **Red-Black Tree** under the hood.
- **Example:**
```
 TreeSet<Integer> treeSet = new TreeSet<>();
  treeSet.add(50);
  treeSet.add(10);
```

5. HashMap (Implements Map):
- A hash table-based implementation of the **Map** interface.
- Provides constant-time performance for **put** and **get**.
- Allows one null key and multiple null values.
- **Example**:
```
 Map<String, Integer> map = new HashMap<>();
  map.put("Apple", 3);
  map.put("Banana", 5);
```

Traversing Collections

Java provides various ways to traverse or iterate over collections. The most common techniques include:

1. **Iterator**:
- An interface that allows sequential access to elements in a collection.
- Provides methods like **hasNext**(), **next**(), and **remove**().
- **Example**:
```
 Iterator<String> iterator = list.iterator();
  while (iterator.hasNext()) {
     System.out.println(iterator.next());
  }
```

2. Enhanced for-loop:
- A shorthand loop to iterate over collections.
- **Example**:
```
  for (String element : list) {
     System.out.println(element);
  }
```

3. forEach() method (Java 8 and later):
- A default method in the **Iterable** interface, used for functional-style traversal.
- **Example**: `list.forEach(System.out::println);`

4. Stream API (Java 8 and later):
- Provides functional programming capabilities to work with collections.
- **Example**:

```
list.stream().filter(s -> s.startsWith("J")).forEach(System.out::println);
```

Working with Maps

A **Map** represents a collection of key-value pairs and is not part of the **Collection** interface hierarchy. Important methods in Map include:

- **put**(K key, V value): Adds a key-value pair to the map.
- **get**(Object key): Retrieves the value associated with a given key.
- **remove**(Object key): Removes the key-value pair for the specified key.
- **containsKey**(Object key): Checks if a specific key exists in the map.

Key Map Implementations:

1. **HashMap**:
   - Provides constant-time performance for basic operations.
   - Allows one null key and multiple null values.

2. TreeMap:
- A **Map** that maintains its entries in sorted order (based on the natural ordering of keys or a custom comparator).

3. LinkedHashMap:
- Maintains the insertion order or the access order of elements.

Example of traversing a Map:

```
Map<String, Integer> map = new HashMap<>();
map.put("Apple", 10);
map.put("Banana", 5);

for (Map.Entry<String, Integer> entry : map.entrySet()) {
   System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

Working with Sets

A **Set** is a collection that does not allow duplicate elements. Common operations on sets include:

- **add**(E element): Adds an element to the set.
- **remove**(Object element): Removes the specified element from the set.
- **contains**(Object element): Checks if the set contains the element.

Key Set Implementations:

1. HashSet:
- Backed by a hash table, provides constant-time performance for add, remove, and contains.

2. TreeSet:
- Maintains elements in sorted order and provides logarithmic time complexity for most operations.

3. LinkedHashSet:
- Maintains insertion order and is backed by a hash table and a linked list.

Example of working with Set:

```
Set<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");
set.add("Apple"); // This will not be added again

for (String fruit : set) {
   System.out.println(fruit);
}
```

Conclusion

- **Collection Framework** in Java provides a comprehensive architecture for storing and manipulating groups of objects.
- Interfaces like **List**, **Set**, and **Map** define the core behaviour, while classes like **ArrayList**, **HashSet**, and **HashMap** provide concrete implementations.
- Efficient traversal mechanisms like **Iterator** and the **forEach** method simplify working with collections.
- Maps and sets are powerful tools for managing data structures without duplicates and with quick lookup performance.

Links:
https://www.javatpoint.com/collections-in-java
https://docs.oracle.com/javase/8/docs/api/?java/util/Collection.html
https://www.geeksforgeeks.org/collections-in-java-2/