Java Collection Framework

Iterable Interface in Java

The `Iterable` interface in Java is the root interface in the **Collection Framework** hierarchy. It provides the foundation for all collection classes by allowing objects of a class to be iterated over using the enhanced `for` loop.

Key Features of the `Iterable` Interface
1. **Root Interface for Iteration**: Parent interface of all collection types (`Collection`, `List`, `Set`, etc.).
2. **Iterator Support**: Defines the mechanism to retrieve an `Iterator` for traversing elements.
3. **Default Methods** (Java 8+): Includes `forEach()` for functional programming paradigms.

Key Methods of the `Iterable` Interface
- `Iterator<T> iterator()`: Returns an `Iterator` object for traversing the elements.
- `void forEach(Consumer<? super T> action)`: Performs an action for each element.
- `Spliterator<T> spliterator()`: Returns a `Spliterator` for parallel processing.

Example: Using `Iterable`
```java
import java.util.ArrayList;

public class IterableExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");

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

ArrayList in Java

The `ArrayList` class in Java is part of the **Collection Framework** and provides a resizable array implementation.

Key Features of `ArrayList`
1. **Dynamic Sizing**: Automatically resizes itself as elements are added or removed.
2. **Indexed Access**: Provides fast random access using indices.
3. **Ordered**: Maintains the insertion order of elements.
4. **Allows Duplicates**: Can store duplicate elements.

Common Methods
- **Adding Elements**: `list.add("Apple");`, `list.add(1, "Banana");`
- **Accessing Elements**: `String fruit = list.get(0);`
- **Updating Elements**: `list.set(1, "Cherry");`
- **Removing Elements**: `list.remove(0);`

Example: Basic Usage
```java
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");

        for (String fruit : fruits) {
            System.out.println("Fruit: " + fruit);
        }
    }
}
```

Advantages of `ArrayList`
- **Dynamic Resizing**: Automatically adjusts size as needed.
- **Indexed Access**: Efficient `O(1)` access time for elements.
- **Easy to Use**: Provides a wide range of methods for manipulation.

ArrayList vs. LinkedList

Feature	ArrayList	LinkedList
Underlying Structure	Dynamic array	Doubly linked list
Access Time	Fast (`O(1)`)	Slow (`O(n)`)
Insertion/Deletion	Slow for large shifts (`O(n)`)	Fast (`O(1)`)
Memory	Less overhead	Higher memory usage

Example: Sorting an `ArrayList`

```java
import java.util.ArrayList;
import java.util.Collections;

public class ArrayListSorting {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(5);
        numbers.add(1);
        numbers.add(3);

        Collections.sort(numbers);
        System.out.println("Sorted List: " + numbers);

        Collections.sort(numbers, Collections.reverseOrder());
        System.out.println("Reversed List: " + numbers);
    }
}
```

When to Use `ArrayList`

- When you need **fast random access** to elements.
- When the majority of operations involve **adding to the end** or **accessing by index**.
- For **non-thread-safe** environments where synchronization is not a concern.