Interface

"An interface is a collection of method definitions (without implementations) and constant values".

Its purpose is to captures similarities between unrelated classes without forcing a class relationship, and to do this, an interface operates like a contract in the form of a collection of methods and constant declarations. When a class implements an interface, it promises to implement all of the methods declared in that interface.

A class can inherit multiple interfaces at once, and information on what interfaces are inherited by what class can be accessed by the framework and other objects. Hence, an object can identify an object that inherited a specified interface.

This information can be used by a particular framework or another object to determine whether the object is compatible with different objects or statements.

`Iterable` and `Iterator` are the best interface examples there are for you to understand the role of interface. Classes implementing these two interfaces are classified as an enumerable collection. This means that class behaves like any other collection object written for Java, such as an `ArrayList`.

There are many features that define an enumerable collection, and by implementing these two interfaces, the framework guarantees that this object has these features. For example, an iterable collection must have a method called `next`, which moves the cursor within the collection one step forward. By implementing the `Iterable` interface, the class is promising (and is required) to implement this method.

As mentioned above, when a class implements these interfaces and is defined as an iterable collection, it behaves like one as well. This means that it is capable of being iterated using an enhanced `for` statement, just like we can do with an `ArrayList` object.

Iterable

The `Iterable` interface is used by classes that represent a collection of objects such as a list, giving us a way to move through the collection one object at a time. The `Iterable` interface is not used to represent the list itself; it merely represents a way to move through the elements of the list.

Exposes the iterator, which supports a simple iteration over a collection.

```
   public Iterator<T> iterator();
```

Iterator

The `Iterator` interface supports a simple iteration over a collection. Iterators only allow reading the data in the collection and removing the last item iterated.

Method Summary	
`boolean`	**`hasNext()`** Returns *true* if the iteration has more elements. (In other words, returns true if next() would return an element rather than throwing an exception.)
`T`	**`next()`** Removes and returns the next element in the iteration. **Throws:** `NoSuchElementException` - if the iteration has no more elements
`default void`	**`remove()`** Removes from the underlying collection the last element returned by this iterator (optional operation). This method can be called only once per call to next(). The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method. **Implementation Requirements:** The default implementation throws an instance of `UnsupportedOperationException` and performs no other action. **Throws:** `UnsupportedOperationException` - if the remove operation is not supported by this iterator `IllegalStateException` - if the next method has not yet been called, or the remove method has already been called after the last call to the next method

Enhanced For Statement

The `for each` statement repeats a group of embedded statements for each element in an array or an object collection that implements the `Iterable` interface. The `for each` statement is used to iterate through the collection to get the information that you want, but can not be used to add or remove items from the source collection to avoid unpredictable side effects. If you need to add or remove items from the source collection, use a for loop.

```
int[] fib = new int[] { 0, 1, 2, 3, 5, 8, 13 };
for (int i : fib)
{
   System.out.println(i);
}
```

An `ArrayList` is an array whose size is dynamically increased as required. It is implemented as a class and provides methods to access and/or modify the data stored in the array. Create a class called `MyArrayList` class that implements the `Iterable` interface as described in the following summaries.

Field Summary	
`T[]`	**`list`** The array containing elements of type T.
`int`	**`count`** The number of elements in this list.

Constructor Summary
`MyArrayList``()` Constructs an empty list with a default capacity of 10 elements.
`MyArrayList``(int initSize)` Constructs an empty list with the specified initial capacity.

Method Summary	
`void`	**`add`**`(T value)` Adds an object to the end of the list. If count already equals capacity, the capacity of the ArrayList is increased by automatically reallocating the internal array, and the existing elements are copied to the new array before the new element is added.
`void`	**`add(int index, T element)`** Inserts the specified element at the specified position in this list. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices). Throws an `IndexOutOfBoundsException` - if the index is out of range (index < 0 || index > size())
`int`	**`capacity()`** Gets the number of elements that the list can contain.
`void`	**`clear`**`()` Removes all of the elements from this list.
`void`	**`get(int index)`** Returns the value of this list at position index. An `IndexOutOfBoundsException` is thrown if index is less than zero or greater than or equal to count.

`int`	**`indexOf`**`(T element)` Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`Boolean`	**`isEmpty()`** Returns true if this list contains no elements.
`Iterator<t>`	**`iterator()`** Returns an iterator over the elements in this list in proper sequence.
`boolean`	**`remove`**`(T element)` Removes the first occurrence of the specified element from this list, if it is present. Returns *true* if the element was found and removed, *false* otherwise.
`T`	**`remove`**`(int index)` Removes and returns the element at the specified position in this list. An **`IndexOutOfBoundsException`** is thrown if index is less than zero or greater than or equal to count.
`T`	**`set(int index, T element)`** Replaces the element at the specified position in this list with the specified element. Throws an `IndexOutOfBoundsException` - if the index is out of range (index < 0 || index >= size())
`int`	**`size()`** Gets the number of elements actually contained in the list.
`Object[]`	**`toArray`**`()` Returns an array containing all of the elements in this list in proper sequence (from first to last element).
`String`	**`toString()`** Returns a string representation of this linked list. The string representation consists of a list of the linked list's elements in the order they are returned by iterating the list, enclosed in square brackets (`"[]"`). Adjacent elements are separated by the characters `", "` (comma and space).

Execute *MyArrayListTester.java* to test your `MyArrayList` class.

Output:

```
The capacity of list is now 5
Adding "Banana" to the list.
Adding "Apple" to the list.
Adding "Orange" to the list.
Adding "Orange" to the list.
Adding "Lemon" to the list.
Adding "Orange" to the list.
Adding "Peach" to the list.

The capacity of list is now 15
The size of the list is now 7

Adding "Grapefruit" at position 2
Replacing the element at position 5 with Pear

Testing the list iterator:
Banana Apple Grapefruit Orange Orange Lemon Pear Peach

Testing the for each statement:
Banana Apple Grapefruit Lemon Pear Peach

Removing Lemon from the list:

Testing the toString() method:
[Banana, Apple, Grapefruit, Pear, Peach]

Testing the toArray() method:
Banana Apple Grapefruit Pear Peach

The list is empty: false
Clearing the list.
The list is empty: true
```