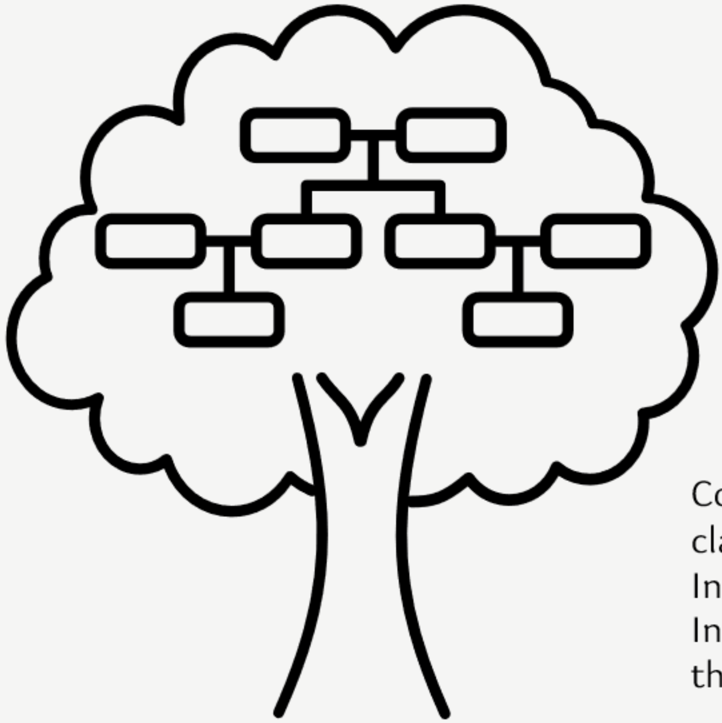
The Collection Interface

Collection is the root interface in the collection hierarchy. A collection represents a group of objects, known as its elements. Some collections allow duplicate elements and others do not. Some are ordered, and others are unordered. Collections that have a defined encounter order are generally subtypes of the SequencedCollection interface. The JDK does not provide any direct implementations of this interface: it provides implementations of more specific subinterfaces like Set and List. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.



The Collection<E> interface outlines the fundamental functionality expected from any collection (excluding maps). It organizes its methods into four categories: adding elements, removing elements, querying the contents of the collection, and making its elements available for additional processing. Below are the important methods from Collection interface,

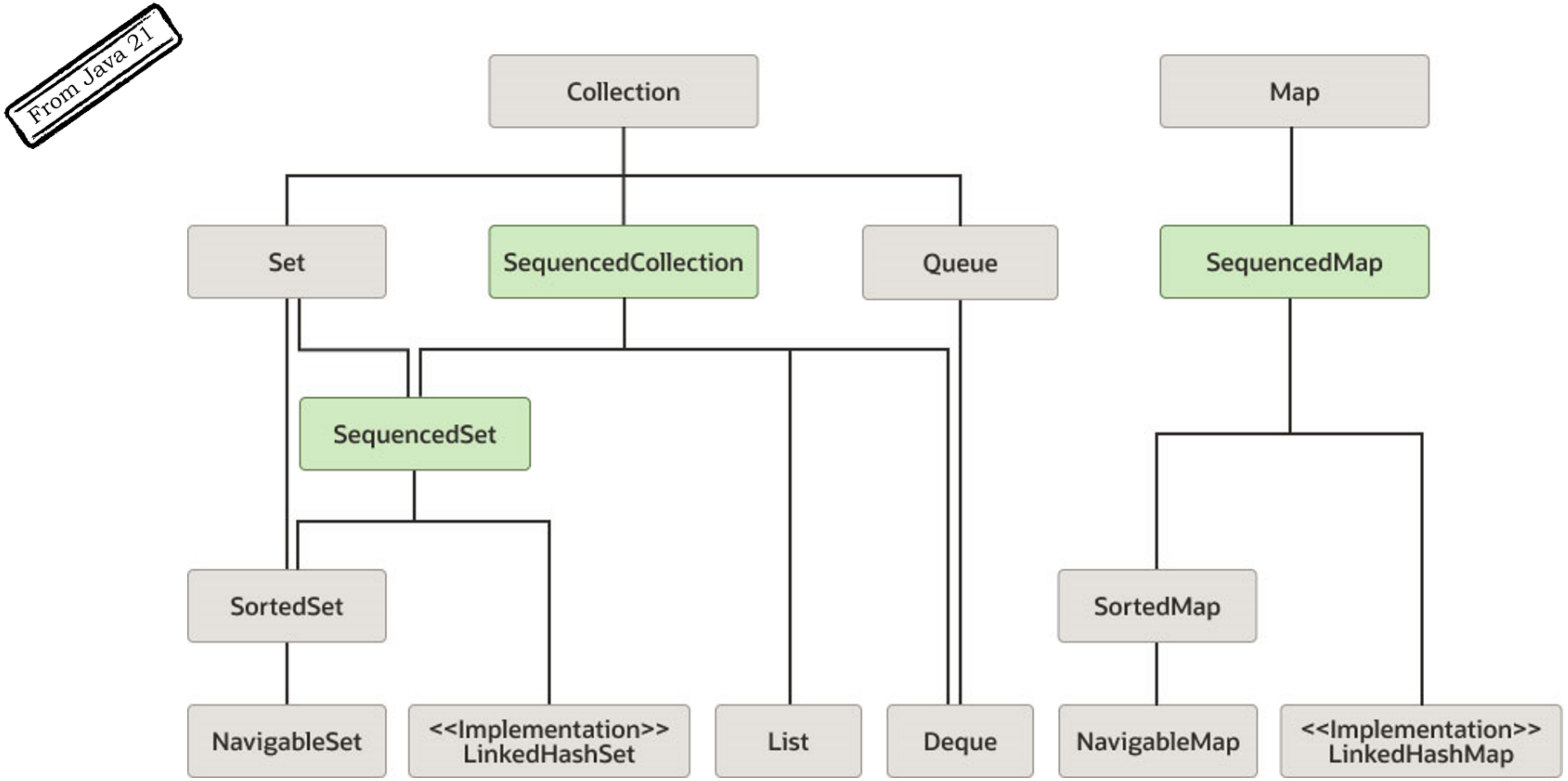
- `boolean add(E e)`
- `void clear()`
- `boolean remove(Object o)`
- `boolean contains(Object o)`
- `boolean isEmpty()`
- `Iterator<E> iterator()`
- `Spliterator<E> spliterator`
- `Stream<E> stream()`
- `Stream<E> parallelStream()`

Collection interface has a size() abstract method which needs to be implemented in all the Collections classes. It should return the number of elements in the collection object. If this collection contains more than Integer.MAX_VALUE elements, returns Integer.MAX_VALUE. The decision to make size return Integer.MAX_VALUE for extremely large collections was probably taken on the assumption that collections this large—with more than two billion elements—will rarely occur.

Collection interface also has three different overloads of toArray() method, all return an array containing the elements of this collection.

The SequencedCollection, SequencedSet, SequencedMap Interfaces

eazy bytes



The SequencedCollection, SequencedSet, SequencedMap Interfaces

eazy bytes

From Java 21

Why do we need Sequenced Collections?

Sequenced Collections aim to address some issues with the Java Collections Framework. One problem is that there's no common type for collections with a specific order, making it tricky to express certain ideas in code. For example, if we want to create a method that takes a collection and returns its first element, we can't use Collection or List because they are either too general or too specific.

Another problem is that accessing the first and last elements of a collection, or iterating in reverse order, isn't uniform across different collections. Each collection has its own way of doing things, and some operations are not straightforward. For example, to get the last element of an ArrayList, we have to use list.get(list.size() - 1). To get the last element of a LinkedHashSet, we have to iterate over the entire set. To iterate over a collection in reverse order, we have to use different methods depending on the type of collection.

Sequenced Collections tackle these problems by introducing new interfaces that define these operations consistently for all collections with a specific order.

Sequenced Collections introduce three new interfaces in the collection hierarchy: SequencedCollection, SequencedSet, and SequencedMap. These interfaces extend their respective superinterfaces (Collection, Set, and Map) and provide additional methods for accessing and manipulating their elements based on their encounter order.

The SequencedCollection, SequencedSet, SequencedMap Interfaces

eazy bytes

From Java 21

The SequencedCollection interface provides methods to add, retrieve, and remove elements at either end of the collection, along with the reversed() method, which provides a reverse-ordered view of the collection.

```java
public interface SequencedCollection<E> extends Collection<E> {
    default void addFirst(E e) { ... }
    default void addLast(E e) { ... }
    default E getFirst() { ... }
    default E getLast() { ... }
    default E removeFirst() { ... }
    default E removeLast() { ... }
    SequencedCollection<E> reversed();
}
```

As we can see, all methods except reversed() are default methods and provide a default implementation. This means that existing collection classes such as ArrayList and LinkedList can implement this interface without changing their code.

The four interfaces that inherit from SequencedCollection—List, SequencedSet, NavigableSet, and Deque—all define covariant overrides for reversed. So, for example, List's declaration of reversed is

List<E> reversed()

The SequencedCollection, SequencedSet, SequencedMap Interfaces

eazy bytes

From Java 21

The SequencedSet interface is specific to Set implementations such as LinkedHashSet. SequencedSet extends SequencedCollection and overrides its reversed() method. The only difference is that the return type of SequencedSet.reversed() is SequencedSet.

```java
public interface SequencedSet<E> extends SequencedCollection<E>, Set<E> {

    SequencedSet<E> reversed();

}
```

The SequencedMap interface is specific to Map implementations such as LinkedHashMap. SequencedMap extends Map and provides methods for accessing and manipulating its entries based on their encounter order.

```java
public interface SequencedMap<K, V> extends Map<K, V> {

    default Map.Entry<K,V> firstEntry() {...}
    default Map.Entry<K,V> lastEntry() {...}
    default Map.Entry<K,V> pollFirstEntry() {...}
    default Map.Entry<K,V> pollLastEntry() {...}
    default V putFirst(K k, V v) {...}
    default V putLast(K k, V v) {...}
    SequencedMap<K, V> reversed();

}
```

Overall, Sequenced Collections are a great addition to the Java Collections Framework. They provide a uniform way to access and manipulate collections that have an encounter order. They also make it easier to express certain useful concepts in APIs. They are compatible with existing collection classes and do not require any code changes to implement them.

List

A List is an ordered Collection where each element can be accessed by its index. List maintains the order of insertion and allow duplicate elements.

0	1	2
element1	element2	element3

Continues.....

The List interface defines several methods for working with lists, including:

add(E element): Adds an element to the end of the list.
add(int index, E element): Inserts an element at the specified index.
remove(int index): Removes the element at the specified index.
get(int index): Returns the element at the specified index.
set(int index, E element): Replaces the element at the specified index with the given element.
size(): Returns the number of elements in the list.
clear(): Removes all the elements from the list.

There are several classes that implement the List interface in Java. Most commonly used classes are **ArrayList** and **LinkedList**.

List

Below are the list methods Inherited from SequencedCollection and their equivalent positional methods that are already present,

SequencedCollection method	List positional equivalent method
addFirst(el)	add(0, el)
addLast(el)	add(el)
getFirst()	get(0)
getLast()	get(size() - 1
removeFirst()	remove(0)
removeLast()	remove(size() - 1)

Apart from the above methods, List also overrides reversed() method from SequencedCollection which provides a reverse-ordered view of the list

ArrayList

The ArrayList class in Java is an implementation of the List interface that employs a dynamic array to store elements. Unlike traditional arrays, it does not have a fixed size, allowing for the addition and removal of elements at any time, making it highly flexible.

The ArrayList class can be found in the java.util package and is analogous to the Vector class in C++. It maintains the insertion order of its elements and can store duplicates. All of the methods available in the List interface can be used with the ArrayList class.

Few important points about ArrayList:

- ArrayList class is unsynchronized, meaning it is not thread-safe.
- Supports random access to elements quickly because they works on an index basis
- However, when it comes to manipulating elements, removing an element from the ArrayList can be slower than using the LinkedList class in Java because it requires shifting of all elements to fill the gap left by the removed element.
- To create an ArrayList of primitive types like int, float, and char in Java, we cannot use the primitive types directly. Instead, we need to use the corresponding wrapper class. For instance:

```
ArrayList<int> intNums = ArrayList<int>(); // Compilation fails
ArrayList<Integer> intNums = new ArrayList<Integer>(); // works fine
```

ArrayList

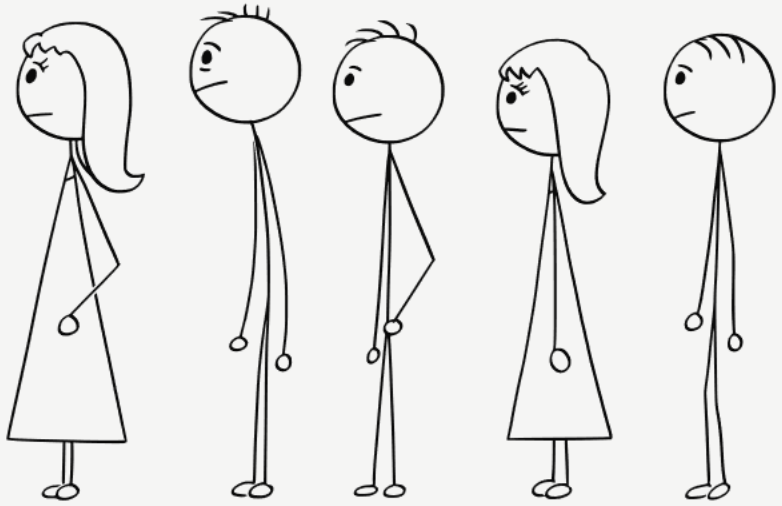
Below are the examples of Creating ArrayList. Any time we want to create a collection object, we need to use **angle brackets** (<>) and inside them, we need to mention the Object or Element Data type.

```
List<Integer> integerNums = new ArrayList<Integer>();
List<String> countryNames = new ArrayList<String>();
List<Character> characters = new ArrayList<Character>();
List<Person> persons = new ArrayList<Person>();
var doubleNums = new ArrayList<Double>();
```

Let's see how array list works by using the below code,

```
List<String> countryNames = new ArrayList<String>();
countryNames.add("India");
countryNames.add("Canada");
countryNames.add("USA");
countryNames.add("Germany");
countryNames.add("India");
```

0	1	2	3	4	5	6	7	8	9
India	Canada	USA	Germany	India	null	null	null	null	null



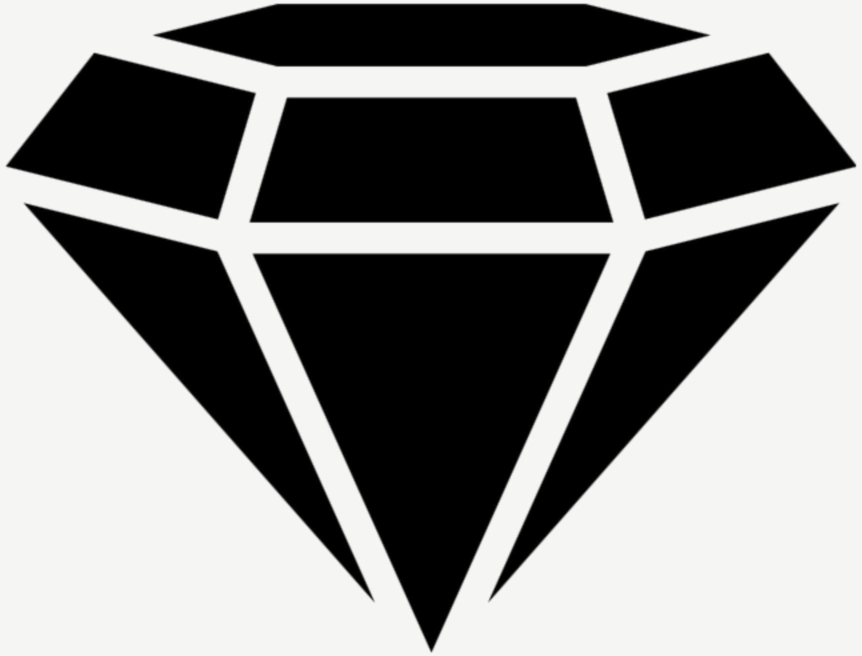
Maintains Order & allow duplicates

When an ArrayList object is created, it starts with an initial capacity of zero. As more elements are added, the ArrayList's capacity will continue to grow dynamically based on the algorithms and formulas specified in the Java library.

From Java 7

Diamond Operator in Java

eazy bytes



In Java, the Diamond Operator <> is used with Generics to improve code readability and reduce the amount of code we need to write. It was introduced in Java 7 and is used to infer the type arguments of a generic class.

Before the Diamond Operator, when creating an instance of a generic class, we had to specify the type arguments twice, once when declaring the variable and again when creating the object. For example:

```
List<String> myList = new ArrayList<String>();
```

With the Diamond Operator, we can omit the second set of type arguments, and the compiler will automatically infer them from the declaration of the variable. The same example with the Diamond Operator would look like:

```
List<String> myList = new ArrayList<>();
```

This reduces the amount of boilerplate code we need to write and makes the code more concise and readable. However, it's important to note that the Diamond Operator can only be used when the type arguments can be inferred from the declaration of the variable, and not when they are used as method parameters or when the type is ambiguous.

ArrayList performance

To understand about the performance of the ArrayList, let's consider below example,

```
// Creating an ArrayList of integers
ArrayList<Integer> numbers = new ArrayList<>();

// Adding elements to the ArrayList
numbers.add(1);
numbers.add(2);
numbers.add(3);
numbers.add(4);
```

0	1	2	3
1	2	3	4

Random-Access Operations:

ArrayList in Java is backed by an array, which means that it offers constant-time performance (O(1)) for random-access operations like get(index) and set(index, element). Random-access implies that you can directly access any element in the list using its index. This is possible as the elements are stored in a contiguous locations.

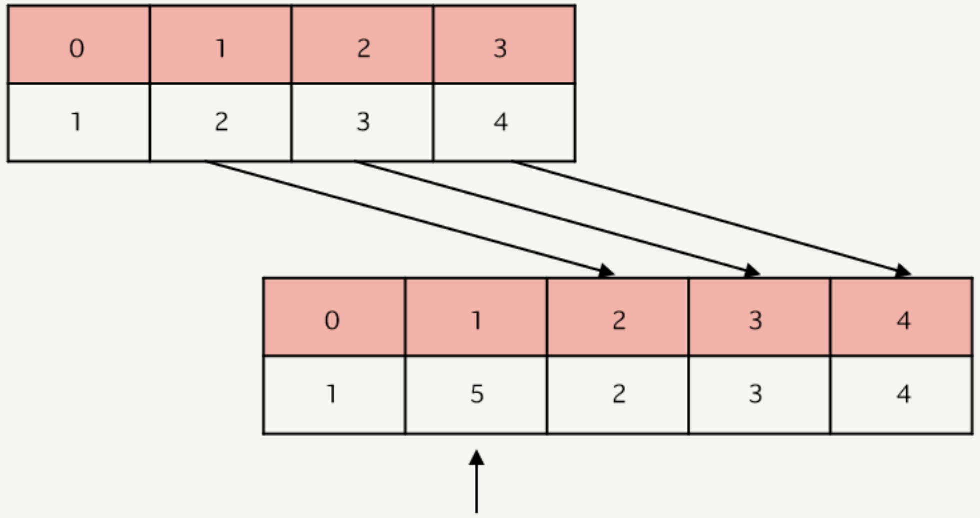
```
int elementAtIndex2 = numbers.get(2); // Random-access operation (O(1)): Get the element at index 2

numbers.set(1, 10); // Random-access operation (O(1)): Set the element at index 1
```

ArrayList performance

Insertion and Removal Performance:

The downside of using an array implementation is in inserting or removing elements at arbitrary positions in the list. When you insert or remove an element at a specific index, it may require adjusting the position of other elements in the array, leading to potential performance overhead.

```
numbers.add(1, 5); // Insertion at arbitrary position (may require shifting elements): Insert 5 at index 1
```



To add a new element at index 1, all the previous elements from index needs to be shifted to right and atlast a new element will be inserted at index 1

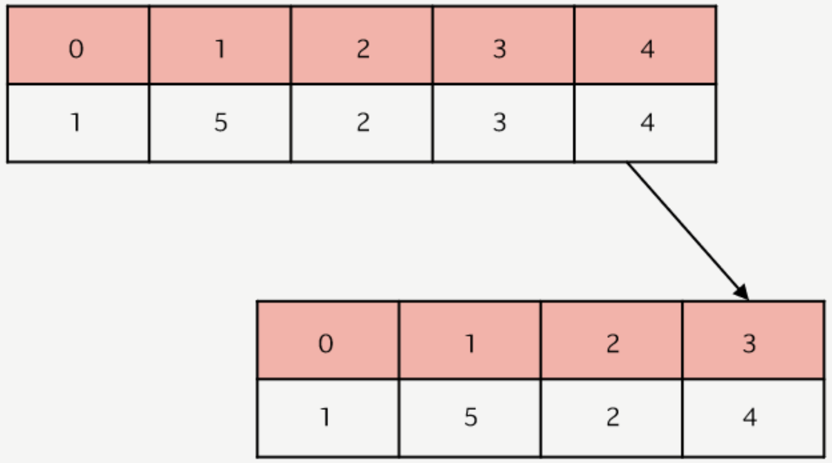
ArrayList performance

eazy bytes

Insertion and Removal Performance:

The downside of using an array implementation is in inserting or removing elements at arbitrary positions in the list. When you insert or remove an element at a specific index, it may require adjusting the position of other elements in the array, leading to potential performance overhead.

```
numbers.remove(3);  // Removal at arbitrary position require shifting elements
```



From Java 9

Creating Immutable Lists

eazy bytes

Immutable List Static Factory Methods

The List.of static factory methods provide a convenient way to create immutable lists. A list is an ordered collection, where duplicate elements are typically allowed. Null values are not allowed. The syntax of these methods is:

```
List.of()
List.of(e1)
List.of(e1, e2) // fixed-argument form overloads up to 10 elements
List.of(elements...)// varargs form supports an arbitrary number of elements or an array
```

In JDK 8:

```
List<String> stringList = Arrays.asList("a", "b", "c");
stringList = Collections.unmodifiableList(stringList);
```

In JDK 9:

```
List stringList = List.of("a", "b", "c");
```

Iterate ArrayList using for-each

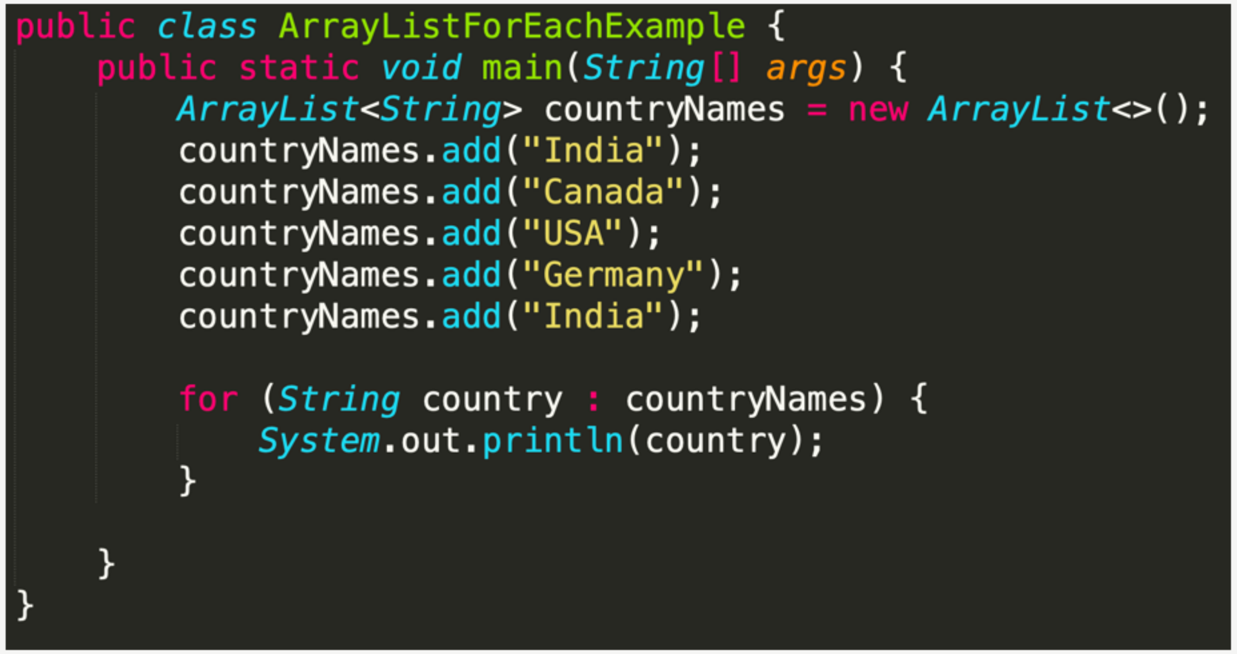
To iterate over an ArrayList using a for-each loop in Java, you can follow these steps:

1) Create an ArrayList of the desired type and add elements to it.

```
ArrayList<String> countryNames = new ArrayList<>();
// add some elements to the list
```

2) Use a for-each loop to iterate over the elements in the list.

```
for (String country : countryNames) {
    // code to execute for each element
}
```



```
public class ArrayListForEachExample {
    public static void main(String[] args) {
        ArrayList<String> countryNames = new ArrayList<>();
        countryNames.add("India");
        countryNames.add("Canada");
        countryNames.add("USA");
        countryNames.add("Germany");
        countryNames.add("India");

        for (String country : countryNames) {
            System.out.println(country);
        }

    }
}
```

Here's an example of iterating over an ArrayList of Strings using an Iterator.

Note that when using a for-each loop to iterate over an ArrayList, you cannot modify the contents of the list during the iteration. If you need to modify the list, you should use an Iterator instead.

Iterate ArrayList using iterator()

To iterate over an ArrayList using an Iterator in Java, you can follow these steps:

1) Create an instance of the Iterator class by calling the ArrayList's iterator() method.

```
ArrayList<String> countryNames = new ArrayList<>();
// add some elements to the list
Iterator<String> iterator = countryNames.iterator();
```

2) Use the hasNext() method of the Iterator to check if there are more elements in the list.

```
while(iterator.hasNext()){
    // code to process for each element
}
```

3) Use the next() method of the Iterator to get the next element in the list.

```
while(iterator.hasNext()){
    String country = iterator.next();
    // code to process for each element
}
```

Note that the Iterator can only be used to traverse the ArrayList in the forward direction. It supports only read and remove operations on the elements in the collection, and not add or modify operations.

If you need to remove elements from the ArrayList during the iteration, you should use the Iterator's remove() method instead of the ArrayList's remove() method, to avoid ConcurrentModificationException.

Iterate ArrayList using listIterator()

To iterate over an ArrayList using an ListIterator in Java, you can follow these steps:

1) Create an instance of the ListIterator class by calling the ArrayList's listIterator() method

```
ArrayList<String> countryNames = new ArrayList<>();
// add some elements to the list
ListIterator<String> iterator = countryNames.listIterator();
```

2) Use the hasNext() & next() methods of the ListIterator to check if there are more elements in the forward direction of the list & get the element.

```
while (iterator.hasNext()) {
    String country = iterator.next();
    // code to process for each element
}
```

3) Use the hasPrevious () & previous () methods of the ListIterator to check if there are more elements in the reverse direction of the list & get the element.

```
while(iterator.hasPrevious()){
    String country = iterator.previous();
    // code to process for each element
}
```

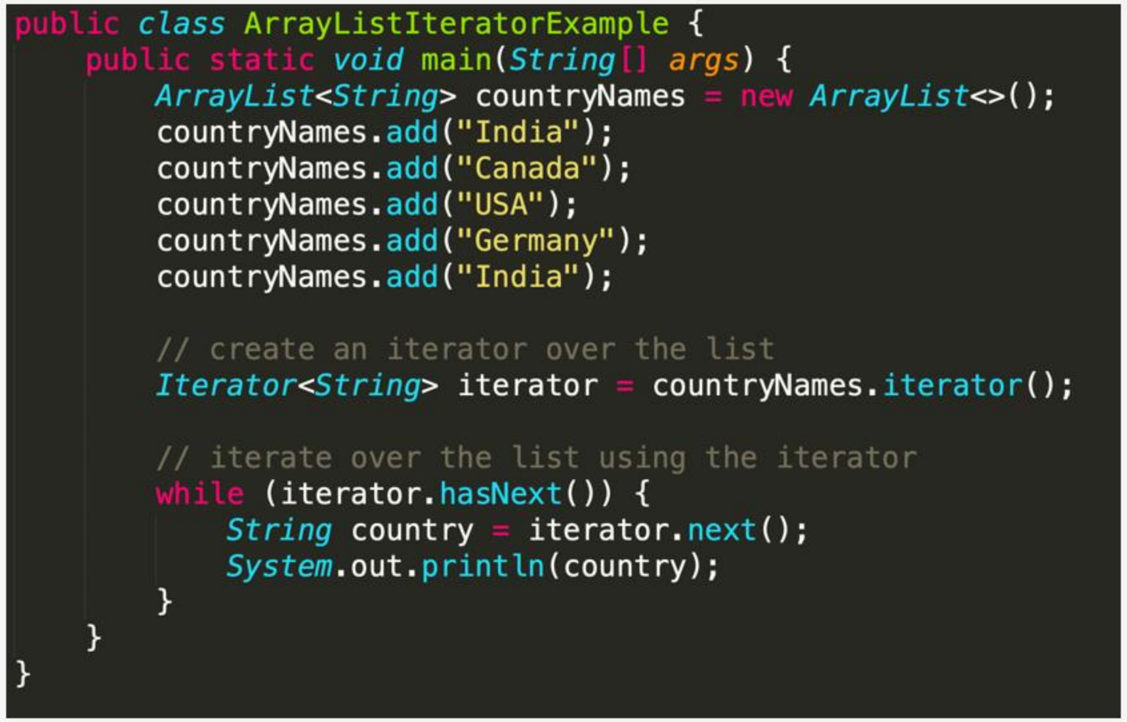
The ListIterator interface is a subinterface of Iterator and is used to iterate over a collection in both forward and backward directions. It supports read, remove, add, and modify operations on the elements in the collection and is slower as it uses more memory compared to Iterator.

Iterator vs ListIterator

eazy bytes

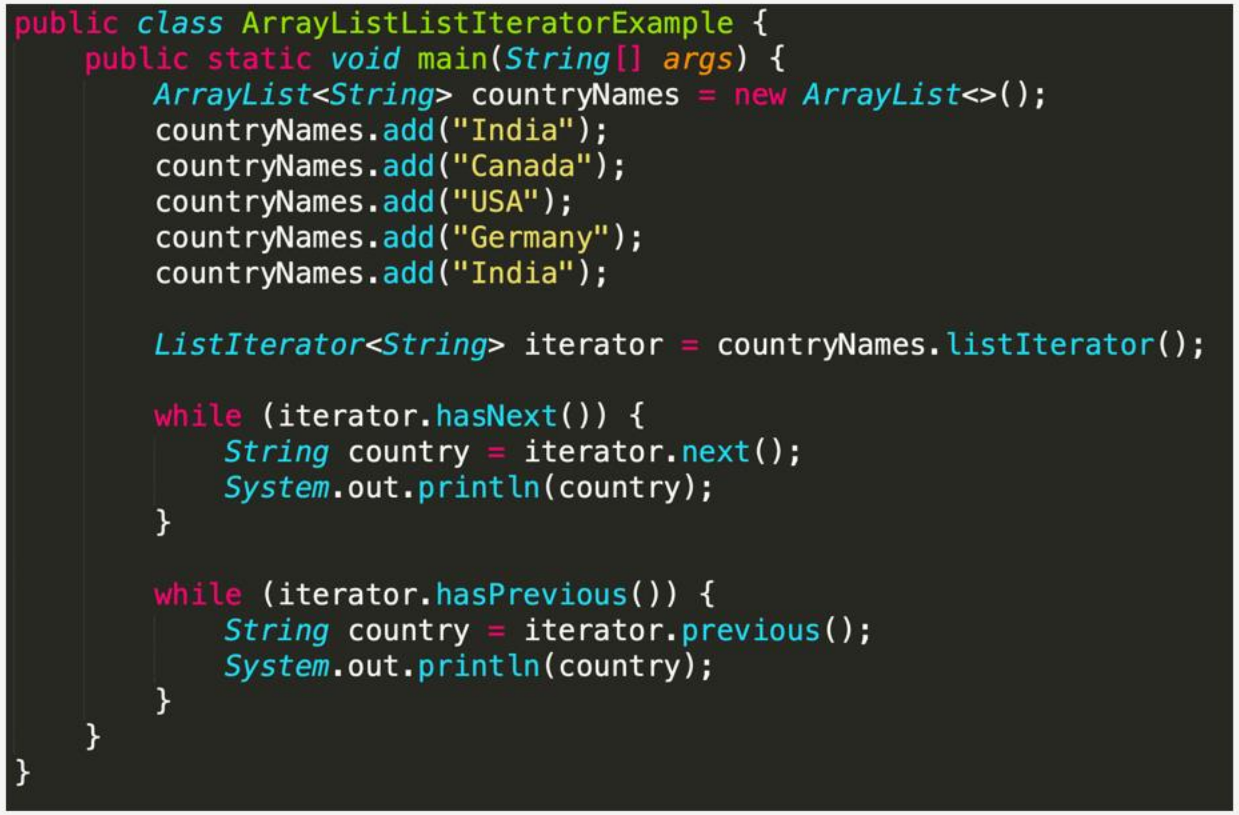
Here's an example of iterating over an ArrayList of Strings using an Iterator & ListIterator :

Using Iterator



```java
public class ArrayListIteratorExample {
    public static void main(String[] args) {
        ArrayList<String> countryNames = new ArrayList<>();
        countryNames.add("India");
        countryNames.add("Canada");
        countryNames.add("USA");
        countryNames.add("Germany");
        countryNames.add("India");

        // create an iterator over the list
        Iterator<String> iterator = countryNames.iterator();

        // iterate over the list using the iterator
        while (iterator.hasNext()) {
            String country = iterator.next();
            System.out.println(country);
        }
    }
}
```

Using ListIterator



```java
public class ArrayListListIteratorExample {
    public static void main(String[] args) {
        ArrayList<String> countryNames = new ArrayList<>();
        countryNames.add("India");
        countryNames.add("Canada");
        countryNames.add("USA");
        countryNames.add("Germany");
        countryNames.add("India");

        ListIterator<String> iterator = countryNames.listIterator();

        while (iterator.hasNext()) {
            String country = iterator.next();
            System.out.println(country);
        }

        while (iterator.hasPrevious()) {
            String country = iterator.previous();
            System.out.println(country);
        }
    }
}
```

Sorting ArrayList using Collections.sort()

Usually, the elements in a ArrayList are stored in an unsorted manner. Let's try to understand how to sort them,

Collections.sort()

Sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must implement the Comparable interface. Furthermore, all elements in the list must be mutually comparable (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the list).The specified list must be modifiable, but need not be resizable.

Collections.sort() example

```
ArrayList<Integer> numbers = new ArrayList<>();
// adding elements to the ArrayList
numbers.add(47);
numbers.add(43);
numbers.add(67);
numbers.add(92);
numbers.add(3);
numbers.add(-67);
numbers.add(-2);
numbers.add(0);
System.out.println(numbers); // Before Sorting - [47, 43, 67, 92, 3, -67, -2, 0]
Collections.sort(numbers); // For ascending order
System.out.println(numbers); // Ascending Order - [-67, -2, 0, 3, 43, 47, 67, 92]
Collections.sort(numbers, Collections.reverseOrder()); // For descending order
System.out.println(numbers); // Descending Order - [92, 67, 47, 43, 3, 0, -2, -67]
```

Sorting ArrayList using sort()

sort()

Sorting can be performed by invoking the instance method sort() using the list object. To this sort() method, we need to provide the Comparator type as an input parameter,

sort() example

```
var countries = new ArrayList<String>();
countries.add("India");
countries.add("USA");
countries.add("Japan");
countries.add("France");
countries.add("Canada");
System.out.println(countries); // Before Sorting - [India, USA, Japan, France, Canada]
countries.sort(Comparator.naturalOrder());
System.out.println(countries); // Ascending Order - [Canada, France, India, Japan, USA]
countries.sort(Comparator.reverseOrder());
System.out.println(countries); // Descending Order - [USA, Japan, India, France, Canada]
```

We we store String elements inside a list, during sorting, the elements will be sorted lexicographically

Sorting ArrayList using custom Comparator

Sometimes, we may want to do the sorting based on our own custom requirements. In these kind of scenarios, we need to create our own Comparator implementation class and need to implement comparing logic in the method compare()

Example to sort the String elements based on last char value,

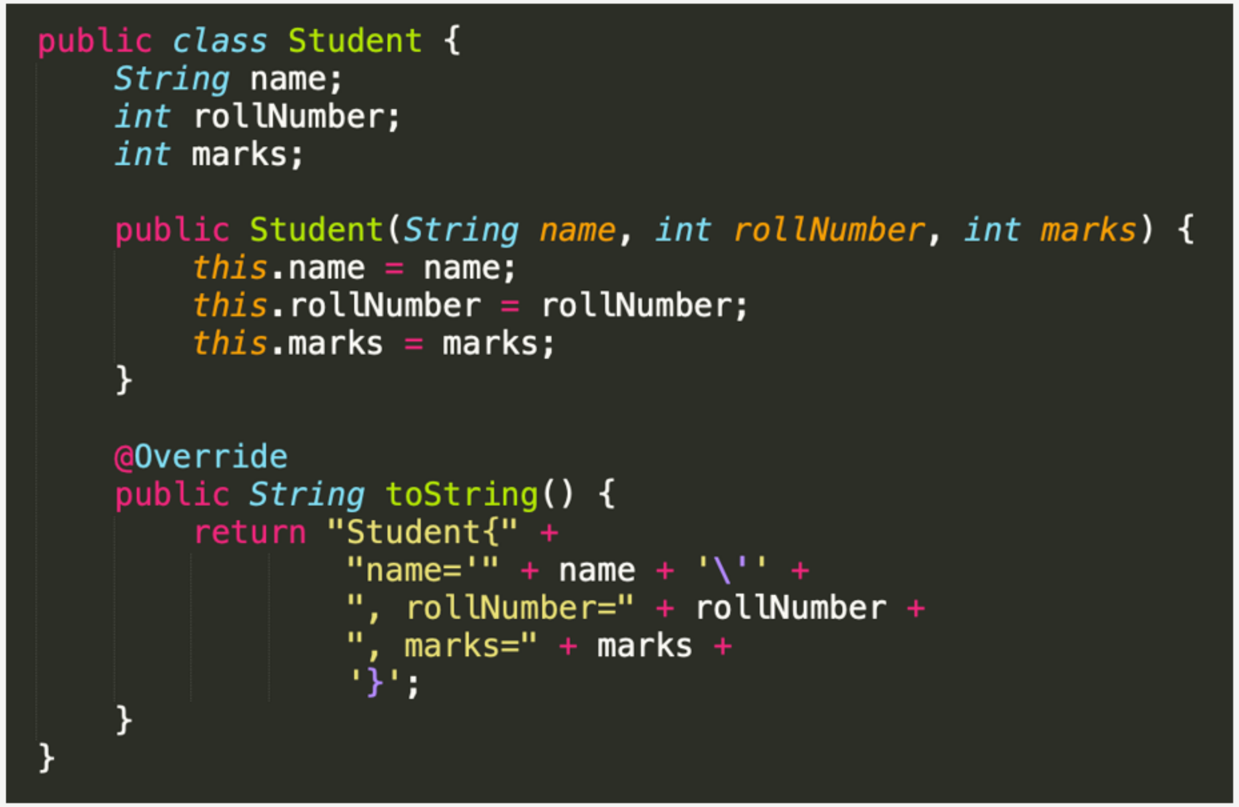
```java
class LastCharComparator implements Comparator<String> {

    @Override
    public int compare(String s1, String s2) {
        // Compare based on the last character of each string
        char lastChar1 = s1.charAt(s1.length() - 1);
        char lastChar2 = s2.charAt(s2.length() - 1);
        return Character.compare(lastChar1, lastChar2);

}
```

```java
ArrayList<String> countries = new ArrayList<>();
countries.add("India");
countries.add("USA");
countries.add("Japan");
countries.add("France");
countries.add("Canada");
System.out.println(countries); // Before Sorting - [India, USA, Japan, France, Canada]
countries.sort(new LastCharComparator());
System.out.println(countries); // After Sorting - [USA, India, Canada, France, Japan]
```

Sorting ArrayList elements of custom types

When dealing with custom types or attempting to compare objects that don't have a natural ordering, it becomes necessary to employ a comparison strategy. This strategy can be crafted by leveraging either the Comparator or Comparable interfaces.

Let's take an example of a Student class, where we want to sort the Students based on their marks and rollNumber,



```java
public class Student {
    String name;
    int rollNumber;
    int marks;

    public Student(String name, int rollNumber, int marks) {
        this.name = name;
        this.rollNumber = rollNumber;
        this.marks = marks;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", rollNumber=" + rollNumber +
                ", marks=" + marks +
                '}';
    }
}
```

```java
List<Student> students = new ArrayList<>();
students.add(new Student("John", 167, 97));
students.add(new Student("Smith", 168, 92));
students.add(new Student("Will", 169, 92));
students.add(new Student("Madan", 170, 99));
System.out.println("Before Sorting : " + students);
Collections.sort(students); // Compilation fails
System.out.println("After Sorting : " + students);
```

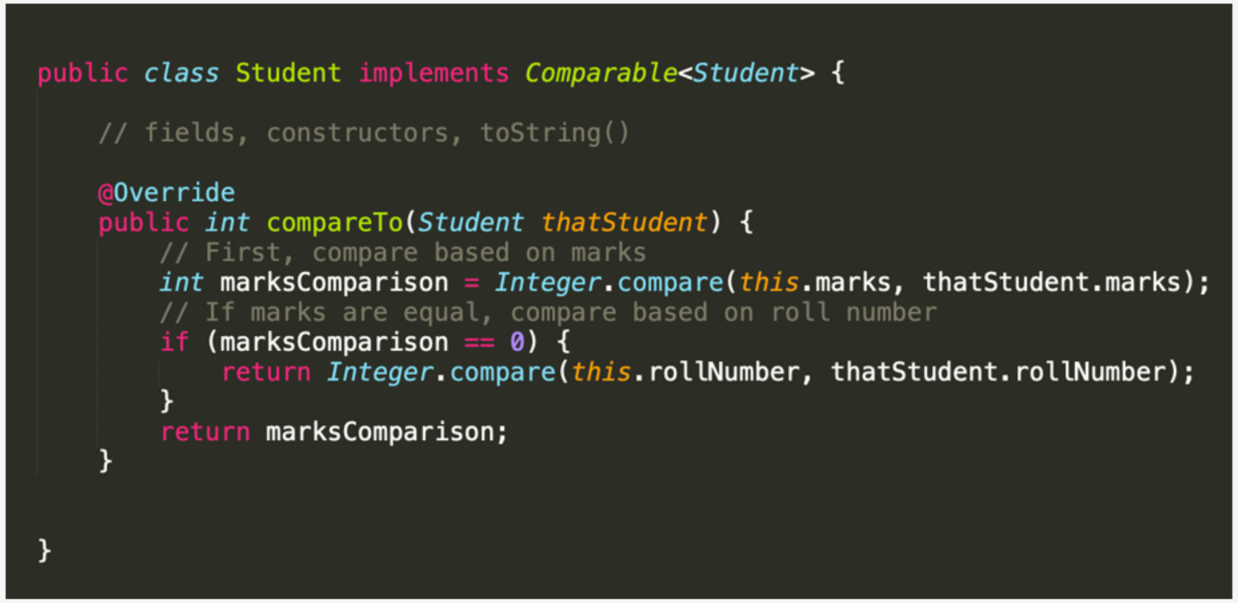
If we try to sort the Student objects using Collection.sort(), we will receive an compilation error.

As the name suggests, Comparable is an interface defining a strategy of comparing an object with other objects of the same type. This is called the class's "natural ordering."

In order to be able to sort, we must define our Student object as comparable by implementing the Comparable interface:

Sorting ArrayList elements of custom types

Student class objects can be compared only when they implement Comparable interface and compareTo() like shown below. The sorting order is decided by the return value of the compareTo() method. The Integer.compare(x, y) returns -1 if x is less than y, 0 if they're equal, and 1 otherwise. The method returns a number indicating whether the object being compared is less than, equal to, or greater than the object being passed as an argument.



```java
public class Student implements Comparable<Student> {

    // fields, constructors, toString()

    @Override
    public int compareTo(Student thatStudent) {
        // First, compare based on marks
        int marksComparison = Integer.compare(this.marks, thatStudent.marks);
        // If marks are equal, compare based on roll number
        if (marksComparison == 0) {
            return Integer.compare(this.rollNumber, thatStudent.rollNumber);
        }
        return marksComparison;
    }


}
```

```java
List<Student> students = new ArrayList<>();
students.add(new Student("John", 167, 97));
students.add(new Student("Smith", 168, 92));
students.add(new Student("Will", 169, 92));
students.add(new Student("Madan", 170, 99));
System.out.println("Before Sorting : " + students);
Collections.sort(students); // Sorting success
System.out.println("After Sorting : " + students);
```

When using the Comparable approach, we are writing the comparing logic inside the POJO class itself. Inside the compareTo() method, we compare with the current (this) object with another object received as an input to the method.

Sorting ArrayList elements of custom types

The other option to compare custom types is by creating a Comparator implement class which is specific to the custom type like Student. Below is an example of the same. The Comparator interface defines a compare(arg1, arg2) method with two arguments that represent compared objects, and works similarly to the Comparable.compareTo() method.

```java
public class StudentComparator implements Comparator<Student> {

    @Override
    public int compare(Student student1, Student student2) {
        // First, compare based on marks
        int marksComparison = Integer.compare(student1.marks, student2.marks);
        // If marks are equal, compare based on roll number
        if (marksComparison == 0) {
            return Integer.compare(student1.rollNumber, student2.rollNumber);
        }
        return marksComparison;
    }

}
```

```
Collections.sort(students, new StudentComparator());
```

(or)

```
students.sort(new StudentComparator());
```

With Comparator approach, the comparing logic is maintained in a separate logic with out making any changes inside Student. This allow us to create any number of Comparator implementation classes for Student and use them based on our needs.

Comparable vs Comparator

The Comparable interface is suitable for establishing the default ordering or when it serves as the primary means of comparing objects.

However, there are situations where employing a Comparator is more appropriate for several reasons:

1. In cases where we lack the ability to modify the source code of the class we want to sort, rendering the use of Comparable unfeasible.

2. The use of Comparators allows us to sidestep the necessity of adding extra code to our domain classes.

3. By employing Comparators, we can define multiple distinct comparison strategies, a flexibility not achievable when relying solely on Comparable.

Arrays vs ArrayList

Arrays have a fixed size and cannot be resized. To change the size of an array, a new array must be created.	ArrayLists can be resized dynamically by adding or removing elements as needed.
Memory allocation for arrays is static and contiguous.	Memory allocation for ArrayLists is dynamic and contiguous
Arrays can hold both primitives and objects.	ArrayLists can only hold objects, not primitives.
The length of an array can be obtained using the length property.	The length of an ArrayList can be obtained using the size() method.

```
ArrayList<String> countryNames = new ArrayList<>();
countryNames.add("India");
countryNames.add("Canada");
countryNames.add("USA");
countryNames.add("Germany");
countryNames.add("India");
// Sample code to convert ArrayList to Array & viceversa
String[] countries = countryNames.toArray(new String[countryNames.size()]);
List<String> newList = new ArrayList<String>();
newList =  Arrays.asList(countries);
```

Vector

For experienced Java developers, the Vector class has historically been a go-to choice for dynamic arrays. However, there's a compelling reason to favor ArrayList over Vector: all methods of the Vector class are synchronized.

While it ensures the safety of accessing a Vector object from multiple threads, the majority of scenarios involve single-threaded access. In such cases, the synchronization overhead in Vector methods becomes unnecessary.

On the other hand, ArrayList methods are not synchronized, making them more efficient in single-threaded scenarios. It is advisable to opt for ArrayList over Vector when synchronization is not required.