

Array List in java:

The ArrayList class is a part of the Java Collections Framework and is used to create resizable arrays. It implements the List interface and provides dynamic arrays that can grow or shrink as needed. Unlike regular arrays in Java, ArrayLists can store elements of any type.

Here are some important characteristics and features of ArrayLists:

1. **Resizable:** ArrayLists automatically resize themselves as elements are added or removed. This means you don't need to worry about specifying the size upfront.
2. **Indexed Access**: ArrayLists provide efficient indexed access to elements. Each element in the ArrayList is assigned a unique index starting from 0, and you can retrieve or modify elements using these indices.
3. **Dynamic**: ArrayLists can grow or shrink as needed. When elements are added, the ArrayList automatically increases its capacity to accommodate the new elements. Similarly, when elements are removed, the capacity is reduced.
4. **Generic:** ArrayLists support generics, which means you can specify the type of elements that an ArrayList can hold. For example, you can create an ArrayList of integers, strings, objects, or any other Java class.
5. **Null Elements and Duplicates:** ArrayLists can contain null elements, and they can also store duplicate elements. It does not enforce uniqueness unless you explicitly implement your own logic.

Now, let us look at some commonly used methods and operations provided by the ArrayList class:

- **Adding Elements:**
    - add(element): Adds the specified element to the end of the ArrayList.
    - add (index, element): Inserts the specified element at the specified position in the ArrayList.
- **Accessing Elements**:
    - get(index): Retrieves the element at the specified index.
    - set(index, element): Replaces the element at the specified index with the specified element.
- **Removing Elements:**
    - remove(index): Removes the element at the specified index from the ArrayList.
    - remove(element): Removes the first occurrence of the specified element from the ArrayList.

- **Size and Capacity:**
    - size(): Returns the number of elements currently stored in the ArrayList.
    - isEmpty(): Checks if the ArrayList is empty.
    - trimToSize(): Reduces the capacity of the ArrayList to the current size.
- **Iteration and Conversion:**
    - iterator(): Returns an iterator over the elements in the ArrayList.
    - toArray(): Returns an array containing all elements in the ArrayList.

These are just a few examples of the methods available in the ArrayList class. There are many more methods and functionalities provided by the class to manipulate and work with the ArrayList.

Remember that ArrayLists are not synchronized, meaning they are not thread-safe by default. If you need to use ArrayLists in a multi-threaded environment, you should consider using the java.util.concurrent.CopyOnWriteArrayList class or synchronize access to the ArrayList using synchronization constructs.

Resizing array mechanism:

Resizing an ArrayList in Java involves adjusting its internal array to accommodate more or fewer elements. The resizing operation is performed automatically by the ArrayList class when necessary, based on the number of elements stored and the current capacity. Let's explore the resizing process in more detail:

1. **Initial Capacity:**
    - When an ArrayList is created without specifying an initial capacity, it starts with a default initial capacity, which is usually 10. The initial capacity can also be set explicitly during construction.
    - The internal array of the ArrayList is initially created with this initial capacity, regardless of the number of elements initially added.
2. **Adding Elements:**
    - As elements are added to the ArrayList using the add() method, the size of the ArrayList increases. When the size becomes equal to the current capacity, the ArrayList needs to be resized to accommodate more elements.
    - Resizing involves creating a new array with a larger capacity and copying all existing elements from the old array to the new array.
    - By default, when resizing is necessary, the new capacity is calculated by doubling the current capacity. However, this behavior can be modified by specifying a custom resizing strategy using the ensureCapacity() method.

3. **Resizing Process:**

   - The resizing process starts by creating a new array with the increased capacity.
   - All existing elements from the old array are copied to the new array. This involves iterating over the elements of the old array and assigning them to the corresponding positions in the new array.
   - Once all elements are copied, the internal reference of the ArrayList is updated to point to the new array, and the old array becomes eligible for garbage collection.

4. **Removing Elements:**

   - When elements are removed from the ArrayList using the remove() method, the size of the ArrayList decreases. However, the capacity remains the same.
   - Unlike adding elements, removing elements does not trigger an immediate resize operation. The capacity remains unchanged unless explicitly modified using the trimToSize() method.

5. **Manually Adjusting Capacity:**

   - If you know in advance the maximum number of elements the ArrayList will hold, you can set the initial capacity accordingly to avoid frequent resizing.
   - To manually adjust the capacity of an ArrayList, you can use the ensureCapacity() method to increase the capacity or the trimToSize() method to reduce the capacity to match the current size.

It's important to note that resizing an ArrayList involves a performance cost due to the copying of elements from the old array to the new array. Therefore, it's recommended to set a reasonable initial capacity and avoid frequent resizing operations when possible.

Additionally, be aware that ArrayLists can be created with the ArrayList<E> constructor, where E represents the type of elements to be stored. The ArrayList<E> constructor can take an initial capacity as an argument to pre-allocate memory if the number of elements is known in advance.

HashSet in java:

The HashSet class is a part of the Java Collections Framework and implements the Set interface. It is used to create an unordered collection that does not allow duplicate elements. HashSet uses a hash table to store elements, providing constant-time performance for basic operations like adding, removing, and checking for the presence of elements.

Here are some important characteristics and features of HashSet:

1. **Unordered Collection:** HashSet does not maintain the order of elements. The order in which elements are inserted into a HashSet is not necessarily the order in which they are stored.

2. **No Duplicates:** HashSet does not allow duplicate elements. If you attempt to add an element that already exists in the HashSet, the add() method will return false, indicating that the element was not added.
3. **Hashing Mechanism:** HashSet uses the concept of hashing to efficiently store and retrieve elements. It computes the hash code of each element and uses this hash code to determine the bucket (position) where the element is stored in the underlying hash table.
4. **Hash Collision**: In some cases, two or more elements may have the same hash code, resulting in a hash collision. HashSet handles hash collisions by using a linked list to store multiple elements with the same hash code in the same bucket.
5. **Null Elements:** HashSet allows null elements. You can add a single null element to a HashSet.

Now, let's look at some commonly used methods and operations provided by the HashSet class:

- **Adding Elements:**
    - add(element): Adds the specified element to the HashSet if it is not already present.
- **Removing Elements:**
    - remove(element): Removes the specified element from the HashSet if it is present.
    - clear(): Removes all elements from the HashSet.
- **Checking Existence:**
    - contains(element): Checks if the HashSet contains the specified element.
    - isEmpty(): Checks if the HashSet is empty.
- **Size and Iteration:**
    - size(): Returns the number of elements in the HashSet.
    - iterator(): Returns an iterator over the elements in the HashSet.
- **Set Operations:**
    - **addAll(collection):** Adds all elements from the specified collection to the HashSet.
    - **retainAll(collection):** Removes all elements from the HashSet except those present in the specified collection.
    - **removeAll(collection):** Removes all elements from the HashSet that are also present in the specified collection.

These are some of the commonly used methods provided by the HashSet class. HashSet does not provide direct indexed access to elements since it does not maintain any specific order.

It's important to note that HashSet is not synchronized, meaning it is not thread-safe by default. If you need to use HashSet in a multi-threaded environment, you should consider using the java.util.concurrent.ConcurrentHashSet class or synchronize access to the HashSet using synchronization constructs.

LinkedList in java:

The LinkedList class is a part of the Java Collections Framework and implements the List interface. It provides a doubly-linked list data structure, where each element in the list is represented by a node containing a reference to the previous and next nodes. LinkedList allows for efficient insertion and deletion operations, but accessing elements by index is slower compared to ArrayList.

Here are some important characteristics and features of LinkedList:

1. **Doubly-Linked Structure:** Each element in the LinkedList is represented by a node that contains references to the previous and next nodes. This allows for efficient insertion and removal of elements, as only the references need to be updated.
2. **Dynamic Size:** LinkedList can grow or shrink dynamically as elements are added or removed. Unlike regular arrays, LinkedList does not require a fixed size.
3. **No Random Access:** Unlike ArrayList, LinkedList does not provide efficient indexed access to elements. To access an element, you need to traverse the list from the beginning or end until the desired element is reached.
4. **Efficient Insertion and Removal:** Inserting or removing elements in the middle of the LinkedList is faster compared to ArrayList. This is because LinkedList only needs to update the references of neighboring nodes, while ArrayList requires shifting elements to accommodate the change.
5. **Null Elements:** LinkedList allows null elements, meaning you can store null values in the list.

Now, let's look at some commonly used methods and operations provided by the LinkedList class:

- **Adding Elements:**
    - add(element): Adds the specified element to the end of the LinkedList.
    - add(index, element): Inserts the specified element at the specified position in the LinkedList.
- **Accessing Elements:**
    - get(index): Retrieves the element at the specified index by traversing the LinkedList.
    - getFirst(): Retrieves the first element in the LinkedList.
    - getLast(): Retrieves the last element in the LinkedList.
- **Removing Elements:**

- remove(index): Removes the element at the specified index from the LinkedList.
- removeFirst(): Removes and returns the first element from the LinkedList.
- removeLast(): Removes and returns the last element from the LinkedList.

- **Size and Iteration:**
  - size(): Returns the number of elements in the LinkedList.
  - isEmpty(): Checks if the LinkedList is empty.
  - iterator(): Returns an iterator over the elements in the LinkedList.
- **Conversion:**
  - toArray(): Returns an array containing all elements in the LinkedList.

These are some of the commonly used methods provided by the LinkedList class. LinkedList is particularly useful when frequent insertion and removal operations are required, but indexed access is not critical for your application.

It's important to note that LinkedList is not synchronized, meaning it is not thread-safe by default. If you need to use LinkedList in a multi-threaded environment, you should consider using the java.util.concurrent.ConcurrentLinkedDeque class or synchronize access to the LinkedList using synchronization constructs.

Difference between ArrayList and LinkedList:

The main differences between LinkedList and ArrayList in Java are related to their underlying data structures and the performance characteristics of the operations they support. Here are the key distinctions:

1. **Data Structure:**
   - LinkedList: It is implemented as a doubly-linked list, where each element is stored in a separate node that contains references to the previous and next nodes. This structure allows for efficient insertion and removal of elements at any position in the list.
   - ArrayList: It is implemented as a resizable array. Elements are stored in contiguous memory locations, and accessing elements by index is efficient. ArrayList provides fast indexed access but has slower insertion and removal operations in the middle of the list.
2. **Performance:**
   - Indexed Access: ArrayList provides constant-time (O(1)) performance for indexed access, as it directly accesses elements based on their position in the array. LinkedList requires traversing the list to reach the desired element, resulting in linear-time (O(n)) performance for indexed access.

- Insertion and Removal: LinkedList has faster insertion and removal operations in the middle of the list, as it only requires updating references. In ArrayList, inserting or removing an element in the middle of the list requires shifting subsequent elements, resulting in slower performance.
- Dynamic Behavior: LinkedList is more efficient in dynamically growing or shrinking the list, as it does not require resizing and copying like ArrayList. ArrayList performs better when the size is known in advance or when the focus is primarily on indexed access.

3. **Memory Overhead:**
   - LinkedList: It requires extra memory to store the references for the previous and next nodes. In addition, each element is stored in a separate node object, which incurs additional memory overhead.
   - ArrayList: It has a more compact memory representation since it stores elements in a contiguous array. However, it may need to allocate extra capacity to accommodate potential future growth.
4. **Use Cases:**
   - LinkedList is suitable when frequent insertion and removal of elements in the middle of the list are required, or when the order of elements is important.
   - ArrayList is suitable when indexed access to elements is crucial, and when the list size is relatively fixed or changes less frequently.

It's important to note that both LinkedList and ArrayList provide similar functionality and support the List interface, including common operations like adding, removing, and searching for elements. The choice between them depends on the specific requirements and performance considerations of your application.

Examples in class :

```
package setdemo;

import iteratordemo.Person;

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;

public class HashSetDemo {
   public static void main(String[] args) {
      Set<Person> pSet = new HashSet<>();
      Person p1 = new Person(20, "John1");
```



```
        Person p2 = new Person(20, "John1");
        Person p3 = new Person(20, "John3");
        Person p4 = new Person(20, "John4");
        pSet.add(p1);
        pSet.add(p2);
        pSet.add(p3);
        pSet.add(p4);
        //Added p1 again
        pSet.add(p1);
        System.out.println("Size is "+pSet.size());
        Iterator<Person> pIterator = pSet.iterator();
        //loop till hasNext method returns false which will be end of the list
        while (pIterator.hasNext()){
            Person p = pIterator.next();
            System.out.println("Name of person id "+p.getName());
        }
        Set<String> strSet = new HashSet<>();
        strSet.add("abc");
        strSet.add("def");
        strSet.add("abc");

        System.out.println("Size of strSet : "+strSet.size());

    }
}
```

Linked list program:

```
public class Node {
    private String value;
    private Node nextNode;
    //private int age
    //private Address address;
    public String getValue() {
        return value;
    }
    public void setValue(String value) {
        this.value = value;
    }
    public Node getNextNode() {
        return nextNode;
    }
    public void setNextNode(Node nextNode) {
        this.nextNode = nextNode;
```



```
    }
    @Override
    public String toString() {
        return "Node{" +
                "value='" + value + '\'' +
                ", nextNode=" + nextNode +
                '}';
    }
}
// abc -> def -> ijk
//Node{value='abc', nextNode=Node{value='def', nextNode=Node{value='ijk', nextNode=null}}}
//Node{value='abc', nextNode=Node{value='def', nextNode=Node{value='ijk', nextNode=null}}}
//                    Node{value='abc', nextNode=Node{value='def', nextNode=Node{value='ijk',
nextNode=Node{value='xyz', nextNode=null}}}}
//MyLinkedList{currentNode=Node{value='abc', nextNode=Node{value='def',
nextNode=Node{value='ijk', nextNode=Node{value='xyz', nextNode=null}}}}}
```

```
public class MyLinkedList {
    private Node currentNode;

    public void addElement(String value){
        if(currentNode ==  null){
            Node node = new Node();
            node.setValue(value);
            currentNode = node;
        }else{
            Node lastNode = currentNode;
           while (lastNode.getNextNode() != null){
               lastNode = lastNode.getNextNode();
           }
           Node newNode = new Node();
           newNode.setValue(value);
           lastNode.setNextNode(newNode);
        }
    }

    public Node getCurrentNode() {
        return currentNode;
    }

    public void setCurrentNode(Node currentNode) {
        this.currentNode = currentNode;
    }

    @Override
```



```
    public String toString() {
        return "MyLinkedList{" +
                "currentNode=" + currentNode +
                '}';
    }
}
```

```
import java.util.LinkedList;

public class LinkedListDemo {

    public static void main(String[] args) {
//        Node firstNode = new Node();
//        firstNode.setValue("abc");
//
//        Node defNode = new Node();
//        defNode.setValue("def");
//        firstNode.setNextNode(defNode);
//
//        Node ijkNode = new Node();
//        ijkNode.setValue("ijk");
//        firstNode.getNextNode().setNextNode(ijkNode);
//        //defNode.setNextNode(ijkNode);
//
//        Node xyzNode = new Node();
//        xyzNode.setValue("xyz");
//        firstNode.getNextNode()
//                .getNextNode()
//                .setNextNode(xyzNode);
//
//        System.out.println(firstNode);

        MyLinkedList myLinkedList = new MyLinkedList();

        myLinkedList.addElement("abc");
        myLinkedList.addElement("def");
        myLinkedList.addElement("ijk");
        myLinkedList.addElement("xyz");

        System.out.println(myLinkedList);


        LinkedList<String> linkedList = new LinkedList<>();
        linkedList.add("abc");
```



```
        linkedList.add("def");
        linkedList.add("ijk");
        linkedList.add("xyz");
        System.out.println(linkedList);



    }
}
```