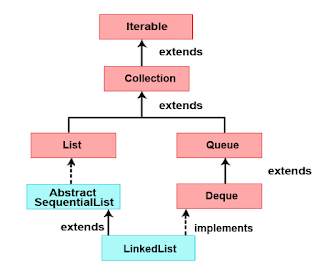
**Guide to LinkedList Class**

In this guide, we will see the Implementation of *LinkedList* class with examples.

Java *LinkedList* class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the *AbstractList* class and implements *List* and *Deque* interfaces.

**The important points about Java LinkedList are:**

****

* Java *LinkedList* class can contain duplicate elements.
* Java *LinkedList* class maintains insertion order.
* In Java *LinkedList* class, manipulation is fast because no shifting needs to have occurred.
* The *LinkedList* class implements *Queue* and *Deque* interfaces. Therefore, It can also be used as a Queue, Deque or Stack.
* Java *LinkedList* is not thread-safe. You must explicitly synchronize concurrent modifications to the LinkedList in a multi-threaded environment.

**1. What Will We Learn?**

1. Overview of *LinkedList* class
2. Add elements *LinkedList* API's with Examples

* *add(String element)*
* *add(int index, String element)*
* *addFirst(String e)*
* *addLast(String e)*
* *addAll(Collection<? extends String> c)*

1. Remove elements *LinkedList* API's with Examples

* *removeFirst()*
* *removeLast()*
* *remove(Object o)*
* *removeIf(Predicate<? super String> filter)*
* *clear()*

1. Retrieve elements *LinkedList* API's with Examples

* *getFirst()*
* *getLast()*
* *get(int index)*

1. Search elements *LinkedList* API's with Examples

* *contains(Object o)*
* *indexOf(Object o)*
* *lastIndexOf(Object o)*

1. Iterate *LinkedList* using Java 8 *forEach()* method

* *forEach(Consumer<? super String> action)*
* *Iterator java.util.AbstractSequentialList.iterator()*

1. Difference between *ArrayList* and *LinkedList*

**2. Add elements *LinkedList* API's with Examples**

Let's discuss adding elements to *LinkedList* using different methods.

LinkedList<String> linkedList = new LinkedList<>();

***add(String element)***

Adding new elements to the end of the LinkedList using *add()* method.

LinkedList<String> linkedList = new LinkedList<>();

linkedList.add("A");

linkedList.add("B");

linkedList.add("C");

***add(int index, String element)***

Adding an element at the specified position in the LinkedList using index method - *add(index, element)*

LinkedList<String> linkedList = new LinkedList<>();

linkedList.add("A");

linkedList.add("B");

linkedList.add("C");

linkedList.add(3, "L");

System.out.println("After add(3, \"D\") : " + linkedList);

***addFirst(String e)***

Adding an element at the beginning of the LinkedList using *addFirst()* method

LinkedList<String> linkedList = new LinkedList<>();

linkedList.addFirst("R");

***addLast(String e)***

Adding an element at the end of the LinkedList using *addLast()* method

LinkedList<String> linkedList = new LinkedList<>();

linkedList.add("A");

linkedList.add("B");

linkedList.add("C");

linkedList.add(3, "L");

linkedList.addLast("J");

System.out.println("After addLast(\"F\") : " + linkedList);

***addAll(Collection<? extends String> c)***

Adding all the elements from an existing collection to the end of the LinkedList

List<String> secondList= new ArrayList<>();

secondList.add("Jesse");

secondList.add("Walt");

linkedList.addAll(secondList);

System.out.println("After addAll(secondList) : " + linkedList);

**3. Remove elements *LinkedList* API's with Examples**

Let's first create *LinkedList* with few fruits and then use remove methods to remove fruits from *LinkedList*.

LinkedList<String> fruitList = new LinkedList<>();

fruitList.add("Apple");

fruitList.add("banana");

fruitList.add("mango");

fruitList.add("Pinaple");

System.out.println("Initial LinkedList = " + fruitList);

***removeFirst()***

Remove the first element in the LinkedList. Throws NoSuchElementException if the LinkedList is empty.

String element = fruitList.removeFirst();

System.out.println("Removed the first element " + element + " => " + fruitList);

***removeLast()***

Remove the last element in the *LinkedList*. Throws *NoSuchElementException* if the LinkedList is empty

element = fruitList.removeLast();

System.out.println("Removed the last element " + element + " => " + fruitList);

***remove(Object o)***

Remove the first occurrence of the specified element from the LinkedList

boolean isRemoved = fruitList.remove("banana");

if(isRemoved) {

System.out.println("Removed banana => " + fruitList);

}

***removeIf(Predicate<? super String> filter)***

Removes all of the elements of this collection that satisfy the given predicate. Errors or runtime exceptions thrown during iteration or by the predicate are relayed to the caller.

fruitList.removeIf(programmingLanguage -> programmingLanguage.startsWith("C"));

System.out.println("Removed elements starting with C => " + fruitList);

***clear()***

Removes all of the elements from this list. The list will be empty after this call returns.

fruitList.clear();

System.out.println("Cleared the LinkedList => " + fruitList);

**4. Retrieve elements *LinkedList* API's with Examples**

A LinkedList containing Stock Prices of a company for the last 6 days

LinkedList<Double> stockPrices = new LinkedList<>();

stockPrices.add(45.00);

stockPrices.add(51.00);

stockPrices.add(62.50);

stockPrices.add(42.75);

stockPrices.add(36.80);

stockPrices.add(68.40);

***getFirst()***

Returns the first element in this list.

Double firstElement = stockPrices.getFirst();

System.out.println("Initial Stock Price : " + firstElement);

***getLast()***

Returns the last element in this list.

Double lastElement = stockPrices.getLast();

System.out.println("Current Stock Price : " + lastElement);

***get(int index)***

Getting the element at a given position in the LinkedList

Double stockPriceOn3rdDay = stockPrices.get(2);

System.out.println("Stock Price on 3rd Day : " + stockPriceOn3rdDay);

**5. Search elements *LinkedList* API's with Examples**

Let's create a list of employees and apply *LinkedList* search methods.

LinkedList<String> employees = new LinkedList<>();

employees.add("John");

employees.add("David");

employees.add("Lara");

employees.add("Chris");

employees.add("Steve");

employees.add("David");

***contains(Object o)***

Check if the LinkedList contains an element

System.out.println("Does Employees LinkedList contain \"Lara\"? : " + employees.contains("Lara"));

***indexOf(Object o)***

Find the index of the first occurrence of an element in the LinkedList

System.out.println("indexOf \"Steve\" : " + employees.indexOf("Steve"));

System.out.println("indexOf \"Mark\" : " + employees.indexOf("Mark"));

***lastIndexOf(Object o)***

Find the index of the last occurrence of an element in the LinkedList

System.out.println("lastIndexOf \"David\" : " + employees.lastIndexOf("David"));

System.out.println("lastIndexOf \"Bob\" : " + employees.lastIndexOf("Bob"));

**6. Iterating over *LinkedList***

package com.javaguides.collections.linkedlistexamples;

import java.util.Iterator;

import java.util.LinkedList;

import java.util.ListIterator;

public class IterateOverLinkedListExample {

public static void main(String[] args) {

LinkedList < String > progLangs = new LinkedList < > ();

progLangs.add("C");

progLangs.add("C++");

progLangs.add("Core Java");

progLangs.add("Java EE");

progLangs.add("Spring Framework");

progLangs.add("Hibernate Framework");

System.out.println("=== Iterate over a LinkedList using Java 8 forEach and lambda ===");

progLangs.forEach(name - > {

System.out.println(name);

});

System.out.println("\n=== Iterate over a LinkedList using iterator() ===");

Iterator < String > iterator = progLangs.iterator();

while (iterator.hasNext()) {

String speciesName = iterator.next();

System.out.println(speciesName);

}

System.out.println("\n=== Iterate over a LinkedList using iterator() and Java 8 forEachRemaining() method ===");

iterator = progLangs.iterator();

iterator.forEachRemaining(speciesName - > {

System.out.println(speciesName);

});

System.out.println("\n=== Iterate over a LinkedList using descendingIterator() ===");

Iterator < String > descendingIterator = progLangs.descendingIterator();

while (descendingIterator.hasNext()) {

String speciesName = descendingIterator.next();

System.out.println(speciesName);

}

System.out.println("\n=== Iterate over a LinkedList using listIterator() ===");

// ListIterator can be used to iterate over the LinkedList in both forward and backward directions

// In this example, we start from the end of the list and traverse backwards

ListIterator < String > listIterator = progLangs.listIterator(progLangs.size());

while (listIterator.hasPrevious()) {

String speciesName = listIterator.previous();

System.out.println(speciesName);

}

System.out.println("\n=== Iterate over a LinkedList using simple for-each loop ===");

for (String name: progLangs) {

System.out.println(name);

}

}

}

**Output**

=== Iterate over a LinkedList using Java 8 forEach and lambda ===

C

C++

Core Java

Java EE

Spring Framework

Hibernate Framework

=== Iterate over a LinkedList using iterator() ===

C

C++

Core Java

Java EE

Spring Framework

Hibernate Framework

=== Iterate over a LinkedList using iterator() and Java 8 forEachRemaining() method ===

C

C++

Core Java

Java EE

Spring Framework

Hibernate Framework

=== Iterate over a LinkedList using descendingIterator() ===

Hibernate Framework

Spring Framework

Java EE

Core Java

C++

C

=== Iterate over a LinkedList using listIterator() ===

Hibernate Framework

Spring Framework

Java EE

Core Java

C++

C

=== Iterate over a LinkedList using simple for-each loop ===

C

C++

Core Java

Java EE

Spring Framework

Hibernate Framework

**7. Difference Between ArrayList and LinkedList**

1. *ArrayList* internally uses a dynamic array to store the elements. *LinkedList* internally uses the doubly linked list to store the elements.
2. Manipulation with *ArrayList* is slow because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory. Manipulation with *LinkedList* is faster than *ArrayList* because it uses a doubly linked list so no bit shifting is required in memory.
3. *ArrayList* class can act as a list only because it implements *List* only. *LinkedList* class can act as a list and queue both because it implements *List* and *Deque* interfaces.
4. *ArrayList* is better for storing and accessing data. *LinkedList* is better for manipulating data.