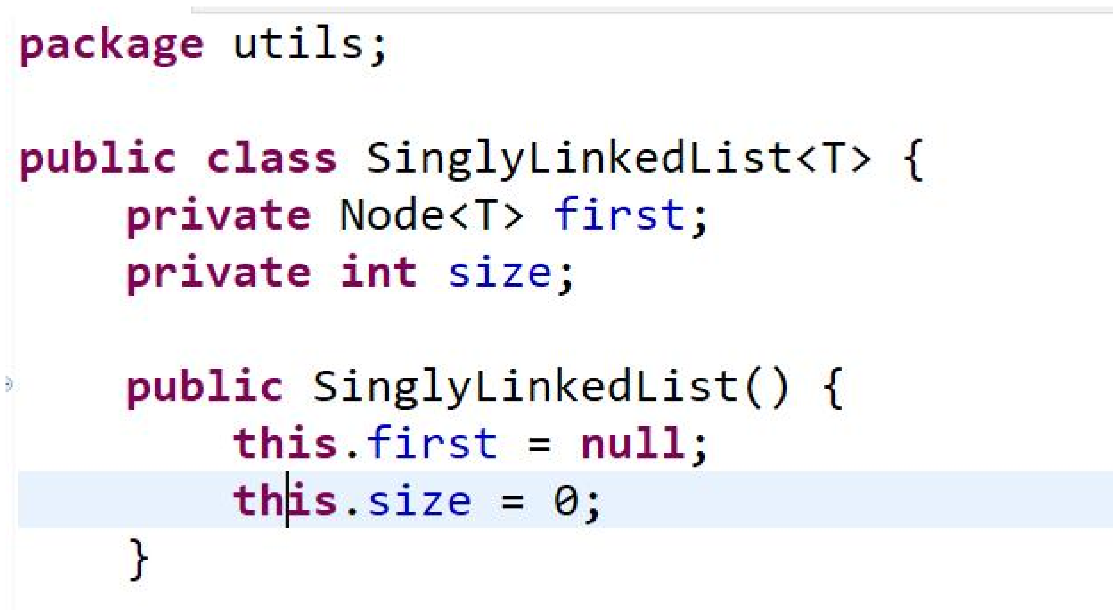
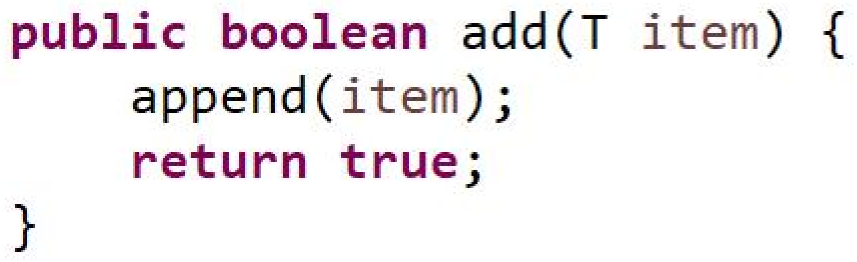


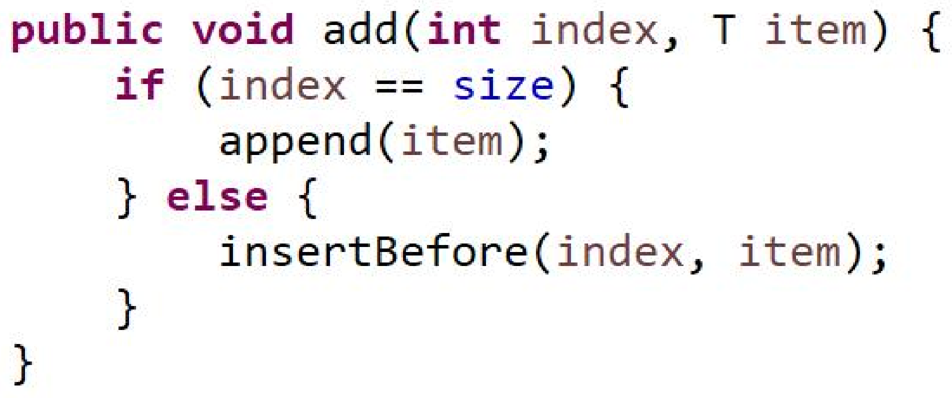
```java
package utils;

public class SinglyLinkedList<T> {
    private Node<T> first;
    private int size;

    public SinglyLinkedList() {
        this.first = null;
        this.size = 0;
    }
```

This portion of code defines the SinglyLinkedList class and its constructor.
first is a reference to the first node in the linked list, initially set to null.
size keeps track of the number of elements in the linked list, initially set to 0.



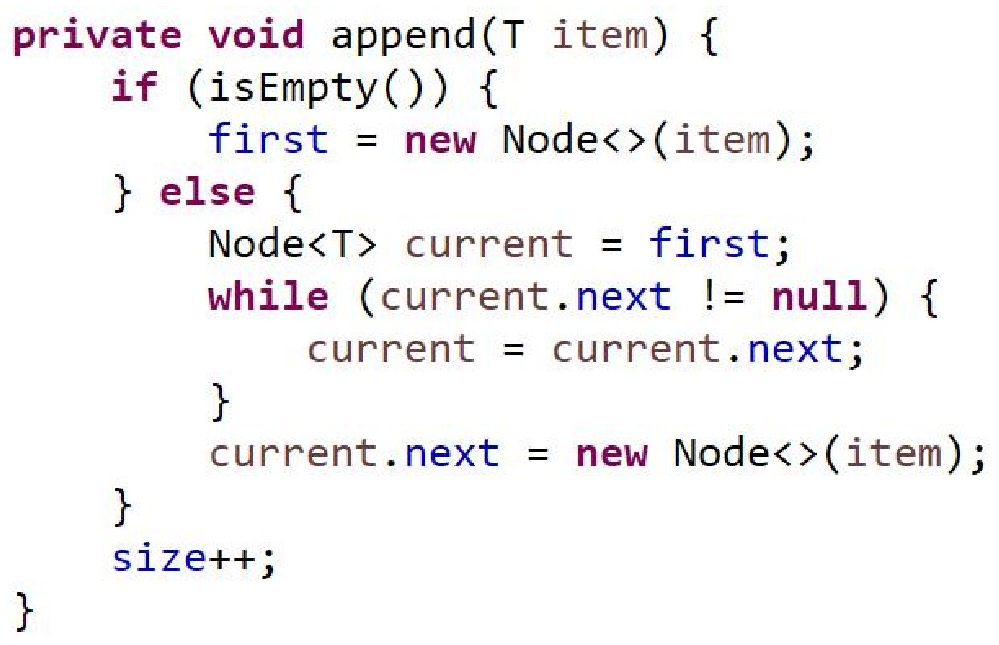
```java
    public boolean add(T item) {
        append(item);
        return true;
    }
```

The add method is used to add an element to the end of the linked list.
It calls the append method to add the element to the end of the linked list and returns true.



```java
    public void add(int index, T item) {
        if (index == size) {
            append(item);
        } else {
            insertBefore(index, item);
        }
    }
```

This is an overloaded version of the add method, which allows inserting an element at a specified index.
If the index is equal to the size of the linked list, it calls the append method to add the element to the end of the list.
Otherwise, it calls the insertBefore method to insert the element before the specified index.
java



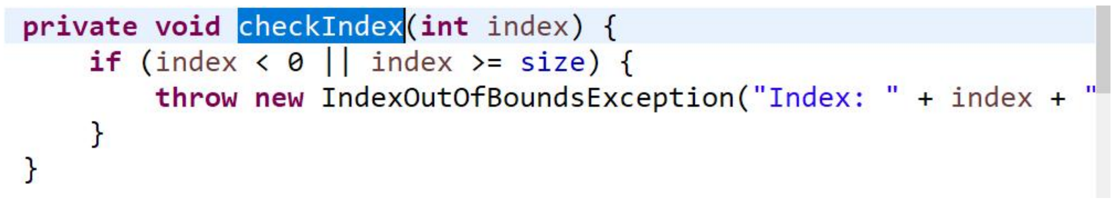
```java
private void append(T item) {
    if (isEmpty()) {
        first = new Node<>(item);
    } else {
        Node<T> current = first;
        while (current.next != null) {
            current = current.next;
        }
        current.next = new Node<>(item);
    }
    size++;
}
```

The append method is used to add an element to the end of the linked list.

If the list is empty, it creates a new node and sets it as the first node.

Otherwise, it finds the last node in the list and links the new node to the next reference of the last node.

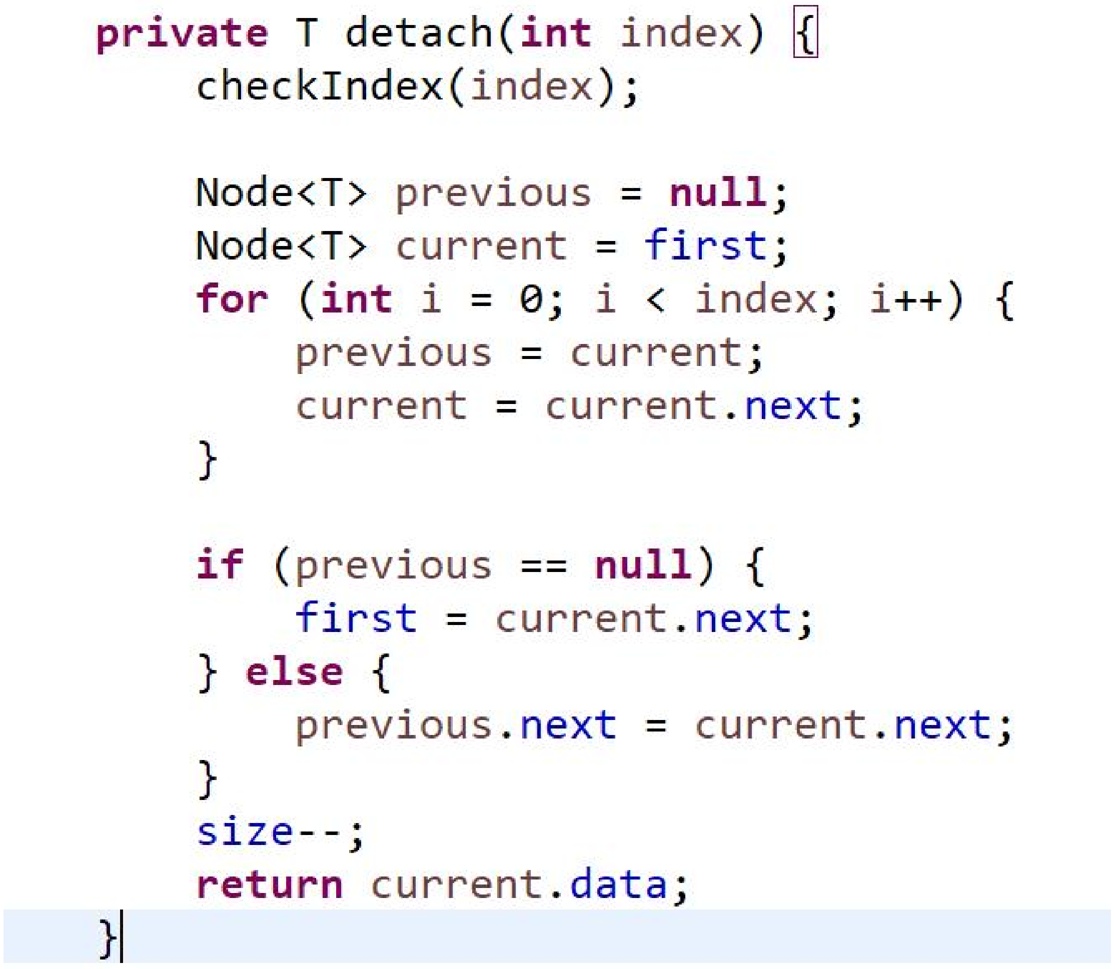
Finally, it increments the size of the list.



```java
private void checkIndex(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException("Index: " + index + "
    }
}
```

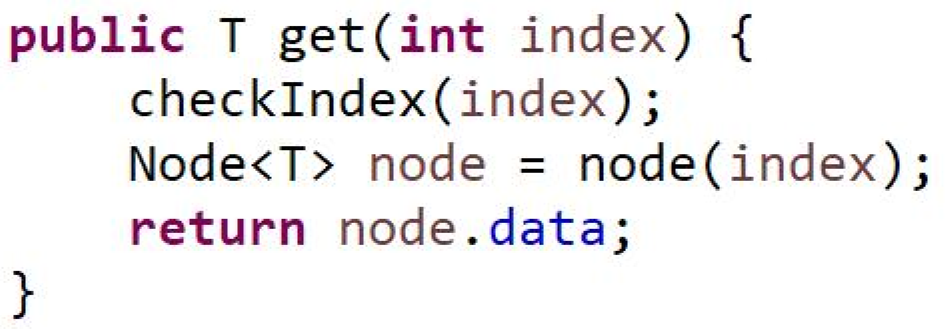
The checkIndex method is used to validate if the index is within bounds.

If the index is less than 0 or greater than or equal to the size of the list, it throws an IndexOutOfBoundsException with an appropriate message.



```java
private T detach(int index) {
    checkIndex(index);

    Node<T> previous = null;
    Node<T> current = first;
    for (int i = 0; i < index; i++) {
        previous = current;
        current = current.next;
    }

    if (previous == null) {
        first = current.next;
    } else {
        previous.next = current.next;
    }
    size--;
    return current.data;
}
```

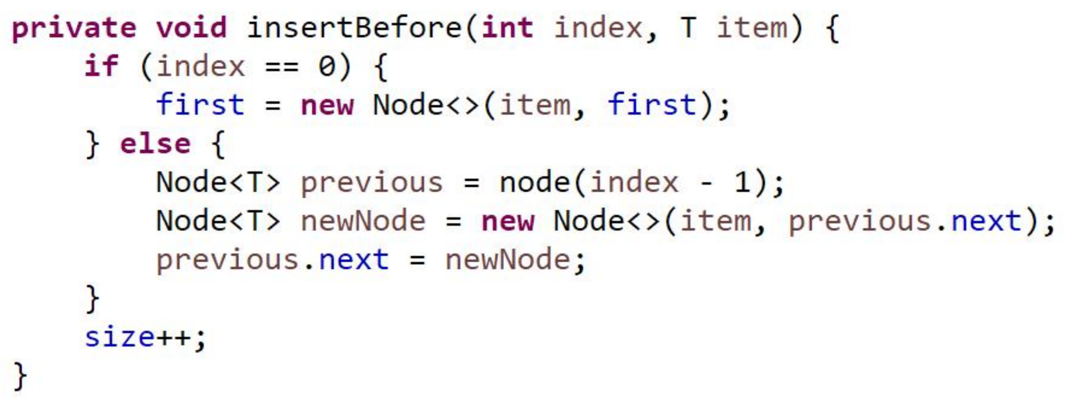
The detach method is used to detach and return the element at the specified index in the linked list.
It first checks the validity of the index by calling the checkIndex method.
It iterates through the list using previous and current references until it finds the node at the specified index.
If the node is the first node, it updates the first reference to the next node.
Otherwise, it updates the next reference of the previous node to skip over the current node.
Finally, it decrements the size of the list and returns the data of the detached node.



```java
public T get(int index) {
    checkIndex(index);
    Node<T> node = node(index);
    return node.data;
}
```

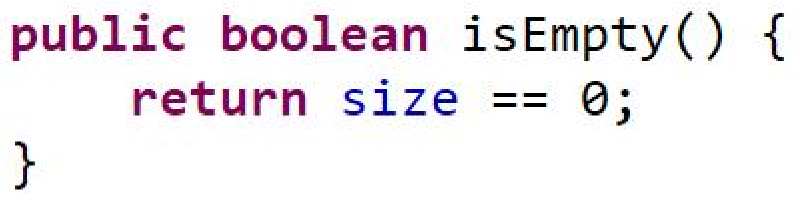
The get method is used to retrieve the element at the specified index in the linked list.
It first checks the validity of the index by calling the checkIndex method.

Then, it retrieves the node at the specified index by calling the node method and returns the data of that node.



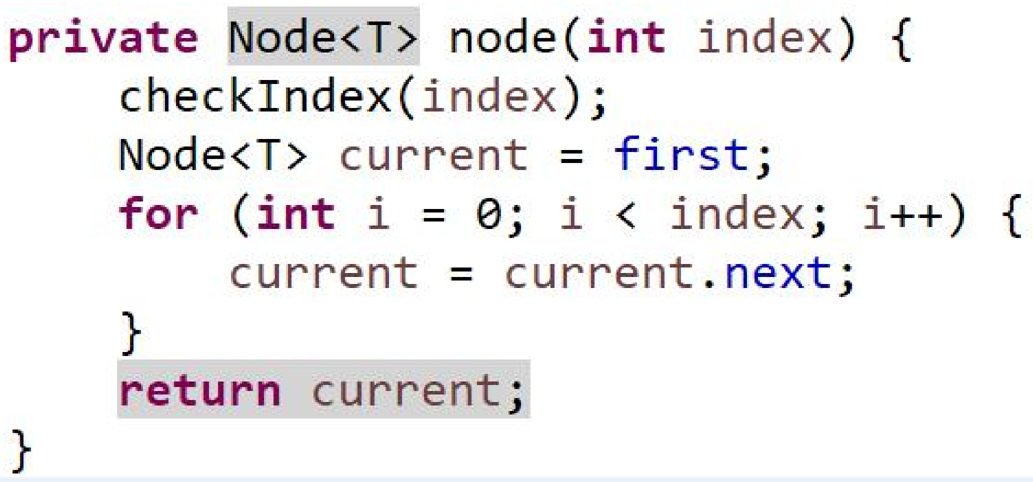
```java
private void insertBefore(int index, T item) {
    if (index == 0) {
        first = new Node<>(item, first);
    } else {
        Node<T> previous = node(index - 1);
        Node<T> newNode = new Node<>(item, previous.next);
        previous.next = newNode;
    }
    size++;
}
```

The insertBefore method is used to insert an element before the specified index in the linked list.
If the index is 0, it creates a new node and sets it as the first node.
Otherwise, it first finds the node before the specified index.
Then, it creates a new node and sets its next reference to the next reference of the previous node.
Finally, it updates the next reference of the previous node to the new node and increments the size of the list.



```java
public boolean isEmpty() {
    return size == 0;
}
```

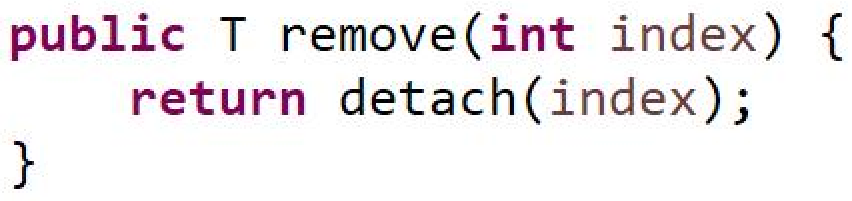
The isEmpty method is used to check if the linked list is empty.
It returns true if the size of the list is 0, otherwise, it returns false.



```java
private Node<T> node(int index) {
    checkIndex(index);
    Node<T> current = first;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current;
}
```

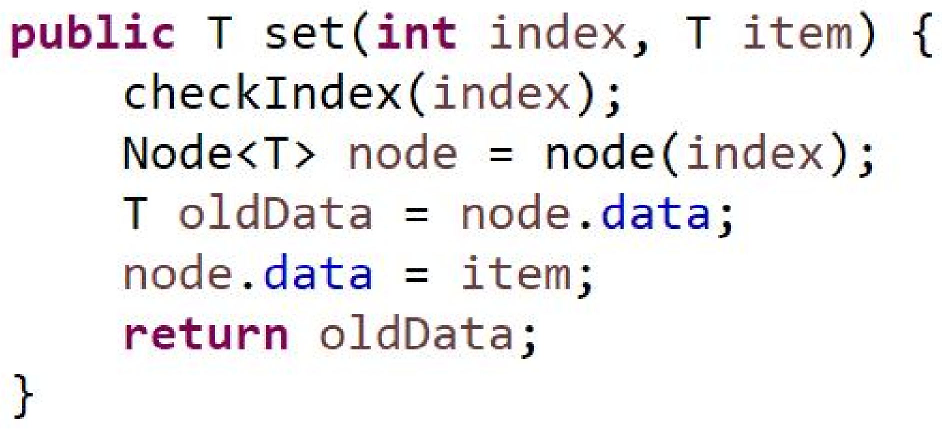
The node method is used to retrieve the node at the specified index in the linked list.
It first checks the validity of the index by calling the checkIndex method.
Then, it iterates through the list using the current reference until it reaches the node at the

specified index.
Finally, it returns the node at that index.



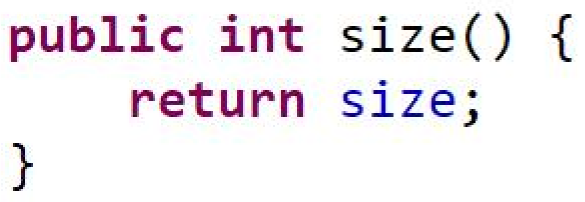
```java
public T remove(int index) {
    return detach(index);
}
```

The remove method is used to remove and return the element at the specified index in the linked list.
It calls the detach method to detach the node at the specified index and returns the data of that node.



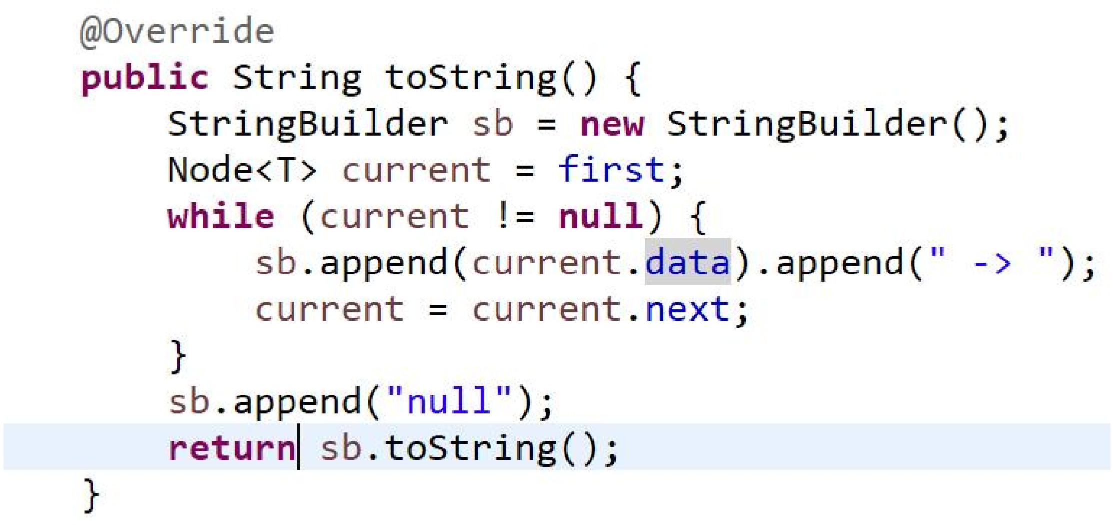
```java
public T set(int index, T item) {
    checkIndex(index);
    Node<T> node = node(index);
    T oldData = node.data;
    node.data = item;
    return oldData;
}
```

The set method is used to replace the element at the specified index in the linked list with a new element and returns the replaced old element.
It first checks the validity of the index by calling the checkIndex method.
Then, it retrieves the node at the specified index by calling the node method.
It stores the data of the node in oldData, updates the data of the node with the new element, and returns oldData.



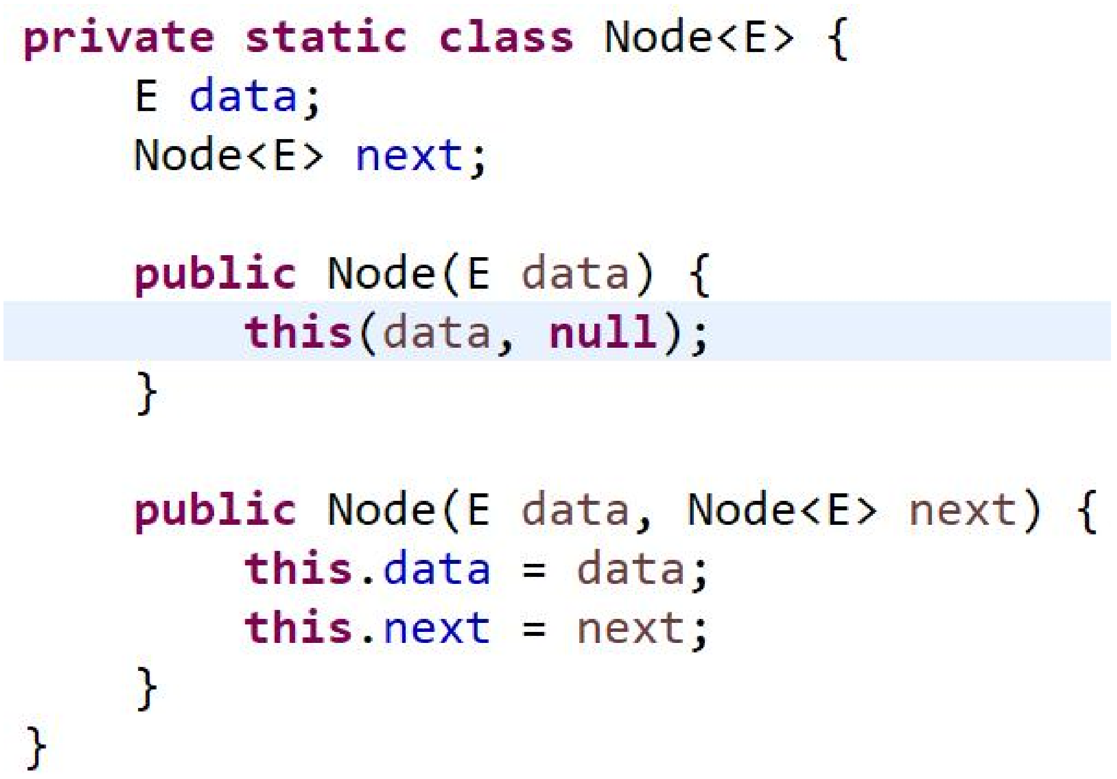
```java
public int size() {
    return size;
}
```

The size method is used to retrieve the size of the linked list, i.e., the number of elements.
It returns the size of the list.



```
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        Node<T> current = first;
        while (current != null) {
            sb.append(current.data).append(" -> ");
            current = current.next;
        }
        sb.append("null");
        return sb.toString();
    }
```

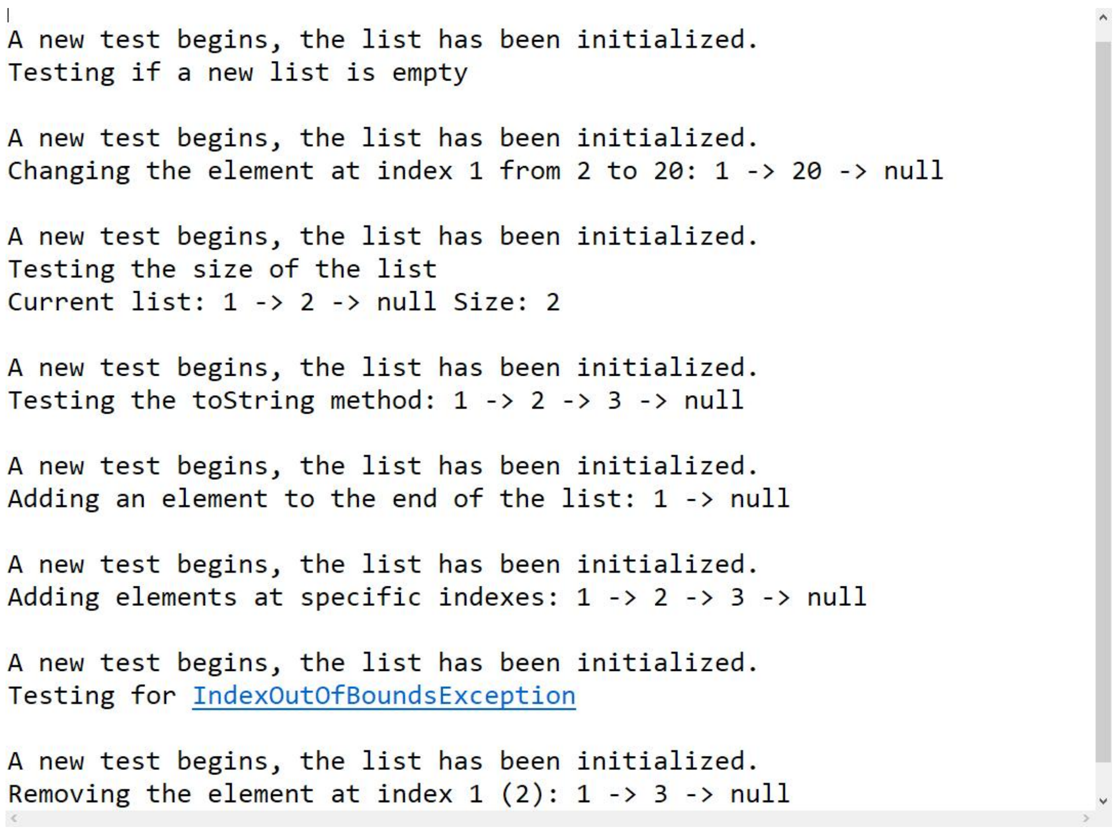
The toString method is used to generate the string representation of the linked list.
It uses a StringBuilder to build the string.
It iterates through the nodes in the list, adding the data of each node to the string.
Finally, it appends "null" to indicate the end of the list and returns the generated string.



```
private static class Node<E> {
    E data;
    Node<E> next;

    public Node(E data) {
        this(data, null);
    }

    public Node(E data, Node<E> next) {
        this.data = data;
        this.next = next;
    }
}
```

This portion of code defines the Node class, representing a node in the linked list.
The Node class is a nested class and can only be accessed internally by the SinglyLinkedList class.
Each node contains a data item and a reference to the next node.
The Node class also provides two constructors for creating node objects and initializing the node's data and the next reference.

The running result



```
A new test begins, the list has been initialized.
Testing if a new list is empty

A new test begins, the list has been initialized.
Changing the element at index 1 from 2 to 20: 1 -> 20 -> null

A new test begins, the list has been initialized.
Testing the size of the list
Current list: 1 -> 2 -> null Size: 2

A new test begins, the list has been initialized.
Testing the toString method: 1 -> 2 -> 3 -> null

A new test begins, the list has been initialized.
Adding an element to the end of the list: 1 -> null

A new test begins, the list has been initialized.
Adding elements at specific indexes: 1 -> 2 -> 3 -> null

A new test begins, the list has been initialized.
Testing for IndexOutOfBoundsException

A new test begins, the list has been initialized.
Removing the element at index 1 (2): 1 -> 3 -> null
```

The unit code as follows:

```
import utils.SinglyLinkedList;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class SinglyLinkedJUnitTest {
    private SinglyLinkedList<Integer> list;

    @BeforeEach
    void setUp() {
        list = new SinglyLinkedList<>();
        System.out.println("\nA new test begins, the list has been initialized.");
    }

    @Test
    void testIsEmptyOnNewList() {
        System.out.println("Testing if a new list is empty");
        assertTrue(list.isEmpty());
    }

    @Test
    void testAddToEnd() {
        list.add(1);
        System.out.println("Adding an element to the end of the list: " + list.toString());
        assertFalse(list.isEmpty());
        assertEquals(1, list.size());
        assertEquals(Integer.valueOf(1), list.get(0));
    }

    @Test
    void testAddAtIndex() {
        list.add(0, 1);
        list.add(1, 3);
        list.add(1, 2);
        System.out.println("Adding elements at specific indexes: " + list.toString());
        assertEquals(3, list.size());
        assertEquals(Integer.valueOf(1), list.get(0));
        assertEquals(Integer.valueOf(2), list.get(1));
        assertEquals(Integer.valueOf(3), list.get(2));
    }

    @Test
    void testRemove() {
        list.add(1);
        list.add(2);
        list.add(3);
        Integer removed = list.remove(1);
        System.out.println("Removing the element at index 1 (" + removed + "): " +
list.toString());
        assertEquals(2, list.size());
        assertEquals(Integer.valueOf(1), list.get(0));
        assertEquals(Integer.valueOf(3), list.get(1));
    }

    @Test
    void testSet() {
        list.add(1);
        list.add(2);
        Integer old = list.set(1, 20);
        System.out.println("Changing the element at index 1 from " + old + " to 20: " +
list.toString());
        assertEquals(Integer.valueOf(20), list.get(1));
    }

    @Test
    void testSize() {
        System.out.println("Testing the size of the list");
        assertEquals(0, list.size());
        list.add(1);
        assertEquals(1, list.size());
        list.add(2);
        assertEquals(2, list.size());
        System.out.println("Current list: " + list.toString() + " Size: " + list.size());
    }

    @Test
    void testToString() {
        list.add(1);
        list.add(2);
        list.add(3);
        System.out.println("Testing the toString method: " + list.toString());
        String expected = "1 -> 2 -> 3 -> null";
        assertEquals(expected, list.toString());
    }

    @Test
    void testCheckIndexOutOfBoundsException() {
        System.out.println("Testing for IndexOutOfBoundsException");
        assertThrows(IndexOutOfBoundsException.class, () -> list.get(0)); // When the list is
empty
        list.add(1);
        assertThrows(IndexOutOfBoundsException.class, () -> list.get(-1)); // Negative index
        assertThrows(IndexOutOfBoundsException.class, () -> list.get(1)); // Index greater than
list size
    }
}
```