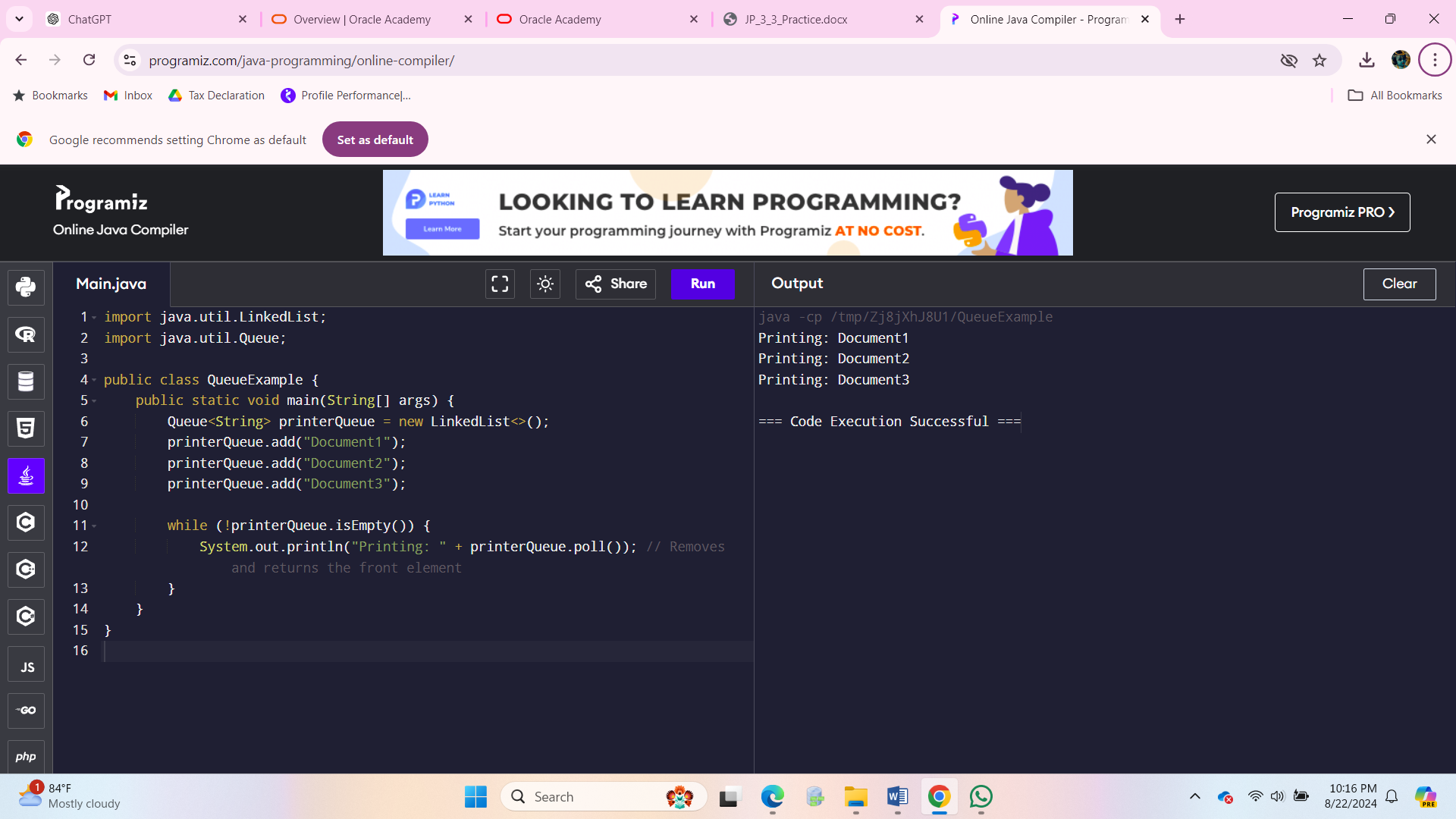
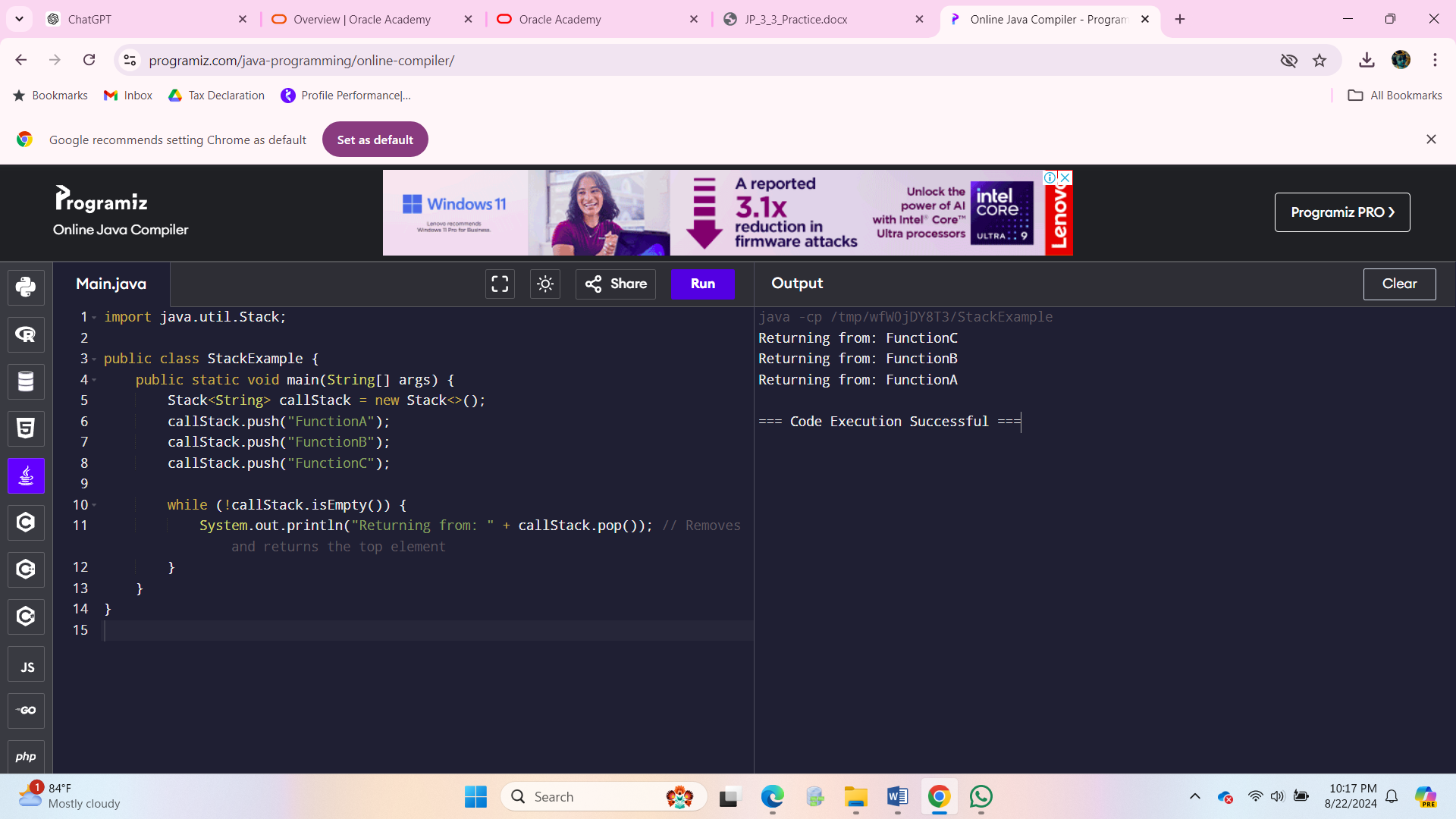
**Queue**

* **Definition**: A queue is a First-In-First-Out (FIFO) data structure. The first element added to the queue will be the first one to be removed. Think of a queue as a line of people waiting for service: the person who arrives first is served first.
* **Operations**:
  + enqueue: Add an element to the end of the queue.
  + dequeue: Remove an element from the front of the queue.
  + peek (or front): View the element at the front of the queue without removing it.
  + isEmpty: Check if the queue is empty.
* **Example**: A queue can be used to manage tasks in a printer. Tasks are printed in the order they are submitted.



**Stack**

* **Definition**: A stack is a Last-In-First-Out (LIFO) data structure. The last element added to the stack is the first one to be removed. Think of a stack as a stack of plates: the last plate placed on top is the first one to be removed.
* **Operations**:
  + push: Add an element to the top of the stack.
  + pop: Remove an element from the top of the stack.
  + peek (or top): View the element at the top of the stack without removing it.
  + isEmpty: Check if the stack is empty.
* **Example**: A stack can be used to manage function calls in a program (the call stack).



### GenericStackException Class

java

Copy code

// File: GenericStackException.java

public class GenericStackException extends RuntimeException {

public GenericStackException(String message) {

super(message);

}

}

### 2. GenericStack Class

java

Copy code

// File: GenericStack.java

import java.util.ArrayList;

public class GenericStack<T> {

private ArrayList<T> items;

private int top;

// Constructor

public GenericStack() {

this.items = new ArrayList<>();

this.top = 0;

}

// Private method to check if stack is empty

private boolean isEmpty() {

return top == 0;

}

// Add an item to the stack

public void push(T item) {

items.add(item);

top++;

}

// Remove an item from the stack

public T pop() {

if (isEmpty()) {

throw new GenericStackException("Underflow Error");

}

T item = items.get(top - 1);

items.remove(top - 1);

top--;

return item;

}

}

### 3. StackDriver Class

java

Copy code

// File: StackDriver.java

public class StackDriver {

public static void main(String[] args) {

GenericStack<Integer> stack = new GenericStack<>();

// Push elements to the stack

stack.push(1);

stack.push(2);

stack.push(3);

stack.push(4);

// Attempt to pop 5 times

for (int i = 0; i < 5; i++) {

try {

System.out.println("Popped: " + stack.pop());

} catch (GenericStackException e) {

System.out.println(e.getMessage());

}

}

}

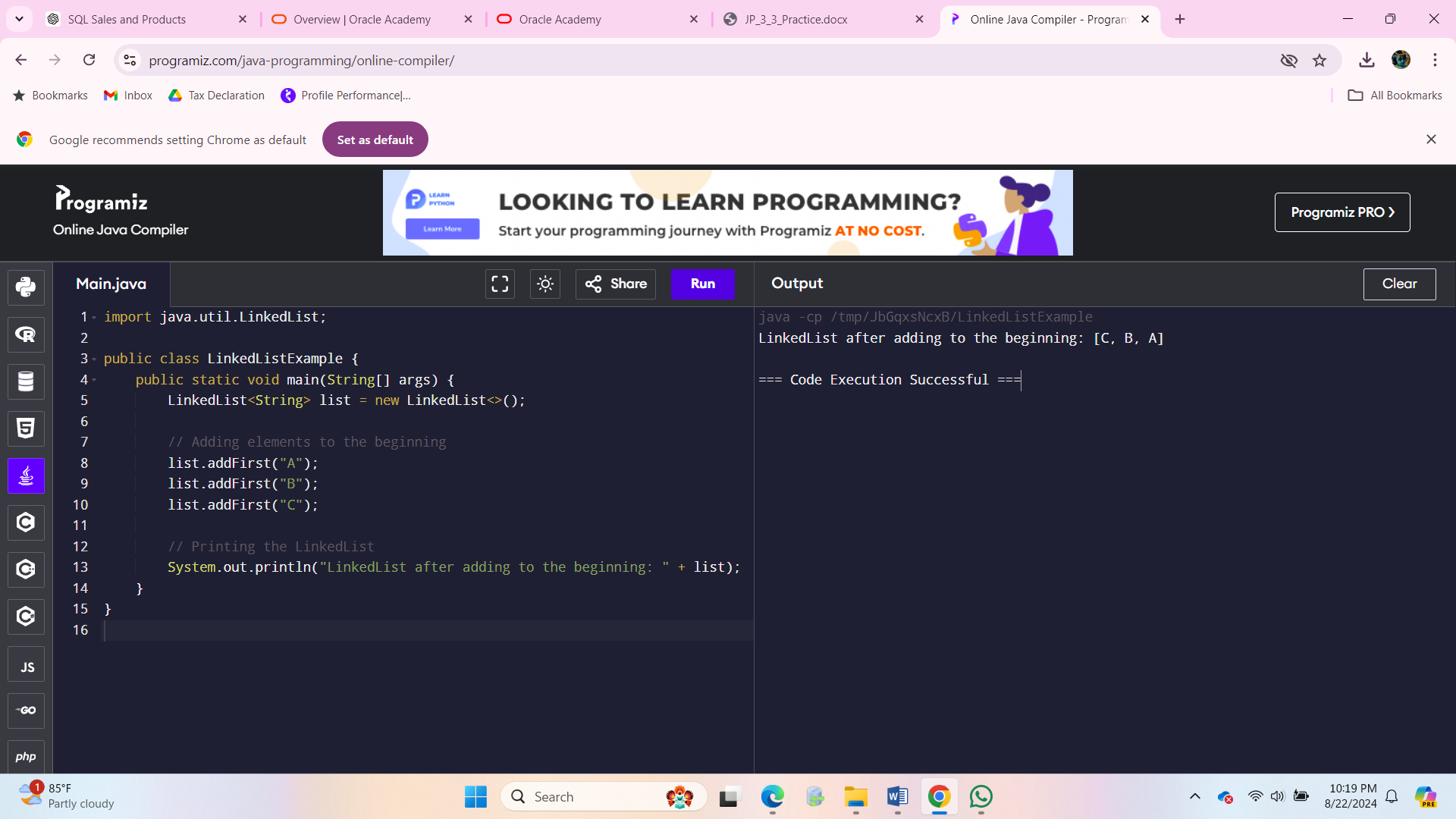
}

Yes, it is possible to add nodes to both the beginning and the end of a LinkedList in Java. The LinkedList class in Java, which is part of the Java Collections Framework, provides methods for adding nodes (elements) to both positions.

Here's how you can do it:

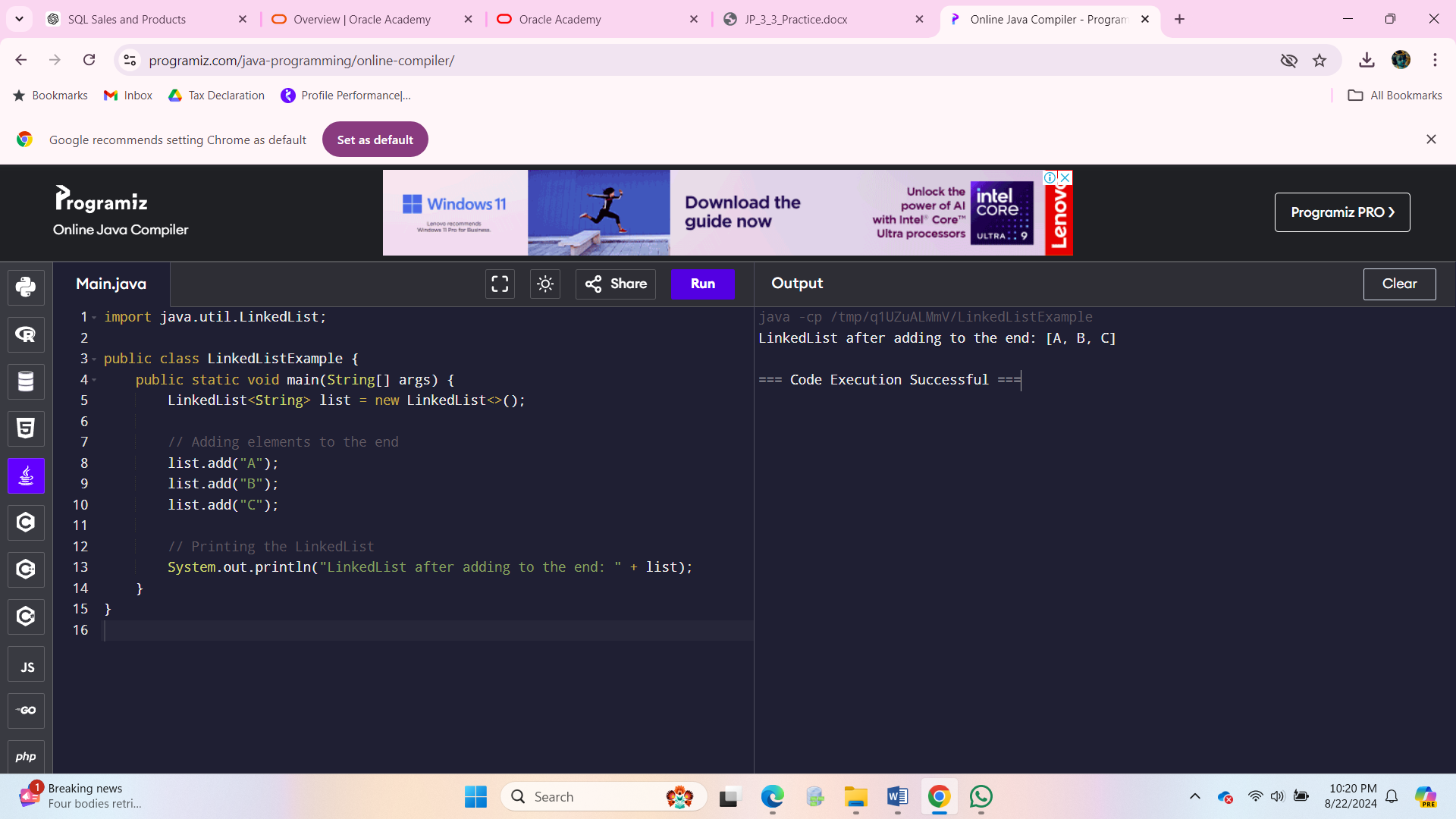
### Adding Nodes to the Beginning of a LinkedList

To add a node to the beginning of a LinkedList, you use the addFirst method. This method inserts the specified element at the beginning of the list.



### Adding Nodes to the End of a LinkedList

To add a node to the end of a LinkedList, you use the add method. This method appends the specified element to the end of the list.



The Comparable interface in Java is used to define a natural ordering for objects of a class. When a class implements the Comparable interface, it provides a way to compare instances of that class to one another, which is essential for sorting and ordering operations.

**Purpose of Implementing the Comparable Interface**

1. **Define Natural Ordering**: By implementing Comparable, a class can define its natural ordering. This is useful for sorting collections of objects, such as arrays or lists, where a consistent and predictable order is needed.
2. **Enable Sorting**: Classes that implement Comparable can be used with sorting algorithms that rely on comparison, such as those provided by the Collections.sort() method or the Arrays.sort() method.
3. **Facilitate Data Structures**: Some data structures, like TreeSet and TreeMap, require elements to be ordered. Implementing Comparable allows objects of the class to be stored and managed in such ordered collections.

**How to Implement the Comparable Interface**

1. **Implement the compareTo Method**: The Comparable interface has a single method, compareTo(T o), which must be overridden. This method compares the current object with another object of the same type and returns:
   * A negative integer if the current object is less than the other object.
   * Zero if the current object is equal to the other object.
   * A positive integer if the current object is greater than the other object.
2. **Ensure Consistency**: The compareTo method should provide a consistent ordering. If a.compareTo(b) returns a positive value, then b.compareTo(a) should return a negative value.

public class Person implements Comparable<Person> {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public int compareTo(Person other) {

return Integer.compare(this.age, other.age);

}

@Override

public String toString() {

return name + " (" + age + ")";

}

public static void main(String[] args) {

List<Person> people = new ArrayList<>();

people.add(new Person("Alice", 30));

people.add(new Person("Bob", 25));

people.add(new Person("Charlie", 35));

Collections.sort(people);

System.out.println("Sorted people by age: " + people);

}

}

To store and manage a collection of courses and their codes, a suitable data structure in Java would be a HashMap. This allows you to store key-value pairs where each course code maps to its corresponding course name, and it provides efficient lookup, insertion, and deletion operations.

### Using a HashMap to Store Courses and Their Codes

Here's how you can implement this:

#### 1. **Create and Populate the** HashMap

