**Exercise 7: Financial Forecasting**

**1. Understanding Recursive Algorithms**

**Concept of Recursion**

* Recursion is a programming technique where a **function calls itself** to solve smaller instances of the same problem.
* It simplifies complex problems by breaking them down into smaller, more manageable subproblems.

**Key components of recursion:**

* **Base case:** The simplest instance of the problem that can be solved directly
* **Recursive case:** The function calls itself with a modified input to progress toward the base case

**How it simplifies financial forecasting:**

* Financial growth often follows recursive patterns (e.g., compound growth)
* Recursion naturally models these repeated calculations
* Cleaner code compared to iterative approaches.

**2. Setup: Future Value Calculation**

**Recursive Future Value Formula**

The future value (FV) of an investment can be calculated recursively:

FV(0) = Present Value (PV) [Base case]

FV(n) = FV(n-1) × (1 + growthRate) [Recursive case]

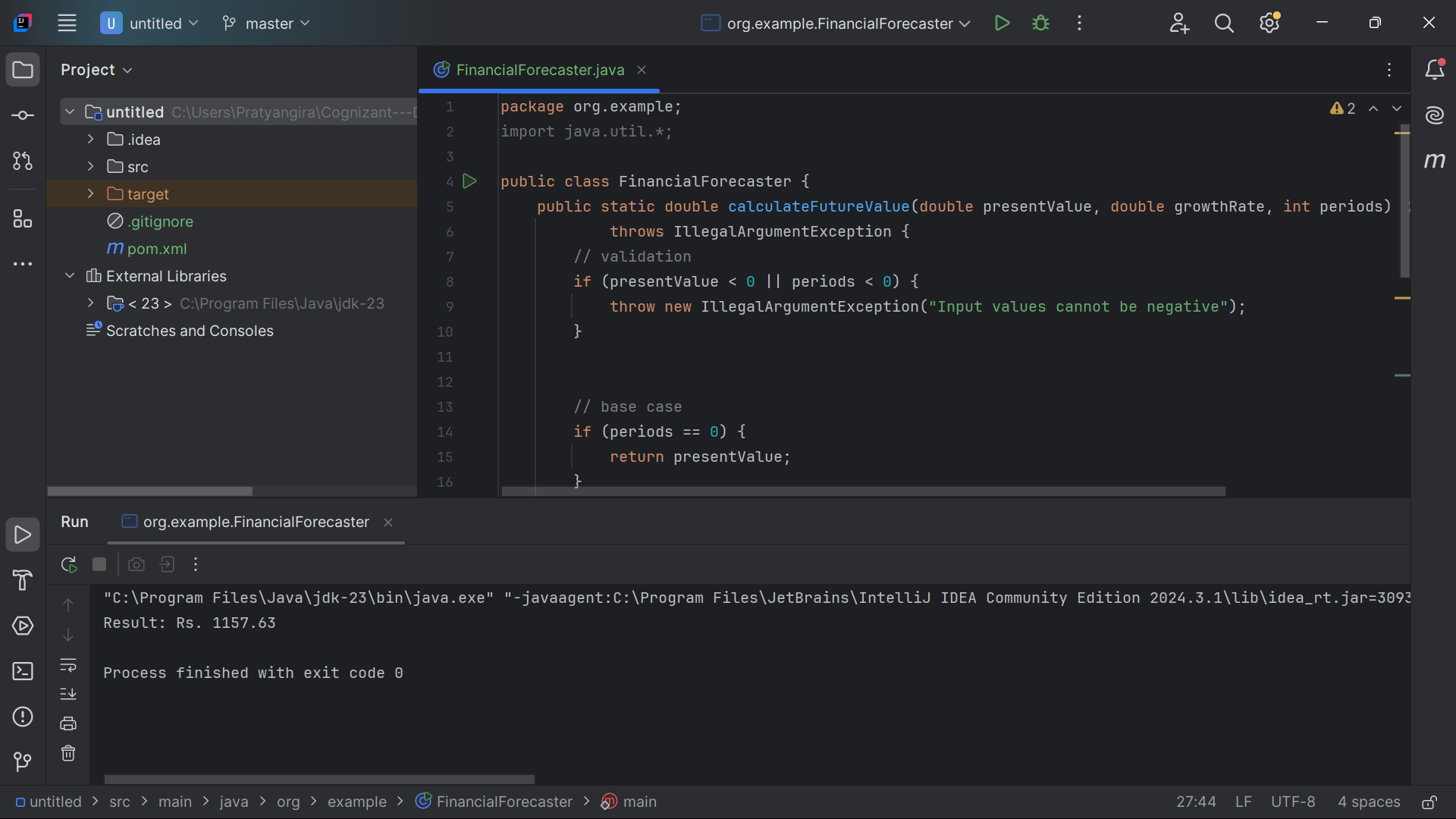
**3. Implementation**

**Java code**

FinancialForecaster.java

package org.example;  
import java.util.\*;  
  
public class FinancialForecaster {  
 public static double calculateFutureValue(double presentValue, double growthRate, int periods)  
 throws IllegalArgumentException {  
 // validation  
 if (presentValue < 0 || periods < 0) {  
 throw new IllegalArgumentException("Input values cannot be negative");  
 }  
  
  
 // base case  
 if (periods == 0) {  
 return presentValue;  
 }  
  
 // recursive calculation  
 double result = *calculateFutureValue*(presentValue, growthRate, periods - 1) \* (1 + growthRate);  
 return result;  
 }  
  
  
 public static void main(String[] args) {  
 try {  
 double memoResult = FinancialForecaster.*calculateFutureValue*(1000, 0.05, 3);  
 System.*out*.printf("Result: Rs. %.2f%n", memoResult);  
 } catch (IllegalArgumentException e) {  
 System.*err*.println("Error: " + e.getMessage());  
 }  
 }  
}

**Result**



**4. Analysis**

**1. Basic Recursive Solution (No Memoization)**

**Time Complexity:**

* O(n) – Linear time, where n = number of periods.
* Each recursive call computes FV(n) = FV(n-1) × (1 + r), leading to n nested calls.

**Space Complexity:**

* O(n) – Due to the recursion call stack.

**2. Memoized Recursive Solution (Optimized)**

* The problem with basic recursion is the repeated calculation of the same subproblems (e.g., calculating intermediate periods multiple times).
* This can be solved by caching or storing the results of previously computed periods, allowing them to be reused for the same subproblems.

**Benefits:**

* Reduces time complexity from O(n) to O(1) for subsequent calls
* Each period is calculated only once

**Time Complexity:**

* First call: O(n) – Computes and caches all values from FV(0) to FV(n).
* Subsequent calls: O(1) – Retrieves precomputed results from cache.

**Space Complexity:**

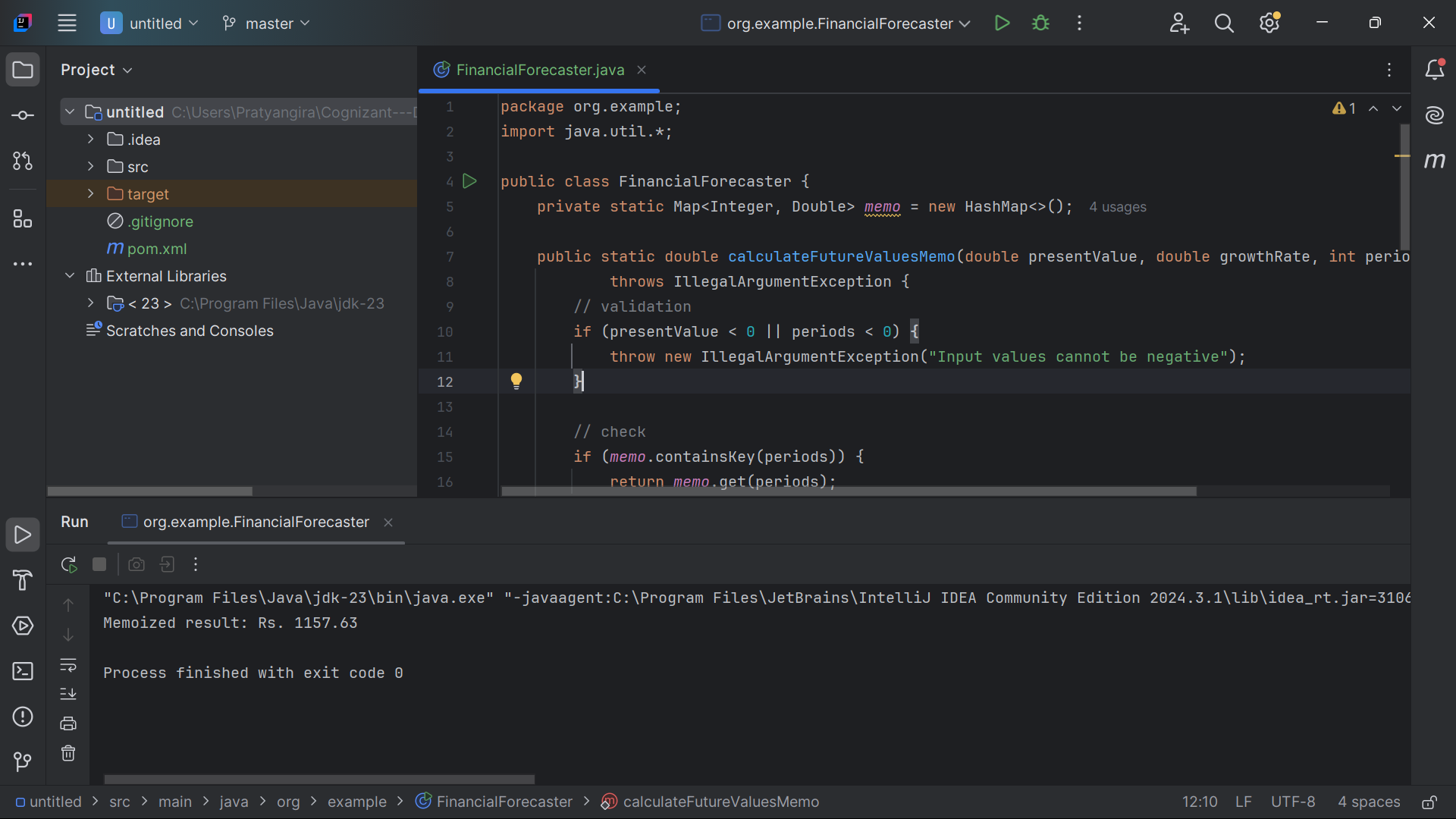
* O(n) – Stores n results in the memoization table.

**Java code:**

FinancialForecaster.java

package org.example;  
import java.util.\*;  
  
public class FinancialForecaster {  
 private static Map<Integer, Double> *memo* = new HashMap<>();  
  
 public static double calculateFutureValuesMemo(double presentValue, double growthRate, int periods)  
 throws IllegalArgumentException {  
 // validation  
 if (presentValue < 0 || periods < 0) {  
 throw new IllegalArgumentException("Input values cannot be negative");  
 }  
  
 // check  
 if (*memo*.containsKey(periods)) {  
 return *memo*.get(periods);  
 }  
  
 // base case  
 if (periods == 0) {  
 *memo*.put(0, presentValue);  
 return presentValue;  
 }  
  
 // recursive calculation  
 double result = *calculateFutureValuesMemo*(presentValue, growthRate, periods - 1) \* (1 + growthRate);  
 *memo*.put(periods, result);  
 return result;  
 }  
  
  
 public static void main(String[] args) {  
 try {  
 double memoResult = FinancialForecaster.*calculateFutureValuesMemo*(1000, 0.05, 3);  
 System.*out*.printf("Memoized result: Rs. %.2f%n", memoResult);  
 } catch (IllegalArgumentException e) {  
 System.*err*.println("Error: " + e.getMessage());  
 }  
 }  
}

**Result:**



**5. Performance Benchmarking**

When both functions are called in the **same thread** (or different threads, as in the given code):

1. **Memoized version runs faster** after the first call.
2. **Non-memoized version always takes O(n) time**, even for the same inputs.

**Reason:**

* The memoized function **caches results** after the first computation.
* The basic recursive function **repeats all calculations** every time.

**Java code:**

FinancialForecaster.java

package org.example;  
import java.util.\*;  
  
public class FinancialForecaster {  
 private static Map<Integer, Double> *memo* = new HashMap<>();  
  
 public static double calculateFutureValuesMemo(double presentValue, double growthRate, int periods)  
 throws IllegalArgumentException {  
 // validation  
 if (presentValue < 0 || periods < 0) {  
 throw new IllegalArgumentException("Input values cannot be negative");  
 }  
  
 // check  
 if (*memo*.containsKey(periods)) {  
 return *memo*.get(periods);  
 }  
  
 // base case  
 if (periods == 0) {  
 *memo*.put(0, presentValue);  
 return presentValue;  
 }  
  
 // recursive calculation  
 double result = *calculateFutureValuesMemo*(presentValue, growthRate, periods - 1) \* (1 + growthRate);  
 *memo*.put(periods, result);  
 return result;  
 }  
 public static double calculateFutureValue(double presentValue, double growthRate, int periods)  
 throws IllegalArgumentException {  
 // validation  
 if (presentValue < 0 || periods < 0) {  
 throw new IllegalArgumentException("Input values cannot be negative");  
 }  
  
 // base case  
 if (periods == 0) {  
 return presentValue;  
 }  
  
 // Recursive calculation  
 double result = *calculateFutureValue*(presentValue, growthRate, periods - 1) \* (1 + growthRate);  
  
 return result;  
 }  
  
 public static void main(String[] args) {  
 try {  
 Thread memoizedThread = new Thread(() -> {  
 double memoResult = FinancialForecaster.*calculateFutureValuesMemo*(1000, 0.05, 3);  
 System.*out*.printf("Memoized result: Rs. %.2f%n", memoResult);  
 });  
  
 Thread nonMemoizedThread = new Thread(() -> {  
 double normalResult = FinancialForecaster.*calculateFutureValue*(1000, 0.05, 3);  
 System.*out*.printf("Non-memoized result: Rs. %.2f%n", normalResult);  
 });  
  
 memoizedThread.start();  
 nonMemoizedThread.start();  
  
 } catch (IllegalArgumentException e) {  
 System.*err*.println("Error: " + e.getMessage());  
 }  
 }  
}

**Result:**

