**Exercise 7: Financial Forecasting**

Data structures and Algorithms:

**Step 1: Understanding Recursive Algorithms**

**Recursion** is a programming technique where a method calls itself to solve a smaller version of the same problem.

It's like saying:

"To solve the big problem, solve the small part first, then build up."

In real life:

* To find the value after 5 years, first find the value after 4 years, then apply 1 more year’s growth.
* This continues until you reach the starting point (base case), like year 0.

**Why recursion helps?**

* Makes code shorter and easier to understand in problems that involve repetitive patterns like compound growth, trees, or mathematical sequences.

**Step 2: Setup**

To begin, we define a method that uses recursion to calculate the **future value** of money:

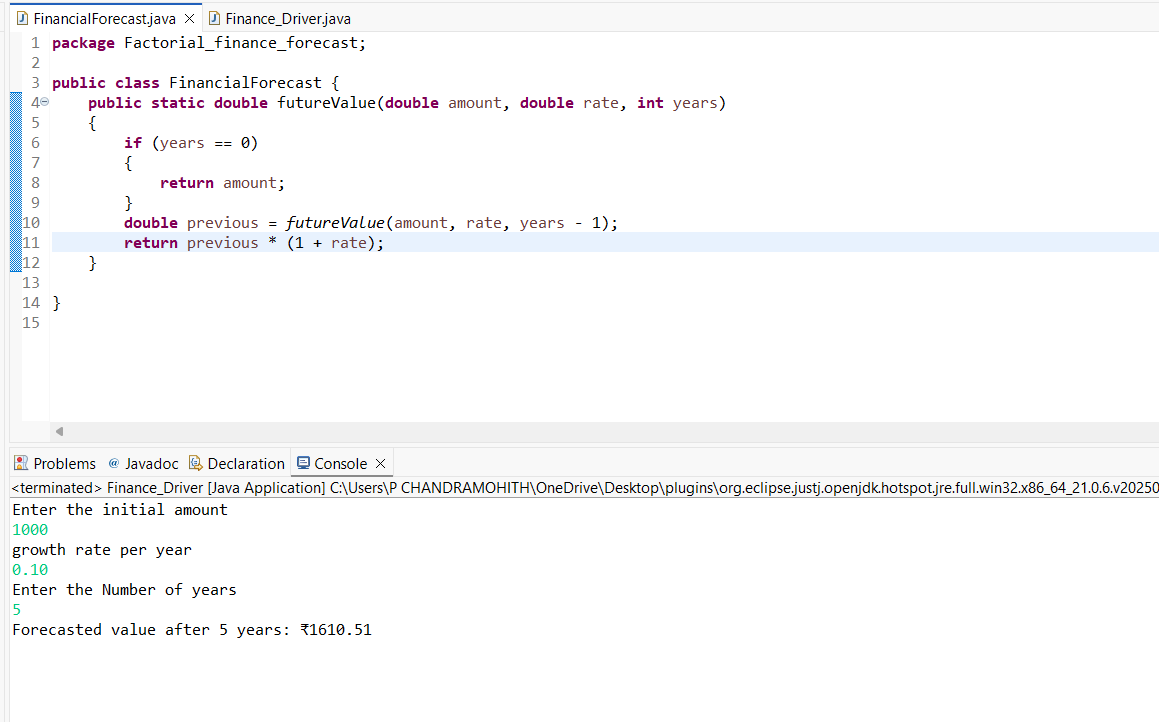
**public static double futureValue(double amount, double rate, int years)**

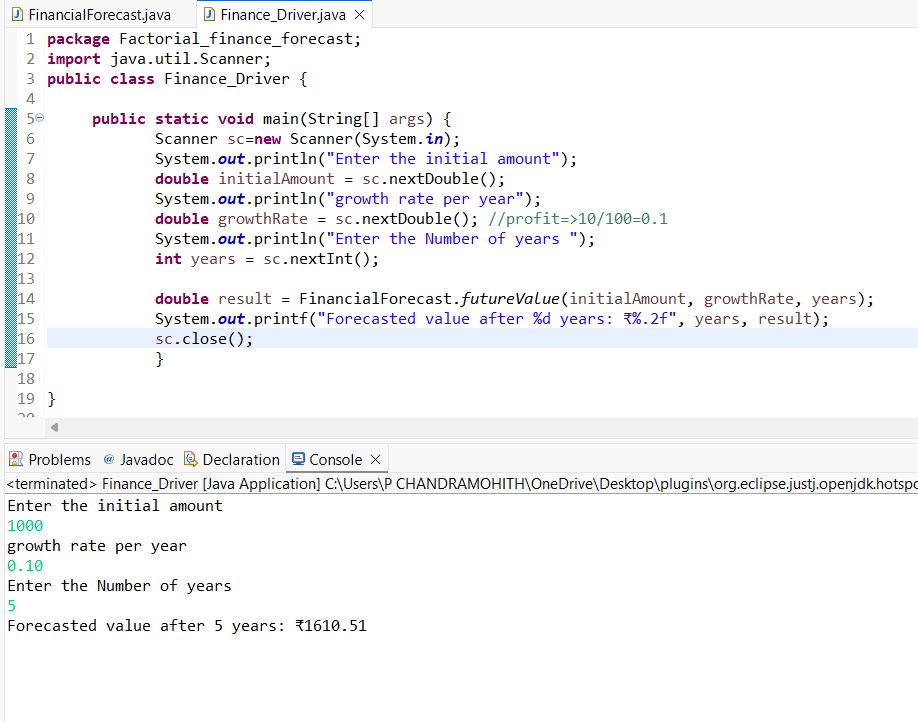
Where:

* amount = the initial investment (e.g., ₹1000)
* rate = annual growth rate (e.g., 10% → 0.10)
* years = number of years in the future

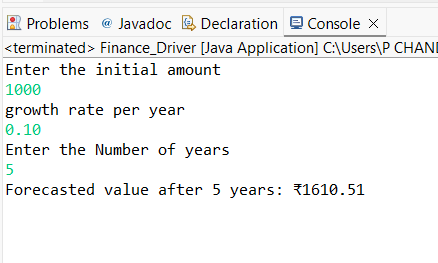
**Step 3: Implementation**

**FinanceForecast.java**

****

**Finance\_Driver.java**

**OUTPUT:**

****

**Step 4: Analysis**

**Time Complexity Analysis**

The recursive function we wrote follows a simple pattern — it reduces the years value by 1 in every call until it reaches 0. This means:

* For n years, it makes n recursive calls.
* So, the time complexity is O(n) (linear time).

This is quite efficient for small to moderately large values (like 10, 20, or 50 years), and the code remains clean and easy to read**.**

**Limitations of Recursion**

While recursion looks elegant, it also comes with some caveats:

* Each recursive call consumes memory on the call stack.
* If you call the method for very large values (e.g., 10000 years), it may lead to stack overflow (memory exhaustion due to too many function calls).

**How to Optimize the Recursive Solution**

Since this specific recursion does not repeat calculations (no overlapping subproblems like in Fibonacci), we don't need to use memoization here.

However, there are still a couple of smart ways to optimize:

1. Convert to Iterative Version  
   If you're working in performance-critical environments, recursion can be converted into a loop to save stack memory:

**for (int i = 0; i < years; i++) {**

**amount \*= (1 + rate);**

**}**

**return amount;**

1. Tail Recursion (if supported by compiler)  
   Tail recursion is a form of recursion where the recursive call is the last statement. Some languages optimize this internally, but Java doesn’t do it natively. So in Java, it’s better to switch to iteration if needed.

**Why Recursion Works Well Here**

Even with its limits, recursion is a great fit here because:

* The problem is naturally repetitive and builds year by year.
* It allows us to express the logic in fewer lines.
* It helps clarify the mathematical relationship between each year’s value.

**Final Thought:**

In conclusion, recursion gives us a neat and logical approach to predict financial growth. For a simple forecasting tool with predictable depth (like 5–20 years), recursion is more than sufficient. For real-time or large-scale financial applications, an iterative or formula-based approach is preferred to improve performance and scalability.