**Task 1 –**

**Algorithm**1. Define a function called greetUser that takes a string parameter (name).

2. Inside the function, print a greeting message with the name.

3. In the main method, call greetUser three times using different names.

**Psudocode**

START

DEFINE function greetUser(String name)

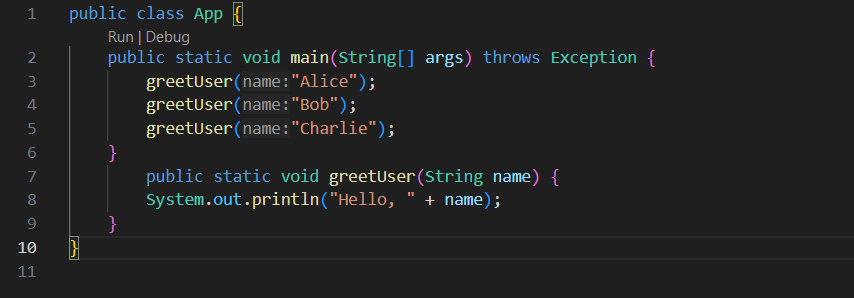
PRINT "Hello, " + name

CALL greetUser with "Alice"

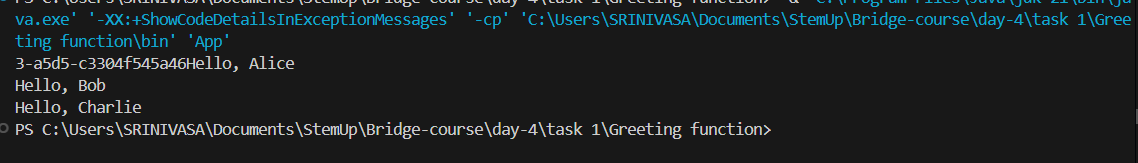
CALL greetUser with "Bob"

CALL greetUser with "Charlie"

END

**Code(java)  
**

**Output (test case 1)**



**Observations**- This problem demonstrates defining and calling a method with a string parameter.

- Calling the method multiple times with different arguments shows function reuse.

- Simple example to understand the syntax and structure of methods in Java.

**Task 1 –**

**Algorithm**1. Define a method named calculateSquare that takes an integer input.

2. Inside the method, return the square of the number.

3. In the main method:

a. Call calculateSquare, store result in a variable, and print it.

b. Also call it directly inside a print statement to show inline use of return.

**Psudocode**

START

DEFINE function calculateSquare(int number)

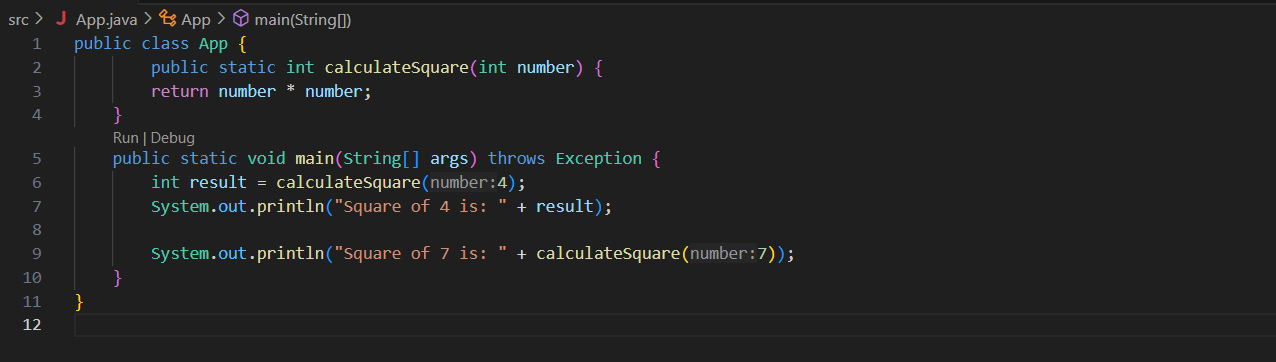
RETURN number \* number

CALL calculateSquare with 4 → store in variable result

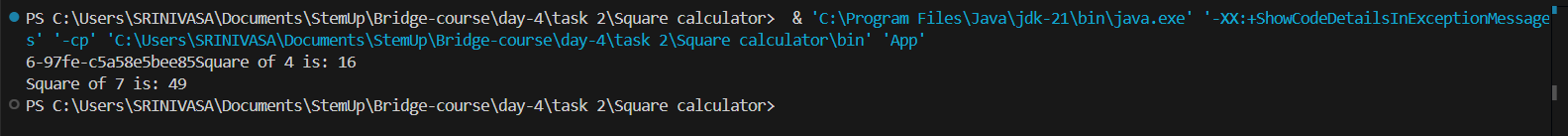
PRINT result

PRINT calculateSquare(7)

END

**Code(javascript)  
**

**Output (test case 1)**



**Observations**

- This problem focuses on using return values from functions.

- Shows how functions can be reused and called in expressions.

- Demonstrates both storing return values in variables and printing them directly.

**Task 3 – Sum of two numbers**

**Algorithm**1. Define a method called addNumbers that accepts two double parameters.

2. Return the sum of the two numbers.

3. In the main method, call addNumbers with two values and print the result.

**Psudocode**

START

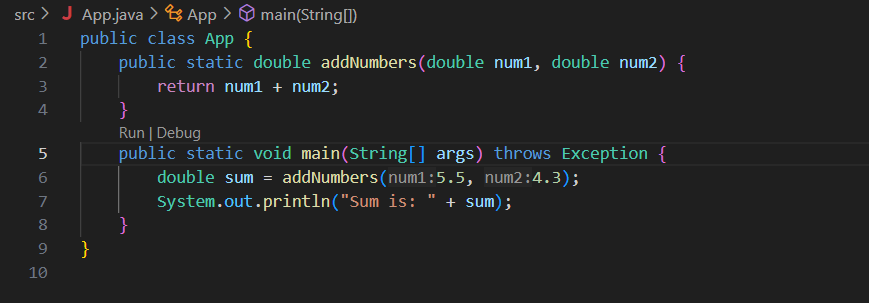
DEFINE function addNumbers(double num1, double num2)

RETURN num1 + num2

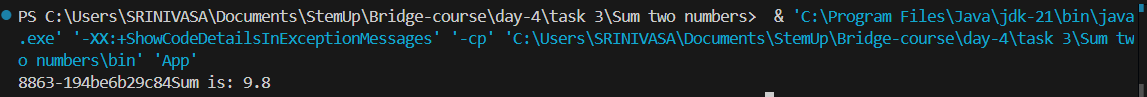
CALL addNumbers with 5.5 and 4.3

PRINT result

END

**Code(javascript)  
**

**Output (test case 1)**



**Output (test case 2)**

**Output (test case 3)**

**Observations**

- This exercise demonstrates the use of methods with multiple parameters.

- Shows how to perform arithmetic with double types and return the result.

- Encourages modular code design through reusable methods.

**Task 4 – Temperature converter**

**Algorithm**1. Define a method called celsiusToFahrenheit that accepts a double and converts it using the formula: (celsius \* 9/5) + 32.

2. Define another method called fahrenheitToCelsius that accepts a double and converts it using the formula: (fahrenheit - 32) \* 5/9.

3. In the main method, test both methods with sample values and print the results.

**Psudocode**

START

DEFINE function celsiusToFahrenheit(double celsius)

RETURN (celsius \* 9 / 5) + 32

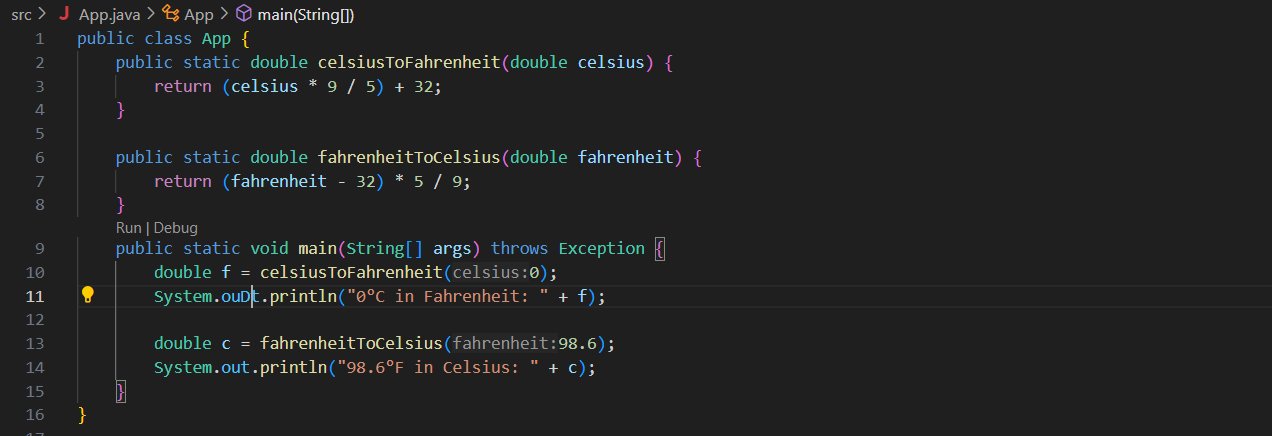
DEFINE function fahrenheitToCelsius(double fahrenheit)

RETURN (fahrenheit - 32) \* 5 / 9

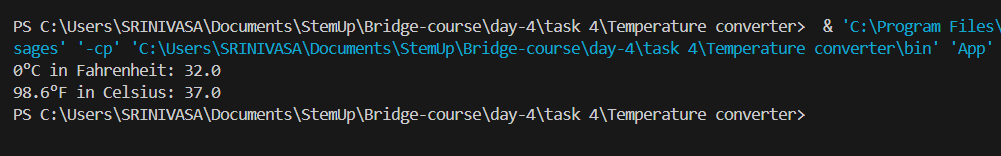
CALL celsiusToFahrenheit with 0 → print result

CALL fahrenheitToCelsius with 98.6 → print result

END

**Code(javascript)  
**

**Output (test case 1)**



**Observations**- Demonstrates method creation with mathematical formulas.

- Shows how to convert temperature between Celsius and Fahrenheit.

- Reinforces accurate use of return values and double precision in Java.

**Task 6 –**

**Algorithm**1. Declare a static global variable named globalMessage.

2. Define a static method displayMessages that:

- Declares a local variable localMessage.

- Prints globalMessage using System.out.println().

3. In the main method:

- Call displayMessages.

- Try to access localMessage (observe that it causes a compile-time error).

**Psudocode**

START

DEFINE static global variable globalMessage = "I am global!"

DEFINE method displayMessages()

DECLARE local variable localMessage = "I am local!"

PRINT globalMessage

IN main method

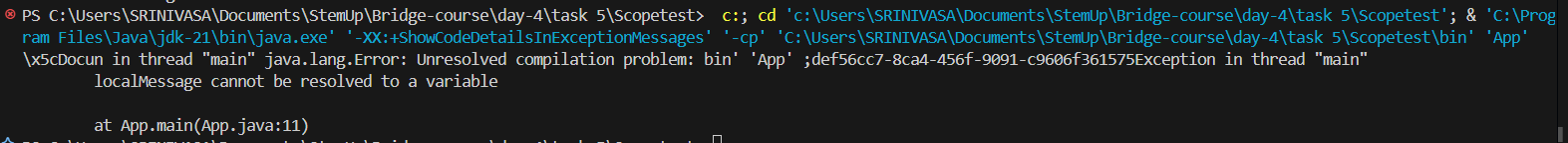
CALL displayMessages()

TRY TO PRINT localMessage → should cause error

END

**Code(java)  
**

**Output (test case 1)**

  
**Observations**

- globalMessage is accessible across all static methods because it has class-level scope.

- localMessage is defined inside displayMessages and cannot be accessed outside it.

- Demonstrates the concept of scope: local vs global.

- Accessing localMessage in main() will result in a compile-time "cannot find symbol" error.

**Task 7 – Price Calculator (Function Composition)**

**Algorithm**1. Define a method calculateDiscount that returns the discounted price.

2. Define a method calculateTax that returns the tax amount on a given amount.

3. Define a method calculateFinalPrice that:

- Uses calculateDiscount to reduce the price.

- Uses calculateTax on the discounted price.

- Returns the final price after adding tax to the discounted price.

4. In the main method, call calculateFinalPrice with sample values and print the result.

**Psudocode**

START

DEFINE function calculateDiscount(originalPrice, discountPercentage)

RETURN originalPrice - (originalPrice \* discountPercentage / 100)

DEFINE function calculateTax(amount, taxRate)

RETURN amount \* taxRate / 100

DEFINE function calculateFinalPrice(itemPrice, discountPerc, taxRate)

discountedPrice = calculateDiscount(itemPrice, discountPerc)

tax = calculateTax(discountedPrice, taxRate)

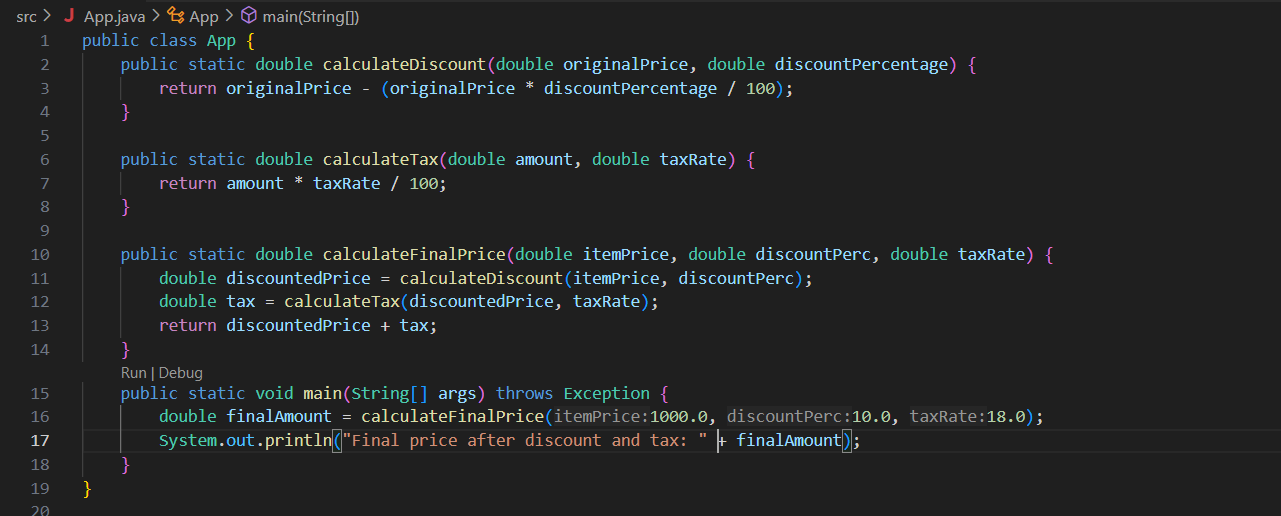
RETURN discountedPrice + tax

IN main method

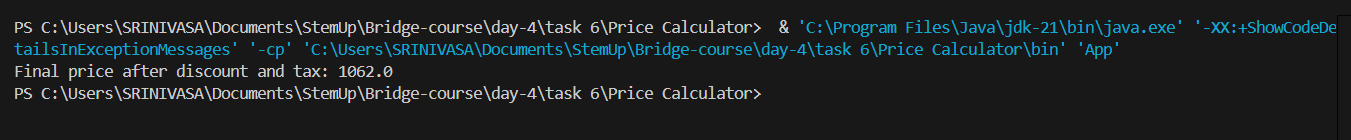
CALL calculateFinalPrice with sample values

PRINT the final result

END

**Code(java)  
**

**Output (test case 1)**



**Observations**- Demonstrates function composition: one method uses results from others.

- Helps separate logic into smaller reusable parts.

- Uses return values effectively to compute step-by-step final price.

- Real-world use case involving tax and discount calculations.

**Task 8 – Customizable Greeting (Overloading)**

**Algorithm**1. Define a method customGreet(String name, String greeting) that prints a custom greeting with the name.

2. Define an overloaded method customGreet(String name) that uses a default greeting like "Hello".

3. Define another overloaded method customGreet() that uses default name and greeting.

4. In the main method, call all three versions to demonstrate method overloading.

**Psudocode**

START

DEFINE method customGreet(name, greeting)

PRINT greeting + ", " + name

DEFINE method customGreet(name)

CALL customGreet(name, "Hello")

DEFINE method customGreet()

CALL customGreet("Guest", "Welcome")

IN main method

CALL customGreet("Alice", "Hi")

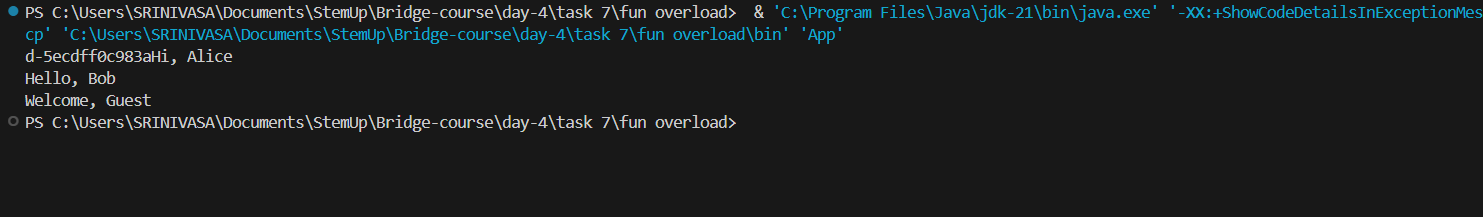
CALL customGreet("Bob")

CALL customGreet()

END

**Code(java)  
**

**Output (test case 1)**



**Observations**- Demonstrates method overloading using different parameter combinations.

- Each version of customGreet handles a different level of input flexibility.

- Useful for simplifying method usage by providing defaults when needed.

- Enhances readability and reuse through function chaining.

**Task 9 – Power Calculator**

**Algorithm**1. Define a method myPower(int base, int exponent).

2. Inside the method:

- Initialize result = 1.

- Use a loop to multiply result by base, exponent number of times.

- Return result.

3. In the main method:

- Call myPower and print the result.

- Also compare it with Math.pow and print that value.

**Psudocode**

START

DEFINE method myPower(base, exponent)

SET result = 1

FOR i = 1 to exponent

result = result \* base

RETURN result

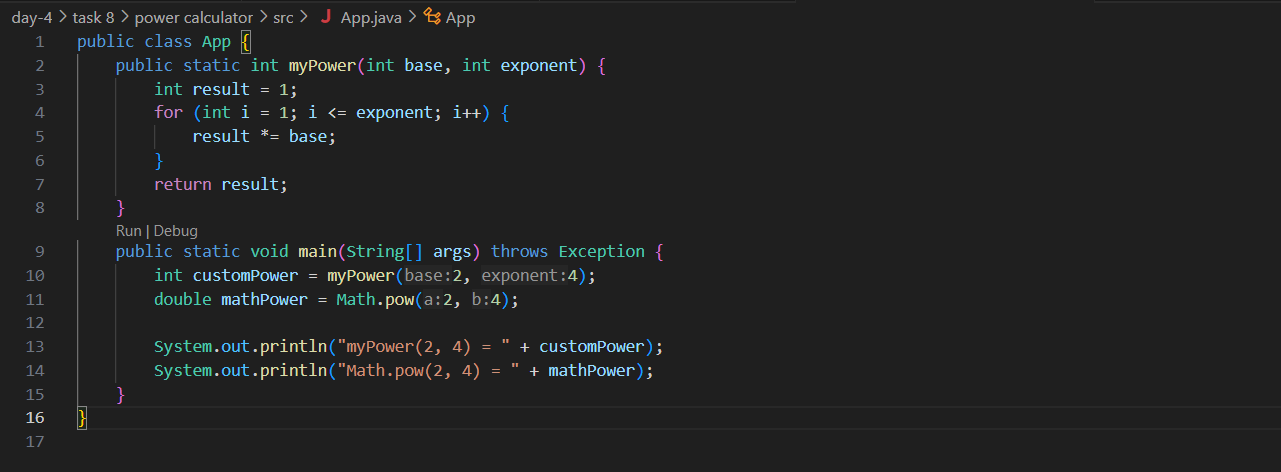
IN main method

CALL myPower(2, 4) → store in result1

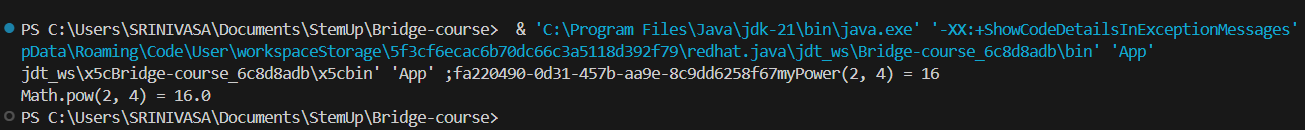
CALL Math.pow(2, 4) → store in result2

PRINT both result1 and result2

END

**Code(java)  
**

**Output (test case 1)**



**Observations**

- Shows how to implement exponentiation manually using a loop.

- Demonstrates method return, iteration, and comparison with built-in Math.pow().

- Useful for understanding how basic operations can be replicated from scratch.

**Task 10 – Trace the flow**

**Algorithm**1. Create method A:

- Calls method B to get a number.

- Uses that number as input to method C.

2. Create method B:

- Returns an integer value.

3. Create method C:

- Takes an integer input and prints it.

4. In the main method:

- Call method A to initiate the flow.**Psudocode**START

DEFINE method B()

RETURN a number (e.g., 42)

DEFINE method C(value)

PRINT "C received: " + value

DEFINE method A()

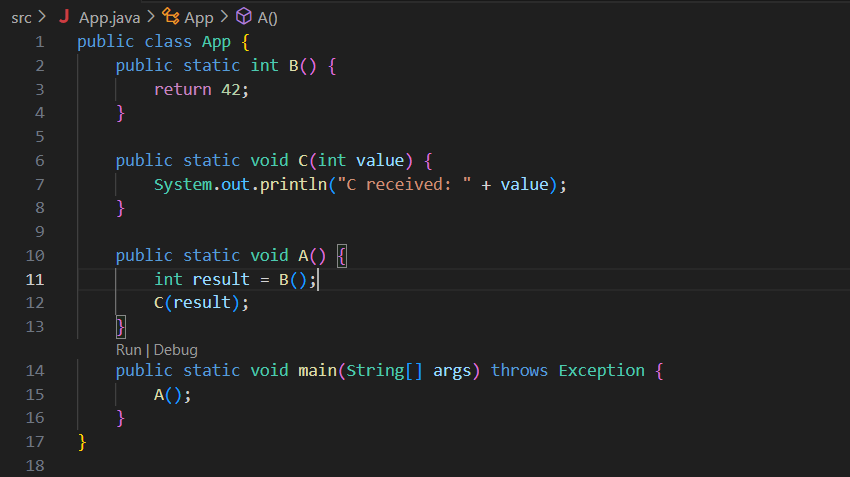
CALL B() → store in result

CALL C(result)

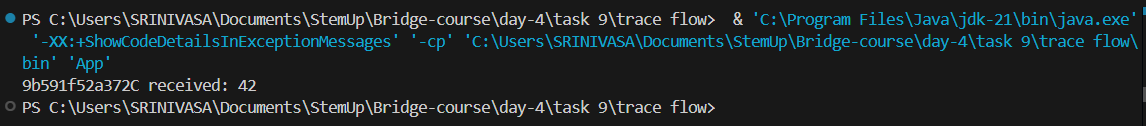
IN main method

CALL A()

END

**Code(java)  
**

**Output (test case 1)**



**Observations**

- Function A starts the flow and depends on B and C.

- Function B returns a value to A.

- Function A passes that value to C.

- C prints the result, completing the flow.

- Helps understand function chaining and execution order in Java.