**WEEK 2**

**PL/SQL programming**

**Exercise 1: control structures**

table setup

begin

execute immediate 'drop table customers';

exception

when others then null;

end;

/

begin

execute immediate 'drop table loans';

exception

when others then null;

end;

/

create table customers (

customer\_id number,

name varchar2(100),

age number,

balance number,

interest\_rate number(5,2),

isvip varchar2(10)

);

create table loans (

loan\_id number,

customer\_id number,

due\_date date

);

insert into customers values (1, 'alice', 65, 12000, 5.0, 'false');

insert into customers values (2, 'bob', 45, 8000, 6.0, 'false');

insert into customers values (3, 'charlie', 70, 9500, 5.5, 'false');

insert into loans values (101, 1, sysdate + 10);

insert into loans values (102, 2, sysdate + 40);

insert into loans values (103, 3, sysdate + 20);

commit;

**scenario 1:**

The bank wants to apply a discount to loan interest rates for customers above 60 years old.

**question:**

write a pl/sql block that loops through all customers, checks their age, and if they are above 60, apply a 1% discount to their current loan interest rates

begin

for customer\_rec in (

select customer\_id from customers where age > 60

) loop

update customers

set interest\_rate = interest\_rate - (interest\_rate \* 0.01)

where customer\_id = customer\_rec.customer\_id;

end loop;

commit;

end;

/

begin

for rec in (

select customer\_id, name, age, interest\_rate from customers

) loop

dbms\_output.put\_line('customer id: ' || rec.customer\_id ||

', name: ' || rec.name ||

', age: ' || rec.age ||

', interest rate: ' || rec.interest\_rate);

end loop;

end;

/

**scenario 2**:

A customer can be promoted to vip status based on their balance.

**question:**

write a pl/sql block that iterates through all customers and sets a flag isvip to true for those with a balance over $10,000

begin

for cust in (

select customer\_id from customers where balance > 10000

) loop

update customers

set isvip = 'true'

where customer\_id = cust.customer\_id;

end loop;

commit;

end;

/

begin

for rec in (

select customer\_id, name, balance, isvip from customers

) loop

dbms\_output.put\_line('customer id: ' || rec.customer\_id ||

', name: ' || rec.name ||

', balance: ' || rec.balance ||

', vip status: ' || rec.isvip);

end loop;

end;

/

**scenario 3:**

The bank wants to send reminders to customers whose loans are due within the next 30 days.

**question:**

write a pl/sql block that fetches all loans due in the next 30 days and prints a reminder message for each customer.

begin

for loan\_rec in (

select customer\_id, loan\_id, due\_date

from loans

where due\_date between sysdate and sysdate + 30

) loop

dbms\_output.put\_line('reminder: loan id ' || loan\_rec.loan\_id ||

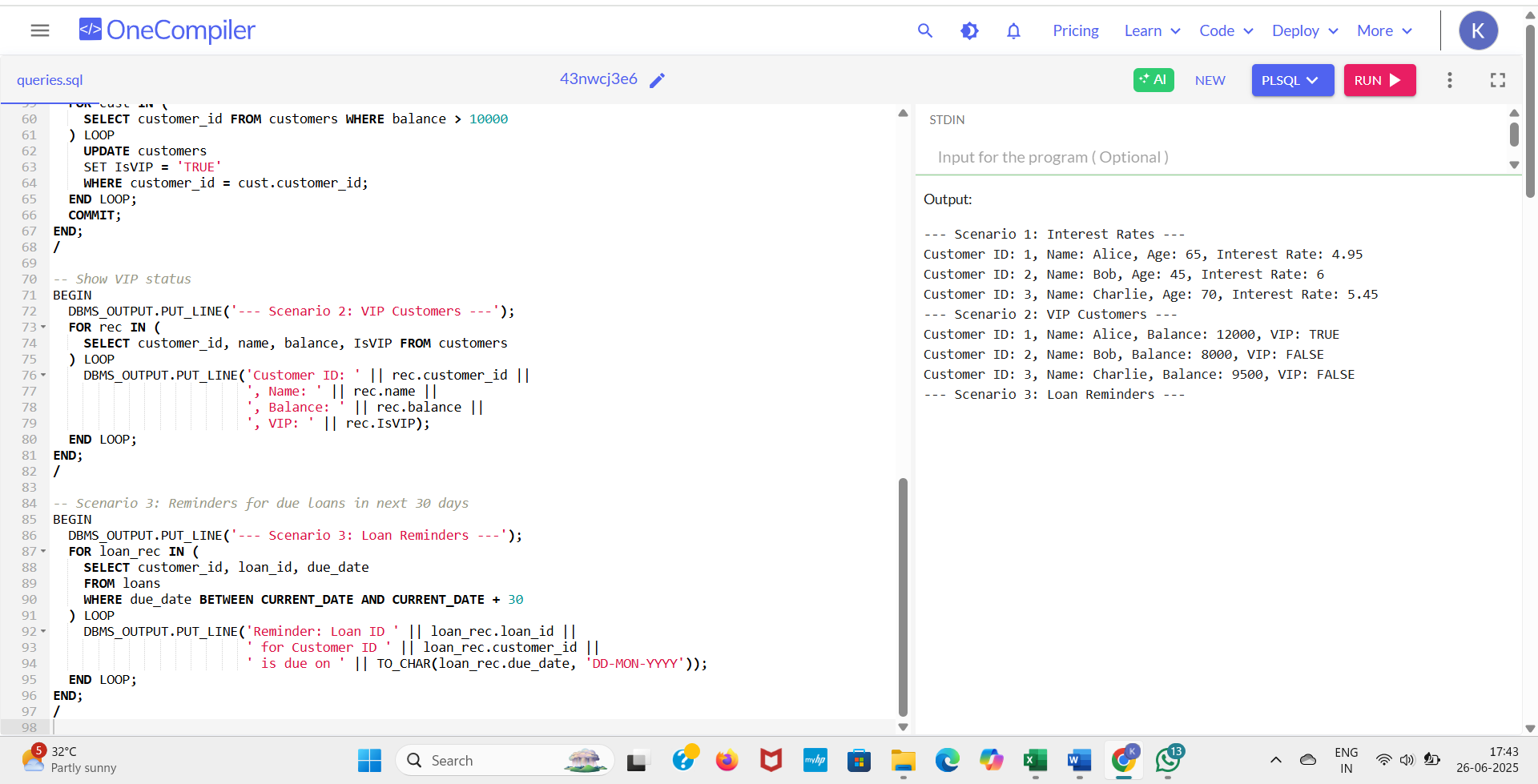
' for customer id ' || loan\_rec.customer\_id ||

' is due on ' || to\_char(loan\_rec.due\_date, 'dd-mon-yyyy'));

end loop;

end;

/

OUTPUT

**Exercise 3: stored procedures**

begin

execute immediate 'drop table accounts';

exception

when others then null;

end;

/

begin

execute immediate 'drop table employees';

exception

when others then null;

end;

/

create table accounts (

account\_id number primary key,

account\_type varchar2(20),

balance number(10, 2)

);

create table employees (

employee\_id number primary key,

name varchar2(100),

department\_id number,

salary number(10, 2)

);

insert into accounts values (101, 'savings', 1000);

insert into accounts values (102, 'savings', 2000);

insert into accounts values (103, 'current', 1500);

insert into employees values (1, 'alice', 10, 30000);

insert into employees values (2, 'bob', 20, 35000);

insert into employees values (3, 'charlie', 10, 32000);

commit;/

scenario 1: the bank needs to process monthly interest for all savings accounts.

question: write a stored procedure processmonthlyinterest that calculates and updates the balance of all savings accounts by applying an interest rate of 1% to the current balance

create or replace procedure processmonthlyinterest is

begin

for acc in (

select account\_id, balance from accounts where account\_type = 'savings'

) loop

update accounts

set balance = acc.balance + (acc.balance \* 0.01)

where account\_id = acc.account\_id;

end loop;

commit;

end;

/

**scenario 2:**

The bank wants to implement a bonus scheme for employees based on their performance.

**question:**

write a stored procedure updateemployeebonus that updates the salary of employees in a given department by adding a bonus percentage passed as a parameter.

create or replace procedure updateemployeebonus (

dept\_id in number,

bonus\_percent in number

) is

begin

update employees

set salary = salary + (salary \* bonus\_percent / 100)

where department\_id = dept\_id;

commit;

end;

/

**scenario 3:**

Customers should be able to transfer funds between their accounts.

**question**:

write a stored procedure transferfunds that transfers a specified amount from one account to another, checking that the source account has sufficient balance before making the transfer.

create or replace procedure transferfunds (

from\_account in number,

to\_account in number,

amount in number

) is

insufficient\_balance exception;

source\_balance number;

begin

select balance into source\_balance from accounts where account\_id = from\_account;

if source\_balance < amount then

raise insufficient\_balance;

else

update accounts

set balance = balance - amount

where account\_id = from\_account;

update accounts

set balance = balance + amount

where account\_id = to\_account;

commit;

end if;

exception

when insufficient\_balance then

dbms\_output.put\_line('transfer failed: insufficient funds.');

when others then

dbms\_output.put\_line('error: ' || sqlerrm);

end;

/

begin

processmonthlyinterest;

dbms\_output.put\_line('--- after monthly interest ---');

for acc in (select \* from accounts) loop

dbms\_output.put\_line('account ' || acc.account\_id || ', type: ' || acc.account\_type || ', balance: ' || acc.balance);

end loop;

end;

/

begin

updateemployeebonus(10, 10); -- 10% bonus to department 10

dbms\_output.put\_line('--- after bonus update ---');

for emp in (select \* from employees) loop

dbms\_output.put\_line('employee ' || emp.employee\_id || ', name: ' || emp.name || ', salary: ' || emp.salary);

end loop;

end;

/

begin

transferfunds(101, 103, 500); -- transfer ₹500 from 101 to 103

dbms\_output.put\_line('--- after fund transfer ---');

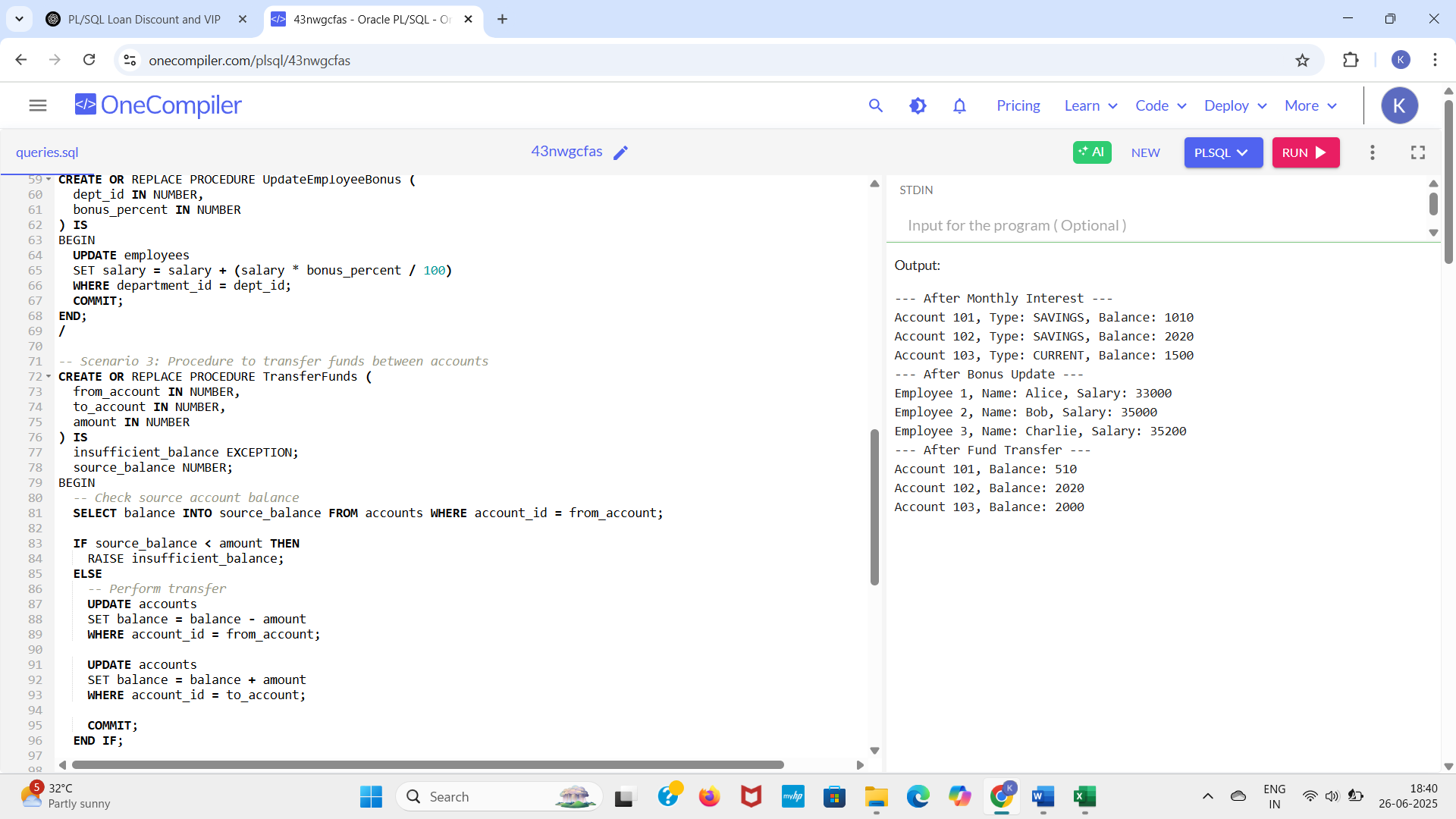
for acc in (select \* from accounts) loop

dbms\_output.put\_line('account ' || acc.account\_id || ', balance: ' || acc.balance);

end loop;

end;

/

**OUTPUT**

**TDD using JUnit5 and Mockito**

**Exercise 1: setting up junit**

**Scenario:** you need to set up junit in your java project to start writing unit tests.

steps: 1. create a new java project in your ide (e.g., intellij idea, eclipse).

2. add junit dependency to your project.

3. create a new test class in your project.

code:

package week2;

public class calculator {

public int add(int a, int b) {

return a + b;

}

}

package week2;

import org.junit.test;

import static org.junit.assert.*assertequals*;

public class calculatortest {

public void testadd() {

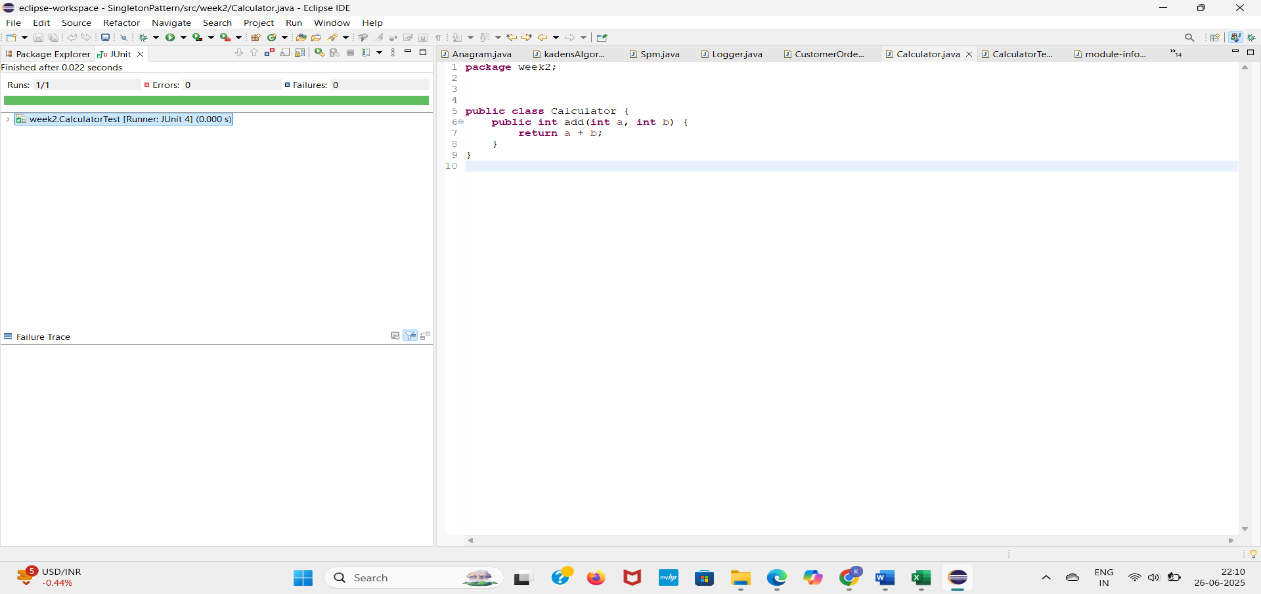
calculator calc = new calculator();

int result = calc.add(2, 3);

*assertequals*(5, result);

}

}

OUTPUT

**Exercise 3: assertions in junit**

**Scenario:** you need to use different assertions in junit to validate your test results. steps: 1. write tests sing various junit assertions.

**Code:**

package week2;

import org.junit.test;

import static org.junit.assert.\*;

public class assertionstest {

public void testassertions() {

// assert equals

assertequals(5, 2 + 3);

// assert true

asserttrue(5 > 3);

// assert false

assertfalse(5 < 3);

// assert null

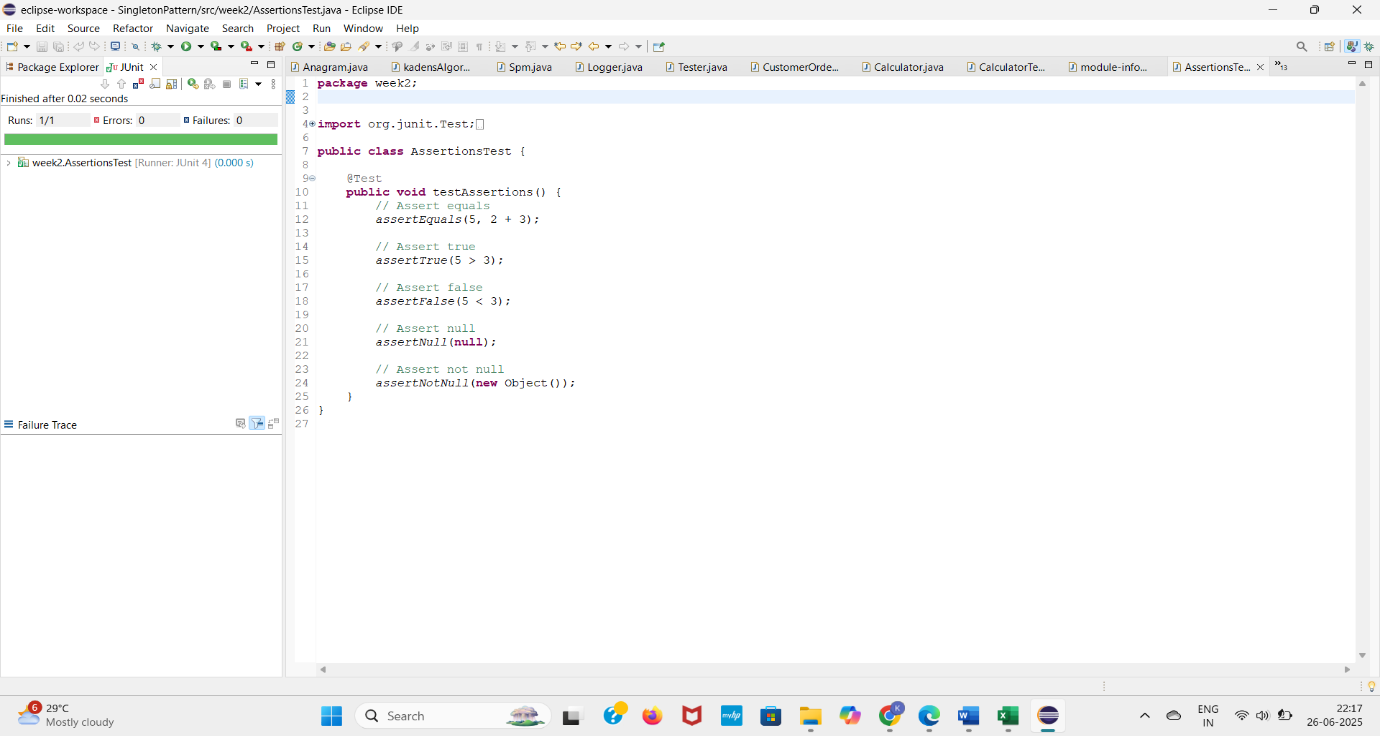
assertnull(null);

// assert not null

assertnotnull(new object());

}

}

**OUTPUT**

**Exercise 4:** **arrange-act-assert (aaa) pattern, test fixtures, setup and teardown methods in junit**

**Scenario:**

you need to organize your tests using the arrange-act-assert (aaa) pattern and use setup and teardown methods.

steps:

1. write tests using the aaa pattern.

2. use @before and @after annotations for setup and teardown methods.

**Code:**

package week2;

import org.junit.before;

import org.junit.after;

import org.junit.test;

import static org.junit.assert.\*;

public class calculatortestaaa {

private calculator calculator;

public void setup() {

system.out.println("setting up...");

calculator = new calculator(); // arrange

}

@after

public void teardown() {

system.out.println("tearing down...");

calculator = null;

}

@test

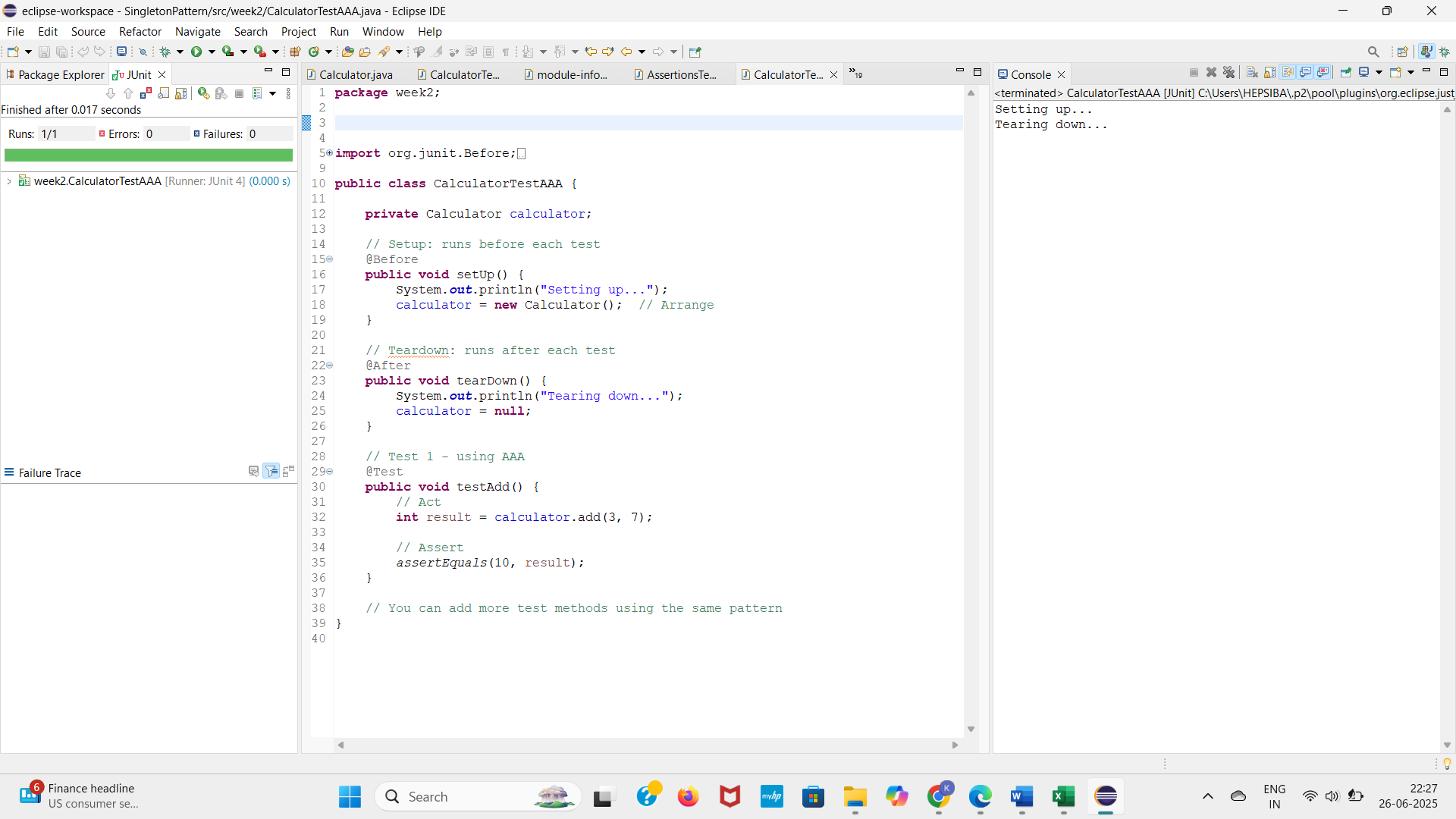
public void testadd() {

int result = calculator.add(3, 7);

assertequals(10, result);

}

}

**OUTPUT**

**Exercise 1: mocking and stubbing**

**Scenario:** you need to test a service that depends on an external api. use mockito to mock the external api and stub its methods.

steps:

1. create a mock object for the external api.

2. stub the methods to return predefined values.

3. write a test case that uses the mock object.

code:

import static org.mockito.mockito.\*;

import static org.junit.jupiter.api.assertions.\*;

import org.junit.jupiter.api.test;

public class myservicetest {

public void testexternalapi() {

externalapi mockapi = mock(externalapi.class);

when(mockapi.getdata()).thenreturn("mock data");

myservice service = new myservice(mockapi);

string result = service.fetchdata();

assertequals("mock data", result);

}

}

public class myservice {

private externalapi api;

public myservice(externalapi api) {

this.api = api;

}

public string fetchdata() {

return api.getdata();

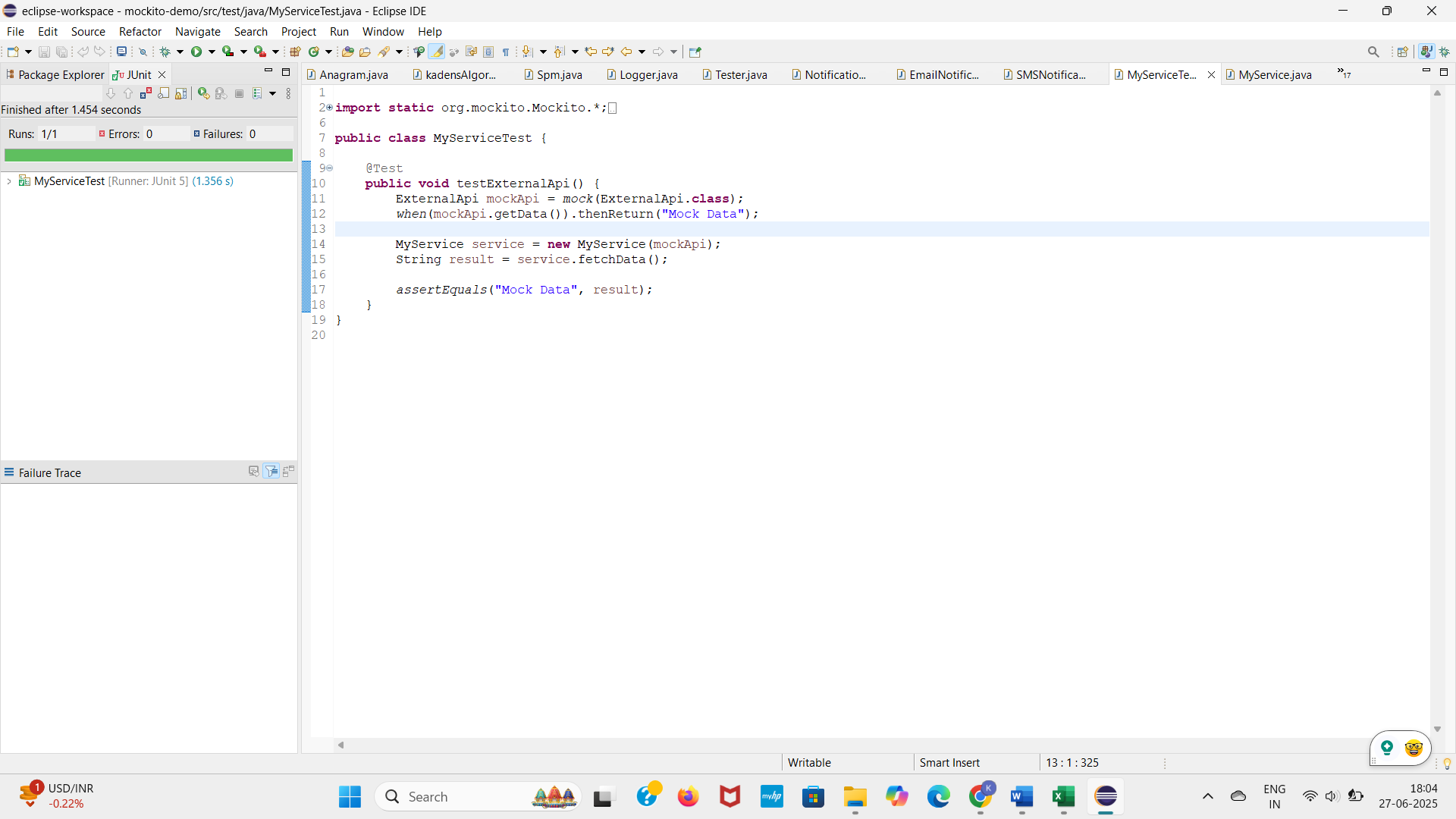
}

}

public interface externalapi {

string getdata();}

**OUTPUT**



**Exercise 2: verifying interactions**

**Scenario:** you need to ensure that a method is called with specific arguments.

steps: 1. create a mock object.

2. call the method with specific arguments.

3. verify the interaction

**Code**

import static org.mockito.mockito.\*;

import org.junit.jupiter.api.test;

import org.mockito.mockito;

interface externalapi {

void getdata();

}

class myservice {

private externalapi api;

public myservice(externalapi api) {

this.api = api;

}

public void fetchdata() {

api.getdata(); // the interaction we want to verify

}

}

public class set {

public void testverifyinteraction() {

externalapi mockapi = mockito.mock(externalapi.class);

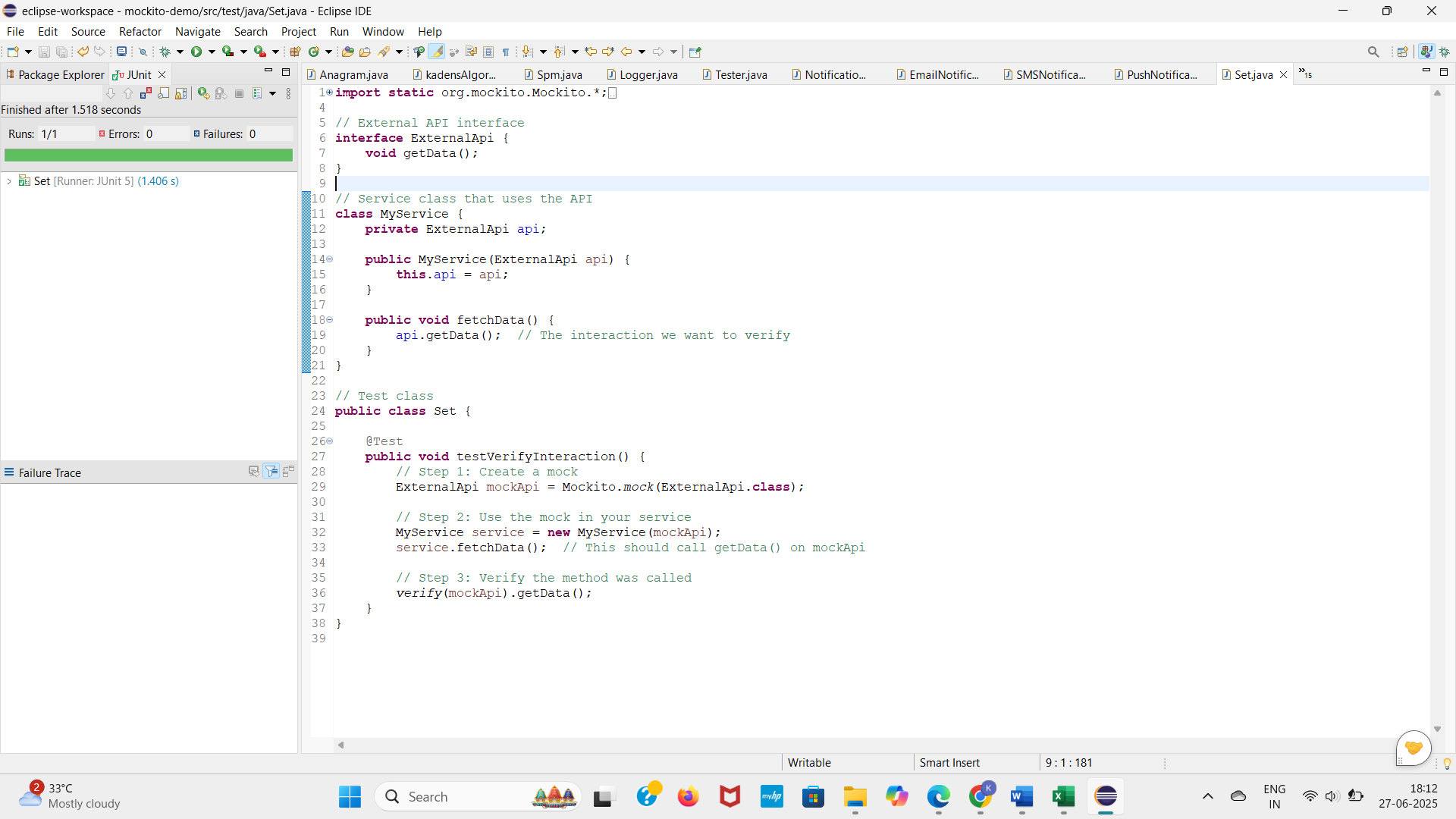
myservice service = new myservice(mockapi);

service.fetchdata(); // this should call getdata() on mockapi

verify(mockapi).getdata();

}

}

**OUTPUT**

**logging using slf4j**

**Exercise 1: logging error messages and warning levels**

**Task:**

write a java application that demonstrates logging error messages and warning levels using slf4j.

**CODE**

import org.slf4j.logger;

import org.slf4j.loggerfactory;

public class loggingexample {

private static final logger *logger* = loggerfactory.*getlogger*(loggingexample.class);

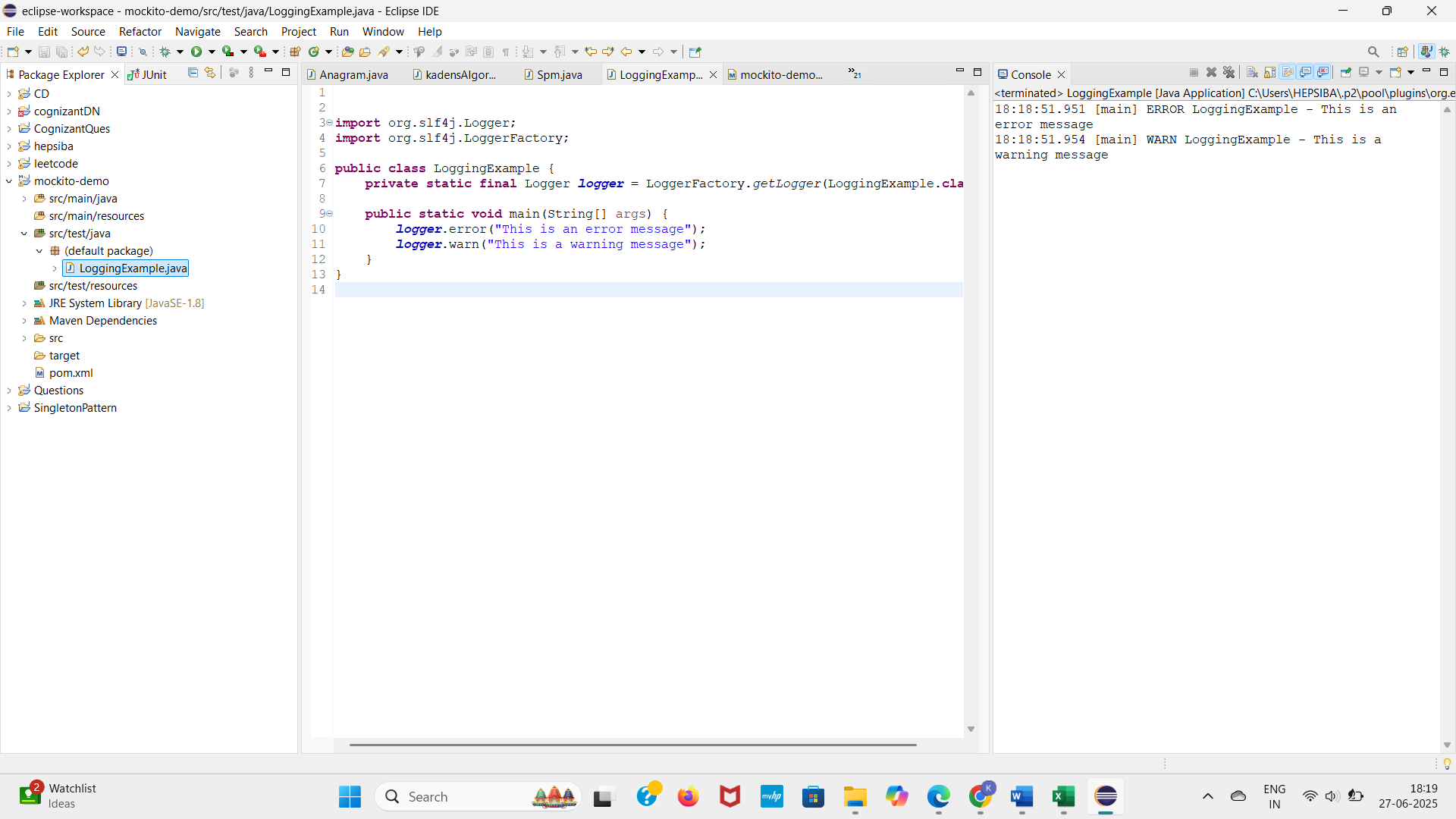
public static void main(string[] args) {

*logger*.error("this is an error message");

*logger*.warn("this is a warning message");

}

}

OUTPUT