

Lab Assignment-3.3

Name : Saini.Deepthi
Hall ticket No. : 2403A52014
Batch No. : 24BTCAIAIB02
Course Title : Ai Assisted Coding

TASK -1

Description:

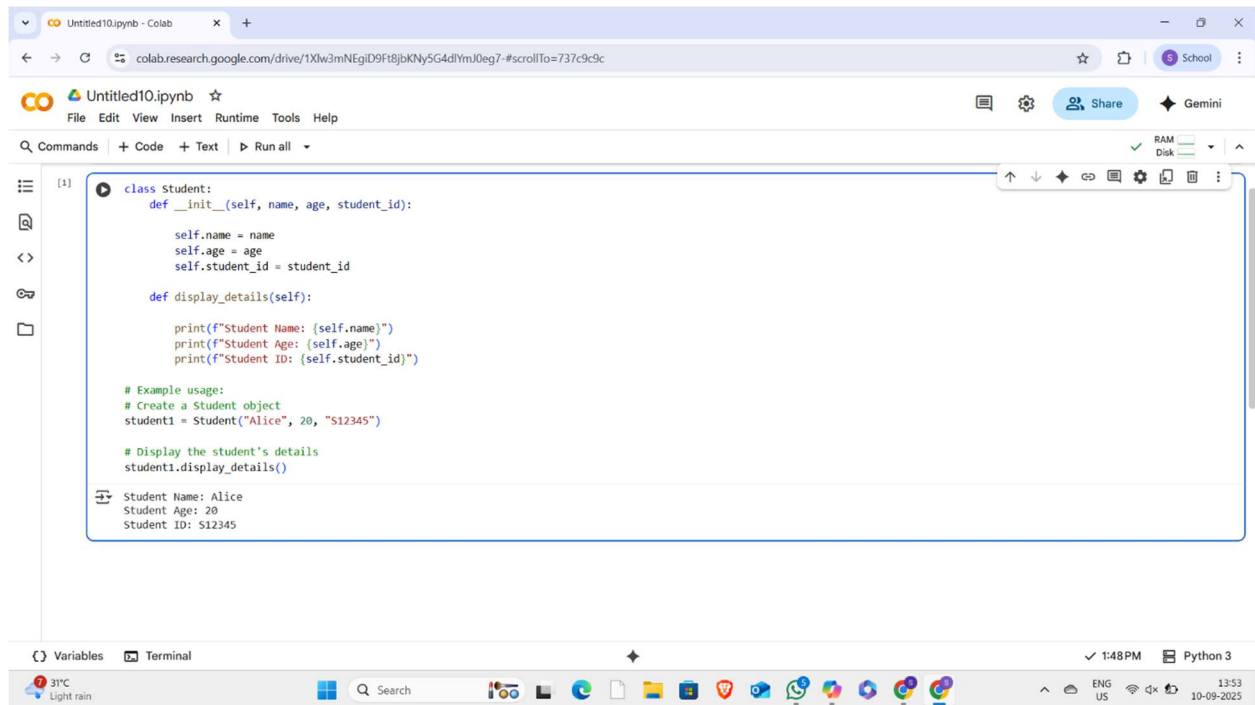
>> Use AI to complete a Student class with attributes and a method.
>> Check output.
>> Analyze the code generated by AI tool.

Expected Output:

>> Class with constructor and display_details() method.

Prompt:

>> Create a Python class named Student with attributes and a method .Class with constructor and display_details() method give a code.



The screenshot shows a Google Colab notebook titled 'Untitled10.ipynb'. The code defines a 'Student' class with an '__init__' method that takes 'name', 'age', and 'student_id' as arguments and assigns them to instance variables. It also has a 'display_details' method that prints the student's name, age, and ID. Below the code, the output shows the successful execution of the code, creating a 'Student' object named 'student1' and displaying its details: 'Student Name: Alice', 'Student Age: 20', and 'Student ID: S12345'.

```
[1] class Student:
    def __init__(self, name, age, student_id):
        self.name = name
        self.age = age
        self.student_id = student_id

    def display_details(self):
        print(f"Student Name: {self.name}")
        print(f"Student Age: {self.age}")
        print(f"Student ID: {self.student_id}")

# Example usage:
# Create a Student object
student1 = Student("Alice", 20, "S12345")

# Display the student's details
student1.display_details()
```

Student Name: Alice
Student Age: 20
Student ID: S12345

Observation:

>> Based on the code you ran, the `display_details()` method of the `Student` class successfully printed the name, age, and student ID of the `student1` object you created.

Explanation:

>>The code you executed successfully created a representation of a student named Alice with her age and student ID, and then displayed these details. Now that you have a working `Student` class, you could create more student objects, add new capabilities to the `Student` class (like updating their information), or explore more advanced concepts like creating different types of students based on the current class.

TASK -2

Description:

>> Prompt AI to complete a function that prints the first 10 multiples of a number using a loop.

>> Analyze the generated code.

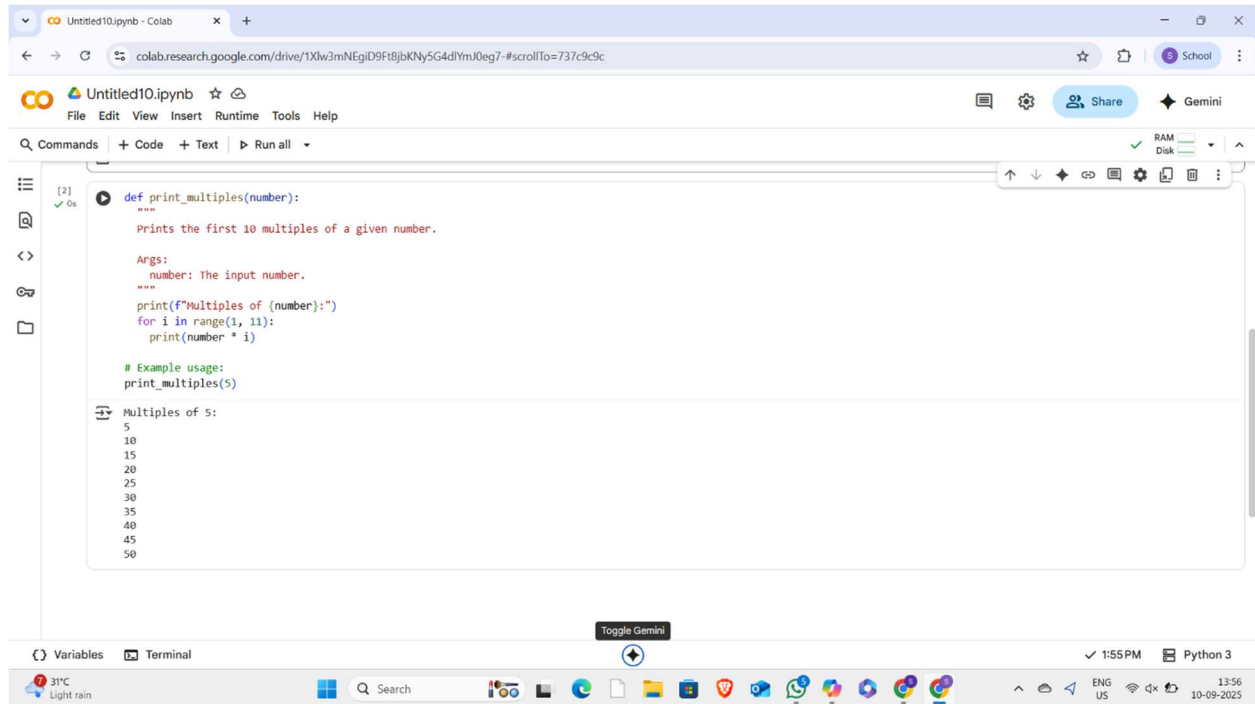
>>Ask AI to generate code using other controlled looping.

Expected Output:

>>Correct loop-based implementation.

Prompt1:

>> Write a Python function that takes a number as input and prints its first 10 multiples using a for loop.



The screenshot shows a Google Colab notebook titled 'Untitled10.ipynb'. The code cell contains a Python function `print_multiples` that takes a number as input and prints its first 10 multiples using a for loop. The function is defined as follows:

```
def print_multiples(number):  
    """  
    Prints the first 10 multiples of a given number.  
    Args:  
        number: The input number.  
    """  
    print(f"Multiples of {number}:")  
    for i in range(1, 11):  
        print(number * i)  
  
# Example usage:  
print_multiples(5)
```

The output of the function is displayed below the code cell, showing the first 10 multiples of 5:

```
Multiples of 5:  
5  
10  
15  
20  
25  
30  
35  
40  
45  
50
```

The bottom of the notebook shows the 'Variables' and 'Terminal' tabs, and the system tray at the bottom of the browser window displays the date and time as 13:56 on 10-09-2025.

Prompt2:

>> Write a Python function that prints the first 10 multiples of a number using a while loop.

```
[4] def print_multiples_while(number):  
    """  
    Prints the first 10 multiples of a given number using a while loop.  
  
    Args:  
        number: The input number.  
    """  
    print(f"Multiples of {number}:")  
    i = 1  
    while i <= 10:  
        print(number * i)  
        i += 1  
  
    # Example usage:  
    print_multiples_while(5)  
  
[2] Multiples of 5:  
5  
10  
15  
20  
25  
30  
35  
40  
45  
50
```

Observation:

>> For Loop

- Iterates over a fixed range (range(1, 11)).
- Ideal for tasks with a known number of repetitions.
- Code is concise and easy to read.
- Less flexible if dynamic exit conditions are needed.

->>While Loop

- Uses a condition and manual counter (while count <= 10).
- Offers more control over loop behavior.
- Slightly more verbose; requires careful counter management.
- Risk of infinite loop if condition or increment is mishandled.

Explanation:

>> Controlled loops repeat actions based on a set condition or count.

A for loop is ideal for fixed iterations, like printing 10 multiples.

A while loop offers more flexibility but needs manual counter control.

Both loops achieve the same result but differ in structure and use cases.

Choosing between them depends on whether simplicity or dynamic control is needed.

Key Difference:

>> for loop is cleaner and safer for fixed repetitions.

>> while loop gives more control but needs careful handling of the counter to avoid infinite loops.

TASK -3

Description:

>> Ask AI to write nested if-elif-else conditionals to classify age groups.

>> Analyze the generated code

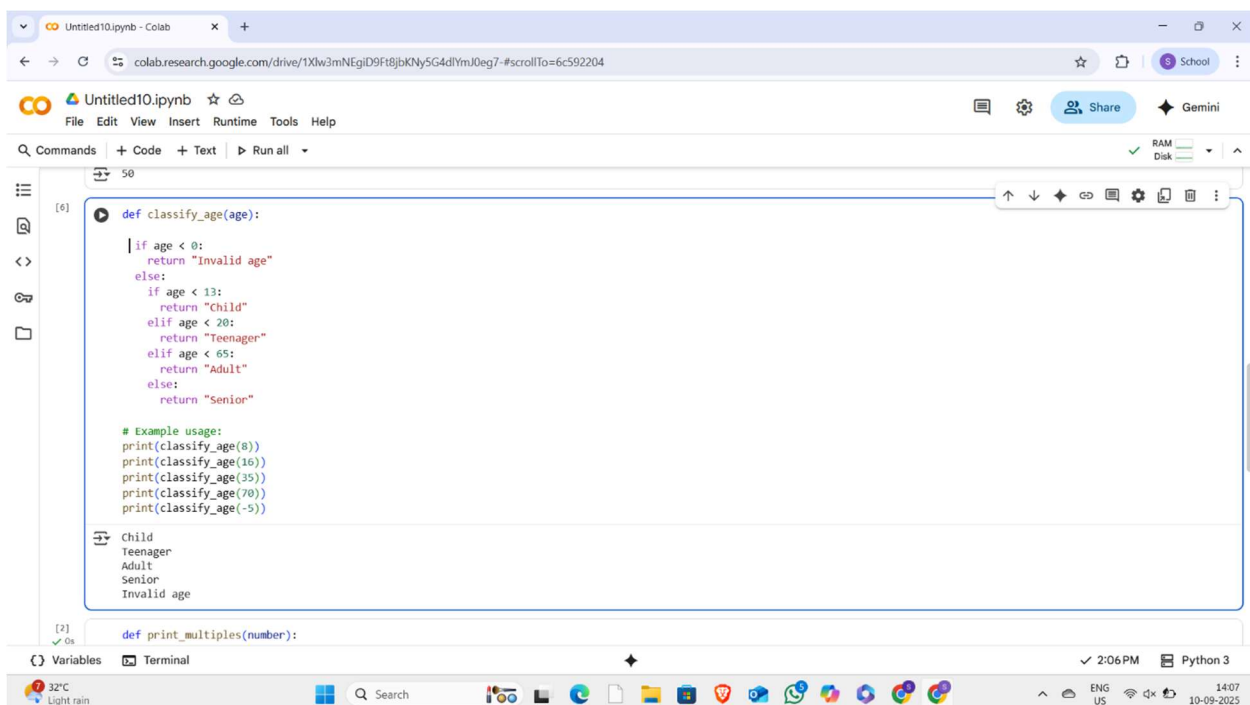
>> Ask AI to generate code using other conditional statements

Expected Output:

>> Age classification function with appropriate conditions and with explanation

Prompt1:

>> Write a Python function using nested if-elif-else statements to classify a person's age.

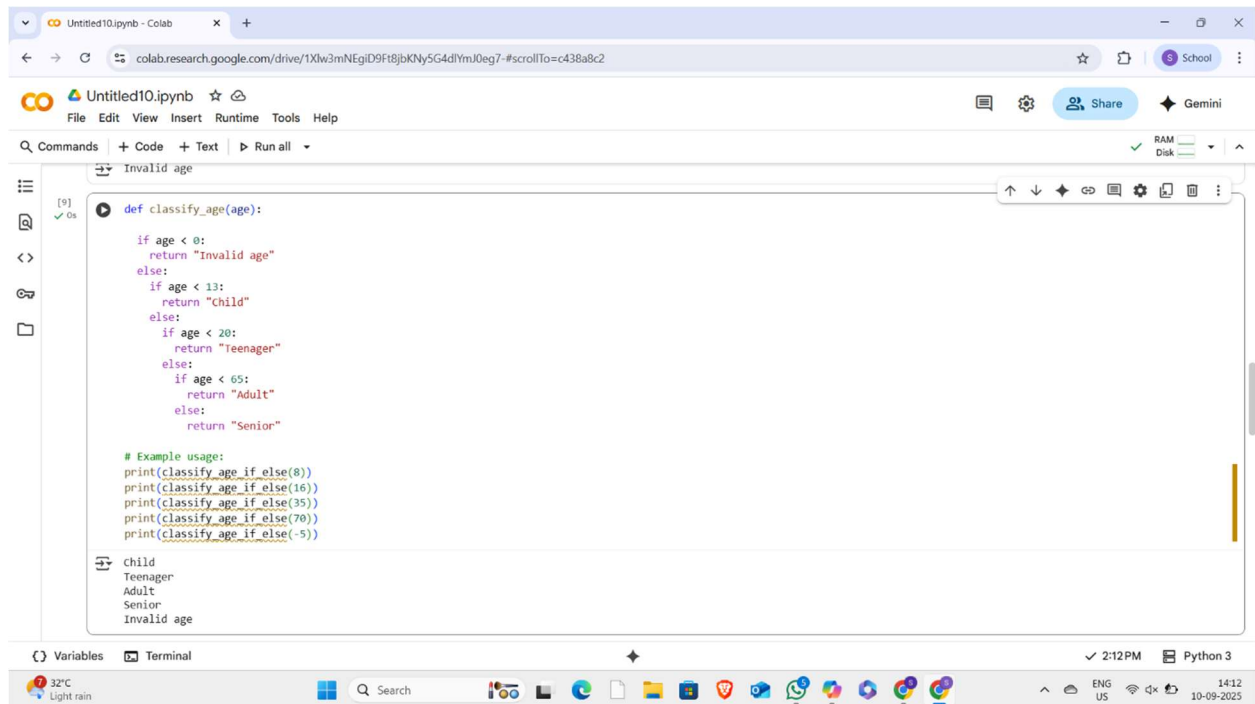


The screenshot shows a Google Colab notebook interface. The main code cell contains a Python function `def classify_age(age):` with nested if-elif-else statements to classify age groups. The function returns "Invalid age" for ages less than 0, "Child" for ages 0-13, "Teenager" for ages 13-20, "Adult" for ages 20-65, and "Senior" for ages 65 and above. Below the function definition, there is an example usage section with five print statements: `print(classify_age(8))`, `print(classify_age(16))`, `print(classify_age(35))`, `print(classify_age(70))`, and `print(classify_age(-5))`. The output of these print statements is displayed in a separate cell below the code, showing "Child", "Teenager", "Adult", "Senior", and "Invalid age" respectively. The notebook interface includes a menu bar with options like File, Edit, View, Insert, Runtime, Tools, and Help. The bottom status bar shows the system clock as 2:06 PM, the language as Python 3, and the date as 10-09-2025.

```
[6] def classify_age(age):  
    if age < 0:  
        return "Invalid age"  
    else:  
        if age < 13:  
            return "Child"  
        elif age < 20:  
            return "Teenager"  
        elif age < 65:  
            return "Adult"  
        else:  
            return "Senior"  
  
    # Example usage:  
    print(classify_age(8))  
    print(classify_age(16))  
    print(classify_age(35))  
    print(classify_age(70))  
    print(classify_age(-5))  
  
Child  
Teenager  
Adult  
Senior  
Invalid age  
  
[2] def print_multiples(number):
```

Prompt2:

Write a Python function using nested if-elif-else statements to classify a person's age.



The screenshot shows a Google Colab notebook titled 'Untitled10.ipynb'. The code cell contains a Python function `def classify_age(age):` with nested if-elif-else statements. The function returns 'Invalid age' for `age < 0`, 'Child' for `0 <= age < 13`, 'Teenager' for `13 <= age < 20`, 'Adult' for `20 <= age < 60`, and 'Senior' for `age >= 60`. Below the function, there is an example usage section with print statements for ages 8, 16, 35, 70, and -5. The output of the code cell shows the results: 'Child', 'Teenager', 'Adult', 'Senior', and 'Invalid age'.

```
def classify_age(age):  
    if age < 0:  
        return "Invalid age"  
    elif age < 13:  
        return "Child"  
    elif age < 20:  
        return "Teenager"  
    elif age < 60:  
        return "Adult"  
    else:  
        return "Senior"  
  
# Example usage:  
print(classify_age(8))  
print(classify_age(16))  
print(classify_age(35))  
print(classify_age(70))  
print(classify_age(-5))  
  
Child  
Teenager  
Adult  
Senior  
Invalid age
```

Observation:

- >>The function clearly separates age groups using logical conditions.
- >>It includes a check for invalid input, making it more reliable.
- >> Each age range is exclusive, so there's no overlap or confusion.
- >>The structure is easy to read and modify if age brackets change.
- >>It's practical for real-world use like surveys, memberships, or analytics.

Explanation:

- >> `age < 0`: Handles invalid input like negative ages.
- >> `age <= 12`: Classifies children (typically up to primary school age).
- >> `age <= 19`: Covers teenagers (adolescents in middle/high school).
- >> `age <= 59`: Defines adults (working-age population).
- >> `else`: Captures seniors (usually 60 and above).

This structure uses if-elif-else for clean, readable logic and ensures each age falls into exactly one category.

TASK -4

Description:

>>Generate a sum_to_n() function to calculate sum of first n numbers

>> Analyze the generated code

>> Get suggestions from AI with other controlled looping

Expected Output:

>>Python code with explanation

Prompt1:

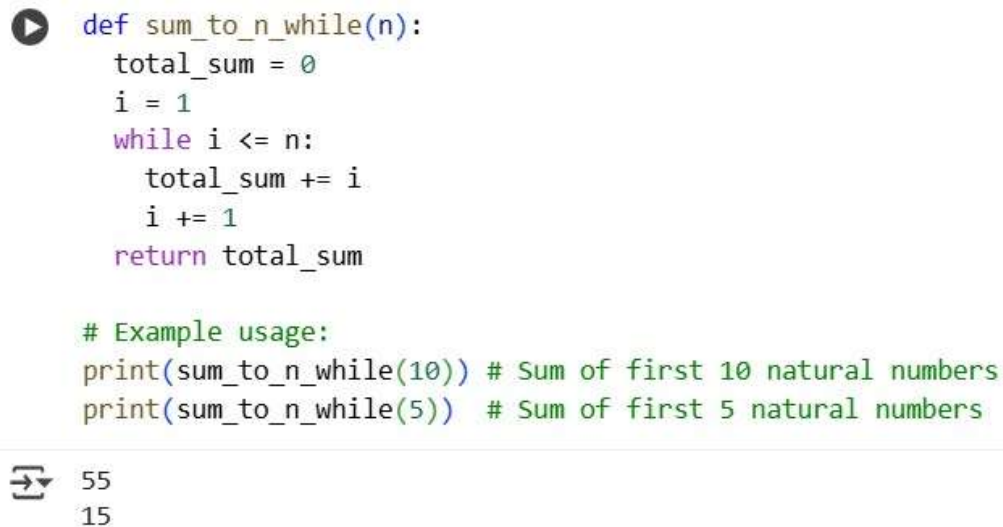
>> Generate a sum_to_n() function to calculate sum of first n numbers in for loop

```
def sum_to_n(n):  
    if n < 1:  
        return -1 # Or raise an error, depending on desired behavior for invalid input  
    else:  
        total = 0  
        for i in range(1, n + 1):  
            total += i  
        return total  
  
# Example usage:  
print(f"The sum of the first 5 numbers is: {sum_to_n(5)}")  
print(f"The sum of the first 10 numbers is: {sum_to_n(10)}")  
print(f"The sum of the first 0 numbers is: {sum_to_n(0)}")
```

```
↩ The sum of the first 5 numbers is: 15  
The sum of the first 10 numbers is: 55  
The sum of the first 0 numbers is: -1
```

Prompt2:

>> Generate a sum_to_n() function to calculate sum of first n numbers in while loop.



```
def sum_to_n_while(n):
    total_sum = 0
    i = 1
    while i <= n:
        total_sum += i
        i += 1
    return total_sum

# Example usage:
print(sum_to_n_while(10)) # Sum of first 10 natural numbers
print(sum_to_n_while(5))  # Sum of first 5 natural numbers
```

55
15

Observation:

- >>The function adds numbers from 1 to n using a loop.
- >>It checks if n is less than 1 and returns -1 for invalid input.
- >>It uses a for loop to keep adding each number to a total.
- >> The result is the sum of the first n natural numbers.
- >> It's easy to understand and works well for small values of n.

Explanation:

>>The function calculates the sum of the first n natural numbers using a loop. It starts by checking if n is less than 1—if so, it returns -1 to show the input is invalid. If the input is valid, it sets a variable total to 0 and then uses a for loop to add each number from 1 to n to that total. After the loop finishes, it returns the final sum.

TASK -5

Description:

- >>Use AI to build a BankAccount class with deposit, withdraw, and balance methods.
- >>Analyze the generated code.
- >>Add comments and explain code.

Expected Output:

- >> Python code with explanation.

Prompt:

>> Write a Python class called BankAccount that includes methods for depositing money, withdrawing money, and checking the current balance.

```
class BankAccount:
    def __init__(self, account_holder, initial_balance=0):
        # Initialize account holder name and balance
        self.account_holder = account_holder
        self.balance = initial_balance

    def deposit(self, amount):
        # Check if the deposit amount is positive
        if amount > 0:
            # Add the deposit amount to the balance
            self.balance += amount
            print(f"Deposit of {amount} successful. New balance: {self.balance}")
        else:
            # Print an error message for invalid deposit amount
            print("Invalid deposit amount. Amount must be positive.")

    def withdraw(self, amount):
        # Check if the withdrawal amount is positive
        if amount <= 0:
            print("Invalid withdrawal amount. Amount must be positive.")
        # Check if there are sufficient funds
        elif amount > self.balance:
            print("Insufficient funds.")
        else:
            # Subtract the withdrawal amount from the balance
            self.balance -= amount
            print(f"Withdrawal of {amount} successful. New balance: {self.balance}")

    def check_balance(self):
        # Print the current balance
        print(f"Current balance for {self.account_holder}: {self.balance}")
```

Example usage:

```
account1 = BankAccount("Alice Smith", 1000)
account1.check_balance()
account1.deposit(500)
account1.withdraw(200)
account1.check_balance()
account1.withdraw(2000) # Example of insufficient funds
account1.deposit(-100) # Example of invalid deposit amount
```

➤ Current balance for Alice Smith: 1000
Deposit of 500 successful. New balance: 1500
Withdrawal of 200 successful. New balance: 1300
Current balance for Alice Smith: 1300
Insufficient funds.
Invalid deposit amount. Amount must be positive.

Observation:

>>The class helps manage money by letting you deposit, withdraw, and check your balance.

>>It keeps track of the balance using a variable inside the class.

>>The deposit method adds money, and the withdraw method subtracts it.

>>It should check if there's enough money before allowing a withdrawal.

>> The balance method shows how much money is left in the account

Explanation:

>>The BankAccount class helps you manage money like a real bank account.

>>It has a deposit method to add money, a withdraw method to take money out, and a balance method to check how much is left.

>>If you try to withdraw more than you have, it should show an error or warning.

>>The account keeps track of your money using a variable called balance.

>>This kind of class is useful for learning how to store and update values in programs.

