We are constantly riddled with problems in our daily lives that we want to solve to make our lives better. To solve these problems, we first need to identify the problem and form a plan of action, then execute that action and evaluate whether that action successfully solved the problem. In the following exercises, you will learn how to identify the problem, how to devise a plan, how to execute that plan and finally how to evaluate the solution.

The image above shows the general process of solving problems. Your initial plan may not be perfect, but through execution and validation you will be able to refine your solutions so that they are useful and efficient.

Okay, let’s do some examples to help you better understand the process of problem-solving.
In the above diagram, how would you help Mario reach the flag? First, let’s understand the problem.

### 1 The problem:

The main problem is that Mario needs to reach the flag. However, there is a hole in the ground and two enemies in front of him. He needs to avoid the hole and the enemies and reach the flag. He can also kill the enemy by jumping on their head.

Therefore, we have identified the problem. Now we devise a plan.

### 2 Initial Plan:

A problem can be solved many ways, so your solution plan may be different from this one. The coherence and validity of the plan is determined after the execution and during evaluation. This is one possible solution for this problem:

1. Mario moves forward.
2. If there is a hole in the ground, jump over it.
3. Continue moving until there is an enemy in front of Mario.
4. Jump over the first enemy and move forward until there is a second enemy.
5. Jump over the second enemy and move forward until Mario reaches the flag.
6. When Mario reaches the flag, we have success.

### 3 Execute and Evaluate Initial Plan

If you visually trace the initial plan, you will see that the plan is valid as it solves the problem. However, if we look at the steps we encounter some questions about the problem:

- Do these steps work in other scenarios?
- Mario can also jump on the enemies’ head and kill them. That would also solve the problem. Can we create a solution that allows the user to choose how they want to solve the problem?

### 4 Revised plan:

1. Move Mario forward.
2. Check if there is a hole or enemy in front of him.
   a. If there is hole, then jump over it.
   b. If there is an enemy in front of him, then jump over the enemy or jump on the enemy to kill it.
   c. if there is a flag, go to #4
3. Continue steps 1-3
4. END Mario reaches the flag.

In this second situation the initial plan does not work, so we need to make a plan that is going to be applicable to all scenarios. See step 3
Suppose you are traveling to London, where the temperature is measured in Celsius. Since we use Fahrenheit for temperature in the United States, you want to make a simple program that would take Celsius as input and convert the temperature to Fahrenheit.

### Understanding the problem:
I think the problem is that we need a way to convert Celsius temperature into Fahrenheit.

### Initial plan:
1. Take the input in Celsius.
2. Convert the input to Fahrenheit.
3. Write the output.

### Execution and Evaluation
Even though the instruction conveys the logic correctly, the steps are not very clear. If one must read through the steps, they could encounter questions like:

- Where do I get the input from?
- Where do I write the values for Celsius and Fahrenheit?
- What is the formula for converting from Celsius to Fahrenheit?

### Revising the plan:
Rewrite the instructions addressing the errors found during the evaluation of our plan. Remember:

1. The instructions are not just for you, other people should be able to find the answer using your plan.
2. Your instructions must include obvious steps and sufficient information.

### Revised Plan
1. Read the temperature in Celsius
2. Write the temperature in Celsius as a variable C.
3. Write another variable called F which will store the value for the temperature in Fahrenheit.
4. Use the conversion formula:  
   \[ F = \frac{9}{5} C + 32 \]
5. Write the value of F i.e. temperature in Fahrenheit.

**Pseudocode VS Instructions:**
We have seen from the previous examples that instructions can be quite helpful when creating a problem-solving plan. Pseudocode is a form of instructions that are specifically directed towards computer programmers. Therefore, we should be more specific when writing pseudocode. Below, we will practice some computing concepts with the help of pseudocode:
Suppose you are designing an elevator with very simple logic: it has three buttons corresponding to three floors; when a user presses a button, it moves the elevator to that floor. How would you write the pseudocode for this program?

Remember the guidelines for problem-solving!

1. **Understanding the problem:**

   We are developing an elevator program, where a user can press a number from 1 to 3 and the elevator will move to the floor corresponding to that number.

2. **Initial plan:**

   1. Show the user the floor numbers.
   2. Get the floor number from the user.
   3. Move the elevator to that floor number.

3. **Execution and Evaluation**

   Reading through the initial plan, we encounter some questions like:
   - What are the floor numbers?
   - Where do I show the numbers?
   - Where do I store the user’s choice?
   - Where should I write the output?
   - How does the user know whether the floor has been reached?
Here you can see I have CAPITALIZED the actions words.
This keeps the pseudocode formal and specific.
I have made indentations when writing the IF statements.
This allows the programmer to see which statements are executed if the IF statement is TRUE.
These are some standard pseudocode practices. We will see more of these in further examples.
You can also find these standards in this link: CLICK ME

4 Revised plan:

1. PRINT: “Please enter the floor number you want to go to:
   1: First Floor
   2: Second Floor
   3: Third Floor”.
2. CREATE variable called userChoice to store the user’s choice.
3. STORE the number the user entered userChoice.
4. IF userChoice is equal to 1
   a. Move the elevator to first floor
   b. PRINT: “You have reached the first floor.”
ELSE IF userChoice is equal to 2
   a. Move the elevator to second floor.
   b. PRINT: “You have reached the second floor.”
ELSE IF userChoice is equal to 3
   a. Move the elevator to third floor
   b. PRINT: “You have reached the third floor.”

5 Execution

When the program runs, each step is executed as such:

1. The program prints, “Please enter the floor number you want to go to:
   1: First Floor.
   2: Second Floor.
   3: Third Floor.”
2. A variable called userChoice is created.
   a. Assume, the user entered: 2
3. userChoice = 2.
4. Since, userChoice = 2, the elevator will move to second floor and print: “You have reached the second floor”

6 Evaluation

This new plan is much better but while executing new questions emerged:

- What if the user enters a number that is not within 1 to 3, or what if it is not even a number?
- What if the user made a mistake, and needs to go to a different floor? This code runs just one time. Can we let the user confirm that they are indeed in the correct floor?

Try to solve the above questions and write a more detailed plan:
Driving around cities in the United States, you may have seen large water towers, like the one pictured. These water towers provide water to your house. Each of these towers can contain millions of gallons of water. Let’s say you are designing a program that starts filling the reservoir when it’s empty and stops when it’s full. Can you design a pseudocode for this program?

**Problem Statement:**

We are developing a program that detects when a water tower is not full and fills it with water.

**Solution plan:**

1. CREATE a variable called reservoirCap to store how much water a reservoir can hold
2. PRINT: “Please enter the capacity of the reservoir:”
3. READ user input into variable reservoirCap.
4. CREATE a variable called waterLevel, STORE 0 into it.
5. CHECK WHILE (waterLevel is not equal to reservoirCap)
   5.1. Add one gallon of water.
   5.2. ADD 1 to the waterLevel.
6. ENDWHILE
7. PRINT: “The reservoir is full.”

**Execution**

1. A variable called reservoirCap is created.
2. The program writes: “Please enter the capacity of the reservoir:”
3. Suppose the user enters 10, reservoirCap = 10
4. A variable called waterLevel is created.
5. Since waterLevel = 0 and reservoirCap = 10,
   5.1 the program adds one gallon of water
   5.2 and adds 1 to waterLevel.
6. Step 5 continues while waterLevel is not equal to reservoirCap.
7. The system writes: “The reservoir is full.”

**Evaluation:**

The program works as expected.
As a kid, I remember having one of those multiplication rulers and being one of the cool kids in school (simple times!). Now that we all pretty much have calculators/smartphones we don’t really need them anymore, but nevertheless multiplication still is a necessary process of math. So, design a program that takes a number as user input and displays its multiplication table from 1 to 10.

**Problem Statement**

The program will ask a user for a number and then display its multiplication table from 1 to 10.

**Solution Plan**

1. PRINT “Please enter a number: ”
2. READ user input into variable number
3. START FOR multiplier from 1 to 10
   3.1 PRINT number, “ X “, multiplier, “=”, (number * multiplier)
4. END FOR

**Execution**

1. The console prints: Please enter a number
2. Suppose user enters 6, number = 6
3. Since number = 6 and in the first run of the for-loop multiplier = 1
   3.1 The console prints: 6 X 1 = 6
   The for loop runs 10 times, changing the value of multiplier from 1 to 10. In the 10th run the console prints: 6 X 10 = 60
4. The for loop ends

**Evaluation**

The program seems to be working fine when the user enters a number. But problems may happen if the user enters a value that is not a number.
One day, I saw a man standing on one leg in front of an ATM, curious I asked him why he was standing on one leg like that. He said, “nothing man! I’m just checking my balance” (Ba dum tssss!!)

Now that you are almost dying of laughter, why don’t we make a simple ATM program which has two functions, deposit and withdraw. Both functions will take the account number, a pin code and some amount of money to deposit or withdraw. You need to call the functions inside a test program and show something like: “You have deposited/withdrew [amount] in the account [account number].”

**Problem Statement**

The program will create and call two ATM functions called deposit and withdraw, which will have the parameters account number, pin code and some value for money. The functions when called will print “You deposited/withdrew [amount] in/from the account [account number].”

**Solution Plan**

1. CREATE METHOD deposit (parameters: accNum, pinCode, amount)
   1.1. PRINT “You deposited”, amount, “in the account”, accNum

2. CREATE METHOD withdraw (parameters: accNum, pinCode, amount)
   2.1. PRINT “You withdrew”, amount, “from the account”, accNum

3. BEGIN MAIN
   3.1. PRINT “Please enter your account number: ”
   3.2. READ user input into variable accountNumber
   3.3. PRINT “Please enter your pin code: ”
   3.4. READ user input into variable pinCode
   3.5. PRINT “Enter 1 to Deposit money”
   3.6. PRINT “Enter 2 to Withdraw money”
   3.7. PRINT “Enter 0 to exit ATM application”
   3.8. READ user input into variable choice
   3.9. START SWITCH (choice)
      3.9.1. CASE 1
         3.9.1.1. PRINT “Please enter the amount toDeposit”
         3.9.1.2. READ user input into variable amount
         3.9.1.3. CALL METHOD deposit(accountNumber, pinCode, amount)
      3.9.2. CASE 2
         3.9.2.1. PRINT “Please enter the amount to Withdraw”
         3.9.2.2. READ user input into variable amount
         3.9.2.3. CALL METHOD withdraw(accountNumber, pinCode, amount)
      3.9.3. CASE 0
         3.9.3.1. PRINT “You’ve exited application. Thank you for using the ATM.”
   3.10. END SWITCH

4. END MAIN
Execution

1. Method deposit is created.
2. Method withdraw is created.
3. Main method begins
   3.1. The console prints: Please enter your account number:
   3.2. Suppose user enters 10011, accountNumber = 10011
   3.3. The console prints: Please enter your pin code:
   3.4. Suppose user enters 1234, pinCode = 1234
   3.5. The console prints: Enter 1 to Deposit money
   3.6. The console prints: Enter 2 to Withdraw money
   3.7. The console prints: Enter 0 to exit ATM application
   3.8. Suppose user enters 1, choice = 1
   3.9. START SWITCH (1)
      3.9.1. Since choice is 1 the program runs CASE 1
         3.9.1.1. The console prints: Please enter the amount to Deposit
         3.9.1.2. Suppose the user enters 5000, amount = 5000
         3.9.1.3. The console prints: You deposited 5000 in the account 10011
   3.10. Switch case ends
4. Main method ends

Evaluation

The program works as intended, But, there are some comments about the program:
1. maybe the program should enforce a digit limit for account number and pin code for more realistic representation of ATM functionalities.
2. Furthermore, there is no way for a user to rerun the ATM functionalities without rerunning the entire program. Maybe, a loop can be used to run the program as long as the user wants.
Building an ATM (Classes and Objects)

You made two functions of an ATM in the example before but in this one we are going to create an ATM from scratch. We know each ATM has location, bank affiliation and some maximum amount of money it can hold. Write a program that creates an ATM class with three attributes expressed above and the two functions deposit and withdraw we already made in the example before.

**Problem Statement**

The program will create an ATM class with three attributes location, bank affiliation, and the maximum money it can hold. The class will also have two methods deposit and withdraw.

**Solution Plan**

1. CREATE CLASS ATM  
   1.1. CREATE variable location  
   1.2. CREATE variable bankAffiliation  
   1.3. CREATE variable maxAmount  
   1.4. CREATE CONSTRUCTOR ATM (parameter: nLocation, nBankAffiliation, nMaxAmount)  
      1.4.1. Location <- nLocation  
      1.4.2. bankAffiliation <- nBankAffiliation  
      1.4.3. maxAmount <- nMaxAmount  
   1.5. END CONSTRUCTOR  
   1.6. CREATE METHOD deposit (parameters: accNum, pinCode, amount)  
      1.6.1. PRINT “You deposited ”, amount, “ in the account ”, accNum  
   1.7. END METHOD  
   1.8. CREATE METHOD withdraw (parameters: accNum, pinCode, amount)  
      1.8.1. PRINT “You withdrew ”, amount, “ from the account ”, accNum  
   1.9. END METHOD  
2. END CLASS

**Execution**

1. Class called ATM is created  
   1.1 – 1.3. Three class variables are created  
   1.4. The class constructor is created.  
   1.6. The class method deposit is created.  
   1.8. The class method deposit is created.  
2. The ATM class ends

**Execution**

It is not possible to say if the class works as expected. We must create a test program to create an ATM object using the constructor and test the functions.
Suppose you have a deck of cards with four suits: Clubs, Diamonds, Hearts and Spades and 13 ranks: ace, 2, 3, 4, 5, 6, 7, 8, 9, Jack, Queen and King. Build a program that when run prints a random card from the deck. For example, “Queen of Spades” or “2 of Hearts”

**Problem Statement**

The program will create a deck of card with four suits and 13 ranks, and then call a function to pick a random card from the deck.

**Solution Plan**

1. CREATE Suits[] = {“Club”, “Diamond”, “Heart”, “Spade”}
2. CREATE Ranks[] = {“Ace”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9”, “Jack”, “Queen”, “King”}
3. CREATE Deck[52]
4. COMMENT:
   4.1. I want the elements in the array deck to be from “Ace of Club” to “King of Spade”
   4.2. So, I am going to create two for loops, one of the Suits and one for the Ranks and combine the values and store them inside the array Deck.
5. START FOR i = 0 to suits.length -1
   5.1. START FOR j = 0 to Ranks.length -1
       5.1.1. Deck[i*Ranks.length + j] = Ranks[j] + “ of ” + Suits[i]
   5.2. END FOR
6. END FOR
7. CREATE variable randCard
8. ASSIGN a random value between 0 to 51 in randCard
9. PRINT Deck[randCard]

**Execution**

1. An array called Suits is created with the values “Club”, “Diamond”, “Heart”, “Spade”
2. An array called Ranks is created with the values “Ace”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9”, “Jack”, “Queen”, “King”
3. An array called Deck is created which can hold 52 values
4. The first for loop goes through the suits array. The second for loop goes through the ranks array and the combined values of each index is stored in deck array.
   In the first run i = 0, j = 0 so deck[0*13 + 0] = Ranks[0] + “ of ” + Suits[0] so, deck[0] = “Ace of Club”
   In the second run i = 0, j = 1 so deck[0*13 + 1] = Ranks[1] + “ of ” + Suits[0] so, deck[1] = “2 of Club”
5. A variable called randCard is created
6. Assume the random value between 0 to 51 is 34, so randCard = 34
7. Since Deck[34] = 8 of Hearts, the console prints: 8 of Hearts
Evaluation

1. The program works as intended.
2. However, the program is not flexible enough for anyone to create different types of decks of cards. Maybe the program can be modified so that any users can generate their own versions of card decks like UNO, Cards Against Humanity etc.
3. Also, I think it would be better if the random card pick part can be accessed as a separate function, in that way it can pick random cards from any set of cards.