Identifying Patterns

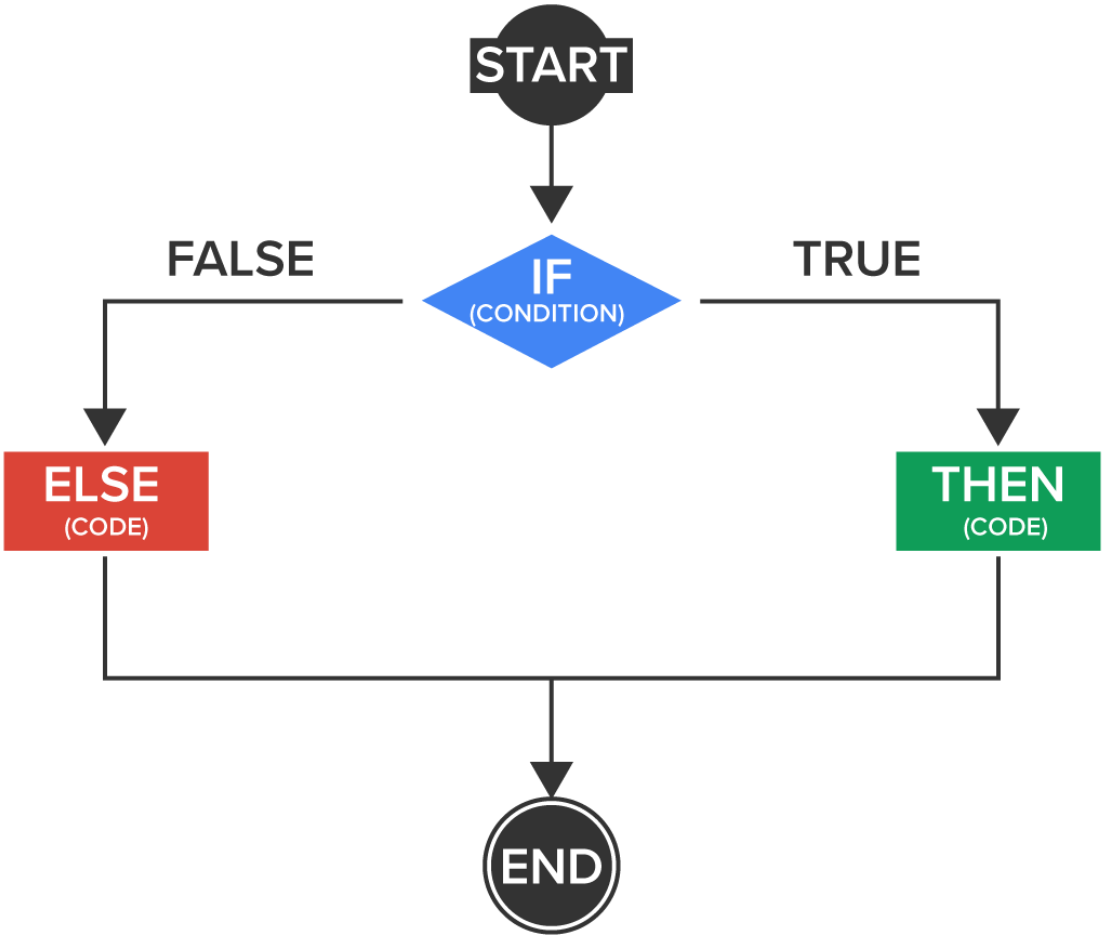
by Sophia

WHAT'S COVERED

In this lesson, you will learn about grouping steps into patterns. Specifically, this lesson covers:

1. Breaking Things Down

Now that you have the initial set of high-level steps, you will want to start breaking some of the steps down further. Let's take a look at the steps and start to work out patterns. In particular, you want to think about where the process comes to a fork in the road and what conditional statements may need to be used. You also need to think about which parts of the code will be repeated and could benefit from the use of loops. Are there pieces of functionality that work as a unit that could be turned into a method? Remember back to Unit 1 when you started to define the algorithms for the guessing game? You looked at ways to break things down through conditional statements and loops in the algorithms.



BRAINSTORM

To help with finding patterns with conditional statements, we should start by looking at situations where the program may need to branch. Typically, a good place may be after user input or after a calculation. If we're prompting the user for a value, we should think about whether we need to do different things based on the input. After a calculation, do we have different output based on the result of that calculation? Some of these calculations and results may have been performed already. Are there variables that contain the results of those previous steps that can be used again?

When it comes to loops, there are some common ways to use them. If you find yourselves needing to perform the same task over and over again, using a loop would help with that. An example could be a program that has a menu of options. You may have the user continuously being presented the menu of options until the user enters in an option to exit the loop. There are other situations that would benefit from using a loop such as iterating through a list or other data collection, or performing calculations on a set of criteria that requires a loop like calculating averages, reading data from a file, etc.

HINT

Remember that a for loop is one that would be used when you have a known number of times that a loop should be executed. A while loop should be used if the number of iterations isn't known before the start of the loop.

When thinking about methods that you may need to write, remember that separate methods generally would be useful if we have a set of statements that would be used in more than one place in a program. Think of all of those times when you needed input and provided output. Rather than writing the lines of code to perform those tasks, we can simply call a function or method that performs those repeating tasks. To make functions or methods useful to multiple programs, they should only perform a single simple task.

Remember in our game that we based things on the initial criteria and questions. Having those specifically defined is important since we want to ensure that we've captured each criterion and consideration.

EXAMPLE Here are the questions and answers you originally had:

Question	Answer
How is the game of craps played?	The game is played by rolling two standard six-sided dice. For each roll, the pips on the top sides of the dice are added up to get a total for the roll. If the first roll produces a sum of 7 or 11, the player wins the game on the first roll. If the sum of the dice on the first roll is 2, 3, or 12 (these values are called "craps"), the player loses the game on the first roll. If the total of the dice on the first roll is any of the remaining possible values (4, 5, 6, 8, 9, or 10), that value is called the "point" and the player continues rolling the dice until the point is rolled again or the roll produces a 7. If the player rolls the point again before rolling a 7, the player wins the game (no matter the number of rolls), but if the player rolls a 7, the game is lost.
What would the rules be for the game?	Player rolls two dice the first time. • If 2, 3, or 12 is rolled on the first time, the player loses. • If 7 or 11 is rolled on the first time, the player wins. • If any other number is rolled, that number becomes the point value or initial sum. If the player has not won or lost, roll again. • If the sum of the roll of the two dice is equal to 7, the player loses. • If the sum of the roll of the two dice is equal to the initial sum, the player wins. • If the sum of the two dice is anything else, roll again.

STEP BY STEP

We used a very simple format to define the steps for a specific attempt at a game, but within it, there may be many other steps involved. Let's revisit these now:

Scenario 1:

1. Roll two six-sided random dice for the first time, getting a 1 and a 2.
2. Add the values on the top of the two dice, getting 3.
3. The player loses the game.

Scenario 2:

1. Roll two six-sided random dice for the first time, getting a 3 and a 4.
2. Add the values on the top of the two dice, getting 7.
3. The player wins the game.

Scenario 3:

1. Roll two six-sided random dice for the first time, getting a 1 and a 4.
2. Add the values on the top of the two dice, getting 5.
3. The value is not 2, 3, 7, 11, or 12, so the game continues.
4. Roll the two dice again, getting a 3 and a 6.
5. Add the values on the top of the two dice to get 9.
6. That value is not equal to either 7 or 5, so the game continues.
7. Roll the two dice again, getting a 4 and a 3.
8. Add the values on the top of the two dice to get 7.
9. The player loses the game.

Let's break down the logic, as that's the crucial part of the program to determine how the game works. There will be other parts of the program that may also be needed, but we'll focus purely on the logic here.

One initial step is to determine what the variables may need to be. A good starting point would be to identify the variables that would be used to store values. This would include things like the player, the dice, the condition for whether the player won or lost, the point value, and the rolled value. The other part is to define patterns. With the full scenario (scenario 3), we'll see that steps 4–6 repeat again until the condition is met. In the first and second scenarios (1 and 2), the player could win right from the start.

Let's start to break this down into pseudocode.

Remember that pseudocode is English-like statements that describe the steps in a program or algorithm. It is a way to map out the logical steps but not get too lost in the syntax of the specific programming language.

Based on our steps above, we can start defining these variables and patterns that we see:

EXAMPLE

```
Set the rolled value to roll first dice + roll second dice
If the rolled value is equal to 2, 3, or 12
   Set player status to lose
Else If the rolled value is equal to 7 or 11
   Set player status to win
Else
   Set point value to rolled value
   Set the rolled value to roll first dice + roll second dice
   If the rolled value is equal to 7
      Set player status to lose
   Else If rolled value is equal to point value
      Set player status to win
   Else
      Set the rolled value to roll first dice + roll second dice
      If the rolled value is equal to 7
         Set player status to lose
      Else If rolled value is equal to point value
         Set player status to win
      Else
         Reroll again and continue over and over
```

This is what we have so far in terms of the logic for our program. We started to create some variables like “rolled value,” “player status,” and “point value,” although not using standard variable naming conventions since it is only pseudocode, but you can start to see where they would come in. You should also see that we’ve run into a situation where we have some repeated items that are obvious to create a loop for.

In revising using loops, it will look like the following:

EXAMPLE

```
Set the rolled value to roll first dice + roll second dice
If the rolled value is equal to 2, 3, or 12
   Set player status to lose
Else If the rolled value is equal to 7 or 11
   Set player status to win
Else
   Set point value to rolled value
   While the player status is not set, run the following
      Set the rolled value to roll first dice + roll second dice
      If the rolled value is equal to 7
```

```
    Set player status to lose
Else If rolled value is equal to point value
    Set player status to win
```

Now that you have the logic in place, it should be easier to build this further, one layer at a time. In every program, there are aspects where we may have some of those items that are going to need to be expanded on, but at this point, we're looking strictly at the logic.

THINK ABOUT IT

With a roll of the dice, think at a high level. What should this actually look like in code?

When it comes to your own program, you'll want to do the same thing in terms of the overall logic. In the next lesson, we'll start expanding this to another level.

THINK ABOUT IT

This is a great starting point, but you now have some additional questions. For example, how do we store the information about the dice or even how many sides are on the dice? Someone who plays craps may know that they are using two six-sided dice, but there are four-sided, eight-sided, 10-sided, 12-sided and even 100-sided dice as well. You will need to consider this as part of the actual program. You will also need to think about what a die consists of and what would be needed. For a die, you will need to know how many numbers there are and how the die is rolled. In our case, each die has six sides so the number needs to be between 1 and 6.

2. Guided Brainstorming

Now that you have an example, it's time to start with yours! Break down the steps and find where those repeating patterns are. Are there certain conditions or examples where you can see those items being repeated? What variables exist that you would want information stored in? These are some things to think about with each of those steps to really break them down further.

Looking back at the Drink Order program from Unit 1 again, we had a few drink choice options. We defined the following four potential runs of the program based on the selection in the last lesson:

EXAMPLE

Hot Water:

1. User selects water.
2. User selects hot water.
3. Output water, hot.

Cold Water:

1. User selects water.
2. User selects cold water.
3. User selects ice.
4. Output water, cold, ice.

Coffee:

1. User selects coffee.
2. User selects decaffeinated.
3. User selects milk.
4. User selects sugar.
5. Output coffee, decaffeinated, milk, sugar.

Tea:

1. User selects tea.
2. User selects green tea.
3. Output tea, green.

THINK ABOUT IT

This would end up being a bad example for an entry for Part 3 of the journal, as this consists of specific choices rather than what the choices were. Remember that you want to break down the steps prior to actually developing the code. The logic is the part that you want to consider since that generally is the most complex

part. Although we already had a solution in place for the code, you should find it helpful to see what that would look like as pseudocode.

If you remember from this completed program from the end of Unit 1, you had the initial menu structure of what can be selected:

EXAMPLE

```
Water
   Hot
   Cold
      Ice/No Ice
Coffee
   Decaffeinated or Not?
   Milk or Cream or None
   Sugar or None
Tea
   Green
   Black
```

Using this, you can follow the same structure as well. You have three main entries as a first step:

EXAMPLE

```
Ask user to enter in “water, coffee, or tea”
Store input into drink
If drink is equal to water
...
Else If drink is equal to coffee
...
Else If drink is equal to tea
...
```

This gives us that first set of criteria that we can build on. Each of the items based on the drink selection has its own selections, so there is no overlap between them. Next, we can start to add the components for the water:

EXAMPLE

```
Ask user to enter in “water, coffee, or tea”
Store input into drink
Store drink in outputString
If drink is equal to water
```

```
    Ask user to enter in hot or cold
    Store input in heat
    If heat equal to hot
        Add hot to outputString
    Else If heat equal to cold
        Add cold to outputString
        Ask user to enter in ice or not
        If ice is yes
            Add ice to outputString
        Else
            Add no ice to output String
Else If drink is equal to coffee
...
Else If drink is equal to tea
...
```

Next, you can work on the coffee part. Each of the choices could overlap, which is different than with the water choices. In the water choices, the ice selection would only be there if cold was selected for the heat. However, with the choices if coffee is selected, each of the selections are independent from one another.

This is something that you want to think about when designing algorithms and finding patterns:

EXAMPLE

```
Ask user to enter in “water, coffee, or tea”
Store input into drink
Store drink in outputString
If drink is equal to water
    Ask user to enter in hot or cold
    Store input in heat
    If heat equal to hot
        Add hot to outputString
    Else If heat equal to cold
        Add cold to outputString
        Ask user to enter in ice or not
        If ice is yes
            Add ice to outputString
        Else
            Add no ice to output String
Else If drink is equal to coffee
    Ask user to enter in decaf or not
    Store input in decaf
```

```
    If decaf equal to Yes
        Add decaf to outputString
    Ask user to enter in Milk, cream, or none
    Store input in milkCream
    If milkCream equal to milk
        Add milk to outputString
    Else If milkCream equal to cream
        Add cream to outputString
    Ask user to enter in sugar or not
    Store input in sugar
    If sugar equal to Yes
        Add sugar to outputString
Else If drink is equal to tea
```

Notice with our algorithm that we only looked at the instance where items like decaf are set to yes. We don't worry about the "no" input because for any of these, we're not adding any items to the outputString.
Lastly, we'll just have to complete the algorithm for the tea. This will be easier, as it's only doing a check between green and black tea:

EXAMPLE

```
Ask user to enter in "water, coffee, or tea"
Store input into drink
Store drink in outputString
If drink is equal to water
    Ask user to enter in hot or cold
    Store input in heat
    If heat equal to hot
        Add hot to outputString
    Else If heat equal to cold
        Add cold to outputString
        Ask user to enter in ice or not
        If ice is yes
            Add ice to outputString
        Else
            Add no ice to output String
Else If drink is equal to coffee
    Ask user to enter in decaf or not
    Store input in decaf
    If decaf equal to Yes
        Add decaf to outputString
    Ask user to enter in Milk, cream, or none
```

```
    Store input in milkCream
    If milkCream equal to milk
        Add milk to outputString
    Else If milkCream equal to cream
        Add cream to outputString
    Ask user to enter in sugar or not
    Store input in sugar
    If sugar equal to Yes
        Add sugar to outputString
Else If drink is equal to tea
    Ask user to enter in teaType
    Store input in teaType
    If teaType equal to green
        Add green to outputString
    Else If teaType equal to black
        Add black to outputString
print outputString
```

This concludes this part of the algorithm around this program.

TRY IT

This is a time for you to take your different examples/scenarios and work through each to build out the algorithm and determine what patterns that you have. Think about each scenario but also consider if you may have missed any scenarios, as there may have been other factors or paths that are missing.

REFLECT

Decisions, decisions. To solve most problems will involve a series of decisions. These decisions may be within your code, or they may be just about how you decide to write it, i.e., where and how to use a loop or a method. Determining the decision points within your code will help you to break it down into various branches or paths that you can then follow through to check if it's doing what you expect it to do. Deciding on how to write your code in the best way takes practice that you've now been exposed to. Are you iterating through some variable as you're performing a task? This is probably a good place to write a loop. Are you doing a specific task over and over again? This is a great place to call a method that you write once and use whenever needed. Try to apply what you've seen in the examples above to the algorithm for your problem.

SUMMARY

In this lesson, you started to **break things down** by taking the demonstration program and looking at ways to build an algorithm using conditional statements and loops in basic pseudocode. You started to determine where we would need variables to store input, conditions, and status of the player. You then looked for patterns to see if there were opportunities to simplify and improve the algorithm. Again, you

used an old program from Unit 1 in the **Guided Brainstorming** section to set up an algorithm using the expected user menu to break down the choices.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/