Identifying the Patterns

by Sophia

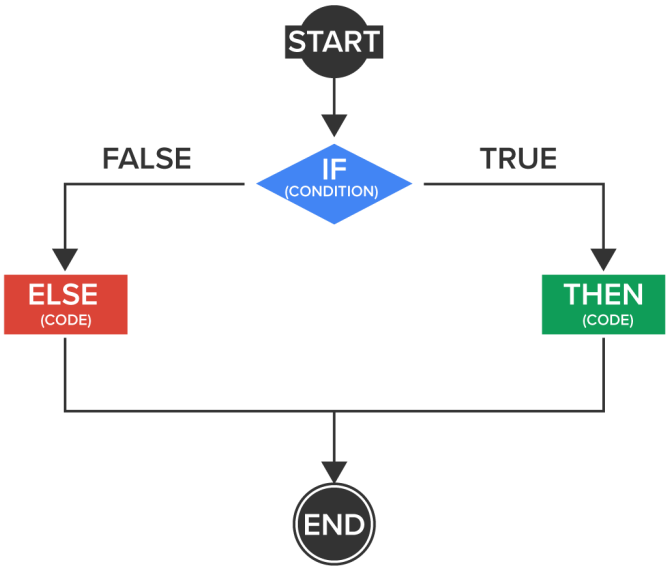
WHAT'S COVERED

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

1. Breaking Things Down
2. Guided Brainstorming

1. Breaking Things Down

Now that we have the initial set of high-level steps, we will want to start breaking some of the steps down further, if it makes sense, to start defining what the overall program should do. Let's take a look at the steps and start to define patterns. In particular, we want to think about what conditional statements may need to be used, what statements could benefit from the use of a loop, as well as what pieces of code may be repeated which could be broken down into a function or method. Remember back to Unit 1 when we started to define the algorithms for the guessing game. We 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 example 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.

When it comes to loops, there are some common ways to use them. If we find ourselves 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. We may have the user continuously being presented with a menu of options until the user enters 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 type, 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 prior to the loop. A `while` loop should be used if it is an unknown number.

With methods or functions, this 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 in past code examples 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, 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. Here is the example of questions and answers we originally had:

EXAMPLE

- How would the game of casino craps be played?
  - With the game of craps, the game is made with players putting bets on the first roll of the sum of two dice, also known as the come-out roll. In this round, the players must decide whether the dice will land on 7 or 11 (pass bet) or 2, 3, or 12 (don't pass bet). The round ends if the total dice value is 7 or 11 (known as a natural), or 2, 3, or 12 (craps). If the total of the dice value rolled is a 4, 5, 6, 8, 9, or 10, then that number becomes the point and starts the next stage of the game. The shooter then rolls the dice again and repeats until the shooter rolls a 7 or the point number, which ends the craps game. If a 7 is rolled, they "sevens out" and lose. If they rolled the initial point score, they win the round.
- 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.

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 plays. 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 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 pseudocode is English-like statements that describe the steps in a program or algorithm. It is a way to layout logical steps but not get too lost in syntax.

So, 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 we can start to see where they would come in. We 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:

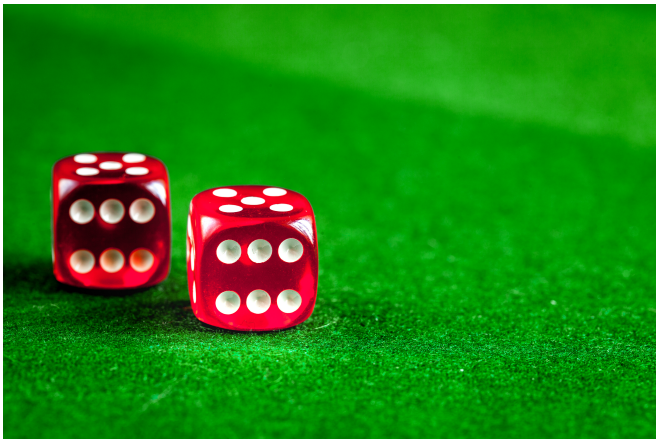
EXAMPLE

```
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 we 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. For example, with a roll of the dice, think at a high level. What should this actually look like in code? We'll get into that later on.

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 we 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 a six-sided die but there are four-sided, eight-sided, 10-sided, 12-sided, and even 100-sided die as well. We need to consider this as part of the actual program. We need to think about what a die consists of and what would be needed. For a die, we 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. By default, we can set the value to 1.

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 this down further.

In 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

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

If you remember this completed program from the end of Unit 1, we 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, we can follow the same structure as well. We 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, we can work on the coffee part. Each of the choices and selections 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 is independent of 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

Directions: 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 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 may have been missing.

SUMMARY

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

Best of luck in your learning!