# Working an Example

*by Sophia*

WHAT'S COVERED

In this lesson, you will learn about working through the input, processing an output of a program for a problem. Specifically, this lesson covers:

## Table of Contents



# 1. Starting With Input/Output


In the prior lesson, you decided on a problem to solve via programming. Breaking down the individual overall steps to actually solve the problem using a program can be challenging. First, it can be helpful to determine what the goal of the program is and what the output should be. To help with that process, look at our initial example and build from there. As you start to work through this example, try to think about how you'd start to break down your program. Think back to Unit 1's Programming Mindset lesson and how we looked at input, processing, and output. Then, as part of the Thinking Through Examples lesson, we looked at real-world examples like making orange juice. Remember how we needed to break the process down into specific steps.

### THINK ABOUT IT

The main logic of the program is the goal, at this point. This part of the process does not yet require you to break down the problem into specific programming steps. This part determines what steps we should include and how we should order them. As a developer, you won't worry about the syntax of the language to be used during this stage of the process. The goal is to be able to use any programming language to create the output that we want to produce. The input aspects should be straightforward since you want to simply determine what and from where it is received. This could be user inputs from the screen/console or actual data from files.

In the demonstration program, the only input from the user would be the player's name and the number of games to play. This simplifies the example, but there may be other types of input that you will need to consider later, such as input values that would be used for selections in a menu or reading data from a file, and so forth.

### BRAINSTORM

Let's start with the idea and the questions and answers that we had in the prior lesson to use as an initial guide of what the output should be. Let's focus purely on the questions that are related to the output.

| Question | Answer |
| --- | --- |
| If multiple games are played in one run of the program, what information should be tracked? | Ideally, I would like to be able to enter a number of games to play. In part, that would be based on my gambling budget for the day. After determining the number of games to play, I would like to keep track of how many rolls of the dice, on average, it took to win and how many it took to lose. In addition, I'd like to know what my winning percentage (of the games played) is. |
| What information should be stored? | I'd like the player's name and statistics about the games I played stored in the file. The file can just keep track of this information for each run of the program. |
| What kind of output does the user want? | I would like to see on screen the number of games played, how many times out of those games I won, how many times I lost, what the number of rolls per game was, and the percentage of games won. |
| Where would they like the information tracked? | In this case, I would like the information being tracked in a file for every time the program is run. |

There are some key criteria that we can pull from here. We know that the output to the file should consist of the following:

↱ EXAMPLE

- Player's name
- Date and time for the run of the program
- Total number of wins
- Total number of losses
- Percentage of games won

Now, what our friend/user wasn't specific about was the formatting of the output. We could follow up on this with our user to identify if there's a specific format or if they just wanted the information presented as is. Some may want it in a specific table format while others won't care how it is formatted. This could be a sample output of what it could look like:

↱ EXAMPLE

Date and time = 06/16/2022 14:23:37.
The total number of wins is 2.
The total number of losses is 3.
The winning percentage is 0.4.

Once we have an idea of what it could look like, we can use that to build a foundation for what the output will

look like and work backwards towards some of the variables that we would need to store the output:

↗ EXAMPLE

Output the name of the player.
Output the date and time.
Output the total number of wins.
Output the total number of losses.
Output the winning percentage.

THINK ABOUT IT

When it comes to your program, think about what the output should look like. Would the output be produced throughout the run of the program or would it be produced all at the end? Would the output be sent to the screen or to a file or perhaps both?

---

# 2. Moving to Processing

The next part you will have to think about is the processing within the program. There's a lot in a program that you may have to think about for this part, but let's focus on the core of the processing. This is a crucial element and depending on your program, you may have a core part of the processing but also potentially have supplemental parts as well. You will want to focus on one part at a time. Get that part working and then move on to the next. The logic should be nailed down to what the core of the program is. Let's go back to our example program, pull the key questions, and build off of that. Focus on the simplest process that produces the correct results. Most of the time, unnecessary complexity will come back to haunt you.

THINK ABOUT IT

| 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 are the rules of the game? | • Player rolls 2 dice the first time.<br>  • If 2, 3, or 12 is rolled on the first time, the player loses.<br>  • If 7 or 11 is rolled on the first time, the player wins.<br>  • If any other number is rolled, that number becomes the point value or initial sum.<br>• If the player has not won or lost, roll again.<br>  • If the sum of the roll of the two dice is equal to 7, the player loses.<br>  • If the sum of the roll of the two dice is equal to the initial sum, the player wins.<br>  • If the sum of the two dice is anything else, roll again. |

Interestingly, the way that the second question was answered helps us determine some of the functionality already. One of the best ways to start to break down a program is by taking the responses and determining a complete run step by step.

First, let’s go through one example of a game where the player wins or loses in the first roll:

↱ EXAMPLE Player loses a game on the first roll:

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.

↱ EXAMPLE Player wins a game on first roll:

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.

These can cover the first instance of the game, but challenges can arise when the player has not rolled a 2, 3, 7, 11, or 12 that allows them to win or lose in the first roll. Let's go through an example of a full game where the player doesn't win or lose the game on the first roll:

↱ EXAMPLE Player loses on a few rolls of the dice:

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.

Now we have a full set of steps where the dice rolling had to occur three total times. You should see from this example that there's some consistency that we can start to bring together now. You'll notice that each of these steps is clear and has a specific order.

HINT

As you work through your own program, you'll want to define each item step by step, as they'll help you work through and identify those repeating patterns. If you only submitted the first example with a winner after the first roll, you'd miss out on the other examples to expand the game in a realistic fashion.

---

# 3. Adding a Second Journal Entry

THINK ABOUT IT

At this point, we are ready to submit our second journal entry for the Touchstone. After breaking down a few examples/scenarios step by step (sample runs of the craps game), we started to see some patterns that we can apply in our next lesson. We want to include steps for a few outcomes to ensure we identify all possible outcomes.

## 3a. Bad Example of Journal Entry for Part 2

Again, we will start off with a not so good example of a journal entry first:

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.

REFLECT

This is a correct step-by-step example; however, it does not go through all outcomes that are possible

### 3b. Good Example of Journal Entry for Part 2

Here are a few possible step-by-step examples.

Scenario 1: Player loses on first roll:

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: Player wins on first roll:

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: Player loses on a few rolls of the dice:

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.


As you can see, the good example does include distinct examples of possible outcomes. The more you can break out all possible outcomes, the easier it is to identify any patterns that you can carry over to Part 3 of the Java Journal, which is creating pseudocode for patterns and algorithms.


If you preview the Example Java Journal Submission document, you will see this was added as the entry to Part 2.

---

# 4. Guided Brainstorming

Given the coding project that you are trying to solve, it's a good idea to try and think about how it should be ordered. If you remember our Drink Order program at the end of Unit 1, you started to break down more of the specifics of the program by looking at the menu structure of what was possible in a selection for drinks.

↱ EXAMPLE

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

You can use this as part of defining how the program works, to help with the step-by-step breakdown that we are looking for in this lesson. In looking at these options, the user should be first prompted with the choice of water, coffee, and tea. Based on each of those selections, there are additional prompts that are unique to them. This will show how the program is broken down with the individual steps that would be included.

↷ EXAMPLE In this case, we have three main key items as the first input:

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

This is great to start, but it doesn't help us see some of the other possibilities for water.

↷ EXAMPLE What happens if the user selects cold water since there's an extra input?

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

It's better that we have both of those. At this point, it may seem like the program would cover most of the task already, but this isn't the case. You want to ensure that you've gone through each of the steps of the possibilities in as much detail as possible.

↷ EXAMPLE Go down each set of possibilities:

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

That covers a set for the coffee. There weren't any scenarios where there was a separate option like we had with the cold water.

↷ EXAMPLE We have one more option to go for tea:

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


THINK ABOUT IT

Now this covers all of the selections and would be a great journal entry with all four of those examples. Addressing them individually would not be enough since this wouldn't cover the key differences between each option.


TRY IT

**Directions:** Now would be a good time for you to create these different examples/scenarios and think about each step in a specific order for your program. Think about each question and step one at a time to ensure that you've covered each key example. Ask yourself if you've covered the main types of examples or scenarios that would be needed. Review the example of a good entry for Part 2 in the Example Java Journal Submission document and add your journal entry for Part 2 to your Java Journal.


REFLECT

Our bodies, as well as our minds, are input, processing, and output (I-P-O) machines. We eat food as an input

to our bodies. We then process this food to give us energy to perform a task as an output (as well as other by-products). Any physical task we do is performed in this way, and now translating into programming, this I-P-O machine is even more applicable since the input, processing, and output are all limited to data. As you're thinking about the problem you've selected, what is the input (food) for it? Is it a series of numbers, text representing a set of names, ID numbers, etc., or a combination of the two? And is it all in a file, or is a user typing in this information? What will the output (task) look like? For the craps game, it's the player's name, date, and game statistics. For the drink order machine above, it's a set of commands to create your desired drink. For your problem, it will be whatever you want your code to accomplish when it is complete. Finally, the processing within your code will take your input and turn it into your output. It's really that simple. Your code will make a great I-P-O machine.

## SUMMARY

In this lesson, you learned that breaking down the individual overall steps to a future program can be challenging. What can be helpful is to first determine what the **input and output** should be. Once that is identified, you can **move into the expected processing** of the program and look at some possible examples that we can break down and detail out into steps. This can help us identify patterns early in the design process. You had an opportunity to see some specific steps for a **bad example of journal entry for Part 2** of the demonstration program, contrasted with a **good example of journal entry for Part 2**, and to **submit that as our second journal entry**. Finally, you used an old program from Unit 1 in the **Guided Brainstorming** section to reemphasize the need to break down the steps of a program, including all aspects of expected outcomes.

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/**