Unit 4 Tutorials: Project

INSIDE UNIT 4

Java Touchstone Overview

by Sophia

WHAT'S COVERED

In this lesson, you will learn about all aspects of the final Touchstone, including requirements, grading, helpful aids, and submission details. Specifically, this lesson covers:

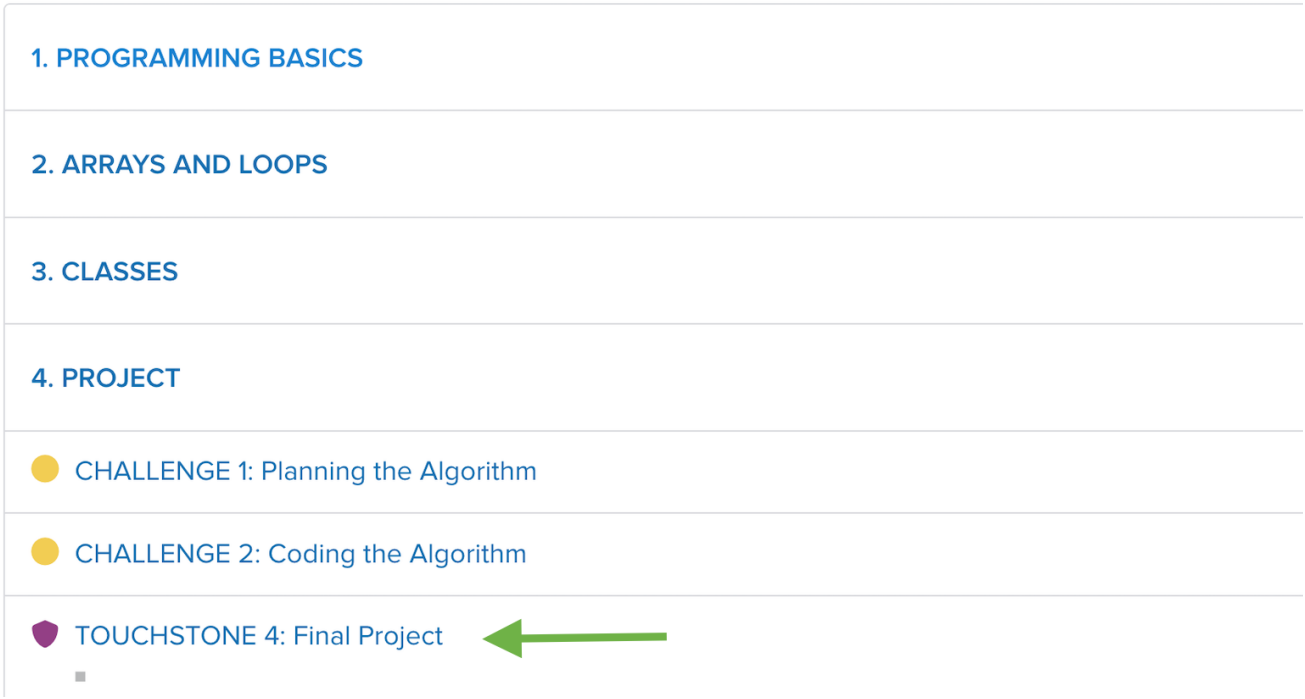
1. What Is the Touchstone?

UNIT 4 — TOUCHSTONE 4: Final Project

Not Submitted | Submitted | Scored

The Touchstone is the pinnacle of this course. Up to this point, we have discussed Java syntax, and you have coded many programs. Now is the time to shine with what you've learned. The Touchstone is a human-graded activity that allows you to submit an actual program that you design, build, and test. Don't worry, we will be with you every step of the way. There are documents to help you and the lessons in Unit 4 will use an example program that can inspire your own program. Using what you have learned over the entirety of this course, it is time to design and code your own program!

Units



The Touchstone consists of one (1) submission. You will be submitting a Java Journal at the conclusion of Unit 4. However, this journal has six (6) entries. The easiest way to become familiar with this Java Journal is to visit the actual Touchstone page. The Java Journal's information is included as the last item in Unit 4 (see image above).

BIG IDEA

On the TOUCHSTONE 4: Java Final Project's page, you will see the following:

- Assignment directions with a link to the Java Journal Template
- Rubrics which will be used for grading the project
- Overall project requirements
- An additional resource that includes an Example Java Journal Submission document

TRY IT

Directions: Now that you know more about the Touchstone it is time to visit the Touchstone page and perform the following tasks:

1. Visit the Unit 4. PROJECT - TOUCHSTONE 4: Final Project page.
2. Carefully read through this page to get a better understanding of the overall project, the assignment directions, how grading works, the requirements for the project, and an additional resource. We suggest that you keep this page open while you follow the lesson, so you can easily reference each part.
3. Download the Touchstone JAVA Journal Template document and be prepared to open it on your device later in this lesson.

REFLECT

You may be thinking, “Touchstone, templates and journals, oh my!” These are some more new concepts to figure out. The good thing is, though, you are ready. You've learned not only a lot about Java throughout this course, but also gained knowledge on some great coding concepts and methodologies that are applicable to any type of problem solving. The Touchstone work you are about to enter is just a way to capture what you now already know how to do—code your own program—and the templates and journals will document this journey. As you might hear before a classic boxing match—let's get ready to rumble, or rather, code!

---

2. What Is My Program?

THINK ABOUT IT

So, what program am I expected to design and code?

That is something you will think about as you move through Unit 4. Throughout this unit, you can follow along

as you design and code a demonstration program. You will also create journal entries for this demo program throughout the unit. The lessons are designed to help you brainstorm your own program. There are inspirational questions for each part of the journal as well as the “Guided Brainstorming” section in each Unit 4 lesson to give you some additional thoughts.

Creating a program can seem daunting, but as you take each step through Unit 4, you will be adding to your ideas and given some thought-provoking questions to help. Your idea and program can be as simple or complex as you wish; just make sure to keep your journal entries tied to the requirements for each part of the journal. This can be a program for yourself, a friend, a family member, or anyone! You will be using a casino game as an example and bringing back thoughts and ideas on the drink program that we created together back in Unit 1. So, you can design and code your program to do anything you want. Maybe it's a program to convert measurements or collect movie reviews, or maybe it's a receipt saver or a financial planner. Really, the sky's the limit. However, remember to make sure it is something you feel you can design and build because it is a graded project.

It may seem obvious, but it needs to be stated clearly: please do not submit your Touchstone project using the demonstration program.

This project should be your own idea and code. Try to start simple and add as you progress throughout the unit. The Touchstone should only be submitted once you feel good about all of the entries you added to your journal. Feel free to rewrite, recode, and reestablish a new problem you would like to solve. Your project is only graded when you submit it on the Touchstone page in the course.

---

3. The Java Journal Template Document

The Java Journal Template document can be found in the Touchstone page of the course. Visit the Unit 4. PROJECT - TOUCHSTONE 4: Final Project page. In the Assignment section, you will notice a link for the Touchstone Java Journal Template document. Download this document and open it. Please read the directions at the top, if you haven’t already.

3a. What Is in the Java Journal Template Document?

The template provides you with space for each part (entry) of the project. Below the first page, you will see each entry task along with the requirements that need to be addressed for a good score. When you are entering your journal entries throughout Unit 4, make sure you review the requirements for the entry since they are aligned with the rubric. Below the requirements for each entry, there are inspirational questions to help you start your brainstorming. Depending on the program you choose to design and build, they may or may not influence your ultimate entry submission.

Directions: Open your Touchstone Java Journal Template document and skim the tasks, requirements, and inspirations for the six (6) journal entries.

3b. When Do I Add an Entry to My Java Journal Template Document?

As you progress through the lessons in Unit 4, you will see a demonstration program being designed and coded in steps that match the Java Journal Template document. Feel free to type in and build the demonstration program in Replit to gain additional practice and insights, but remember you need to add journal

entries for your own originally designed and coded program. Throughout Unit 4, you will see the demonstration program being entered into the Example Java Journal Submission document as entries. As you are working with your project, you will be encouraged to add entries about your project to your Java Journal.

3c. What Is the Example Java Journal Submission Document?

When you visit the Additional Resources area on the Touchstone page, you will see a link to the Example Java Journal Submission document. This is essentially a properly filled out Java Journal. This is the journal that is being created in lessons throughout Unit 4 with the demonstration program. You will have the opportunity to see some good journal entries as well as some not so good entries as you progress through Unit 4.

TRY IT

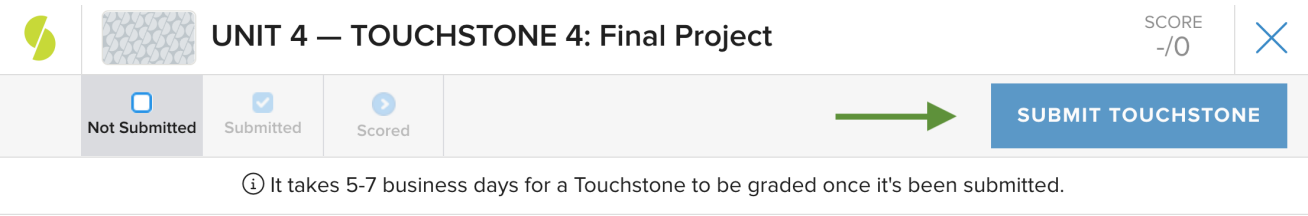
Directions: If you haven't already, now is a good time to download or view the Example Java Journal Submission document to get an idea of a well-laid-out Touchstone submission. You will find this document linked to the Unit 4 - Touchstone 4: Final Project page under Section D. Additional Resources.

3d. When Do I Submit My Java Journal?

You can only submit the Touchstone (your Java Journal) once, so make sure you have completed all the required entries (six in total) before submitting your Java Journal. Make sure all entries have good grammar and spelling in the earlier entries and that your code has been correctly pasted in the later entries. Make sure your Replit share link works and has been added to both the top page and as the last entry. Finally, ensure your name is added and the date you are submitting the Touchstone is on the first page.

3e. How Do I Submit My Java Journal?

Once you have your Java Journal Template filled out completely, you can submit the journal using the “Submit Touchstone” button at the top right of the Touchstone page.



HINT

Remember, you only have a one-time submission, so ensure your journal is complete before submitting. The final lesson also contains the directions to submit your journal.

4. Requirements and Grading

This Touchstone is worth 100 points. That breaks down to about 30% of the total course score, so make sure you take your time and review the requirements for the Java Journal. They are found on the Touchstone page in Section C. Requirements. You can see how your journal will be graded after submission by visiting Section B. Rubric. Each journal entry (Part 1 – 6) is a criteria row and each column is a level of performance that has an associated grade percentage. As you move from the left to the right, the percentage grade decreases. If the journal entry is missing any or all of the requirements listed on the Java Journal Template document, the grade received will decrease. Make sure you are familiar with the grading rubrics and review your journal entries

before final submission.

5. Unit 4 Overview

Each lesson of Unit 4 may consist of some review or mention of learning from a previous lesson; they are meant to prepare you for the Touchstone submission.

Here is a synopsis of each lesson in Unit 4. You will notice that with each lesson, we will continue with the completion of a demonstration program, and each will include a “Guided Brainstorming” section that will include ideas and extra examples to aid you in brainstorming your original idea that will eventually be coded into a program that you can submit for the Touchstone:

Challenge 1 (Planning the Algorithm)

1. **Java Touchstone Overview**. This lesson is the overview of the Touchstone, your program, Java Journal Template document, the Example Java Journal Submission document, and all requirements and grading of the Touchstone.
2. **Identifying a Problem to Solve**. This lesson will introduce you to the Unit demonstration program and walk through the questions you may need to ask to start the process of identifying a problem to solve. At the end of this lesson, we will demonstrate a good vs. bad example of a PART 1 entry of the Java Journal Template document. This would be a good time to add your journal entry as well.
3. **Working an Example**. In this lesson, we will continue to use the demonstration program to set up a series of steps required to solve the problem. Remember this is still early on in designing a program, but listing out the steps in a logical order is an important one. At the end of this lesson, we will again demonstrate a good vs. bad example for a PART 2 entry of the Java Journal Template document. This would be a good time to add your second journal entry as well.
4. **Identifying the Patterns**. In this lesson, we will be searching for patterns in the steps we listed out in the previous lesson. We will also be grouping steps and starting to look for any patterns that we can replicate.
5. **Forming the Algorithm**. In this lesson, we take the patterns and steps that we have identified and form algorithms based on them. Then, we can generalize our step-by-step algorithms into pseudocode. At the end of this lesson, we will demonstrate a good vs. bad example of a PART 3 entry of the Java Journal Template document. Once again, you will be reminded that while forming algorithms and pseudocode is fresh in your mind, this would be a good time to add your third journal entry.

Challenge 2 (Coding the Algorithm)

1. **Translating to Code**. In this lesson, we will take the demonstration program's pseudocode from the previous lesson's journal entry and start referencing code elements for the patterns.
2. **Writing the Program**. In this lesson, we will complete our coding of the demonstration program to include all the conditionals, loops, classes, etc., that we learned in this course.
3. **Testing As You Go**. In this lesson, we will see our demonstration program go through some tests that we learned about in past lessons. We will try to "break" the demonstration program. This is the QA (quality assurance) section, and since we do not have a dedicated QA individual/team, we need to do the test ourselves. At the end of this lesson, a good vs. bad example will be identified, and we will enter the PART 4 entry into the Java Journal Template document. As we have before, you will be reminded to add your fourth entry to your journal.

4. **Commenting Your Code**. In this lesson, we will see the demonstration program get the needed commenting so that the grader, or anyone for that matter, will be able to tell what our program is doing even if they are not Java-savvy. At the end of this lesson, a good vs. bad example regarding commenting will be shared, and we will finish the entry for PART 5 in the Java Journal Template document. This is also a chance to add comments to your program and enter it into your journal as the fifth entry.
5. **Course Wrap-Up**. In this lesson, we will obtain a Replit Share link and add that link as the final PART 6 entry into the Java Journal Template document.

SUMMARY

This lesson was set up to be an informational page for all things Touchstone-related. We discussed that this **Touchstone** is a human-graded project that involves the design and programming of a solution to an identified problem. We also discussed that this **program** can be as simple or complex as we feel comfortable with. The project submission will be based on the **Java Journal Template document**, which contains six journal entries to be written out during Unit 4 and submitted at the end. Also contained in the Java Journal Template document are directions, requirements, and inspirational questions to ask ourselves for **when we add each entry** of the journal. We reviewed how this project will be **graded** and some basic submission **requirements**. Finally, we identified some **support materials** including an **example submission** and reviewed what is expected to be covered and recommended tasks in each of Unit 4's lessons.

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/**

Identifying a Problem to Solve

by Sophia

WHAT'S COVERED

In this lesson, you will learn how to get started on defining a problem to solve with coding. Specifically, this lesson covers:

1. Getting Started With an Idea

For this lesson, you will work on defining a problem or a project that you can build a program around. Throughout this class, you have been given code to work through. Although you acquired some skills using this approach, it's a good idea to work on some code for a program that you devise yourself. You should challenge yourself to exercise as much as you can what you've learned and create a project that reflects it. This will not only help develop your coding ability but also the skills that you need to break down projects into pieces that lead to workable solutions. This will also allow you to have improved conversations with others about what you've built.

BRAINSTORM

Getting started can be a challenge. A good program should perform a task that makes life easier in some way. This could be a task for yourself, someone you know, or for a larger audience. If you think about the things that you do on a regular basis, is there some way that you could automate a part of that with a program? It's worth taking the time to write down all of your ideas. Even if an idea seems ridiculous or impossible to do, you could think about what ways it could be improved on. Another way to brainstorm ideas is to look at other programs that you may use. Are there things that they do that could be improved upon, or do they have areas that are lacking? Could they be done more effectively?

THINK ABOUT IT

It's best to try and think of a basic project with limited scope since it's easier to start simple and then add features one at a time. If you start with an overly complex project, you may not have a working project at the end. In preparation for this type of development, you'll be working on documenting the features and functionality that you intend to achieve with the project. The more details about what the program should do, provided early on, the easier it will be to start to write the underlying code.

BIG IDEA

Think back to the first unit where we defined what a program consists of. You noted that most programs need to have input, processing, and output to be useful. The input is the information that a program needs from the user or other sources, such as a file the program will pull data from. The processing includes all of the calculations or transformations performed on the input. The output is the result of the processing and may be presented to the user in the console or written to a file.

Generally, as a programmer or developer, you would be providing a piece of software to an end user, though in some cases, you may be the end user of the program that you create. One of the first steps is to understand what the users are looking for and what they want the program to accomplish. For example, a DJ (Disc Jockey) may want a program that allows the selection of songs to be played in a specific order and produce a play list. A photographer may want to take a set of photos and keep track of the names of the individuals in each photo. It is important to get into the mindset of the customers and consider what they expect.

THINK ABOUT IT

When planning a program, you generally first think about what the result of running the program should be (usually what output the program should produce). This is what the end user is looking for. Once you know what the result should be, you can identify what input is needed and start planning what processing steps are needed to get to that end result.

2. An Example Problem to Solve

Let’s start by identifying a potential problem to solve. Imagine a person who is going to Las Vegas and would like to learn more about what the odds are of winning a dice game like craps. This is a game that involves some strategy, skill and luck. This person would like to get some practice to have a better sense for the odds and what it's like to play the game.

THINK ABOUT IT

This doesn’t sound like a bad place to start, but there are additional questions that need to be asked, since this request isn’t a fully defined problem.

- Is the person interested in a program that plays a single sample game or multiple games?
- What are the rules for the game?
- What information should be tracked?
- What kind of output should be produced?

Typically, it is important to ask questions and get the customer to tell the story of what the program should do and how it should work.

EXAMPLE

Here are some typical questions and responses.

Question	Answer
Would they be interested in having a single sample game to see how the program works before playing multiple games?	Yes, I would like to see how the game works before deciding how many games to play.
	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

How is craps played?	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 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, reroll again.
If multiple games are played, what information would need to 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. In knowing the number of games to play, I would like to know how many rolls on average it took to win and how many it took to lose. In addition, I’d like to know what my winning percentage was at the end of the day.
What information would they like to store?	On a daily basis for each set of games, I’d like the statistics about the games I played stored in a file. The file can just keep track of each time a game is played.
What kind of output would they want?	I would like to see on screen the number of times the game was played, the total number of rolls, how many times out of those games I won, how many times I lost, and the percentage of wins.
Where would they like the information tracked?	In this case, I would like the statistics about the games to be placed in a file for every time I played the game.

As part of logging these questions and answers along with other details, it can help to place these details into

documentation.

↗ EXAMPLE

The documentation may include some of the following:

- Forms (paper forms, online forms, surveys, etc., to get data for input)
- Reports (how the output should look like, raw data items, items that need to be calculated)
- Files (data files, .csv files, etc.)
- Sample results (example runs of what the program is expected to do)
- Mockups (design of the application, input fields, workflow of the program)
- Existing tools (anything that the client/user currently uses to help workaround the program that they need)
- Any other item that comes from the client/user

Throughout this process, you'll be following the same type of format where you'll be documenting your progress for the Touchstone using the template to help as a guide.

At this point, start with your formal idea that you plan to go through the process of designing and developing. If you don't have an idea, you can always ask those around you if they have a recommendation or a need for a program. Once you have your idea, pose (and document) as many of the questions that you can think of to figure out how the program needs to work and include the answers to those questions. Use the examples given to help as a guide with your questions, but each idea/problem will have different questions that you may need to ask.

---

3. The First Journal Entry

For the initial steps, it's important to present the questions and answers that would help us eventually define a solution.

↗ EXAMPLE

Even if we have the right questions to ask, imagine if you had the following answers to our prior questions:

Question	Answer
Would they be interested in having a single sample game to see how lucky they would be?	Not sure.
How would the game of casino craps be played?	Like the game found in the casino.
What would the rules be for the game?	Roll dice and see who wins.
If they played multiple games, what information would they need to track?	All information.
What information would they like to store?	All information.
What kind of output would they want?	The results of the game.
Where would they like the information tracked?	In a folder.

In looking at the answers above, there's not much information that can be derived for how the program would actually work. If you do not get adequate responses, you should continue to ask questions until you get all of

the necessary information that you need to start creating the algorithms for the pseudocode on the project. The more details that can be provided at this stage, the easier it will be to develop the program.

THINK ABOUT IT

At this point, we are ready to submit the first journal entry for the Touchstone. Based on the questions we asked and the answers our friend provided, we have information that can address the requirements of our task, which was to state a problem that we are looking to solve.

What we would not want to submit for our journal entry is the inadequate answers to our questions that we just witnessed. For example, this would be a bad entry.

3a. Bad Example of Journal Entry for Part 1

EXAMPLE

“The problem to solve is to create a program that plays casino craps like one found in a casino. My input data would be to roll dice and see who wins. Our friend would like to track all game information and then store that information. Our output would be the results of the game.”

REFLECT

This entry does not adequately define the problem or present questions and answers that we developed during our brainstorming session to help us refine the problem.

3b. Good Example of Journal Entry for Part 1

EXAMPLE

"Casino craps is a dice game in which players bet on the outcome of the roll of a pair of dice. The problem to solve is to help a user better understand the odds of winning and losing at casino craps at an actual casino. I will create a simulation of the craps game to solve this problem. When executed, the program will first provide a sample game as an example for the user. Second, the program will ask the user to input the number of games to play. The program will play the requested number of games. After the last game is played, the program will output (display) the results of the games. The program will store a record of the simulations in an external log file. By playing the game multiple times and reviewing the statistics, users will get a better understanding of their chances of winning.”

REFLECT

A better entry would break down the requirements of the entry and add as much detail as possible to truly define the problem and expected solution based on answers to our questions. If you preview the Example Java Journal Submission document, you would see this was added as the entry to Part 1.

---

4. Guided Brainstorming

There are a lot of different, complex potential problems that you may have, but it is important to try and limit the complexity of the program to something that you can create as your first application. This could be a program that stores email addresses for a newsletter. It could be a recipe program that stores a list of recipes, ingredients, and directions. It could be a program that does conversions of different measurements between each other. The possibilities are endless, but what’s important is to try and fully define the initial problem, ask the right questions, and provide the answers. Remember that the foundation of any program is based on the input.

THINK ABOUT IT

If you're trying to create a recipe program, you should ask the question of what a program like that would do. There are so many different ways that you can have a recipe program. For example, you may have existing recipes that are just being displayed from a file. You could even extend it further to have a menu that would prompt if the user would like to add a recipe or list the recipe based on a name that has been entered. You could even extend it further to have the program display images of the recipes as well. The options and selections are endless, but you will want to try and limit how far you go with this program to ensure that you will complete it.

Using the casino craps game that you defined, we could have two different players interactively play the game and take in bets rather than just showing the wins and losses. Having an interactive game would be a lot more fun, especially if some animation of the dice being rolled would also be shown. However, having a more interactive program could also make it a lot more complex as well as take longer to fully test out. If we've done a good job in defining and bounding our program, it will make it easier to test and even expand on later.

TRY IT

Directions: Now would be a good time for you to define your problem. Think about a program you would like to build to solve that problem. Ask the questions to gather the requirements for this task. Review the example of a good entry for Part 1 in the Example Java Journal Submission document and add your journal entry for Part 1 to your Java Journal.

REFLECT

Craps and other casino games may not be your forte, but you don't have to be an expert on them to really learn from the demonstration program given. In fact, we experience a significant amount of learning from games as children, and it's great to carry this into adulthood. Focus on the process regardless of the context. At this stage, we are identifying and defining a problem. The old saying, "knowing that you have a problem is the first step to solving it," is somewhat applicable here. You have to know, and thus understand, what the problem is to solve, or for us, code it. The questions and answers about the craps game define what we need to do so that we truly understand what we need to code and bound the problem so that we can reach our goal in a reasonable amount of time.

SUMMARY

In this lesson, you explored how to **get started with an idea** for a program to build. If the program is for ourselves or for someone else, there are questions we need to ask and answers we need to collect to make sure we are preparing for the correct input, processing, and output that we need to do in the program. You considered an **example problem that you might want to solve**. A casino craps simulation will be used throughout Unit 4 as an example program. Moving on to **the first journal entry**, you first saw a **bad example of journal entry for Part 1**, followed by a **good example of journal entry for Part 1**, noting that it should be as comprehensive as possible to define and characterize the input, what the program will solve for, and expected output. Finally, you explored some **guided brainstorming** to examine some additional opportunities for ideas while thinking about your own program to build.

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/**

Identifying Patterns

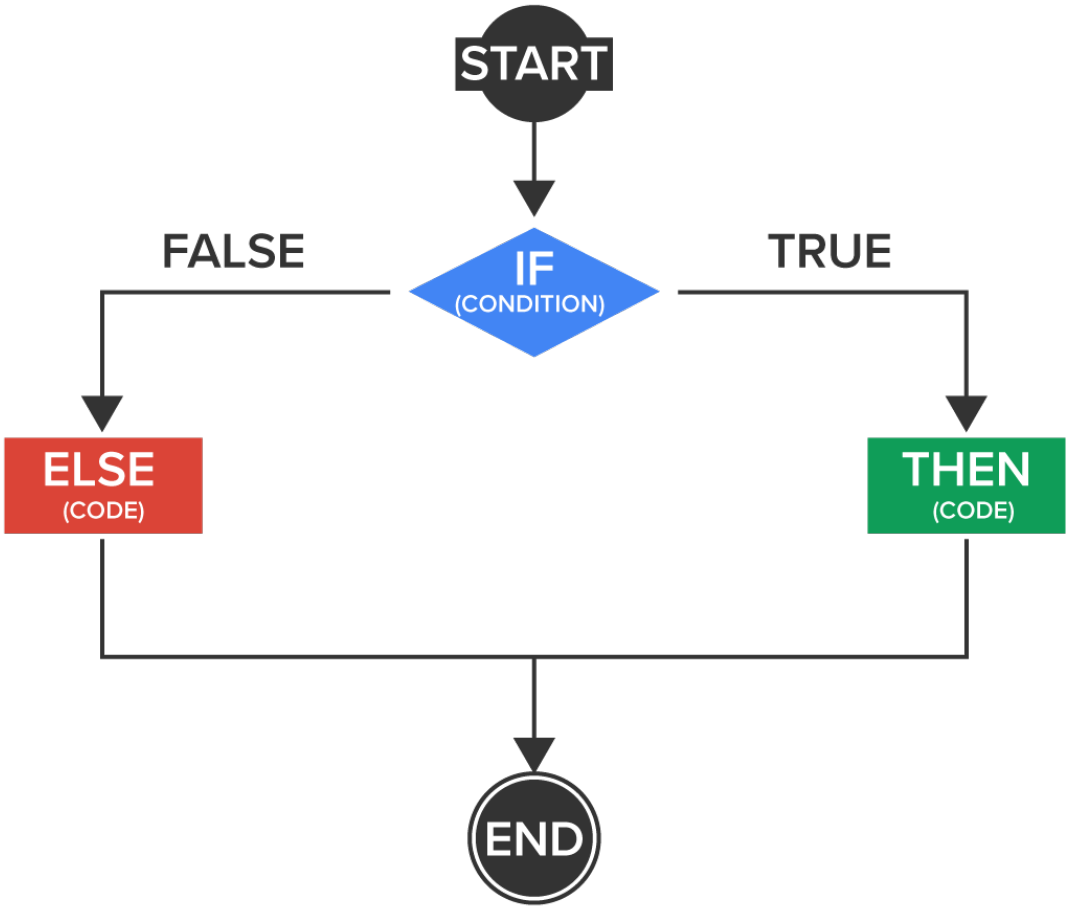
by Sophia

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.
	Player rolls two dice the first time. • If 2, 3, or 12 is rolled on the first time, the player loses.

What would the rules be for the game?	• 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.

BIG IDEA

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.

CONCEPT TO KNOW

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/**

Organizing the Algorithm

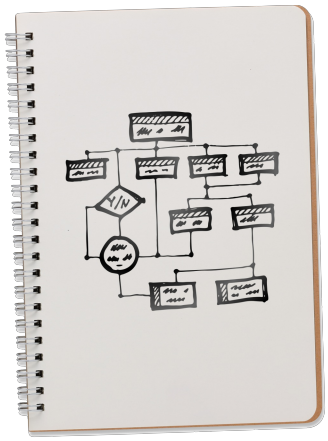
by Sophia

WHAT'S COVERED

In this lesson, you will learn how to plan algorithms to ensure that all of your coding is incorporated. Specifically, this lesson covers:

1. Planning the Algorithms

There will likely be some common algorithms that can be used or added to your code since they represent common ways of approaching a problem. Rather than reinvent the wheel each time, using these algorithms can be useful to avoid issues in your code. Let's look at some of the different repeating patterns that could be used.



You previously started with some key patterns related to the program. Although this is a great starting point, you do have a lot more to expand on as well.

You will need to be able to think about how the entire program works, as we currently only have the logic around a single attempt of the game:

EXAMPLE

```
Get randomly generated values for roll of dice
If the sum of the rolled dice 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
```

There is more to the program that we need to consider beyond the logic of a single play. If you remember, there's specific output that our friend wanted in the program that would be used to track the process, including:

↗ EXAMPLE

- Total number of wins
- Total number of losses

You know already that our friend wanted to be able to track the data across multiple iterations of the game. This would need to be added to the existing logic.
You could use a function that we'll call "play," in which we'll just have the following pseudocode put into it:

↗ EXAMPLE

```
Function play()
    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, to help track statistics for multiple plays of the game, including wins, losses, and number of rolls, we would need to place the logic into a loop. We would wrap some of that logic into a larger program by first defining the variables to handle each of those items.

↗ EXAMPLE

```
Set Wins = 0
Set Losses = 0

Set Games Played to 0
```

Next, we can define the loop of how the game would be played. The logic of most of the program would be in the play function, but the added code just fits into the key criteria:

↗ EXAMPLE

```
Input Times to Play from user
Loop until Games Played = Times to Play
   Add 1 to Games Played
   Call function play()
   Get if the player won or lost
   Get the number of rolls
   If the player won
      Add 1 to Wins
   Else if player lost
      Add 1 to Losses
   Add the rolls values to the Total of Loss Rolls
```

Once done with defining the looping of the game, we have to calculate the final values.

You will want to indicate the calculation for the winning percentage, which would be by taking the number of Wins and dividing it by the Games Played:

↱ EXAMPLE

```
Winning Percentage = Wins / Games Played
```

Putting this together now, we have a few more parts but note that this is just a starting process for the design of the program. It's not meant to include all of the functional elements, but it is more of a guide for the algorithms that the program should take:

TRY IT

Directions: Before moving on, see if you can pseudocode out what steps you believe this program will need. Once you are done, see how close you are to the code below:

↱ EXAMPLE

```
Function play()
   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
```

```
Main Program
   Set Wins = 0
   Set Losses = 0
   Set Total of Win Rolls = 0
   Set Total of Loss Rolls = 0
   Set Games Played to 0

   Input Times to Play from user
   Loop until Games Played = Times to Play
      Add 1 to Games Played
      Call function play()
      Get if the player won or lost
      Get the number of rolls
      If the player won
         Add 1 to Wins
      Else if player lost
         Add 1 to Losses


   Average Number of Rolls Per Win = Total of Win Rolls / Wins
   Average Number of Rolls Per Loss = Total of Loss Rolls / Loss
   Output results
```

And finally, we add the winning percentage:

```
Winning Percentage = Wins / Games Played
```

REFLECT

The pseudocode above now has a main program and a function called play. You can think of main as the baseline structure of the program that sets everything up by getting what it needs from the input, and collects the processed data to place it in the output. The core processing for the game all occurs in the function play. How can you plan out your algorithm? Is there a structure you should put in place to just capture the input and gather the output? Should your core processing be placed in a function? Maybe you will need multiple common functions. These decisions are yours to make, but you have the knowledge and experience now to make good ones.

---

2. Adding the Third Journal Entry

As you start building your logic, you want to have as much of the algorithm detailed out as possible to make the conversion to code more easily performed. Remember that the pseudocode is meant to describe the application at a high level. The pseudocode should be as optimized as you can have it to be, before conversion to the actual programming language.

2a. Bad Example of Journal Entry for Part 3

A bad entry would not fully break down and identify the patterns to add conditional statements, loops, or functions where necessary.

A bad entry could be more based on a single run of the program similar to what we created as part of the previous lesson, like:

↱ EXAMPLE

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.

REFLECT

The problem with this type of code is that it has not been fully broken down and uses specific examples. The reason why? This pseudocode is using specific values vs. variables. It doesn't use conditional statements or loops to optimize the code. It doesn't improve on the prior entry by adding to or improving on the patterns.

2b. Good Example of Journal Entry for Part 3

The following would be considered as a good entry for the journal:

↱ EXAMPLE

```
Function play()
    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

Main Program
```

```
Set Wins = 0
Set Losses = 0
Set Total of Win Rolls = 0
Set Total of Loss Rolls = 0
Set Games Played to 0

Input Times to Play from user
Loop until Games Played = Times to Play
   Add 1 to Games Played
   Call function play()
   Get if the player won or lost
   Get the number of rolls
   If the player won
      Add 1 to Wins
   Else if player lost
      Add 1 to Losses

Average Number of Rolls Per Win = Total of Win Rolls / Wins
Average Number of Rolls Per Loss = Total of Loss Rolls / Loss
Winning Percentage = Wins / Games Played
Output results
```

REFLECT

Although you could have a lot more code to break things out like defining the output of the results, this gives us enough information to move on.

If we preview the Example Java Journal Submission document, we see this was added as the entry to Part 3.

---

3. Guided Brainstorming

As you start building your program, you'll need to start thinking about the bigger picture and the different aspects of it. With the pseudocode, you are only worrying about the logic, but you'll want to think about where the functionality needs to be enhanced and expanded on. Each program will have some unique criteria or a menu structure.

THINK ABOUT IT

Think about where the exceptions may be. For example, are you working with files? If so, how do you ensure that the program can handle issues with the files not being found or openable? These are things to keep in mind as you design the code. Can you walk through the program from start to finish with all the examples? If so, you're ready to move on to start designing your code.

If you have trouble here, think about the algorithms, and try to break things down into small parts. From here, you should be able to identify the variables that need to be used and what needs to be

calculated afterwards.

With what we reviewed in the prior lesson, this was well defined already:

↷ 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
```

In this case, we wouldn’t need to change anything unless we wanted to add in a menu structure or allow multiple drinks per order. Those things are elements that you want to think about in your own program.

TRY IT

Directions: At this point, you should have a good idea of what your program should look like. Take the time to think about the big picture of the program and go beyond just the core logic of the program. Also look at what the completed program should look like. Review the example of a good entry for Part 3 in the Example Java Journal Submission document and add your entry for Part 3 to your Java Journal.

SUMMARY

In this lesson, you continued **planning for the algorithm** using simple pseudocode and the demonstration program. Based on what our friend wishes for this program, you included some additional logic to store some statistical information. You created a single attempt of the game play and placed it into a play function. Then you added the functionality to create the statistics and allow for multiple game plays by calling the play function you created. Then, you had an opportunity to see both a **bad example** and a **good example** for our **third journal entry**. In the **Guided Brainstorming** section, you identified another example of good pseudocode formation.

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/**

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:

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.

	• Player rolls 2 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.

What are the rules of the game?	• 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.

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.

REFLECT

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.

HINT

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/**

Translating the Code

by Sophia

WHAT'S COVERED

In this lesson, you will learn about the initial steps to translate pseudocode to code. Specifically, this lesson covers:

1. Coding Framework

Looking back at the pseudocode that you created for the demonstration program, you first defined the output the program should produce. Now you will need to think about the key pieces of functionality. The overall game is made up of one or more dice rolls. The DiceRoll class represents a roll of a pair of dice. It needs to generate two random integers in the range from 1 to 6. In Java (like many other programming languages), this involves using a random number generator. This requires a bit of setup which we will demonstrate below. The class needs to keep track of the values of the two dice and also needs to report the sum of the dice. A comment outline for the DiceRoll class might look like this:

EXAMPLE

```
// Attribute to hold the rolls of the 2 dice
// Attribute to hold the sum of the 2 dice
// Attribute for the random number generator from the Random class
```

As previously noted, you would need a single roll of a pair of dice, and the values will not change, nor will the sum of the dice change. This means that you can use the `DiceRoll()` constructor to generate the numbers and get the sum. An outline for the constructor using comments might look like this:

EXAMPLE

```
// Create the random number generator
// Get the random number for the 1st die and store it
// Get the random number for the 2nd die and store it
// Add the values and store
```

Using these comments as the scaffolding, the DiceRoll class ends up like this (with a couple of additional comments to document the code) :

```
import java.util.Random;  // Needed for Random

public class DiceRoll {
```

```
  // The values of the 2 dice (an array of int)
  private final int[] die = new int[2];
  // Attribute to hold the sum of the dice
  private int sum;
  // Attribute for random number generator instance
  private Random randomNumberGenerator;

  public DiceRoll() {
    // Create the random number generator
    randomNumberGenerator = new Random();
    // Get the random number for 1st die in range from 1 to 6 & store in array
    die[0] = randomNumberGenerator.nextInt(6) + 1;
    // Get the random number for the 2nd die in range from 1 to 6 & store in array
    die[1] = randomNumberGenerator.nextInt(6) + 1;
    // Add values of dice and store
    sum = die[0] + die[1];
  }

  // Accessor method to get sum of dice
  public int getSum() { return sum; }

  // String representation of the dice roll
  public String toString() {
    return "Dice: " + die[0] + "  " + die[1] + "  Sum = " + sum;
  }
}
```

TRY IT

Directions: Try adding these comments in a file called DiceRoll.java in Replit. Then add the lines of code that implement the steps outlined in the comments. The DiceRoll class won't run on its own, so compile it using javac DiceRoll.java first.

REFLECT

This will reveal if there are syntax errors or other problems that prevent it from building. Remember that if javac is successful, there won't be any messages in the console.

The `Game` class represents the results from a collection of dice rolls that result in a win or loss for the user. The `Game` class also keeps track of the player's name and the date and time at the start of the game.

Here is a list of comments corresponding to the attributes needed in this class:

EXAMPLE

```
// Player's name
// The date & time
// Collection for dice rolls
// Value of "point" if 1st roll is not a win or loss
// The result of the game ("Win" or "Lose")
```

Since it is best to have the code in the driver class handle the input and output, the player's name will be passed to the `Game()` constructor as a parameter.

The comment "outline" of the `Game()` constructor might look like this:

↷ EXAMPLE

```
// Assign player's name to the attribute for the player's name
// Set the attribute for the date/time to the current date/time
// Create a collection to hold the DiceRoll objects that make up the game
```

The central action in the Game class will go in the `play()` method.

A set of notes about the process for this method is captured by comments like these:

↷ EXAMPLE

```
// Call the constructor for the DiceRoll class to get a roll of the dice
// Store the roll in the collection of rolls for the game
// Check the sum of the roll
   // If the sum is 2, 3, or 12, the result is set to "Lose"
      // Set value so that rolls do not continue
   // If the sum is 7 or 11, the result is set to "Win"
      // Set value so that rolls do not continue
   // Other sum for the roll is the value of the "point"
      // Keep rolling the dice
      // If sum of roll is 7, set status to "Lose"
         Set value so that rolls do not continue
      // If sum roll is point, set status to "Win"
         Set value so that rolls do not continue

   // When the play() method is done, it returns the collection of DiceRoll objects

   // The Game class will need appropriate accessor ("getter") methods:
   // get the game result ("Win" or "Lose")
   // get the number of dice rolls for the game
   // get the name of the player
   // get the date/time of the game
```

Using these comments as a framework, here is the implementation of the `Game` class. Since the collection of dice rolls in a game will vary and may contain duplicates, the code uses an `ArrayList<DiceRoll>` to hold the rolls of the dice.

```
import java.time.LocalDateTime;  // Needed for date & time
import java.util.ArrayList;  // Needed for ArrayList to hold
```

```
public class Game {
  private String playerName = "";
  private LocalDateTime gameDateTime;
  // ArrayList to hold dice rolls for player
  private ArrayList<DiceRoll> diceRolls;

  private int point = 0;
  private boolean keepRolling = true;
  // String to hold game result msg: "Win" or "Lose"
  private String gameResult = "";

  public Game(String playerName) {
    gameDateTime = LocalDateTime.now();
    this.playerName = playerName;
    diceRolls = new ArrayList<DiceRoll>();
  }

  public ArrayList<DiceRoll> play() {
    // 1st roll for player
    DiceRoll roll = new DiceRoll();
    diceRolls.add(roll);
    int sum = roll.getSum();

    // Check for "craps" & resulting loss
    if(sum == 2 || sum == 3 || sum == 12) {
      gameResult = "Lose";
      keepRolling = false;
    }

    // Check for immediate win
    else if(sum == 7 || sum == 11) {
      gameResult = "Win";
      keepRolling = false;
    }
    else {
      // Sum of dice is now player's "point"
      point = roll.getSum();
    }

    // Loop for subsequent rolls
    while(keepRolling) {
      roll = new DiceRoll();
      diceRolls.add(roll);
      if(roll.getSum() == 7) {
        gameResult = "Lose";
        keepRolling = false;
      }
      else if(roll.getSum() == point) {
        gameResult = "Win";
        keepRolling = false;
      }
```

```
    }
    return diceRolls;
  }

  public String getResult() {
    return gameResult;
  }

  // Get the number of dice rolls for game
  public int getRollCount() {
    // Number of DiceRoll objects in the ArrayList is number of rolls
    return diceRolls.size();
  }

  public String getPlayerName() {
    return playerName;
  }

  public LocalDateTime getGameDateTime() {
    return gameDateTime;
  }
}
```

TRY IT

Directions: Add the comments shown above to a file named Game.java in Replit. Then add the lines of code that implement the steps outlined by the comments. As you might expect, the Game class won't run on its own. You will need to compile it using javac Game.java first.

REFLECT

Running the program will reveal if there are syntax errors or other problems that prevent it from building. Remember that if javac is successful, there won't be any messages in the console.

With the `Game` class (composed of `DiceRoll` objects) in place, we can outline the process that the code in the application's `main()` will work through. The driver class is the `Craps` class, since it represents a craps session for a player that is made up of one or more games. This class will need to track statistics, handle user input, provide output, and do the logging to a file. Here are some comments to note the key variables needed in the `main()` method. These are variables in the body of the main() method, not class-level attributes:

EXAMPLE

```
// win count
// loss count
// roll count
// log file
// Scanner for reading user input
// player's name
// Game object
// ArrayList for rolls in game
```

Now, here are the key steps in the process that the code in main() will work though:

↷ EXAMPLE

```
// Create log file
// Declare Scanner to read from System.in
// Prompt for input of player name
// Read the player's name and store it

// Run the sample game
    // Create Game object
    // Call the play() method and store result in ArrayList
    // Loop through the ArrayList to print out dice rolls in the game
    // Print the result (Win/Lose) for sample game

// Prompt for number of games player wants to play
// Read input as integer
// Track if number entered is valid
    // Repeat prompt/input if entry is not valid

// Loop to play requested number of games
    // Create a new Game object
    // Call the play() method & store result in ArrayList
    // Loop through ArrayList to print out dice rolls
    // If getResult() returns "Win" add 1 to win count
    // Else add 1 to loss count
    // Format date and time for output
    // Assemble String with statistics
    // Calculate percentage of wins by dividing win count by number of games played
    // Print statistics data to console
    // Write statistics data to log file
```

TRY IT

Directions: Add the comments above from the demonstration program to Replit and follow along with the writing of the code.

Next, it can be helpful to identify the variables that we are planning to use in our program and start to define them in Java.

CONCEPT TO KNOW

Generally, it's a good idea to initialize numeric values to 0 or its intended value and strings to an empty string or their intended values. This way, we won't be surprised by incorrect values being displayed.

TRY IT

Directions: Start by coding with the `main()` method first:

```
int winCount = 0;
int lossCount = 0;
// Total number of dice rolls for game
int rollCount = 0;

// log file
File logFile = new File("game.log.txt");
// Scanner for reading user input
Scanner input = new Scanner(System.in);
// String variable for player name
String playerName  = ""; // Later gets value from input.nextLine();
```

TRY IT

Directions: Then, start working on the other parts of the program:

```
// Prompt for user name
// Read the input as a String
// Play sample game so player can see how it works
// Display result from the sample game
// Prompt for number of games player wants to play
//  Read input as integer
// Track if number entered is valid
   // Repeat prompt/input if entry is not valid

// Loop to play requested number of games
   // Create a new Game object
   // Call the play() method & store result in ArrayList
   // Loop through ArrayList to print out dice rolls
   // If getResult() returns "Win" add 1 to win count
   // Else add 1 to loss count
   // Format date and time for output
   // Assemble String with statistics
   // Calculate percentage of wins by dividing win count by number of games played
   // Print statistics data to console
   // Write statistics data to log file
```

REFLECT

Notice we have a method called `play()`. This method is provided by the `Game` class, so the code in `main()` will need to instantiate a `Game` object to use to call this method. While this step is specific to an object-oriented language and may not always be reflected in pseudocode, the pseudocode above does reflect this step.

TRY IT

Directions: Next, let's code the first input that we need the user to enter.

↗ EXAMPLE

```
// Prompt for user name
System.out.print("Enter Player Name: ");
// Read the player's name and store it
playerName = input.nextLine();
```

The next step in the coding process is to play the sample game by constructing a Game object, calling its play() method, loop through the ArrayList that play() returns to show the dice rolls, and then display the result. This ends up being a bit more complex than the initial pseudocode indicates, but it's not uncommon to find that more steps (or smaller steps) are needed as the code gets written.

↷ EXAMPLE

```
// Run sample game
System.out.println("\nRunning Sample Game: ");
Game game = new Game(playerName);
// ArrayList to track roles of dice in a game
ArrayList rolls = game.play();
// Loop through DiceRoll objects in ArrayList & display
for(DiceRoll roll : rolls) {
    // println() will automatically call DiceRoll object's toString() method
    System.out.println("\t" + roll);
}
// Call Game object's accessor methods to get result (win/lose) and # of rolls for game
System.out.println("\nResult of Sample Game: " + game.getResult() + " in " +
        game.getRollCount() + " roll(s)\n");
// Add an extra blank line in output for spacing
System.out.println();
```

Just as some steps in the pseudocode may break down into multiple steps when the code is written, it is also possible that not all comments will translate into separate statements. The pseudocode and comments derived from the pseudocode are meant to be guides and not to dictate the code exactly.

You will see the rest of the code for the main() method in the next lesson. It is possible, though, to take the code for main() written so far and test it.

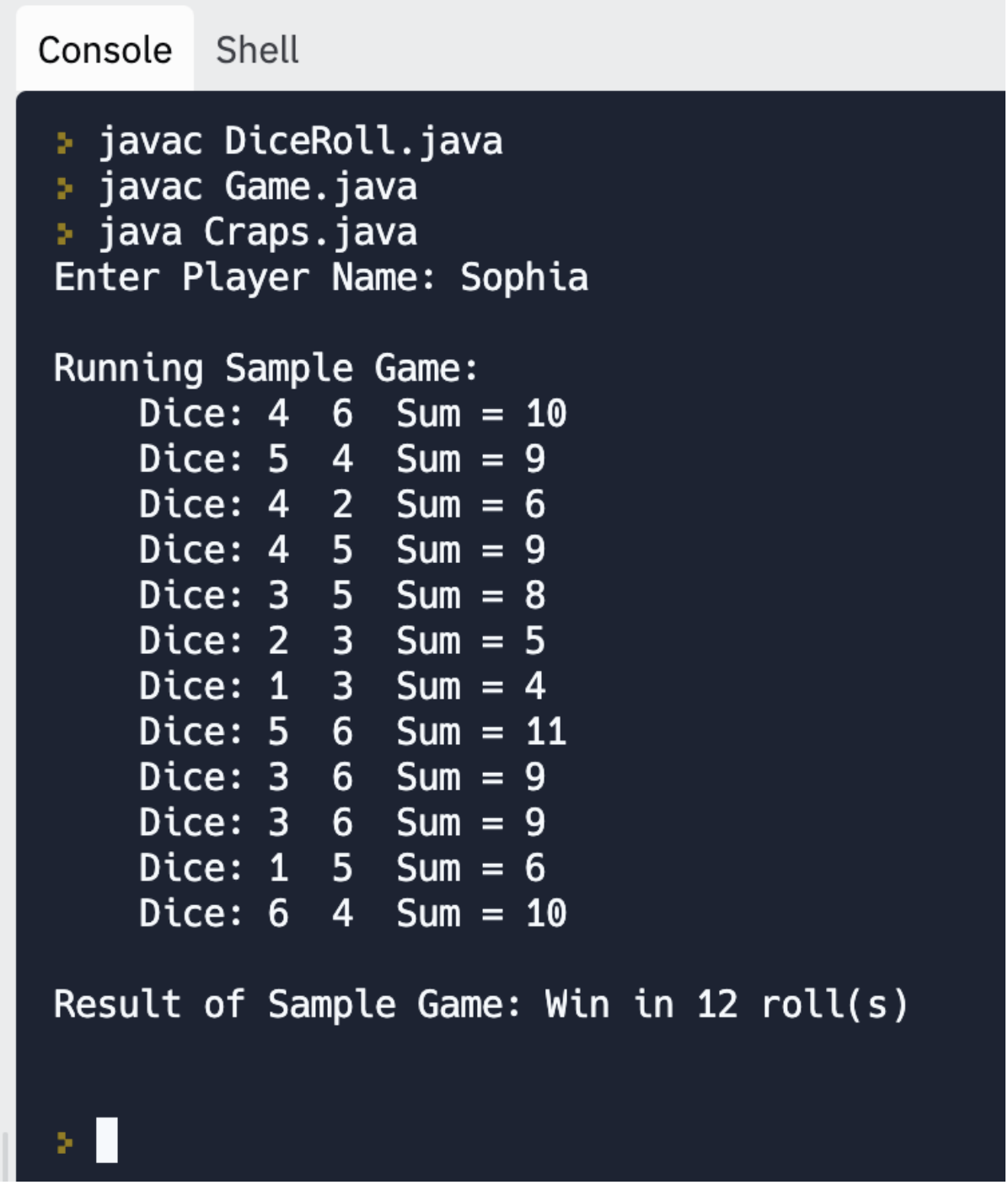
TRY IT

Directions: In Replit, type the following code into a file named Craps.java. Remember that the DiceRoll and Game classes need to be compiled using javac first in order to use those classes in this code. Don't forget to add the import statements for the different library classes.

```
import java.io.File;
import java.util.Scanner;
import java.util.ArrayList;

public class Craps {
  public static void main(String[] args) {
```

```
    int winCount = 0;
    int lossCount = 0;
    // Total number of dice rolls for game
    int rollCount = 0;

    // log file
    File logFile = new File("game.log.txt");

    // Scanner for reading user inputCra
    Scanner input = new Scanner(System.in);
    // String variable for player name
    String playerName  = ""; // Later gets value from input.nextLine();
    // Prompt for user name
    System.out.print("Enter Player Name: ");
    // Read the player's name and store it
    playerName = input.nextLine();

    // Run sample game
    System.out.println("\nRunning Sample Game: ");
    Game game = new Game(playerName);
    // ArrayList to track roles of dice in a game
    ArrayList<DiceRoll> rolls = game.play();
    // Loop through DiceRoll objects in ArrayList & display
    for(DiceRoll roll : rolls) {
      // println() will automatically call DiceRoll object's toString() method
      System.out.println("\t" + roll);
    }
    // Call Game object's accessor methods to get result (win/lose) and # of rolls for game
    System.out.println("\nResult of Sample Game: " + game.getResult() + " in " +
          game.getRollCount() + " roll(s)\n");
    // Add an extra blank line in output for spacing
    System.out.println();
  }
}
```

The result of running this code should look something like this (though of course, the rolls will be random and vary in number):



Core functionality will be expanded on in the upcoming lesson.

---

2. Guided Brainstorming

The goal of these initial steps is for us to understand how to break down our pseudocode into smaller pieces so that you can implement them part by part. It's a good idea to identify all of the variables needed first. As that is completed, you can then move on to breaking down any functions or methods that we have. In this step, you would use the `pass` statement to create a function or method without fully implementing the functions.

As you start to write our code, begin to remove comments for logic that you have fully implemented. Also, consider the bigger picture. Beyond the pure logic of the program, think about the program that you plan to build.

Remember the Drink Order program? You could describe its process like the following pseudocode.

↗ EXAMPLE

```
// Ask if user wants 1) water, 2) coffee, or 3) tea
// If drink choice is 1 (water)
    // Add "water" to output string
    // Ask to enter 1) hot or 2) cold
    // If hot, add "hot" to output string
    // else if cold, add "cold" to the output string
        // Ask if user would like ice
        // If response is 'Y' or 'y', add "with ice" to output string
        // Treat any other response as no - no further action
// Else if choice is 2 (coffee)
    // Add "coffee" to the output string
    // Ask if user would like decaf (Y/N)
    // If 'Y' or 'y', add "decaf" to output string
    // Treat other input as 'N' - do nothing
    // Ask if user would like 1) milk, 2) cream, or 3) none
    // If choice is 1 (milk), add "milk" to the output string
    // else if choice is 2 (cream), add "cream" to output string
    // Else, any other choice is ignored
    // Ask if user would like sugar (Y/N)
    // If response is 'Y' or 'y', add "sugar" to output string
    // Else, do nothing
// Else if choice is 3 (tea)
    // Add "tea" to output string
    // Ask user about tea type: 1) black, 2) green
    // If tea type is 1, add "black" to output string
    // Else if tea type is 2, add "green" to output string
    // Else treat any other response as black & add "black" to output string
// Else not a valid drink selection - print message
// Print out final drink choice with options
```

Here is the pseudocode translated into a series of comments using a double slash (//) and breaking some steps down a bit more:

↗ EXAMPLE

```
// Ask if user wants 1) water, 2) coffee, or 3) tea
// If drink choice is 1 (water)
    // Add "water" to output string
    // Ask to enter 1) hot or 2) cold
    // If hot,
        // add "hot" to output string
    // else if cold,
```

```
        // add "cold" to the output string
        // Ask if user would like ice
        // If response is 'Y' or 'y',
            // add "with ice" to output string
        // Treat any other response as no - no further action
    // Else if choice is 2 (coffee)
        // Add "coffee" to the output string
        // Ask if user would like decaf (Y/N)
        // If 'Y' or 'y',
            // add "decaf" to output string
        // Treat other input as 'N' - do nothing
        // Ask if user would like 1) milk, 2) cream, or 3) none
        // If choice is 1 (milk),
            // add "milk" to the output string
        // else if choice is 2 (cream),
            // add "cream" to output string
        // Else, any other choice is ignored
        // Ask if user would like sugar (Y/N)
        // If response is 'Y' or 'y',
            // add "sugar" to output string
        // Else, do nothing
    // Else if choice is 3 (tea)
        // Add "tea" to output string
        // Ask user about tea type: 1) black, 2) green
        // If tea type is 1,
            // add "black" to output string
        // Else if tea type is 2,
            // add "green" to output string
        // Else treat any other response as black
            // add "black" to output string
    // Else not a valid drink selection - print message
    // Print out final drink choice with options
```

For the Drink Order program, we'll implement the functionality that's in the pseudocode in the next lesson.

TRY IT

Directions: This lesson is meant to just prep your program for conversion to code. It always helps to just comment your pseudocode from the start and start to fill in sections of the code that you're comfortable with in the same steps that we took. Now is a good time to convert your pseudocode into comments for your project.

SUMMARY

In this lesson, you examined the **coding framework**, as you started to translate the pseudocode into code. This is a great starting point since the lines of pseudocode can act as both our foundation of what to code as well as any starting comments. You started to code our demonstration program by identifying the variables that we are planning to use in the program. You also coded some initial

functions and expected output. Then, in the **Guided Brainstorming** section, you converted the pseudocode of the drink order program into comments to set that example up as well.

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/**

Writing the Program

by Sophia

WHAT'S COVERED

In this lesson, you will learn about the completed program for the application that we've defined. Specifically, this lesson covers:

1. Writing Core Functionality

The creation of the entire program from start to finish should ideally be done in parts. Try to focus on writing the code around a single comment regarding a piece of the functionality before moving on to the next comment. This will help make the overall process of creating the program easier. Each program will be different. What you'll do here is explore the demonstration program and walk through the code for it, so you can apply some of those same steps to your program.

In the last lesson, we created a good solid framework of the application and used this to build out some of the functionality of the main program. We completed each of those steps and now have a larger program. We now want to break our code up even further.

Although a program may work as it already is, you may want to modularize our code for easier future reuse. Certain pieces may be easy to reuse as modules in other programs. For example, if we wanted to use the

functionality for a die outside of casino craps, we could easily do so. Writing our code as modules gives us a lot more flexibility by keeping all of the elements of the code consistent between implementations. What you will end up doing is splitting our code into individual reusable classes, including a DiceRoll class that encapsulates the data and behavior for a dice roll and a Game class that consists of the code about the play of the game and a bit of information about the player.

THINK ABOUT IT

How you break up the program into classes and methods is up to you, but typically the data in and functionality of a class represents a key part or "actor" in the program (such as the roll of the dice or a single craps game). There are many reasons why this is done, including for the manageability of the code. When we create a larger problem, it can be difficult to stay focused on a single piece of code. When we break things down into individual tasks, it becomes less overwhelming as you start to develop the code. Also, by breaking things down, we have the ability to work with others on a larger problem. Since the work from different people can be compiled together to create a larger program, we can develop things a lot faster. The reuse of modules ensures that we can reuse parts that already work. If we have a piece of code that works well for a particular function, we don't have to redevelop that part over and over again.

For our demonstration program, you will continue this lesson by creating two classes to encapsulate the data and functionality for the roll of the dice and a game consisting of one or more rolls of the dice. As usual, each class will be put in its own .java file. First up is the DiceRoll class.

---

2. DiceRoll Class

The DiceRoll class simulates the roll of two 6-sided dice (with 1 to 6 pips on the sides). The roll consists of two randomly generated numbers in the range from 1 to 6. The code below uses an array of integers with the size 2, but it could be designed to use two separate integer variables to represent the dice. The class also needs to use an instance of the Random class to generate random values. The parameter passed to the Random object's nextInt() method is one more than the highest value desired, so in the code below, callingnextInt() with an argument of 6 will produce a random number in the range from 0 to 5. Since a die has values 1 to 6, the code adds one to the result produced by the random number generator. It's always best to keep in mind that while people usually start counting with 1, computers tend to start with 0. The results of the two calls to nextInt() are saved in the two-element array of integers.

It is worth noting that the generation of the random numbers for the dice roll is handled by the class's constructor, since each DiceRoll object represents the result of one roll of the dice. There is also no value in having a DiceRoll object containing "unrolled" dice, so there is no need to separate the getting of the random values from the creation of a DiceRoll object. Since a game will consist of one or more dice rolls, you also want to be able to keep track of each roll, so the class does not need a method to roll the dice again.

Since the sum of the values showing on the two dice is important, the class includes agetSum() method. To facilitate the display of the results, the class also defines a toString() method to report the result of the roll. Review the following code:

```
import java.util.Random;

public class DiceRoll {
  // Array with 2 elements to hold rolls of 2 dice
  private final int[] die = new int[2];
```

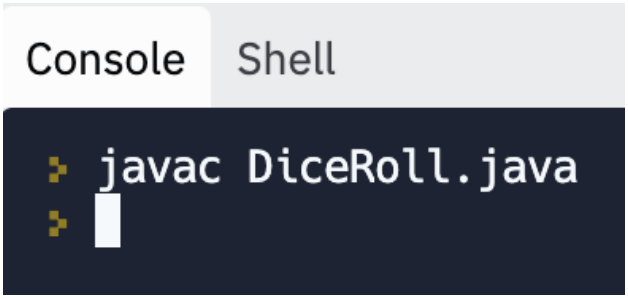
```
  // Sum of the 2 dice
  private int sum;
  // Random number generator from java.util library
  private Random randomNumberGenerator;

  public DiceRoll() {
    // Create random number generator
    randomNumberGenerator = new Random();
    // Argument of 6 means generator will produce a # in the range 0 - 5
    // + 1 adjusts the result to be in the range from 1 to 6
    die[0] = randomNumberGenerator.nextInt(6) + 1;
    die[1] = randomNumberGenerator.nextInt(6) + 1;
    sum = die[0] + die[1];
  }

  // Accessor method to get total of the 2 dice
  public int getSum() { return sum; }

  public String toString() {
    return "Dice: " + die[0] + "  " + die[1] + "  Sum = " + sum;
  }
}
```

TRY IT

Directions: Go ahead and type in the DiceRoll class into a file named DiceRoll.java. While the code will not run by itself, you can make sure that there are no syntax errors by compiling the class using this command:

```
javac DiceRoll.java
```

If the class compiles without problems, there won't be any output in the console and the system prompt will reappear:



Now that you are done with the DiceRoll class, we can turn our attention to the Game class that uses it (since a game consists of one or more rolls of the dice).

---

3. The Game Class

Since the Game class is a separate class, it needs to be entered into a file named Game.java. Next, we'll create a class called Game. This class is going to simulate the play of a game of craps until the player wins or loses. Because a game is made up of one or more rolls of the dice, this class will make use of the DiceRoll class.

Before working on the Game class, it is important to compile the code for the DiceRoll class, as noted above, before trying to compile the Game class.

Generally, for any variables that you will need to access, you will use a method that returns the value. In this case, the number of rolls will be returned with the getRollCount() method that you will define in the class demonstrated in the following code:

```
  // Get the number of dice rolls for game
 public int getRollCount() {
  // Number of DiceRoll objects in the ArrayList is number of rolls
  return diceRolls.size();
 }
```

As you have seen, the actual generation of the random numbers for the dice is handled by the DiceRoll method. The DiceRoll represents just a single roll of the dice, so it can't "know" about the number of rolls in the game.

↗ EXAMPLE

Counting the number of rolls is relevant to the game overall and not a single specific roll of the dice.

After each roll of the dice, the Game class will determine if the player has won or lost. This will be handled by the play() method in the Game class.

While these are not the only attributes in the Game class, here are the attributes used in the play() method:

↗ EXAMPLE

```
// Collection to keep track of dice rolls
private ArrayList diceRolls;
// Keeps track of point if player doesn't win or lose on first roll
private int point = 0;
private boolean keepRolling = true;
// String to hold game result msg: "Win" or "Lose"
private String gameResult = "";
```

The ArrayList for the DiceRoll objects is created in the constructor for the Game class, since there will be just one collection of rolls per game:

```
public Game(String playerName) {
  gameDateTime = LocalDateTime.now();
  this.playerName = playerName;
  diceRolls = new ArrayList<DiceRoll>();
}
```

In addition to the collection, the constructor assigns the player name, which is passed in as a parameter, to the local attribute. The constructor also records the date and time when the game starts.

Of course, the play() method is where most of the action is. Here is the code for the play() method:

```
 public ArrayList<DiceRoll> play() {
  // 1st roll for player
  DiceRoll roll = new DiceRoll();
  diceRolls.add(roll);
  // Call the DiceRoll class's getSum() to get total for the roll
  int sum = roll.getSum();

  // Check for "craps" & resulting loss
  if(sum == 2 || sum == 3 || sum == 12) {
   gameResult = "Lose";
   keepRolling = false;
  }

  // Check for immediate win
  else if(sum == 7 || sum == 1) {
   gameResult = "Win";
   keepRolling = false;
  }
  else {
   // Sum of dice is now player's "point"
   point = roll.getSum();
  }

  // Loop for subsequent rolls
  while(keepRolling) {
   roll = new DiceRoll();
   diceRolls.add(roll);
   if(roll.getSum() == 7) {
    gameResult = "Lose";
    keepRolling = false;
   }
   else if(roll.getSum() == point) {
    gameResult = "Win";
    keepRolling = false;
   }
  }
  return diceRolls;
 }
```

As the declaration of the play() method indicates, it returns the ArrayList with all of the dice rolls for the specific game. The play() method also sets the value of the gameResult attribute to "Win" or "Lose" to indicate how the player fared.

TRY IT

Directions: Here is the complete code for the `Game` class. Again, this code should be entered into a file in Replit named Game.java.

```
import java.time.LocalDateTime;
import java.util.ArrayList;
```

```
public class Game {
  private String playerName = "";
  private LocalDateTime gameDateTime;
  // ArrayList to hold dice rolls for player
  private ArrayList<DiceRoll> diceRolls;

  private int point = 0;
  private boolean keepRolling = true;
  // String to hold game result msg: "Win" or "Lose"
  private String gameResult = "";

  public Game(String playerName) {
    gameDateTime = LocalDateTime.now();
    this.playerName = playerName;
    diceRolls = new ArrayList<DiceRoll>();
  }

  public ArrayList<DiceRoll> play() {
    // 1st roll for player
    DiceRoll roll = new DiceRoll();
    diceRolls.add(roll);
    int sum = roll.getSum();

    // Check for "craps" & resulting loss
    if(sum == 2 || sum == 3 || sum == 12) {
      gameResult = "Lose";
      keepRolling = false;
    }

    // Check for immediate win
    else if(sum == 7 || sum == 1) {
      gameResult = "Win";
      keepRolling = false;
    }
    else {
      // Sum of dice is now player's "point"
      point = roll.getSum();
    }

    // Loop for subsequent rolls
    while(keepRolling) {
      roll = new DiceRoll();
      diceRolls.add(roll);
      if(roll.getSum() == 7) {
        gameResult = "Lose";
        keepRolling = false;
      }
      else if(roll.getSum() == point) {
        gameResult = "Win";
        keepRolling = false;
      }
```

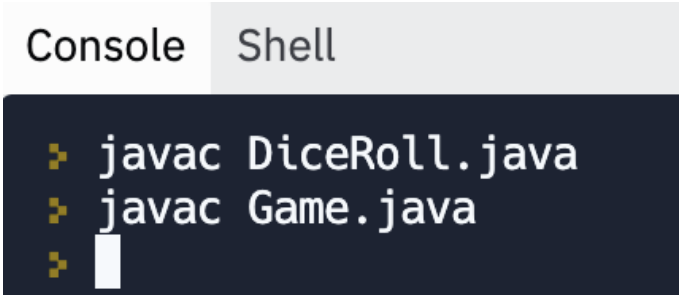
```
    }
    return diceRolls;
  }

  public String getResult() {
    return gameResult;
  }

  // Get the number of dice rolls for game
  public int getRollCount() {
    // Number of DiceRoll objects in the ArrayList is number of rolls
    return diceRolls.size();
  }

  public String getPlayerName() {
    return playerName;
  }

  public LocalDateTime getGameDateTime() {
    return gameDateTime;
  }
}
```

As with the DiceRoll code, it is important to compile this class before continuing. Any time there is a change in the code, it is best to recompile all of the classes in order:

↗ EXAMPLE

```
javac DiceRoll.java
javac Game.java
```

Lack of output indicates that the compiler did its work successfully:



4. Finishing the Main Program

With the DiceRoll and Game classes in place, it's time to work on the application's driver class. Remember that the driver class is the class that contains the program's main() method and is responsible for running the program. In this case, the driver class is named Craps (unsurprisingly) and is in a file called Craps.java.

Inside the “main” program (the Craps.java file), you would explain what the code is meant to be doing with comments. This way, any time that anyone else reviews the code, it will be obvious what the code is doing.

↷ EXAMPLE

Here is the code for the complete Craps class package:

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.StandardOpenOption;
import java.util.Scanner;
import java.util.ArrayList;
import java.text.DecimalFormat;
import java.time.format.DateTimeFormatter;
import java.util.InputMismatchException;

public class Craps {
   public static void main(String[] args) {
      // Variables for stats
      int winCount = 0;
      int lossCount = 0;
      int gameCount = 0;
      int rollCount = 0;

      // Log file for player statistics
      File logFile = new File("game.log.txt");

      // Scanner to read player name & desired number of games
      Scanner input = new Scanner(System.in);
      System.out.print("Enter Player Name: ");
      String playerName  = input.nextLine();

      // Run sample game first
      System.out.println("\nRunning Sample Game: ");
      Game game = new Game(playerName);
      // ArrayList to track roles of dice in a game
      ArrayList<DiceRoll> rolls = game.play();
      for(DiceRoll roll : rolls) {
         System.out.println("\t" + roll);
      }
      System.out.println("\nResult of Sample Game: " + game.getResult() + " in " +
         game.getRollCount() + " roll(s)\n");
      System.out.println();

      // Track if input of requested # of games to play is valid
      boolean inputValid = false;
      int numberOfGamesToPlay = 0;

      // Assume invalid input of number until it proves to be correct
      while(!inputValid) {
         try {
            System.out.print("How many games would you like to play?: ");
            numberOfGamesToPlay = input.nextInt();
```

```
            // If previous statement doesn't throw exception, input valid
            inputValid = true;
        }
        catch(InputMismatchException ex) {
            System.out.println("Not a valid number.");
            // Clear out input buffer
            input.nextLine();
        }

    }
    System.out.println(); // Add blank line in output

    // Loop to play desired number of games
    for(int i = 0; i < numberOfGamesToPlay; i++) {
        game = new Game(playerName);

        // Need to use parentheses so value is calculated before concatenation
        System.out.println("Game " + (i + 1) + ":");
        rolls = game.play();
        rollCount += game.getRollCount();
        for(DiceRoll roll : rolls) {
            System.out.println("\t" + roll);
        }
        if(game.getResult().equals("Win")) {
            winCount++;
        }
        else {
            lossCount++;
        }
        System.out.println("\nResult: " + game.getResult() + " in " +
        game.getRollCount() + " roll(s)\n");
    }

    // Compose String with summary statistics
    String summary = "Statistics for " + game.getPlayerName() + ": " + "\n";
    String gameDateTimeFormatString = "MM/dd/YYYY HH:mm:ss\n";
    DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern(gameDateTimeFormatString);
    summary += "Game Date & Time: ";
    summary += dateTimeFormatter.format(game.getGameDateTime());
    summary += "# of Games: " + numberOfGamesToPlay + "\n";
    summary += "# of Dice Rolls: " + rollCount + "\n";
    summary += "# of Wins: " + winCount + "\n";
    summary += "# of Losses: " + lossCount + "\n";
    DecimalFormat percentageFormat = new DecimalFormat("0.00%");
    double winPercentage = (double) winCount / numberOfGamesToPlay;
    summary += "% Wins: " + percentageFormat.format(winPercentage) + "\n\n";

    // Display statistics on screen
    System.out.println(summary);

    // Write statistics to log file
```

```
        try {
            // Create log file if it doesn't exist and then append to it.
            Files.writeString(logFile.toPath(), summary, StandardOpenOption.CREATE,
                    StandardOpenOption.APPEND);
        }
        catch(IOException ex) {
            System.out.println("Error writing to log file: " + ex.getMessage());
        }
    }
}
```

In this case, all of the code for the driver class is in the `main()` method. The Game class handles the details of each game (and the DiceRoll class handles all of the dice rolls for a game), so the code in main() handles getting user input (such as the player's name and the number of games the player wants to play) and presents the results. `main()` also handles writing the statistics to the log file. Depending on the circumstances and design choices, the driver class could contain static methods besides main(). For instance, the logging could be handled by a separate method, but since the output to the screen and to the log file have much in common and the logging doesn't have to do anything more than write the data, it seems like a good choice to have the code in `main()` take care of the logging.

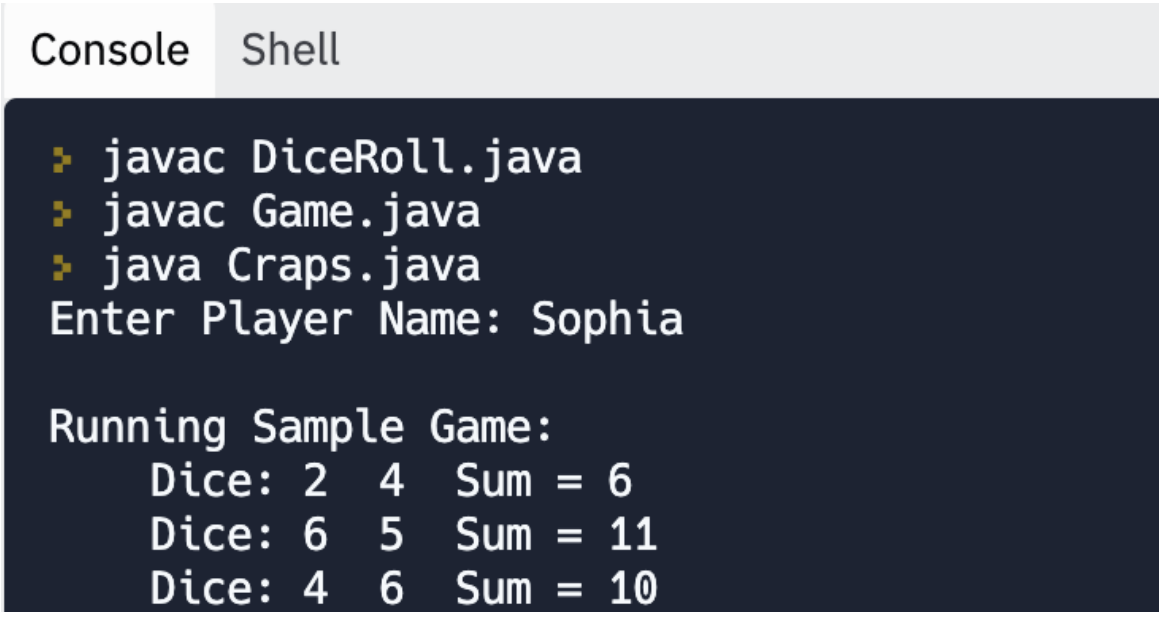
TRY IT

Directions: Type in the code listed above for the Craps class. Since the Craps class is the driver class, it is important that it not be compiled using javac. It will be run directly using the java command in the console window. Just to make sure that everything is in order, it is a good idea to recompile the DiceRoll and Game classes with javac and then run the program (Craps.java) using the Java command.

EXAMPLE

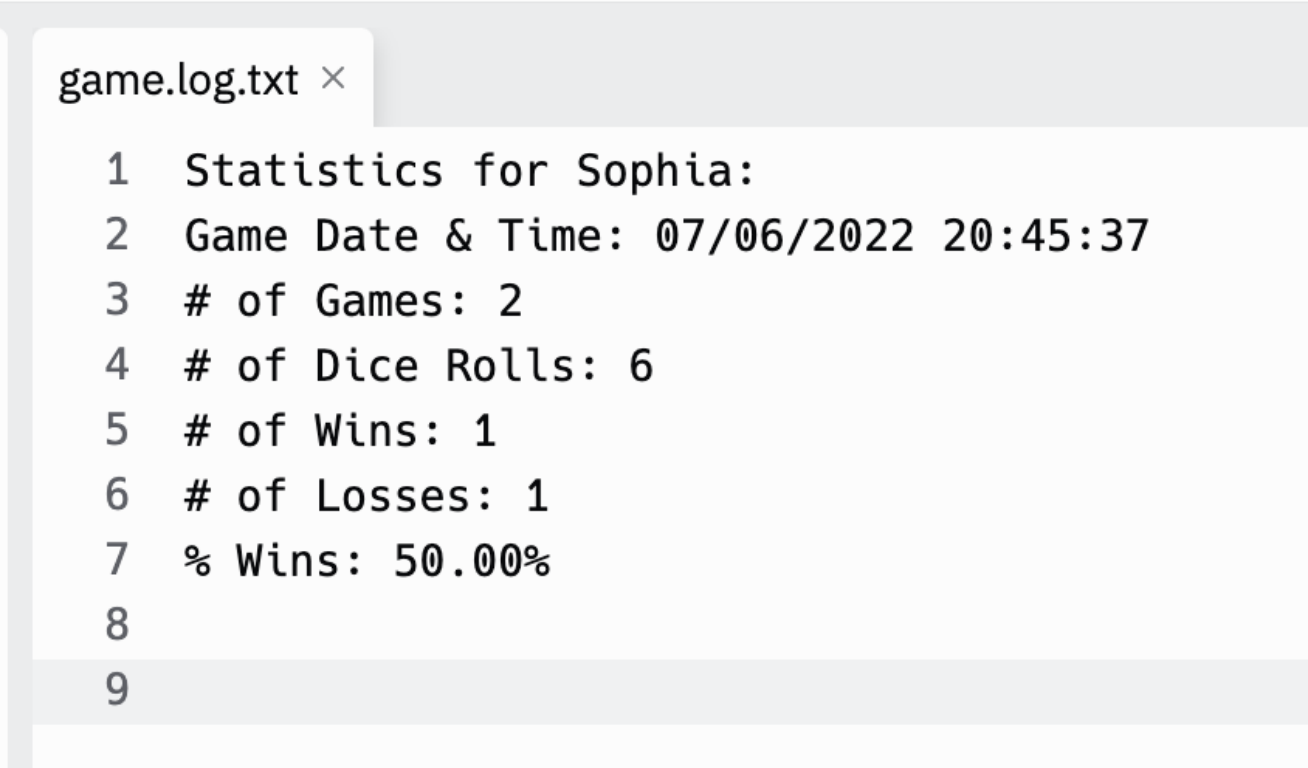
```
javac DiceRoll.java
javac Game.java
java Craps.java
```

The results of running the program should look something like this (this sample requests just two games to keep things brief):



```
    Dice: 6  5  Sum = 11
    Dice: 1  5  Sum = 6

Result of Sample Game: Win in 5 roll(s)


How many games would you like to play?: 2

Game 1:
    Dice: 4  6  Sum = 10
    Dice: 6  3  Sum = 9
    Dice: 5  5  Sum = 10

Result: Win in 3 roll(s)

Game 2:
    Dice: 3  3  Sum = 6
    Dice: 2  6  Sum = 8
    Dice: 1  6  Sum = 7

Result: Lose in 3 roll(s)

Statistics for Sophia:
Game Date & Time: 07/06/2022 20:45:37
# of Games: 2
# of Dice Rolls: 6
# of Wins: 1
# of Losses: 1
% Wins: 50.00%
```

The contents of the log file (game.log.txt) look like this when opened in Replit (by double clicking on the game.log.txt file):



The program appends to this file, so the log will grow as more games are played.

REFLECT

The code near the end of main() handles writing the statistics for the session to the log file because the log entries need to reflect the outcome for the number of games that the player chose to play. Why wouldn't this be possible if logging were added to the Game or DiceRoll classes? It is not part of the specified requirements, but if logging were added to the Game and DiceRoll classes, what sort of information could be recorded?

---

5. Guided Brainstorming

THINK ABOUT IT

This demonstration program is likely a bit more complex than the Sophia graders expect to see. For the sample program, the goal was to use many of the concepts and Java constructs that we learned along the way. Remember your program can be as simple or complex as you feel it should be. The demonstration program had the added elements of splitting the program into classes for reuse. It also handles output to a file. As you develop your own program, try to keep it simple at the beginning. Once you're comfortable and have things working, you can start making the program more complex. You can add features for enhancement and reuse if desired.

Now, back to the drink order program that you referenced from Unit 1. Here is the pseudocode from the previous lesson:

↱ EXAMPLE

```
// Ask if user wants 1) water, 2) coffee, or 3) tea
// If drink choice is 1 (water)
   // Add "water" to output string
```

```
    // Ask to enter 1) hot or 2) cold
    // If hot, add "hot" to output string
    // else if cold, add "cold" to the output string
        // Ask if user would like ice
        // If response is 'Y' or 'y', add "with ice" to output string
        // Treat any other response as no - no further action
  // Else if choice is 2 (coffee)
    // Add "coffee" to the output string
    // Ask if user would like decaf (Y/N)
    // If 'Y' or 'y', add "decaf" to output string
    // Treat other input as 'N' - do nothing
    // Ask if user would like 1) milk, 2) cream, or 3) none
    // If choice is 1 (milk), add "milk" to the output string
    // else if choice is 2 (cream), add "cream" to output string
    // Else, any other choice is ignored
    // Ask if user would like sugar (Y/N)
    // If response is 'Y' or 'y', add "sugar" to output string
    // Else, do nothing
  // Else if choice is 3 (tea)
    // Add "tea" to output string
    // Ask user about tea type: 1) black, 2) green
    // If tea type is 1, add "black" to output string
    // Else if tea type is 2, add "green" to output string
    // Else treat any other response as black & add "black" to output string
  // Else not a valid drink selection - print message
  // Print out final drink choice with options
```

The resulting program after converting the pseudocode to code without any other changes results in the following:

```
import java.util.Scanner;
import java.util.InputMismatchException;

public class DrinkOrder {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    boolean validInput = false;
    // String variable to hold drink details
    // Need to declare before start of loop to avoid scope problems
    String drinkDetails = "No drink chosen.";
    // Note need to declare choice variable before loop
    // to avoid problems with scope.
    int choice = 0;
    while(!validInput) {
      System.out.println("What type of drink would you like to order?");
      // Note use of new line \n to print 3 lines with 1 statement.
      System.out.println("1. Water\n2. Coffee\n3. Tea");
```

```
  System.out.print("Drink selection #: ");
  try {
    choice = input.nextInt();
    validInput = true;
  }
  catch(InputMismatchException ex) {
    System.out.println("Not a valid selection.");
  }
  // Remove \n left in input to avoid problems with later inputs
  input.nextLine();
}
if(choice == 1) {
  drinkDetails = "Water";
  System.out.println("Would you like that 1) hot or 2) cold?");
  System.out.print("Enter temperature selection #: ");
  choice = input.nextInt();
  // Remove new line left in input stream to avoid problems with later inputs
  input.nextLine();
  if(choice == 1) {
    drinkDetails += ", hot";
  }
  else if(choice == 2) {
    drinkDetails += ", cold";
    System.out.print("Would you like ice? (Y/N) ");
    // Read input as a String
    String response = input.nextLine();
    // Extract 1st char
    char yesNo = response.charAt(0);
    // Y or y is yes, anything else interpreted as no
    if(yesNo == 'Y' || yesNo == 'y') {
      drinkDetails += ", with ice";
    }
  }
  else {
    System.out.println("Not a valid temperature selection.");
  }
}
else if(choice == 2) {
  drinkDetails = "Coffee";
  System.out.print("Would you like decaf? (Y/N): ");
  String decafResponse = input.nextLine();
  char decafYesNo = decafResponse.charAt(0);
  if(decafYesNo == 'Y' || decafYesNo == 'y') {
    drinkDetails += ", decaf";
  }
  System.out.println("Would you like 1) milk, 2) cream, or 3) none?");
  System.out.print("Enter choice #: ");
  int milkCreamChoice = input.nextInt();
  // Remove new line left in input stream to avoid problems with later inputs
  input.nextLine();
  if(milkCreamChoice == 1) {
```

```
      drinkDetails += ", milk";
    }
    else if(milkCreamChoice == 2) {
      drinkDetails += ", cream";
    }
    System.out.print("Would you like sugar? (Y/N): ");
    String sugarResponse = input.nextLine();
    char sugar = sugarResponse.charAt(0);
    if(sugar == 'Y' || sugar == 'y') {
      drinkDetails += ", sugar";
    }
   }
   else if(choice == 3) {
    drinkDetails = "Tea";
    System.out.print("Type of tea: 1) Black or 2) Green: ");
    int teaChoice = input.nextInt();
    // Remove \n left in input to avoid problems with later inputs
    input.nextLine();
    if(teaChoice == 1) {
      drinkDetails += ", black";
    }
    else if(teaChoice == 2) {
      drinkDetails += ", green";
    }
    else {
      // Invalid selection - assume black tea
      drinkDetails += ", black";
      System.out.println("Not a valid choice. Assuming black tea.");
    }
   }
   else {
    System.out.println("Sorry, not a valid drink selection.");
   }

   // Print out final drink selection
   System.out.println("Your drink selection: " + drinkDetails + ".");
  }
}
```

Be sure as you start your development that you're including all of the core functionality that you need from the beginning. Without doing so, you're going to get into some issues along the way that can be harder for you to decipher as the program gets more complex. As you code your program, think of the following questions:

- Does the program work as intended? Are you seeing the expected output?
- Does the program implement all of the requirements from the client?
- Can the program be optimized using conditional statements, loops, functions, and classes?

This will ensure that you're meeting all of the requirements as you progress in programming your solution.

Directions: At this point, you should be building out your program to put it into a state that is testable to ensure that everything is working as expected. For any areas that you have not implemented yet, it would be a good

idea to comment those sections out as needed.

REFLECT

Always remember that even the longest program is written a few lines at a time. It is important to check your code frequently by compiling and running it. Of course, it may not be possible to compile and run the code after each line, since some constructs such as selection statements and loops require a few lines to be syntactically correct. As you write your code, keep a lookout for points where enough new code has been written to allow checking for syntax or logical errors.

SUMMARY

In this lesson, you broke down the demonstration program based on **core functionalities**. You wanted the ability to be able to reuse some of the functionality for other cases. To do that, you created the **DiceRoll and Game classes** that provide important functionality to the main program. You returned to the main program and added play functions that allow for a single and multiple games. To **finish the main() method in the program**, you added a section of code to output the results to a file. In the**Guided Brainstorming** section, you took the pseudocode of the Drink Order program and converted that to code. In the next tutorial, you will try testing these programs.

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/**

Testing as You Go

by Sophia

WHAT'S COVERED

In this lesson, you will learn about testing as we progress. Specifically, this lesson covers:

1. Checking for Errors

Errors, bugs, and exceptions are always there in any program that we develop. These errors and issues can cause all types of problems with the program. As you work through the code, it's a good idea to test the program as much as you can. You will want to try to do everything that you can to break it and then find ways to help protect it from breaking in the future.

EXAMPLE Some examples of things we should test and debug are:

1. Invalid inputs—if the program asks the user to enter in an integer, try to enter something else, like a character.
2. Edge and corner cases for conditional statements.
3. Uppercase and lowercase inputs for conditional statements compared to strings.
4. Complete testing of end user use, or the program, from start to finish.

There may be times while debugging code when you cannot find a solution for an error. If so, you can move on temporarily, and you may simply add the code into a comment area and test it again later on.
In the sample program, the only required input is for the number of games the player would like to play. The user is prompted to enter a number. What happens if you enter a character other than a digit? The code in the relevant section of the application's main() could just try to read the number of games to play, like the code below.

EXAMPLE This is a simplified version of the code shown in the previous lesson that handles input using a more naive approach:

```
int numberOfGamesToPlay = 0;
System.out.print("How many games would you like to play?: ");
// Read input as an integer
numberOfGamesToPlay = input.nextInt();
System.out.println(); // Add blank line in output
```

If the Craps.java code is run using just this way of handling input and the user enters a character that is not a valid digit, the result looks like this:

Console Shell

```
java Craps.java
Enter Player Name: Sophia

Running Sample Game:
    Dice: 2  4  Sum = 6
    Dice: 6  3  Sum = 9
    Dice: 2  1  Sum = 3
    Dice: 5  1  Sum = 6

Result of Sample Game: Win in 4 roll(s)


How many games would you like to play?: ?
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:939)
    at java.base/java.util.Scanner.next(Scanner.java:1594)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
    at Craps.main(Craps.java:40)
```

TRY IT

Directions: Try running the Craps.java program from the previous lesson with the "simplified" approach to input shown above. While entering a valid number won't cause any problems, see what happens if you enter a letter or other symbol instead of a number.

EXAMPLE Take a look again at how the code presented in the previous unit actually handles this input:

```
// Track if input of requested # of games to play is valid
boolean inputValid = false;
int numberOfGamesToPlay = 0;

// Assume invalid input of number until it proves to be correct
while(!inputValid) {
   try {
      System.out.print("How many games would you like to play?: ");
      numberOfGamesToPlay = input.nextInt();
      // If previous statement doesn't throw exception, input valid
      inputValid = true;
   }
   catch(InputMismatchException ex) {
      System.out.println("Not a valid number.");
      // Clear out input buffer
```

```
        input.nextLine();
    }
}
System.out.println(); // Add blank line in output
```

While this version is longer and a bit more complex, it will provide for a better user experience.

In the code above for the input of the number of games to play, a boolean variable,inputValid, is declared and initialized to false. The while loop is then set up to keep running as long as inputValid is not true. The value of this variable is set to true after the Scanner has successfully read an integer. If the user does not enter a valid integer, the call to nextInt() throws an exception (an InputMismatchException) and control switches to the catch block, so the statement that sets inputValid to true does not run. This means that the loop will keep going until the user makes a valid entry. This is a common pattern used with user input in Java. This approach has the advantage of avoiding a program crash due to invalid input.

TRY IT

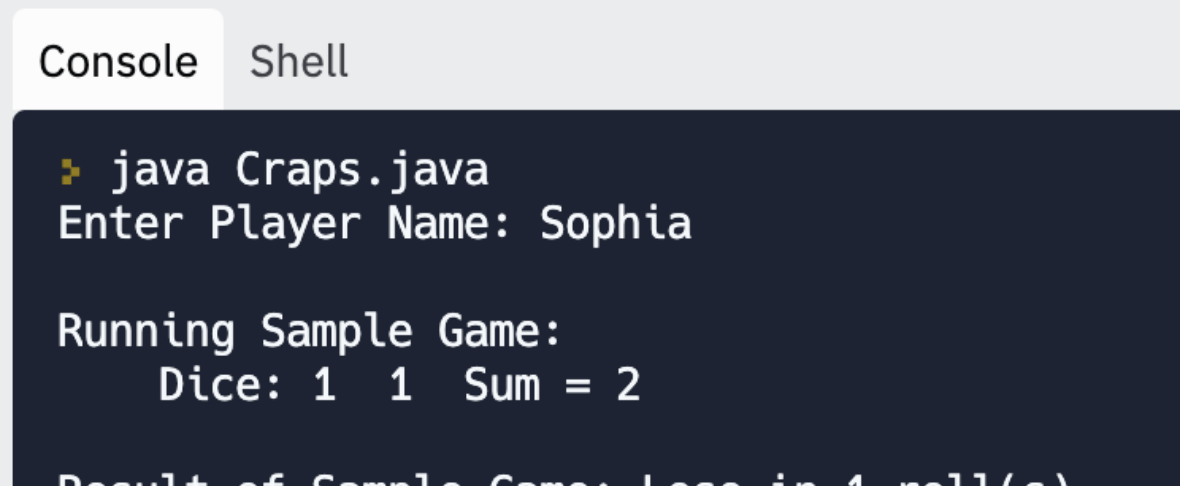
Directions: Try the original code for Craps.java with the improved input handling shown above. To make sure that you have the code entered correctly, here is a complete listing of the Craps.java file (as covered in the previous lesson) with the input section of the code in bold:

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.StandardOpenOption;
import java.util.Scanner;
import java.util.ArrayList;
import java.text.DecimalFormat;
import java.time.format.DateTimeFormatter;
import java.util.InputMismatchException;

public class Craps {
  public static void main(String[] args) {
    // Variables for stats
    int winCount = 0;
    int lossCount = 0;
    int rollCount = 0;

    // Log file for player statistics
    File logFile = new File("game.log.txt");

    // Scanner to read player name & desired number of games
    Scanner input = new Scanner(System.in);
    System.out.print("Enter Player Name: ");
    String playerName = input.nextLine();

    // Run sample game first
    System.out.println("\nRunning Sample Game: ");
```

```
Game game = new Game(playerName);
// ArrayList to track roles of dice in a game
ArrayList<DiceRoll> rolls = game.play();
for(DiceRoll roll : rolls) {
  System.out.println("\t" + roll);
}
System.out.println("\nResult of Sample Game: " + game.getResult() + " in " +
      game.getRollCount() + " roll(s)\n");
System.out.println();

// Track if input of requested # of games to play is valid
boolean inputValid = false;
int numberOfGamesToPlay = 0;

// Assume invalid input of number until it proves to be correct
while(!inputValid) {
  try {
    System.out.print("How many games would you like to play?: ");
    numberOfGamesToPlay = input.nextInt();
    // If previous statement doesn't throw exception, input valid
    inputValid = true;
  }
  catch(InputMismatchException ex) {
    System.out.println("Not a valid number.");
    // Clear out input to remove \n
    input.nextLine();
  }
}
System.out.println(); // Add blank line in output

// Loop to play desired number of games
for(int i = 0; i < numberOfGamesToPlay; i++) {
  game = new Game(playerName);
  // Need to use parentheses so value is calculated before concatenation
  System.out.println("Game " + (i + 1) + ":");
  rolls = game.play();
  rollCount += game.getRollCount();
  for(DiceRoll roll : rolls) {
    System.out.println("\t" + roll);
  }
  if(game.getResult().equals("Win")) {
    winCount++;
  }
  else {
    lossCount++;
  }
  System.out.println("\nResult: " + game.getResult() + " in " +
```

```
            game.getRollCount() + " roll(s)\n");
    }

    // Compose String with summary statistics
    String summary = "Statistics for " + game.getPlayerName() + ": " + "\n";
    String gameDateTimeFormatString = "MM/dd/YYYY HH:mm:ss\n";
    DateTimeFormatter dateTimeFormatter =
DateTimeFormatter.ofPattern(gameDateTimeFormatString);
    summary += "Game Date & Time: ";
    summary += dateTimeFormatter.format(game.getGameDateTime());
    summary += "# of Games: " + numberOfGamesToPlay + "\n";
    summary += "# of Dice Rolls: " + rollCount + "\n";
    summary += "# of Wins: " + winCount + "\n";
    summary += "# of Losses: " + lossCount + "\n";
    DecimalFormat percentageFormat = new DecimalFormat("0.00%");
    double winPercentage = (double) winCount / numberOfGamesToPlay;
    summary += "% Wins: " + percentageFormat.format(winPercentage) + "\n\n";

    // Display statistics on screen
    System.out.println(summary);

    // Write statistics to log file
    try {
      // Create log file if it doesn't exist and then append to it.
      Files.writeString(logFile.toPath(), summary, StandardOpenOption.CREATE,
              StandardOpenOption.APPEND);
    }
    catch(IOException ex) {
      System.out.println("Error writing to log file: " + ex.getMessage());
    }
   }
}
```

Running this code should produce output like the following when an incorrect value is entered for the number of games to play:



```
Result of Sample Game: Lose in 1 roll(s)


How many games would you like to play?: ?
Not a valid number.
How many games would you like to play?: X
Not a valid number.
How many games would you like to play?: 2

Game 1:
    Dice: 3  3  Sum = 6
    Dice: 3  5  Sum = 8
    Dice: 3  3  Sum = 6

Result: Win in 3 roll(s)

Game 2:
    Dice: 3  1  Sum = 4
    Dice: 5  2  Sum = 7

Result: Lose in 2 roll(s)

Statistics for Sophia:
Game Date & Time: 07/10/2022 19:41:09
# of Games: 2
# of Dice Rolls: 5
# of Wins: 1
# of Losses: 1
% Wins: 50.00%
```

REFLECT

The key constructs here are the while loop that keeps running until the user enters a valid number and the try/catch blocks. If the call to nextInt() fails due to invalid input, the program routes to the catch block immediately, so the boolean variable is not set to true, which means the loop keeps running. The use of try and catch avoids an ugly error message (and crash), and the loop makes sure the user is prompted for a number until a valid entry is made.

1a. Debugging

If you have errors or issues that you can't figure out, remember you can temporarily add System.out.println()

statements to display the values of variables and make sure they are being set and updated as expected. If it is hard to figure out how far the program has gotten before an error occurs, it can be helpful to add some outputs (using System.out.println()) with messages like "OK to line 50", etc. Just don't forget to remove these extra debugging statements when the problem has been fixed.

HINT

Adding brief comments like // DEBUG above or after these temporary statements can help you find and remove them later.

---

2. Adding the Fourth Journal Entry

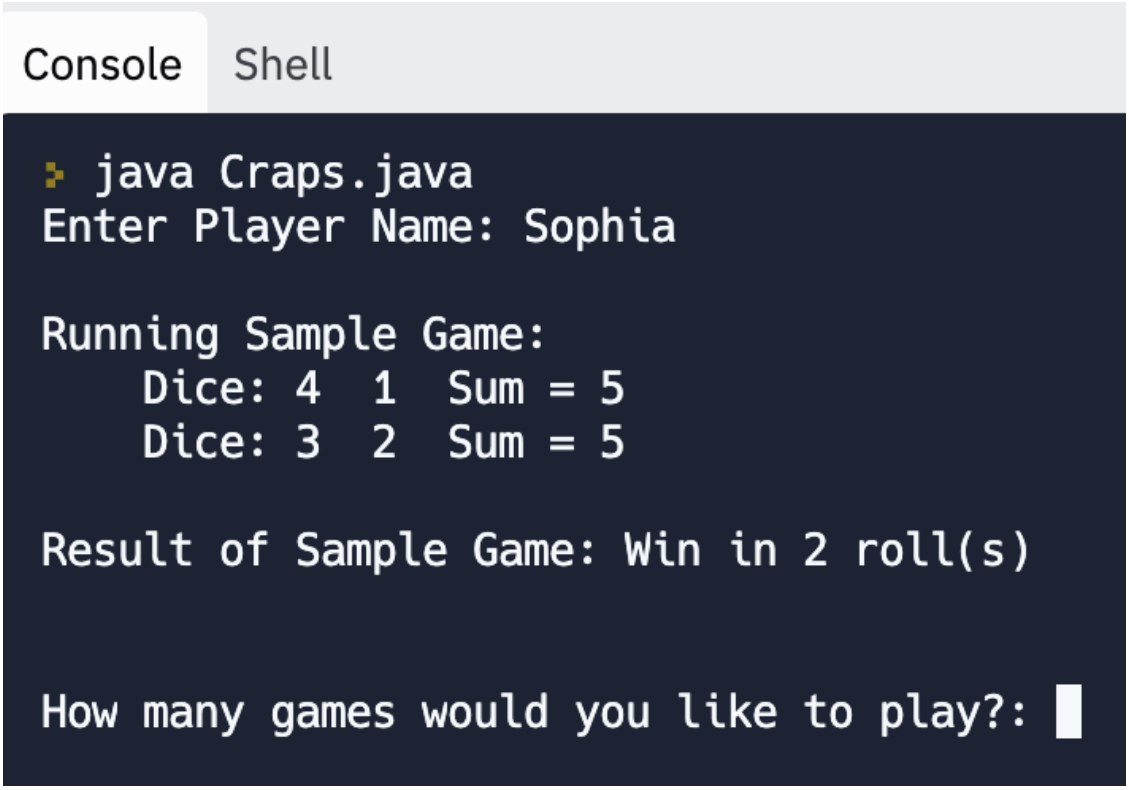
THINK ABOUT IT

At this point, we are ready to add the next journal entry (Part 4) for the Touchstone. The testing of your program is something you want to not only perform, but also show. You should document what you did for it and show the fixes that you made to resolve the problems that occurred.

What we would not want to add for our journal entry are the details of what was wrong without explaining what the issues were and how we fixed the issues. For example, this would be a bad entry for Part 4.

2a. Bad Example of Journal Entry for Part 4

Here is my test of the craps game.



HINT

A better entry for Part 4 would explain what the errors and issues were that came up and what was done to fix them. Simply looking at this output statement, you can't identify what the issue was specifically.

2b. Good Example of Journal Entry for Part 4

After changing my initial code in the main program to check for invalid user entries by adding the try and catch blocks, the program no longer crashed, but more was needed so that it would prompt the user to enter a valid number of games to play. The revised code uses a while loop so that the user is prompted for input of the number of games until a valid number is entered.

---

3. Guided Brainstorming

You will notice that as you program and test, you will likely change the code as you go. Doing so will ensure that the program works correctly every step of the way and simplifies debugging. Every program will be different, so it is important that you're thinking about the problem.

Let's look at our Drink Order program that we've been working on:

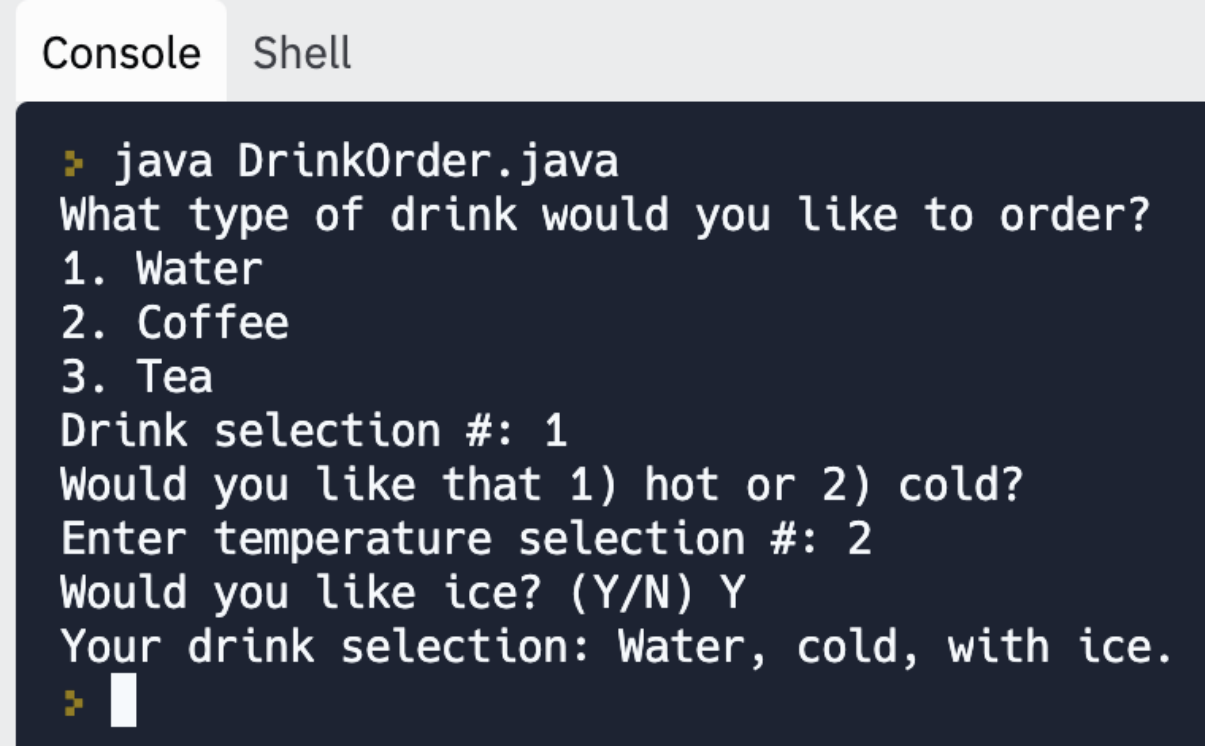
```
import java.util.Scanner;

public class DrinkOrder {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.println("What type of drink would you like to order?");
    // Note use of new line \n to print 3 lines with 1 statement.
    System.out.println("1. Water\n2. Coffee\n3. Tea");
    System.out.print("Drink selection #: ");
    // String variable to hold drink details
    String drinkDetails = "No drink chosen.";
    int choice = input.nextInt();
    // Remove \n left in input to avoid problems with later inputs
    input.nextLine();
    if(choice == 1) {
      drinkDetails = "Water";
      System.out.println("Would you like that 1) hot or 2) cold?");
      System.out.print("Enter temperature selection #: ");
      choice = input.nextInt();
      // Remove new line left in input stream to avoid problems with later inputs
      input.nextLine();
      if(choice == 1) {
        drinkDetails += ", hot";
      }
      else if(choice == 2) {
        drinkDetails += ", cold";
        System.out.print("Would you like ice? (Y/N) ");
        // Read input as a String
        String response = input.nextLine();
        // Extract 1st char
        char yesNo = response.charAt(0);
        // Y or y is yes, anything else interpreted as no
        if(yesNo == 'Y' || yesNo == 'y') {
```

```
      drinkDetails += ", with ice";
    }
  }
  else {
    System.out.println("Not a valid temperature selection.");
  }
}
else if(choice == 2) {
  drinkDetails = "Coffee";
  System.out.print("Would you like decaf? (Y/N): ");
  String decafResponse = input.nextLine();
  char decafYesNo = decafResponse.charAt(0);
  if(decafYesNo == 'Y' || decafYesNo == 'y') {
    drinkDetails += ", decaf";
  }
  System.out.println("Would you like 1) milk, 2) cream, or 3) none?");
  System.out.print("Enter choice #: ");
  int milkCreamChoice = input.nextInt();
  // Remove new line left in input stream to avoid problems with later inputs
  input.nextLine();
  if(milkCreamChoice == 1) {
    drinkDetails += ", milk";
  }
  else if(milkCreamChoice == 2) {
    drinkDetails += ", cream";
  }
  System.out.print("Would you like sugar? (Y/N): ");
  String sugarResponse = input.nextLine();
  char sugar = sugarResponse.charAt(0);
  if(sugar == 'Y' || sugar == 'y') {
    drinkDetails += ", sugar";
  }
}
else if(choice == 3) {
  drinkDetails = "Tea";
  System.out.print("Type of tea: 1) Black or 2) Green: ");
  int teaChoice = input.nextInt();
  // Remove \n left in input to avoid problems with later inputs
  input.nextLine();
  if(teaChoice == 1) {
    drinkDetails += ", black";
  }
  else if(teaChoice == 2) {
    drinkDetails += ", green";
  }
  else {
    // Invalid selection - assume black tea
    drinkDetails += ", black";
    System.out.println("Not a valid choice. Assuming black tea.");
  }
}
```

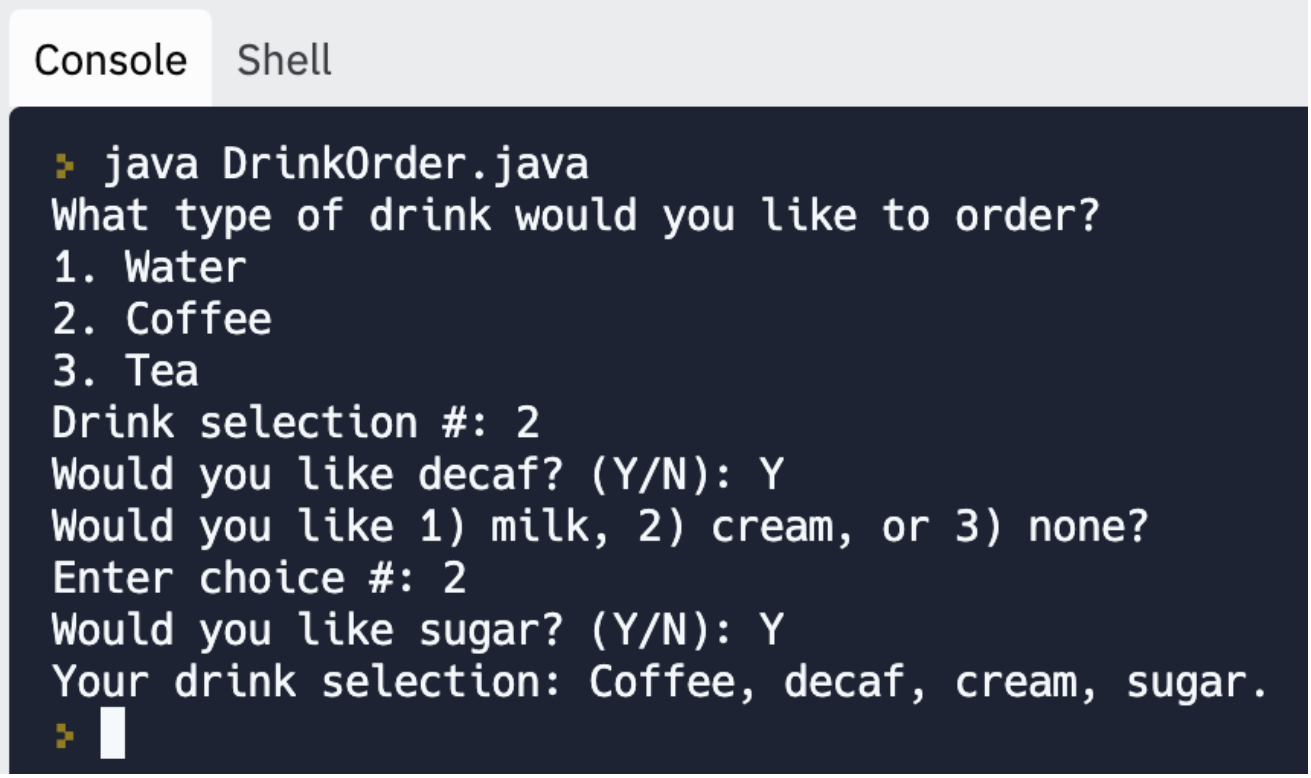
```
    else {
      System.out.println("Sorry, not a valid drink selection.");
    }

    // Print out final drink selection
    System.out.println("Your drink selection: " + drinkDetails + ".");
  }
}
```

You should test each of the branches and scenarios of the valid values between the water, coffee, and tea to ensure it works as expected.

↗ EXAMPLE Let's try entering a 1 when choosing the beverage to see what happens:

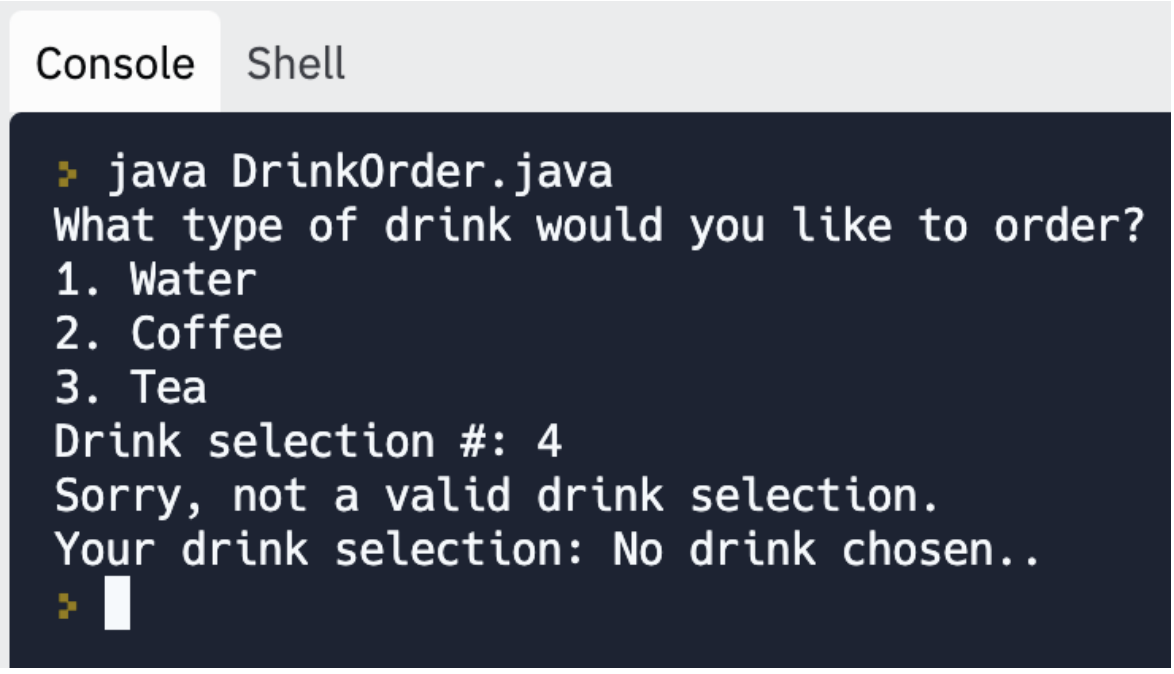


↗ EXAMPLE Let's try entering a 2 when choosing the beverage:



This is fine so far, but what happens if the user makes an invalid selection?

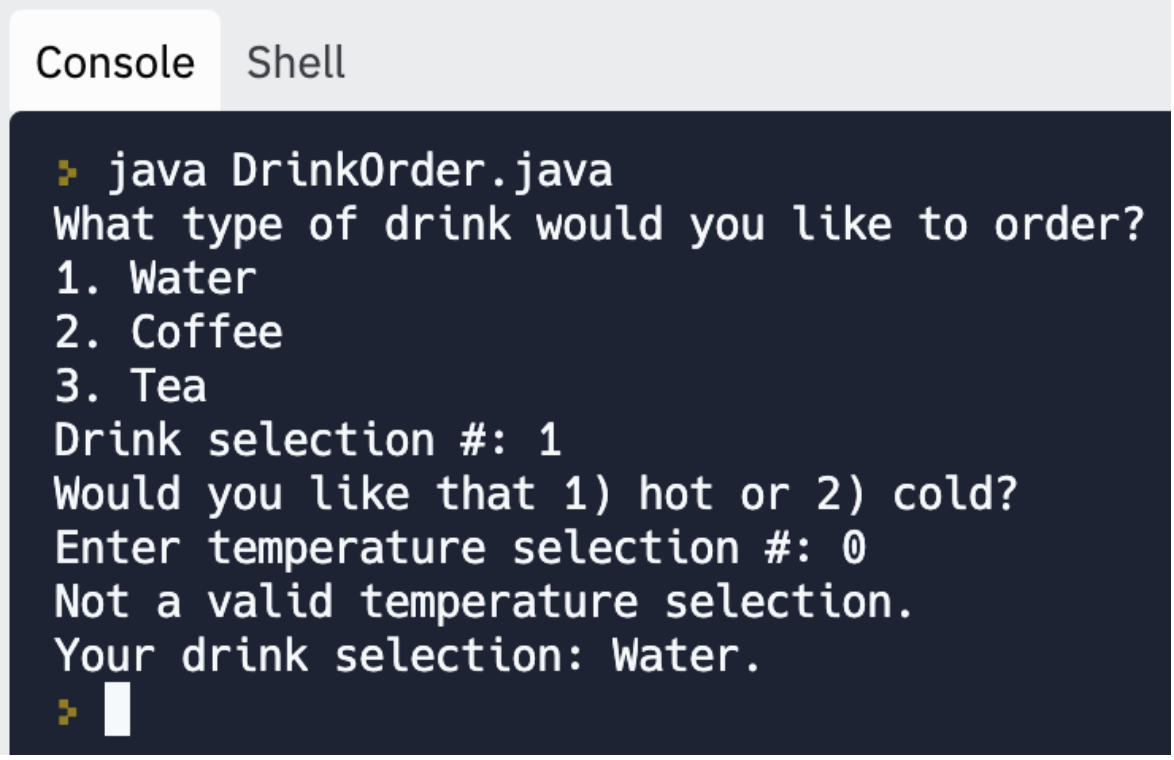
↱ EXAMPLE Let's try entering a 4 when choosing the beverage:



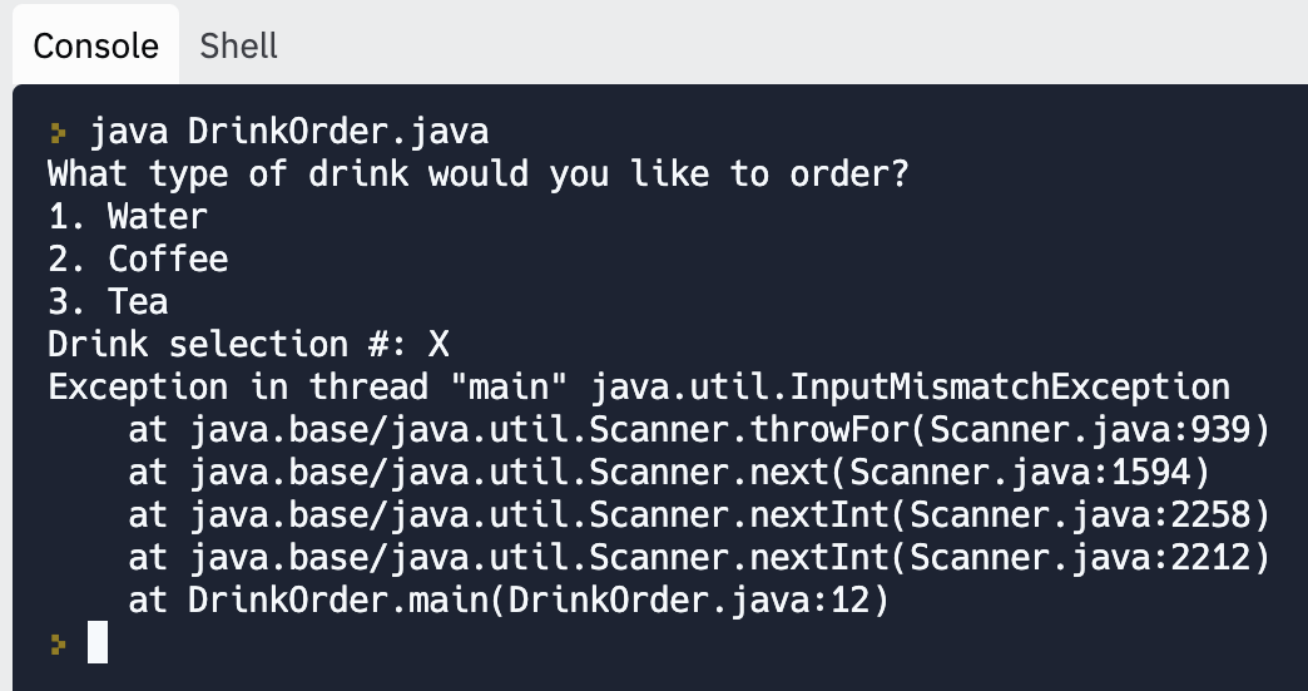
The program ends with an indication that the entry was not a valid drink selection. This is okay, but here is a place where introducing a loop to run until the user makes a valid selection could provide a better user experience (instead of requiring the person to start the program again).

Something similar happens if the person makes an incorrect entry later in the program.

↱ EXAMPLE Let's try entering the letter "O" when choosing the temperature selection:



If the user enters a non-digit when a numeric choice is expected, the program will crash (as we have seen before):



As with the craps game, it would be a good idea to use a combination of a loop around try and catch blocks to deal with problems reading the input. Here is a revised start of the code that handles invalid input similar to the solution for the number of games to play for the craps program. This is not a complete listing—just the start to show how the changes could be made.

↗ EXAMPLE

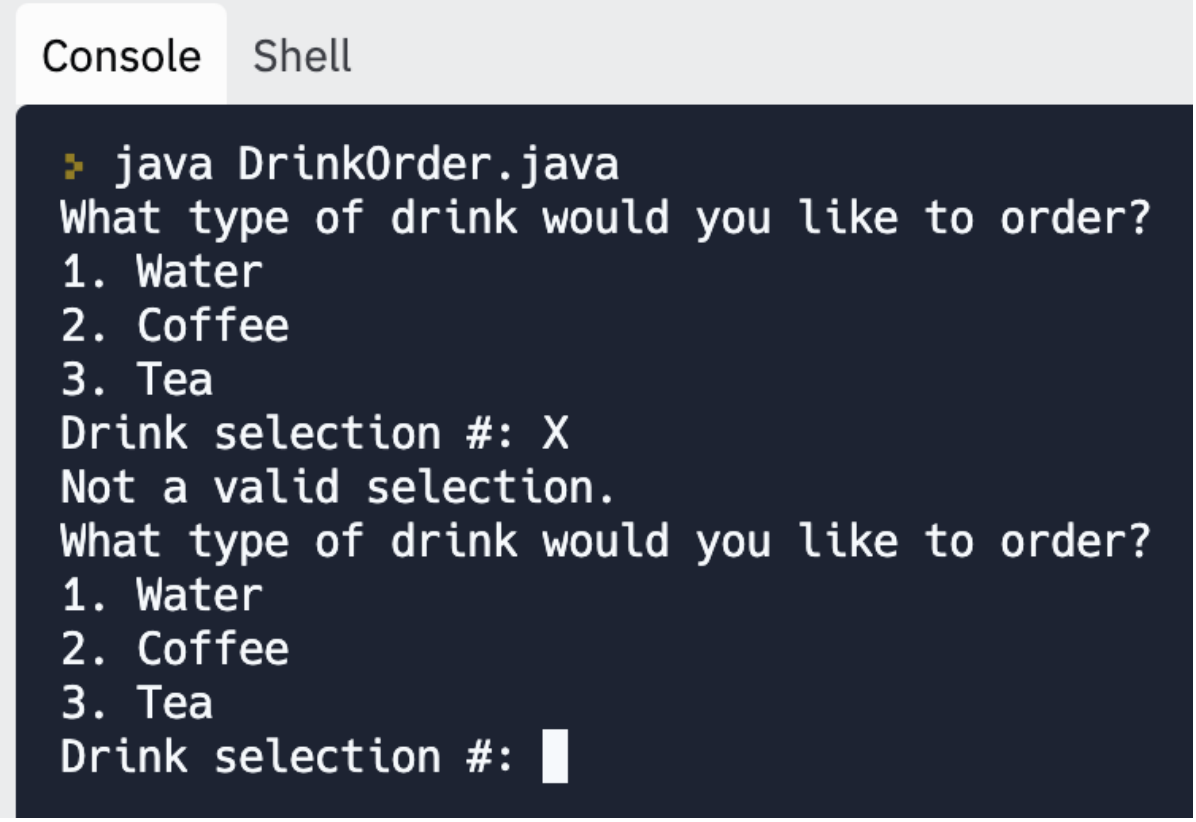
```
import java.util.InputMismatchException;
```

The version below includes `import java.util.InputMismatchException;` so the program can catch an `InputMismatchException` when `nextInt()` is called.

```
import java.util.Scanner;
import java.util.InputMismatchException;

public class DrinkOrder {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    boolean validInput = false;
    // String variable to hold drink details
    // Need to declare before start of loop to avoid scope problems
    String drinkDetails = "No drink chosen.";
    // Note need to declare choice variable before loop
    // to avoid problems with scope.
    int choice = 0;
    while(!validInput) {
      System.out.println("What type of drink would you like to order?");
      // Note use of new line \n to print 3 lines with 1 statement.
      System.out.println("1. Water\n2. Coffee\n3. Tea");
      System.out.print("Drink selection #: ");
      try {
        choice = input.nextInt();
        validInput = true;
      }
      catch(InputMismatchException ex) {
        System.out.println("Not a valid selection.");
      }
      // Remove \n left in input to avoid problems with later inputs
      input.nextLine();
    }
  }
}
```

The program now prompts the user again, if the input is not a valid digit.

↷ EXAMPLE Let's try entering the letter "X" at the drink selection prompt:



Now the program lets the user know that our drink selection was invalid. The code still does not allow another attempt if the number entered is out of range, though.

THINK ABOUT IT

What happens if you enter the number 4 as your drink selection? You might want to think about how you could modify the code to handle numeric user entry errors like we did for letter entry errors.

TRY IT

Directions: Needless to say, you should be testing your code as you are writing, but there are a lot of logical issues that can occur, especially with mistakes that may not be caught right away. It's easier to test cases that are supposed to work as intended versus ones that may break the code. For example, if the program is asking for a number, we could test with a letter, or if a program is asking for numbers 1 through 10, we enter in 0, 11, or -5. Take the time to ensure that you have tested those edge cases as well as the completely incorrect cases. Review the example of a good entry for Part 4 in the Example Java Journal Submission document and add your entry for Part 4 to your Java Journal.

SUMMARY

In this lesson, you tested the demonstration program by **checking for errors**. You first checked for invalid user inputs and needed to add some exception handling. Then, you noticed that the program did not repeat the input request after an invalid value. You needed to add a loop to continue the input request until a valid value was added. Remember that you can temporarily add extra **debugging** statements if you have errors or issues that you can't figure out; just don't forget to remove them later when the problem has been fixed. You then had an opportunity to see both a **bad example and a good example** for our **fourth journal entry**. The good example had both what you were testing for and how you solved the need. In the **Guided Brainstorming** section, you identified other examples of good test cases to ensure the program is working the way it is intended to.

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/**

Commenting Your Code

by Sophia

WHAT'S COVERED

In this lesson, we will learn about commenting our code. Specifically, this lesson covers:

1. Usefulness of Comments

Although you have previously placed comments within our algorithm, they may no longer be fully accurate, as our code could have changed during the process of coding and debugging. In addition, those comments may not be specific to our final program, so it's a good idea to revisit the commenting process. The reason they may not be specific is that we converted from the pseudocode, which was defined line by line rather than explaining each section.

Programming is rarely done entirely on our own. You will want to ensure that others know what is being done in the code. Even if this code is only meant for us, you will also want to make it easy to remember what the code is doing months or years later. Rather than trying to figure out what the code is doing, you can look at the comments and quickly remember. Let's go back to our demonstration program, modify the existing comments, and add additional comments.

---

2. Modifying the Program Comments

For most of our program, we have very detailed comments.

The DiceRoll and Game classes have many of the comments in place for all the methods for each, so that we're able to know exactly what they do.

Here is the DiceRoll class:

```
import java.util.Random;

public class DiceRoll {
  // Array with 2 elements to hold rolls of 2 dice
  private final int[] die = new int[2];
  // Sum of the 2 dice
  private int sum;
  // Random number generator from java.util library
  private Random randomNumberGenerator;

  public DiceRoll() {
    // Create random number generator
```

```
    randomNumberGenerator = new Random();
    // Argument of 6 means generator will produce a # in the range 0 - 5
    // + 1 adjusts the result to be in the range from 1 to 6
    die[0] = randomNumberGenerator.nextInt(6) + 1;
    die[1] = randomNumberGenerator.nextInt(6) + 1;
    sum = die[0] + die[1];
  }

  // Accessor method to get total of the 2 dice
  public int getSum() { return sum; }

  // Method to produce a String representation of the dice roll
  public String toString() {
    return "Dice: " + die[0] + "  " + die[1] + "  Sum = " + sum;
  }
}
```

For the Game class, we nearly have the same level of detail.

```
import java.time.LocalDateTime;
import java.util.ArrayList;

public class Game {
  private String playerName = "";
  private LocalDateTime gameDateTime;
  // ArrayList to hold dice rolls for player
  private ArrayList<DiceRoll> diceRolls;

  private int point = 0;
  private boolean keepRolling = true;
  // String to hold game result msg: "Win" or "Lose"
  private String gameResult = "";

  // Constructor to create Game object. Player name as String parameter
  public Game(String playerName) {
    gameDateTime = LocalDateTime.now();
    this.playerName = playerName;
    diceRolls = new ArrayList<DiceRoll>();
  }

  // Carry out dice rolls to play game.
  public ArrayList<DiceRoll> play() {
    // 1st roll for player
    DiceRoll roll = new DiceRoll();
    diceRolls.add(roll);
    int sum = roll.getSum();

    // Check for "craps" & resulting loss
    if(sum == 2 || sum == 3 || sum == 12) {
      gameResult = "Lose";
```

```
      keepRolling = false;
    }

    // Check for immediate win
    else if(sum == 7 || sum == 11) {
      gameResult = "Win";
      keepRolling = false;
    }
    else {
      // Sum of dice is now player's "point"
      point = roll.getSum();
    }

    // Loop for subsequent rolls
    while(keepRolling) {
      roll = new DiceRoll();
      diceRolls.add(roll);
      if(roll.getSum() == 7) {
        gameResult = "Lose";
        keepRolling = false;
      }
      else if(roll.getSum() == point) {
        gameResult = "Win";
        keepRolling = false;
      }
    }
    // play() returns ArrayList of DiceRoll objects that make up the game
    return diceRolls;
  }

  // Get String indicating result ("Win" or "Lose")
  public String getResult() {
    return gameResult;
  }

  // Get the number of dice rolls for game
  public int getRollCount() {
    // Number of DiceRoll objects in the ArrayList is number of rolls
    return diceRolls.size();
  }

  public String getPlayerName() {
    return playerName;
  }

  public LocalDateTime getGameDateTime() {
    return gameDateTime;
  }
}
```

Near the end of the `play()` method, though, it would be a good idea to document what the returned object is.

↗ EXAMPLE

```
// play() returns ArrayList of DiceRoll objects that make up the game
return diceRolls;
```

The `getRollCount()` accessor method is already commented to explain how it does its work.

↗ EXAMPLE

```
// Get the number of dice rolls for game
  public int getRollCount() {
    // Number of DiceRoll objects in the ArrayList is number of rolls
    return diceRolls.size();
}
```

Some of the other accessor methods would benefit from comments. For instance, the `getGameDateTime()` method could use a comment to explain the date and time marks.

↗ EXAMPLE

```
// Return date and time for the start of the games for the game
public LocalDateTime getGameDateTime() {
    return gameDateTime;
}
```

Having a short snippet to explain the method is helpful. It's also a good idea to explain the detailed code.

You should include some detailed comments to explain the `while` loop and the `try and catch` blocks, including what they are doing. It does not have to be extremely detailed. However, having a general explanation will help. Here is the section of the code in main() that we worked on in the last lesson to handle incorrect input for the number of games to play. It included comments, but more will help, since the logic for the loop may seem counterintuitive at first (combining logical not (!) with false produces true).

```
// Track if input of requested # of games to play is valid
boolean inputValid = false;
int numberOfGamesToPlay = 0;

// Assume invalid input of number until it proves to be correct
// inputValid initialized to false, so not (!) will make it true and loop will run
while(!inputValid) {
    try {
        System.out.print("How many games would you like to play?: ");
        numberOfGamesToPlay = input.nextInt();
        // If previous statement doesn't throw exception, input valid
        inputValid = true;
    }
    catch(InputMismatchException ex) {
```

```
        // If program flow ends up here, loop variable is still false
        System.out.println("Not a valid number.");
        // Clear out input to remove \n
        input.nextLine();
      }
    }
    System.out.println(); // Add blank line in output
```

That should be clearer.

TRY IT

Directions: Try adding additional comments to all the modules in the demonstration program. When you are done, review the JAVA Journal Sample.pdf to see what was included for comments and see how close you were. Did you add more or less comments?

---

3. Adding the Fifth Journal Entry

THINK ABOUT IT

The commenting in your code should be something that has been done up to this point, but it's quite possible that you may not have included all of the key details when it comes making your code easier to follow.

3a. Bad Example of Journal Entry for Part 5

A bad entry for the journal and for everyday programming would be to not have any comments included. But even having some comments, if they are not easily understood or very limited, can have negative results if someone else is trying to understand the program's logic. And yes, this can include the programmer of the program as well. Maybe it has been a while since they have been "in" the code. With enough well placed and thorough comments, they should be able to get back "up to speed" rather quickly.

Here is a bad journal entry for Part 5. Note that this section of code is not the complete main():

```
//The main() the point of entry
public static void main(String[] args) {
   // Variables for stats
   int winCount = 0;
   int lossCount = 0;
   int rollCount = 0;

   File logFile = new File("game.log.txt");

   Scanner input = new Scanner(System.in);
   System.out.print("Enter Player Name: ");
   String playerName  = input.nextLine();

   System.out.println("\nRunning Sample Game: ");
   Game game = new Game(playerName)
   ArrayList<DiceRoll> rolls = game.play();
```

```
    for(DiceRoll roll : rolls) {
      System.out.println("\t" + roll);
    }
    System.out.println("\nResult of Sample Game: " + game.getResult() + " in " +
           game.getRollCount() + " roll(s)\n");
    System.out.println();

    boolean inputValid = false;
    int numberOfGamesToPlay = 0;

    while(!inputValid) {
      try {
        System.out.print("How many games would you like to play?: ");
        numberOfGamesToPlay = input.nextInt();
        inputValid = true;
      }
      catch(InputMismatchException ex) {
        System.out.println("Not a valid number.");
        input.nextLine();
      }
    }
    System.out.println(); // Add blank line in output
```

A better entry for Part 5 would have comments with more details on what the logic is doing. Although we don't have to comment on every single line, we should at least comment on each section of code to describe what it is doing.

3b. Good Example of Journal Entry for Part 5

Let's see what a good entry would look like:

```
// main() method is the entry point for program
public static void main(String[] args) {
    // Variables for stats
    int winCount = 0;
    int lossCount = 0;
    int rollCount = 0;

    // Log file for player statistics
    File logFile = new File("game.log.txt");

    // Scanner to read player name & desired number of games
    Scanner input = new Scanner(System.in);
    System.out.print("Enter Player Name: ");
    String playerName  = input.nextLine();

    // Run sample game first
    System.out.println("\nRunning Sample Game: ");
    Game game = new Game(playerName);
```

```
// ArrayList to track roles of dice in a game
ArrayList<DiceRoll> rolls = game.play();
for(DiceRoll roll : rolls) {
  System.out.println("\t" + roll);
}
System.out.println("\nResult of Sample Game: " + game.getResult() + " in " +
      game.getRollCount() + " roll(s)\n");
System.out.println();

// Track if input of requested # of games to play is valid
boolean inputValid = false;
int numberOfGamesToPlay = 0;

// Assume invalid input of number until it proves to be correct
while(!inputValid) {
  try {
    System.out.print("How many games would you like to play?: ");
    numberOfGamesToPlay = input.nextInt();
    // If previous statement doesn't throw exception, input valid
    inputValid = true;
  }
  catch(InputMismatchException ex) {
    System.out.println("Not a valid number.");
    // Clear out input to remove \n
    input.nextLine();
  }
}
System.out.println(); // Add blank line in output
```

Remember this was just an example of part of the `main()` method. The other classes in the application should also be documented with sufficient comments.

If we preview the Example Java Journal Submission document, we will see **ALL** components of our program with good comments added as the entry to Part 5.

With this example, each section of code is commented to describe what it is doing. As programs have more lines of code, we will want to ensure that the comments also increase to clearly describe the additional functionality that comes with more code.

---

4. Guided Brainstorming

When it comes to comments, you can never add too much, but you can add too little. In looking at the code now, are you still unsure about some aspects? If so, that's a great place to add comments. In addition, think about explaining the program to someone who may not read code, or perhaps we may be coming back to the program later. If you are going from the pseudocode to translate it to Java, it'll be easier to include the comments. However, as you have seen between the start and finished code, there's a lot more functionality, code, and comments as we march towards the finished program. It's always easier to comment as we go rather than leave them all until the end.

Let's go back to the drink order program. The code already includes comments from when it was written, but more comments are a good idea.

The added comments are in bold in the following code:

```
import java.util.Scanner;

public class DrinkOrder {
public static void main(String[] args) {
   // Scanner to read user input
   Scanner input = new Scanner(System.in);
   System.out.println("What type of drink would you like to order?");
   // Note use of new line \n to print 3 lines with 1 statement.
   System.out.println("1. Water\n2. Coffee\n3. Tea");
   System.out.print("Drink selection #: ");
   // String variable to hold drink details.
   // Initialized to "No drink chosen" in case user doesn't make a valid selection
   String drinkDetails = "No drink chosen.";
   int choice = input.nextInt();
   // Remove \n left in input to avoid problems with later inputs
   input.nextLine();
   // Drink menu choice 1 is water
   if(choice == 1) {
    drinkDetails = "Water";
    // Get options related to water
    // Hot or cold?
    // With ice?
    System.out.println("Would you like that 1) hot or 2) cold?");
    System.out.print("Enter temperature selection #: ");
    choice = input.nextInt();
    // Remove new line left in input stream to avoid problems with later inputs
    input.nextLine();
    if(choice == 1) {
     drinkDetails += ", hot";
    }
    else if(choice == 2) {
     drinkDetails += ", cold";
     System.out.print("Would you like ice? (Y/N) ");
     // Read input as a String
     String response = input.nextLine();
     // Extract 1st char
     char yesNo = response.charAt(0);
     // Y or y is yes, anything else interpreted as no
     // Allow either uppercase or lowercase response
     if(yesNo == 'Y' || yesNo == 'y') {
      drinkDetails += ", with ice";
     }
    }
    else {
```

```
    // Display error message. No temperature appended to String
    System.out.println("Not a valid temperature selection.");
  }
 }
 // Drink menu choice 2 is coffee
 else if(choice == 2) {
  drinkDetails = "Coffee";
  // Get options related to coffee
  // Decaf?
  // Milk/cream?
  // Sugar?
  System.out.print("Would you like decaf? (Y/N): ");
  String decafResponse = input.nextLine();
  // Extract response as single character
  char decafYesNo = decafResponse.charAt(0);
  // Allow either uppercase or lowercase response
  if(decafYesNo == 'Y' || decafYesNo == 'y') {
   drinkDetails += ", decaf";
  }
  System.out.println("Would you like 1) milk, 2) cream, or 3) none?");
  System.out.print("Enter choice #: ");
  int milkCreamChoice = input.nextInt();
  // Remove new line left in input stream to avoid problems with later inputs
  input.nextLine();
  if(milkCreamChoice == 1) {
   drinkDetails += ", milk";
  }
  else if(milkCreamChoice == 2) {
   drinkDetails += ", cream";
  }
  System.out.print("Would you like sugar? (Y/N): ");
  String sugarResponse = input.nextLine();
  // Extract response as single character
  char sugar = sugarResponse.charAt(0);
  // Allow either uppercase or lowercase response
  if(sugar == 'Y' || sugar == 'y') {
   drinkDetails += ", sugar";
  }
 }
 // Drink menu choice 3 is tea
 else if(choice == 3) {
  drinkDetails = "Tea";
  // Get options related to tea
  System.out.print("Type of tea: 1) Black or 2) Green: ");
  int teaChoice = input.nextInt();
  // Remove \n left in input to avoid problems with later inputs
  input.nextLine();
```

```
      if(teaChoice == 1) {
        drinkDetails += ", black";
      }
      else if(teaChoice == 2) {
        drinkDetails += ", green";
      }
      else {
        // Invalid selection - assume black tea
        drinkDetails += ", black";
        System.out.println("Not a valid choice. Assuming black tea.");
      }
    }
    // If drink menu selection not 1,2 or 3, end up here
    else {
      System.out.println("Sorry, not a valid drink selection.");
    }

    // Print out final drink selection
    System.out.println("Your drink selection: " + drinkDetails + ".");
  }
}
```

BIG IDEA

A good habit is to include comments for the following, at the very least:

- Classes
- Methods
- Loops
- Selection statements
- Try and catch blocks
- Complex lines of code

TRY IT

Directions: Now it's time for you to go back and add in comments if you haven't already done so. Remember that each segment should be commented to explain what the code is meant to do for a non-programmer. A good habit is to comment on classes, functions, methods, loops, conditional statements, try and catch statements, or any complex code segments. Review the example of a good entry for Part 5 in the Example Java Journal Submission document and add your entry for Part 5 to your Java Journal. Remember this should include all parts of your program if it contains multiple (reusable) components.

SUMMARY

In this lesson, we discussed again how **useful comments** are when added to our code. If someone is trying to figure out what a program does and the comments are either missing or very limited, this can make it very difficult to understand the program or programmer's logic. This can also apply to the

programmer too if it has been a while since they were in the program. You **modified and added comments** in the demonstration program to add more clarity on how the program is working. You then had an opportunity to compare a **bad example and a good example for the fifth journal entry**. Finally, in the **Guided Brainstorming** section, you saw some more good examples of commenting with the drink order program.

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/**

Course Wrap Up

by Sophia

WHAT'S COVERED

In this lesson, you will cover everything needed to finish the Java Journal and submit it as the Unit 4 Touchstone. Specifically, this lesson covers:

1. Adding the Sixth and Final Journal Entry

So, your Java Journal is nearly complete! There is only one item left to add to the journal and that is the Replit join link. Time to allow others to see your great work.

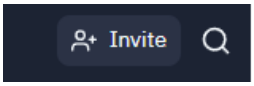
Follow the steps below to get your Replit join link.

HINT

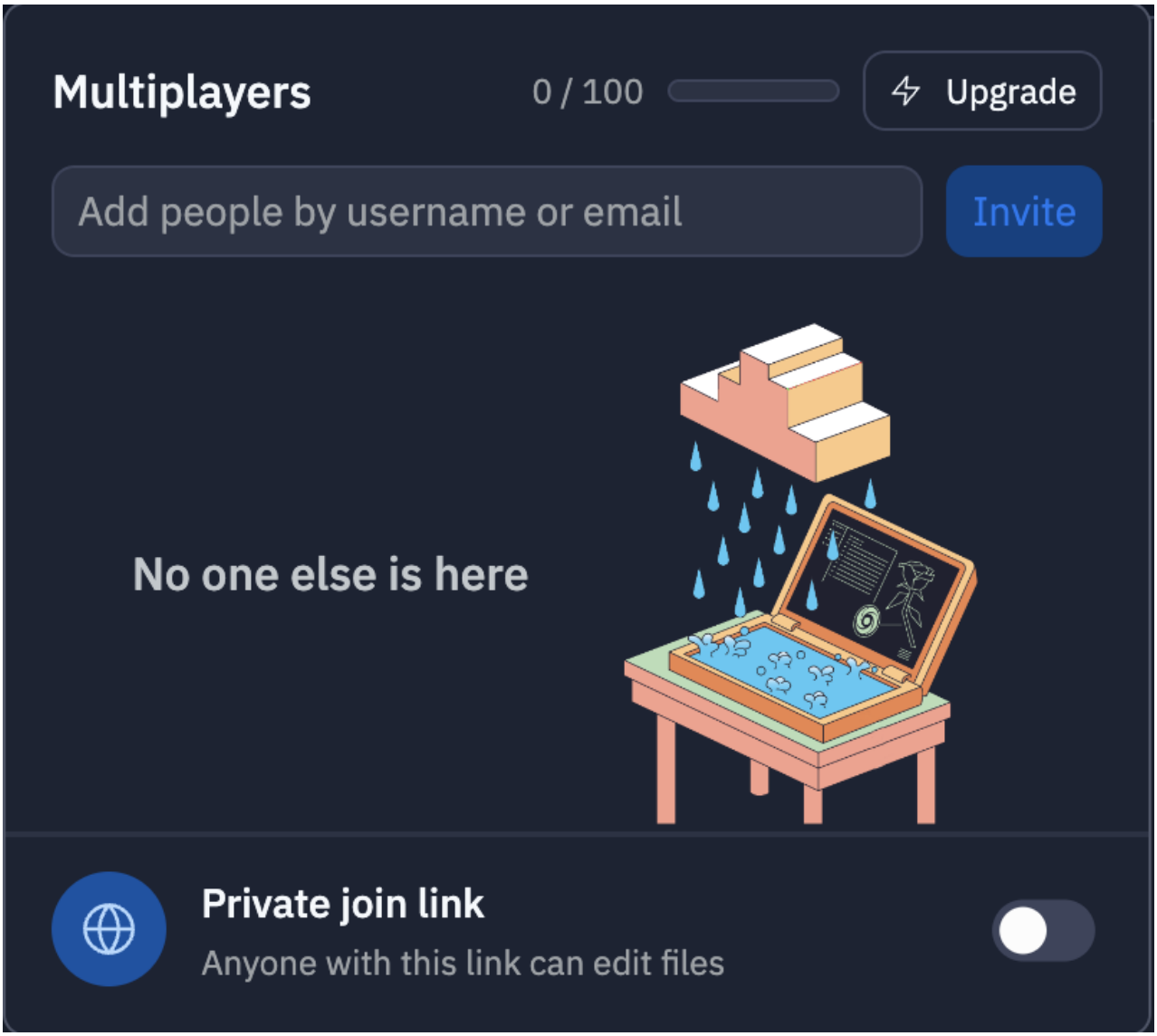
Visit the 'Help Video' section of the first lesson in Unit 4 (Java Touchstone Overview) to watch a demonstration of the process for generating your Replit join/share link, if needed.

STEP BY STEP

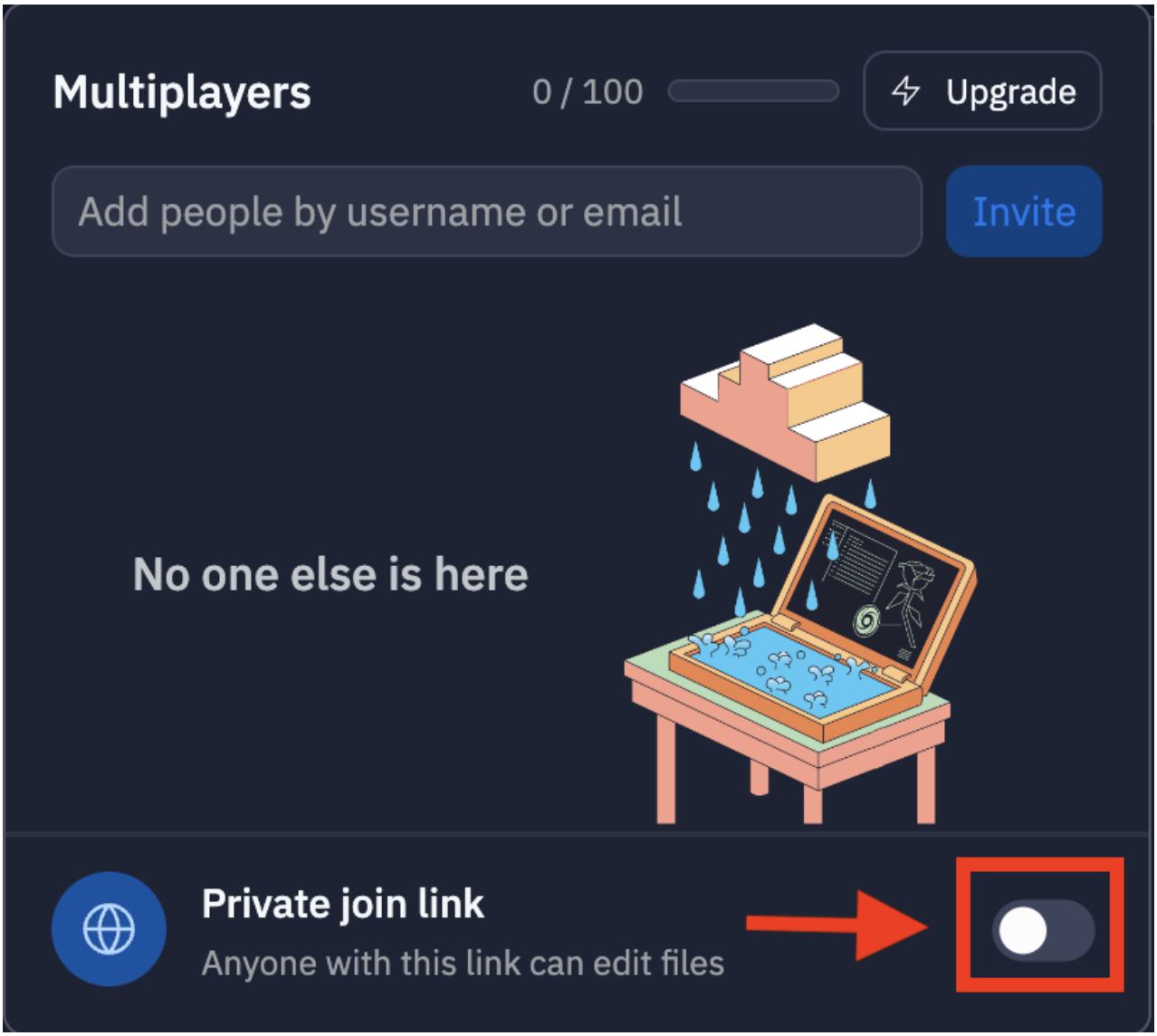
1. In Replit and your project program, select the 'Invite' button in the top right-hand corner of the screen.



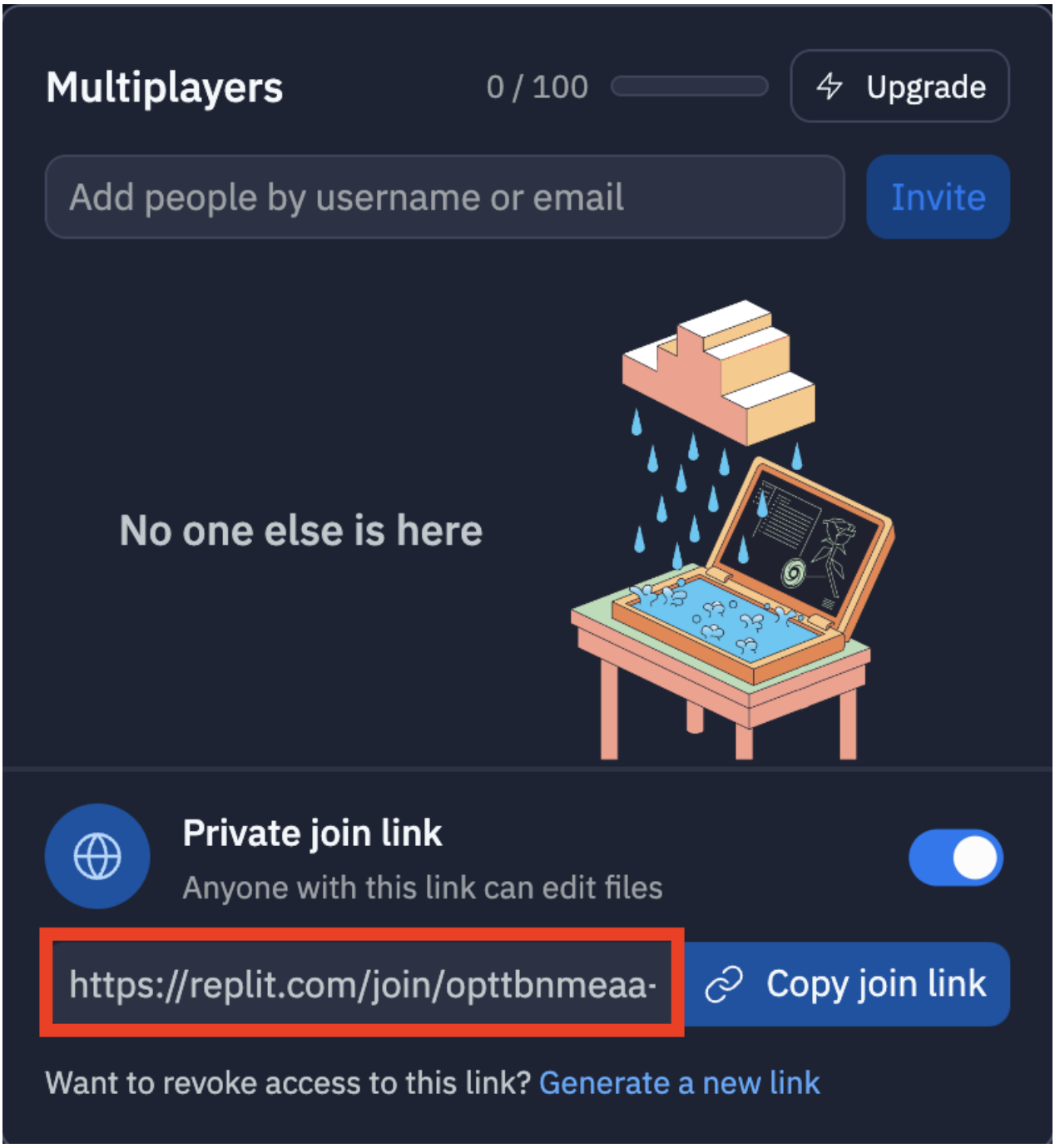
2. A pop up window will appear with various options for sharing your work.



3. Click the toggle button at the bottom of this window.

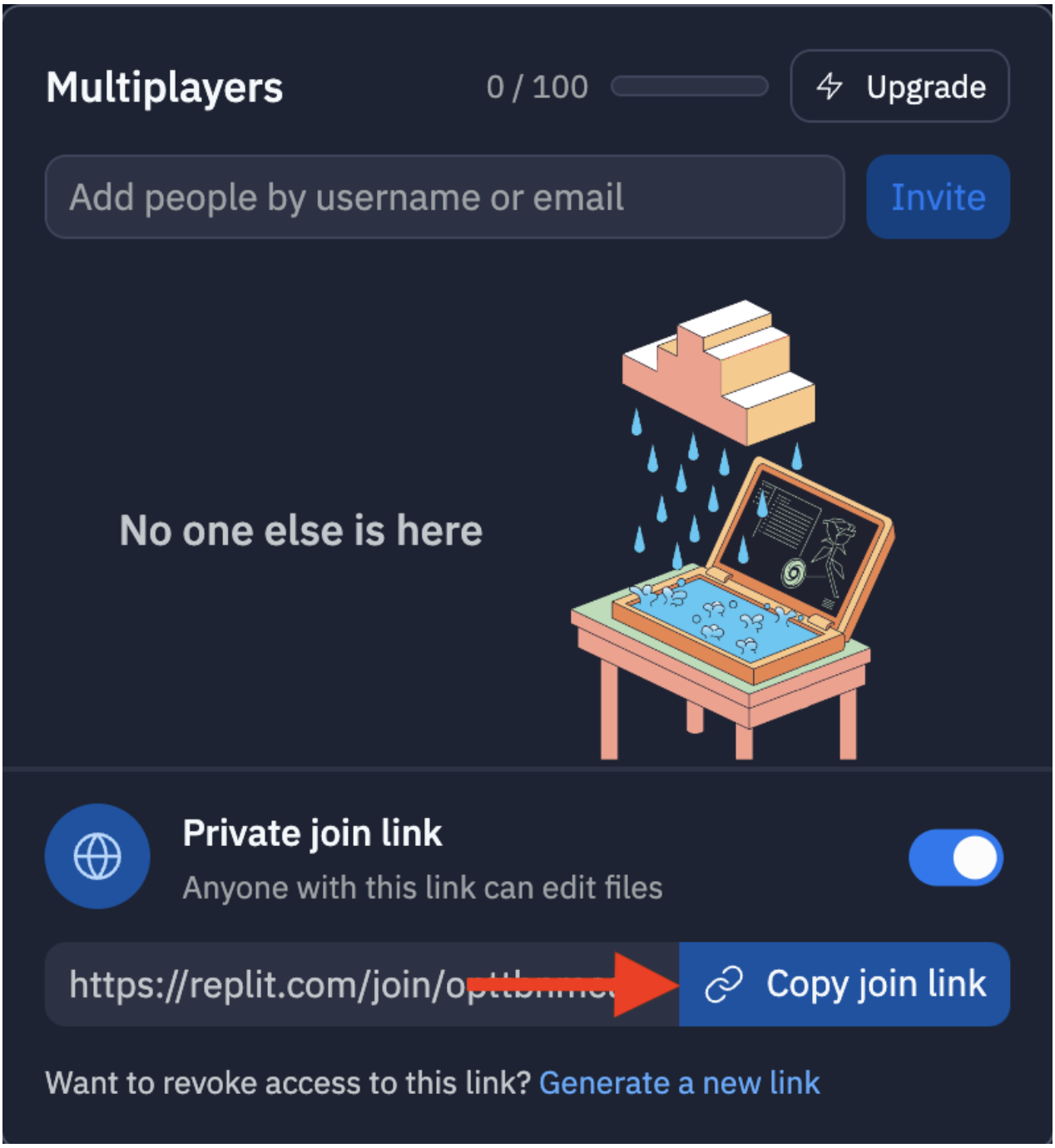


4. A join link will be generated.



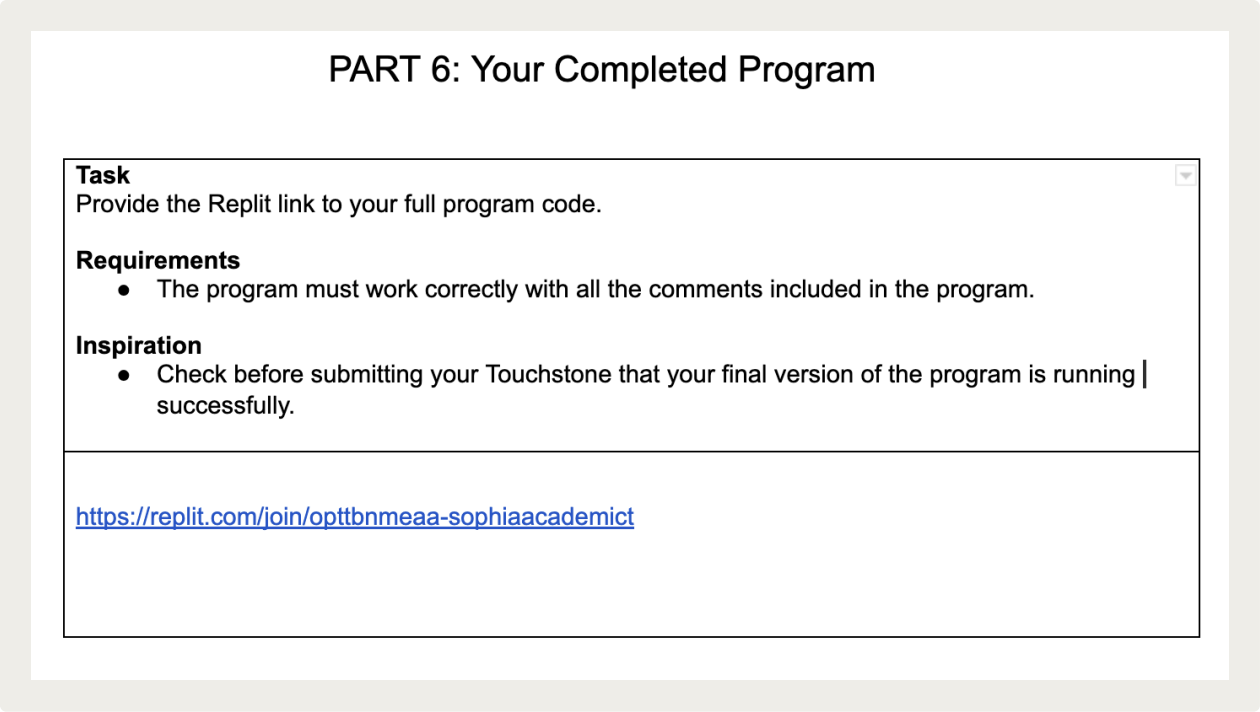
IMPORTANT: Each time you generate a join link, the previous link will not work anymore. Make sure that only the final join link is added to your Java Journal. If the graders cannot access your Replit Program, you will not receive a grade for this project.

5. Click the 'Copy join link' button to copy the link to your clipboard.



6. Add this copied Replit join link as your sixth journal entry.

↗ EXAMPLE


PART 6: Your Completed Program

Task
Provide the Replit link to your full program code.

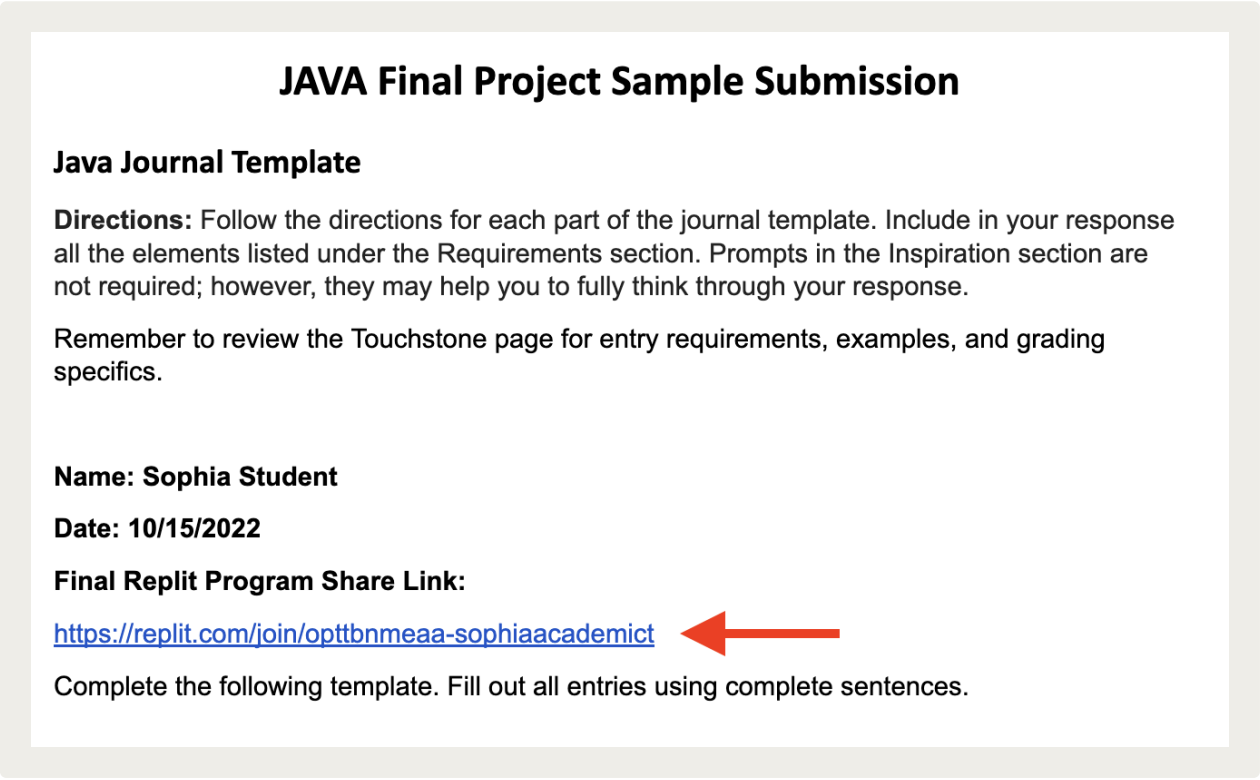
Requirements
- The program must work correctly with all the comments included in the program.

Inspiration
- Check before submitting your Touchstone that your final version of the program is running successfully.

https://replit.com/join/opttbnmeaa-sophiaacademict

5. Remember to add this link to the first page of your Java Journal as well.

↗ EXAMPLE


JAVA Final Project Sample Submission

Java Journal Template

Directions: Follow the directions for each part of the journal template. Include in your response all the elements listed under the Requirements section. Prompts in the Inspiration section are not required; however, they may help you to fully think through your response.

Remember to review the Touchstone page for entry requirements, examples, and grading specifics.

Name: Sophia Student

Date: 10/15/2022

Final Replit Program Share Link:

https://replit.com/join/opttbnmeaa-sophiaacademict

Complete the following template. Fill out all entries using complete sentences.

TRY IT

Directions: If you are ready, review the steps above and generate your Replit join/share link. Add the link to both areas on your Java Journal. With the two Replit join links in place on your journal, you should be ready to submit your Touchstone.

2. Submitting the Java Journal

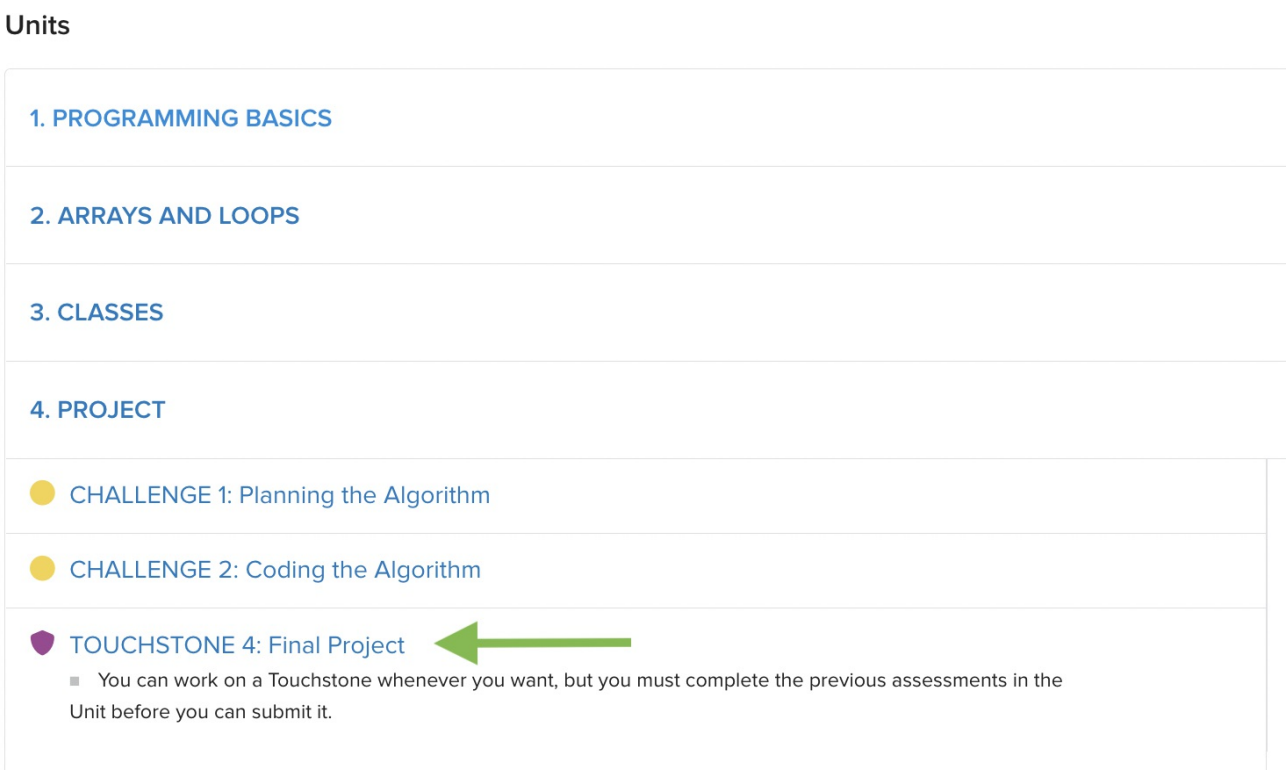
Your Java Journal is complete! Now it is time to review it before submitting. Make sure the following is complete:

1. Ensure that your Java Journal is a Word document. Only a .doc or .docx file is accepted.
2. Confirm that the first page of your journal contains your name, submission date, and the Replit share/join link that was generated in the topic above.
3. Double-check that all six journal entries are entered and that you have addressed the requirements for each entry.

IMPORTANT: Remember, you only have a one-time submission, so ensure your journal is complete before submitting.
If you are ready to submit your journal, please follow the steps below to submit your Java Journal for the final Touchstone.

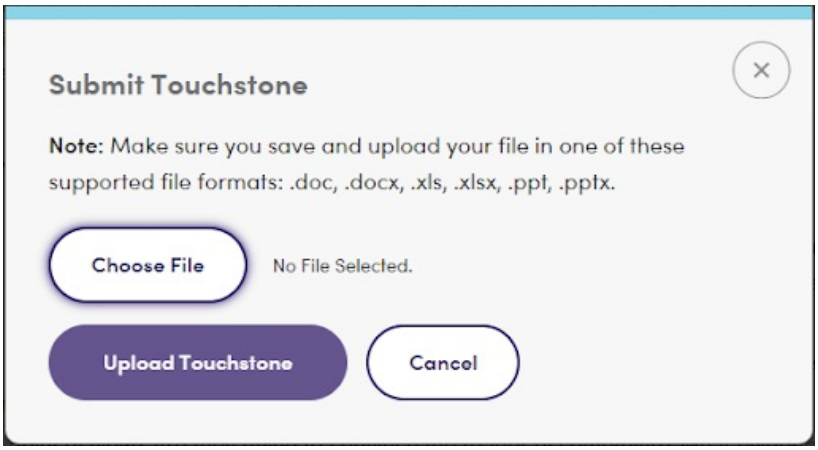
STEP BY STEP

1. Visit the Unit 4 Touchstone: Java Final Project's page.

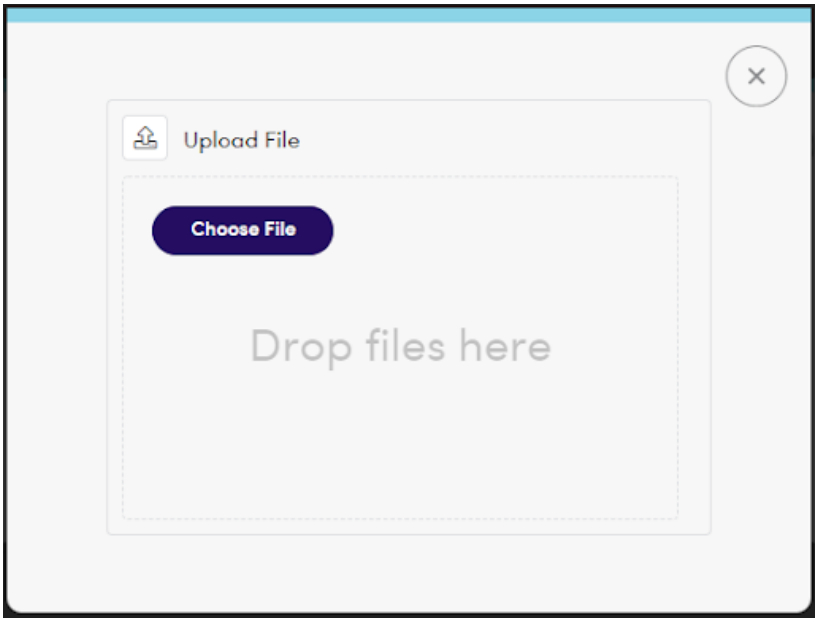


2. In the top right-hand corner of the page, select the 'SUBMIT TOUCHSTONE' button.

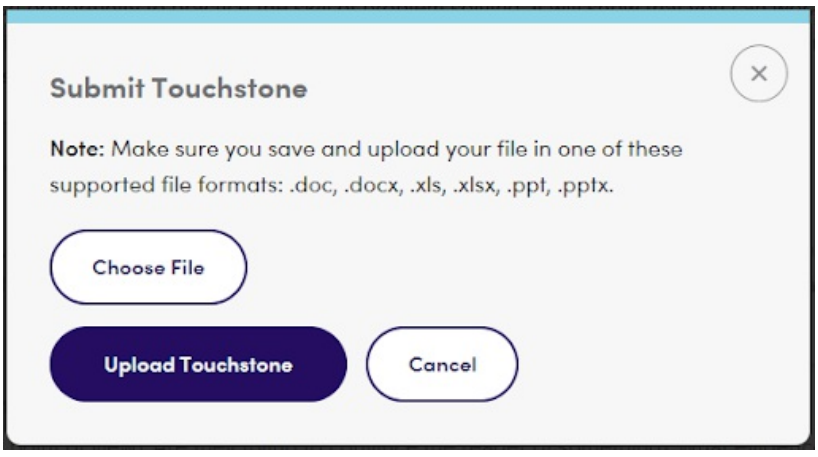
3. Once the Submit Touchstone pop-up window appears, select the 'Choose File' button.



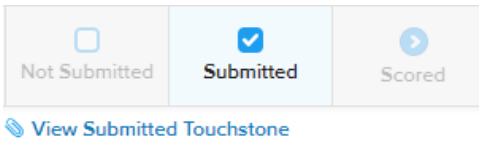
4. Here you have two choices to add the journal: you can select the 'Choose File' button again to search your system for the file to add, or you can drag and drop the file to this upload feature.



5. Finally, to submit the Touchstone, select the 'Upload Touchstone' button.



Note: On the top left side of the Touchstone page, you will notice three icons that indicate the status of the Touchstone. Once submitted, the ‘Submitted’ icon is now highlighted.



View Submitted Touchstone

TRY IT

Directions: When you have reviewed your Java Journal and feel it is complete, follow the steps above and submit your Touchstone.

3. What's Next

Congratulations on submitting your Touchstone! This also marks the completion of this course. Even though this was an introductory course to the Java programming language, this was not an easy course. You should be commended on your perseverance.

So what's next, now that you have some basic skills in Java programming?

Writing programs can be a very creative and rewarding activity. You can write programs for many reasons, ranging from making your living or solving a difficult data analysis problem to having fun or helping someone else solve a problem. And since you've completed this class, you can begin to be both the programmer and the end user of your programs. As you gain skill as a programmer, and programming feels more creative to you, your thoughts may turn toward developing programs for others.

THINK ABOUT IT

Next time that you do something manually, think about how you could potentially automate it to make it more efficient. There are always ways to help you avoid making constant mistakes and errors if you have to do things manually. Perhaps it could be creating a user registration list, or making a fantasy football league, or getting a list of guests for a party, or building a grocery list or another game. The possibilities are endless, and if you can think about it, you can build it!

Computer programming is an art form. In fact, one of the best books on programming is called "The Art of Computer Programming," written by one of the seminal leaders of computing, Donald Knuth. He started this book, which turned into a series of them, way back in 1968, and although this is an old work now, the principles introduced in it are still applicable today. Write your programs as if you are painting a work of art. Keep them simple, but make them elegant like a work of art. The world will appreciate your contribution.

THINK ABOUT IT

Remember as well that typically, you will have many other developers to work with, so don't worry if a program seems daunting to you. A team environment can make it much easier when you're building smaller parts at a time. Go out and build some programs!

SUMMARY

In this lesson, you followed some steps to obtain the Replit share/join link and **added that as the sixth and final journal entry**. Since each part (all entries) were now complete, you were ready to submit the journal. After ensuring everything was filled out on the journal, you were able to **submit your Java Journal** using the steps provided and the video, if needed. Finally, you started to explore and ask yourself **what's next**, now that you have completed a course in basic Java programming.

Congratulations and best of luck in your programming journey!

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/**