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:

Table of Contents

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:

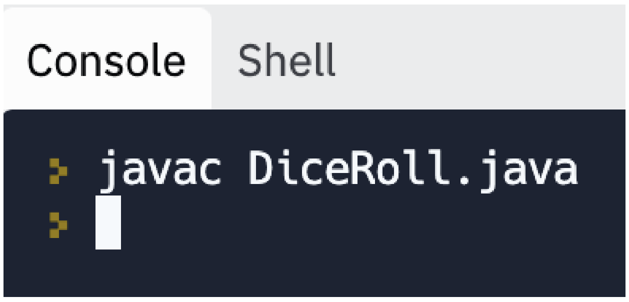
```
import java.util.Random;

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:



```
> javac DiceRoll.java
> 
```

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.

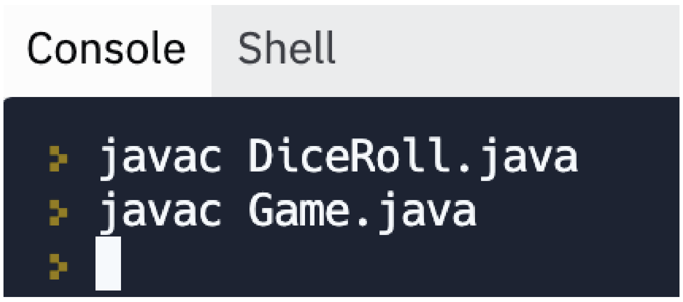
```
import java.time.LocalDateTime;
import java.util.ArrayList;

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:



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

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.

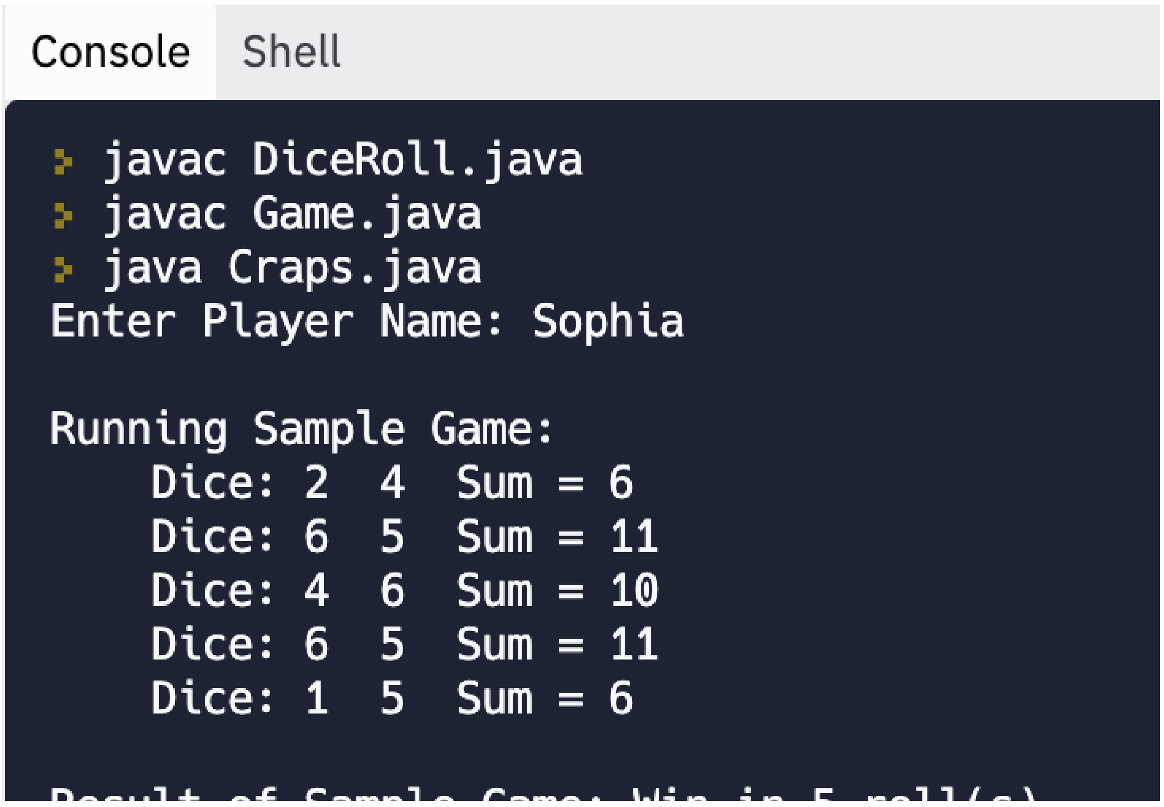
TRY IT

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.

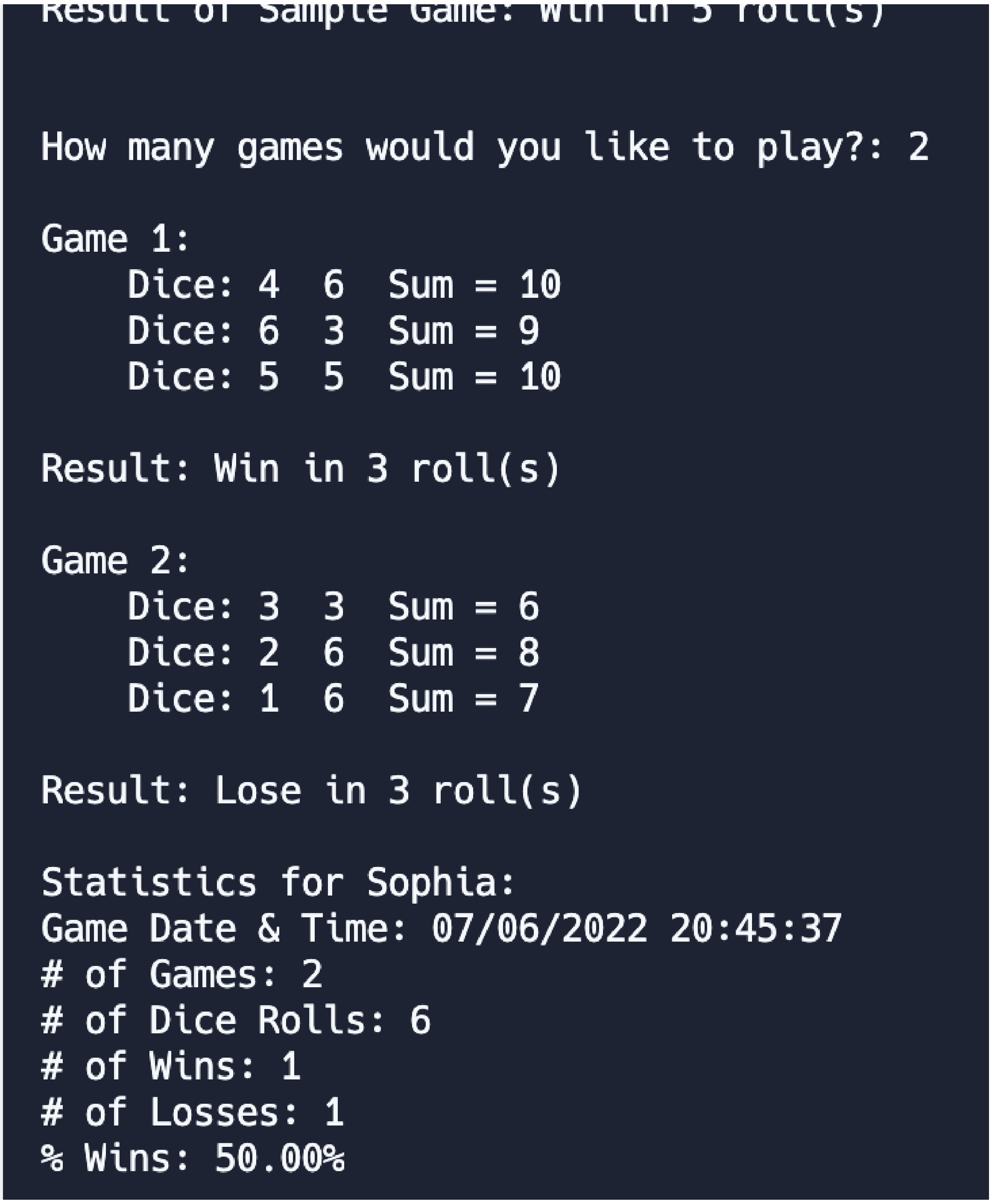
EXAMPLE

```
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):

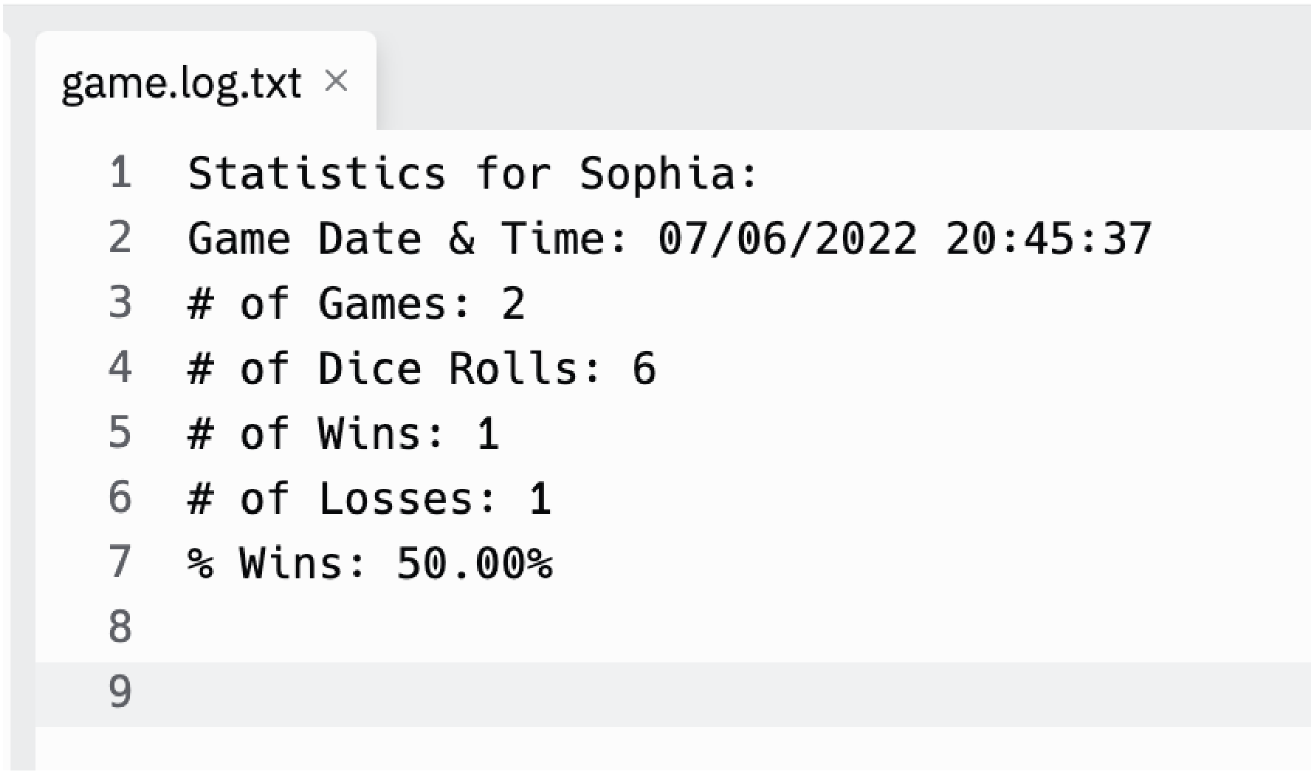

Console Shell

```
> javac DiceRoll.java
> javac Game.java
> java Craps.java
Enter Player Name: Sophia

Running Sample Game:
    Dice: 2  4  Sum = 6
    Dice: 6  5  Sum = 11
    Dice: 4  6  Sum = 10
    Dice: 6  5  Sum = 11
    Dice: 1  5  Sum = 6

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



```
Result of Sample Game: Win in 3 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):



```
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 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.

TRY IT

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/