Commenting Your Code

by Sophia

WHAT'S COVERED

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

Table of Contents

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/