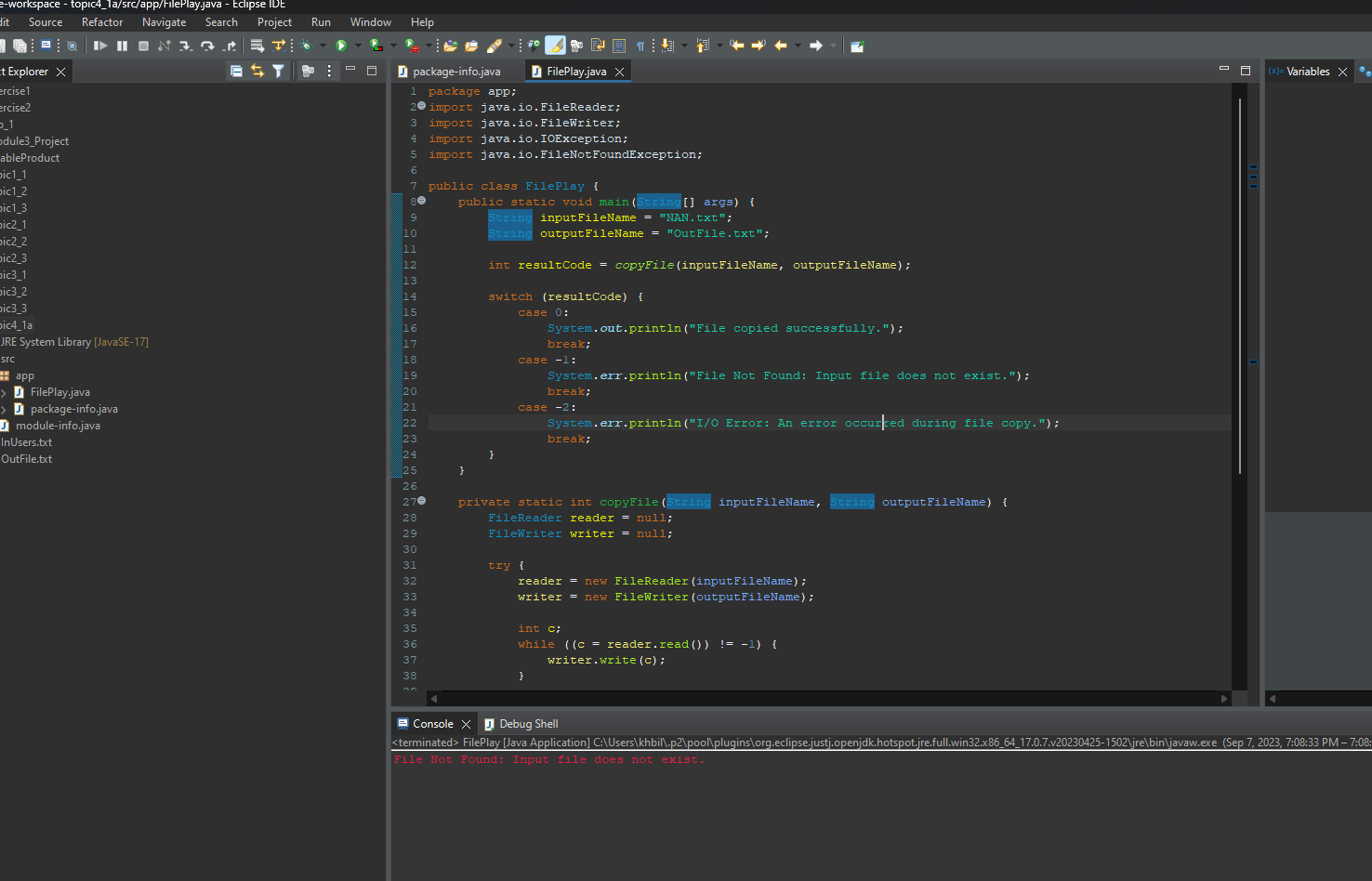
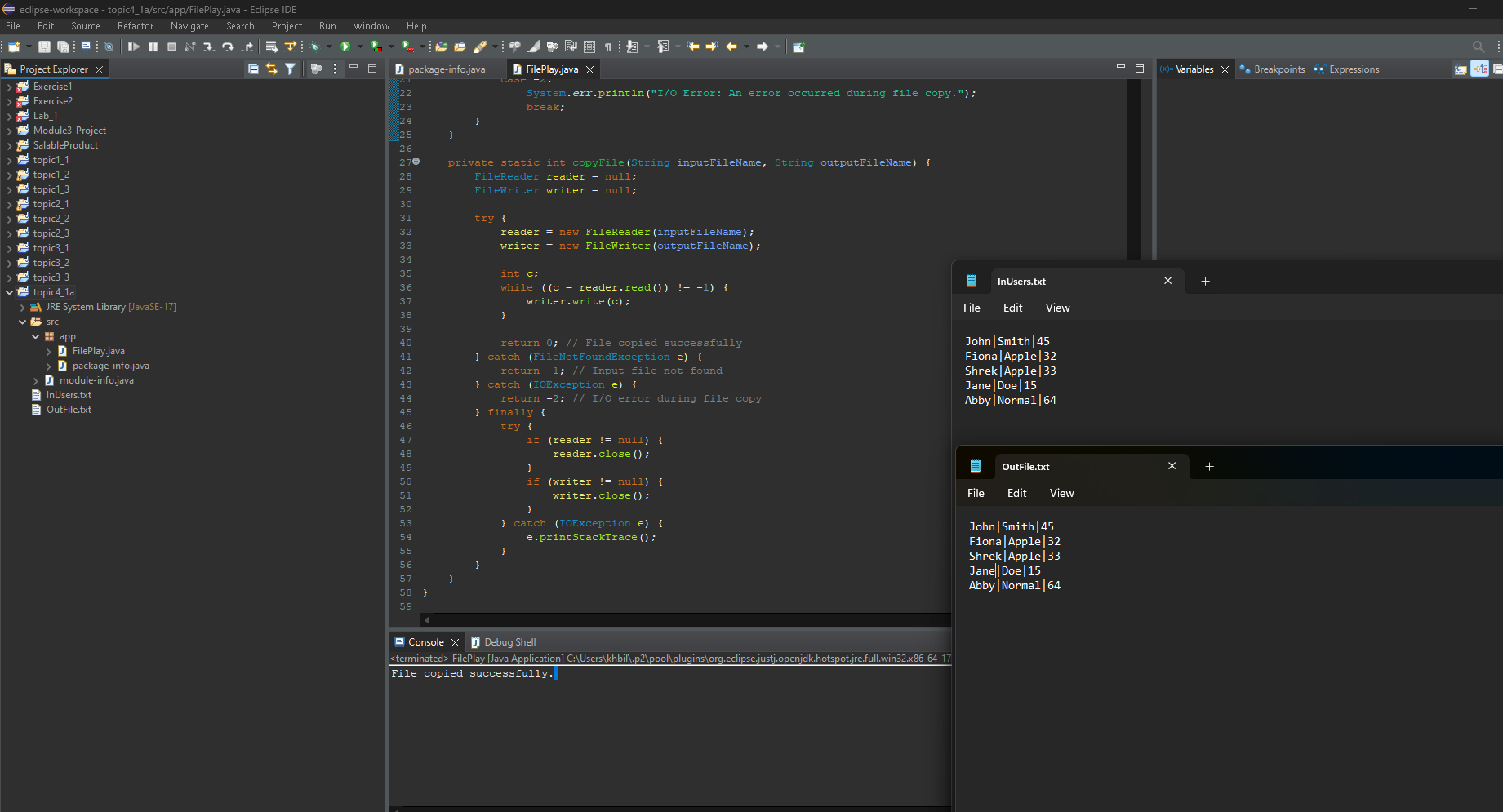
Topic 4

KariAnn Harjo

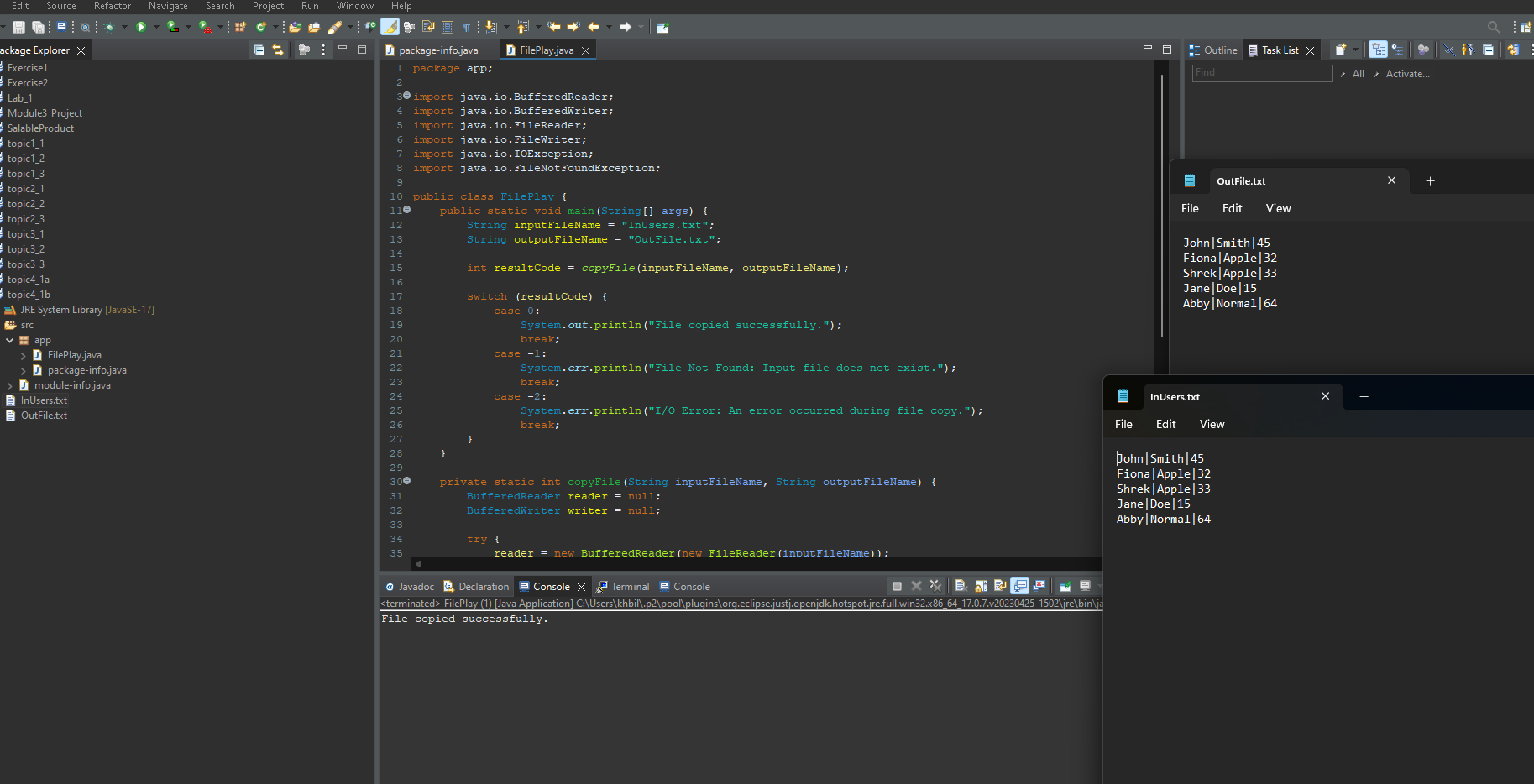
09/10/2023

Part 1: Reading and Writing Text Files

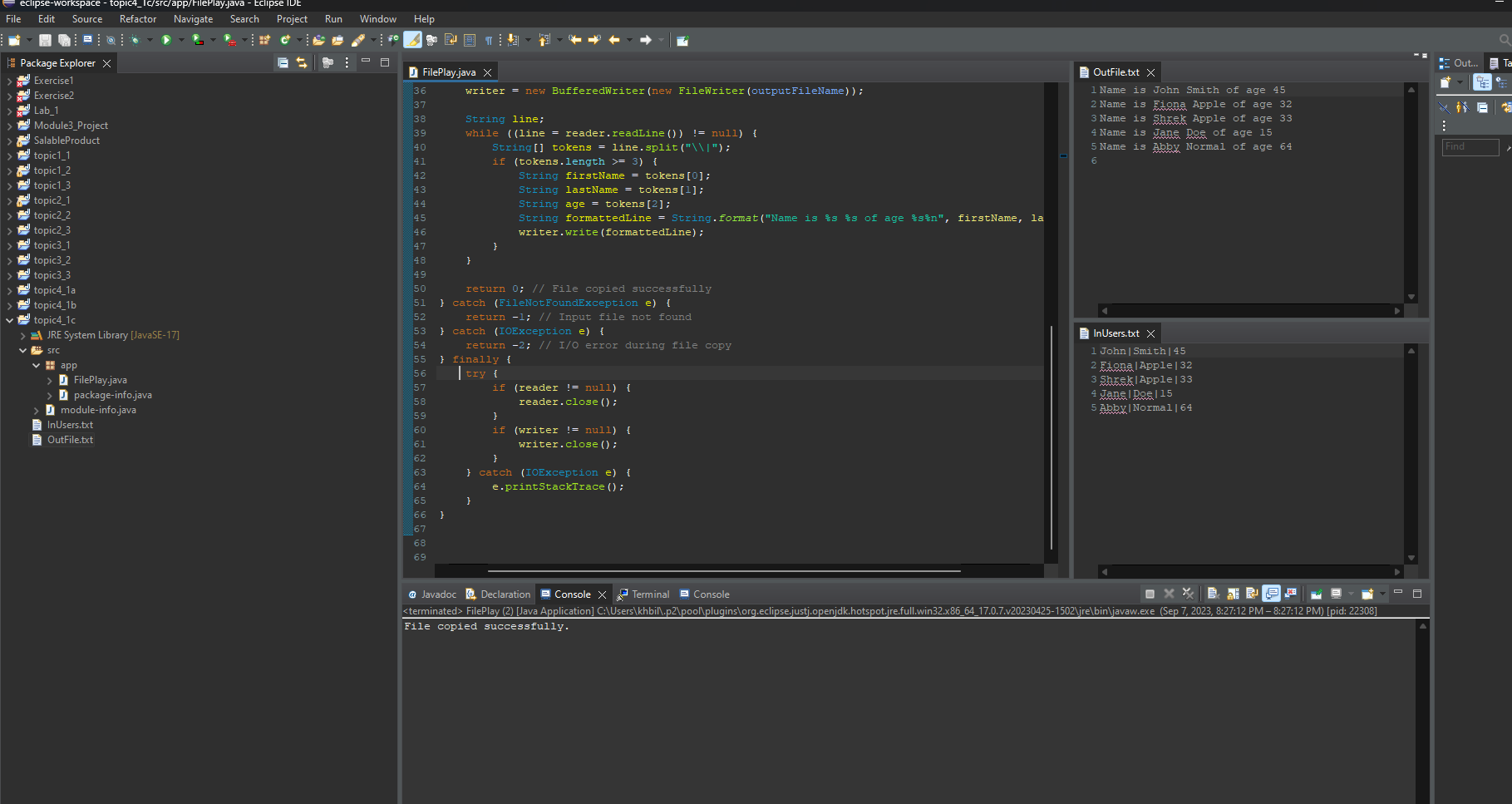
Part 1a:



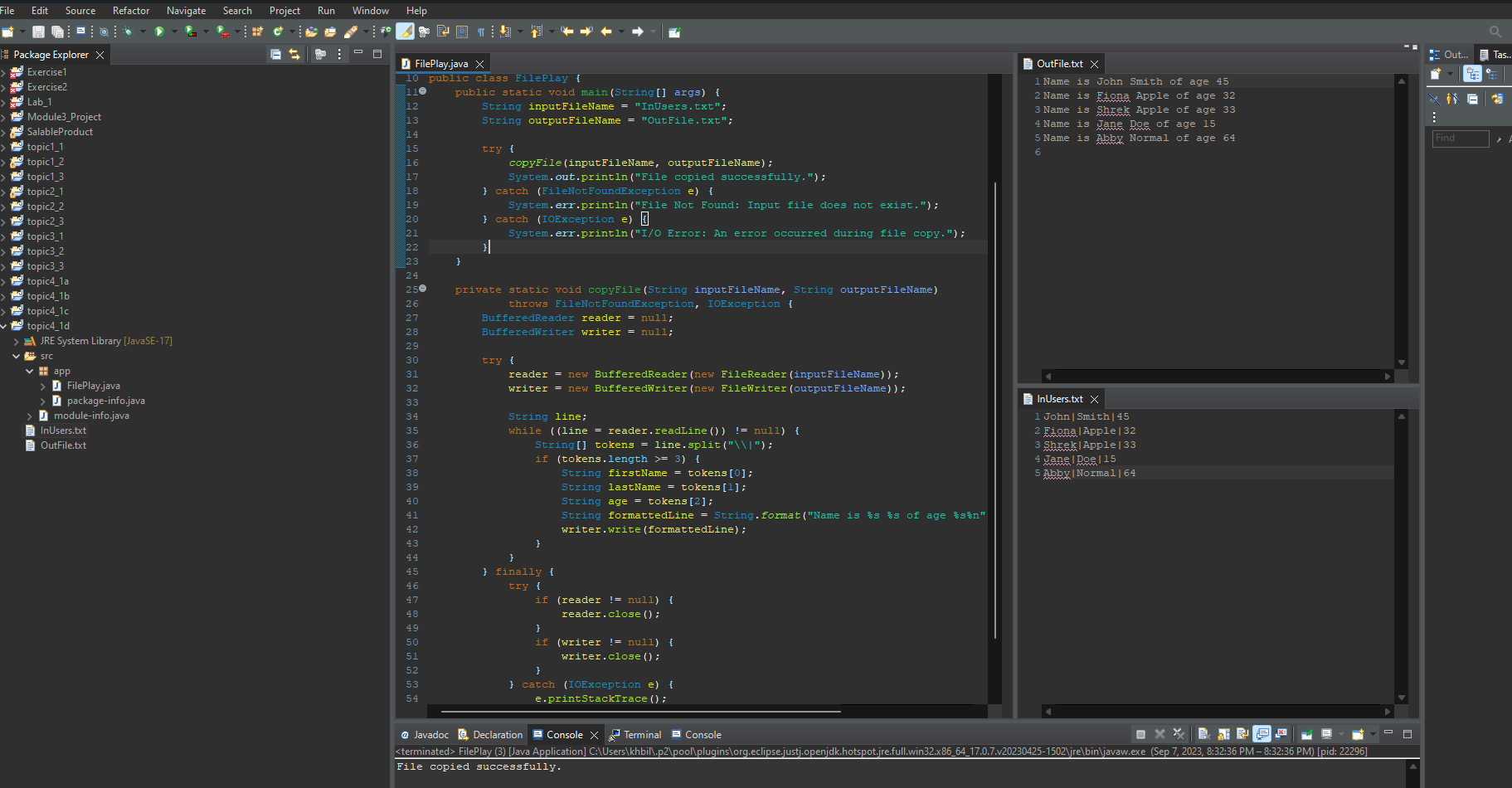
Part 1b:



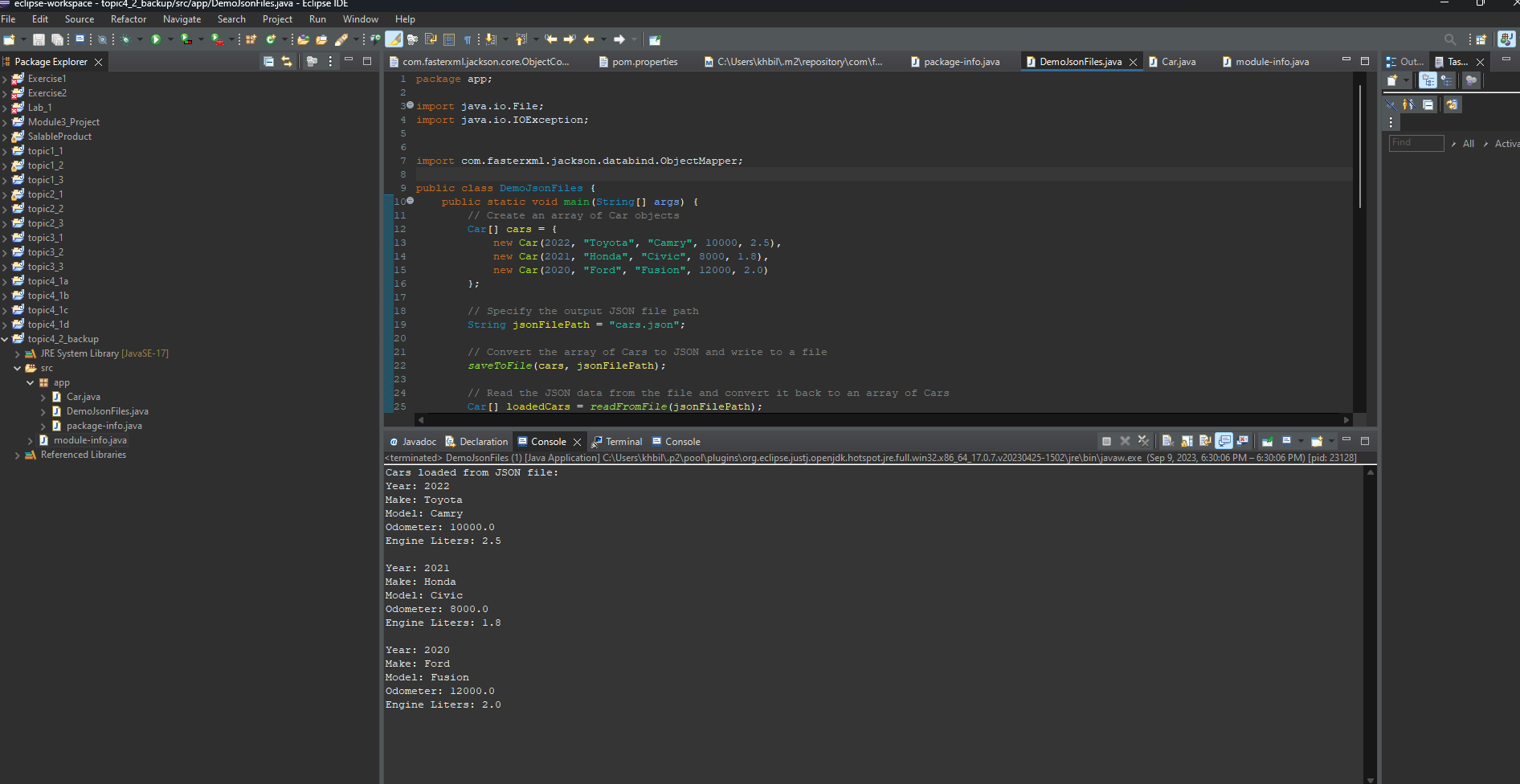
Part 1c:



Part 1d:



**Part 2: Reading and Writing JSON Files**



**Part 3: Research Questions**

1)

Exception handling is an important mechanism in Java, enhancing the reliability and robustness of code by gracefully managing errors and exceptional situations. Java classifies exceptions into two main categories: checked exceptions (compile-time exceptions) and unchecked exceptions (runtime exceptions). Understanding the differences between these two types of exceptions and when to design each type is essential for writing efficient and maintainable Java code.

**Checked exceptions** are those that the Java compiler obliges developers to handle explicitly, either through try-catch blocks or by declaring them in the method signature with the `throws` keyword. These exceptions typically signify anticipated, recoverable errors, such as file I/O issues or network connectivity problems. By mandating explicit handling, checked exceptions ensure that specific error conditions are addressed during program execution, promoting code reliability.

One key advantage of checked exceptions is their ability to guide developers in designing robust and resilient code. They act as safety nets, forcing programmers to consider error scenarios and implement appropriate recovery strategies. This results in more maintainable code and improved fault tolerance.

In contrast, **unchecked exceptions**, often referred to as runtime exceptions, do not require explicit handling by the compiler. These exceptions extend from the `RuntimeException` class and typically represent unexpected and, frequently, unrecoverable errors. Examples include null pointer exceptions or array index out-of-bounds errors.

Unchecked exceptions serve as red flags, indicating critical issues or bugs in the code. They are not intended to be caught and handled directly during regular program execution. Instead, they play a vital role during development and debugging, highlighting problematic areas that need attention. Unchecked exceptions offer a powerful means of identifying and addressing flaws in the code.

The decision of whether to design a checked or unchecked exception hinges on the nature of the error condition:

- **Checked exceptions** are appropriate when you expect and desire explicit handling of specific error scenarios. By enforcing error handling, they enhance code reliability and maintainability. Examples include scenarios like file reading or network communication, where you want to ensure graceful degradation in the face of issues.

-**Unchecked exceptions** are reserved for situations that signal severe problems or bugs in the code. These exceptions are not meant to be captured and processed as part of regular program flow. Instead, they act as beacons, drawing developers' attention to critical issues that demand immediate attention.

In conclusion, Java's exception-handling mechanism comprises two fundamental categories: checked and unchecked exceptions. Checked exceptions emphasize recoverable and expected errors, promoting reliable and maintainable code. Unchecked exceptions, on the other hand, pinpoint unexpected and often irreparable issues, serving as invaluable debugging tools. By strategically employing these two types of exceptions, Java developers can craft software that combines reliability with rapid issue detection, ultimately resulting in more robust applications.

2)  
Let's relate this to coffee because, honestly, I could use more right about now.

Pass by value:

In this scenario, we'll represent a coffee order using an integer to signify the number of sugars. We'll create a method to change the number of sugars, demonstrating pass by value for primitive data types.

public class CoffeeOrderExample {  
 public static void main(String[] args) {  
 int sugars = 2;  
 customizeCoffee(sugars);  
 System.out.println("Sugars after customization: " + sugars);  
 }

public static void customizeCoffee(int sugars) {  
 sugars = 0; // Set the local copy of 'sugars' to 0  
 }  
}

In this case, `sugars` remains 2 after invoking `customizeCoffee()` because the method operates on a copy of the integer, and changes inside the method do not affect the original value.

Pass by reference:  
Now, let's look at a pass by reference using an object, representing a coffee order with various attribute

public class CoffeeOrderExample {  
 public static void main(String[] args) {  
 CoffeeOrder order = new CoffeeOrder("Latte", 2, false);  
 customizeCoffee(order);  
 System.out.println("Coffee order after customization: " + order.toString());  
 }

public static void customizeCoffee(CoffeeOrder coffee) {  
 coffee.setSugars(1); // Modify the 'sugars' attribute of the original CoffeeOrder object  
 }  
}

In this case, the `coffee` object's sugars attribute becomes 1 after calling `customizeCoffee()`, showing that changes within the method affect the original coffee order.

These coffee order examples mirror pass by value for primitive data types and pass by reference (reference copy) for objects, where changes to object attributes persist beyond the method's scope. Understanding these concepts is vital when designing Java applications involving complex data structures like coffee orders.