Exercise 1: Book Festival (Kinds of Fields and Methods)

This exercise will help you to understand the difference between class and instance variables (also known as static and non-static fields), between class and instance methods (static and non-static methods), and between public and private methods.

Part (a) Look at the Book class defined below. Say which fields are class variables and which are instance variables. Do the same for the methods.

```java
public class Book {
    private String title;
    private double price;
    private static int totalNumBooks;
    private String author;
    public int numSold;
    public static final double MIN_PRICE = 2.0;
    public Book(String aTitle, double aPrice, String anAuthor) { ... }
    public double getPrice() { ... }
    private void lowerPrice() { ... }
    private static double newPrice(double oldPrice) { ... }
}
```

Part (b) Assume that the following (nonsense) code is in a static method (e.g., the main method) in the Book class. Find and fix (or cross out) all of the code that would cause compile-time errors.

```java
Book oneBook = new Book("Harry Potter", 19.95, "Rowling");
MIN_PRICE = oneBook.getPrice();
oneBook.lowerPrice();
double price = Book.getPrice();
System.out.println(oneBook.numSold);
System.out.println(MIN_PRICE + Book.newPrice(5.75));
```

Part (c) Now assume that the code given above is in a method that is not in the Book class. What new errors arise?
Exercise 2: Where’s Waldo? (Name Declarations and Uses)

A Java program can include many declarations and many uses of the same name. To understand what happens during execution, you need to know how to match declarations and uses. You should have learned some Java rules to help with this problem in class, so let’s try using those rules on code that is intentionally confusing.

The Waldo class is defined below (with line numbers included for reference). This class declares and uses the name Waldo many times (in a confusing way that you should never do yourself)!

```java
1. public class Waldo {
2.   private int Waldo;
3. 
4.   public Waldo(int Waldo) {
5.       this.Waldo = Waldo;
6.   }
7. 
8.   public void setWaldo(int val) {
9.       int Waldo;
10.      Waldo = val;
11.      this.Waldo = Waldo;
12.   }
13. 
14.   public void printWaldo(int Waldo) {
15.       for (int k=0; k<Waldo; k++) {
16.           System.out.println(this.Waldo);
17.       }
18.   }
19. 
20.   public void Waldo() {
21.       printWaldo(Waldo);
22.   }
23. }
```
**Part (a).** Start by finding every **declaration** of a Waldo in the code. Draw a square box around the declaration, including the name (Waldo) and the type it’s declared to have (except if it’s a class, since classes don’t have types).

**Part (b).** Now annotate each declaration that you found with one of the following numbers:

1. A declaration of a **class**.
2. A declaration of a **field** of a class (also called an **instance variable**).
3. A declaration of a **constructor** of a class.
4. A declaration of a **method** of a class.
5. A declaration of a **formal parameter** of a method.
6. A declaration of a **local variable** of a method.

**Part (c).** Now find and **circle** each **use** of the name Waldo in the code (that is, each occurrence of the name Waldo that is not a declaration of Waldo).

Match each use of Waldo with the corresponding declaration of Waldo by drawing a line to connect the use and the declaration.
Exercise 3: Designing a Class (instance vs class variables and methods)

Part (a): For this exercise, we will design a Student class. Choose the fields (variables) and methods of the class based on the information given below. For each field and each method, choose an appropriate name and type, decide whether it should be static or not, and decide whether it should be final or not. Then write the class definition.

- Every student has a first name, a last name, a GPA (a number between 0.0 and 4.0), and a 4-digit ID number. The first student’s ID number will be 1000, the next student’s ID number will be 1001, then 1002, and so on.

- A Student object can be created given first and last names only (in which case the GPA is initialized to 0.0), or given first and last names and a GPA. In both cases, the student’s ID is initialized to be the next available ID.

- A student’s names and ID can be accessed, but cannot be modified. The GPA can be both accessed and modified.

- It should be possible to compare two Student objects to see if they are the same person. Two students with the same ID are considered equal.

- It should be possible to find out how many Student objects have been created so far.

Part (b): Now use a laptop to type in your class definition. Make sure it compiles, then write and run a test class, too.
Exercise 4: Logical Thinking and Debugging

For this exercise, we’ll think about what you can do when a logical error in your code causes a test to fail. How can you find that logical error?

One way to debug your program is to “think backwards” from the point of the failure (the place where a bad value was printed, or an exception was thrown): what might have happened just before the failure, and what might have happened just before that, and so on, until you get to the point of the actual mistake in your code.

For this exercise, we will practice thinking backward in the context of a chess game. Keep the following in mind:

• A player is not allowed to move into check. (Among other things, this means that two kings can never be next to each other horizontally, vertically, or diagonally.)

• A pawn normally moves one space forward. However, on its very first move it is allowed to move two spaces forward, and to capture an opponent’s piece it moves one space forward diagonally.

• If a pawn arrives at the end of the board, it is promoted: replaced by a queen, castle, bishop, or knight.
Part (a): Look at the board below, which shows a position in a game of chess. You are told that black moved last. Your job is to figure out what black’s last move was, and what was white’s move before that. There are two solutions: the easier solution only works if the person playing white is sitting at the *north* end of the board (i.e., the white pawn is moving south); the other (more difficult) solution works whether the person playing white is sitting at the north or the south end of the board (i.e., the white pawn can be moving either way). Note that the moves are not necessarily good ones, but they are all legal.
Part (b): Now look at the board below, which shows the final position of a game (the game is over because white has check-mated black).

This time, your job is to figure out whether the person playing white was sitting on the north or the south side of the board. To do that, you will need to figure out what white’s last move was. There are several possibilities, so to narrow it down, you’ll need to figure out what black’s last move before that could have been in each case. In all but one case you should find that in fact there would have been no possible move for black, and that should let you figure out where the person playing white was sitting.