Learning Outcomes

After completing this lab, a student will be able to

1. Implement a simple object class.
2. Apply appropriate access types to protect class data and provide for class behavior (encapsulation).
3. Write a class that works with a client class.

This lab has **seven (7)** checkpoints.

Clone or copy the Gitea repo, `F23-203/F23-203-l04.git`. Create a new local repo and copy the `BookClient.java` file into the new repo. The new local repo is where you should write all your code for the lab. You will push the code to Gitea at the end of the lab.

Execution tracing

1. Trace the execution of the following code to determine what is output.

   DO NOT use the compiler.

   The following lines of code appear in the `main` method:

   ```
   int[] numbers = new int[5];
   fillMe(numbers);
   System.out.println(Arrays.toString(numbers));
   ```

   Here is the `fillMe` method:

   ```
   static void fillMe(int[] inArray) {
       for (int i = 0; i < inArray.length; i++) {
           inArray[i] = i * i;
       }
   }
   ```

   What is output with the last line from the `main` method?

   ✓ Show the lab instructor your completed trace and explain your solution.

Writing your own class

For the remainder of the lab checkpoints, you will be writing a class to manage information about a book, and using that class with a client file.

In the `src` directory in the repo you check out from Gitea, there is a file named `BookClient.java`. This is a client class that you will use to test your implementation of the `Book` class. If you compile the file now, compilation will fail: the `Book` class doesn't yet exist. You need to write the code that will make the tests in the client file work.

2. Create a new file in the `src` directory named `Book.java`. Start by writing your file header comments at the top of the file.

   Create the `public` class named `Book`. Declare the following instance variables, making them `private` access:

   ```
   String title
   String author
   int pubYear
   double price
   int pages
   ```

   ✓ Show the instructor your class so far.

3. Add a parameterized constructor to your class. Your constructor will take the values of each of the instance variables as parameters to initialize the variables' values. You can look at the client code to see how the constructor is used. Remember that any method that the client needs to use must have `public` access.

   ✓ Show the instructor the updated version of your class.

4. Write the `toString` method for your `Book` class. The `toString` must return a `String` representation of the fields that looks like this:

   ```
   [The Hobbit, J. R. R. Tolkien, 1967, $3.95, 294]
   ```

   Now that we can produce some output, we can finally compile and test our `Book` class with the client. Be sure that only the 3 lines of code between the comments for Checkpoint 4 and Checkpoint 5 are uncommented (this should be the case if you haven't modified the file that you checked out).

   When you compile multi-file Java programs, most of the time **you only compile the client class** — in this case, `BookClient.java`. It is not necessary to compile `Book` separately. Java will fold the `Book` class into the compilation. To compile and run, simply type:

   ```
   javac BookClient.java
   java BookClient
   ```

   If you have implemented your class correctly, the program will compile, run, and produce some output.

   ✓ Show the instructor the `BookClient` class compiling and running with the `Book` class.

5. Add a default constructor to your `Book` class. A default constructor takes no parameters and initializes the values of all instance variables to their appropriate 0-equivalent value. Most often, any *object* type, like `String`, is initialized to `null`. Uncomment the code for Checkpoint 5, save/compile/run to test the default constructor.

   ✓ Show the instructor the `BookClient` class compiling and running with the `Book` class.

6. Add `getter` methods (*i.e.,* accessors) to your `Book` class. That is, write the methods `getTitle(), getAuthor(), getYear(), getPrice(), getPages()`. These methods must return the current value of the specified field without making any changes to the object. Getters are value-returning methods whose return type will match the type of the field it accesses.

   Uncomment the block of code for Checkpoint 6. Recompile and run the updated versions of `Book` and `BookClient`.

   ✓ Show the instructor the running `BookClient` class with the new functionality and your modified `Book` class.

7. Add `setter` methods (*i.e.,* mutators) to your `Book` class. That is, write the methods `setTitle(), setAuthor(), setYear(), setPrice(), setPages()`. These methods will allow the client to update the values of the fields. Setters nearly always have `void` return type, since the data flow is into the object.

   Uncomment the block of code for Checkpoint 7. Recompile and run the updated versions of `Book` and `BookClient`.

   ✓ Show the instructor the running `BookClient` class with the new functionality and your modified `Book` class.

   If you haven't already, create the remote repo for your files in your class organization on Gitea. Push your `Book.java` file to the Gitea repo.

   Turn in your checksheet to the instructor before you leave lab.