CS 3500 — Exam 2 Part A

Instructions:

- *This is a do-it-all-by-yourself exam. You are not allowed to communicate with any current and past students of the course in any way during the exam.*
- *You are allowed to use an IDE, or any other text editor on your computer. The only online resources you are allowed to use are the course web page, and the official Java documentation by Oracle.*
- *Please submit on time. The times will be enforced strictly by the server. If you try to submit in the final seconds and are unable to due to a slow server response, you will not be able to submit anything! You cannot use any late days, or otherwise submit later for partial credit.*
- *If you have a question, please contact your instructor privately over Teams (quicker) or email.*

Good luck!

Figure 1: Class and interface index

The questions on this exam are related to a system that can be used to compute federal income taxes for any person with an income in the US.

The provided code contains a `TaxableIncome` interface that represents a record of various earnings for an individual person, or a married couple. There are two types of supported incomes. “Ordinary” income is normal wages, tips, interest earned in savings accounts, etc. It is taxed at ordinary, higher income tax rates. “Qualified” income is income from sale of stock, sale of a home, qualified dividend on owned stock, etc. It is taxed at reduced, capital gains tax rates. The `SimpleIncomeRecord` provides an implementation of this interface. Meanwhile, the provided `TaxCalculator` interface represents a tax calculator with a method to calculate taxes for any `TaxableIncome` object. The `SimpleTaxCalculator` class provides an implementation that uses a constant tax rate for ordinary income, and another constant tax rate for qualified income. In reality, tax computation is significantly more complicated!

Please submit your answers in Part A for your section on the submission server.

1. (20 points) Read the documentation provided for the `SimpleIncomeRecord` (line 39 on p. 7) and `SimpleTaxCalculator` (line 103 on p. 9) classes. Now write one or more complete JUnit tests that verify that these two classes work as per their specification.
2. (15 points) There are at least two bugs in the provided code, due to which correctly and comprehensively written tests for them will not pass. In two sentences each, identify what each bug is and how to fix it.

The provided design has some limitations that prevent this program from being commercially viable (i.e., it is not attractive enough for anybody to buy).

Most tax filers have several sources of ordinary income (wages, tips, interest from savings accounts, etc.) and qualified income (profit from sale of stock, qualified dividend on purchased stock, etc.). It is unreasonable to assume that they will know the total incomes before they open their tax software. Many tax software programs ask the user to go through what they have step by step ("Do you have a salary," "do you have another salary," "did you earn qualified dividend on any stock you own," etc.). Thus the income record is built incrementally.

Computing taxes is also not as simple as these classes lead you to believe. Filers can reduce their taxable ordinary income using multiple “deductions” and “exemptions.” Filers can directly reduce their taxes by claiming “tax credits.”

The remaining questions will explain these terms in more detail, and give you related design tasks.

3. Deductions and exemptions:

   A deduction or exemption directly reduces the taxable ordinary income of the filer: tax is less because the taxable amount becomes less.

   A standard deduction is a fixed amount (different amount for individuals and married couples). For example, an individual income can take a standard deduction of $2000 on their overall ordinary income, while a married couple filing jointly can take a standard deduction of $4000 on their overall (joint) ordinary income.

   A personal exemption is a fixed amount per filer and dependent (e.g., if there is a personal exemption of $1000 per person, a married couple with two kids can claim a personal exemption of $4000 on their overall ordinary income).

   Deductions and exemptions are stackable (i.e., several can be claimed simultaneously).

   In this question, you must enhance the provided code and design to support standard deductions and exemptions. In particular, the following should be possible:

   1. You should be able to "add" a standard deduction to an existing income record. Your design for this should accept a deduction amount for individual, a deduction amount for married couples, and "apply" the appropriate one to the existing record.
   2. You should be able to "add" a personal exemption to an existing income record. Your design for this should accept the number of dependents and an exemption amount per person, and "apply" the exemption to the existing record.
   3. It should be possible to add support for other types of deductions and exemptions in the future.
   4. In some way it should be possible to add multiple deductions and exemptions to an income record.
   5. One should be able to calculate tax on such an income record in exactly the same way as now.
   6. For auditing purposes, you should be able to remember (in some way) each deduction/exemption that was applied rather than their total effect to the ordinary income, even after the tax has been calculated.

   (a) (20 points) Summarize your design ideas as a point-by-point list. Each item in the list must begin with "Add," "Edit," or "Remove" followed by a field, method, class or interface as applicable. Each item should be no longer than 2 sentences. Write supporting code for each item (e.g., if you are adding a new method, provide the method signature. If you are adding a field, provide the full declaration of that field). There is no need to provide entire implementations of methods.

   (b) (20 points) As a way to illustrate your design, write a complete JUnit test that starts an individual filer with $5000 in ordinary income and $1000 in qualified income, adds a standard deduction of $500, a personal exemption (1 kid) of $100, verifies the setup and ensures that the tax liability of $1225 is correctly computed (the amounts are correct; don't bother checking the math).

4. (25 points) Importing from other places

   Amit has written a simple program that he uses every year to do his taxes. His sources of income are simple, and he does not trust any cloud-based tax software with his data. The provided `AmitsTaxes` (line 121 on p. 9) class is his hacky-but-works implementation, updated with this year's numbers.

   Amit realizes that his tax situation is becoming complicated. It is no longer worth his time to correctly implement the doTaxes() method in his class. He has read about the software you are designing in this exam, and wishes to use it.

   Write code that will allow somehow importing his income record from an `AmitsTaxes` object into your design.

   1. Your design must encapsulate this process in some way (in a method, class, etc.).
   2. Don't make Amit do a lot of the work on his end. Amit wants to do this only because he does not want to spend time and effort to write code for taxes this year.
   3. Remember that Amit is not an employee of your company. He is a prospective customer, who happens to have some coding ability.
   4. In addition to creating Amit's income record, your software should give him the standard deductions and exemptions that he is eligible for (only the ones you have designed for above).
   5. After importing, it should be possible to compute Amit's taxes based on the existing design in your program that computes taxes.

   Your answer should contain all the code necessary to do this. If you are making any assumptions, please write them clearly in your code wherever applicable.

TaxableIncome.java

```java
package tax;

/**
 * This interface represents a set of operations for a taxable income profile
 * . A profile consists of two kinds of income: non-qualified income (e.g.
 * wages) and qualified income (e.g. capital).
 */
public interface TaxableIncome { //

  /**
   * Get the total ordinary income (income that is not treated at
   * special rates for tax computation)
   * @return the total non-qualified income
   */
  float getOrdinaryIncome();

  /**
   * Get the total qualified income (income that enjoys preferential tax
   * treatment)
   * @return the total qualified income
   */
  float getQualifiedIncome();

  /**
   * Return if this taxable income is for single person or married filing
   * jointly
   * @return true if this income is for joint filers, false otherwise
   */
  boolean isJointFiler();
}
```

SimpleIncomeRecord.java

```java
package tax;

/**
 * This class represents a simple income profile for an individual or a
 * married couple filing jointly. It accepts directly the total ordinary
 * income and the total qualified income, without the opportunity to modify
 * them later.
 */
public class SimpleIncomeRecord implements TaxableIncome { //

  private final float ordinaryIncome;
  private final float qualifiedIncome;
  private final boolean isJoint;

  /**
   * Create a SimpleIncomeRecord based on given data explicitly
   * @param ordinaryIncome the total ordinary income to be taxed at normal rates
   * @param q the total qualified income to be taxed at a preferential rate
   * @param isJoint true if married filing jointly, false otherwise
   * @throws IllegalArgumentException if invalid incomes are provided
   */

  public SimpleIncomeRecord(float ordinaryIncome, float q, boolean isJoint) {
    if ((ordinaryIncome<0) && (q<0)) {
      throw new IllegalArgumentException("Invalid␣income(s)␣provided");
    }
    this.ordinaryIncome = ordinaryIncome;
    qualifiedIncome = q;
    this.isJoint = isJoint;
  }

  @Override
  public float getOrdinaryIncome() {
    return this.ordinaryIncome;
  }

  @Override
  public float getQualifiedIncome() {
    return this.qualifiedIncome;
  }

  @Override
  public boolean isJointFiler() {
    return this.isJoint;
  }
}
```

TaxCalculator.java

```java
package tax;

/**
 * This interface represents a simple way to compute the taxes for a given
 * income profile. Implementations define specific details of this tax
 * computation.
 */
public interface TaxCalculator { //

  /**
   * Get the tax owed due to the provided income
   * @param income the TaxableIncome for which tax is computed
   * @return the tax owed
   */
  float getTax(TaxableIncome income);
}
```

SimpleTaxCalculator.java

```
package tax;

/**
 * This class represents a simple tax calculator. It imposes a constant tax
 * rate of 25% on ordinary (unqualified) income, and 15% on qualified income.
 *
 * This class guarantees that there will not be negative tax liability, i.e.
 * it will not claim that the government owes money to the filer. In such
 * cases it will return a tax owed of 0
 */
public class SimpleTaxCalculator implements TaxCalculator { //


  @Override
  public float getTax(TaxableIncome income) {
      if (income==null) {
        return 0;
      }
      return 0.25f*income.getOrdinaryIncome()
              + 0.15f * income.getQualifiedIncome();
  }
}
```

AmitsTaxes.java

```
package amit;

/**
 * My own custom tax situation. I just customize this every year
 */

public class AmitsTaxes { //
  //regular job, should be paid per Piazza post!
  private final float teachersSalary = 70000.0f;
  //my wife's smarter, she went to industry!
  private final float wifeSalary = 100000.0f;
  //the little cherubs...
  private final int numKids = 2;

  /* this year I added to my income...*/

```

```java
  //uber gig on weekends only! Ordinary income, but hey!
  private final float uber = 2000.0f;
  //amateur photographer: another gig, but ordinary income
  private final float photos = 3000.0f;
  //this year was good!
  private final float profitFromStockSales = 10000.0f;



  public float teachersSalary() {
    return this.teachersSalary;
  }

  public float uberIncome() {
    return this.uber;
  }

  public float saleOfPhotos() {
    return photos;
  }

  public float profitFromStockSales() {
    return this.profitFromStockSales;
  }

  public float wifeIncome() {
    return this.wifeSalary;
  }

  public int howManyKids() {
    return this.numKids;
  }

  public float doTaxes() {
    /*

    Every year I pull in the tax tables from IRS,
    and write code here to to compute my taxes.
    I'm a geek with too much time!

    But seriously, I need to use some tax software now.

    */

    return 0.0f;
  }

}
```