Storing an Inheritance

Due Tuesday, February 14 at 8 a.m.

CSE 1325 - Spring 2023 - Homework #4 - 1 - Rev 1

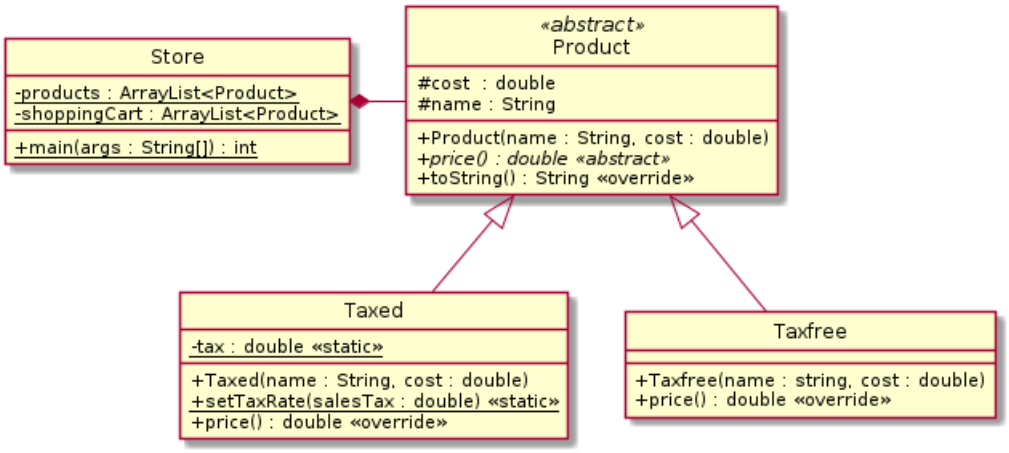
Assignment Overview

Inheritance is one of the three foundations of object-oriented programming, so let's dip our toes gently into the waters with a trip to the local grocery store.

Full Credit

In your git-managed directory **cse1325/P04/full_credit**, write an *abstract* Product class matching the following UML class diagram to represent items for sale in a grocery store. (NOTE: A class is *abstract* if it includes at least one *abstract* method, in this case, *price*.) Then derive two subclasses from it - Taxed and Taxfree - representing products for which sales tax is or is not due, respectively.

Note that we've used stereotypes to *emphasize* abstract classes and methods, static methods and fields, and overrides in this diagram, but this will not be the case for the exam or future diagrams. You should be able to recognize what needs to be abstract (italicized), static (underlined), and overridden (from context).



The fields of Product, all of which are immutable (i.e., can't be changed after construction), are as follows:

- **name** is just a string representation of the product that is for sale, e.g., "Milk" or "Apple".
- **cost** is the amount of cash in pre-tax dollars necessary to buy the item. It's a double, which we don't use for money in the real world due to rounding errors. But this isn't the real world, is it? :-)

The executable members of Product are as follows:

- The **constructor** should initialize name and cost to the matching parameters. Throw a runtime exception if cost is negative.
- **price** is abstract (it has no implementation in this class). It will be overridden in the subclasses.
- **Overloaded toString()**, streaming out e.g., "Cheese ($0.99) $0.99" showing the name, cost, and price() values respectively. *Always* show 2 digits to the right of the decimal point for money. Note that since Taxed and Taxfree are derived from this class, they may take advantage of this overload as well!

Taxfree adds no additional fields or methods.

- The **constructor** simply delegates to Product's constructor.
- **price** overrides Product.price() and simply returns the cost since no sales tax applies.

Taxed adds one additional static field and one additional static method.

- **salesTaxRate** is the sales tax rate for this product, for example, 0.0825. This field is *static*, that is, it is a single variable in one memory location for **all** instances of Taxed. Remember that a static field must be initialized in-line where it is declared, NOT in the constructor. In this case, set salesTaxRate to 0.0 (no tax). It may be changed later with the setTaxRate static method.
- The **constructor** simply delegates to Product's constructor.
- **setTaxRate** is a static method that simply assigns its parameter to the salesTaxRate static field. Remember, static methods can be called from the class without an object, for example, `Taxed.setTaxRate(0.0725)`.
- **price** overrides Product.price() and returns the total prices as cost x (1+salesTaxRate).

(At this point, you may be asking why we didn't simply use a boolean to indicate whether a product requires collecting sales tax or not. The answer is: For the educational benefit, of course. You need to practice inheritance, and this is one of the simplest examples around!)

Store provides the main method that predefines several taxed and taxfree grocery items.

It has two fields.

- Static ArrayList products a reference to each item available for purchase. No duplicates are expected here, though you needn't enforce this rule.
- Static ArrayList shoppingCart holds a reference to each of the items in products that has been selected for purchase. Multiple references (for buying more than one of the same item) is permissible.

In method main:

- Add at least 3 tax-free (milk, bread, cheese, eggs, fruit, and other staples) and at least 3 taxed (Poptarts, donuts, ice cream, and other less nutritious options as well as cleaning products and such) to the products ArrayList. More motivated students can find out which products require sales tax in Texas at https://comptroller.texas.gov/taxes/publications/96-280.php.
- **In the main loop:**
    - List all of the taxable and taxfree items you predefined (at least 3 of each).
    - List the contents of the shoppingCart, if any.
    - Ask the user to select an item to purchase. If the item number is negative, exit.
    - If positive, add the selection to the shoppingCart.

Handle all exceptions without an uncontrolled abort - that is, at a minimum, catch the exception, report it, and return an error code to the operating system if it is not recoverable. See Lecture 05.

Add, commit, and push all files.

Example Full Credit Output

Bonus

In Lecture 05, we discuss error handling and testing. You've demonstrated your skills with exceptions, so let's try a bit of testing!

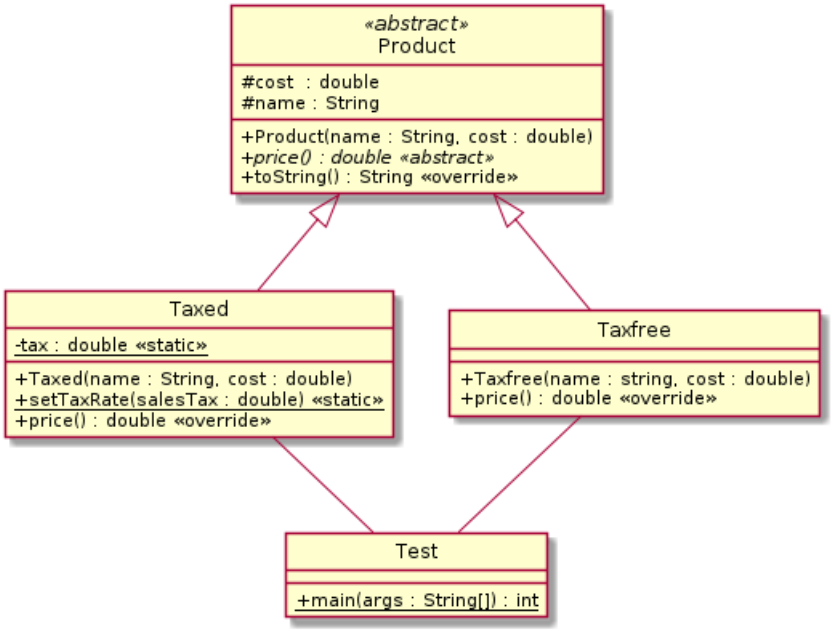
Write a simple regression test (that is, a second main() method in file Test.java) that performs 5 tests on our class hierarchy.

- Instance a Taxfree product, and verify that each of its fields (both inherited and local) were properly constructed and that toString() works with it. This is 1 test.
- Do the same for a Taxed product. Test both with the default (0.000% tax rate) and non-default tax rate (set by static method setTaxRate). This is 2 tests.
- For both Taxed and Taxfree, ensure that providing a negative cost results in a RuntimeError exception. This is 2 tests.

Remember: **Regression tests are fully automatic - no human interation! - and print NOTHING unless an error is found.** If one or more errors are found, report each one clearly but concisely on one or more separate lines. If any test fails, the last thing the test should print is simply "FAIL" on a separate line and return a non-zero result to the operating system, but if all tests pass, print nothing at all and return 0 (which Java does by default).

Some examples for inspiration or derision (as appropriate) may be found at Lecture 05.

Add, commit, and push all files.



Extreme Bonus

In Lecture 08, we learned about *interfaces* - pure abstract classes. Replace abstract class Product with *interface* Product and *implement* it in classes Taxed and Taxfree with identical behavior.

The class diagram will give you some hints, such as one approach to avoiding code duplication (DRY - Don't Repeat Yourself).

Use your regression test from the Bonus to ensure your program's behavior has not changed!