a1c # assignment

Assignment 1c – Bookstore Implementation

School of Engineering and Technology, University of Washington Tacoma
TCSS 305 Programming Practicum, Winter 2026
Value: 8% of the course grade

Due Date

Sunday, 27 January 2026, 23:59:59

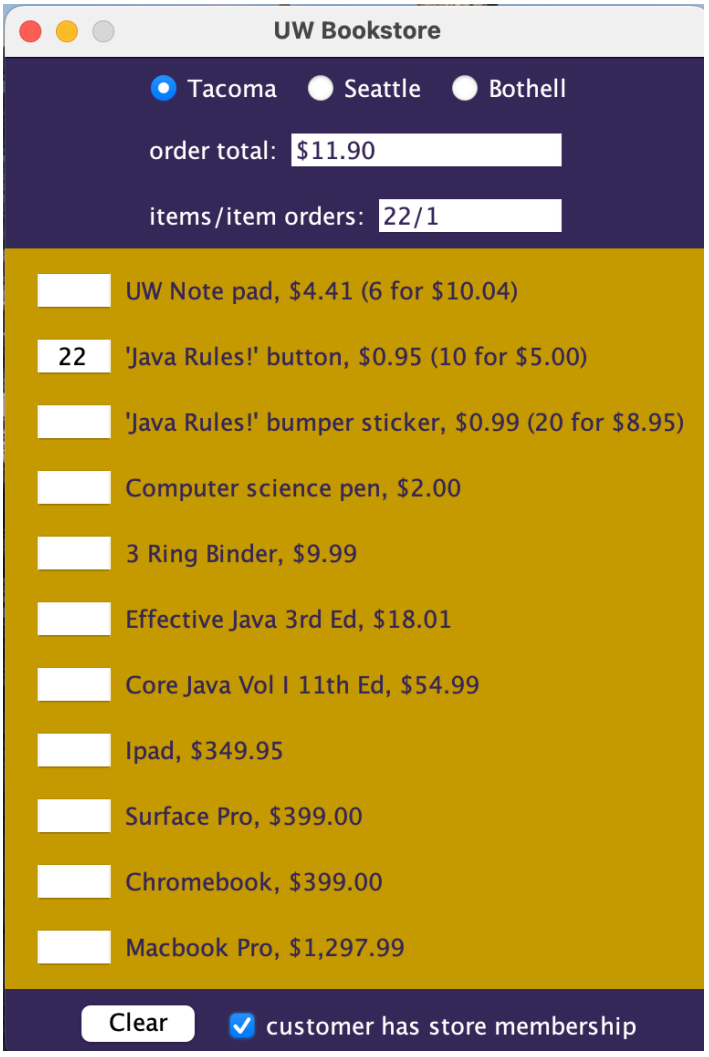
Description

This assignment gives you practice writing Java classes to a provided API and using Collections. You will write a set of supporting classes for a simple shopping cart. The professor has provided the Graphical User Interface (GUI) that will provide the "front end" or "view" of your program. You will write the back end (what is often referred to as the "domain-specific code" or the "model").

Prices are expressed as real numbers and quantities are expressed as integers (i.e., you can't buy 2.345 units of something). Notice that some of the items have a discount when you buy more. For example, Silly Putty normally costs $4.41 per unit, but you can buy 6 for $10.04. These items have, in effect, two prices: a single item price and a bulk item price for a bulk quantity. The bulk quantity discount is only used if the membership checkbox at the bottom of the GUI is selected.

When computing the total with the bulk item discount, apply as many of the bulk quantity as you can and then use the single item price for any leftovers. For example, the user is ordering 12 buttons that cost $0.95 each but can be bought in bulk at 10 for $5.00. The first 10 are sold at that bulk price ($5.00) and the two extras are charged at the single item price ($0.95 each) for a total of $6.90. If the user were ordering 22 buttons, the total would be $11.90 (two bulk quantities plus two extras).

At the bottom of the GUI there is a checkbox for a discount for customers who have a store membership. If this box is checked, the bulk pricing discount is applied to any items with bulk pricing as described above.



The UW Bookstore application showing items with bulk pricing and the membership checkbox.

🎯 Learning Objectives

By completing this assignment, you will:

- Practice writing Java classes to a provided API specification
- Implement classes using inheritance (extending abstract classes)
- Work with Java Collections (List, Map, etc.)
- Use BigDecimal for precise currency calculations
- Practice defensive programming with input validation
- Apply test-driven development by integrating previous tests

- Work with sealed classes and records (modern Java features)
- Practice Git branching and pull request workflows

✅ Before You Begin

Ensure you have completed the following:

- [ ] Completed Assignment 1b (your tests will be used in this assignment)
- [ ] Reviewed the API specifications in this document
- [ ] Familiar with BigDecimal operations
- [ ] Understand the equals/hashCode contract

Environment Setup: No additional setup required. Continue using your environment setup from previous assignments.

📦 Project Setup

GitHub Classroom

Accept Assignment

1. Click the GitHub Classroom assignment link above.
2. Accept the assignment to create your private repository.
3. Clone the repository to your local machine using IntelliJ:
   - Open IntelliJ and select **Get from VCS** (or **File → New → Project from Version Control**)
   - Paste your repository URL and click **Clone**
   - Click **Trust Project** when prompted
   - Wait for IntelliJ to index the project and resolve dependencies
4. Verify the project structure includes the starter code and GUI components.

Project Will Not Compile Initially

The project will not compile when first opened. Existing code relies on three missing classes (`StoreItem`, `StoreBulkItem`, and `StoreCart`) that you are required to create.

Create Your Development Branch

Before writing any code, create a development branch.

In IntelliJ:

1. Click **Git** → **New Branch** (or click the branch name in the bottom-right status bar)
2. Enter `dev` as the branch name
3. Ensure **Checkout branch** is checked
4. Click **Create**

Or from the terminal:

```
git checkout -b dev
```

All your work should be on this branch. Commit frequently as you progress. When you're ready to submit, you'll push this branch and create a pull request to merge into `main`.

Guide

Git Branching and Pull Requests — Understanding branches and the PR workflow.

📁 Project Structure

Before diving into the requirements, understand the model structure you'll be working with.

Class Hierarchy

The model uses a **sealed class hierarchy**:

```
Item (sealed interface) ← PROVIDED
└── AbstractItem (sealed abstract class) ← YOU IMPLEMENT (stub provided)
    ├── StoreItem (final) ← YOU IMPLEMENT
    └── StoreBulkItem (final) ← YOU IMPLEMENT

Cart (interface) ← PROVIDED
```

```
└── StoreCart (final) ← YOU IMPLEMENT

ItemOrder (record) ← PROVIDED
Cart.CartSize (record) ← PROVIDED (nested in Cart)
```

Guide

Sealed Types and Records — Understanding sealed classes and record types used in this assignment.

Provided Types

The following types are **provided** — you do NOT implement these, but you will use them:

`Item` Interface (PROVIDED)

The sealed `Item` interface defines the contract for all items.

```
String getName()
```

Returns the name for this Item.

```
BigDecimal getPrice()
```

Returns the unit price for this Item.

```
BigDecimal calculateTotal(int quantity, boolean
useMembershipPricing)
```

Calculates the total price for the given quantity. The `useMembershipPricing` parameter allows items to apply membership discounts. Throws `IllegalArgumentException` if quantity is negative.

```
String getFormattedDescription()
```

Returns a formatted description suitable for display (e.g., `"Pen, $2.00"`).

`ItemOrder` Record (PROVIDED)

The `ItemOrder` record represents a purchase order for an item with a specific quantity.

```
ItemOrder(Item item, int quantity)
```

Constructor. Throws `NullPointerException` if item is null. Throws `IllegalArgumentException` if quantity is negative.

```
Item item()
```

Returns the Item in this order.

```
int quantity()
```

Returns the quantity.

Records automatically provide `equals()`, `hashCode()`, and `toString()`.

`Cart` Interface (PROVIDED)

The `Cart` interface defines the contract for shopping carts.

```
void add(ItemOrder order)
```

Adds an order to the cart. Replaces any previous order for the same item. If quantity is 0, removes the item from the cart. Throws `NullPointerException` if order is null.

```
void setMembership(boolean membership)
```

Sets whether the customer has a store membership.

```
BigDecimal calculateTotal()
```

Returns the total cost of all items. Uses `HALF_EVEN` rounding with scale of 2.

```
void clear()
```

Removes all orders from the cart.

```
CartSize getCartSize()
```

Returns a `CartSize` record with `itemOrderCount` (unique items) and `itemCount` (total quantity).

`Cart.CartSize` Record (PROVIDED)

Nested record in Cart interface.

```
record CartSize(int itemOrderCount, int itemCount)
```

- `itemOrderCount` — the number of unique ItemOrder objects in the cart
- `itemCount` — the total quantity of all items in the cart

```
int itemOrderCount()
```

Returns the number of unique ItemOrder objects.

```
int itemCount()
```

Returns the total quantity of all items.

✏️ Requirements

Your task is to implement four classes that make the shopping cart work:

- `AbstractItem` — complete the provided stub
- `StoreItem extends AbstractItem` — simple pricing (no bulk discounts)
- `StoreBulkItem extends AbstractItem` — bulk pricing for members
- `StoreCart implements Cart` — shopping cart implementation

Implementation Notes

Method Signatures: You must not change any method signatures or return types defined in this assignment. You may change parameter names but not parameter data types.

Access Modifiers: You must not introduce any other non-private methods to these classes, although you may add your own private helper methods.

BigDecimal Calculations: When calculating totals, do NOT convert BigDecimal values to double or float types. Use the provided methods of the BigDecimal class to perform mathematical calculations.

Requirement 1: AbstractItem

Guide

Defensive Programming — Constructor validation patterns and fail-fast principles.

Complete the `AbstractItem` abstract class stub provided in the starter code. This class provides common functionality shared by all item types.

`AbstractItem` Class API

The sealed `AbstractItem` class provides common functionality for all items. You must complete the implementation based on the stub provided.

```
protected AbstractItem(String name, BigDecimal price)
```

Constructor. Validates and stores the name and price. Throws `NullPointerException` if name or price is null. Throws `IllegalArgumentException` if name is empty or price is negative.

```
public String getName()
```

Returns the name for this Item.

```
public BigDecimal getPrice()
```

Returns the unit price for this Item.

```
protected static final NumberFormat CURRENCY_FORMAT
```

A shared NumberFormat for formatting currency (US locale). Initialize this field so subclasses can use it.

Using CURRENCY_FORMAT

Use the `CURRENCY_FORMAT` field in subclass `getFormattedDescription()` implementations:

```
CURRENCY_FORMAT.format(getPrice())  // Returns "$2.00"
```

Requirement 2: StoreItem

> ⚠ **Similar to A1a**
>
> This class is similar to, but not the same as, the class you created in Assignment 1a. Pay careful attention to the differences in the API.

`StoreItem` represents a simple item with standard pricing. Membership status has **no effect** on `StoreItem` pricing — the price is always `quantity × unit price`.

`StoreItem` Class API

The `StoreItem` class represents a simple item with standard pricing. It extends `AbstractItem` and ignores membership status when calculating totals.

```
public StoreItem(String name, BigDecimal price)
```

Constructor. Calls the `AbstractItem` constructor. Throws `NullPointerException` if name or price is null. Throws `IllegalArgumentException` if name is empty or price is negative.

```
public BigDecimal calculateTotal(int quantity, boolean
useMembershipPricing)
```

Returns `price × quantity`. The membership parameter is **ignored** for StoreItem. Throws `IllegalArgumentException` if quantity is negative.

```
public String getFormattedDescription()
```

Returns `"name, $price"` (e.g., `"Computer Science Pen, $2.00"`). Use the `CURRENCY_FORMAT` from `AbstractItem`.

```
public String toString()
```

Returns a debug representation. Recommended format: `StoreItem[name='...', price=...]`

```
public boolean equals(Object obj)
```

Two StoreItems are equal if they have the same name and price. A `StoreItem` is **never** equal to a `StoreBulkItem`.

```
public int hashCode()
```

Consistent with equals().

getFormattedDescription() Example

```
Computer Science Pen, $2.00
```

Required Overrides: You must override `toString`, `equals`, and `hashCode` in this class.

Guide

BigDecimal and BigInteger — Working with precise monetary calculations.

Requirement 3: StoreBulkItem

`StoreBulkItem` represents an item with bulk pricing available for members. When membership pricing is enabled, the bulk price is applied to as many complete bulk sets as possible, with any remainder charged at the standard unit price.

Example: An item costs $1.00 each or 10 for $5.00 (bulk). Buying 23 items with membership: (2 × $5.00) + (3 × $1.00) = $13.00.

`StoreBulkItem` Class API

The `StoreBulkItem` class represents an item with bulk pricing available for members. It extends `AbstractItem` and applies bulk discounts when membership pricing is enabled.

```
public StoreBulkItem(String name, BigDecimal price, int
bulkQuantity, BigDecimal bulkPrice)
```

Constructor. Calls the `AbstractItem` constructor for name and price validation. Throws `NullPointerException` if name, price, or bulkPrice is null. Throws `IllegalArgumentException` if name is empty, price is negative, bulkQuantity is negative, or bulkPrice is negative.

```
public BigDecimal calculateTotal(int quantity, boolean
useMembershipPricing)
```

Calculates the total price for the given quantity. If `useMembershipPricing` is false or `bulkQuantity` is 0, uses standard pricing. Otherwise, applies bulk pricing to as many complete bulk sets as possible, with remainder at unit price. Throws `IllegalArgumentException` if quantity is negative.

```
public String getFormattedDescription()
```

Returns `"name, $price (bulkQty for $bulkPrice)"`. If bulkQuantity is 0, omits the bulk pricing portion. Use the `CURRENCY_FORMAT` from `AbstractItem`.

```
public String toString()
```

Returns a debug representation. Recommended format: `StoreBulkItem[name='...', price=..., bulkQuantity=..., bulkPrice=...]`

```
public boolean equals(Object obj)
```

Two StoreBulkItems are equal if they have the same name, price, bulkQuantity, and bulkPrice. A `StoreBulkItem` is **never** equal to a `StoreItem`.

```
public int hashCode()
```

Consistent with equals().

getFormattedDescription() Examples

With bulk pricing:

```
'Java Rules!' button, $0.95 (10 for $5.00)
```

With bulkQuantity of 0 (no bulk info shown):

```
UW Note pad, $4.41
```

Required Overrides: You must override `toString`, `equals`, and `hashCode` in this class.

Requirement 4: StoreCart

`StoreCart` implements the shopping cart functionality. It stores `ItemOrder` objects and calculates the total cost based on membership status.

`StoreCart` Class API

The `StoreCart` class implements the shopping cart functionality. It manages a collection of `ItemOrder` objects and calculates totals by delegating to each item.

```
public StoreCart()
```

Constructor that creates an empty shopping cart. The customer does not have a membership by default.

```
public void add(ItemOrder order)
```

Adds an order to the shopping cart, replacing any previous order for the same item with the new order. **If the order quantity is 0, removes the item from the cart entirely.** Throws `NullPointerException` if order is null.

```
public void setMembership(boolean membership)
```

Sets whether the customer for this shopping cart has a store membership (true means the customer has a membership, false means they don't).

```
public BigDecimal calculateTotal()
```

Returns the total cost of this shopping cart as a BigDecimal. For each `ItemOrder`, delegate to the `Item.calculateTotal()` method passing the quantity and membership status. The returned BigDecimal should have scale of 2 and use the `HALF_EVEN` rounding rule.

```
public void clear()
```

Removes all orders from the cart.

```
public CartSize getCartSize()
```

Returns a `Cart.CartSize` record that contains:

- `itemOrderCount` — the number of unique ItemOrder objects in the cart
- `itemCount` — the total quantity of all items in the cart

```
public String toString()
```

Returns a String representation of this Cart. You may use any reasonable format.

Implementation Hint: Consider which Java Collection would best support adding, replacing, and removing orders efficiently.

Zero Quantity: Adding an order with quantity 0 removes that item from the cart entirely.

Information Expert Pattern

Notice how `calculateTotal()` in `StoreCart` delegates to `Item.calculateTotal()`. This is the Information Expert pattern — the Item knows how to calculate its own price, so the cart just asks each item for its total and sums the results.

Requirement 5: Code Quality

Before submitting, ensure your code passes all linting checks.

1. Run Checkstyle and fix any violations
2. Run IntelliJ inspections and resolve all warnings
3. Ensure no compiler warnings remain

No Console Output: No console output should appear when running the BookstoreMain program or when running your unit tests.

Your code should be **free of all warnings** before submission.

> **Guide**
>
> Linters and Code Quality

Requirement 6: Unit Testing

Move all tests created in Assignment 1b into the tests folder of your Assignment 1c project. Run the tests for coverage. Fix any mistakes found in the tests or classes under test. If your tests fail to reach 100% coverage of the class under test, add tests to gain full coverage.

Testing: Use the unit tests you wrote in Assignment 1b to verify your implementations.

Running Tests with Coverage: In IntelliJ, right-click your test class or `tests` folder and select **Run with Coverage**. The coverage report shows which lines of your code were executed by your tests. Aim for 100% coverage of your model classes.

Requirement 7: Executive Summary

Included in the project is a file called `executive-summary.md`. This file documents the repository and project. Cloud-based repository services typically display the contents on the landing page for the repository.

External Resources

- About READMEs
- Markdown Cheat Sheet

Edit the file to personalize it for your submission. Carefully read all the text found inside of the enclosing square braces `[]`. Remove all this text (and square braces), replacing it with your own specific details about the project.

📚 Guide Reference

Guide	Description
Git Branching and Pull Requests	Working on branches and merging via PR
Inheritance Hierarchies	Understanding the Item → AbstractItem pattern
Sealed Types and Records	Sealed classes and record types
Implementing equals, hashCode, and toString	Implementing Object methods correctly
BigDecimal and BigInteger	Precise monetary calculations
Defensive Programming	Constructor validation, fail-fast principles
Writing JUnit 5 Tests	Annotations, assertions, test structure

---

🚀 Submission and Grading

Git Workflow Required

For this assignment, you must work on a **development branch** and submit via **pull request**.

Submitting Your Work

1. **Ensure all your work is committed** to your `dev` branch
2. **Push your dev branch** to GitHub:

   In IntelliJ: Click **Git** → **Push**, then click **Push**

   Or from the terminal:

   ```
   git push -u origin dev
   ```

3. **Create a Pull Request** on GitHub:
   - Go to your repository on GitHub.com
   - Click **Pull requests** → **New pull request**
   - Set base: `main` and compare: `dev`
   - Add a descriptive title and summary of your changes
   - Click **Create pull request**
4. **Merge the pull request** to main
5. **Verify:** Visit your repository on GitHub.com and confirm your changes appear on the `main` branch

Note: You can commit and push multiple times. Only your final state of `main` at the deadline will be graded.

Please see the rubric in Canvas for a breakdown of the grade for this assignment.