a1b # assignment

Assignment 1b – Test-Driven Development

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

Due Date

Sunday, 18 January 2026, 23:59:59

📄 Description

This assignment introduces **Test-Driven Development (TDD)** — the practice of writing tests *before* writing implementation code. You will write a comprehensive JUnit 5 test suite for the UW Bookstore model classes, testing against a provided library.

The model classes you'll test include items (simple and bulk-priced), item orders, and shopping carts. Your tests will verify that the implementation follows the API specification and handles edge cases correctly.

Tip

Writing tests first forces you to think carefully about the expected behavior before coding. This leads to better-designed, more reliable code.

Guide

Test-Driven Development — Understand the TDD philosophy before starting.

🎯 Learning Objectives

By completing this assignment, you will:

- Practice Test-Driven Development (TDD) methodology
- Write comprehensive JUnit 5 test suites
- Test code from API specifications without seeing the implementation
- Identify and document bugs through systematic testing
- Work with Java records and sealed class hierarchies
- Understand the importance of edge case testing

✅ Before You Begin

> **Guide**
>
> Writing JUnit 5 Tests — Annotations, assertions, and test structure.

Ensure you have completed the following:

- Familiar with JUnit 5 testing (review lecture materials and guides above)
- Read and understood the API specifications and requirements in this document
- Reviewed the provided `ItemOrderTest.java` as a template

⚙️ Environment Setup

> **Note**
>
> No additional setup required. Continue using your environment setup in Assignment 1a.

The starter project includes a library (`lib/tcss305-a1b-model.jar`) containing the model classes you will test. This library is pre-configured in the project.

📦 Project Setup

GitHub Classroom Assignment

Accept Assignment 1b

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:
   - `lib/tcss305-a1b-model.jar` — The model library to test against
   - `test/edu/uw/tcss/model/ItemOrderTest.java` — Example tests (provided)

Troubleshooting IntelliJ + GitHub Authentication

Last time we're including these instructions!

Starting with Assignment 1c, you are expected to have your Git/GitHub workflow established. If you need a refresher, refer back to Assignment 1a or the Git Version Control guide.

If IntelliJ prompts for credentials or shows authentication errors, try one of these solutions:

Option A: Personal Access Token (Recommended)

Option B: GitHub Desktop

✏️ Finishing the Assignment

Part I: Unit Testing

You will write unit tests for the model classes using Test-Driven Development. Write your tests based on the API specifications below — do NOT attempt to decompile or reverse-engineer

the provided library. (Spoiler: you won't find any implementation code to copy.)

All tests must be written as JUnit 5 tests and placed in `test/edu/uw/tcss/model/`. Test ALL public methods explicitly, even if they're used indirectly in other tests. Include tests for both valid inputs and error conditions.

> **Tip**
>
> Study the provided `ItemOrderTest.java` as an example, along with `StoreItemTest.java` from Assignment 1a. Use these as templates for structure, naming conventions, and testing patterns.

Model Architecture & Shared Types

> **API Documentation**
>
> View the complete Javadoc API

Before diving into the requirements, familiarize yourself with the model structure.

Class Hierarchy

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

```
Item (sealed interface)
└── AbstractItem (sealed abstract class)
    ├── StoreItem (final) – Simple pricing
    └── StoreBulkItem (final) – Bulk pricing for members

--- Not part of sealed hierarchy ---

ItemOrder (record) – Represents an item and quantity
Cart (interface) – Shopping cart operations
└── StoreCart (final) – Implementation of Cart
```

Testing Abstract Classes

You'll notice `AbstractItem` is an abstract class in the hierarchy. **You do not create a separate test class for abstract classes.** Instead, test abstract class behavior through the concrete subclasses (`StoreItem` and `StoreBulkItem`). The inherited methods like `getName()` and `getPrice()` are tested as part of your concrete class tests.

Note

Test `getName()` and `getPrice()` in BOTH `StoreItemTest` and `StoreBulkItemTest`. Technically, testing them once would suffice, but this assignment is about practicing test writing — so practice!

Item Interface

All items implement the `Item` interface. You'll use these methods when testing concrete item classes:

`Item` Interface Methods

These methods are defined in the `Item` interface and implemented by all item classes.

```
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. Throws `IllegalArgumentException` if quantity is negative.

```
String getFormattedDescription()
```

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

ItemOrder Record

`ItemOrder` is a **record** representing a purchase order for an item. You'll create `ItemOrder` objects when testing `StoreCart`. A complete test class (`ItemOrderTest.java`) is provided as an example.

Java Records

Java Records — Oracle documentation on record classes.

`ItemOrder` Record

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.

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

Auto-generated by record. Two ItemOrders are equal if they have the same item and quantity.

```
int hashCode()
```

Auto-generated by record. Consistent with equals().

```
String toString()
```

Auto-generated by record. Format: `ItemOrder[item=..., quantity=...]`.

Requirement 1: StoreItemTest.java

Create a test class for `StoreItem`, which represents a simple item with standard pricing. Membership pricing has no effect on `StoreItem`.

StoreItem API

`StoreItem` Class

The `StoreItem` class represents a simple item with standard pricing.

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

Constructor. Throws `NullPointerException` if name or price is null. Throws `IllegalArgumentException` if name is empty or price is negative.

```
String getName()
```

Returns the name.

```
BigDecimal getPrice()
```

Returns the unit price.

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

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

```
String getFormattedDescription()
```

Returns `"name, $price"` (e.g., `"Computer Science Pen, $2.00"`).

```
String toString()
```

Returns debug representation (format not tested).

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

Two StoreItems are equal if they have the same name and price.

```
int hashCode()
```

Consistent with equals().

What to Test

Constructor tests:

- Test constructor with valid name and price
- Test constructor with null name (expect `NullPointerException`)
- Test constructor with null price (expect `NullPointerException`)
- Test constructor with empty name (expect `IllegalArgumentException`)
- Test constructor with negative price (expect `IllegalArgumentException`)

Getter tests:

- Test `getName()` returns correct value
- Test `getPrice()` returns correct value

calculateTotal() tests:

- Test `calculateTotal()` with various quantities (0, 1, many)
- Test `calculateTotal()` with negative quantity (expect `IllegalArgumentException`)
- Test `calculateTotal()` returns same result regardless of membership parameter

Other tests:

- Test `getFormattedDescription()` returns correct format
- Test `equals()` with equal and non-equal items
- Test `hashCode()` consistency with equals

Warning

This list provides a starting point, not a comprehensive test plan. You are expected to think critically about edge cases and add additional tests beyond what is listed here.

Guide

Writing JUnit 5 Tests — Annotations, assertions, and test structure.

Guide

The equals/hashCode Contract — Understanding equality testing.

Requirement 2: StoreBulkItemTest.java

Create a test class for `StoreBulkItem`, which represents an item with bulk pricing available for members.

StoreBulkItem API

StoreBulkItem Class

The `StoreBulkItem` class represents an item with bulk pricing available for members.

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

Constructor. 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.

```
String getName()
```

Returns the name.

```
BigDecimal getPrice()
```

Returns the unit price.

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

Calculates total. If `useMembershipPricing` is false OR bulkQuantity is 0, returns `price × quantity`. If membership pricing is enabled, applies bulk price to complete sets and unit price to remainder. Example: 23 items at $1.00 each, 10 for $5.00 bulk = (2 × $5.00) + (3 × $1.00) = $13.00. Throws `IllegalArgumentException` if quantity is negative.

```
String getFormattedDescription()
```

Returns `"name, $price (bulkQty for $bulkPrice)"` (e.g., `"UW Note pad, $4.41 (6 for $10.04)"`). If bulkQuantity is 0, omits the bulk pricing portion.

```
String toString()
```

Returns debug representation (format not tested).

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

Two StoreBulkItems are equal if they have the same name, price, bulkQuantity, and bulkPrice. Note: A StoreItem and StoreBulkItem are NEVER equal, even with identical name and price.

```
int hashCode()
```

Consistent with equals().

What to Test

Constructor tests:

- Test constructor with valid arguments
- Test constructor with null name, price, or bulkPrice (expect `NullPointerException`)
- Test constructor with empty name (expect `IllegalArgumentException`)
- Test constructor with negative price, bulkQuantity, or bulkPrice (expect `IllegalArgumentException`)

calculateTotal() tests:

- Test WITHOUT membership — should use unit price regardless of bulk settings
- Test WITH membership and quantity < bulkQuantity — unit price only
- Test WITH membership and quantity = bulkQuantity — exactly one bulk set
- Test WITH membership and quantity > bulkQuantity — bulk sets + remainder
- Test WITH membership and bulkQuantity of 0 — should use unit price
- Test with negative quantity (expect `IllegalArgumentException`)

Other tests:

- Test `getFormattedDescription()` includes bulk pricing info
- Test `getFormattedDescription()` when bulkQuantity is 0 (no bulk info shown)
- Test `equals()` with equal and non-equal items
- Test that StoreItem and StoreBulkItem are NEVER equal

> **Warning**
>
> This list provides a starting point, not a comprehensive test plan. You are expected to think critically about edge cases and add additional tests beyond what is listed here.

Requirement 3: StoreCartTest.java

Create a test class for `StoreCart`, which implements the shopping cart functionality. This is the most complex class to test.

StoreCart API

`StoreCart` implements the `Cart` interface:

`StoreCart` Class

The `StoreCart` class implements the `Cart` interface for shopping cart functionality.

```
StoreCart()
```

Constructor. Creates an empty cart with no membership.

```
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 hasMembership)
```

Sets the customer's membership status. Members receive bulk pricing on eligible items.

```
BigDecimal calculateTotal()
```

Returns the total cost of all items. Uses `HALF_EVEN` (banker's) 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).

```
String toString()
```

Returns debug representation (format not tested).

Warning

Do NOT write tests for the `Cart.CartSize` record. Records auto-generate their methods and don't require testing.

What to Test

add() method:

- Add one ItemOrder, verify cart size
- Add multiple different ItemOrders
- Add ItemOrder with same item — should replace, not accumulate
- Add ItemOrder with quantity 0 — should remove item from cart
- Add null (expect `NullPointerException`)

setMembership() method:

- Verify membership affects bulk pricing in calculateTotal()
- Test toggling membership on and off

calculateTotal() method:

- Empty cart should return $0.00
- Single StoreItem, verify simple pricing
- Multiple StoreItems
- StoreBulkItem without membership — unit price
- StoreBulkItem with membership — bulk pricing applies
- Mixed StoreItem and StoreBulkItem
- Verify result has scale of 2 and uses HALF_EVEN rounding

clear() method:

- Verify clear() empties the cart
- Verify cart size is 0 after clear

getCartSize() method:

- Empty cart returns (0, 0)
- Verify itemOrderCount = number of unique items
- Verify itemCount = total quantity of all items

> ⚠ **Warning**
>
> This list provides a starting point, not a comprehensive test plan. You are expected to think critically about edge cases and add additional tests beyond what is listed here.

> **Tip**
>
> Test `calculateTotal()` last. It depends on many other components working correctly. Build confidence in `add()`, the Item classes, and `setMembership()` first. Additionally, complete `StoreItemTest` and `StoreBulkItemTest` before starting `StoreCartTest` — the cart relies on items working correctly!

Requirement 4: Bug Discovery

The provided library contains intentional bugs. Discovering them through testing demonstrates the value of systematic testing.

To earn bug discovery credit:

1. Write a test that exposes the bug (it will fail)
2. Add a comment in your test class using the template below
3. Document the bug in your README.md using the same template
4. Keep the failing test in your submission

Bug Report Template

Use this exact format when documenting bugs in your test class comments AND in your README:

```
## Bug Report: [Brief Title]

**Class:** [e.g., StoreCart]
**Method:** [e.g., calculateTotal()]

**Expected Behavior:**
[What the API specification says should happen]

**Actual Behavior:**
[What actually happens when you run your test]

**Test That Demonstrates Bug:**
[Name of your test method, e.g., testCalculateTotalRounding()]

**Additional Notes:** (optional)
[Any other observations]
```

Example:

```
## Bug Report: getName Returns Wrong Value

**Class:** StoreItem
```

```
**Method:** getName()

**Expected Behavior:**
According to the API, getName() should return the name passed to the
constructor.

**Actual Behavior:**
getName() returns the name in all uppercase letters instead of preserving the
original case.

**Test That Demonstrates Bug:**
testGetNamePreservesCase()

**Additional Notes:**
Passing "Computer Science Pen" to constructor, but getName() returns "COMPUTER
SCIENCE PEN".
```

Requirement 5: Executive Summary

Your project includes a file called `executive-summary.md`. This file documents your project and submission details.

The `.md` file extension stands for Markdown. Markdown is a markup language used to format plain text. You may already know a markup language — the M in HTML and XML stands for markup. (Note: markup languages are NOT programming languages but instead are tools for formatting text.)

> **External Resource**
>
> Markdown Cheat Sheet

Edit the `executive-summary.md` 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.

Your executive summary should include:

- Your name and course information
- Brief description of the tests you wrote
- Bug reports for any bugs you discovered (use the Bug Report Template from Requirement 4)

📚 Guide Reference

Guide	Description
Test-Driven Development	TDD philosophy and methodology
Writing JUnit 5 Tests	Annotations, assertions, test structure
The equals/hashCode Contract	Testing equality contracts
Introduction to Unit Testing	Unit testing fundamentals
Linters and Code Quality	Keeping test code clean
Checkstyle Rules Reference	Fixing Checkstyle violations
IntelliJ Inspections Reference	Fixing IntelliJ warnings

🚀 Submission and Grading

Submit your work by committing and pushing to your GitHub repository.

Submitting Your Work

1. In IntelliJ, click **Git → Commit**
2. Select all changed files and write a descriptive commit message
3. Click **Commit and Push**
4. **Verify:** Visit your repository on GitHub.com and confirm your changes appear

🔧 Submitting with GitHub Desktop (Option B users only) ›

🔥 Tip

You can commit and push multiple times. Only your final push before the deadline will be graded.

Important

Your tests will be graded by running them against both the provided (buggy) library AND a correct implementation. Tests that incorrectly fail on the correct implementation will lose points.

The **only** tests that should fail when you submit are tests that expose bugs you have documented in your executive summary. Any other failing tests will result in lost points.