

CSC 240: Computer Science III
Fall 2026 – Section 01
Homework 3 – Warehouse (Chapter 11)

Name: _____

WCU ID: _____

Assigned: Thursday, October 22, 2026

Due: Tuesday, Nov 10, 2026, 11:59 PM

This homework contains 8 pages and 5 questions, worth a total of 100 points. It counts for **12%** of your course grade (twice HW1 and HW2), and you have three weeks. You will build a small warehouse system that uses every Chapter 11 tool from Classes 12–16: a family of **custom exception classes**, **throw** with validation, **checked vs. unchecked**, multiple **catch** blocks in the right order, **wrapping an exception with a cause**, **try-with-resources**, a loader that *reports instead of crashes*, and – in Q5 – **object serialization** and **random-access files** (Chapter 11.3, taught Oct 27/29; Q1–Q4 need only what you already know).

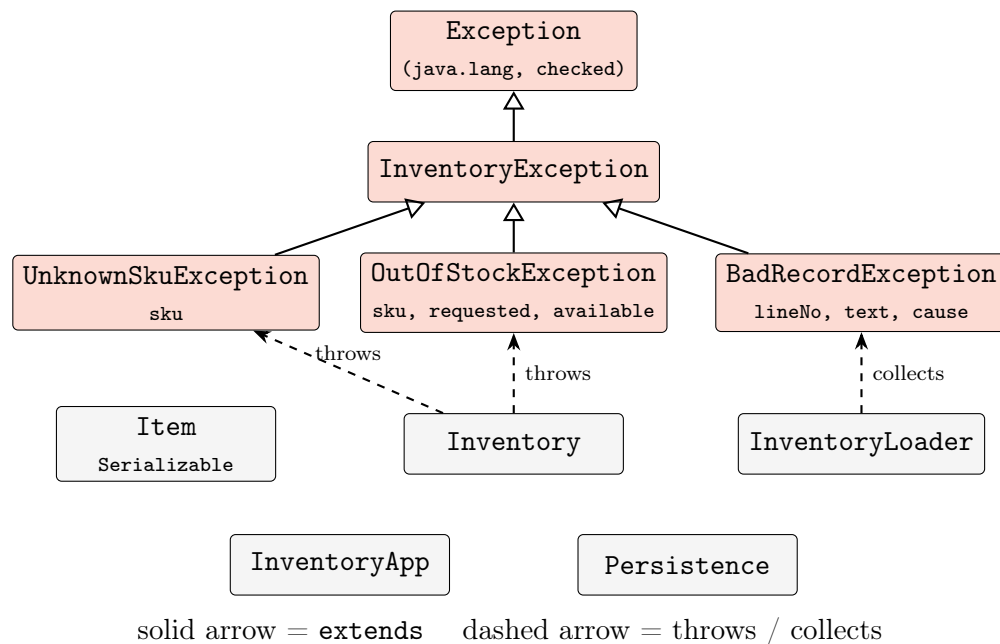
(**DO NOT** copy code from the book, slides, **StackOverflow**, **YouTube**, **GitHub**, **ChatGPT**, etc. See Section 6 of the syllabus.)

Instructions – read before you start

- **Individual work.** You may discuss ideas, but you may not share, exchange, or jointly write code. You must be able to explain and modify every line you submit.
- **Starter code.** Download `CSC240_HW3_starter.zip` from the course website (wcu.ninja/csc240) or D2L. It contains nine `.java` files plus `inventory.csv` and `orders.txt`. The starter compiles as-is. **Do not modify the two data files**, keep every given method signature *except where a TODO tells you to change a **throws** clause*, and add your code where the `TODO` comments are. You may add private helper methods.
- **Submission.** Put your name in the comment header of each file. Submit the **nine** `.java` files in one zip named `LastName_FirstName_HW3.zip` to the D2L dropbox before the deadline. Do not include `.class`, `.dat`, or `.txt` files. The D2L timestamp is authoritative.
- **Code must compile** with the JDK version listed in D2L. A file that does not compile receives at most half of that question's points. Partially working programs receive partial credit when you clearly comment what works and what does not.
- **Exception messages must match exactly.** Every message is spelled out in the `TODO` comments and in this handout. We test your classes directly (not just `main`), so a method that *prints* instead of *throws*, or throws the wrong type, earns no credit for that part.
- **No empty catch blocks, no catch** (`Exception e`), **no** `System.exit`. Each of these costs -3 wherever it appears.
- **Style** (up to -5 points overall): meaningful names, consistent indentation, `@Override` where you override, and a short comment for each part.
- **Late work:** no credit unless a valid excuse is communicated before the deadline (syllabus, Section 7).

Question:	1	2	3	4	5	Total
Points:	15	20	20	25	20	100
Score:						

The design



1. A family of exceptions (15 points) – custom exception classes, `super(message, cause)`

All four classes are **checked** (they descend from `Exception`), because every one of them describes something the caller can reasonably handle: an unknown product, an empty shelf, a typo in a data file.

- (3 points) `InventoryException` is the base of the family. Give it two constructors, `(String message)` and `(String message, Throwable cause)`, each passing its arguments to `super`. (Without them, the three subclasses below cannot compile – why?)
- (4 points) `UnknownSkuException`: a private `sku`, a constructor `(String sku)` whose message is `sku + ": unknown SKU"`, and `getSku()`.
- (4 points) `OutOfStockException`: fields `sku`, `requested`, `available`; a constructor `(String sku, int requested, int available)` and the three getters. Its message is

```
sku + ": out of stock (requested " + requested + ", available " +
    available + ")"
```

- (4 points) `BadRecordException`: fields `lineNo` and `text`; a constructor `(int lineNo, String text, Throwable cause)` that passes `cause` to `super`; getters `getLineNo()` and `getText()`. Its message is

```
"line " + lineNo + ": \" + text + "\""
```

The original exception must survive: `e.getCause()` on a `BadRecordException` returns it.

2. Item and Inventory (20 points) – throw, validation, checked vs. unchecked

- (a) (5 points) **Item validation.** The constructor must refuse bad values *before storing anything* by throwing `IllegalArgumentException` with exactly these messages, checked in this order:

```
"sku must not be blank"           // null or only
    whitespace
"name must not be blank"
"quantity must not be negative, got " + quantity
"unitPrice must not be negative, got " + unitPrice
```

Store `sku.trim()` and `name.trim()`. `setQuantity` does the same quantity check. These are **unchecked**: a negative quantity is the *caller's* bug, so no `throws` clause and no `try` inside `Item`.

- (b) (4 points) `Item.fromCsv(String line)` builds an `Item` from a line `sku,name,quantity,unitPrice`; use `split(",", -1)` and trim the numbers. Do **not** catch anything here. A short line throws `ArrayIndexOutOfBoundsException`, "four" throws `NumberFormatException`, a negative quantity throws the constructor's `IllegalArgumentException`. What to do about a bad line is the loader's decision (Q3), not this method's.
- (c) (3 points) `Inventory.add(Item)` adds the item, unless an item with the same SKU is already present: then throw `IllegalArgumentException("duplicate SKU " + sku)`.
- (d) (3 points) `Inventory.find(String sku)` returns the item, or throws `UnknownSkuException`. The starter already declares it in the header: checked, because "we don't carry that" is something every caller must plan for.
- (e) (5 points) `Inventory.remove(String sku, int quantity)` ships quantity units and returns the item. Three things can go wrong, in this order:

```
quantity <= 0    -> throw new IllegalArgumentException(
                    "quantity must be positive, got " + quantity);
SKU not present -> UnknownSkuException: call find(), let it propagate
                    (do NOT catch and re-throw)
too few units   -> throw new OutOfStockException(sku, quantity, available)
                    ;
```

Otherwise lower the quantity with `setQuantity`. *Self-check: after `remove("A100", 10)` on the given file, A100 has 30 left.*

3. InventoryLoader (20 points) – a loader that returns, collects, or throws

`InventoryLoader.load(fileName)` is the Class 14 capstone, for real. The rule: the loader **never prints and never exits** – it does not know whether a console app, a GUI, or a test is calling it. It *returns* the good items, *collects* the bad lines, and *throws* one exception when the whole file is unusable.

- (a) (8 points) **One bad line never stops the load.** Wrap the `inv.add(Item.fromCsv(line))` call so that a line that throws is added to rejected as `new BadRecordException(lineNo, line, e)` and the loop continues. Catch exactly the types Q2 can throw (two names in one multi-catch is enough – remember that `NumberFormatException` *is an* `IllegalArgumentException`). A duplicate SKU (Q2c) is also a bad record.
- (b) (4 points) **try-with-resources.** Replace the `BufferedReader` declaration and `br.close()` with `try (BufferedReader br = ...) { ... }`. No `finally`, no `close()` anywhere in this file.
- (c) (8 points) **The whole file is unusable.** If the file cannot be opened or read (`IOException`), throw `new InventoryException("cannot open " + fileName, e)` – keep `e` as the cause – and change the header to `throws InventoryException`. After this change, the word `IOException` must not appear in any `throws` clause of this class: the caller sees only our family of exceptions. (This means `InventoryApp` will stop compiling until you do Q4a. That is the compiler enforcing the contract.)

4. **InventoryApp (25 points)** – catch where you can do something

`inventory.csv` has a header row and twelve data lines, four of them bad on purpose. `orders.txt` has one order per line, `<sku> <quantity>`, some of them impossible. Run from the folder that contains both files; also try `java InventoryApp nosuch.csv`.

- (a) (7 points) **Load and report.** Catch `InventoryException` around `loader.load(csv)`, print `Could not load <csv>: <message> (cause: <cause's simple class name>)` and return. Remove `throws IOException` from `main` – nothing in `main` may propagate. Then print one line per rejected record, in file order: `Rejected <message> (<cause simple name>: <cause message>)` followed by the two summary lines already in the starter. The cause's name is `e.getCause().getClass().getSimpleName()`.
- (b) (12 points) **Ship the orders.** Read `orders.txt` with `try-with-resources`. For each non-blank line, inside *one* `try`: split on whitespace, `parseInt` the quantity, call `inv.remove(sku, qty)`, print `Shipped <qty> x <name> (<left> left)`, and remember `String.format("%s,%d,%s,%.2f", sku, qty, name, qty * unitPrice)` for part (c). Then the catches:
- `OutOfStockException` and `UnknownSkuException` (two separate catches, each with its own counter): print `Cannot ship <e.getMessage()>`.
 - `IllegalArgumentException` | `ArrayIndexOutOfBoundsException`: print `Bad order line <lineNo>: "<line>" (<e.getMessage()>)`. This one block covers "twenty" (a `NumberFormatException`), a quantity of 0, and a line with no quantity. Try writing `NumberFormatException | IllegalArgumentException` instead and read what javac says.

Finish with the **Orders**: ... summary line from the starter. If `orders.txt` itself cannot be read, print `Could not read orders.txt: <message>` and still print the summary. *Think*: why must the two `InventoryException` subclasses be caught before a catch of `InventoryException` could be, and what would happen to your counters if you caught only the base class?

- (c) (6 points) **Write shipped.txt**. Using a `PrintWriter` in try-with-resources, write the remembered lines (one per shipped order) and print `Shipped orders written to shipped.txt`. `PrintWriter(String)` throws `FileNotFoundException`: catch it and print `Could not write shipped.txt: <message>`.

5. **Persistence (20 points)** – object serialization and random access (Ch. 11.3, Oct 27/29)

- (a) (7 points) **Serialization**. Make `Item` implements `Serializable` (add `private static final long serialVersionUID = 1L;`). `Persistence.save` writes the *whole* `ArrayList<Item>` with one `writeObject` call; `Persistence.load` reads it back with `readObject` and casts. `readObject` also declares `ClassNotFoundException` (checked): catch it and throw `new IOException("file does not contain an item list: " + fileName, e)`, so callers deal with one type. Both methods use try-with-resources.
- (b) (5 points) **Fixed-length records**. `writeQuantities` opens a `RandomAccessFile` in mode `"rw"`, calls `setLength(0)`, and writes each item's quantity with `writeInt` in list order, so record i occupies bytes $4i \dots 4i + 3$. `countQuantities` returns `length() / 4`.
- (c) (5 points) **Seek, don't scan**. `readQuantity(fileName, index)` does `seek(4L * index)` then `readInt()`. `updateQuantity(fileName, index, quantity)` seeks and `writeInts` *in place*, leaving every other record untouched. Both throw `IllegalArgumentException("no record " + index)` when `index` is out of range (check it against `length() / 4` *before* seeking). Reading record 2 must not read records 0 and 1.
- (d) (3 points) **Wire it into main** exactly as the TODO Q5(d) comment describes: one try with a catch for `IllegalArgumentException` (Random access error: `<message>`) and one for `IOException` (File error: `<message>`). The last call, `readQuantity("quantities.dat", 99)`, is *supposed* to end in the first catch. Note the last "normal" line: the file now says 20 but the `Inventory` object still says 6 – a random-access file is storage, not a cache of your objects.

Expected output (complete)

java InventoryApp must print this exactly, including spacing and blank lines.

```
=== Loading inventory.csv ===
Rejected line 5: "A103,27-inch monitor,four,219.50" (NumberFormatException: For input string: "four")
Rejected line 8: "A106,HDMI cable,25" (ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3)
Rejected line 9: "A107,Desk lamp,-3,18.75" (IllegalArgumentException: quantity must not be negative, got -3)
Rejected line 11: "A109,,10,5.00" (IllegalArgumentException: name must not be blank)
Loaded 8 items, rejected 4 lines.
Inventory value: $3509.98

=== Processing orders.txt ===
Shipped 10 x USB-C cable (30 left)
Shipped 2 x Mechanical keyboard (6 left)
Cannot ship A105: out of stock (requested 1, available 0)
Cannot ship A103: unknown SKU
Bad order line 5: "A101 twenty" (For input string: "twenty")
Shipped 5 x Portable SSD 1TB (0 left)
Cannot ship A111: out of stock (requested 1, available 0)
Bad order line 8: "A104 0" (quantity must be positive, got 0)
Shipped 2 x Noise-cancelling headphones (4 left)
Orders: 4 shipped, 2 out of stock, 1 unknown SKU, 2 bad lines
Shipped orders written to shipped.txt

=== Saving and reloading ===
Saved 8 items to inventory.dat
Reloaded 8 items; first: A100 USB-C cable          30 @ $   9.99
quantities.dat: 8 records, record 2 = 6
After update: record 2 = 20, inventory still says 6
Random access error: no record 99
```

shipped.txt must then contain exactly these four lines:

```
A100,10,USB-C cable,99.90
A102,2,Mechanical keyboard,178.00
A111,5,Portable SSD 1TB,499.50
A108,2,Noise-cancelling headphones,298.00
```

and java InventoryApp nosuch.csv must print:

```
=== Loading nosuch.csv ===
Could not load nosuch.csv: cannot open nosuch.csv (cause: FileNotFoundException)
```

We will also test your classes with **other** CSV files, order files, and direct calls (new Item("", ...), remove on an empty inventory, readQuantity with index -1, a BadRecordException's getCause(), and so on), so compute everything in code and don't print these lines directly.

Before you submit – checklist

- ☐ All nine `.java` files compile with no errors, and each has my name in its header comment.
- ☐ All four exception classes extend `InventoryException` / `Exception`; `BadRecordException` keeps its cause.
- ☐ `Item` and `Inventory` **throw**; they never print and never catch.
- ☐ `InventoryLoader` has no `System.out`, no `close()`, and no `IOException` in a **throws** clause.
- ☐ `main` has no **throws** clause; there is no `catch (Exception e)`, no empty catch, no `System.exit`.
- ☐ Every file is opened with try-with-resources.
- ☐ My output matches the expected output exactly, `shipped.txt` has four lines, and `nosuch.csv` is handled.
- ☐ Zip named `LastName_FirstName_HW3.zip` (nine `.java` files only) uploaded to D2L before the deadline.