Problem Set
CODING EXAM ONE
VERSION 1.0

This is the first of three timed coding exams for CS250. The exam is worth **5 points total**, and you will complete *two problems*: one 2-point problem and one 3-point problem. The goal is to practice solving small, well-scoped programming tasks under time constraints. This is exactly the skill you'll need in coding interviews, where you're expected to read a prompt, validate inputs, design a clean approach, and implement it correctly without outside help. Just as importantly, these problems reinforce computational thinking: breaking problems into steps, reasoning about edge cases, and translating logic into reliable code.

How do I study for the exam?

Prepare the way you'd prepare for a performance: you don't just "understand" the ideas; you practice doing them quickly, correctly, and calmly without outside help. Start by solving these problems on your own *first*. Aim for at least one problem per weekday and 2-3 over the weekend.

The bigger goal is consistency: sit down, read the prompt carefully, and produce a complete solution from scratch. Don't copy-paste old code that you wrote for some other classes/prjects. The exam is designed to assess students who can reliably recreate common patterns (validation, loops, string scanning, maps, simple parsing) under time pressure.

When you work on a problem, begin by understanding it deeply *before you type*. Identify inputs, outputs, edge cases, and what "invalid input" looks like. Write a short plan: clean input → validate → compute → print. This is particularly important for problems in this set such as ciphers, day-of-year, prime-factorization, and evaluators where small mistakes in validation or ordering can sink an otherwise correct solution. Next, code in a way that is easy to debug: small helper methods, clear variable names, and early `return`s when something is invalid.

Studying with friends can help a lot, but only if you do it the right way. Keep it high level: collaboratively break down the problem, agree on validation rules, talk through algorithm choices, and compare approaches. Use the group time to practice explaining solutions out loud (as if teaching), do timed "mock exam" runs, and quiz each other on edge cases. Avoid the unhelpful pattern where one person dominates and "solves it all" while others watch. That doesn't build skill, it builds dependency. A good group session has everyone contributing: one person drives, one person reviews, one person plays "test case adversary," then rotate. If you do this for each problem, you are likely to benefit much, much more.

Finally, treat LLMs and search as study tools, not solving tools. If you must use them for this exam?Use them after you've made a serious attempt: to clarify a tricky concept, confirm a rule (like leap years or a cipher detail), or compare your solution to a clean reference. Because the exam will measure what you can do alone, so your practice should also look like the exam: you producing working code from scratch. Keep a brief log after each session: what went wrong, what edge case surprised you, and what pattern you'll reuse next time. That log becomes your personal "no-internet" reference in your head.

However, you cannot talk to the Professor or TAs about the solutions. You're welcome to ask about clarifications on the problem statement (inputs, output format, edge cases), but not "is this approach right?" or "what's the trick?" If you're stuck, use your notes, work through small examples, and lean on your own debugging before you lean on anyone else. Think of this as an "airplane mode" exam; your brain is the only signal you're allowed to use.

You are expected to follow standard Java naming conventions. This includes both class names and package names, as specified in the problem description.

Although some programs may crash when presented with invalid input, your solutions to the problems you attempt in this exam must not. Your program should be capable of detecting malformed input and handling it gracefully. For example, if the program asks the user for a positive integer in base-10, then any input that does not meet that expectation must be caught and addressed and do so without causing the program to fail.

You must include a README file. This file should crisply explain the purpose of your program, as well as how to compile and run it. Imagine you are writing instructions for someone unfamiliar with your work; the more detailed your README, the more useful it becomes.

Table of Contents

[2 points]
Problem 1: Reversing the digits
Write a program that repeatedly prompts the user to enter a number. The number must be a positive integer, expressed in the standard decimal numbering system. Each time the user provides an input, the program must evaluate whether it qualifies as valid. If the input fails to meet the criteria (whether due to alphabetic characters, punctuation marks, decimal points, negative signs, or any other non-conforming element) the program should reject it and prompt the user again.

Once the number is confirmed to be a positive, integer number you are required to perform the following operations:

1. Reverse the digits of the number so that the last digit becomes the first, the second-to-last becomes the second, and so on.
2. Examine each digit in the reversed number and determine whether it is a prime digit. Only the digits 2, 3, 5, and 7 qualify.
3. Compute the sum of all prime digits identified in the reversed number.
4. However, if no prime digits are found in the reversed number, the program must instead print the message: print "No prime digits found in the number:" [number]

The main class should in the cs250.exam package and named NumberReverser. You may have other classes if you'd like.
Usage: java cs250.exam. NumberReverser

Sample Input/Output
```
Enter a number: 2534
Reversed number: 4352
Prime digits in reversed number: 2, 5, 3
Sum of prime digits: 10

Enter a number: 2x789.21
The number you entered [2x789.21] is not a valid number
```

[2 points]
Problem 2: Palindrome Checker
Write a Java program that take reads a string from the command line and checks if it is a palindrome (e.g., a word, phrase, or sequence that reads the same backward as forward). For this problem, you are required to **ignore** spaces, punctuation, and case sensitivity. For example. madam and "Nurses run" are both examples of Palindromes.

The main class should in the cs250.exam package and named PalindromeChecker:

Usage: java cs250.exam.PalindromeChecker <text>

Sample Input/Outputs

Example-1:
```
java cs250.exam.PalindromeChecker earth

The provided text [earth] is not a Palindrome
```

Example-2:
```
java cs250.exam.PalindromeChecker A man, a plan, a canal, Panama

The provided text [A man, a plan, a canal, Panama] is a Palindrome
```

[2 points]
Problem 3: Finding Armstrong Numbers
Write a program that produces the set of Armstrong Numbers within a specified range. An Armstrong Number is a number that is equal to the sum of its own digits each of which is raised to the power of the number of digits in that number. For example, 153 is an Armstrong Number because there are 3-digits in that number and 153 $\rightarrow 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$.

Each time the user provides an input, the program must evaluate whether it qualifies as valid. If the input fails to meet the criteria (whether due to alphabetic characters, punctuation marks, decimal points, negative signs, or any other non-conforming element) the program should reject it and prompt the user again.

The program should read two numbers from the command line and perform the following checks and operations:

1. Deal with corner cases where the number may be specified incorrectly i.e. with characters, special characters, decimal points, etc.
2. List all the Armstrong Numbers between the specified range.

The main class should in the cs250.exam package and named ArmstrongNumber. You may have other classes if you'd like.
Usage: java cs250.exam.ArmstrongNumber <lower> <upper>

Sample Input/Outputs
Example-1:
```
java cs250.exam.ArmstrongNumber  100    b
```

Example-2:
```
java cs250.exam.ArmstrongNumber 100   1000
The number range that was provided [100 – 1000 ] is valid.

The Armstrong Numbers in this range are

153
370
371
407
```

[2 points]
Problem 4. Summing the Digits in a Number
Write a program that reads in a number from the command line and adds up all the digits in that number. The program should read in an arbitrary length number from the command line and perform the following checks and operations:

1. Deal with corner cases where the number may be specified incorrectly. If the input fails to meet the criteria (whether due to alphabetic characters other than a comma, punctuation marks, negative signs, or any other non-conforming element) the program should reject it and prompt the user again.
2. Ignore decimal points and commas that be supplied as part of the number
3. Add up all the digits in that number
4. Find the twin-primes that are closest to that number that was computed in (3). A twin prime is a prime number that is either 2 less or 2 more than another prime number—for example, either member of the twin prime pair (17, 19) or (41, 43).

The main class should in the cs250.exam package and named NumberSummer:

Usage: java cs250.exam. NumberSummer <number>

Sample Input/Outputs

Example-1:
java cs250.exam.NumberSummer 134&78

The number that was provided [134&78] is not a valid number.

Example-2:
java cs250.exam.NumberSummer 32456

For example: Consider a number 302,451.23

The number that was provided [302,451.23] is a valid number.

The sum of the digits in this number is: 20

The twin primes closest to 20 is: (17, 19)

[2 points]
Problem 5: Sentence Transformer
Write a Java program that takes a single sentence (provided as one command-line argument enclosed in quotation marks) and applies the following sequence of transformations in the specified order:

1. Remove all punctuation. Strip the sentence of all punctuation marks, preserving only letters, digits, and whitespace.
2. Reverse each word individually while preserving word order. For every word in the sentence, reverse its characters, but keep the original sequence of words unchanged.
3. Replace each vowel with the next vowel in the cycle. Substitute each vowel with the next one in the cyclic sequence a → e → i → o → u → a, treating uppercase and lowercase vowels equivalently and preserving the original case in the output.
4. Print the final result: Output the fully transformed sentence as a single string.

If no sentence is provided, print a helpful usage message.

The main class should be in the cs250.exam package and named SentenceTransformer. You may have other classes if you'd like.

Usage: java cs250.exam. SentenceTransformer “<sentence>” where:

Sample Input/Output

java SentenceTransformer “Programming is fun, isn't it?”

Gnimmargerp se snf n'snt te?

[2 points]
Problem 6: Pangram Checker

A pangram is a sentence that contains every letter of the English alphabet at least once. Your task is to implement a function that checks whether a given sentence is a perfect pangram, a regular pangram, or not a pangram at all.

- A *perfect* pangram contains every letter of the alphabet exactly once.
- A *regular* pangram contains every letter at least once, but some may repeat.
- Not a pangram is any sentence that doesn't contain all the letters of the alphabet.

Usage: java cs250.exam.Pangram “<sentence>” where:

Sample Input/Output

Example 1:

```
java cs250.exam.Pangram  "The quick brown fox jumps over a lazy dog"

This is a pangram
```

Example 2:

```
Java cs250.exam.Pangram “Mr. Jock, TV quiz PhD, bags few lynx.”
This is a perfect pangram
```

[2 points]
Problem 7: Twin Primes
Prime numbers are the fundamental building blocks of arithmetic, defined as numbers greater than 1 that are divisible only by 1 and themselves. Their distinct properties make them central to many areas of mathematics and computer science, including number theory and cryptography. A twin prime is one of a pair of prime numbers that differ by exactly 2, such as (3, 5), (5, 7), (11, 13), (17, 19), (29, 31), or (41, 43). These pairs are often described as the inseparable siblings of the number world—always close together and always both prime.

Your goal is to write a program that lists all twin-primes within a range (inclusive of lower and upper bounds) specified at the command line. To do that, you need to write your own method (and not use the isPrime method from the language library) to decide whether a number is prime. A practical approach is to test whether the number has any divisors other than 1 and itself. The nice shortcut is that you only need to try divisors up to the square root of the number. If the number had a factor larger than that, it would have to be paired with a smaller factor you would have already tested. For example, to test 37 you only need to try 2 through 6, because 6×6=36 but 7×7=49 is already bigger than 37.

Each time the user provides an input, the program must evaluate whether it qualifies as valid. If the input fails to meet the criteria (whether due to alphabetic characters, punctuation marks, decimal points, negative signs, or any other non-conforming element) the program should reject it and prompt the user again. This means that your program should gracefully reject invalid inputs like letters ("abc"), punctuation ("@#$"), decimal numbers ("10.5"), or anything that isn't a clean, positive whole number. It should also check, if the given upper bound number is larger than the given lower bound. If something's off, prompt the user to enter valid numbers again.

Usage: java cs250.TwinPrimes <lower> <upper>

Sample Input/Outputs
Example-1:
```
java cs250.exam.TwinPrimes  100    b

The specified upper range ("b") is not a number.
```

Example-2:
```
java cs250.TwinPrimes 100   300
The number range that was provided [100 – 300] is valid.

The Twin Primes in this range are

(101,103)
(107,109)
(137,139)
(149,151)
(179,181)
(191,193)
(197,199)
(227,229)
(239,241)
(269,271)
(281,283)
```

[2 points]
Problem 8: Day-of-Year Calculator

Dates look simple until you remember leap years and different month lengths. Your objective is to take a date string in the YYYY-MM-DD format as a command line argument. Please note that the input must be exactly 10 characters long in the form YYYY-MM-DD (e.g., 2024-03-01); months and days must use two digits (so March is 03, not 3) and year must use 4-digits, so the year is 2026 not 26.

You will then perform two operations. First, you will validate it (including month lengths and leap years). Next, you will compute the corresponding day of year (Jan 1 → 1). Date parsing + leap-year logic is a classic real-world validation task. It's also a clean test of conditionals and careful boundary checking. Your program should print exactly one line either with the day-of-year result or with a clear error message. For error handling you must add add explicit checks: (1) Year must be a positive integer (e.g., 0000 is not allowed)., (2) Month must be between 01 and 12., and (3) Day must be valid for the given month (including leap-year rules for February).

Months with 31 days are January, March, May, July, August, October, and December.
Months with 30 days are April, June, September, and November.
February has 28 days in a normal year, and 29 days in a leap year.

Leap year rule: a year is a leap year if it is divisible by 4; years divisible by 100 are not leap years unless also divisible by 400.

Usage: java cs250.exam.DayOfYear <YYYY-MM-DD>

Task: Input is a date string.

Error handling expected:
- Missing arg → usage.
- Wrong format → reject.
- Impossible dates (e.g., 2025-02-29) → reject with explanation.

Sample Input/Outputs
Input: java cs250.exam.DayOfYear 2024-03-01
Output: 2024-03-01 is day 61 of the year (leap year)

Input: java cs250.exam.DayOfYear 2025-02-29
Output: Invalid date: 2025-02-29 (2025 is not a leap year)

Input: java cs250.exam.DayOfYear 03/01/2024
Output: Invalid format. Expected YYYY-MM-DD.

[3 points]
Problem 11: Cash Register at a Toy Shop

Goals:
- Modulo and division operators to give out change.
- Experience with input and outputs

Imagine that you are working in a toy store. Kids are buying toys using cash. The cash register does not automatically dispense change. Your task is to write a Java program that looks at the cost of the item and the amount provided by the customer with the objective of reducing the number of lower-denomination coins that you return; i.e., you should return as much as possible in bigger denomination coins.

Now, there’s a catch: the register you are using is charmingly old-fashioned. It doesn’t figure out the change for you. That’s your job. But not just any change … you want to be efficient. Graceful. Elegant, even. You want to return change in the fewest coins possible, leaning towards the larger denominations when you can.

Your task is to write a Java program that takes two numbers: the price of the toy and the amount the customer gives you. From these inputs, your program will calculate the change and express it using the fewest coins, favoring dollars, then quarters, then dimes, then nickels, then pennies. Like a good shopkeeper you want to make things simple and smart.

Here are the 5 steps in this assignment:
Step 1: Prompt the user to enter:
- The cost of the item (in dollars and cents, for example ‘18.28’).
- The amount paid (in dollars and cents, for example ‘20.00’).

Step 2: Convert both amounts to cents (integers) to avoid floating-point precision errors during arithmetic.

Step 3: Calculate the change by subtracting the cost from the amount paid. If the amount paid is less than the cost, display an error message and stop the program.

Step 4: Use division (/) and modulo (%) operators to break the change down into:
Dollars
Quarters (25¢)
Dimes (10¢)
Nickels (5¢)
Pennies (1¢)

Step 5: Output the total change amount in dollars, along with a clear, line-by-line breakdown of the number of each coin and dollar needed to make the change.

Sample Output
For example: If the toy costs $18.28 and the customer pays $20.00, you should return $1.72

```
Change to return: $1.72
Dollars: 1
Quarters: 2
Dimes: 2
Nickels: 0
Pennies: 2
```

[3 points]
Problem 12: ISBN-10 Validator
Books often come with a little "checksum" digit that helps catch typos when someone copies the number. In this problem, you'll take an ISBN-10 (hyphens/spaces are fine) and decide whether it's valid. You'll clean the input, compute the checksum, and print a clear VALID/NOT VALID message.

An ISBN-10 is 10 characters long (after you ignore hyphens/spaces). The last character is a check digit. It's computed from the first 9 digits (and itself) so that many common typing mistakes get caught.

Usage: java cs250.exam.ISBN10-Validator <ISBN>

Step 1: Clean the input. Your system should be accept inputs with spaces and hypens. For example:

- `0-306-40615-2`
- `0306406152`
- `0 306 40615 2`

As part of the cleaning step, you should remove hyphens "-" and spaces; you should keep everything else (for now) and then validate (step-2). After cleaning, you must have exactly 10 characters.

If not → print: `ISBN [...] is not valid (must be 10 characters after cleaning)`

Step 2: Validate character rules

Once you have 10 characters:

- Characters 0 through 8 (i.e., the first 9 characters) must all be digits `0–9`.
- Character 9 (i.e., the last one) can be:
  - a digit 0–9, or
  - X (uppercase; you can decide whether to accept lowercase x too, but be consistent)

If any character violates these rules → reject with a helpful message, e.g.:

- `Invalid character 'A' at position 4`
- `Only the last character may be X`

Step 3: Convert characters to numeric values. In particule, you should turn each character into a number:

- For positions 0–8: `'0'..'9'` → 0..9
- For the last position:
  - `'0'..'9'` → 0..9
  - `'X'` → 10

Now you have 10 integer values: `d1 d2 ... d10`.

Step 4: Compute the weighted checksum. Note that ISBN-10 uses weights 10 down to 1.
Compute the sum: $10 \cdot d_1 + 9 \cdot d_2 + 8 \cdot d_3 + \cdots + 2 \cdot d_9 + 1 \cdot d_{10}$

Then check: $\text{sum} \ mod\, 11 = 0$

- If the remainder is 0 → VALID
- Otherwise → NOT valid (checksum failed)

Walkthrough example (valid)

Example:0-306-40615-2
Cleaned:0306406152

Digits: d1=0 d2=3 d3=0 d4=6 d5=4 d6=0 d7=6 d8=1 d9=5 d10=2

Compute weighted sum:

- 10*0 = 0
- 9*3 = 27 (total 27)
- 8*0 = 0 (27)
- 7*6 = 42 (69)
- 6*4 = 24 (93)
- 5*0 = 0 (93)
- 4*6 = 24 (117)
- 3*1 = 3 (120)
- 2*5 = 10 (130)
- 1*2 = 2 (132)

Now: 132 % 11 = 0 → **VALID**

Walkthrough example (NOT VALID)
Example: 030640615X
Last char is X → d10 = 10.

Weighted sum:

- same first 9 contributions as above up to d9: total was 130 before the last digit
- last: 1 * 10 = 10
  Total = 140

Check:

- 140 % 11 = 8 → **NOT valid**

So 030640615X is NOT valid (even though X is allowed).

Typical edge cases that you are required to handle cleanly

- Too short after cleaning: "123-45" → reject based on length
- Bad character in the middle: "03A6406152" → reject based on positional information
- X not at the end: "X306406152" → reject
- Empty after cleaning: "-- --" → reject based on length

[3 points]
Problem 13: Credit Card Validator (Luhn Check)
Credit card numbers include a built-in "checksum" digit that helps catch typos when someone types the number. In this problem, you'll take a credit-card number (spaces and hyphens are fine) and decide whether it is valid under the Luhn algorithm. You'll clean the input, apply the Luhn checksum, and print a clear VALID/NOT VALID message.

A credit-card number is typically 13–19 digits long (after you ignore spaces/hyphens). The last digit is a check digit. The Luhn algorithm is designed so that many common typing mistakes (like a single wrong digit, or swapping two adjacent digits) will cause the checksum to fail.

Usage: java cs250.exam.LuhnValidator <card-number>

Step 1: Clean the input
Your system should accept inputs with spaces and hyphens. For example:

- `4539-1488-0343-6467`
- `4539148803436467`
- `4539 1488 0343 6467`

As part of the cleaning step, remove hyphens - and spaces; keep everything else (for now) and then validate (Step 2). After cleaning, you must have between 13 and 19 characters (inclusive). If that is not the case , print:

```
Card number [...] is not valid (must be 13 to 19 digits after cleaning)
```

Step 2: Validate character rules. Once you have the cleaned string: Every character must be a digit `0-9`. If any character violates this rule, reject the input with a helpful message, e.g.:

- `Invalid character 'A' at position 6`
- `Card number must contain digits only (after cleaning)`

Step 3: Convert characters to numeric values
Convert each character `'0'..'9'` into an integer `0..9`.
Now you have a list of digits: `d1 d2 ... dn` (where `n` is 13–19).

Step 4: Compute the Luhn checksum

1. The Luhn algorithm works from right to left:
2. Starting at the rightmost digit, move left.
3. Double every second digit (i.e., the digits in positions 2, 4, 6, … counting from the right).
4. If doubling produces a value 10 or more, subtract 9 from it.
   - Example: 7 doubled is 14 → subtract 9 → 5
5. Add up all digits (including the ones you didn't double).
6. If `sum % 10 == 0` → **VALID**; Otherwise → **NOT valid** (checksum failed)

Walkthrough example (VALID)
Example:`4539 1488 0343 6467`
Cleaned:`4539148803436467`
Work from right to left. Double every second digit:

Digits (right→left): 7, 6, 6, 4, 6, 3, 4, 3, 0, 3, 4, 3, 0, 8, 8, 4, 1, 9, 3, 5
(You do **not** have to print this; this is just for explanation.)

Doubling every second digit (from the right):

- 6 → 12 → 3
- 4 → 8
- 3 → 6
- 3 → 6
- 3 → 6
- 8 → 16 → 7
- 4 → 8
- 9 → 18 → 9
- …and so on.

After applying the rule to the appropriate digits, the total sum is divisible by 10 →**VALID**.
Print: `Card number [4539 1488 0343 6467] is VALID`

Walkthrough example (NOT VALID)
Example: `4539-1488-0343-6466`
Cleaned: `4539148803436466`

Note that this number above differs by one digit from the valid example above, and the checksum will fail.
Print: `Card number [4539-1488-0343-6466] is NOT valid (checksum failed)`

Error handling
You are required to handle several error conditions cleanly

1. Missing arg → print usage
2. If the card number is too short/too long after cleaning? Reject based on length
3. If the card number contains a non-digit after cleaning (letters, punctuation other than spaces/hyphens)? Reject with positional information
4. If the card number is empty after cleaning (e.g., `"-- --"`)? Reject based on length.

Sample Input/Outputs
Input: java cs250.exam.LuhnValidator 4539-1488-0343-6467
Output: Card number [4539-1488-0343-6467] is VALID

Input: java cs250.exam.LuhnValidator 4539-1488-0343-6466
Output: Card number [4539-1488-0343-6466] is NOT valid (checksum failed)

Input: java cs250.exam.LuhnValidator 123-45
Output: Card number [123-45] is not valid (must be 13 to 19 digits after cleaning)

Input: java cs250.exam.LuhnValidator 4539 1488 0343 64A7
Output: Card number [4539 1488 0343 64A7] is not valid (invalid character 'A' at position 15)

Input: java cs250.exam.LuhnValidator
Output: Usage: java cs250.exam.LuhnValidator <card-number>

[3 points]
Problem 14: Expression Evaluator
Arithmetic expressions look simple until you remember operator precedence. In this problem, you'll write a small evaluator that can correctly compute expressions like `12+3*4-5` (where multiplication happens before addition/subtraction). This is the kind of logic used in calculators, interpreters, and parsers … just in a simplified, approachable form. Your goal is to take a single expression string as a command line argument, validate it carefully, and then compute its integer result.

Usage: java cs250.exam.ExpressionEval <expression>

Allowed expressions

- Digits `0-9`(forming non-negative integers)
- Operators:`+`,`-`,`*`
- Optional spaces anywhere (you should ignore spaces)
- No parentheses
- No unary minus (so `-5+3` is not allowed; but `10-5+3` is allowed)

Step 1: Clean the input. Remove all spaces. Keep everything else for now.
If the cleaned expression is empty? print:
```
Expression [...] is not valid (empty expression)
```

Step 2: Validate character rules
After cleaning, every character must be either a digit or one of `+ - *`. If any character violates these rules? Reject with a helpful message, e.g.:
```
Expression […] is not valid (invalid character '@' at position 4)
```

Step 3: Validate structural rules
Your expression must follow this pattern: `number (operator number)*`. This means that:

- It must start with a digit
- It must end with a digit
- You must never have two operators in a row (12++3)
- You must never have an operator right at the beginning (+12) or end (12-)
- You must not have negative numbers written as -5 (unary minus is not supported)

If a rule is violated? Reject with a clear message, e.g.:

- `Expression [...] is not valid (cannot start with an operator)`
- `Expression [...] is not valid (two operators in a row at position 5)`
- `Expression [...] is not valid (cannot end with an operator)`

Step 4: Evaluate with correct precedence (* before + and -)
Compute the expression using standard precedence rules:

- Multiplication * happens first
- Then handle + and − from left to right

You may implement this in any reasonable way (e.g., a two-pass scan, or building lists of numbers/operators), but your program must produce the correct integer result.

Print: `Result: <value>`

Walkthrough example (VALID)
Example: `12+3*4-5`

- `3*4` happens first → `12+12-5`
- then left-to-right: `12+12 = 24, 24-5 = 19`
- Output:`Result: 19`

Error handling you are required to handle cleanly

- Missing arg? Print usage
- Empty after cleaning? Reject
- Invalid character (letters, punctuation, decimal points)? Reject with position
- Starts or ends with operator? Reject
- Two operators in a row? Reject with position
- Unary minus not allowed (`-5+2`)? Reject

Sample Input/Outputs
Input:java cs250.exam.ExpressionEval `"12+3*4-5"`
Output: Result: 19

Input: java cs250.exam.ExpressionEval `"7*8+2"`
Output: Result: 58

Input: java cs250.exam.ExpressionEval `"10--3"`
Output: Expression [10--3] is not valid (two operators in a row at position 3)

Input: java cs250.exam.ExpressionEval `"-5+2"`
Output: Expression [-5+2] is not valid (cannot start with an operator)

Input: java cs250.exam.ExpressionEval `"12+3a-5"`
Output: Expression [12+3a-5] is not valid (invalid character 'a' at position 5)

Input: java cs250.exam.ExpressionEval
Output: Usage: java cs250.exam.ExpressionEval <expression>

[3 points]
Problem 15: Anagram Grouper
Words can look different but still be rearrangements of the same letters. These are called anagrams, and grouping them is a classic way to practice string normalization and map-based aggregation; skills that show up in indexing, search, and text processing. Your goal is to take a comma-separated list of words, validate them, group them into anagram sets, and print the groups in a clean, predictable way.

Usage: java cs250.exam.AnagramGroups <comma-separated-words>

Step 1: Clean the input
- Split the input on commas.
- Trim spaces around each word.
- Treat words case-insensitively for grouping (so `Eat` and `tea` belong together), but print the words in their cleaned form.

If there are no words after splitting/cleaning? Print: `No words provided.`

Step 2: Validate word rules. Each word must:
- contain only letters `A-Z` or `a-z`
- be non-empty after trimming

If any word violates this? Reject with a helpful message, e.g.:
- `Invalid word: "b@t" (words must contain letters only)`
- `Invalid word: "" (empty word between commas)`

Step 3: Compute an anagram “signature”
To group anagrams, convert each word to a canonical signature:
1. lowercase it
2. sort its letters alphabetically
3. For example:
   - `Eat` → `aet`
   - `tea` → `aet`
   - `ate` → `aet`

Words with the same signature belong in the same group.

Step 4: Print groups (consistent format)
Print each group on its own line as: `<signature>: word1, word2, word3`

Requirements:
- Within each group, print words in alphabetical order (case-insensitive).
- Print groups in alphabetical order by signature.

Walkthrough example
Input: `eat, tea, ate, bat, tab, tan, nat`

Signatures:

- `eat/tea/ate →aet`
- `bat/tab → abt`
- `tan/nat → ant`

Output groups (sorted by signature):

- `abt: bat, tab`
- `aet: ate, eat, tea`
- `ant: nat, tan`

Error handling that you are required to handle cleanly

- Missing arg? Print usage
- Empty input / no valid words after cleaning? Reject
- Any word contains non-letters? Reject and name the word
- Empty word caused by "," or trailing comma? Reject

Sample Input/Outputs
Input: java cs250.exam.AnagramGroups "eat, tea, ate, bat, tab, tan, nat"
Output:
abt: bat, tab
aet: ate, eat, tea
ant: nat, tan

Input: java cs250.exam.AnagramGroups "Eat, TEA, ate "
Output: aet: ate, Eat, TEA

Input: java cs250.exam.AnagramGroups "bat,,tab"
Output: Invalid word: "" (empty word between commas)

Input: java cs250.exam.AnagramGroups "bat, b@t"
Output: Invalid word: "b@t" (words must contain letters only)

Input: java cs250.exam.AnagramGroups
Output: Usage: java cs250.exam.AnagramGroups <comma-separated-words>

[3 points]
Problem 16: Caesar Cipher (Encrypt or Decrypt)

Long before anyone worried about Wi-Fi passwords, Julius Caesar reportedly kept his military messages from curious eyes by using a simple trick: shift every letter a fixed number of steps down the alphabet. That's the *Caesar cipher*: an ancient "alphabet wheel" where A slides to D (if the shift is 3), B slides to E, and so on, wrapping around at the end. It's wonderfully low-tech, but it captures a real idea: turning readable text into something that looks like noise unless you know the key. More generally, Caesar cipher is one of the oldest "shift" ciphers: you slide each letter forward by *k* positions in the alphabet. It's simple, but it teaches careful character handling, wrap-around logic, and clean input validation. In this problem, you'll do the same thing Caesar did; except, your "scroll" is a command-line string and your key is an integer. You'll practice careful character handling (uppercase vs lowercase, punctuation, spaces) and learn why wrap-around logic matters. And while modern cryptography is vastly stronger, today's systems still rely on the same core principle: a small, precise rule can transform information in a controlled way—just with better math and much bigger keys. Your program will support both encryption and decryption by shifting letters while leaving everything else unchanged.

Usage: java cs250.exam.CaesarCipher <mode> <shift> "<message>"

- `<mode>` is either `enc` or `dec`
- `<shift>` is an integer (may be negative or larger than 26)
- `<message>` is a string (may contain spaces; quote it)

Rules

- Shift letters only (`A-Z`, `a-z`)
- Preserve case
- Leave digits, punctuation, and spaces unchanged
- For decryption, shift in the opposite direction (or equivalently use `-shift`)

Step 1: Validate arguments

- If missing args (you need 3 arguments)? Print usage.
- If mode is not `enc` or `dec`? Reject: `Invalid mode: <mode> (expected enc or dec)`
- If shift is not a valid integer? Reject: `Shift [<shift>] is not a valid integer.`

Step 2: Normalize the shift

- Reduce shift to the equivalent in `[0..25]`.
    - Example: shift `29` behaves like shift `3`.
    - (You may use mod 26 logic.)
- If mode is `dec`, you should reverse the direction of the shift.

Step 3: Transform the message
For each character:

- If it's `A-Z`, shift within uppercase range with wrap-around.
- If it's `a-z`, shift within lowercase range with wrap-around.
- Otherwise, copy it as-is.

Print exactly one line: `Result: <ciphertext-or-plaintext>`

Sample Input/Outputs

Input: `javacs250.exam.CaesarCipher enc 3 "Hello, World! 42"`
Output: `Result: Khoor, Zruog! 42`

Input: `java cs250.exam.CaesarCipher dec 3 "Khoor, Zruog! 42"`
Output: `Result: Hello, World! 42`

Input: `java cs250.exam.CaesarCipher enc 29 "abc"`
Output: `Result: def`

Input: `java cs250.exam.CaesarCipher dec x "Hi"`
Output: `Shift [x] is not a valid integer.`

Input: `java cs250.exam.CaesarCipher encrypt 3 "Hi"`
Output: `Invalid mode: encrypt (expected enc or dec)`

Input: `java cs250.exam.CaesarCipher`
Output: `Usage: java cs250.exam.CaesarCipher <mode> <shift> "<message>"`

[3 points]
Problem 17: Atbash Cipher
Atbash comes from the ancient world, traditionally associated with Hebrew writing, and it feels like a mirror held up to the alphabet: the first letter becomes the last, the second becomes the second-to-last, and so on. Instead of shifting by a key, Atbash swaps letters by reflection (A ↔ Z, B ↔Y, C ↔ X) which makes it oddly elegant: the same operation both encrypts and decrypts. Historically, it's the kind of cipher you could do by hand on parchment, which also means it's great for building intuition about how substitution ciphers work. In this task, you'll implement that "alphabet mirror" while keeping everything else (spaces, punctuation, numbers) exactly as it was. The contemporary twist is that this kind of letter-to-letter mapping shows up all over computing, even outside security: think encodings, lookup tables, and translation layers where input symbols must be converted predictably. You are basically writing a small program with a big idea: sometimes the most reliable transformation is the simplest one you can apply consistently.

Quick mapping rule (Atbash):

- Uppercase: `A ↔ Z, B ↔ Y, C ↔ X, …`
- Lowercase: `a ↔ z, b ↔ y, c ↔ x, …`
- Non-letters are unchanged
- Encrypt and decrypt are the same operation (running Atbash twice returns the original text)

Walkthrough #1 (ENCRYPT)
Input: `java cs250.exam.AtbashCipher "Hello, World!"`

Step 1: Validate arguments
One message argument present → OK

Step 2: Apply Atbash letter mapping

- `H ↔ S`
- `e ↔ v`
- `l ↔ o`
- `l ↔ o`
- `o ↔ l`
- `,` unchanged
- space unchanged
- `W ↔ D`
- `o ↔ l`
- `r ↔ i`
- `l ↔ o`
- `d ↔ w`
- `!` unchanged

Output: `Result: Svool, Dliow!`

Walkthrough #2 (ENCRYPT with digits/punctuation)
Input: `java cs250.exam.AtbashCipher "CS250: fun!"`
Step 1: Validate arguments
One message argument present → OK

Step 2: Apply Atbash letter mapping

- `C ↔ X`
- `S ↔ H`
- `2 5 0` unchanged
- `:` unchanged
- space unchanged
- `f ↔ u`
- `u ↔ f`
- `n ↔ m`
- ! unchanged

Output: `Result: XH250: ufm!`

Walkthrough #3 (DECRYPT)
Input: `java cs250.exam.AtbashCipher "Svool, Dliow!"`

Step 1: Validate arguments
One message argument present → OK

Step 2: Apply Atbash letter mapping again
(Same mapping as encryption. Note that Atbash is its own inverse.)

- `S ↔ H`
- `v ↔ e`
- `o ↔ l`
- `o ↔ l`
- `l ↔ o`
- punctuation/space unchanged
- `D ↔ W`
- `l ↔ o`
- `i ↔ r`
- `o ↔ l`
- `w ↔ d`

Output: `Result: Hello, World!`

Walkthrough #4 (DECRYPT)
Input: `java cs250.exam.AtbashCipher "XH250: ufm!"`

Step 1: Validate arguments
One message argument present → OK

Step 2: Apply Atbash letter mapping again

- `X ↔ C`
- `H ↔ S`
- digits/punctuation unchanged
- `u ↔ f`
- `f ↔ u`
- `m ↔ n`

Output: `Result: CS250: fun!`

[3 points]
Problem 18: Prime Factorization Printer
Prime numbers are the "building blocks" of whole numbers: every integer greater than 1 can be written uniquely as a product of primes (this is the **Fundamental Theorem of Arithmetic**). In this problem, you'll take a positive whole number and break it into its prime factors, reporting how many times each prime divides it. This is a classic use of loops and integer arithmetic, and it's the kind of logic that shows up in cryptography, number theory, and performance tuning (e.g., reducing fractions, simplifying computations). Your goal is to read one integer from the command line, validate it carefully, and print its prime factorization in a clean, consistent format.

Usage: java cs250.exam.PrimeFactorization <n>

Step 1**:** Validate arguments
Missing arg? Print usage.

Step 2: Validate numeric rules
The input must be a clean positive whole number (digits only). Reject anything with:

- letters (`abc`)
- punctuation (`@#$`)
- decimals (`10.5`)
- negative signs (`-12`)
- empty input

If invalid? Print: `Input [<n>] is not a valid positive integer.`

Also: If n < 2 → reject (1 and 0 do not have prime factorizations):
`Input [<n>]  is not valid (n must be >= 2).`

Step 3: Compute prime factors using repeated division
Factorization can be done by trying divisors starting from 2:

1. Start with `d = 2`.
2. While `n` is divisible by `d`, divide `n` by `d` and count how many times it divides.
   - That count becomes the exponent for factor d.
3. Move to the next possible divisor:
   - after 2, you can try only odd numbers 3, 5, 7, ...
4. Stop once `d*d > n`:
   - If the remaining n is greater than 1, it is itself a prime factor.

This method uses only loops, %, and /, and runs fast enough for typical exam inputs that we will provide.

Step 4: Print the factorization (required format)
Print exactly one line in this format:
```
<original> = p1^e1 * p2^e2 * ... * pk^ek
```

Rules:

- Factors must appear in increasing order of the prime (2, then 3, then 5, …).
- Always print exponents using ^.
- If a factor occurs only once, print exponent 1 (e.g., 7^1).

Example format: `360 = 2^3 * 3^2 * 5^1`

Walkthrough example (VALID)
Example:n = 360
Try dividing by 2:

- 360 ÷ 2 = 180 (count 1)
- 180 ÷ 2 = 90 (count 2)
- 90 ÷ 2 = 45 (count 3)
  Now 45 is not divisible by 2 → so we have 2^3.

Next divisor is 3:

- 45 ÷ 3 = 15 (count 1)
- 15 ÷ 3 = 5 (count 2)
  Now 5 is not divisible by 3 → so we have 3^2.

Now we check: remaining n = 5. Since 5 > 1, it is a prime factor 5^1.

Final result: 360 = 2^3 * 3^2 * 5^1

Walkthrough example (VALID, prime input)
Example: n = 97

- 97 not divisible by 2, 3, 5, 7, 9…
- Once the divisor gets large enough that d*d > 97, there are no smaller factors left to find.
- Remaining n = 97 → it is prime.

Final result: 97 = 97^1

Error handling you are required to handle cleanly

- Missing arg? `Print` usage
- Non-digit input / decimals / negative sign? Reject with information
- n < 2 → reject with explanation

Sample Input/Outputs
Input: java cs250.exam.PrimeFactorization 360
Output: 360 = 2^3 * 3^2 * 5^1

Input: java cs250.exam.PrimeFactorization 72
Output: 72 = 2^3 * 3^2

Input: java cs250.exam.PrimeFactorization 97
Output: 97 = 97^1

Input: java cs250.exam.PrimeFactorization 1
Output: Input [1] is not valid (n must be >= 2).

Input: java cs250.exam.PrimeFactorization 10.5
Output: Input [10.5] is not a valid positive integer.

Input: java cs250.exam.PrimeFactorization -12
Output: Input [-12] is not a valid positive integer.

Input: java cs250.exam.PrimeFactorization
Output: Usage: java cs250.exam.PrimeFactorization <n>

Version Change History

This section will reflect the change history for the assignment. It will list the version number, the date it was released, and the changes that were made to the preceding version.

Version	**Date**	**Comments**
1.0	1/20/2026	First public release of the problem set.