Programming Exercises

OPTIONAL AND EXTRA CREDIT

VERSION 1.0

The objective of this assignment is to help you hone your programming skills in several Java concepts. This programming exercise is *Optional & Extra Credit*. There are 20 programming exercises, and each accounts for 0.1 points of extra-credit towards your cumulative course grade. The assignments are due in batches as outlined in the due dates on Canvas.

What should I prioritize: the regular HW1 or the programming exercises?
You should prioritize the homework. For e.g., HW1 is worth 5 points towards your cumulative course grade --- i.e., individually HW1 is worth 50 times more than each programming exercise.

DUE DATE:

- 01 through 05 are due on Wednesday, January 28th, @ 8:00 pm MT
- 06 through 10 are due on Wednesday, February 4th, @ 8:00 pm MT
- 11 through 15 are due on Wednesday, February 11th, @ 8:00 pm MT
- 16 through 20 are due on Wednesday, February 18th, @ 8:00 pm MT

Generative AI Use and Consequences
Use of AI tools such as ChatGPT, Claude, Github Co-Pilot, and/or their ilk to write or "improve" your code or written work at *any* stage is prohibited; this includes the ideation phase. It is your responsibility to ensure that you don't have the GitHub Co-Pilot extension installed in your IDE; assignment solutions generated by Co-Pilot aren't written by you. Turning in code or an essay written by generative AI tools will be treated as turning in work created by someone else, namely an act of plagiarism and/or cheating.

Ultimately, you will get out of the class what you put in. Simply copying and pasting code from generative AI tools is neither ethical nor does it contribute to your learning experience. There are multiple reasons why these generative AI tools are detrimental to your learning experience:

1. They rob you of the ability to think and learn the concepts for yourself. Solving problems is an essential step to gaining a solid understanding of the material.
2. You will struggle with the in-classroom quizzes and exams where you will not have access to these tools.
3. While we acknowledge that these tools are likely to become an important part of a software engineer's workflow in the future, you are much more likely to use these tools in an effective manner if you already have expertise in the relevant technical topics. Developing such expertise requires putting in the effort to learn these topics without the assistance of these tools.
4. These tools are prone to generating imperfect or even incorrect solutions, so trusting them blindly can lead to bad consequences.

Auto-grading in seconds
Programming assignments are being autograded and the scores will be reflected in Canvas less than 30 seconds after you have submitted. You have unlimited attempts (till the submission deadline), and your highest score will be retained. Use of these autograders is predicated on you following the outputs exactly as specified. If you are having trouble printing outputs in the prescribed format, please get in touch with the TAs. Don't procrastinate and start early.

1 Programming Exercises

1.1 Hello World

Goals

- Familiarizing with Java Package Structure
- Familiarizing with Java Syntax
- Reading from the command Line

Name/Package

- Package Structure; `cs250/exercises/Exercise1.java`
- File name: `Exercise1.java`

Instructions

1. Print the 1st element passed in through the command line
2.

Restrictions

1. No imports

Submission

- Zip the cs250 folder and submit, i.e cs250 should be the first and only directory in the zip, cs250 should contain the subfolders needed for the package
- The zip can be named anything
- Do not include any other files in your submission except a .java file

1.2 Types, Operators

Goals

- Familiarizing with Java Data Types
- Familiarizing with Operators on Numbers
- Familiarizing with Parsing Strings

Name/Package

- Package Structure; cs250/exercises/Exercise2.java
- File name: Exercise2.java

Instructions

- In the main method, read each argument passed in through the command line (there will be three of types args[0] = 'int', args[1] = 'float', args[2] = 'long' in that order)
- Print to stdout (in the same order):
    1. args[0] as a String
    2. args[1] as a String
    3. args[2] as a String
    4. args[0] + args[1] + args[2]
    5. args[0] - args[1] - args[2]
    6. args[0] * args[1] * args[2]
    7. args[0] / args[1] / args[2]
    8. args[1] ^ args[0]
    9. args[2] % args[0]
    10. args[2] / -args[1]

Example

- Input

```
10 20.5 9223372036854775710
```

- Output

```
10
20.5
9223372036854775710
9.223372036854776e+18
-9.223372036854776e+18
1.890791267555229e+21
5.2887910852951435e-20
13108065732570.703
0
-4.499205871636476e+17
```

Restrictions

1. No imports

Submission

- Zip the cs250 folder and submit, i.e cs250 should be the first and only directory in the zip, cs250 should contain the subfolders needed for the package
- The zip can be named anything
- Do not include any other files in your submission except a .java file

1.3 IO, Random

Goals

- Familiarizing with Random generators
- Familiarizing with Printing to Stdout
- Familiarizing with Reading from Stdin

Name/Package

- Package Structure; `cs250/exercises/Exercise3.java`
- File name: `Exercise3.java`

Instructions

- Use the random class to generate
    1. a float in the range(10.50, 100.75) inclusive and
    2. an int in the range (10, 20) inclusive
- Print the integer first and then the float to stdout
- Use the scanner class to read two inputs from stdin
    - (make sure to use hasNextLine, nextLine, and close)
- Add all 4 values and print the sum to stdout

Example

- Input

```
31.22 65
```

- Output

```
19
10.54
125.76
```

Restrictions

1. No imports except `java java.util.Random` and `java.util.Scanner`

Submission

- Zip the cs250 folder and submit, i.e cs250 should be the first and only directory in the zip, cs250 should contain the subfolders needed for the package
- The zip can be named anything
- Do not include any other files in your submission except a .java file

1.4 Strings

Goals

- Familiarizing with Chars and Strings in Java
- Familiarizing with String methods
- Familiarizing with String manipulation

Name/Package

- Package Structure; `cs250/exercises/Exercise4.java`
- File name: `Exercise4.java`

Instructions

- In the main method, read each argument passed in through the command line (there will be six characters)
- Print to stdout (in the same order):
    1. All characters concatanated together
    2. The count of 'a' in the concatanated string
    3. The concatanated string in all uppercase
    4. The concatanated string in all lowercase
    5. The concatanated string from index 1 to index 5
    6. The concatanated string without the character at index 2
    7. The concatanated string where all . (dot) is replaced by an _ (underscore)
    8. The last index of the character 'e' in the concatanated string
    9. The concatanated string in reverse
    10. The sum of the ASCII values of all characters as an integer

Example

- Input

```
r a N d . M
```

- Output

```
raNd.M
1
RAND.M
rand.m
aNd.M
rad.M
raNd_M
-1
M.dNar
512
```

Restrictions

1. No imports
2. No loops

Submission

- Zip the cs250 folder and submit, i.e cs250 should be the first and only directory in the zip, cs250 should contain the subfolders needed for the package
- The zip can be named anything

- Do not include any other files in your submission except a .java file

1.5 Conditionals

Goals

- Familiarizing with Conditionals
- Familiarizing with Nested Conditionals
- Familiarizing with Boolean and Comparison Operators

Name/Package

- Package Structure; `cs250/exercises/Exercise5.java`
- File name: `Exercise5.java`

Instructions

- In the main method, read the argument passed in through the command line
- Print to stdout (in the same order):
  1. If the string contains a number, print 'number'; else print 'no number'
  2. If the string contains the letter 'a' or the letter 'b', print 'true'; else print 'false'
  3. If the length of the string is greater than 5 and less than 10, print the length in words (all lower), ie. six not 6; else, print the numeric value (ideally use a switch for this).
  4. If the length of the string is odd, print 'odd'; if it is even, print 'even'
  5. If the char at index 3 is not in lowercase, print the char as lowercase; else print it in uppercase

Example

- Input

```
• a6sUNHJn1
```

- Output

```
• number
• true
• nine
• odd
• u
```

Restrictions

1. No imports
2. No loops

Submission

- Zip the cs250 folder and submit, i.e cs250 should be the first and only directory in the zip, cs250 should contain the subfolders needed for the package
- The zip can be named anything
- Do not include any other files in your submission except a .java file

1.6 Arrays

Goals

- Familiarizing with 1D and 2D Arrays in Java
- Familiarizing with array data types
- Familiarizing with array manipulation

Name/Package

- Package Structure; cs250/exercises/Exercise6.java
- File name: Exercise6.java

Instructions

- In the main method, read three argument passed in through the command line
  - the 1st is the type of either ('int', 'float', 'char')
  - the 2nd is the x
  - the 3rd is the y
- Construct the appropriate array of dimensions 4 x 3 with the correct type. If int, fill all with a random int. If float, fill all with a random float. If char, fill all with a random character.
- Print to stdout (in the same order):

1. the 2D array using "Arrays.deepToString()". You will have to import "java.util.Arrays"
2. the value at (x, y)
3. the array at index (x - y) in sorted order

Example

- Input

```
int 3 1
```

- Output

```
[[1, 9, 7], [4, 2, 3], [0, 1, 2], [7, 3, 9]]
3
[0, 1, 2]
```

Restrictions

1. No imports except java.util.Arrays and java.util.random
2. No loops

Submission

- Zip the cs250 folder and submit, i.e cs250 should be the first and only directory in the zip, cs250 should contain the subfolders needed for the package
- The zip can be named anything
- Do not include any other files in your submission except a .java file

1.7 Loops

Goals

- Familiarizing with For Loops
- Familiarizing with 2D arrays

Name/Package

- Package Structure; `cs250/exercises/Exercise7.java`
- File name: `Exercise7.java`

Instructions

- In the main method, read the two arguments passed in through the command line and parse them as integers
- The first is the starting index and the last is the end index
- Create a square matrix of dimensions (end - start + 1) x (end - start + 1)
    - Each cell should be the product of the two at that index
    - For example, given the args (6, 9), the matrix should look like:

36	42	48	54
42	49	56	63
48	56	64	72
54	63	72	81

    - There are 16 cells. The row header is [6, 7, 8, 9] and the column header is [6, 7, 8, 9]
- Print the 2D array using "Arrays.deepToString()". You will have to import "java.util.Arrays"
- Print the sum of the diagonal values starting from the top right to the bottom left. In the example, it is 220.

Example

- Input

```
9 12
```

- Output

```
[[81, 90, 99, 108], [90, 100, 110, 120], [99, 110, 121, 132], [108, 120, 132, 144]]

436
```

Restrictions

1. No imports except `java.util.Arrays`

Submission

- Zip the cs250 folder and submit, i.e cs250 should be the first and only directory in the zip, cs250 should contain the subfolders needed for the package
- The zip can be named anything
- Do not include any other files in your submission except a .java file

1.8 Exceptions

Goals

- Familiarizing with Exceptions in Java
- Familiarizing with Try/Catch/Throw
- Familiarizing with Different Conditions for Exceptions

Name/Package

- Package Structure; `cs250/exercises/Exercise8.java`
- File name: `Exercise8.java`

Instructions

- In the main method, paste this block of code

```
Integer a = Integer.valueOf(Integer.parseInt(args[0]));
Integer b = Integer.valueOf(Integer.parseInt(args[1]));
Integer c = Integer.valueOf(Integer.parseInt(args[2]));
if (a > 100 || b > 100 || c > 100)
    throw new IllegalArgumentException("Args must be less than or equal to 100!");
Integer sum = a + b + c;
Integer d = sum < 100 ? null : sum;
Object value = d < 150 ? (d / c / b / a) : "CORRECT";
String valStr = (String) value;

System.out.println(valStr);
```

  - You can put it in a try block, but do NOT remove any code as that will cause you to lose points
- Catch the potential exceptions (there are 6) and
  - print the name of the exception using `System.out.println(<your_exception>.getClass().getSimpleName())`
  - print the message using `System.out.println(<your_exception>.getMessage())`
  - Do NOT print the stack trace
  - You should not use the general Exception class
  - You program should exit with an exit code 0, not 1
- At the end of the program, print ‘END’ in all upper case in a finally block

Example

- Output

```
ArrayIndexOutOfBoundsException
Index 2 out of bounds for length 2
END
```

Restrictions

1. No imports
2. Do not use Exception to catch exceptions

Submission

- Zip the cs250 folder and submit, i.e cs250 should be the first and only directory in the zip, cs250 should contain the subfolders needed for the package
- The zip can be named anything
- Do not include any other files in your submission except a .java file

1.9 Methods

Goals

- Familiarizing with Methods in Java
- Familiarizing with Sorting
- Familiarizing with methods that belong to the class and object

Name/Package

- Package Structure; cs250/exercises/Exercise9.java
- File name: Exercise9.java

Instructions

- Create two methods with the following method definitions

```
• public static int[] sortInts(int[] arr) {}

• public String[] sortStrings(String[] arr) {}
```

- You should implement the two methods with any sorting algorithm and return a sorted array of strings or ints
- Both methods should be in the Exercise9 class. You can use the main method for testing, but it will not be graded

Restrictions

1. No imports

Submission

- Zip the cs250 folder and submit, i.e cs250 should be the first and only directory in the zip, cs250 should contain the subfolders needed for the package
- The zip can be named anything
- Do not include any other files in your submission except a .java file

1.10 Classes

Goals

- Familiarizing with Classes in Java
- Familiarizing with Abstraction and Inheritence

Name/Package

- Package Structure; cs250/exercises/Exercise10.java
- File name: Exercise10.java

Instructions

- In the same folder as 'Exercise10.java', create a Bank.java and Account.java
  - In Bank.java, copy this abstract class and create a public Bank class that extends BankActions; complete the abstract methods in the Bank class

```
abstract class BankActions {
    protected ArrayList<Account> accounts;
    public abstract void addAccount(Account account);
    public abstract Account findAccount(String name);
    public abstract void listAccounts();

    public void performTransactions(String name, double amount, String type) {
        Account account = findAccount(name);
        if (account != null) {
            if (type.equals("deposit"))
                account.deposit(amount);
            else if (type.equals("withdraw"))
                account.withdraw(amount);
        }
    }
}
```

  - In Account.java, copy this abstract class and create a public Account class that extends AccountActions; complete the abstract methods in the Account class

```
abstract class AccountActions {
    protected String name;
    protected double balance;
    protected AccountActions(String name, double balance) {
```

```
        this.name = name;
        this.balance = balance;
    }
    protected abstract String getName();
    protected abstract double getBalance();
    // make sure to check if the deposit is > 0
    protected abstract void deposit(double amount);
    // make sure to check if  0 < amount <= balance
    protected abstract void withdraw(double amount);

    public String toString() {
        return getName() + ":" + getBalance();
    }
}
```

- Do not modify the two classes above
- You can use your Exercise10.java class for testing
- All interactions should be done through the Bank class

Restrictions

1. No imports except `java.util.ArrayList`

Submission

- Zip the cs250 folder and submit, i.e cs250 should be the first and only directory in the zip, cs250 should contain the subfolders needed for the package
- The zip can be named anything
- You should only include your Bank.java, Account.java, and Exercise10.java in your submission

1.11 Package, Imports

Goals

- Familiarizing with Packages in Java
- Familiarizing with Imports in Java

Name/Package

- Package Structure; cs250/exercises/Exercise11.java
- File name: Exercise11.java

Instructions

- In your cs250 folder, create a new folder called "helpers" with a single file called "ExerciseHelper.java"
- This file should contain one function with the method definition below; complete the method body

```
• public void transposeMatrix(int[][] arr) {}
```

- In your Exercise11.java under cs250/exercises, import this file as well as a file named 'IntMatrix.java' from the root directory under matrices/, i.e the file where cs250 is
- This file will be placed there when testing; the dir at that time will look like:

```
|--cs250
    |--exercises
        |--Exercise11.java
    |--helpers
        |--ExerciseHelper.java
|--matrices
    |--IntMatrix.java
```

- Use the non-static method of the IntMatrix class called generateIntMatrix() to get a randomly generated matrix of any size. The method definition looks like this:

```
public int[][] generateIntMatrix() {}
```

- Use the transposeMatrix method you created to transpose the matrix and print it to stdout using Arrays.deepToString()

Restrictions

1. No imports except java.util.Arrays

Submission

- Zip the cs250 folder and submit, i.e cs250 should be the first and only directory in the zip, cs250 should contain the subfolders needed for the package
- The zip can be named anything
- Your submission should look like:

- |<name>.zip
-    |--cs250
-       |--exercises
-          |--Exercise11.java
-       |--helpers

          |--ExerciseHelper.java

1.12 File, IO

Goals

- Familiarizing with Files in Java
- Familiarizing with reading from and writing to files
- Familiarizing with creating and deleting files

Name/Package

- Package Structure; `cs250/exercises/Exercise12.java`
- File name: `Exercise12.java`

Instructions

- In the main method, read a file that will be copied to the root directory named 'input.csv', i.e the directory where cs250 is located in the zip
    - The file will be a CSV file that contains multiple lines of two things, the file path and the content of the file
    - For example, the file might look like

```
• dir1/file1.txt, some_string_or_num
• file2.txt, hello_world
• dir3/subdir/3/file3.txt, this is a file manipulation exercise
```

- Read each value from the file and create those files with the appropriate directory structure in the root directory
    - The file should have the content defined in input.csv
- At the end, delete the input.txt file from the directory where it is located
- You will be graded on the file structure and the contents of the file

Restrictions

1. No imports except those in `java.io.*` and `java.util.*`

Submission

- Zip the cs250 folder and submit, i.e cs250 should be the first and only directory in the zip, cs250 should contain the subfolders needed for the package
- The zip can be named anything
- Do not include any other files in your submission except a .java file

1.13 Lists

Goals

- Familiarizing with ArrayLists and LinkedLists
- Familiarizing with Performance of Lists
- Familiarizing with Time Comparison

Name/Package

- Package Structure; cs250/exercises/Exercise13.java
- File name: Exercise13.java

Instructions

- Complete the body to the method definition below

```
• public double[][] compareLists(int[] arr, int searchVal) {}
```

- The method should create an ArrayList and a LinkedList with the values in the arr (expect >10,000 values)
- For the 3 data structures, array, ArrayList, and LinkedList, store the time it takes to
    1. Check if the list contains the value for searchVal
    2. Remove searchVal from the list
    3. Add SearchVal to the beginning of the list
- Add the times for each data structure to an array and return that 2D array
    - The result should look like

	Contains	**Remove**	**Add**
Array	1	2	3
ArrayList	1	2	3
LinkedList	1	2	3

- You can assume that the arr will always contain searchVal

Restrictions

1. No imports

Submission

- Zip the cs250 folder and submit, i.e cs250 should be the first and only directory in the zip, cs250 should contain the subfolders needed for the package
- The zip can be named anything
- Do not include any other files in your submission except a .java file

1.14 Maps, Sets

Goals

- Familiarizing with Map operations
- Familiarizing with Set operations
- Familiarizing with Hashing

Name/Package

- Package Structure; `cs250/exercises/Exercise14.java`
- File name: `Exercise14.java`

Instructions

- In the main method, read a file that will be copied to the root directory named 'input.txt', i.e the directory where cs250 is located in the zip
- You will also have to read the arguments list for a list of strings
- For each word delimited by a space in the file, add it to a map and count the number of occurences
    - You should use a HashMap to keep a count of the words (keys) and count (value)
    - Print the HashMap to stdout using toString after completion
- For each word delimited by a space in the file, add it to a HashSet
    - Get the disjunction/union of the HashSet and the list passed through the command line
    - Print the HashSet to stdout using toString after completion

Example

- File Input

```
hello world, I am learning about maps and about sets
```

- Cmd Line Input

```
maps sets world
```

- Outputs

```
{maps=1, sets=1, and=1, world,=1, about=2, I=1, learning=1, hello=1, am=1}
[and, world,, about, I, learning, hello, am]
```

Restrictions

1. No imports except `java.util.HashMap`, `java.util.HashSet`, `java.io.*`

Submission

- Zip the cs250 folder and submit, i.e cs250 should be the first and only directory in the zip, cs250 should contain the subfolders needed for the package
- The zip can be named anything
- Do not include any other files in your submission except a .java file

1.15 Generics, Tuples

Goals

- Familiarizing with Generics in Java
- Familiarizing with Tuples

Name/Package

- Package Structure; cs250/exercises/Exercise15.java
- File name: Exercise15.java

Instructions

- Create a file named Tuple.java and another named TupleStructure.java
  - A tuple is a data structure that is ordered and unchangeable
  - Tuple.java should contain an interface with the following code

```
interface Tuple<T> {
    public boolean contains(T value);
    public T get(int index);
    public int indexOf(T value);
    public int size();
    public T[] toArray();
    public TupleStructure<T> join(TupleStructure<T> tuple);
    public TupleStructure<T> multiply(int times);
}
```

  - In TupleStructure.java, you should implement this interface and override all methods

Restrictions

1. No imports

Submission

- Zip the cs250 folder and submit, i.e cs250 should be the first and only directory in the zip, cs250 should contain the subfolders needed for the package
- The zip can be named anything
- You should have Tuple.java, TupleStructure.java, and an optional Exercise15.java in your submission

1.16 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

1.17 The Universal Register: Change for Every Currency

Goals:

- Parsing structured input and working with configuration files
- Modulo and division operators to enable currency systems modeling.
- Experience with input and outputs

We will now update the program that you wrote in the previous exercise to cope with diverse currencies (i.e., not just the dollar). We also don't just want to deal with muggle money but wizarding coins (from the world of Harry Potter) such as Galleons, Sickles, and Knuts. Because currencies don't have to be decimal-based (1 dollar = 100 cents), our previous prompt to ask the user for the price in the largest unit (e.g. 12.37) does not work properly for a currency like Wizard Coin (1 Galleon = 27 Sickles, 1 Sickle = 29 Knuts) anymore. Instead now we will always ask for the price and payment amount in the smallest given unit. The program now takes in:

- The price of an item in the **smallest given unit**.
- The amount the customer paid in the **smallest given unit**.
- A configuration file that describes the currency system and available denominations. This configuration file allows you to handle any well-formed currency system, not just dollars and cents.

Just like before your program will calculate the change and return it using the fewest number of coins, always favoring larger denominations.

Step 1: Prompt the user to enter:

- Cost of the item (e.g., 12.37 --> 1237 of the smallest given unit, meaning 12.37$ or 1237 Knuts, etc., based on given configuration file)
- Amount paid (e.g., 1500)
- Name of the currency configuration file (e.g., galleons.json)

Step 2: Parse the currency configuration
Read a JSON file that defines the currency system. This will include:

- The name of the currency.
- A list of denominations (in smallest-unit terms, e.g., Knuts or cents).
- Conversion rates (e.g., 1 Galleon = 17 Sickles, 1 Sickle = 29 Knuts)

Info: Be sure to not hardcode the path to the file. Your code should only take in the filename and you should put your configuration files at the default location at the project's root folder.
To confirm your CWD (Current Working Directory), you can use:

```
System.out.println("Looking in: " + Paths.get("").toAbsolutePath())
```

You should also make sure not to use any extra libraries like org.json. For this exercise, you can be sure that the configurations follow the given format exactly.

Here are two examples below:

```
{
  "currencyName": "Wizarding Coin",
  "baseUnit": "Knuts",
  "denominations": [
```

```
    { "name": "Galleons", "value": 493 },
    { "name": "Sickles", "value": 29 },
    { "name": "Knuts", "value": 1 }
  ]
}
```

```
{
  "currencyName": "US Dollar",
  "baseUnit": "Cents",
  "denominations": [
    { "name": "Dollars", "value": 100 },
    { "name": "Quarters", "value": 25 },
    { "name": "Dimes", "value": 10 },
    { "name": "Nickels", "value": 5 },
    { "name": "Pennies", "value": 1 }
  ]
}
```

Step 3: Change calculation

- Convert the cost and payment amounts into the smallest unit (e.g., cents or knuts).
- Compute the change due.
- If the amount paid is less than the cost, show an error and exit.

Step 4:
Use division and modulo operators to break the change into denominations, favoring the largest denominations first. This strategy is a type of algorithm called a *greedy algorithm*, which builds a solution step by step by always choosing the largest or best available option at each stage, aiming for an efficient overall result.

Step 5: Output the total change amount, along with a clear, line-by-line breakdown of the number of each currency unit needed to make the change. Use the smallest unit for the full amount of change, plus the name of the base unit from the JSON file.

An example based on the wizarding currency.

```
Change to return: 172 Knuts
Galleons: 0
Sickles: 5
Knuts: 27
```

Another example where the change is being returned in the U.S. currency.

```
Change to return: 172 Pennies
Dollars: 1
Quarters: 2
Dimes: 2
Nickels: 0
Pennies: 2
```

1.18 Ship navigation system

Goals:

- Learning integer wrap-around using modulo operators (in Java: Math.floorMod())
- Experience with input and outputs

Your crew is looking to modernize their boat steering system to handle inputs from an autopilot helm. You're tasked with handling the **angle wrap-around** when turning left or right.
The system must keep the boat's heading normalized between 0° and 359° at all times!

The angle is measured in its azimuth from North, which is the horizontal angle measured clockwise.

- North (N): 0° = 360°
- East (E): 90°
- South (S): 180°
- West (W): 270°

To test your system, you're asked to write a Java program that takes an input between -359° and +359°, adds the input to the current angle of the boat, and outputs the boat's new angle to the console. The boat starts off by heading North. The program is supposed to then keep this new angle of the boat and allow the user to add another steering input. This goes on until the user closes the program or enters "quit". Reject non-integer inputs and values outside [-359,359] with error messages.

Your output should also include the closest cardinal direction (North, East, South or West). If an angle is exactly between two cardinal directions you should favor the previous direction going clockwise, meaning the one that would be closer to your current angle reduced by 1°.

Angle Range	Cardinal Direction
[316°, 45°]	North
[46°, 135°]	East
[136°, 225°]	South
[226°, 315°]	West

The expected console output after every input (and initially) is:

```
"Current direction: <current-angle>°.
Closest cardinal direction: <closest-cardinal-direction>.
Please input a new steering angle as an integer between -359 and 359:"
```

Sample Output
(Do not output the ">> User Input:" Lines)

```
Current direction: 0°.
Closest cardinal direction: North.
Please input a new steering angle as an integer between -359 and 359:
```

>> User Input: -55

```
Current direction: 305°.
Closest cardinal direction: West.
Please input a new steering angle as an integer between -359 and 359:
```

>> User Input: 100

```
Current direction: 45°.
Closest cardinal direction: North.
Please input a new steering angle as an integer between -359 and 359:
```

>> User Input: dog

```
The specified steering angle is not a number.
```

>> User Input: quit

Hints

- Keep in mind that the modulus of negative numbers is equal to the **smallest positive integer** that should be subtracted from the dividend to make it divisible by the divisor. In Java this **does not** work correctly by just using '%' and instead requires 'Math.floorMod()'.

  Example:
  Math.floorMod(-5,100) = 95

  Proof:
  Quotient × Divisor + Remainder = Dividend
  -1 × 100 + 95 = -5

- To make the CLI (Command line interface) application stateful, you should use \`while\` loops. To get user input from \`System.in\` you can use \`java.util.Scanner\`.

By the way: Integer wrap-around is highly relevant in a data structure called circular buffer! It allows for efficient continuous use of memory without needing to move data around.

1.19 Finding Armstrong Numbers

Write a program that produces the set of Armstrong Numbers within a specified range. Armstrong Numbers — numbers so cool *they literally add up to themselves*. An Armstrong Number is a number that equals the sum of its own digits, each raised to the power of the number of digits. Think of it as a number doing some serious self-reflection and saying, “Yep, I still add up!”.

For example:

- 153 is a 3-digit number.
- So, we raise each digit to the 3rd power:
  $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$.
- That checks out — 153 is an Armstrong Number

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. If something’s off, prompt the user to enter valid numbers 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. Also include semantic errors: 2^{nd} argument number is larger than 1^{st} argument, negative numbers
3. List all the Armstrong Numbers within the specified range (inclusive of lower and upper bounds).

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

Sample Input/Outputs

Example-1:
java cs250.exam.ArmstrongNumber 100 b

The specified upper range (“b”) is not a number.

Example-2:
java cs250.ArmstrongNumber 100 1000

The number range that was provided [100 - 1000] is valid.
The Armstrong Numbers in this range are
153
370
371
407

By the way: While a direct application in Mathematics hasn't been found for Armstrong Numbers yet, they're a numbers puzzle that proves to be highly similar to some hardware-level programming concepts. Extracting and manipulating digits from a number mirrors what processors use in bit-wise manipulation for isolating bits and bit shifts. You might also find that this problem invites to think about optimizations via caching, as the most intuitive methods of calculating Armstrong Numbers over a large range are extremely inefficient.

1.20 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.

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)

By the way: The instructions don't include any need for optimizations, however implementing a testing algorithm for prime numbers from scratch makes it painfully obvious how much optimization can be done. Sieving processes for efficient primality testing algorithms are not just used in cryptography. They often leverage cache-aware memory access patterns that are highly relevant, such as in database indexing optimizations. While the problem itself is mathematical, implementing a memory-efficient and secure algorithm for it requires knowledge of systems engineering.

2 What to Submit

Use the CS250 *Canvas* to submit a single .zip file to the appropriate programming exercise.

3 Grading

The assignments must compile and function correctly on machines in the CSB-120 Lab. Assignments that work on your laptop on your particular flavor of Linux, but not on the Lab machines are considered unacceptable.

You are required to **work alone** on this assignment.

4 Late Policy

Please check the class policy on submitting late assignments. You are allowed to submit assignments up to 2 days with a per-day deduction of 7.5%.

5 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. Changes to the first public release are made to clarify the assignment; the spirit or the crux of the assignment will not change.