

Programming Assignment 0: Ciphers

Background

Cryptography (not to be confused with cryptocurrency and blockchain) is a branch of Computer Science and Mathematics concerned with turning input messages (plaintexts) into encrypted ones (ciphertexts) for the purpose of discreet transfer past adversaries. The most modern and secure of these protocols are heavily influenced by advanced mathematical concepts and are proven to leak **no** information about the plaintext. As the Internet itself consists of sending messages through other potentially malicious devices to reach an endpoint, this feature is crucial! Without it, much of the Internet we take for granted would be impossible to implement safely (giving credit card info to retailers, authenticating senders, secure messaging, etc.) as anyone could gather and misuse anyone else's private information.

In this assignment, you'll be required to implement a number of *classical ciphers* making use of your knowledge of abstract classes and inheritance to reduce redundancy whenever possible. Once completed, you should be able to encode information past the point of any human being able to easily determine what the input plaintext was!



The course staff would like to reinforce a message commonly said by the security and privacy community: **"Never roll your own crypto"**. In other words, do not use this assignment in any future applications where you'd like to encrypt some confidential user information. Classical ciphers are known to be remarkably old and weak against the capabilities of modern computation and thus anything encrypted with them should not be considered secure.

Characters in Java

In this assignment, a potentially important note is that behind-the-scenes Java assigns each character an integer value. (e.g. 'A' is 65, 'a' is 97, and so on). This mapping is defined by the *ASCII* (the American Standard Code for Information Interchange) standard, and can be seen in the following ASCII table:

Because Java has this inherent mapping, we are able to perform the exact same operations on characters as we can on integers. This includes addition (e.g. `'A' + 'B' => 131`), subtraction (e.g. `'B' - 'A' => 1`), and boolean operations (e.g. `'A' < 'B' => true`). We can also easily convert between the integer and character representations by casting (e.g. `(int)('A') => 65` or `(char)(66) = 'B'`).

Getting Started

Download starter code:



Breaking It Down

We've crafted a series of sequential development slides, each guiding you through a specific part of the assignment to eventually build up to our final program. This step-by-step approach is designed to make the learning process more manageable and less daunting. We recommend taking notes as you go through each of the slides as well.

Our Recommendation

We strongly recommend using the sequential development slides, especially for this challenging assignment. It's a step-by-step journey that breaks down the complexity into digestible parts that will hopefully make it a smoother learning experience! However, you do not *have* to work in the order given.

Full Specification

The next slide is the **Full Specification** detailing the entire spec of the assignment. Each developmental slide will also provide the relevant sections of the specification to help in completing the respective slide. We will build up towards the final **Ciphers** slide, where you will see all your hard work come together to complete the full assignment!



WARNING: We've noticed that a majority of students' difficulties with this assignment come from not fully understanding what the spec is asking them to do. **Please make sure that you read the description for a cipher closely before attempting to implement it.** If you have any questions about what the spec is asking, please ask for clarification on Ed!

Full Specification

System Structure

We will represent ciphers with following provided abstract class. You may modify the constants of this class to help with debugging your implementations (we recommend starting with a smaller range like `A - G`). Expand to see the default `Cipher.java` file

► Expand

Required Operations

You must implement the following encryption schemes in this assignment. **You should not create any additional classes beyond the ones listed.** Note that the following descriptions often refer to the "encodable/encryptable range," which is defined by the `Cipher.MIN_CHAR` (lowest value in the range), `Cipher.MAX_CHAR` (highest value in the range), and `Cipher.TOTAL_CHARS` (total number of characters within the range) constants within `Cipher.java`



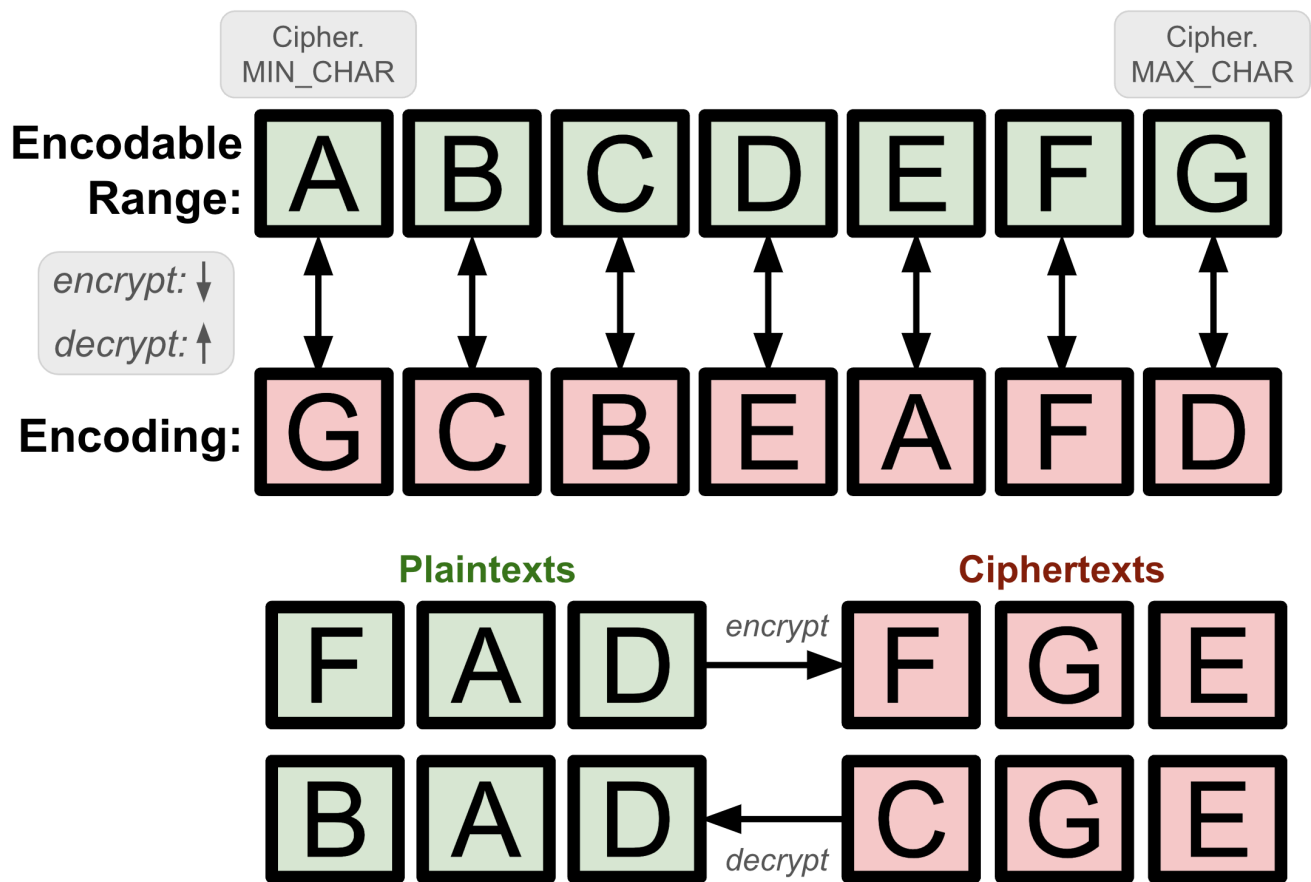
HINT: Check out the "Swap: Example Ciphers" slide for an example implementation of a Cipher!

Substitution.java

The Substitution Cipher is likely the most commonly known encryption algorithm. It consists of assigning each input character a unique output character, ideally one that differs from the original, and replacing all characters from the input with the output equivalent when encrypting (and vice-versa when decrypting).

In our implementation, this mapping between input and output will be provided via a `encoding` string. The `encoding` will represent the output characters corresponding to the input character at the same relative position within the overall range of encodable characters (defined by `Cipher.MIN_CHAR` and `Cipher.MAX_CHAR`). To picture this, we can vertically align this `encoding` string with the encodable range and look at the corresponding columns to see the appropriate character mappings.

Here is an example:



In this example, our encodable range are the letters "ABCDEFG". In code, we represent this as all of the characters between `Cipher.MIN_CHAR` and `Cipher.MAX_CHAR`. We line this up with our given encoding String, which in this case, is "GCBEAFD" such that "ABCDEFG" is directly on top of "GCBEAFD". This means that the letter A will be encrypted to the letter G, the letter B encrypts to the letter C, the letter C encrypts to the letter B, the letter D encrypts to the letter E, the letter E encrypts to the letter A, the letter F encrypts to the letter F, and the letter G encrypts to the letter D.

To decrypt, we would go in the opposite direction. Therefore, the letter G would be decrypted to the letter A, the letter C decrypts to the letter B, the letter B decrypts to the letter C, the letter E decrypts to the letter D, the letter A decrypts to the letter E, the letter F decrypts to the letter F, and the letter D decrypts to the letter G.

Given the encoding string above, the plaintext "FAD" would be encrypted into "FGE" and the ciphertext "CGE" decrypts into the plaintext "BAD".



HINT: Notice what really matters here is the position of each character in the encodable range, and the character at the corresponding location in the encoding String. What are some useful methods or concepts that can help you map from one character to another?

Required Behavior:

Substitution should extend the provided `Cipher.java` **OR** a subclass of `Cipher.java` and contain the following constructors / additional instance method:

```
public Substitution()
```

- Constructs a new Substitution Cipher with an empty encoding.

```
public Substitution(String encoding)
```

- Constructs a new Substitution Cipher with the provided encoding.
- Should throw an `IllegalArgumentException` if the given `encoding` meets any of the following cases:
 - Is null
 - The length of the encoding doesn't match the number of characters within our Cipher's encodable range (`Cipher.TOTAL_CHARS`)
 - Contains a duplicate character
 - Any individual character falls outside the encodable range (`< Cipher.MIN_CHAR` or `> Cipher.MAX_CHAR`).
 - Consider `isCharInRange()`!

```
public void setEncoding(String encoding)
```

- Updates the encoding for this Substitution Cipher.
- Should throw an `IllegalArgumentException` if the given `encoding` meets any of the following cases:
 - Is null
 - The length of the encoding doesn't match the number of characters within our Cipher's encodable range (`Cipher.TOTAL_CHARS`)
 - Contains a duplicate character
 - Any individual character falls outside the encodable range (`< Cipher.MIN_CHAR` or `> Cipher.MAX_CHAR`).
 - Consider `isCharInRange()`!

Since we're allowing clients to set an encoding after construction (via the no-argument constructor and the `setEncoding` method), **encrypt / decrypt should throw an `IllegalStateException` if the encoding was never set:**

```
Substitution a = new Substitution();  
a.encrypt("BAD");    // Should throw an IllegalStateException since the encoding was never set!
```

CaesarShift.java

This encryption scheme draws inspiration from the Substitution Cipher, except it involves shifting all encodable characters to the left by some provided shift amount.

Applying the CaesarShift Cipher is defined as replacing each input character with the corresponding character in `encoding` at the same relative position. This `encoding` should be created by moving all characters within the range to the left `shift` times, moving the value at the front to the end each time.

Similarly, inverting the CaesarShift Cipher is defined as replacing each input character with the corresponding character in the encodable range at the same relative position within `encoding`. This `encoding` should be created by moving all characters within the range to the left `shift` times, moving the value at the front to the end each time.

For example, if the shift is 1 and our encodable range was `A` to `G`, `A` would be replaced with `B`. Additionally, if any character maps to a value greater than the maximum encryptable character (`G` in this case), the replacement character can be found by looping back around to the front of the encodable range. In this example, `G` would map to `A`. If the shift was 3 and our encodable range was `A` to `G`, then `A` would map to `D`, `B` would map to `E`, and so on and so forth, with letters `E-G` wrapping around to map to `A-C` respectively.



NOTE: This mapping from an input character `c` and it's encrypted output `o` after a shift `shift` can be seen in the following expression:

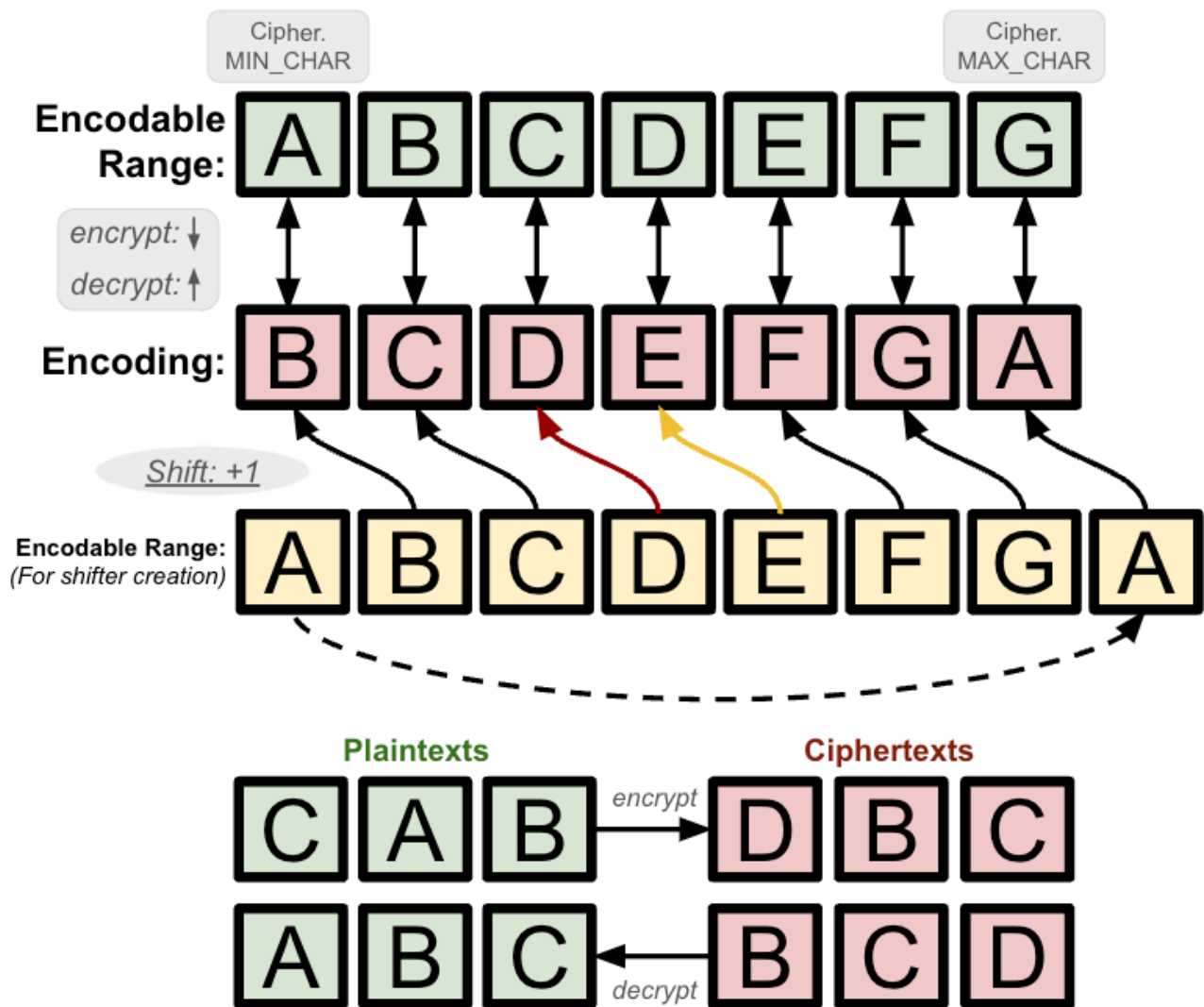
```
shift %= Cipher.TOTAL_CHARS  
o = (char)(Cipher.MIN_CHAR + (c + shift - Cipher.MIN_CHAR) % Cipher.TOTAL_CHARS)
```

Where we add `shift` to `c`, get the displacement of the result by subtracting `Cipher.MIN_CHAR`, mod it by `Cipher.TOTAL_CHARS` in the event that we go past the maximum encryptable character, and re-add the new displacement to `Cipher.MIN_CHAR` to get the encrypted result. Similarly, we can define the inverse expression:

```
shift %= Cipher.TOTAL_CHARS  
c = (char)(Cipher.MIN_CHAR + (o - shift - Cipher.MIN_CHAR + Cipher.TOTAL_CHARS) %  
Cipher.TOTAL_CHARS)
```

Where we remove `shift` from `o`, get the displacement of the result by subtracting `Cipher.MIN_CHAR`, add `Cipher.TOTAL_CHARS` in the event that the displacement is negative, mod it by `Cipher.TOTAL_CHARS` to re-map large displacements to valid ones, and re-add the new displacement to `Cipher.MIN_CHAR` to get the decrypted result.

Consider the following diagram for a visual explanation:



In this example, our encodable range are the letters "ABCDEFGH" (where A is `Cipher.MIN_CHAR` and G is `Cipher.MAX_CHAR`). To create the encoding, we move the character at the front of the encodable range to the end (and in doing so shift all other characters to the left). As the shift value above is just one, this process is repeated one time. If the shift value was two, we'd do it twice.

With a shift value of 1, our encoding String becomes "BCDEFGA". Notice how the first letter, A, was moved from the front to the back. Similarly to `Substitution`, the mapping of letters is made more clear by placing "ABCDEFGH" on top of "BCDEFGA", such that A is encrypted to B, B is encrypted to C, C is encrypted to D, D is encrypted to E, E is encrypted to F, F is encrypted to G, and G is encrypted to A. We go the opposite direction for decryption, so B is decrypted to A, C is decrypted to B, D is decrypted to C, E is decrypted to D, F is decrypted to E, G is decrypted to F, and A is decrypted to G.



HINT: What data structure would help with this process of removing from the front and adding to the back?



HINT: Notice that after creating the encoding String, encrypting and decrypting a given input behaves *exactly*

the same as `Substitution`! Keeping in mind our recently learned concepts, what can we say about the relationship between `CaesarShift` and `Substitution`? How can we take advantage of those similarities to *reduce redundancy* between these two classes?

After creating the encoding string, the process of encrypting / decrypting should exactly match that of the `Substitution` cipher (replace each character of the input with the character at the same relative position in the encoding string for encrypting, or vice-versa for decrypting).

Required Behavior:

`CaesarShift` should extend the provided `Cipher.java` **OR** a subclass of `Cipher.java` and contain the following constructor:

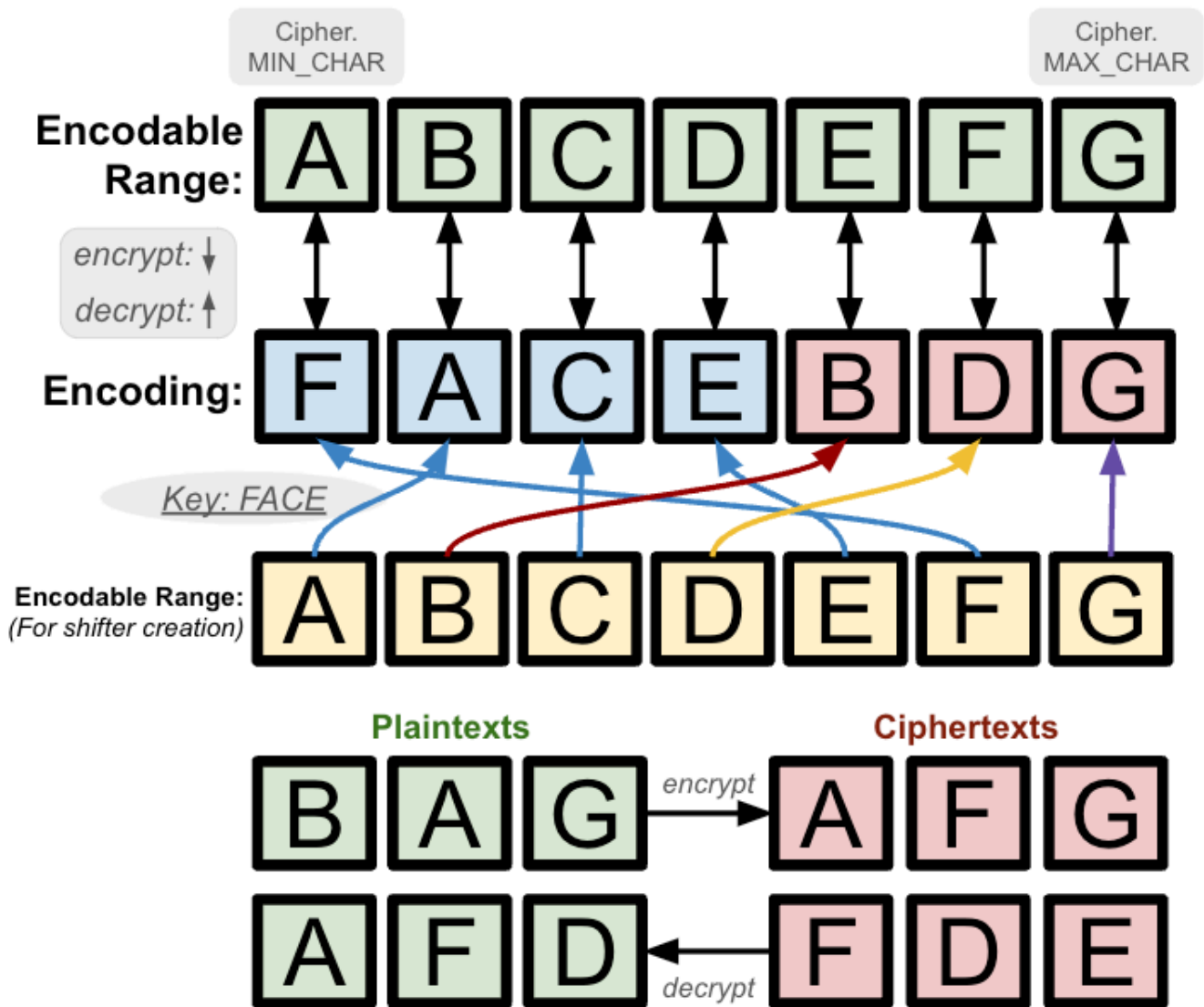
```
public CaesarShift(int shift)
```

- Constructs a new `CaesarShift` with the provided shift value
- An `IllegalArgumentException` should be thrown in the case that `shift < 0`

[CaesarKey.java](#)

The `CaesarKey` scheme builds off of the base `Substitution Cipher`. This one involves placing a key at the front of the substitution, with the rest of the alphabet following normally (minus the characters included in the key). This means that the first character in our encodable range (`(char)(Cipher.MIN_CHAR)`) would be replaced by the first character within the key. The second character in the encodable range (`(char)(Cipher.MIN_CHAR + 1)`) would be replaced by the second character within the key. This process would repeat until there are no more key characters, in which case the replacing value would instead be the next unused character within the encodable range.

Consider the following diagram for a visual explanation:



To build the encoding String, notice that we took the `key` and placed it in the beginning. Then, we go through the characters in our encodable range and add them if they are not already in the encoding string. In the following example, note that the encoding string starts with "FACE" (the key) and then is followed by the encodable range in its original order, excluding characters 'F', 'A', 'C' and 'E' as they're already in the encoding. This results in the encoding String "FACEBDG".

After creating the encoding string, the process of encrypting and decrypting should exactly match that of the Substitution cipher. We see that A is encrypted to F, B is encrypted to A, C is encrypted to C, D is encrypted to E, E is encrypted to B, F is encrypted to D, and G is encrypted to G. We invert this process to decrypt so that F decrypts to A, A decrypts to B, C decrypts to C, E decrypts to D, B decrypts to E, D decrypts to F, and G decrypts to G.



HINT: Notice that after creating the encoding String, encrypting and decrypting a given input behaves *exactly* the same as Substitution! Keeping in mind our recently learned concepts, **what can we say about the relationship between the CaesarKey and Substitution ciphers?** How can we take advantage of those similarities to *reduce redundancy* between these two classes?

At this point, we recommend taking a closer look at the provided example if you haven't done so already!

Required Behavior

CaesarKey should extend the provided `Cipher.java` **OR** a subclass of `Cipher.java` and contain the following constructor:

```
public CaesarKey(String key)
```

- Constructs a new CaesarKey with the provided key value
- This constructor should throw an `IllegalArgumentException` if the given `key` meets any of the following cases:
 - Is null
 - Contains a duplicate character
 - Any individual character falls outside the encodable range (`< Cipher.MIN_CHAR` or `> Cipher.MAX_CHAR`).
 - Consider `isCharInRange()`!



WARNING: We are *requiring* that you do not override `encrypt` / `decrypt` methods within `CaesarKey`. These should be inherited from a superclass.

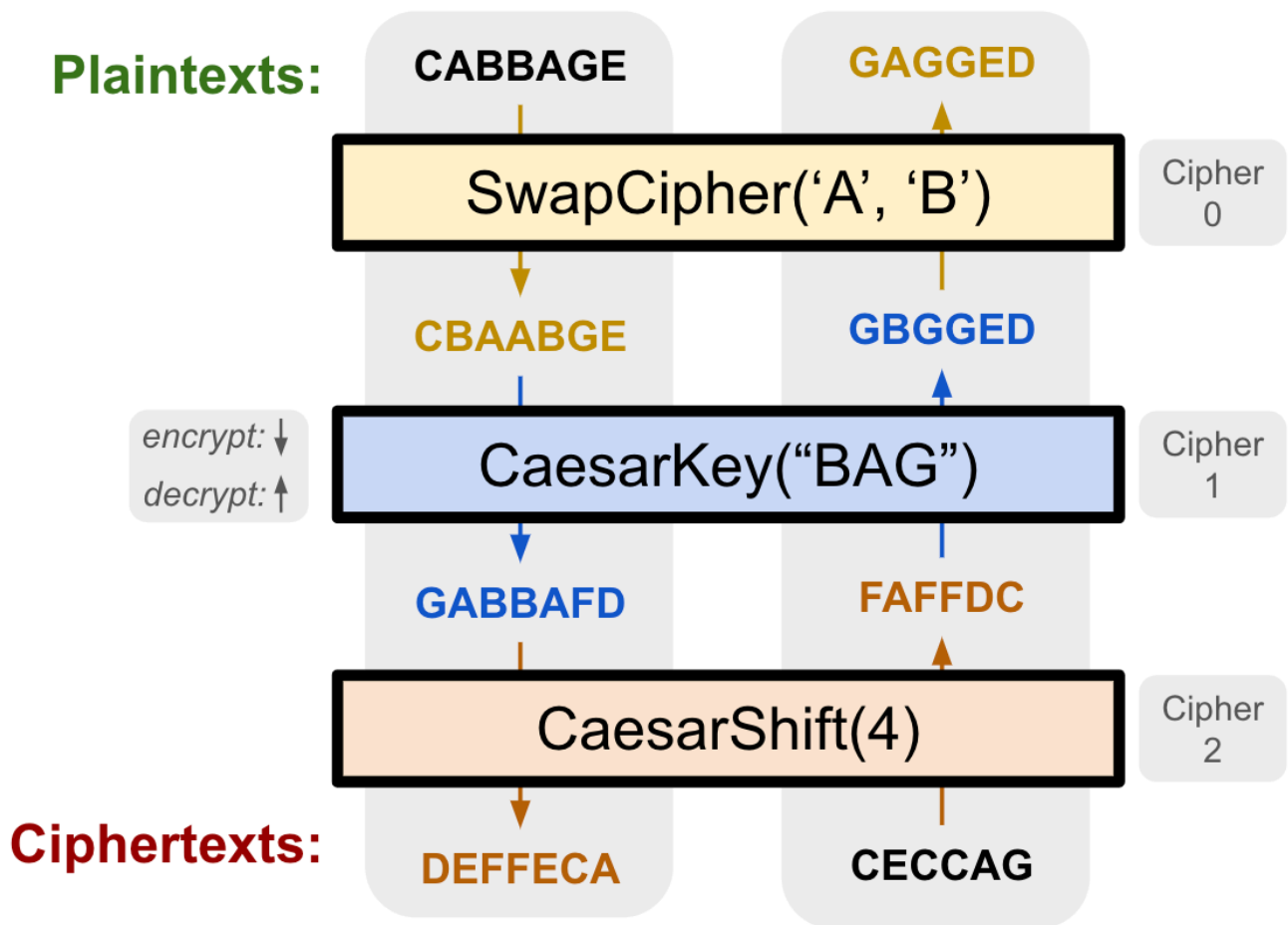
MultiCipher.java

The above ciphers are interesting, but on their own they're pretty solvable. A more complicated approach would be to chain these ciphers together to confuse any possible adversaries! This can be accomplished by passing the original input through a list of ciphers one at a time, using the previous cipher's output as the input to the next. Repeating this through the entire list results in the final encrypted string. Decrypting would then involve the opposite of this: starting with the last cipher and working backward through the cipher list until the plaintext is revealed.

Below is a diagram of these processes, passing inputs through each layer of the cipher list. Consider the following diagram demonstrating the process of encrypting/decrypting a MultiCipher consisting of 3 internal ciphers: a CaesarShift of 4, a CaesarKey with key "BAG", and a CaesarShift of 8.



NOTE: In this example, the encodable range is A - G



On the left in the above example, we start with the plaintext: `CABBAGE` hoping to encrypt it. Encrypting this through the first layer (a `SwapCipher` with arguments 'A' and 'B') results in the intermediary encrypted message `CBAABGE`. This intermediary value is then used as input to the next layer (a `CaesarKey` with key "BAG") resulting in the second intermediary encrypted message `GABBAFD`. This process is repeated one last time, resulting in the final ciphertext of `DEFFECA`.

On the right in the above example, we start at the ciphertext: `CECCAG` hoping to decrypt it. Decrypting this through the last layer (a `CaesarShift` of 4) results in the intermediary still-encrypted message `FAFFDC`. This intermediary value is then used as input to the next layer (a `CaesarKey` with key "BAG") resulting in the second intermediary still-encrypted message `GBGGED`. This process is repeated one last time, resulting in the final plaintext of `GAGGED`.

This is what you'll be implementing in this class: given a list of ciphers, apply them in order to encrypt or in reverse order to decrypt a given message.



NOTE: Unlike in `CaesarKey`, you *may* override `encrypt` and `decrypt` if you think it is necessary.

Required Behavior:

MultiCipher should extend the provided `Cipher.java` **OR** a subclass of `Cipher.java` and contain the following constructor:

```
public MultiCipher(List<Cipher> ciphers)
```

- Constructs a new MultiCipher with the provided List of Ciphers
 - You may assume that any Cipher in the list is non-null and calling `encrypt / decrypt` will not throw an `IllegalStateException`.
- Should throw an `IllegalArgumentException` if the given list is null

Use Your Ciphers!

Now that you're done, set `Cipher.MIN_CHAR = (int)(' ')` and `Cipher.MAX_CHAR = (int)('}')`. Then, using the Client class create a MultiCipher consisting of the following: a `CaesarShift(4)`, a `CaesarKey("123")`, a `CaesarShift(12)`, and a `CaesarKey("lemon")`. Decrypt the following!

```
Yysu(zer{vyly xylw("m(!xy (q ywl}ul!)(Oyt(&e"({le$($xq!(!xy { }u qwu($q (ruvenu(tusn&m!ylwJ(E1
```

Once you've figured it out, revert `MIN_CHAR = (int)('A')` and `MAX_CHAR = (int)('Z')` for the testing portion of the assignment

Testing

You are welcome to use the provided `Client.java` to test and debug your cipher implementations. To do so, make sure to change the `CHOSEN_CIPHER` constant to the cipher you're testing before hitting run. You are also encouraged to modify the constants in `Cipher.java` such that a smaller subset of characters are used by your cipher.

You'll be required to finish the 3 unimplemented tests in `Testing.java`: one for `CaesarKey`, one for `CaesarShift`, and one for `MultiCipher`. Follow the steps outlined in the comments within each method for more guidance.



WARNING: We've provided you a test that checks if your `Testing.java` file compiles and no tests fail. It does not check that the appropriate updates were made according to the comments within the file. It is your responsibility to make sure that you're updating the file correctly.

Implementation Guidelines

As always, your code should follow all guidelines in the [Code Quality Guide](#) and [Commenting Guide](#). In particular, pay attention to these requirements and hints:

- Each type of Cipher should be represented by a class that extends the `Cipher` class (or a subclass of `Cipher`). You should **not** modify `Cipher`. **You should utilize inheritance** to

capture common behavior among similar cipher types and eliminate as much redundancy between classes as possible.

- **You should not create any additional classes beyond the ones listed.**
- In general, you should not need many (if any) modifications to your superclass to implement a subclass. Your subclass should be built off of your superclass, not the other way around.
- You should avoid unnecessary reprocessing in your code when possible. For example, rather than recomputing a result whenever it is needed, write your code in such a way that you compute the result only once, and save the result to use later.
- You should make all of your fields private and you should reduce the number of fields only to those that are necessary for solving the problem.
- Each of your fields should be initialized inside of your constructor(s).
- You should comment your code following the [Commenting Guide](#). You should write comments with basic info (a header comment at the top of your file), a class comment for every class, and a comment for every method other than main.
 - Make sure to avoid including *implementation details* in your comments. In particular, for your object class, a *client* should be able to understand how to use your object effectively by only reading your class and method comments, but your comments should maintain *abstraction* by avoiding implementation details.

Spec Walkthrough

An error occurred.

Try watching this [video on www.youtube.com](https://www.youtube.com), or enable JavaScript if it is disabled in your browser.

Reflection

For this week's reflection, we would like everyone to watch and engage with the following video.

Step 1: Watch [End To End Encryption \(E2EE\) - Computerphile](#) (8m 12s)

An error occurred.

Try watching this video on www.youtube.com, or enable JavaScript if it is disabled in your browser.

(<https://youtu.be/jkV1KEJGKRA>)

Question 1

Step 2: The next 3 questions will require you reflect on privacy, encryption and a few ethical complications.

In 2015, there was a rather [infamous court case](#) resulting from the US government mandating Apple extract encrypted data from criminals' devices. These included devices Apple had no ability to crack with their current tooling; thus, Apple was ordered to develop new software that would enable this decryption to occur.

The most well-known example involved unlocking the phone of a terrorist involved in a shooting that killed 14 and injured 22. The government hoped that unlocking the phone would prevent future terrorist attacks. With this context, do you believe this to be a fair request? Why or why not?

For full credit, you should provide a stance, as well as explain your reasoning.

No response

Question 2

Now, apply what was mentioned in the video - that there's no such thing as a safe backdoor - to this situation. Alternatively stated, should Apple create cracking software (and prove its existence) it's possible a non-government entity could obtain and misuse it.

Does this perspective change your answer to the previous question and why? How would you feel if software capable of decrypting any and all private information on your devices existed?

For full credit, you should answer both questions and provide reasoning.

No response

Question 3

Having answered the above questions, do you believe it's necessary to sacrifice privacy for the "greater good" / safety of modern society? Why or why not?

For full credit, you should provide a stance, as well as explain your reasoning.

No response

Question 4

Step 3: The following questions will ask that you practice **metacognition** to reflect on the topics covered on this assignment and your experience completing it. For each question, focus on your plan and/or process for working through the assignment along with the CS concepts. Think about things like how you organized your working time, what sorts of things tended to go wrong, and how you dealt with those errors or mistakes.

Describe the inheritance hierarchy you chose to create. Which classes extended which other classes? Why did you make those choices?

No response

Question 5

Describe how you went about testing your implementation. What specific situations and/or test cases did you consider? Why were those cases important?

No response

Question 6

What skills did you learn and/or practice with working on this assignment?

No response

Question 7

What did you struggle with most on this assignment?

No response

Question 8

What questions do you still have about the concepts and skills you used in this assignment?

No response

Question 9

About how long (in hours) did you spend on this assignment? (Feel free to estimate, but try to be close.)

No response

Question 10

Was any part of the specification or requirements unclear? If so, which part(s), how was it unclear, and how could it have been made more clear?

No response

Question 11

[OPTIONAL] Do you have any other feedback, questions, or comments about this assignment?

(Note that we may not be able to respond to questions here, so please post on the message board if you would like a response!)

No response