Creative Project 0: Ciphers

Background

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 0 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 `'A' + 'B' = 131`, subtraction `'B' - 'A' = 1`, and boolean expressions `'A' < 'B' = true`. We can also easily convert between the integer and character representations by casting `(int)('A') = 65` or `(char)(66) = 'B'` .

Getting Started

Download starter code:

C0_Ciphers.zip

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!

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. 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 "Concealment: An Example Cipher" slide for an example implementation of a Cipher!

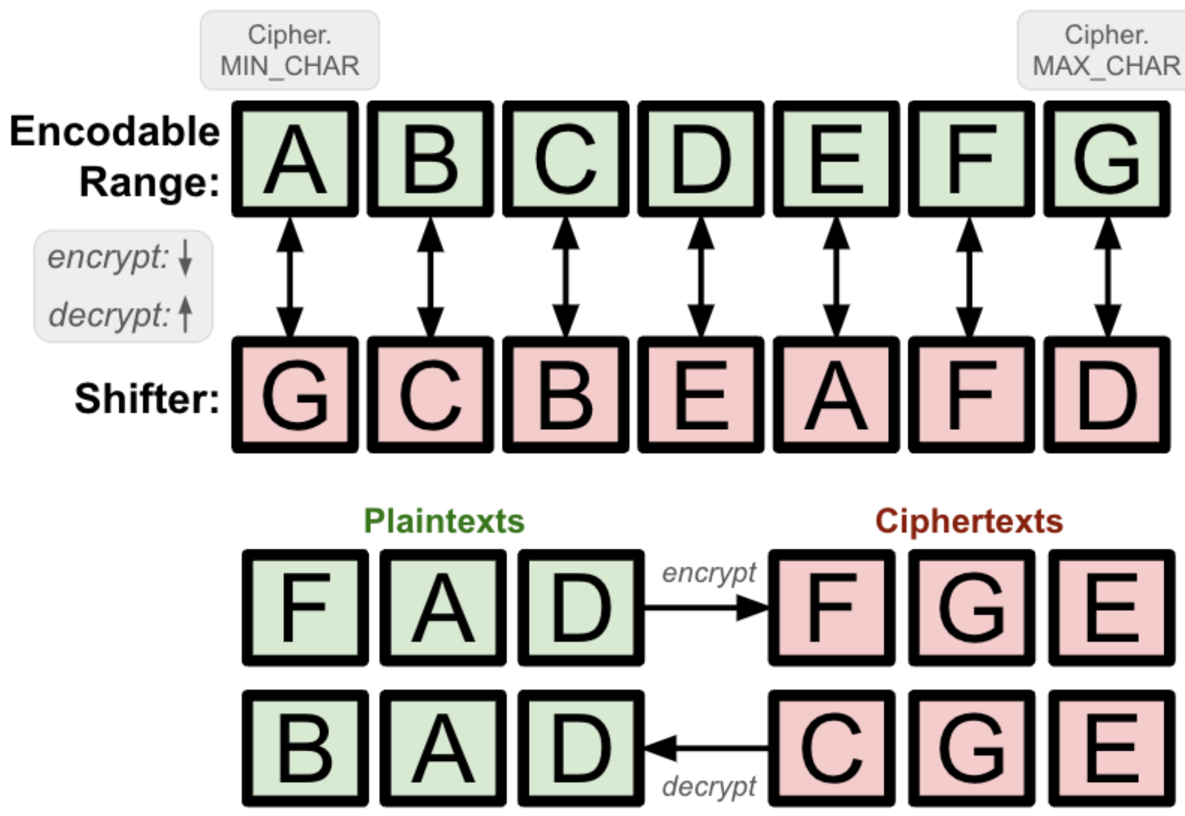
Substitution.java

▼ Expand

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 provided via a `shifter` string. The `shifter` 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 `shifter` 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 change 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 `shifter` String, which in this case, is "GCBEAFD". This means that the letter `A` will be encrypted to the letter `G`, the letter `B` encrypts to the letter `C`, and so on.

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

NOTE: Notice what really matters here is the position of each character in the encodable range, and the character at the corresponding location in the shifter 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 shifter.

```
public Substitution(String shifter)
```

- Constructs a new Substitution Cipher with the provided shifter.
- Should throw an `IllegalArgumentException` if the given `shifter` meets the following cases:
    - The length of the shifter 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`).

```
public void setShifter(String shifter)
```

- Updates the shifter for this Substitution Cipher.
- Should throw an `IllegalArgumentException` if the given `shifter` meets the following cases:
    - The length of the shifter 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`)
      public

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

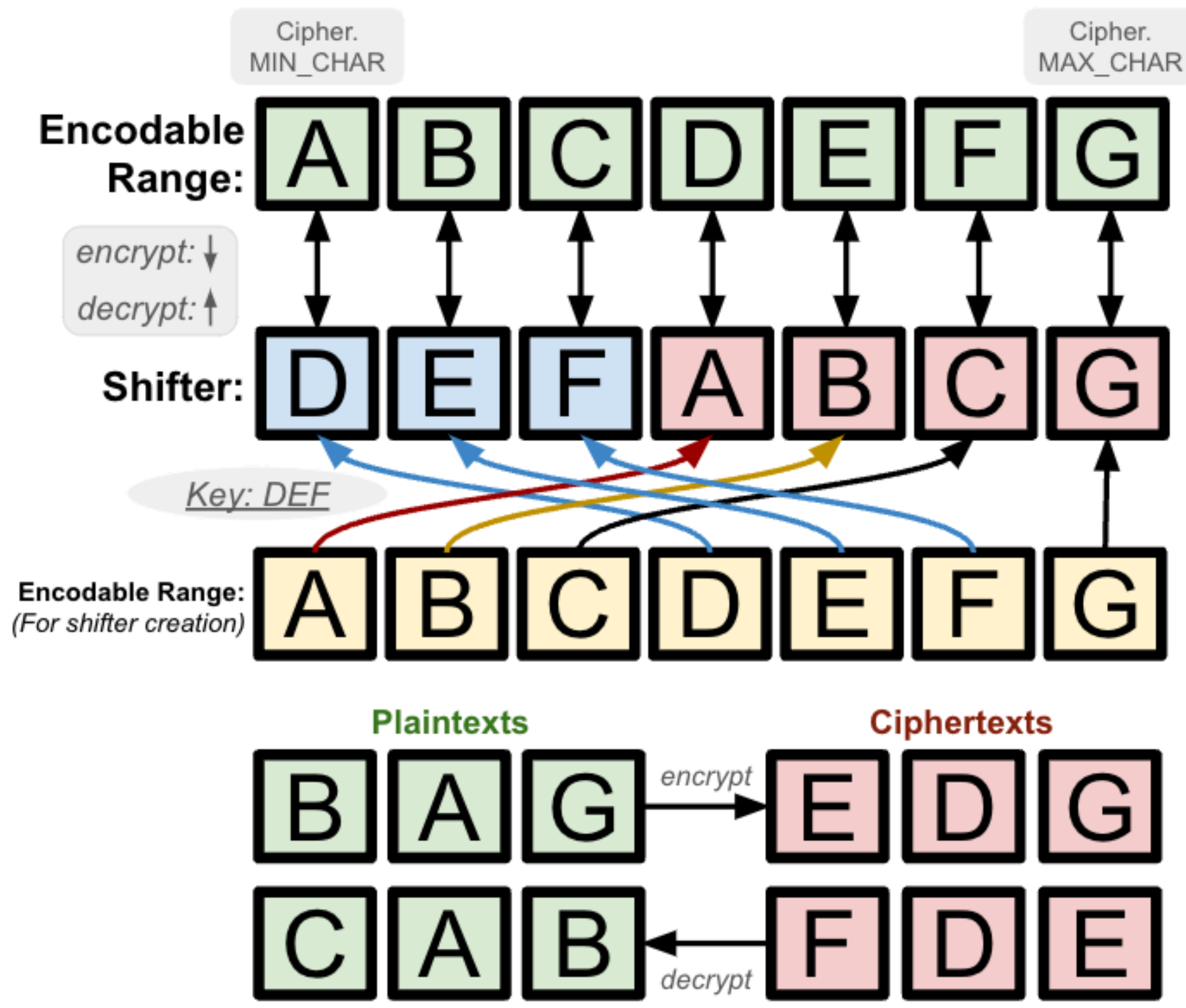
```
Substitution a = new Substitution();
a.encrypt("BAD");    // Should throw an IllegalStateException since the shifter was never set!
```

CaesarKey.java

▼ Expand

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 shifter 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 shifter string. In the following example, note that the shifter string starts with "DEF" (the key) and then is followed by the encodable range in its original order, excluding characters 'D', 'E', and 'F' as they're already in the shifter.

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

Hint: Notice how after creating the shifter 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 key is empty
  - the key contains a character outside our range of valid characters
  - the key contains any duplicate characters

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

CaesarShift.java

▼ Expand

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

Applying the CaesarShift Cipher is defined as replacing each input character with the corresponding character in `shifter` at the same relative position. This `shifter` 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, inversing the CaesarShift Cipher is defined as replacing each input character with the corresponding character in the encodable range at the same relative position within `shifter`. This `shifter` 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, any character `c` would be replaced with `(char)(c + 1)` when encrypting. If shift is two, `c` would be replaced with `(char)(c + 2)`. Importantly, if characters map to a value greater than the maximum encryptable character (think shift=1, `(char)(Cipher.MAX_CHAR) -> (char)(Cipher.MAX_CHAR + 1)`) the replacement character should be found by looping back around to the front of the encodable range (so if shift=1, then `(char)(Cipher.MAX_CHAR)` would *actually* map to `(char)(Cipher.MIN_CHAR)`).

HINT: 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
```

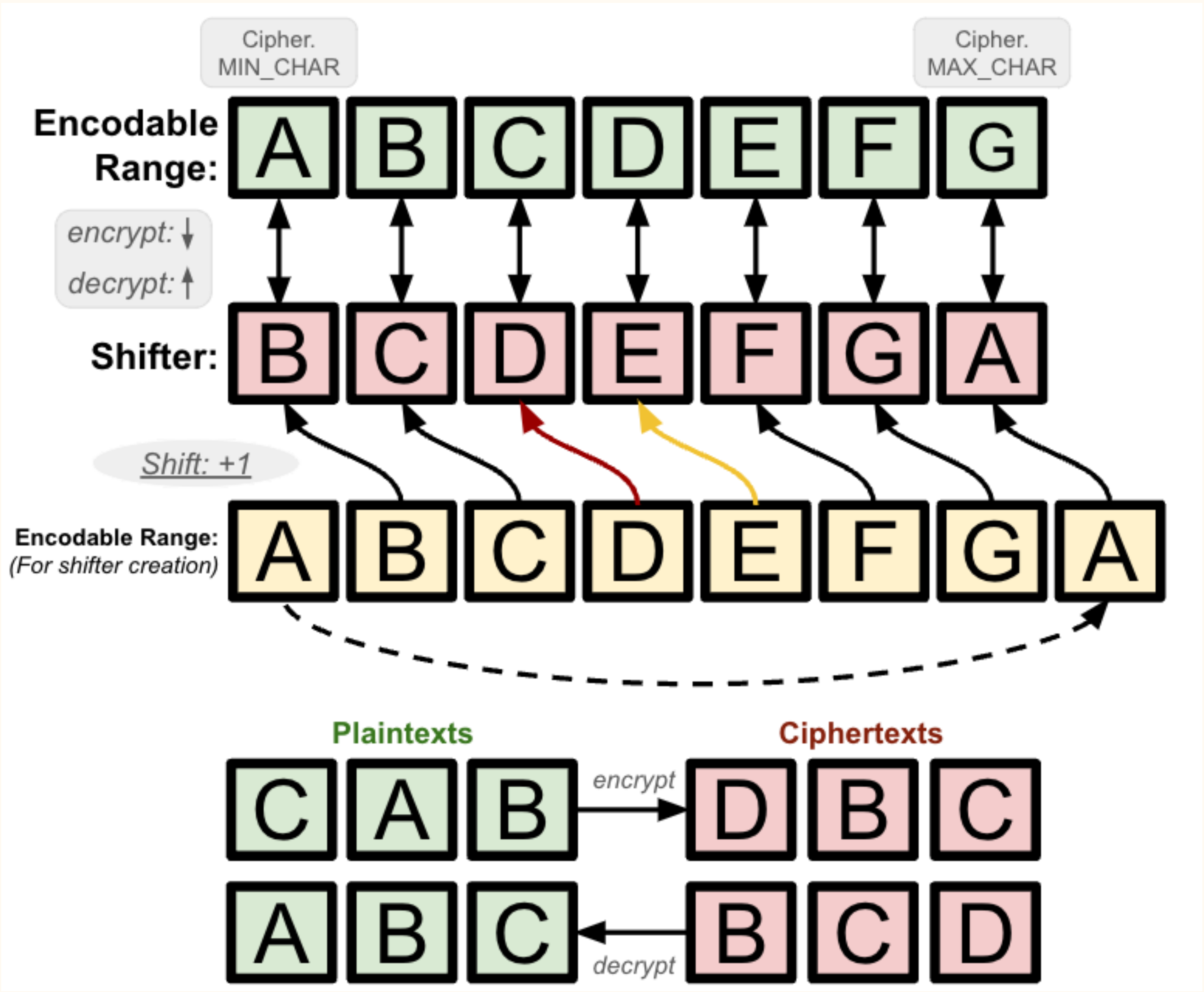
```
o = (char)(Cipher.MIN_CHAR + (c + shift - Cipher.MIN_CHAR) % Cipher.TOTAL_CHARS)
```

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.

An alternative method of approaching this problem can be seen through the following diagram:



Note that this diagram outlines the process of creating shifter in which we physically 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.

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

i **Hint**: Notice how after creating the shifter 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 shifter 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 shifter for encrypting, or vice-versa for decrypting).

Your solution should pick one of the two above approaches to implement such that plaintext characters are shifted in the encodable range by a given amount.

Required Behavior:

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

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

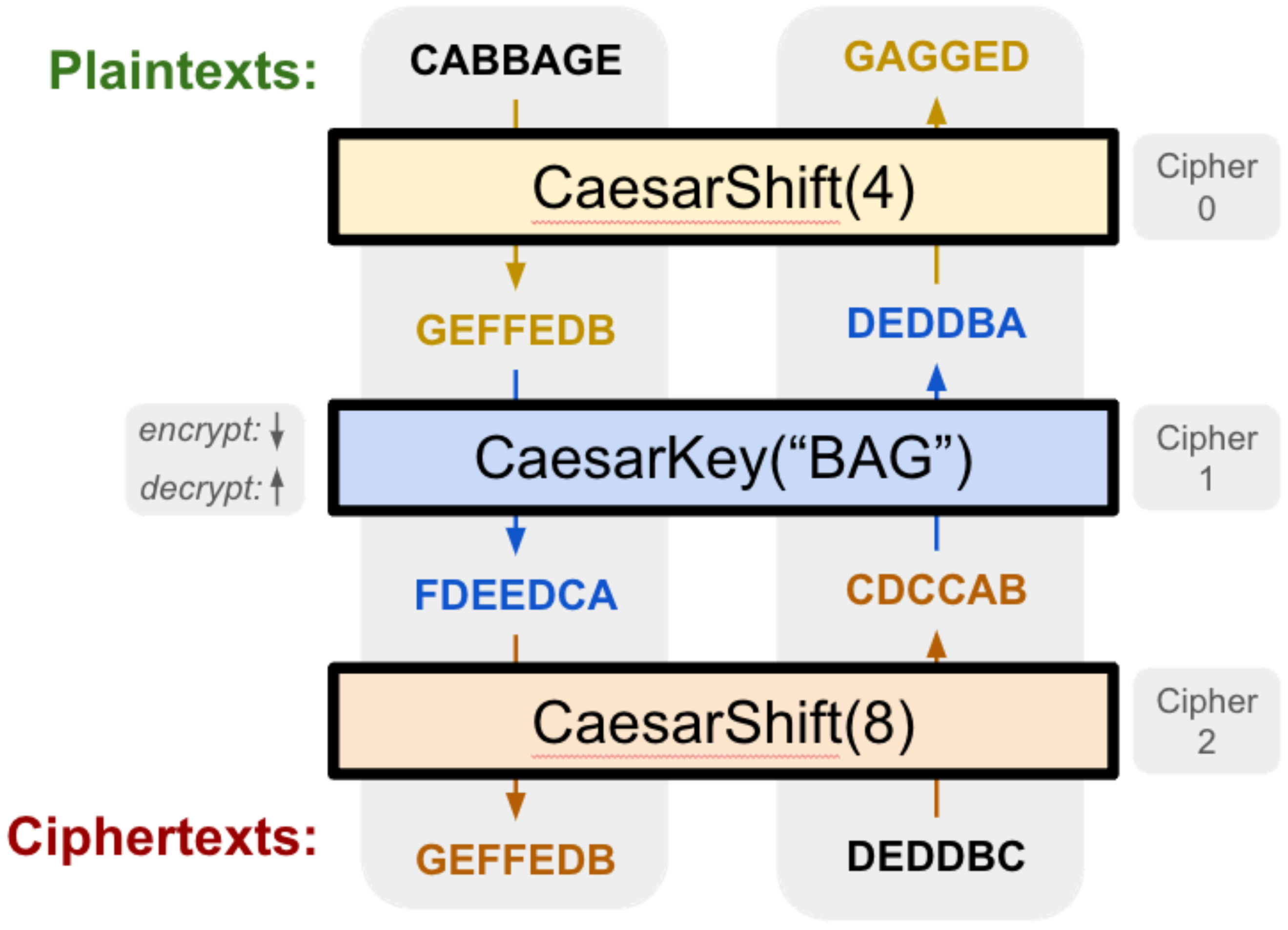
- Constructs a new CaesarShift with the provided shift value
- An `IllegalArgumentException` should be thrown in the case that `shift <= 0`

MultiCipher.java

▼ Expand

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.



On the left in the above example, we start with the plaintext: `CABBAGE` hoping to encrypt it. Encrypting this through the first layer (a CaesarShift of 4) results in the intermediary encrypted message `GEFFEDB`. This intermediary value is then used as input to the next layer (a CaesarKey with key "BAG") resulting in the second intermediary encrypted message `FDEEDCA`. This process is repeated one last time, resulting in the final ciphertext of `GEFFEDB`.

On the right in the above example, we start at the ciphertext: `DEDDBC` hoping to decrypt it. Decrypting this through the last layer (a CaesarShift of 8) results in the intermediary still-encrypted message `CDCCAB`. 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 `DEDDBA`. 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.

Required Behavior:

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

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

- Constructs a new MultiCipher with the provided List of Ciphers
- 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
```

Creative Portion

For the creative portion of this assignment, you'll be implementing another cipher that interests you! Below is the recommended list:

1. SubstitutionRandom
2. Transposition
3. Vigenere
4. Your choice!

You should continue to use concepts taught in class, like inheritance, to reduce redundancy between classes.

1. SubstitutionRandom

▼ Expand

Here, you'll implement another variation of a Substitution Cipher that uses a randomly shuffled shifter string. This initially sounds impossible as if we randomly create the shifter string, how do we possibly decrypt? The answer lies in being able to control a Random object in Java via a seed value. Any two Random objects constructed with the same seed will produce random values in the same order as one another.

Applying the SubstutionRandom Cipher is defined as replacing each input character with the corresponding character in `shifter` at the same relative position. This `shifter` should be created by "randomly" shuffling all characters within the encodable range dependent on a seed value that is then placed somewhere in the ciphertext.
Similarly, inversing the SubstutionRandom Cipher is defined as replacing each input character with the corresponding character in `shifter` at the same relative position. This `shifter` should be created by "randomly" shuffling all characters within the encodable range dependent on a seed

value that is present in the `input` and ignored upon decrypting.

Below is an example:

```java
import java.util.*;
public class Example {
    public static void main(String[] args) {
        int seed = 123;
        Random rand = new Random(seed);
        Random rand2 = new Random(seed);
        System.out.println(rand.nextInt(10) == rand2.nextInt(10));
    }
}
```

Thus, with just the seed value used to create a "random" shifter string, you should be able to recreate it on decrypting (so long as you follow the same steps to do so). Note that this means your implementation must store the seed somewhere in the encrypted message such that it is retrievable on decryption (i.e. front, end, etc.).

✓ **HINT**: In creating this "randomly" shuffled shifter string, we recommend you think of the encodable range as a `List` of all characters able to be encoded by your cipher. Coincidentally, there exists a method that will shuffle the values of a `List` with a given random object called `Collections.shuffle(list, rand)`. We recommend using this approach in your solution.

You should randomly generate a new seed every time you encrypt a message. The length of the seed will be determined by a `digits` parameter provided to the constructor. For example, if `digits` is 4, valid seeds include: 3291, 4039, 6587, 1320, etc.

NOTE: It is your choice if you want to include leading 0's in the number of digits a number has. Alternatively stated, you get to pick whether given 3 digits if the smallest number will be 000 or 100.

After "randomly" creating the shifter string from the given seed value, 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 shifter for encrypting, or vice-versa for decrypting).

Required Behavior:

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

```java
public SubstitutionRandom(int digits)
```

- Constructs a new SubstitutionRandom Cipher with the provided number of digits
- An `IllegalArgumentException` should be thrown if `digits <= 0` or if it is greater than the max number of digits for an integer, which is 9. (Note that `Integer.MAX_VALUE` is

2,147,483,647 which is 10 digits, but larger 10-digit numbers can't be represented, so we subtract one to get 9).

NOTE: If you want a non-magic number way to calculate get this maximum integer width of 9, you can use the following expression:

`(int)(Math.floor(Math.log10(Integer.MAX_VALUE)))`

Additionally, when decrypting, you may assume that only a previously encrypted input is provided on decryption (namely that a seed value will be present at the appropriate location within the `input`).

2. Transposition

▼ Expand

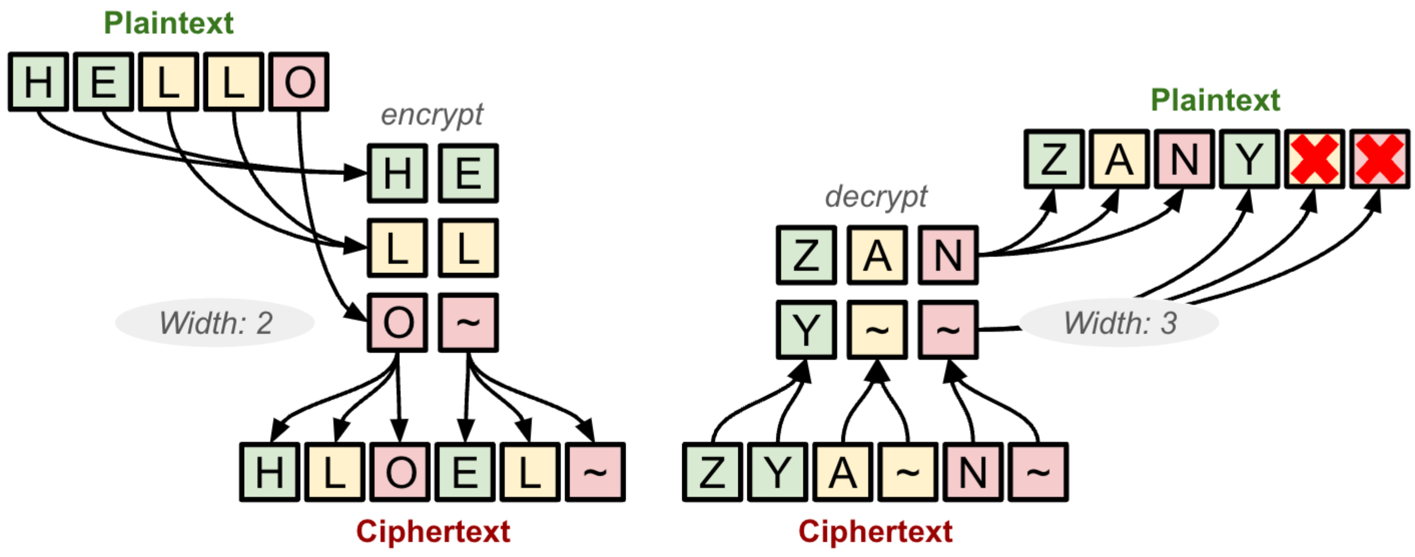
Unlike our previous ciphers, a transposition cipher involves shuffling the positions of characters from the plaintext rather than substituting them with new ones. Most of these involve creating a grid with a certain width, filling it in with an input string, and then traversing the grid in a different way to get the encryption. While there are some fun and interesting traversal options, in this class we'll be implementing the simplest: alternating between row and column.

When encrypting, you'll first create a grid of the desired width (with enough rows to fit the plaintext completely), then fill in character values from the plaintext row-by-row, and finally traverse column-by-column to compute the resulting ciphertext. Any empty spots should be populated with `Cipher.MAX_CHAR + 1`

Inversing the Transposition Cipher is defined as using the `input` to first populate a grid with the appropriate `width` column-by-column, then traversing row-by-row to compute the resulting plaintext. Any spots populated with `Cipher.MAX_CHAR + 1` should be ignored.

It's important to acknowledge that we have no guarantee the length of the input will be an exact multiple of the width of the grid (meaning there might be empty spots left over after filling row-by-row). To counteract this, you should use a character value outside of the encodable range to indicate that something is "empty" - we recommend `Cipher.MAX_CHAR + 1`.

When decrypting, a similar process is followed: first create a grid of the desired width (with enough rows to fit the ciphertext completely), then fill in character values from the ciphertext column-by-column, and finally get the plaintext by traversing the populated grid row-by-row. When decrypting, you should ignore the "empty" characters previously inserted to fill the grid when encrypting. Below is a visual representation of this process:



Again, while alternative traversals are possible, we recommend this approach as it is the easiest to implement (if you choose something different note that your implementation will fall into the "Your choice" category).

Required Behavior:

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

```
public Transposition(int width)
```

- Constructs a new Transposition cipher with the provided grid width value
- An `IllegalArgumentException` should be thrown if `width <= 0` (not possible)

Additionally, when decrypting, you should throw an `IllegalArgumentException` when the length of the `input` is not a multiple of width (`input.length() % width != 0`) as that means it isn't a valid previous encryption.

3. Vigenere

▼ Expand

The Vigenère cipher is a hybrid between the CaesarKey and CaesarShift. It is created with a key that is repeated such that its length matches that of the input plaintext.

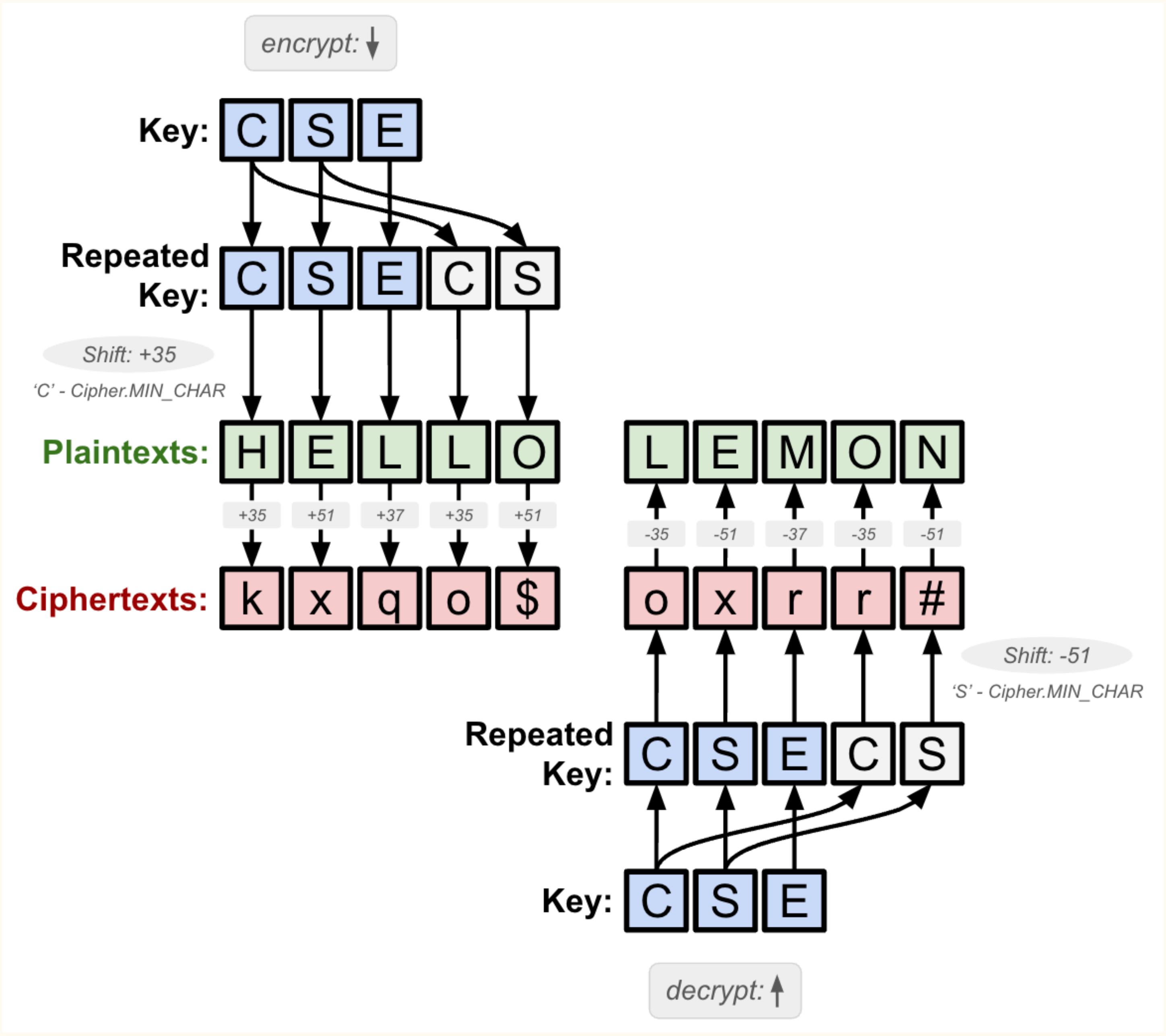
When encrypting, a CaesarShift is applied to each character, where the shift value is determined by the current key character's displacement within the encodable range (i.e. `shift = repeatKey.charAt(i) - Cipher.MIN_CHAR`). If the shifted value falls outside the encodable range (`> Cipher.MAX_CHAR`) it should loop back around to the beginning.

For example, if we encrypt the plaintext is `HELLO` with the key is `CSE`, we first have to repeat the key until it matches the length of the input (`CSECS`), then as the first character is `C` the first shift value is 35 (`'C' - Cipher.MIN_CHAR`), then this shift is applied to `H` resulting in the encrypted character `k` (`'H' + 35 = 107 = (int)'k'`). Repeating this process for all characters in the input computes the appropriate ciphertext!

Decrypting is a similar process: repeat the key until it matches the length of the ciphertext, then calculate the first shift value based on the displacement of the first key character (`shift = repeatedKey.charAt(i) - Cipher.MIN_CHAR`), then remove `shift` from the corresponding ciphertext character to decrypt it. If the shifted value falls outside the encodable range (`< Cipher.MIN_CHAR`) it should loop back around to the end.

NOTE: Remember for both encrypting and decrypting, if you ever go past either end of the encodable range (`< Cipher.MIN_CHAR` or `> Cipher.MAX_CHAR`) you should loop back around to the other end. This process is described more in-depth in the `CaesarShift` section above.

Below is a diagram outlining the process visually:



NOTE: Before continuing, trace through the above example and make sure you understand how to encrypt and decrypt an input given a key.

Interestingly, unlike other classical ciphers the Vigenere has no guarantee that a single character in the plaintext will always map to the same character in the ciphertext (`AB` will become `CC` if the key is `!"`; both `A` and `B` become `C`!). This makes it much more complicated to crack as an adversary.

Required Behavior:

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

```
public Vigenere(String key)
```

- Constructs a new Vigenere Cipher with the provided key
- An `IllegalArgumentException` should be thrown if the key is empty or if the value falls outside of the encodable range.

4. Your choice!

Here, you'll implement an encryption scheme that sounds most interesting to you! There are no constraints on this option, other than your encryption scheme must be one-to-one (every output sequence must have a **single** unique input sequence and vice versa).

If you would like to implement a different Cipher, you will need specify what the Cipher will be as part of your proposal. Your proposed requirements should be similar in scope and complexity to the requirements for the suggested extensions. Post in this Ed thread [TODO] to propose a different Cipher.

Try your new Cipher!

Go ahead and create a MultiCipher that uses some combination of the base assignment and your extension. Note that when using one of the recommended extensions, it should be much, much more difficult to try and decrypt your message! This is because all of the recommended ciphers involve some element of shuffling characters into nonsense inputs / shifting by non-constant amounts. An adversary would have a much harder time trying to crack this combination than your previous MultiCiphers!

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.

Below are the JUnit testing requirements for this assignment:

- Substitution.java: 2 testing methods (each with >= 2 assertion calls) where `Cipher.MIN_CHAR = 'A'` and `Cipher.MAX_CHAR = 'Z'`
- CaesarKey.java: 2 testing methods (each with >= 2 assertion calls) where `Cipher.MIN_CHAR = 'A'` and `Cipher.MAX_CHAR = 'Z'`
- CaesarShift.java: 2 testing methods (each with >= 2 assertion calls) where `Cipher.MIN_CHAR = 'A'` and `Cipher.MAX_CHAR = 'Z'`
- MultiCipher.java: 2 testing methods (each with >= 2 assertion calls) where `Cipher.MIN_CHAR = 'A'` and `Cipher.MAX_CHAR = 'Z'`
- Your extension: 2 testing methods (each with >= 2 assertion calls) where `Cipher.MIN_CHAR = 'A'` and `Cipher.MAX_CHAR = 'Z'`

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 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.

Concealment: Example Ciphers

Sample implementations with annotations for `VariableConcealment` and `ConstantConcealment` have been provided so you can see how certain implementation choices might be made.

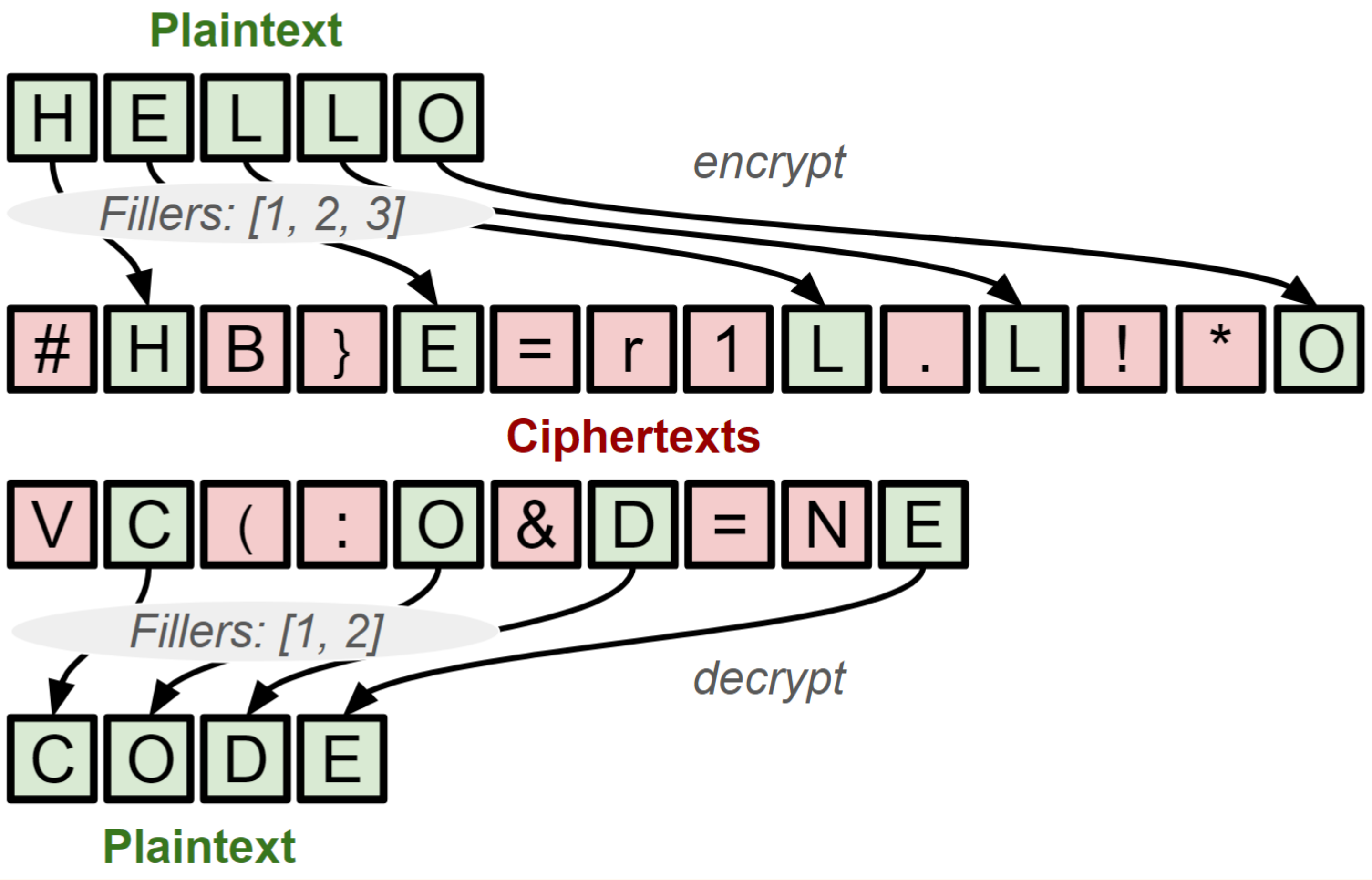
VariableConcealment

▼ Expand

This encryption scheme involves confusing any potential adversary with a jumble of random characters, placing the original message at specific locations within the encrypted message. Although this can manifest a variety of ways (the character that starts every sentence in a paragraph, the left-most words on a physical page of paper, etc.), in this class you'll be placing the characters from the original message after specified random "filler" characters. To make things more confusing, we'll be making it such that the filler values are variable / non-constant!

For example, if the provided `fillers` are [1, 2, 3], you would construct your ciphertext by first placing 1 random character from the encodable range within your ciphertext, followed by the first character from the plaintext. Then 2 random characters followed by the second character from the plaintext. Then 3 random characters and the third character from the plaintext. Loop back to the beginning of the filler values and repeat this process until you run out of characters in the plaintext and you have your encrypted message!

Decrypting involves the opposite - given a string full of junk characters, take out the important ones to form the plaintext. Given you know the filler values, this process just involves concatenating together every (filler + 1)th character from the string for every filler value within the provided `fillers` list. Below is a visual representation of this process:



HINT: To generate "junk" characters, you can randomly generate integers between `Cipher.MIN_CHAR` and `Cipher.MAX_CHAR` inclusive and cast them into characters (more information in the "Background" slide)

Required Behavior

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

```
public VariableConcealment()
```

- Constructs a new VariableConcealment with no filler values

```
public VariableConcealment(List<Integer> fillers)
```

- Constructs a new VariableConcealment with the provided filler values
- An `IllegalArgumentException` should be thrown in the following cases:
  - The provided `fillers` is an empty list
  - Any value within `fillers` is non-positive

```
public void setFillers(List<Integer> fillers)
```

- Updates the fillers for this VariableConcealment Cipher.
- An `IllegalArgumentException` should be thrown in the following cases:

- The provided `fillers` is an empty list
- Any value within `fillers` is non-positive

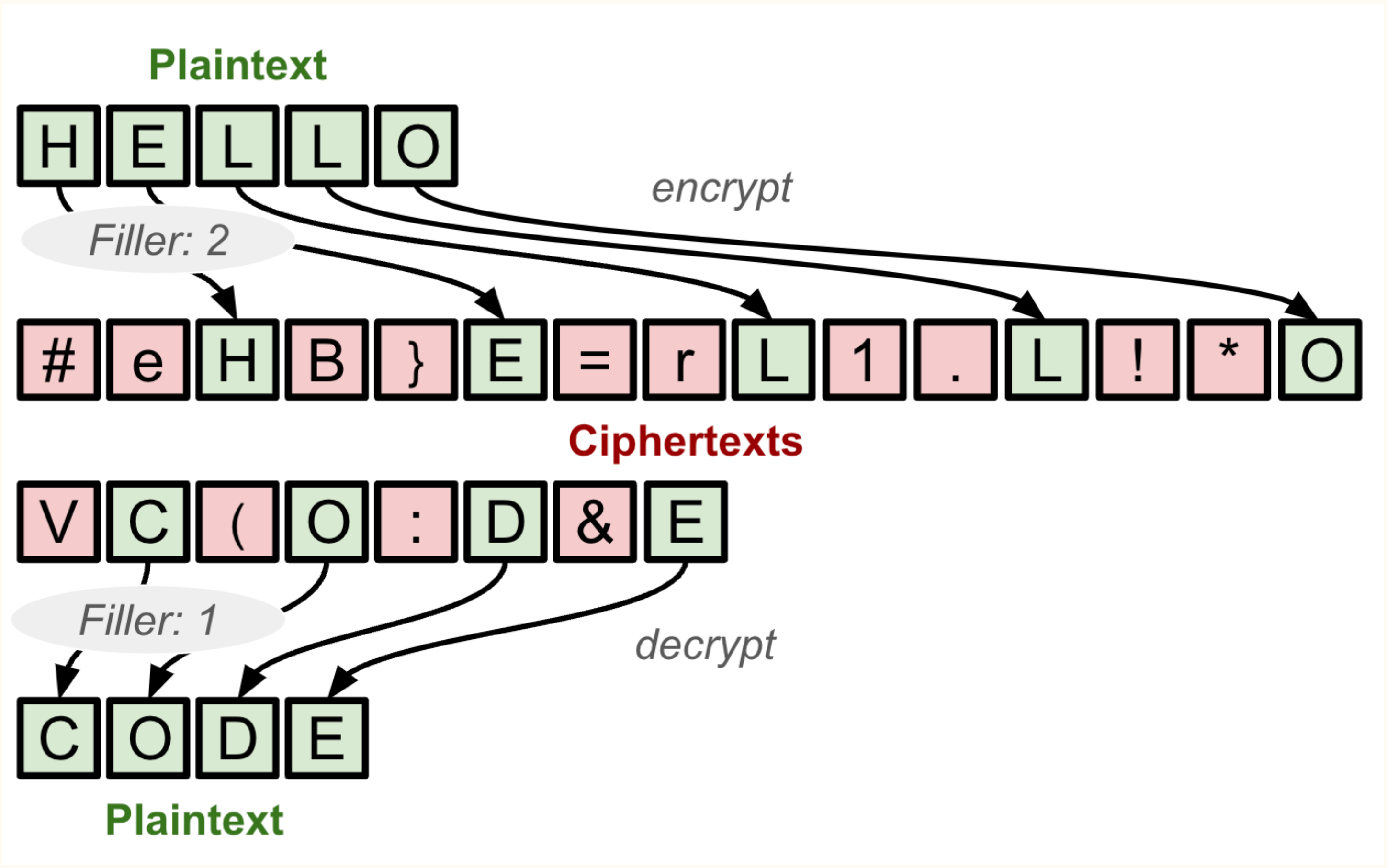
Since we're allowing clients to set filler values after construction (via the no-argument constructor and the `setFillers` method), **`encrypt` / `decrypt` should throw an `IllegalStateException` if the shifter was never set**:

```
VariableConcealment a = new VariableConcealment ();
a.encrypt("BAD");    // Should throw an IllegalStateException since the fillers were never set!
```

ConstantConcealment

▼ Expand

This encryption is a simplified version of the one described above where we'll only allow a constant number of filler characters. Consider the following example in which we set our constant filler value to 2:



✓ **HINT**: Note that this process would exactly match that of `VariableConcealment` given a list of `fillers` with just a single value! Keeping in mind our recently learned concepts, **what can we say about the relationship between `ConstantConcealment` and `VariableConcealment`**? How can we take advantage of those similarities to *reduce redundancy* between these two classes?

Reflection

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.

Please answer all questions.

Question 1

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

No response

Question 2

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 3

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

No response

Question 4

What did you struggle with most on this assignment?

No response

Question 5

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

No response

Question 6

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

No response

Question 7

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 8

[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