

# 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 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:**

 [P0\\_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

**i** **NOTE:** Remember, you should be making use of the class constants within this class rather than hardcoding character values within your implementations.

```
import java.util.*;
import java.io.*;

// Represents a classical cipher that is able to encrypt a plaintext into a ciphertext, and
// decrypt a ciphertext into a plaintext. Also capable of encrypting and decrypting entire files
public abstract class Cipher {
    // The minimum character able to be encrypted/decrypted by any cipher
    // (we encourage you to change this value when testing!)
    public static final int MIN_CHAR = (int)('A');

    // The maximum character able to be encrypted/decrypted by any cipher
    // (we encourage you to change this value when testing!)
    public static final int MAX_CHAR = (int)('Z');

    // The total number of characters able to be encrypted/decrypted by any cipher
    // (aka. the total number of characters in the encodable range)
    public static final int TOTAL_CHARS = MAX_CHAR - MIN_CHAR + 1;

    // Behavior: Applies this Cipher's encryption scheme to the file with the
    //             given 'fileName', creating a new file to store the results.
    // Exceptions: Throws a FileNotFoundException if a file with the provided 'fileName'
    //             doesn't exist
    // Returns: None
    // Parameters: 'fileName' - The name of the file to be encrypted. Should be non-null.
    public void encryptFile(String fileName) throws FileNotFoundException {
        fileHelper(fileName, true, "-encrypted");
    }

    // Behavior: Applies the inverse of this Cipher's encryption scheme to the file with the
    //             given 'fileName' (reversing a single round of encryption if previously applie
    //             creating a new file to store the results.
    // Exceptions: Throws a FileNotFoundException if a file with the provided 'fileName'
    //             doesn't exist
    // Returns: None
}
```

```

// Parameters: 'fileName' - The name of the file to be decrypted. Should be non-null.
public void decryptFile(String fileName) throws FileNotFoundException {
    fileHelper(fileName, false, "-decrypted");
}

// Behavior: Reads from an input file with 'fileName', either encrypting or decrypting
//           depending on 'encrypt', printing the results to a new file with 'suffix'
//           appended to the input file's name
// Exceptions: Throws a FileNotFoundException if a file with the provided 'fileName'
//           doesn't exist
// Returns: None
// Parameters: 'fileName' - the name of the file to be encrypted / decrypted. Should be
//           non-null.
//           'encrypt' - whether or not encryption should occur
//           'suffix' - appended to the fileName when creating the output file. Should be
//           non-null.
private void fileHelper(String fileName, boolean encrypt, String suffix)
    throws FileNotFoundException {
    Scanner sc = new Scanner(new File(fileName));
    String out = fileName.split("\\.txt")[0] + suffix + ".txt";
    PrintStream ps = new PrintStream(out);
    while(sc.hasNextLine()) {
        String line = sc.nextLine();
        ps.println(encrypt ? encrypt(line) : decrypt(line));
    }
}

// Behavior: Applies this Cipher's encryption scheme to 'input', returning the result
// Exceptions: None
// Returns: The result of applying this Cipher's encryption scheme to `input`
// Parameters: 'input' - the string to be encrypted. Should be non-null and all characters o
//           'input' should be within the encodable range.
public abstract String encrypt(String input);

// Behavior: Applies this inverse of this Cipher's encryption scheme to 'input' (reversing
//           a single round of encryption if previously applied), returning the result
// Exceptions: None
// Returns: The result of applying the inverse of this Cipher's encryption scheme to `inp
// Parameters: 'input' - the string to be encrypted. Should be non-null and all characters o
//           'input' should be within the encodable range.
public abstract String decrypt(String input);
}

```

## 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 "Swap: Example Ciphers" slide for an example implementation of a Cipher!

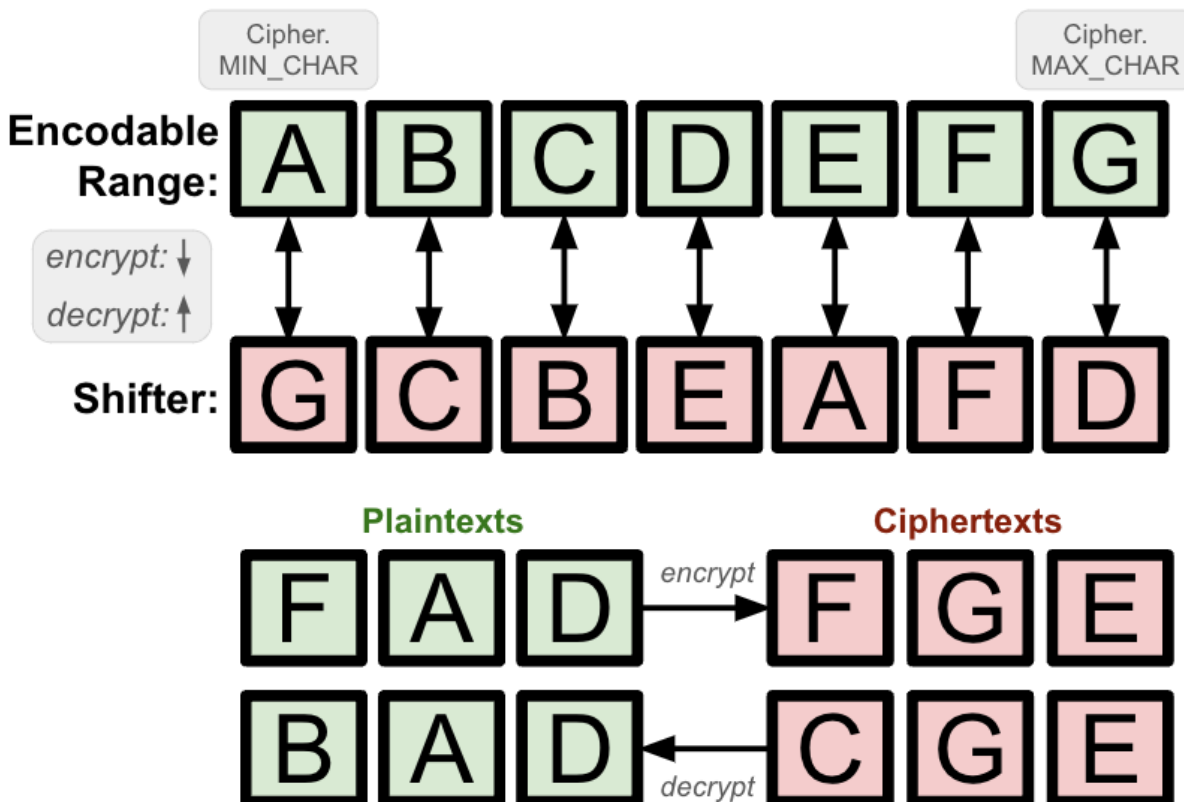
## Substitution.java

▼ 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 be 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 characters 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".

**i** **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 any of 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 any of 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!
```

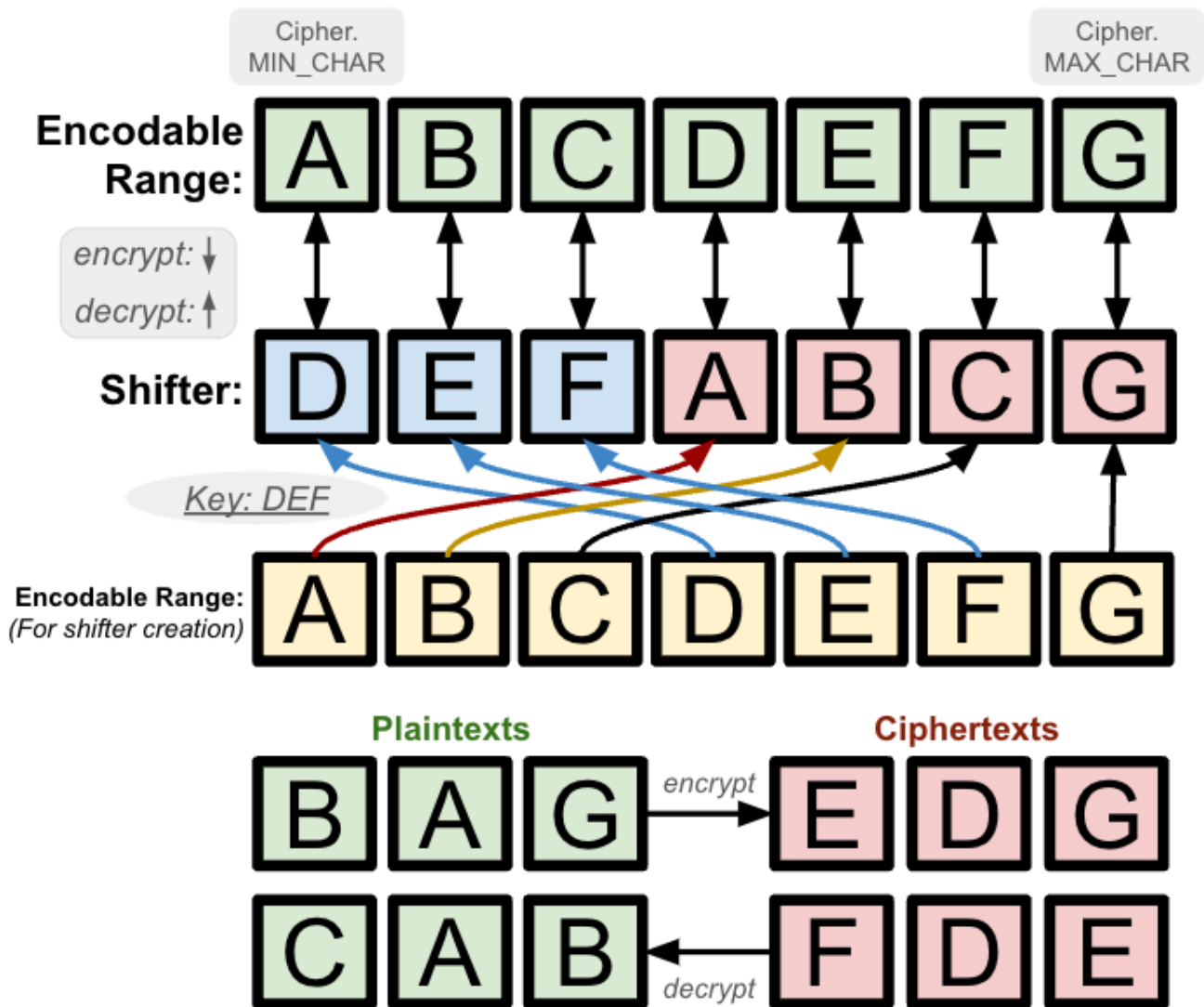
## CaesarKey.java

### ▼ 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 given `key` meets any of the following cases:
  - Is empty
  - Contains a duplicate character
  - Any individual character falls outside the encodable range (`< Cipher.MIN_CHAR` or `> Cipher.MAX_CHAR`)

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

## CaesarShift.java

### ▼ Expand

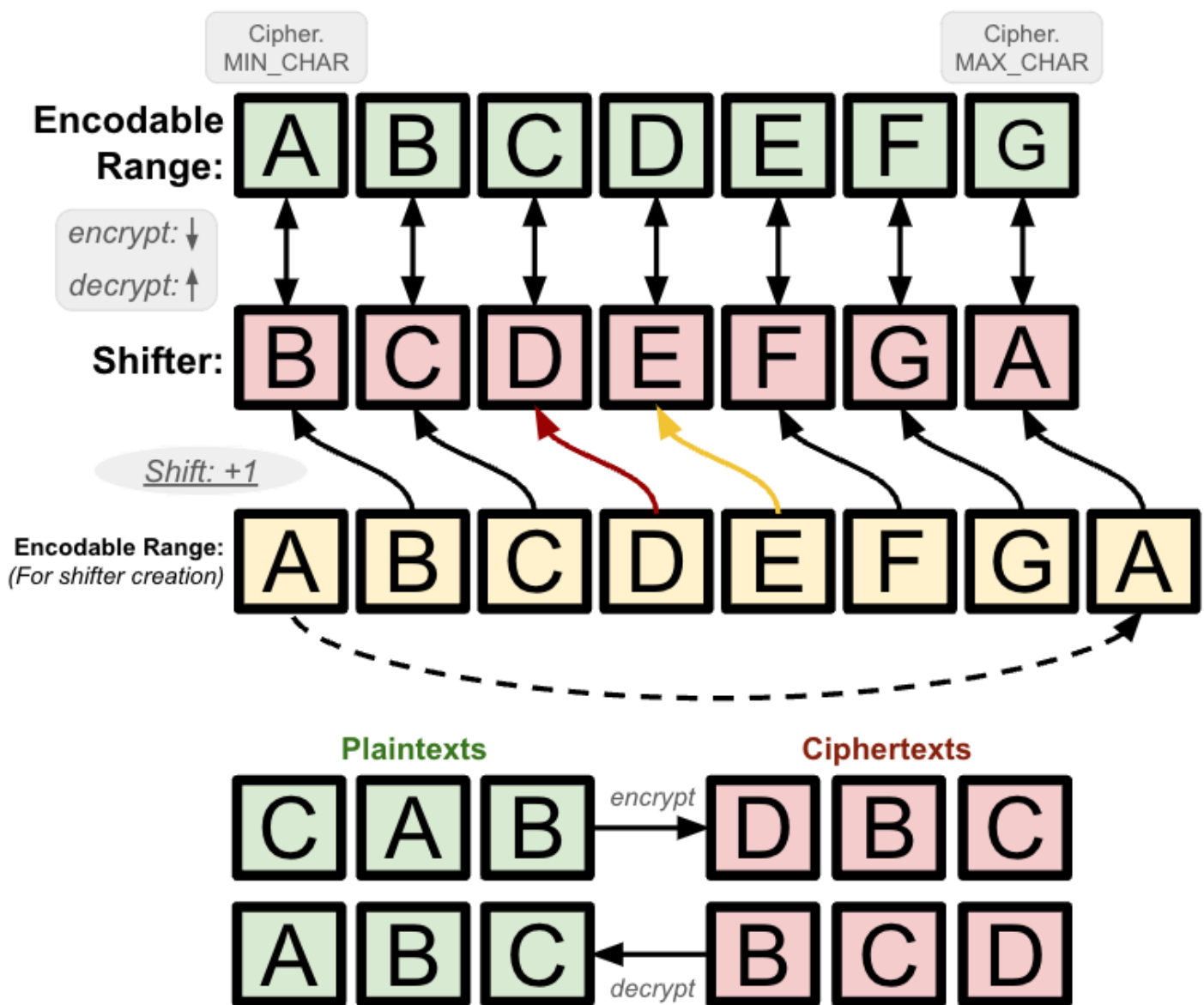
This encryption scheme draws inspiration from the Substitution Cipher, except it involves shifting all encodable characters to the left by some provided shift amount.

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, 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 `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)`).

Consider the following diagram for a visual explanation:



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?

✓ **HINT:** Notice that 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).

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`



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

## MultiCipher.java

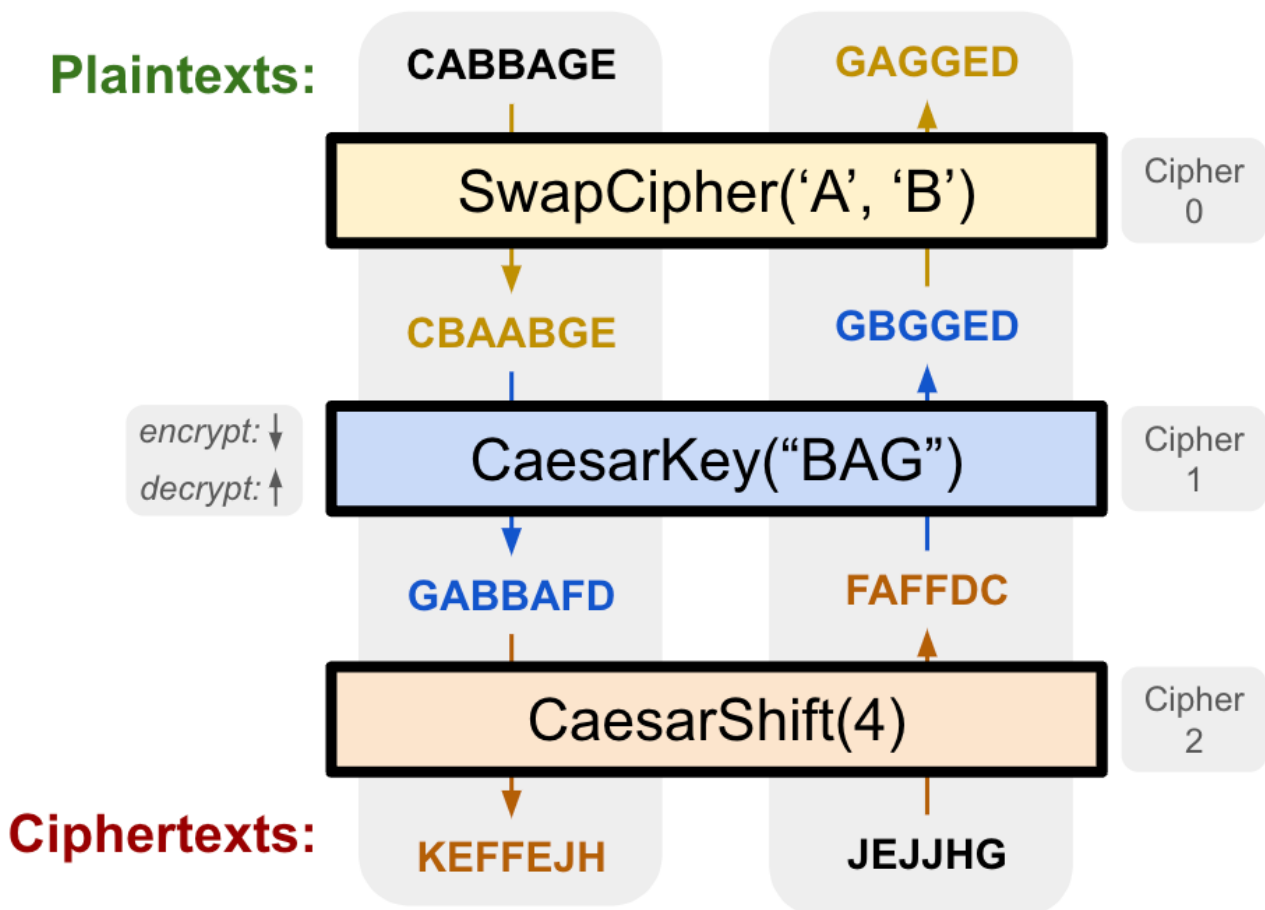
▼ Expand

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 SwapCipher with arguments 'A' and 'B', a CaesarKey with key "BAG", and a CaesarShift of 4.



**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 `KEFFEJH`.

On the right in the above example, we start at the ciphertext: `JEJJHG` 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.

## 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 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



**NOTE:** This walkthrough was created for a previous iteration of this assignment. You might notice that things are slightly different (Concealment changed into Swap, and there is no more Creative Extension). A good majority of the assignment is the same which is why we're still using it, but if you have any questions please let us know!



# Swap: Example Ciphers

Sample implementations with annotations for `MultiSwapCipher` and `SwapCipher` have been provided so you can see how certain implementation choices might be made. These should have been covered in-depth in section, so hopefully they are somewhat familiar! The only major difference is that the `Cipher` interface has been converted into an abstract class such that files are able to be encrypted / decrypted as well.

## MultiSwapCipher

### ▼ Expand

This encryption scheme takes in a list of characters with which to "swap" while encrypting. For a better idea of what this looks like, imagine the following list of swaps `['A', 'B', 'C']`. When encrypting, this means we would swap all As to Bs, Bs to Cs, and Cs to As in our ciphertext. When decrypting, we would do the reverse: As become Cs, Cs become Bs, and Bs become As such that we end up with the same plaintext.

## Required Behavior

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

```
public MultiSwapCipher()
```

- Constructs a `MultiSwapCipher` with no set swaps

```
public MultiSwapCipher(List<Character> swaps)
```

- Constructs a `MultiSwapCipher` using the provided list to determine which characters to swap with one another.
- An `IllegalArgumentException` should be thrown in the following cases:
  - There are < 2 elements within the list (as no encryption would occur)
  - Any of the characters in the list fall outside the encodable range

```
public void setSwaps(List<Character> swaps)
```

- Updates the swap list for this `MultiSwapCipher`
- An `IllegalArgumentException` should be thrown in the following cases:
  - There are < 2 elements within the list (as no encryption would occur)
  - Any of the characters in the list fall outside the encodable range

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

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

## SwapCipher

▼ Expand

This encryption is a simplified version of the one described above where we only ever swap two characters. For a better idea of what this looks like, imagine the following swaps 'A' and 'B'. When encrypting, this means we would swap all As to Bs and Bs to As in our ciphertext. When decrypting, we would do the reverse: Bs become As and As become Bs such that we end up with the same plaintext.

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

## Required Behavior

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

```
public SwapCipher(char a, char b)
```

- Constructs a new `SwapCipher` with the provided characters to swap
- An `IllegalArgumentException` should be thrown in the case that either of the characters falls outside the encodable range

## Test It

To test this Cipher and any others, you may run the provided `Client.java` file. If you wanted to test one of the swap ciphers, you could change the `CHOSEN_CIPHER` on line 6 to be a `MultiSwapCipher` / `SwapCipher` object with appropriate swaps. For example, in the following line, our chosen cipher is the `MultiSwapCipher` cipher with swaps of [A, B, C].

```
public static final Cipher CHOSEN_CIPHER = new MultiSwapCipher(List.of('A', 'B', 'C'));
```

Then, you can hit run and try your own inputs! You can also write JUnit tests within the provided `Testing.java` file.

---

# Substitution

All required files can be found on the `Getting Started` slide



**WARNING:** This slide is *NOT* graded

## Slide Overview

Welcome to the first step, where you will be implementing one of the required files, `Substitution.java` !

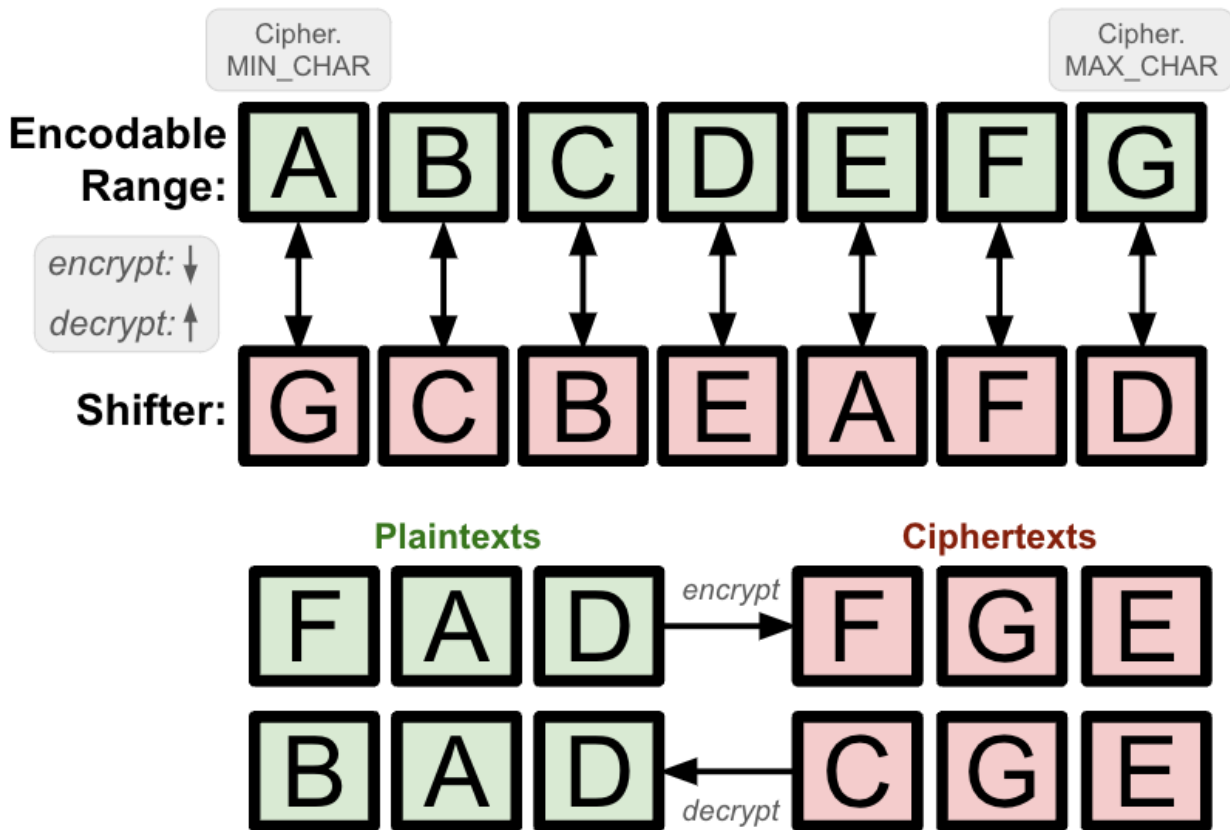
Note that the following descriptions often refer to the "encodable/encryptable range," which is defined by the `Cipher.MIN_CHAR` (lowest value in the range), `Cipher.MAX_CHAR` (highest value in the range), and `Cipher.TOTAL_CHARS` (total number of characters within the range) constants within `Cipher.java`

## 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 `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".

✓ **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 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 any of 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 any of 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!
```

**Once you've successfully passed the tests for this slide:**

Download your `Substitution.java` file and upload it to the next slide, **CaesarKey.java**. **Note that this slide is not graded.** However, we recommend that you do not move on to the rest of the program until you have passed all tests in this slide.

---

# CaesarKey

All required files can be found on the `Getting Started` slide



**WARNING:** This slide is *NOT* graded

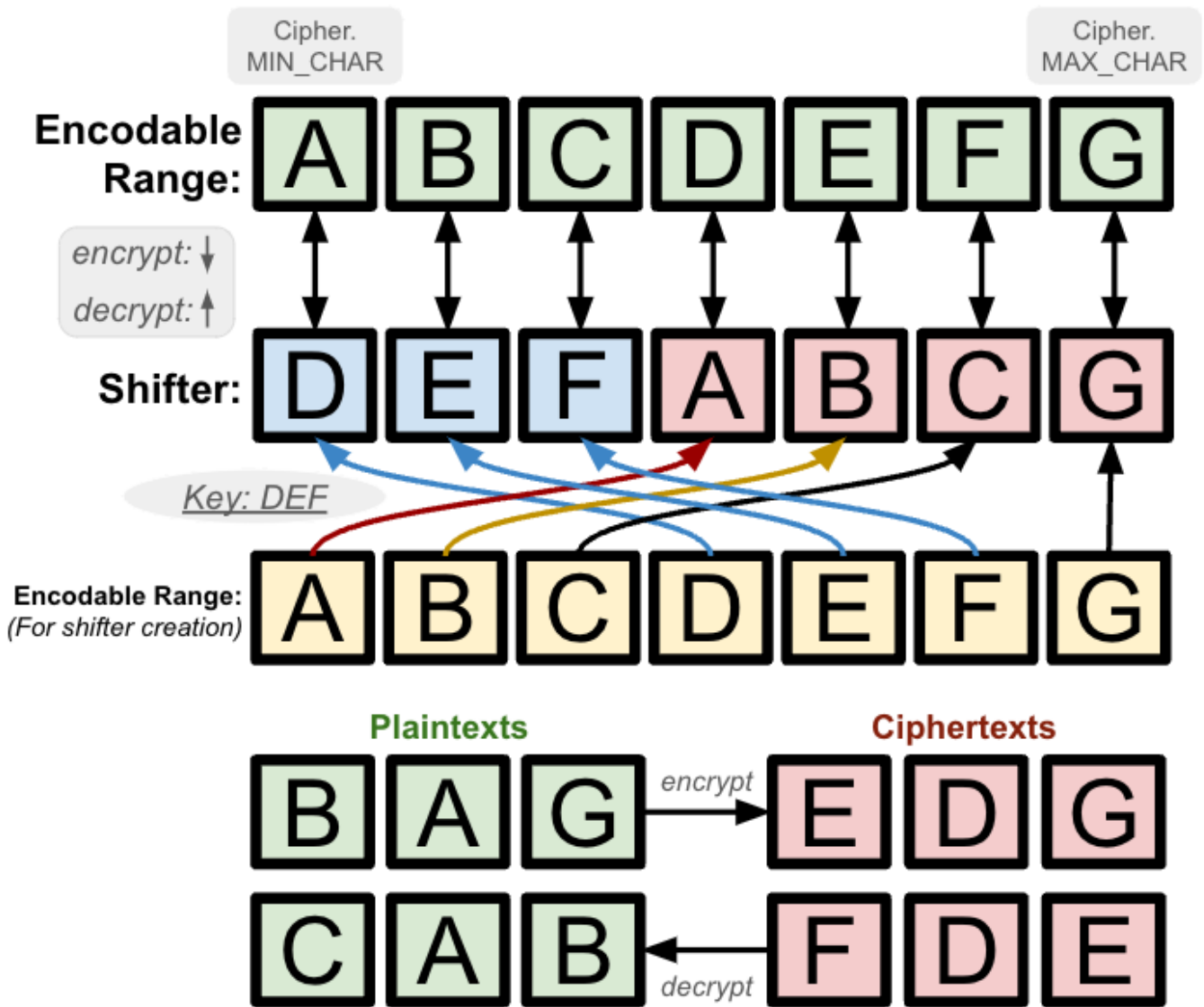
## Slide Overview

Welcome to the next step! Here is where all the inheritance you've been learning over the past week comes into play. As you read through this specification, take note of the behavioral similarities between `Substitution.java` and `CaesarKey.java`

## 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 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 that 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 given `key` meets any of the following cases:
  - Is empty
  - Contains a duplicate character
  - Any individual character falls outside the encodable range ( `< Cipher.MIN_CHAR` or `> Cipher.MAX_CHAR` )



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

## Once you've successfully passed the tests for this slide:

Download your `Substitution.java` and `CaesarKey.java` files and upload them to the next slide, **CaesarShift.java**. **Note that this slide is not graded.** However, we recommend that you do not move on to the rest of the program until you have passed all tests in this slide.

# CaesarShift

All required files can be found on the `Getting Started` slide



**WARNING:** This slide is *NOT* graded

## CaesarShift.java

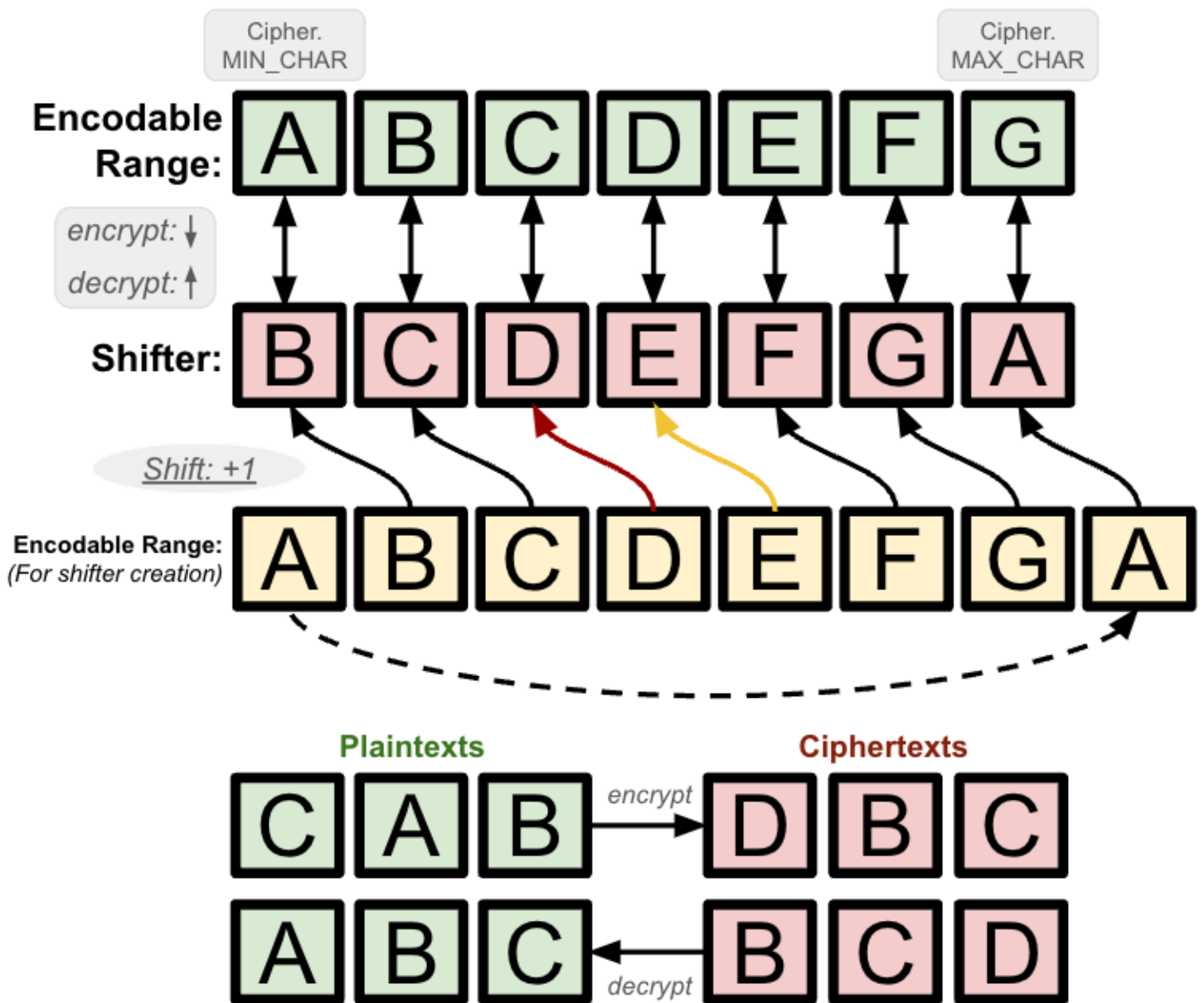
This encryption scheme draws inspiration from the Substitution Cipher, except it involves shifting all encodable characters to the left by some provided shift amount.

Applying the CaesarShift Cipher is defined as replacing each input character with the corresponding character in `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, 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 `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)`).

Consider the following diagram for a visual explanation:



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?

✓ **HINT:** Notice that 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).

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`



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

### Once you've successfully passed the tests for this slide:

Download your `Substitution.java`, `CaesarKey.java` and `CaesarShift.java` files and upload them to the next slide, **MultiCipher.java**. **Note that this slide is not graded.** However, we recommend that you do not move on to the rest of the program until you have passed all tests in this slide.

# MultiCipher

All required files can be found on the `Getting Started` slide



**WARNING:** This slide is *NOT* graded

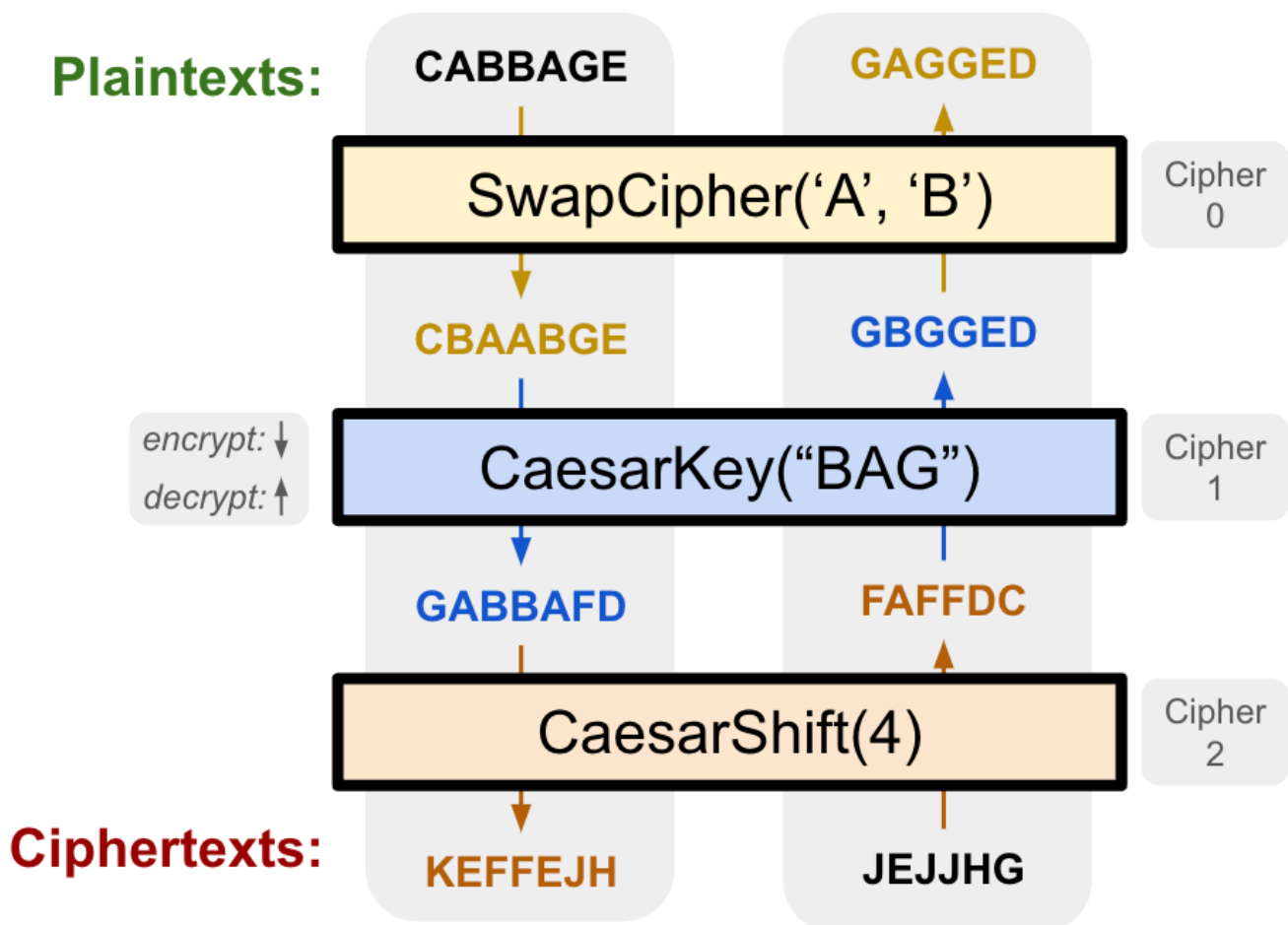
## MultiCipher.java

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 encodabe 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 `KEFFEJH`.

On the right in the above example, we start at the ciphertext: `JEJJHG` 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.

## 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

**Once you've successfully passed the tests for this slide:**

Download your `Substitution.java`, `CaesarKey.java`, `CaesarShift.java`, and `MultiCipher.java` files and upload them to the next slide, `□ Ciphers □`. **Note that this slide is not graded.** However, we recommend that you do not move on to the rest of the program until you have passed all tests in this slide.

## □ Ciphers □

All required files can be found on the [Getting Started](#) slide



**WARNING:** This slide *IS* graded

Write/Upload your implementations to `Substitution.java`, `CaesarShift.java`, `CaesarKey.java`, and `MultiCipher.java` here! Remember that implementations for *all* of these are required for this assignment.

## 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}u!)(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 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.

---

# 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

The next 4 questions will require you reflect on privacy, encryption and a few ethical complications. Start by watching the following video (8m 12s):

An error occurred.

---

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



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

Describe the difference between "encryption in transit" (involving  $K_{AS}$  and  $K_{BS}$ ) and "end-to-end encryption" (involving  $K_{AB}$ ) as described in the video.

*No response*

#### **Question 4**

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?

*No response*

#### **Question 5**

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? How would you feel if software capable of decrypting any and all private information on your devices existed?

*No response*

#### **Question 6**

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?

*No response*

#### **Question 7**

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

*No response*

**Question 8**

What did you struggle with most on this assignment?

*No response*

**Question 9**

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

*No response*

**Question 10**

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

*No response*

**Question 11**

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 12**

[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*

---

□ Final Submission □

□ Final Submission□

Fill out the box below and click "Submit" in the upper-right corner of the window to submit your work.

**Question**

I attest that the work I am about to submit is my own and was completed according to the course [Academic Honesty and Collaboration](#) policy. If I collaborated with any other students or utilized any outside resources, they are allowed and have been properly cited. If I have any concerns about this policy, I will reach out to the course staff to discuss *before* submitting.

(Type "yes" as your response.)

*No response*

---

# Diagrams

[https://docs.google.com/presentation/d/1nVrl0AYpn\\_LckCgJnsHw9uOxmKc5Z6lCasggFfXhJFM/edit?usp=sharing](https://docs.google.com/presentation/d/1nVrl0AYpn_LckCgJnsHw9uOxmKc5Z6lCasggFfXhJFM/edit?usp=sharing)

---

## [Scaffold] Ciphers

Write Implementations to `Substitution.java`, `CaesarShift.java`, `CaesarKey.java`, and `MultiCipher.java` here! Remember that implementations for all of these are required for this assignment.