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:

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! 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

Learning Objectives

By completing this assignment, students will demonstrate their ability to:

- Define relationships between Java classes using inheritance, abstract classes, and references.
- Write a well-designed Java class that extends a given abstract class,
- Produce clear and effective documentation to improve comprehension and maintainability of classes,
- Write classes that are readable and maintainable, and that conform to provided guidelines for style, and implementation

System Structure

We will represent ciphers with the following provided abstract class. You may modify the constants of this class to help with debugging your implementations (we recommend starting with a smaller set of valid characters, such `ABCDEFG` or the example of `RSACLVJ` — notice that the valid characters **do not have to be consecutive or in order**). Expand to see the default `Cipher.java` file

▼ Expand

> **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 valid characters allowed to be encoded or decoded by our Cipher.
    // May or may not be in order. May or may not be contiguous.
    public static final String VALID_CHARS
        = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";

    // Here are some other VALID_CHARS Strings that you can play around with!
    // Uncomment or copy-paste the VALID_CHARS String you want to use, and comment
    // the one you don't want to use:

    /*
    Spec example:
    public static final String VALID_CHARS = "RSACLVJ";

    A-Z, a-z
```

```
    public static final String VALID_CHARS
        = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";

    A-G
    public static final String VALID_CHARS = "ABCDEFG";

    ' ' - '}'
    public static final String VALID_CHARS
        = " !\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`" +
            "bacdefghijklmnopqrstuvwxyz{|}";
    */


    //   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 b
    //             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: Returns whether the character is valid.
    // Exceptions: None
    //    Returns: True if this character is valid. False otherwise.
    // Parameters: 'character' - The character to check
    public static boolean isCharValid(char character) {
        return VALID_CHARS.indexOf(character) != -1;
    }

    //   Behavior: Applies this Cipher's encryption scheme to 'input', returning the result
    // Exceptions: Throws an IllegalArgumentException if 'input' is null
    //    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 contain only valid characters.
    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: Throws an IllegalArgumentException if 'input' is null
    //    Returns: The result of applying the inverse of this Cipher's encryption scheme to `inpu
    // Parameters: 'input' - the string to be encrypted. Should be non-null and all characters o
    //             'input' should contain only valid characters.
    public abstract String decrypt(String input);
}
```

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 "valid characters," which is defined by the `Cipher.VALID_CHARS` constant within `Cipher.java`.

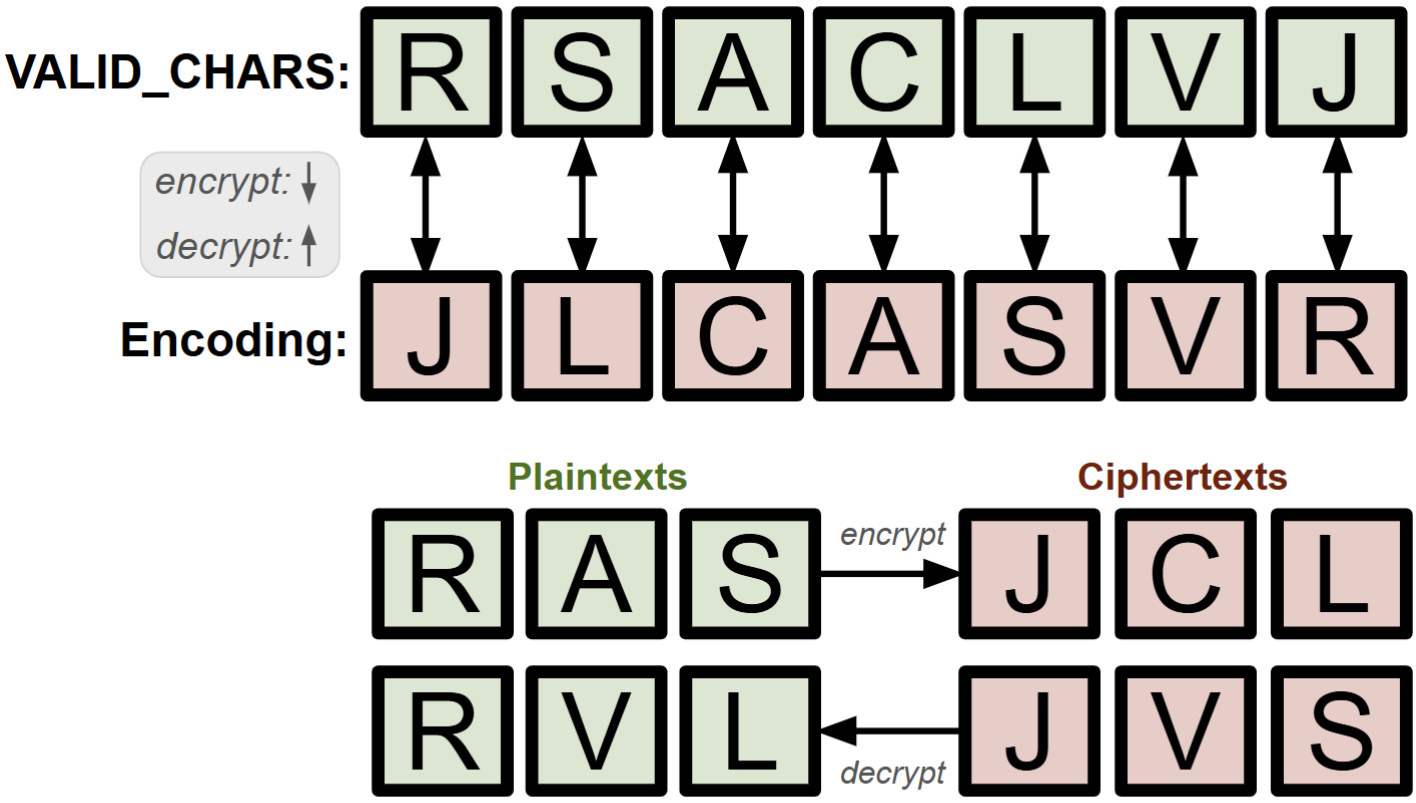
HINT: Check out the "Swap: Example Ciphers" slide for an example implementation of a Cipher!

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 set of valid characters (defined by `Cipher.VALID_CHARS`). To picture this, we can vertically align this `encoding` string with the valid characters and look at the corresponding columns to see the appropriate character mappings.

Here is an example:



In this example, our valid characters are the letters "RSACLVJ". In code, we represent this as all of the characters in `Cipher.VALID_CHARS` . We line this up with our given `encoding` String, which in this case is "JLCASVR", such that "RSACLVJ" is directly on top of "JLCASVR". This means that the letter `R` will be encrypted to the letter `J`, the letter `S` encrypts to the letter `L`, the letter `A` encrypts to the letter `C`, the letter `C` encrypts to the letter `A`, the letter `L` encrypts to the letter `S`, the letter `V` encrypts to the letter `V`, and the letter `J` encrypts to the letter `R`.

To decrypt, we would go in the opposite direction. Therefore, the letter `J` would be decrypted to the letter `R`, the letter `L` decrypts to the letter `S`, the letter `C` decrypts to the letter `A`, the letter `A` decrypts to the letter `C`, the letter `S` decrypts to the letter `L`, the letter `V` decrypts to the letter `V`, and the letter `R` decrypts to the letter `J`.

Given the encoding string above, the plaintext "RAS" would be encrypted into "JCK" and the ciphertext "JVS" decrypts into the plaintext "RVL".

HINT: Notice what really matters here is the position of each character in the set of valid characters, 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 valid characters in our Cipher
  - Contains a duplicate character
  - Any individual character is not a valid character (i.e., is not in `Cipher.VALID_CHARS`).
    - Consider `isCharValid()`!

```
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 valid characters in our Cipher
  - Contains a duplicate character
  - Any individual character is not a valid character (i.e., is not in `Cipher.VALID_CHARS`).
    - Consider `isCharValid()`!

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 Substition();
a.encrypt("RSA");    // Should throw an IllegalStateException since the encoding was never set!
```

CaesarShift.java

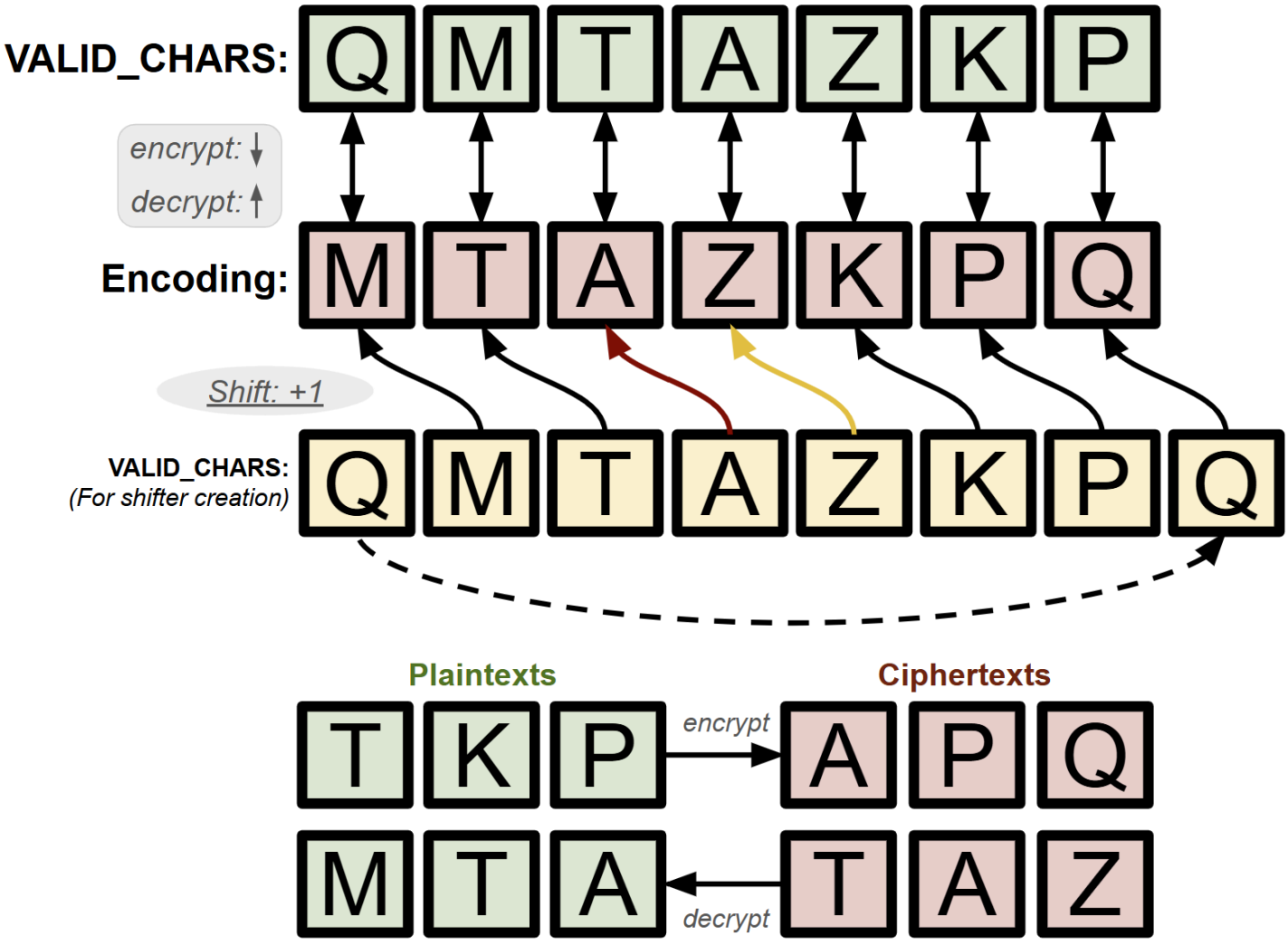
This encryption scheme draws inspiration from the Substitution Cipher, except it involves shifting all valid characters to the left by some provided shift amount.

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, inversing the CaesarShift Cipher is defined as replacing each input character with the corresponding character in the set of valid characters 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 valid characters are "QMTAZKP", `Q` would be replaced with `M`. Additionally, for characters that would shift past the end of the set of valid characters (`P` in this case), the replacement character can be found by looping back around to the front of the valid characters. In this example, `P` would map to `Q`. If the shift was 3 and our valid characters are "QMTAZKP", then `Q` would map to `A`, `M` would map to `Z`, and so on, with Z, K, and P wrapping around to map to Q, M, and T respectively.

Consider the following diagram for a visual explanation:



In this example, our valid characters are the letters "QMTAZKP". To create the encoding, we move the character at the front of the set of valid characters 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 "MTAZKPQ". Notice how the first letter, `Q`, was

moved from the front to the back. Similarly to `Substitution`, the mapping of letters is made clearer by placing "QMTAZKP" on top of "MTAZKPQ", such that `Q` is encrypted to `M`, `M` is encrypted to `T`, `T` is encrypted to `A`, `A` is encrypted to `Z`, `Z` is encrypted to `K`, `K` is encrypted to `P`, and `P` is encrypted to `Q`. We go the opposite direction for decryption, so `M` is decrypted to `Q`, `T` is decrypted to `M`, `A` is decrypted to `T`, `Z` is decrypted to `A`, `K` is decrypted to `Z`, `P` is decrypted to `K`, and `Q` is decrypted to `P`.

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:

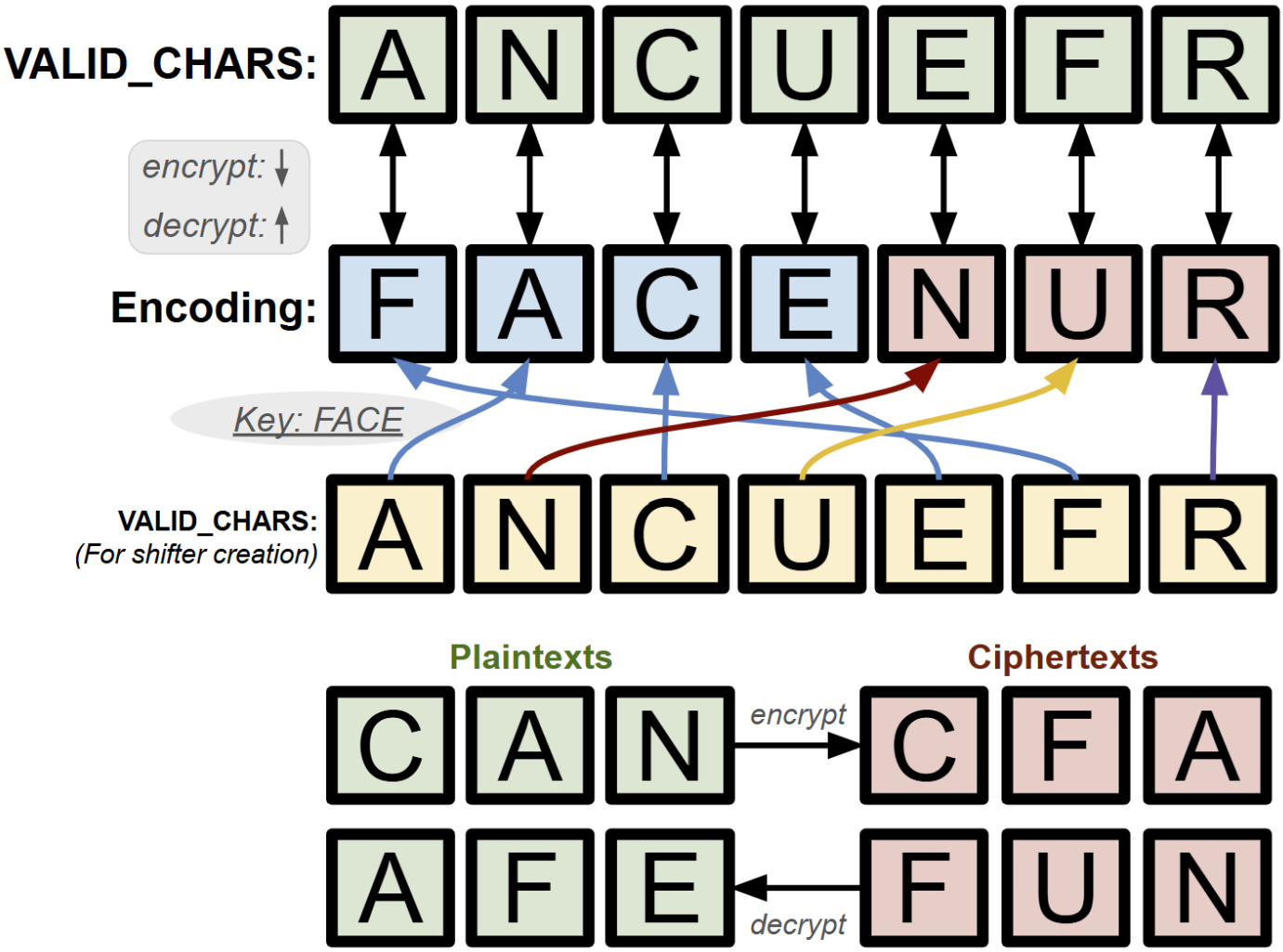
```
public CaesarShift(int shift)
```

- 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 valid characters following normally (minus the characters included in the key). This means that the first character in our valid characters would be replaced by the first character within the key. The second character in the valid characters 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 valid characters.

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 valid characters 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 valid characters in their original order, excluding characters 'F', 'A', 'C', and 'E' as they're already in the encoding. This results in the encoding String "FACENUR".

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`, `N` is encrypted to `A`, `C` is encrypted to `C`, `U` is encrypted to `E`, `E` is encrypted to `N`, `F` is encrypted to `U`, and `R` is encrypted to `R` . We invert this process to decrypt so that `F` decrypts to `A`, `A` decrypts to `B`, `C` decrypts to `C`, `E` decrypts to `U`, `N` decrypts to `E`, `U` decrypts to `F`, and `R` decrypts to `R`.

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 is not a valid character (i.e., is not in `Cipher.VALID_CHARS`).
        - Consider `isCharValid()`!

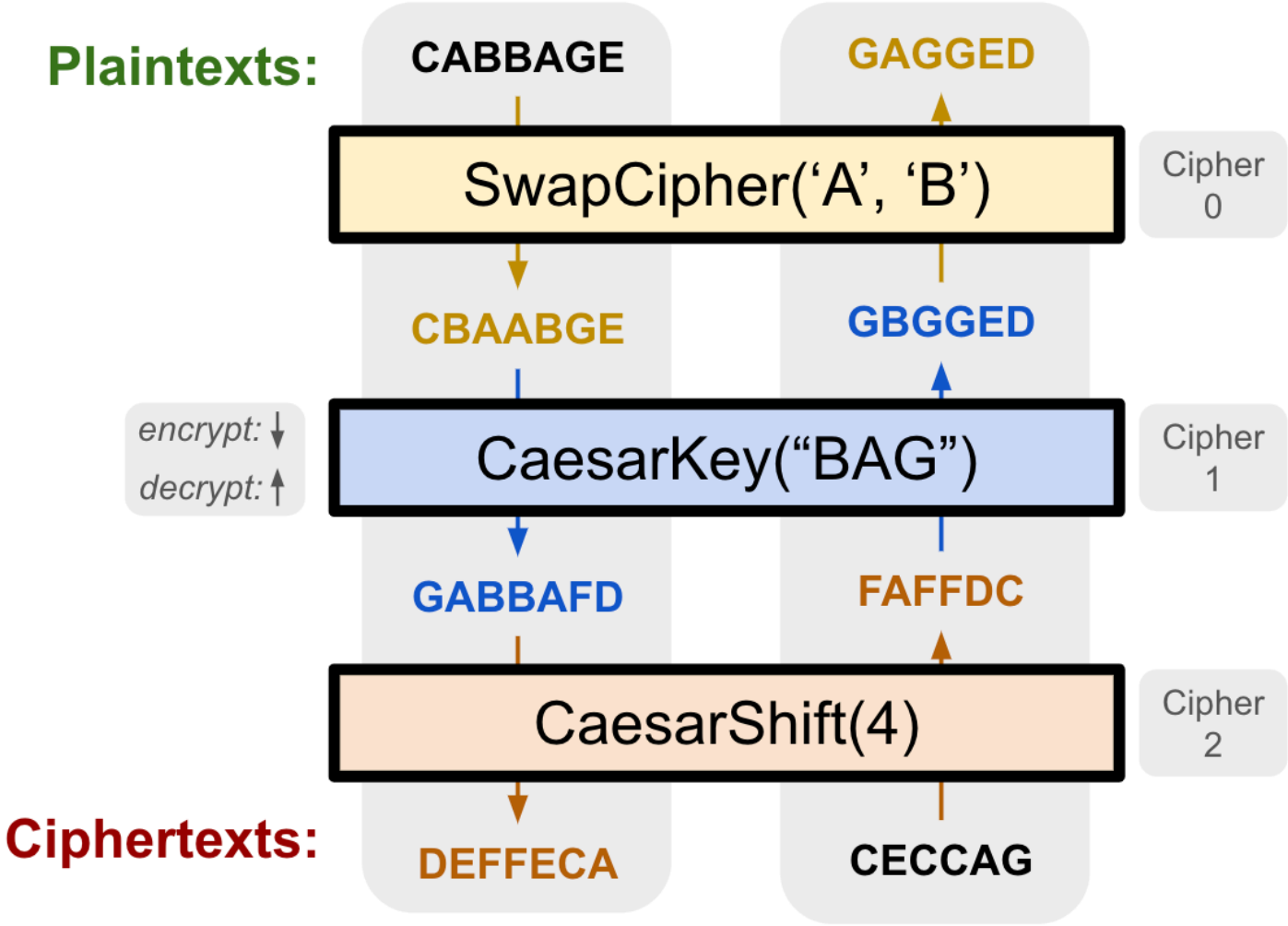
WARNING: We are ***requiring*** that you do not override encrypt / decrypt methods within `CaesarKey`. These should be inherited from a superclass.

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 valid characters are `ABCDEFG`



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.VALID_CHARS = " !\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`" +
                     "bacdefghijklmnopqrstuvwxyz{|}"
```

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 `Cipher.VALID_CHARS` to `"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"` 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.

[NOT GRADED] 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. You do **not** need to implement these ciphers.

MultiSwapCipher

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 is invalid.

```
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 is invalid.

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

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 is invalid.

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.

[NOT GRADED] 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` !

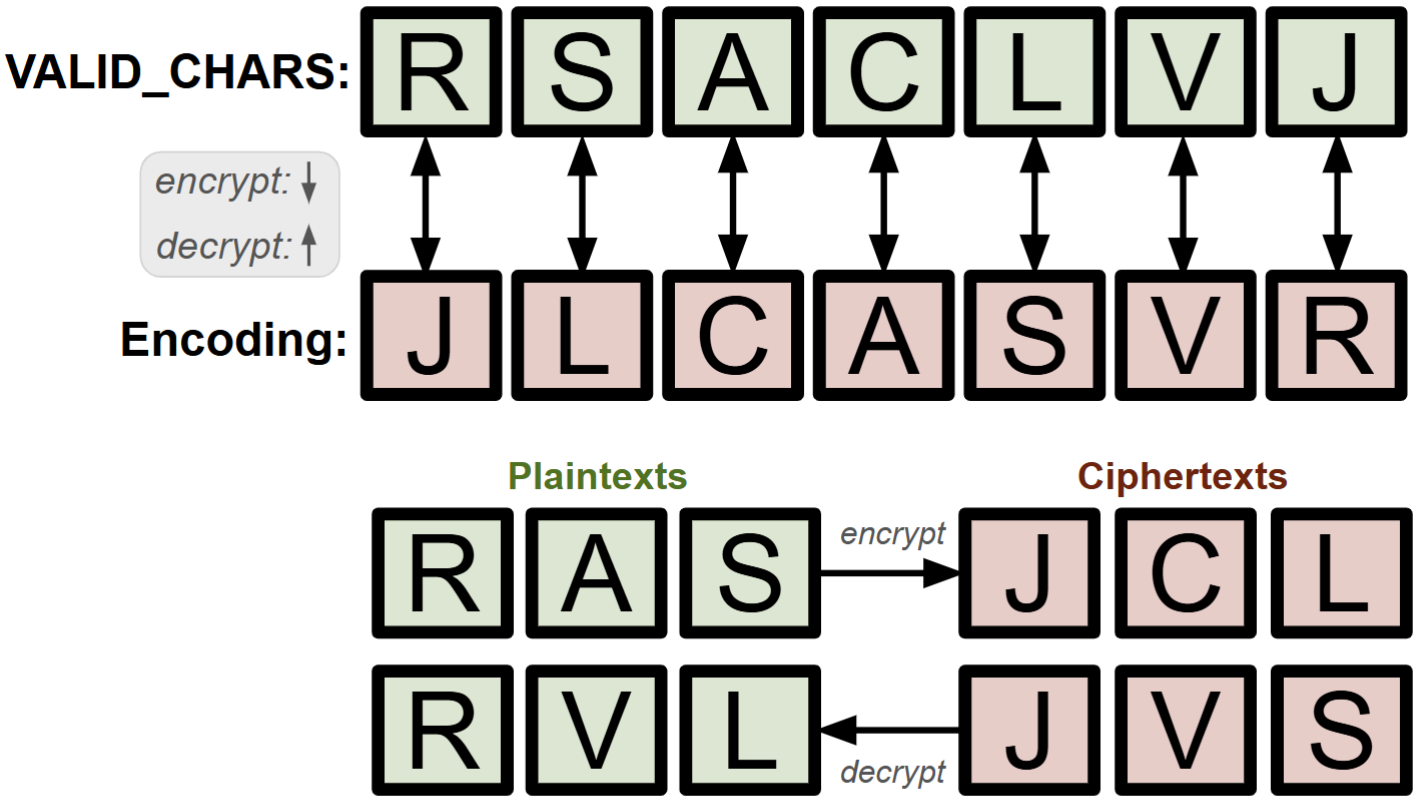
Note that the following descriptions often refer to the "valid characters," which is defined by the `Cipher.VALID_CHARS` constant within `Cipher.java`.

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 `encoding` string. The `encoding` will represent the output characters corresponding to the input character at the same relative position within the set of valid characters (defined by `Cipher.VALID_CHARS`). To picture this, we can vertically align this `encoding` string with the valid characters and look at the corresponding columns to see the appropriate character mappings.

Here is an example:



In this example, our valid characters are the letters "RSACLVJ". In code, we represent this as all of the characters in `Cipher.VALID_CHARS` . We line this up with our given `encoding` String, which in this case is "JLCASVR", such that "RSACLVJ" is directly on top of "JLCASVR". This means that the letter `R` will be encrypted to the letter `J`, the letter `S` encrypts to the letter `L`, the letter `A` encrypts to the letter `C`, the letter `C` encrypts to the letter `A`, the letter `L` encrypts to the letter `S`, the letter `V` encrypts to the letter `V`, and the letter `J` encrypts to the letter `R`.

To decrypt, we would go in the opposite direction. Therefore, the letter `J` would be decrypted to the letter `R`, the letter `L` decrypts to the letter `S`, the letter `C` decrypts to the letter `A`, the letter `A` decrypts to the letter `C`, the letter `S` decrypts to the letter `L`, the letter `V` decrypts to the letter `V`, and the letter `R` decrypts to the letter `J`.

Given the encoding string above, the plaintext "RAS" would be encrypted into "JCK" and the ciphertext "JVS" decrypts into the plaintext "RVL".

HINT: Notice what really matters here is the position of each character in the set of valid characters, 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 valid characters in our Cipher
  - Contains a duplicate character
  - Any individual character is not a valid character (i.e., is not in `Cipher.VALID_CHARS`).
    - Consider `isCharValid()`!

```
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 valid characters in our Cipher
  - Contains a duplicate character
  - Any individual character is not a valid character (i.e., is not in `Cipher.VALID_CHARS`).
    - Consider `isCharValid()`!

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 Substition();
a.encrypt("RSA");    // Should throw an IllegalStateException since the encoding 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, **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.

[NOT GRADED] CaesarShift

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

WARNING: This slide is *NOT* graded

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 `CaesarShift.java`

CaesarShift.java

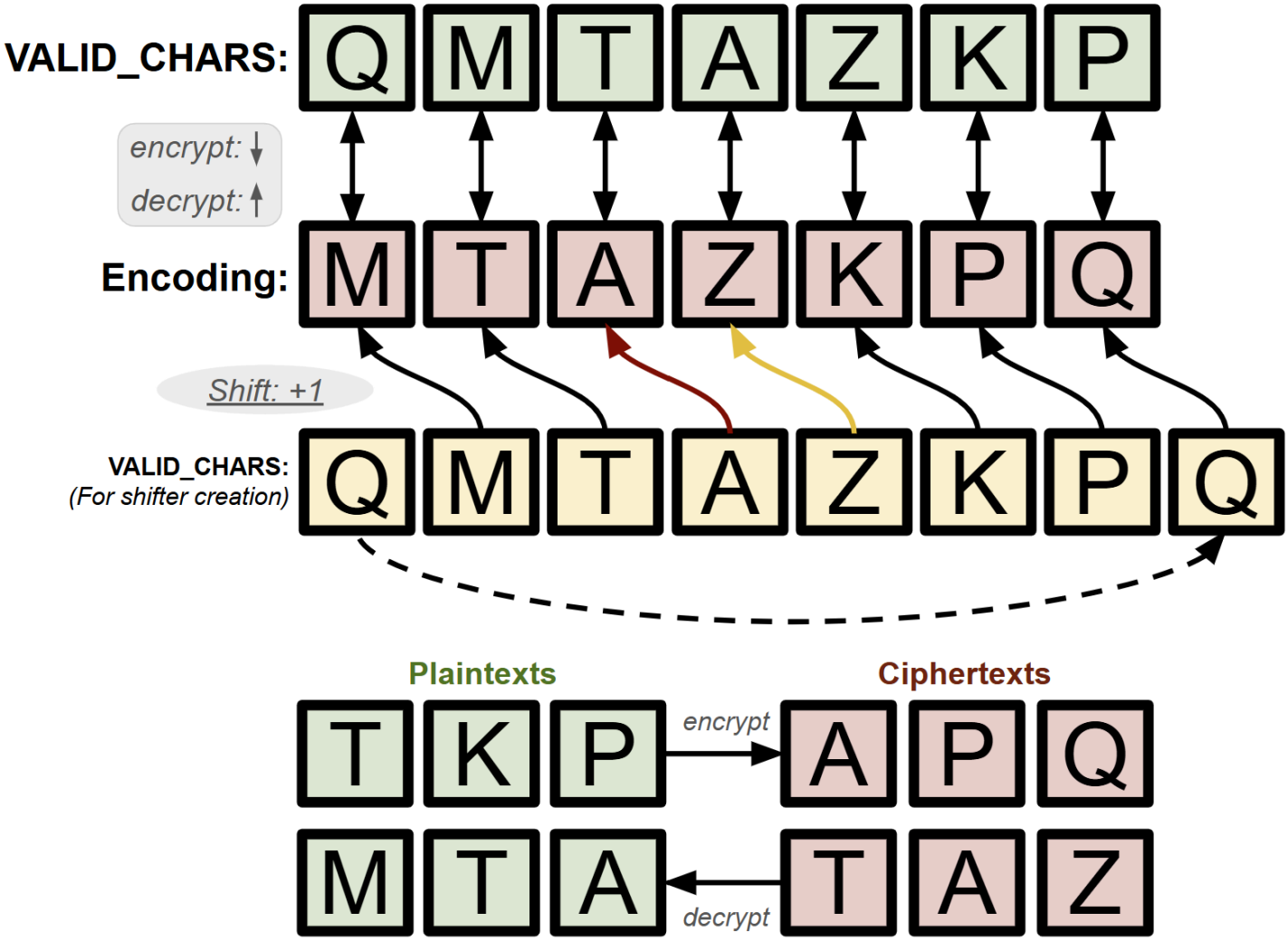
This encryption scheme draws inspiration from the Substitution Cipher, except it involves shifting all valid characters to the left by some provided shift amount.

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, inversing the CaesarShift Cipher is defined as replacing each input character with the corresponding character in the set of valid characters 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 valid characters are "QMTAZKP", `Q` would be replaced with `M`. Additionally, for characters that would shift past the end of the set of valid characters (`P` in this case), the replacement character can be found by looping back around to the front of the valid characters. In this example, `P` would map to `Q`. If the shift was 3 and our valid characters are "QMTAZKP", then `Q` would map to `A`, `M` would map to `Z`, and so on, with Z, K, and P wrapping around to map to Q, M, and T respectively.

Consider the following diagram for a visual explanation:



In this example, our valid characters are the letters "QMTAZKP". To create the encoding, we move the character at the front of the set of valid characters 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 "MTAZKPQ". Notice how the first letter, `Q`, was moved from the front to the back. Similarly to `Substitution`, the mapping of letters is made clearer by placing "QMTAZKP" on top of "MTAZKPQ", such that `Q` is encrypted to `M`, `M` is encrypted to `T`, `T` is encrypted to `A`, `A` is encrypted to `Z`, `Z` is encrypted to `K`, `K` is encrypted to `P`, and `P` is encrypted to `Q`. We go the opposite direction for decryption, so `M` is decrypted to `Q`, `T` is decrypted to `M`, `A` is decrypted to `T`, `Z` is decrypted to `A`, `K` is decrypted to `Z`, `P` is decrypted to `K`, and `Q` is decrypted to `P`.

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`

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

Download your `Substitution.java` and `CaesarShift.java` files and upload them 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.

[NOT GRADED] CaesarKey

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

WARNING: This slide is *NOT* graded

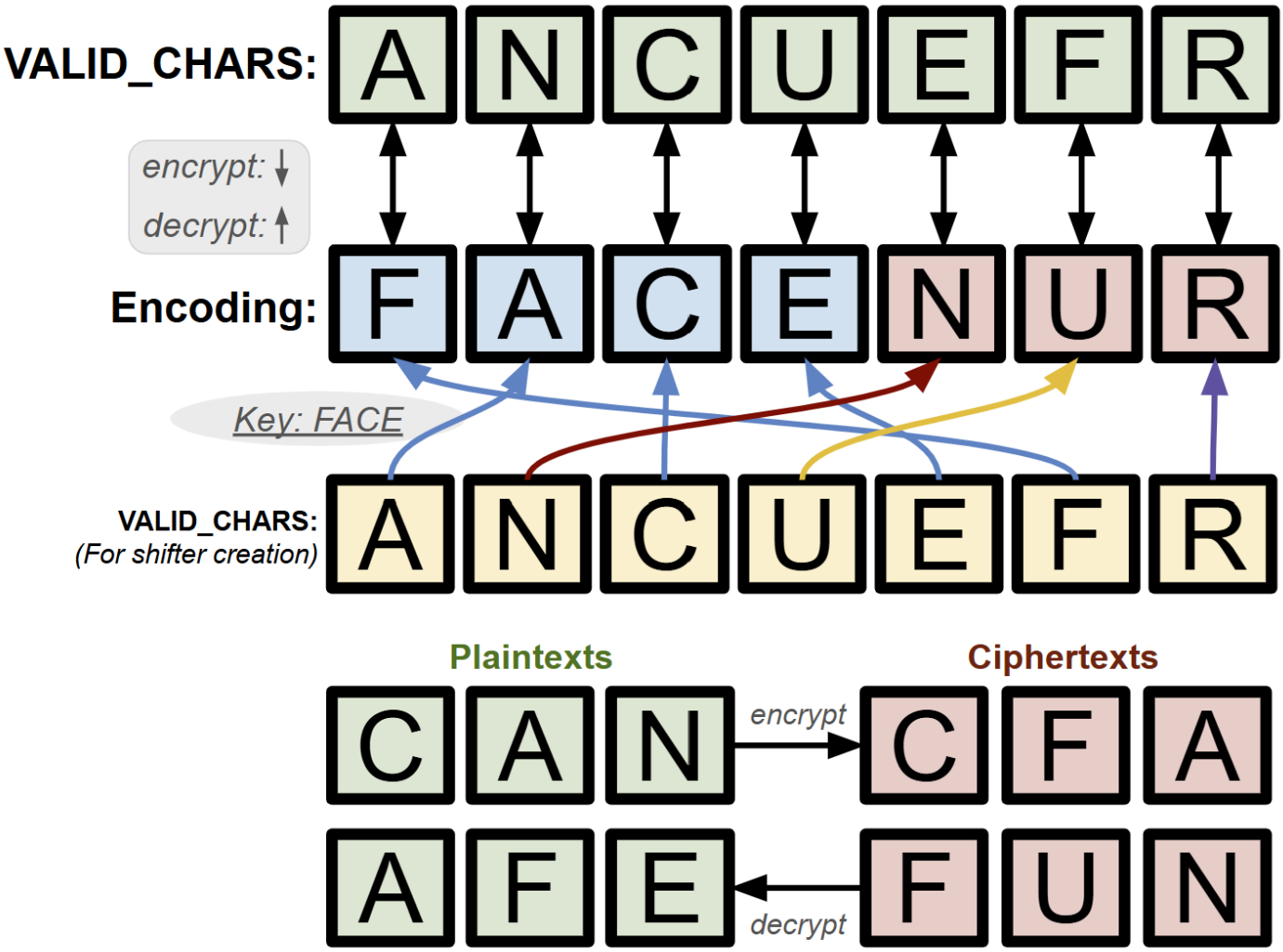
Slide Overview

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 valid characters following normally (minus the characters included in the key). This means that the first character in our valid characters would be replaced by the first character within the key. The second character in the valid characters 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 valid characters.

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 valid characters 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 valid characters in their original order, excluding characters 'F', 'A', 'C', and 'E' as they're already in the encoding. This results in the encoding String "FACENUR".

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`, `N` is encrypted to `A`, `C` is encrypted to `C`, `U` is encrypted to `E`, `E` is encrypted to `N`, `F` is encrypted to `U`, and `R` is encrypted to `R` . We invert this process to decrypt so that `F` decrypts to `A`, `A` decrypts to `B`, `C` decrypts to `C`, `E` decrypts to `U`, `N` decrypts to `E`, `U` decrypts to `F`, and `R` decrypts to `R`.

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 is not a valid character (i.e., is not in `Cipher.VALID_CHARS`).
        - Consider `isCharValid()`!

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`, `CaesarShift.java`, and `CaesarKey.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.

[NOT GRADED] MultiCipher

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

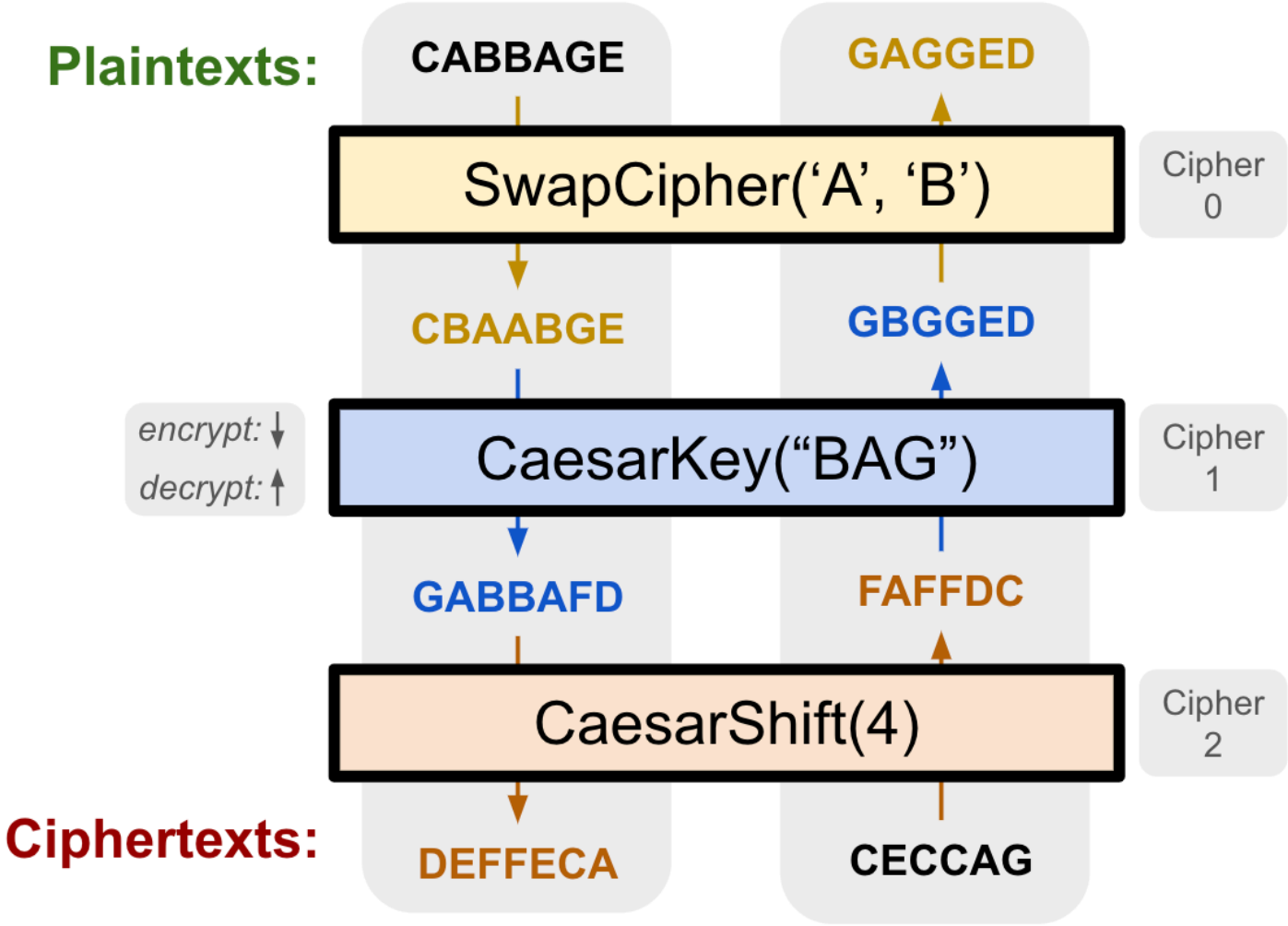
WARNING: This slide is *NOT* graded

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 valid characters are `ABCDEFG`



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

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.

⭐ [GRADED] 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.VALID_CHARS = " !\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`" +
                     "bacdefghijklmnopqrstuvwxyz{|}"
```

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!

```
Ysu(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 `Cipher.VALID_CHARS` to `"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"` 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

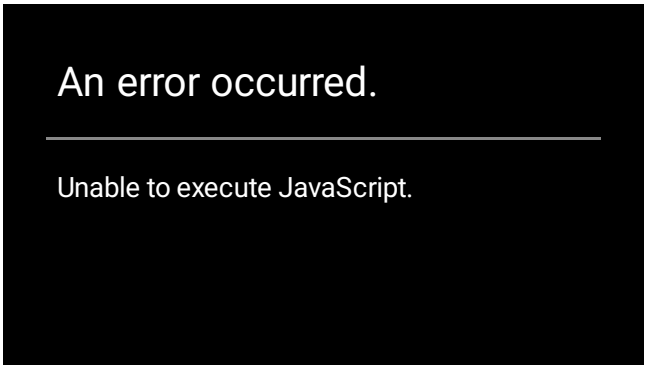
As always, your code should follow all guidelines in the Code Quality Guide and Commenting Guide. In particular, pay attention to these requirements and hints:

- 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.

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)



(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.

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.

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.

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?

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?

Question 6

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

Question 7

What did you struggle with most on this assignment?

Question 8

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

Question 9

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

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?

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!)

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.)