C0

Creative Project 0: Ciphers

Background

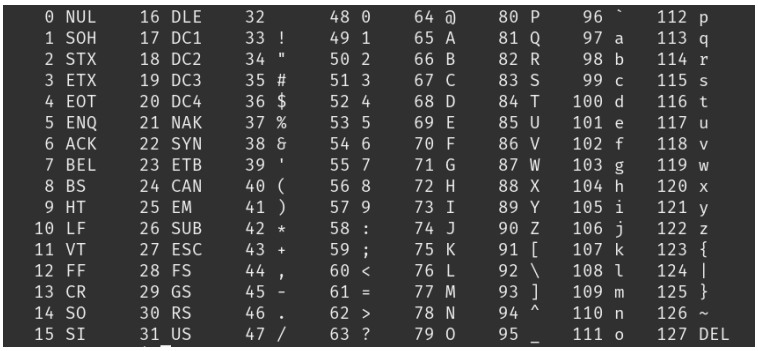
Cryptography (not to be confused with cryptocurrency and blockchain) is a branch of Computer Science and Mathematics concerned with turning input messages (plaintexts) into encrypted ones (ciphertexts) for the purpose of discrete transfer past adversaries. The most modern and secure of these protocols 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:



0	NUL	16	DLE	32		48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

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'` .

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, but you **must revert any changes before marking your assignment.**

```
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
    public static final int MIN_CHAR = (int)('A');

    // The maximum character able to be encrypted/decrypted by any cipher
    public static final int MAX_CHAR = (int)('Z');

    // The total number of characters able to be encrypted/decrypted by any cipher
    // (aka. 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
    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 applied) 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
    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
    //             'encrypt'  - whether or not encryption should occur
    //             'suffix'   - appended to the fileName when creating the output file
    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
    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 `input`
    // Parameters: 'input' - the string to be decrypted
    public abstract String decrypt(String input);
}
```

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

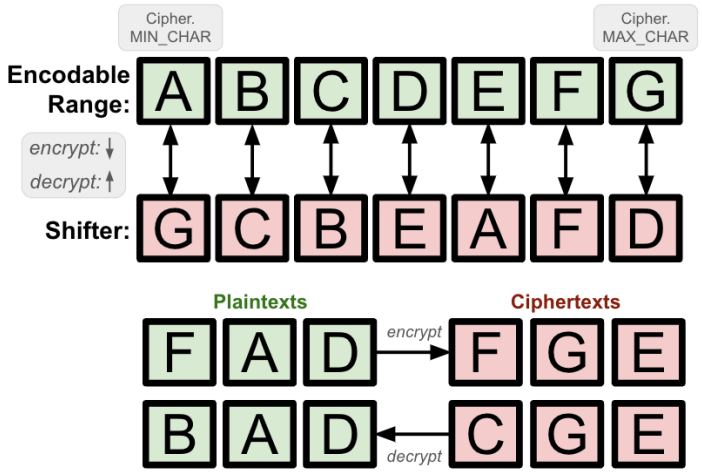
Required Operations

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`

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 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 encryptable range and look at the corresponding columns to see the appropriate character mappings. Consider the following example:



Given the shifter string above, the plaintext "FAD" would be encrypted into "FGE" and the ciphertext "CGE" decrypts into the plaintext "BAD". Below are more in-depth descriptions of the required behavior:

`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 length of the shifter doesn't match the number of characters within our Cipher's encodable range (`Cipher.TOTAL_CHARS`), contains a duplicate character, or 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 length of the shifter doesn't match the number of characters within our Cipher's encodable range (`Cipher.TOTAL_CHARS`), contains a duplicate character, or any individual character falls outside the encodable range (`< Cipher.MIN_CHAR` or `> Cipher.MAX_CHAR`).

`public String encrypt(String input)`

- Applies this Substitution Cipher to the input, returning the result.
  - Applying the Substitution Cipher is defined as replacing each input character with the corresponding character in `shifter` at the same relative position within the ordered encodable range.
  - In general, you may not assume anything about the input other than all characters being in the encodable range.
- Since we're allowing clients to set a shifter after construction, this method should throw an `IllegalStateException` if the shifter is `null` or empty.

`public String decrypt(String input)`

- Inverses this Substitution Cipher on the input, returning the result.
  - Inversing the Substitution Cipher is defined as replacing each input character with the corresponding character in the ordered encodable range at the same relative position within `shifter`.
  - In general, you may not assume anything about the input other than all characters being in the encodable range.
- Since we're allowing clients to set a shifter after construction, this method should throw an `IllegalStateException` if the shifter is `null` or empty.

CaesarShift.java

This encryption scheme draws inspiration from the aforementioned Substitution Cipher, except it involves shifting all encodable characters to the right by some provided shift amount. In the case that shift is 1, any character `c` would be replaced with `(char)(c + 1)` when encrypting. If shift is two, `c` would be replaced with `(char)(c + 2)`. Importantly, if characters map to a value greater than the maximum encryptable character (think shift=1, `(char)(Cipher.MAX_CHAR) -> (char)(Cipher.MAX_CHAR + 1)`) the replacement character should be found by looping back around to the front of the encodable range (so if shift=1, then `(char)(Cipher.MAX_CHAR)` would *actually* map to `(char)(Cipher.MIN_CHAR)`).

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

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

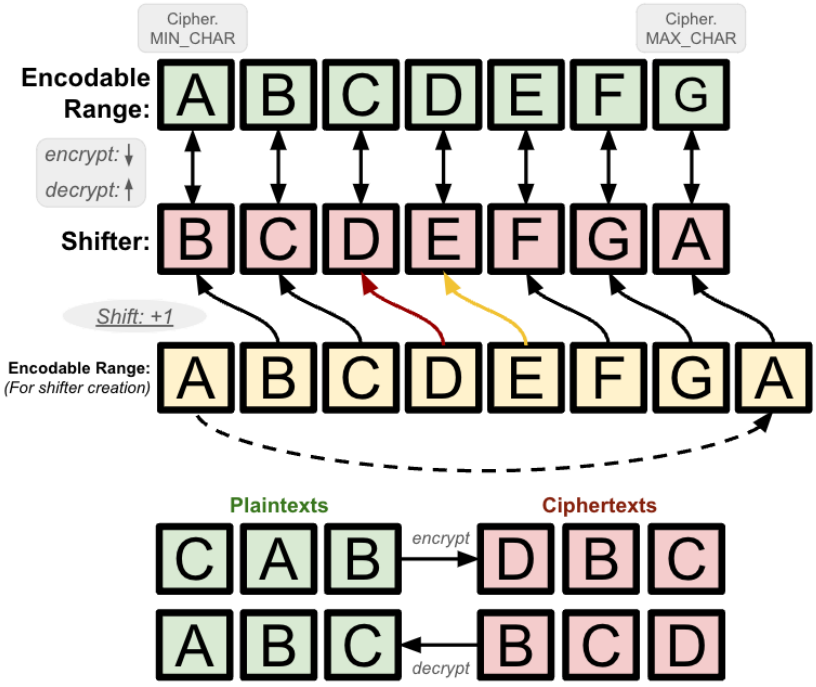
Where we add shift to c, get the displacement of the result by subtracting `Cipher.MIN_CHAR`, mod it by `Cipher.TOTAL_CHARS` in the event that we go past the maximum encryptable character, and re-add the new displacement to `Cipher.MIN_CHAR` to get the encrypted

result. Similarly, we can define the inverse expression:

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

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

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



Note that this diagram outlines the process of creating shifter in which we physically move the character at the front of the encodable range to the end (and in doing so shift all other characters to the left). As the shift value above is just one, this process is repeated one time. If the shift value was two, we'd do it twice.

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

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

Your solution should pick one of the two above approaches to implement such that plaintext characters are shifted in the encodable range by a given amount. Below are more in-depth descriptions of the required behavior:

`public CaesarShift(int shift)`

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

`public String encrypt(String input)`

- Applies the CaesarShift Cipher on the input, returning the result.
  - 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.
  - In general, you may not assume anything about the input other than all characters being in the encodable range.

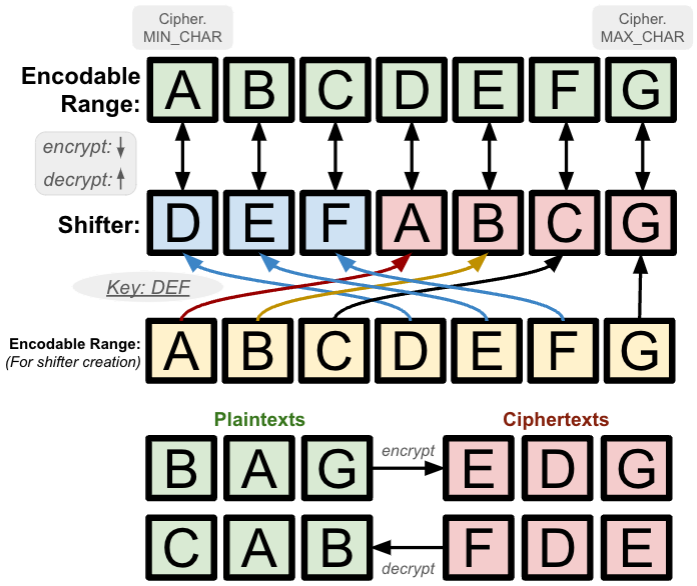
`public String decrypt(String input)`

- Inverses the CaesarShift Cipher on the input, returning the result
  - Inversing the CaesarShift Cipher is defined as replacing each input character with the corresponding character in the encodable range at the same relative position within `shifter`. This `shifter` should be created by moving all characters within the range to the left `shift` times, moving the value at the front to the end each time.
  - In general, you may not assume anything about the input other than all characters being in the encodable range.

CaesarKey.java

Much like the CaesarShift, the CaesarKey scheme also builds off of the base Substitution Cipher. Instead, 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 lowest unused character within the encodable range. Consider the following diagram for a visual explanation:



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

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)

Below are more in-depth descriptions of the required behavior:

`public CaesarKey(String key)`

- Constructs a new CaesarKey with the provided key value
- This constructor should throw an `IllegalArgumentException` in the case that the key is empty, it contains a character outside our range of valid characters, or it contains any duplicate characters

`public String encrypt(String input)`

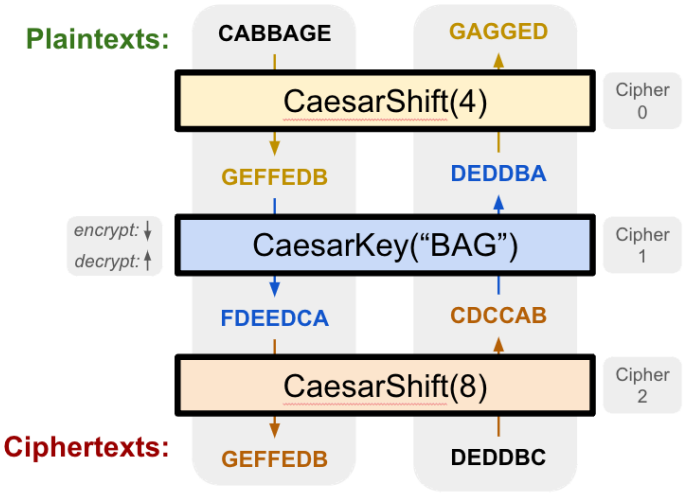
- Applies the CaesarKey Cipher on the input, returning the result.
  - Applying the CaesarKey Cipher is defined as replacing each input character with the corresponding character in `shifter` at the same relative position. This `shifter` should be created by appending the rest of the encodable range to the key in order.
  - In general, you may not assume anything about the input other than all characters being in the encodable range.

`public String decrypt(String input)`

- Inverses the CaesarKey Cipher on the input, returning the result.
  - Inversing the CaesarKey Cipher is defined as replacing each input character with the corresponding character in the ordered encodable range at the same relative position within `shifter`. This `shifter` should be created by appending the key to the rest of the encodable range in order.
  - In general, you may not assume anything about the input other than all characters being in the encodable range.

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 really 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 all the way through the entire list results in the final encrypted string. Decrypting would then involve the opposite of this: starting with the last cipher and working backward through the cipher list until the plaintext is revealed. Below is a diagram of these processes, passing inputs through each layer of the cipher list. Consider the following diagram demonstrating the process of encrypting/decrypting a MultiCipher consisting of 3 internal ciphers: a CaesarShift of 4, a CaesarKey with key "BAG", and a CaesarShift of 8.



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

On the right in the above example, we start at the ciphertext: `DEDDBC` hoping to decrypt it. Decrypting this through the last layer (a CaesarShift of 8) results in the intermediary still-encrypted message `CDCCAB`. This intermediary value is then used as input to the next layer (a CaesarKey with key "BAG") resulting in the second intermediary still-encrypted message `DEDDBA`. This process is repeated one last time, resulting in the final plaintext of `GAGGED`.

This is what you'll be implementing in this class: given a list of ciphers, apply them in order to encrypt or in reverse order to decrypt a given message. Below are more in-depth descriptions of the required behavior:

`public MultiCipher(List<Cipher> ciphers)`

- Constructs a new MultiCipher with the provided List of Ciphers
- Should throw an `IllegalArgumentException` if the given list is null

`public String encrypt(String input)`

- Applies the MultiCipher Cipher on the input, returning the result.
  - Applying the MultiCipher Cipher is defined as encrypting input using the first cipher, then taking the result and using that as input to the second cipher's encrypt method. This should repeat for every cipher until the last one within the MultiCipher's cipher list.

`public String decrypt(String input)`

- Inverses the MultiCipher Cipher on the input, returning the result.
  - Inversing the MultiCipher Cipher is defined as decrypting input using the last cipher, then taking the result and using that as input to the second-to-last cipher's decrypt method. This should repeat for every cipher until the first one within the MultiCipher's cipher list.

Use Your Ciphers!

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

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

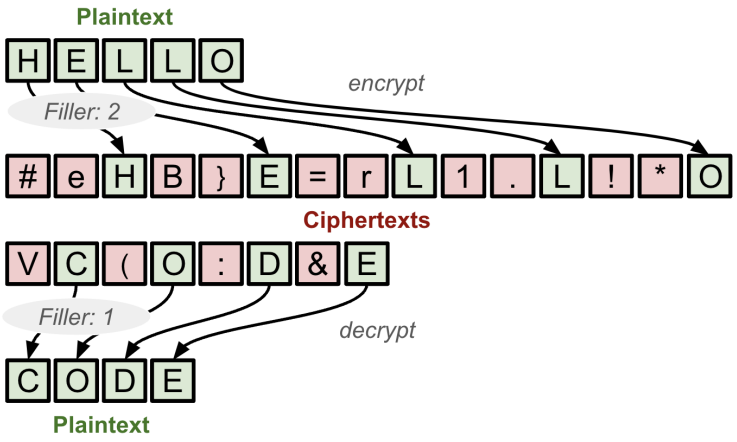
Creative Portion

For the creative portion of this assignment, you'll be implementing another cipher that interests you!

Concealment

This scheme involves confusing any potential adversary with a jumble of random characters, placing the original message at specific locations within the encrypted message. Although this can manifest a variety of ways (the character that starts every sentence in a paragraph, the left-most words on a physical page of paper, etc.), in this class you'll be placing the characters from the original message after a specified number of random "filler" characters. For example, if filler is 2 then you would construct your ciphertext by placing 2 random characters from the encodable range within your ciphertext, followed by the first character from the plaintext. Then another 2 random characters followed by the second character from the plaintext. Repeat this process until you run out of characters in the plaintext and you have your encrypted message!

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



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

Your solution should contain the following behavior:

`public Concealment(int filler)`

- Constructs a new Concealment with the provided filler value
- An `IllegalArgumentException` should be thrown in the case that filler is less than or equal to zero.

`public String encrypt(String input)`

- Applies the Concealment Cipher on the input, returning the result.
  - Applying the Concealment Cipher is defined as placing `filler` random characters before each character from `input`.

`public String decrypt(String input)`

- Inverses the Concealment Cipher on the input, returning the result.
  - Inversing the Concealment Cipher is defined as concatenating together each `filler + 1` th character from `input`.

5. Your choice!

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

Try your new Cipher!

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

Testing

You are welcome to use the provided `Client.java` to test and debug your cipher implementations. To do so, make sure to change the `CHOSEN_CIPHER` constant to the cipher you're testing before hitting run. You are also encouraged to modify the constants in `Cipher.java` such that a smaller subset of characters are used by your cipher. However, these constants must be reverted before marking and your implementation should work regardless of what values are assigned to `Cipher.MIN_VALUE` and `Cipher.MAX_VALUE`.

🔍 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.