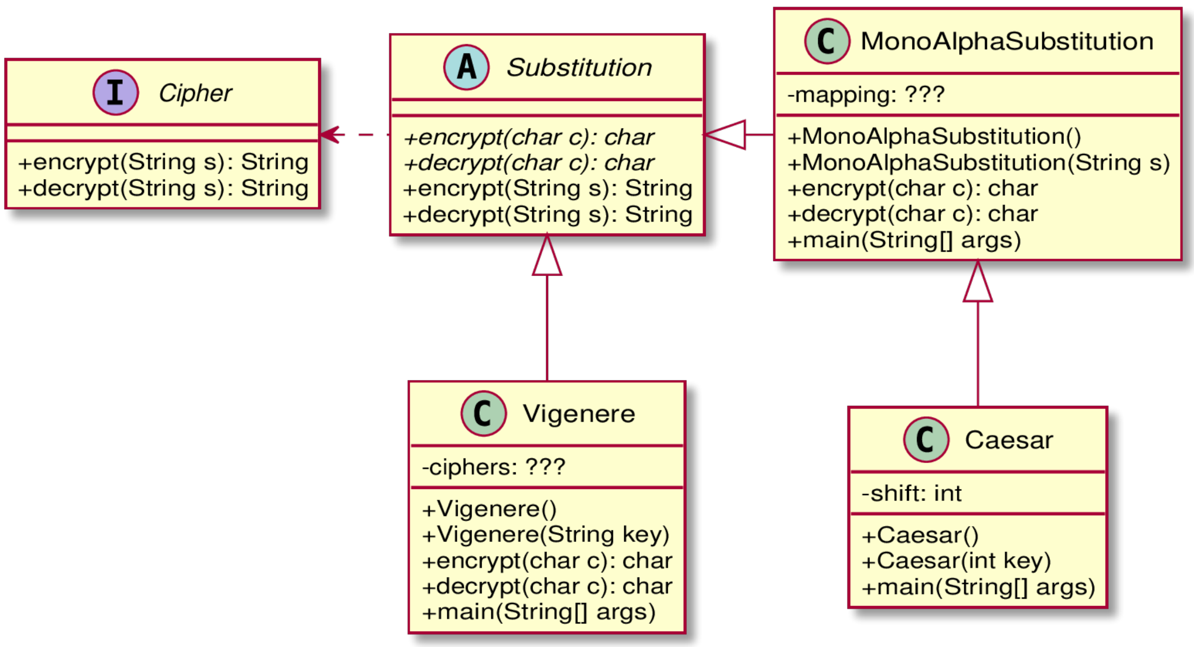
Le chiffre indéchiffrable

In this coursework you will implement a whole bunch of simple substitution ciphers in an object oriented fashion, as depicted in the (incomplete) UML diagram below. The interface `Cipher` is given as part of the exercise and should not be changed in any way.



Part 1: Substitutions

Substitution ciphers de/encrypt one letter (or other small unit) at a time. Write an **`abstract`** class called `Substitution` that implements the given interface `Cipher`. It should do so by implementing the methods `encrypt` and `decrypt` that take and return strings, using two **`abstract`** methods by the same names but which take and return individual characters.

To be clear, this class should have two **`abstract`** methods,

```java
public abstract char encrypt(char c);
public abstract char decrypt(char c);
```

which are left unimplemented, and two methods

```java
public String encrypt(String plaintext);
public String decrypt(String cryptotext);
```

that have concrete implementations. The first one should call the abstract `encrypt` method one character at a time and return the so constructed string, and similarly for the decrypt method.

Part 2: MonoAlphaSubstitution

Monoalphabetical substitution ciphers are ciphers that de/encrypt characters according to *one fixed translation table* (the alphabet). Your task is to write a concrete subclass of `Substitution` called `MonoAlphaSubstitution` that represents such ciphers. This of course means that you will have to overload the two abstract methods `encrypt` and `decrypt` above with concrete implementations.

How the translation table is stored internally is up to you but the class should offer at least two constructors:

1. a default constructor (one without arguments), which results in the trivial identity substitution (‘a’ → ‘a’, ‘b’ → ‘b’,…) where every letter is mapped to itself.
2. a constructor that takes a `String` and interprets it as a mapping where every character at an odd position is the encoding of the one directly before it. The first character (at position 0) will be encryptd as the second (at position 1), the third as the fourth and so on. This way the string `"ABBCCD"`, should be interpreted as ‘A’ → ‘B’, ‘B’ → ‘C’, ‘C’ → ‘D’ and every other letter to itself.

The methods `encrypt` and `decrypt` (that both take a **`char`** and return a **`char`**) should then be implemented so that they translate a plaintext character into its cryptotext variant (and back, respectively) according to the translation table set up in the constructor method.

Your class should have a `main` method, that allows to encrypt/decrypt text and accepts exactly three command line arguments, as follows.

1. `"encrypt"` or `"decrypt"`, to indicate which direction you want to translate
2. The string defining the translation, as for the `String` constructor
3. the text to en/decrypt.

All user input must be read from command line arguments. Do not use `Scanner` or other means for getting user input! The *only* output printed by the program should be the output of the cipher. See below for example outputs. Your program needs to print the *exact* same output to be considered correct.

```
$> java MonoAlphaSubstitution encrypt akbjcidhegffgehdicjbka "Life is wasted on the living."
Lcfg cs wkstgh on tdg lcvcne.

$> java MonoAlphaSubstitution decrypt akbjcidhegffgehdicjbka "Lcfg cs wkstgh on tdg lcvcne."
Life is wasted on the living.

$> java MonoAlphaSubstitution dec akbjcidhegffgehdicjbka "Life is wasted on the living."
```

```
The first parameter must be "encrypt" or "decrypt"!
Usage: java MonoAlphaSubstitution encrypt key "cipher text"

$> java MonoAlphaSubstitution decrypt
Too few parameters!
Usage: java MonoAlphaSubstitution encrypt key "cipher text"

$> java MonoAlphaSubstitution encrypt akbjcidhegffgehdicjbka Life is wasted on the living.
Too many parameters!
Usage: java MonoAlphaSubstitution encrypt key "cipher text"

$> java MonoAlphaSubstitution encrypt akbjcidhegffgehdicjbka Life
Lcfg
```

Part 3: Caesar

You know this one already from the first assignment. The Caesar cipher is a special case of a monoalphabetical substitution, where each character maps to the one k position later in the alphabet, for a fixed shift k which is the secret key.

The task here is to write a subclass `Caesar` of `MonoAlphaSubstitution`. There should be at least a default constructor (leading to the identity again) and another constructor that takes an integer shift, according to which all en/decrypt methods operate.

Objects of type `Caesar` should be ciphers that translate both lower case and capital letters according to the shift specified at instantiation. Every other characters should be translated to itself.

Your class should have a `main` method that allows to encrypt/decrypt text, and that works similar to the one described for `Substitution`. See below for example outputs. Your program needs to print the *exact* same output to be considered correct.

```
$> java Caesar encrypt 3 "The ships hung in the sky in much the same way that bricks don't."
Wkh vklsv kxqj lq wkh vnb lq pxfk wkh vdph zdb wkdw eulfnv grq'w.

$> java Caesar decrypt 3 "Wkh vklsv kxqj lq wkh vnb lq pxfk wkh vdph zdb wkdw eulfnv grq'w."
The ships hung in the sky in much the same way that bricks don't.

$> java Caesar encrypt 13 The ships hung in the sky in much the same way that bricks don't.
Too many parameters!
Usage: java Caesar encrypt n "cipher text"

$> java Caesar encrypt 3
Too few parameters!
Usage: java Caesar encrypt n "cipher text"

$> java Caesar enc 13 "Life is wasted on the living."
The first parameter must be "encrypt" or "decrypt"!
Usage: java Caesar encrypt n "cipher text"
```

Part 4: Vigenère

The Vigenère cipher is a 16th century encryption scheme first described by Giovan Battista Bellaso and later misattributed to French diplomat Blaise de Vigenère. It addresses the main weakness of earlier ciphers which are easily broken with a frequency analysis.

The idea behind the Vigenère cipher is to disguise the plaintext letter frequencies by not encoding each letter according to the same translation table. Instead, it uses one translation table for each letter position in the plaintext. Such ciphers are called *polyalphabetic* substitutions because there is more than one mapping (alphabet) involved. This way, the most common letter 'e' for instance translates to different letters depending on where in the plaintext it occurs. In fact this cipher has resisted crypto-analytic attacks for well over two centuries, which earned it the nick name *le chiffre indéchiffrable*.

What makes this cipher particularly user friendly is the way one specifies the secret key, i.e. the substitutions to use at each position. This is done by arranging several Caesar ciphers with different shifts based on some key *word* which is easy to remember for all communicating parties.

Suppose for example that the keyword is "COMPONETWOTWO". Then the first character of the plaintext would be translated according to a Caesar cipher with shift 2, as `'C'- 'A'== 2`, the first letter 'C' of the keyword is two letters after 'A' in the Roman alphabet. The second character will be translated using a Caesar cipher with shift `14 == 'O'- 'A'` and so on. If the plaintext is longer than the key one continues at the start of the key word, so that the 14th position will again use the Caesar shift 2. If you encrypt the phrase "fun fun fun" with the key above, you should get "hiz thr big". Notice that the letter 'f' maps to a different letter each time! Also notice that the second 'f' maps to 't' because it is shifted by 14, corresponding to `'O'` at the same position in the keyphrase.

Your task is to implement this cipher in a class called `Vigenere` which should be a concrete subclass of `Substitution` (after all, this is still a character-by-character encoding scheme). This means that again, you will have to overload the abstract methods `encrypt` and `decrypt` with **`char`**-parameter with concrete implementations but now consecutive calls to the same method may give different results, so you will need to store the position your are at somehow in the object. You may freely use all other classes that you have written.

The `Vigenere` class should again have at least two constructors:

- a default constructor (no arguments), which results in the identity as before
- one that takes a single `String` argument, the key word used in the cipher.

You may assume that the key word consists only of capital letters without spaces or punctuation.

Your class should have a `main` method that allows to encrypt/decrypt text, and that works similar to the ones described above. See below for example outputs. Your program needs to print the *exact* same output to be considered correct.

```
$> java Vigenere encrypt COMPONETWOTWO "fun fun fun"
hiz thr big

$> java Vigenere decrypt COMPONETWOTWO "hiz thr big"
fun fun fun

$> java Vigenere enc COMPONETWOTWO "fun fun fun"
The first parameter must be "encrypt" or "decrypt"!
Usage: java Vigenere encrypt key "cipher text"

$> java Vigenere decrypt COMPONETWOTWO
Too few parameters!
Usage: java Vigenere encrypt key "cipher text"

$> java Vigenere encrypt COMPONETWOTWO fun fun fun
Too many parameters!
Usage: java Vigenere encrypt key "cipher text"

$> java Vigenere encrypt COMPONETWOTWO fun
hiz
```

Hints

1. the `String` methods `toCharArray` and `charAt` may be useful.
2. For `MonoAlphaSubstitution` your code should cover both lower and upper case letters separately and not have special handling to unify the two cases. This way the String "AxaX" would represent a translation 'A' $\longrightarrow$ 'x', 'a' $\longrightarrow$ 'X'. Notice that this substitution cannot be uniquely reversed, as both 'A' and 'x' map to 'x', and would therefore be useless for en/decryption. We will not test your code with such translations.
3. For `MonoAlphaSubstitution` you do not need to use `HashMap` or other collections. If you want to use that, that's fine, but the exercise can be solved by storing the substitutions otherwise, for example as 2-dim character array.
4. For `Caesar` you could re-use your code for A1 if you want, but there is a much nicer OOP solution here given that Caesar is a special case of `MonoAlphaSubstitution` (with simple translations) and that this superclass already implements the en/decrypt methods. Play with constructors?
5. For `Vigenere`, I would recommend you use an array of `Cipher`s to keep around several `Caesar` objects. Also notice from the examples that the position in the keyphrase determines the shift. This is different and simpler than what some other implementations online do, namely only "move fowrard" in the key if an actual character is translated. You should not add such extra logic here.

6. In all parts, it is perfectly fine to re-use your own code or helper methods from the Java standard library. It is also fine to add your own helper methods to any of the classes in addition to what the exercise requires.
7. Do not use `Scanner` or any other means to read user input. All input should be read from command line arguments only!
8. Make sure that your code compiles and do not only focus on the difficult bits! There are points to be gained for having the prescribed methods. You will not get any points for code that does not compile.
9. All concrete classes above should be usable simply by instantiating and calling the encrypt/decrypt methods. Their behaviour should not depend on any logic inside their respective main methods. Make sure that the following code (for example in the main method of another utility class) would compile and execute successfully (but not print anything):

```
String testInput = "this is a Test!";
String output;

Cipher m = new MonoAlphaSubstitution("ABBCCA");
Cipher c = new Caesar(13);
Ciper v = new Vigenere("COMPONETWOTWO");
output = c.decrypt(testInput)
         + c.decrypt(testInput)
         + v.decrypt(testInput);
```

Submission

Submit you solution using `submit50` just like the lab exercises.

```
submit50 liv-ac-uk/comp122/2021/problems/vigenere
```

To check that your submission is in the correct format you can run `check50` on the same slug.

Fine Print

Submissions are subject to UoL's Code of Practice on Assessment and the usual late penalties apply. This means you can still submit until 120 hours past the deadline, and will incur a -5% penalty for each 24 hour period immediately following the deadline. Submissions after that will not be considered. You can submit multiple times and only the latest version (and it's submission time) will be used for grading.

If you require an extension due to extenuating circumstances please get in touch with the CS student office (csstudy@liv.ac.uk) *before* the submission deadline. We will not grant any extensions afterwards. If you are granted an extension you cannot in addition submit late after your personal deadline.

The results will be made available on Canvas and we aim for a turn-around time of three weeks[1].

[1]The reason it may take longer than you'd expect is that we are not allowed to publish feedback until after all submissions have been handed in.