

# 1 Verschlüsselung nach Caesar

Die Verschlüsselung nach Caesar beruht auf einem Geheimtextalphabet, das um eine bestimmte Stellenanzahl gegenüber dem Klartextalphabet verschoben ist, im unten abgebildeten Beispiel um drei Stellen.

Klartextalphabet	a b c d e f g h i j k l m n o p q r s t u v w x y z
Geheimtextalphabet	D E F G H I J K L M N O P Q R S T U V W X Y Z A B C

Klartext	g e h e i m w o r t
Geheimtext	J H K H L P Z R U W

In der Kryptographie ist es üblich, das Klartextalphabet in Kleinbuchstaben, das Geheimtextalphabet in Großbuchstaben zu schreiben. Dies erleichtert dem Leser, zwischen beiden zu unterscheiden. Auch die ursprüngliche Botschaft, der Klartext, wird klein geschrieben, und die verschlüsselte Botschaft, der Geheimtext, groß geschrieben.

## Übung 1.1:

Codiere und decodiere nach dem obigen Verfahren die Worte im Klartext

- informatikunterricht
- geheimwort

## Übung 1.2:

Gib deinem Nachbarn einen verschlüsselten kompletten Satz. Dein Nachbar soll diesen wieder decodieren.

## Übung 1.3:

Überlege dir, welche Probleme bei der Verschlüsselung eines Satzes mit Hilfe dieses Codierungsverfahrens auftreten können.

## Übung 1.4:

Erstelle eine neue Codiertabelle, so dass das Geheimtextalphabet um eine andere Stellenzahl z.B. 11 gegenüber dem Klartextalphabet verschoben wird. Löse mit deiner neuen Codiertabelle die Übungen 1.1 und 1.2.

## Bemerkung:

Bei der Erstellung unseres Programms in Java, in dem das Codierungsverfahren nach Caesar simuliert, werden wir zunächst alle Leerzeichen, Sonderzeichen

und Satzzeichen nicht berücksichtigen. Dies erleichtert die Erstellung des Quellcodes.

## 1.1 Die Klasse Codewort

### 1.1.1 Der ASCII-Code eines Buchstabens

Übung 1.5:

Informiere dich über den ASCII-Code.

Übung 1.6:

Erstelle den ASCII-Code des Alphabets

- a) in Kleinbuchstaben
- b) in Großbuchstaben

in einer Tabelle schriftlich.

Übung 1.7:

Versuche in **BlueJ** ein Programm *Caesar01a* zu erstellen, das dir den ASCII-Code eines einzelnen Buchstaben wider gibt. Bezeichne diese Klasse mit *Caesar*. Im weiteren Text werden dir einige Hilfestellungen gegeben.

Das in Übung 1.7 zu erstellende Programm muss folgende Aufgaben bewältigen:

1. Es muss den Klartext abfragen: *holeKlartext()*.
2. Es muss diesen Buchstaben in den ASCII-Code umwandeln: *codiereWort()*.
3. Es muss den entsprechenden ASCII-Wert wieder zurückgeben: *gibGeheimtext()*.

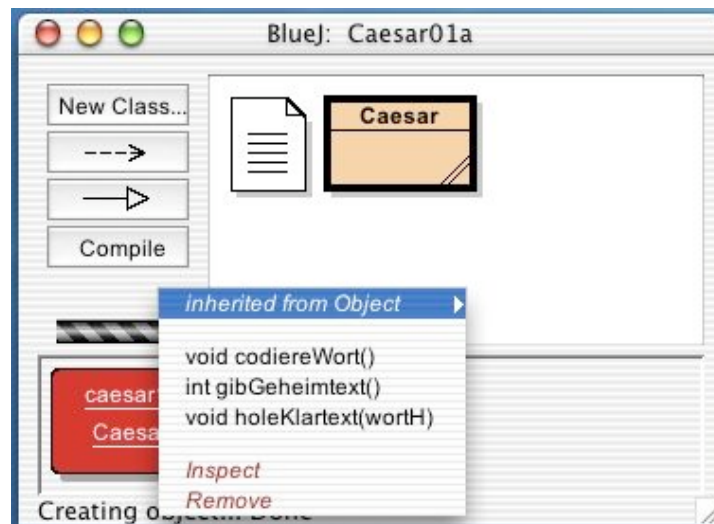


Abbildung 1.1: Die Klasse *Caesar* des Projekts *Caesar01a*

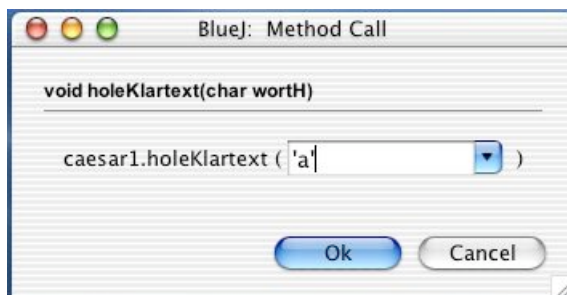


Abbildung 1.2: Die Methode *holeKlartext()*

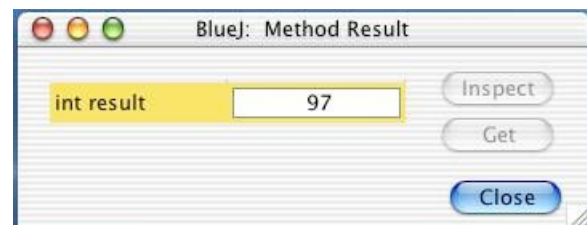


Abbildung 1.3: Die Methode *gibGeheimtext()*

Die Methoden *holeKlartext()* und *gibGeheimtext()* dürften keine Probleme darstellen. Notfalls schaue dir das Programm *Bankkonto* noch einmal genau an. Die Methode *codiereWort()* soll einen Buchstaben in den entsprechenden ASCII-Code umwandeln. In Java erfolgt dieses mit Hilfe der Anweisung

$$x = (\text{int}) \text{wort};$$

wobei die Variable *wort* den entsprechenden Buchstaben enthält und die Variable *x* der zugehörigen ASCII-Code geschrieben wird.

### Übung 1.8:

Schreibe in die Methode *codiereWort()* die Anweisung `wort = Character.toUpperCase(wort)` bzw. `wort = Character.toLowerCase(wort)`.

Beschreibe die Wirkungsweise dieser beiden Anweisungen. Wozu könnte man diese im späteren Codierungsverfahren verwenden?

### Übung 1.9:

Erstelle nun in **BlueJ** ein Programm *Caesar01b*, das aus dem ASCII-Code (Zahl) wieder den entsprechenden Buchstaben ermittelt. Untersuche auch hier die beiden Anweisungen

`wort = Character.toUpperCase(wort)` bzw.

`wort = Character.toLowerCase(wort)`.

#### 1.1.2 Ein Buchstabe wird verschlüsselt

Nun können wir für jeden Klartextbuchstaben den entsprechenden Geheimbuchstaben aus der obigen Tabelle erstellen. Die Idee der Verschlüsselung ist also, den Buchstaben in seinen ASCII-Code umzuwandeln, diese Zahl um 3 zu vergrößern und dann daraus wieder den zugehörigen Buchstaben zu ermitteln.

### Übung 1.10:

Erstelle nun in **BlueJ** ein Programm *Caesar01c*, das einen Buchstaben um denjenigen ersetzt, der im Alphabet um drei Stellen weiter hinten steht. Überprüfe mit deinem Programm die Codiertabelle aus Abbildung 1.1.

### Übung 1.11:

Untersuche speziell die Buchstaben x, y, z und X, Y, Z. Formuliere schriftlich, warum bei diesen Buchstaben das Programm versagt.

#### Bemerkung:

Die Variable *x*, die den entsprechenden ASCII-Wert enthält, wird nur noch in der Methode *codiereWort()* benötigt. Deshalb ist es sinnvoll, die Deklaration dieser Variablen auch in diese Methode zu verlegen. Somit verschwindet diese Variable auch im Inspektor.

Wie bereits erwähnt, ist es üblich, den Klartext in Kleinbuchstaben und den Geheimtext in Großbuchstaben darzustellen. Die entsprechenden Anweisungen in Java kennen wir bereits, um einen Buchstaben in einen Groß- bzw. Kleinbuchstaben umzuwandeln.

### Übung 1.12:

Wir werden die Tabelle aus Übung 1.6 b) (Großbuchstaben) erweitern.

- a) In der dritten Zeile vermindere den ASCII-Wert um 65.
- b) In der vierten Zeile vergrößere jeden Wert um 3.
- c) In der fünften Zeile wende die Funktion modulo 26 auf jeden Wert an.

- d) In der sechsten Zeile vergrößere diesen Wert um 65.
- e) In der siebten Zeile schreibe den zugehörigen Buchstaben.

### Übung 1.13:

Erstelle nun in **BlueJ** ein Programm *Caesar01d*, das in seiner Methode *codiereWort()* zuerst alle Buchstaben in Großbuchstaben umwandelt und dann die Schritte aus Übung 1.12 als Anweisungen enthält. Überprüfe, ob das Programm die Codiertabelle aus Abbildung 1.1 korrekt erstellt. Überprüfe auch speziell die Buchstaben x, y, z.

### Übung 1.14:

Nun soll ein codierter Buchstabe wieder decodiert werden. Überlege dir, welche Änderungen im Quellcode von *Caesar01d* vorzunehmen sind. Überprüfe deine Überlegungen.

## 1.1.3 Ein Wort wird verschlüsselt

Ein ganzes Wort zu verschlüsseln ist nun nicht mehr schwierig. Die Variable *wort* wird nun in den Typ *String* umgeändert, da diese jetzt eine Zeichenkette aufnehmen muss.

Die Verschiebung um drei Stellen gestaltet sich etwas komplizierter. Zuerst wird die Zeichenkette mit *toUpperCase()* in Großbuchstaben dargestellt. Dann muss deren Länge mit *length()* ermittelt werden. Nun wird in einer *for*-Schleife die Zeichenkette durchlaufen, wobei die Methode *charAt()* das jeweilige Zeichen an der *i*-ten Stelle ermittelt. Der dort stehende Klarnbuchstabe wird nach den uns bekannten Anweisungen codiert.

Es ist sinnvoll, alle Geheimbuchstaben nach und nach in einen *StringBuffer* zu sammeln. Ein *StringBuffer* wirkt wie ein Puffer, der einen noch nicht vollendeten String aufnehmen und bearbeiten kann. Er besitzt zum Anhängen weiterer Zeichen die Methode *append()*. Ist der vorläufige String nun vollendet, also das Geheimwort fertig, so wird dieser *StringBuffer* mit der Methode *toString()* in einen String zurückverwandelt und in der Variable *wort* gespeichert.

Der unten dargestellte Quellcode zeigt die oben beschriebenen Anweisungen:

```
wortBuffer = new StringBuffer();
wort = wort.toUpperCase();
wortLaenge = wort.length();
for (int i = 0; i < wortLaenge; i++) {
    ch = wort.charAt(i);
    x = (int) ch;
```

```

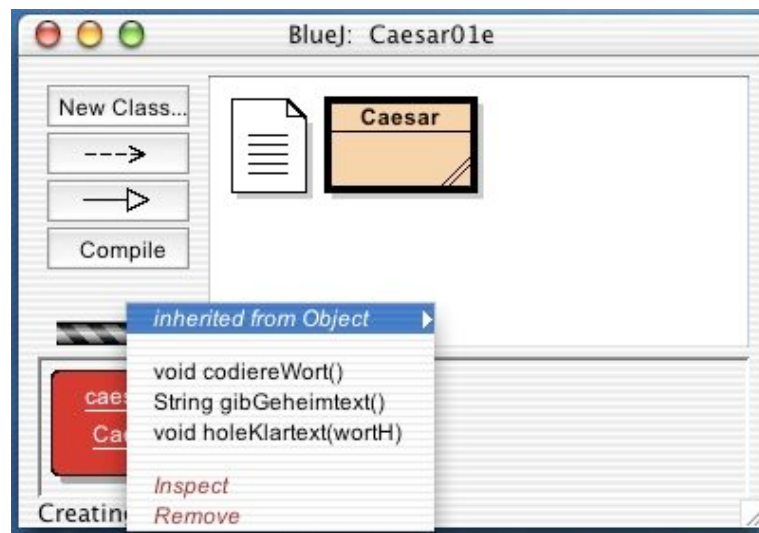
    x = x - 65;
    x = x + 3;
    x = x % 26;
    x = x + 65;
    wortBuffer.append((char) x);
}
wort = wortBuffer.toString();

```

**Abbildung 1.4:** Quellcode der Methode *codiereWort()*

### Übung 1.15:

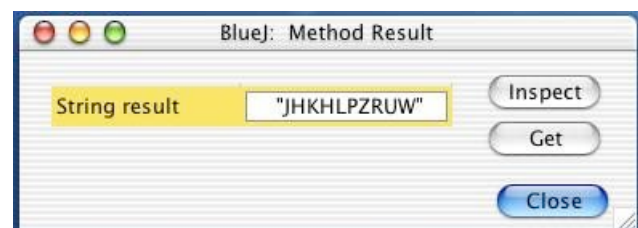
Erstelle nun in **BlueJ** ein Programm *Caesar01e*, das ein komplettes Wort verschlüsseln kann.



**Abbildung 1.5:** Die Klasse *CodeWort* des Projekts *Caesar01e*



**Abbildung 1.6:** Die Methode *holeKlartext()*



**Abbildung 1.7:** Die Methode *gibGeheimtext()*

### Übung 1.16:

Nun soll ein codiertes Wort wieder decodiert werden. Überlege dir, welche Änderungen im Quellcode von *Caesar01e* vorzunehmen sind. Überprüfe deine Überlegungen.

**Bemerkung:**

1. Ein Problem stellen die Umlaute, Leerzeichen und sonstigen Sonderzeichen dar. Diese werden in *Caesar01* nicht korrekt decodiert.
2. Hackenberg Stefan (2005/06: 11. Klasse a) verwendete in der Methode *codiere()* anstelle des *StringBuffer* *wortBuffer* die Anweisung  
`geheimwort += (char) x.`  
Diese Lösung zeigt das Projekt *Caesar01e2*.

## 1.2 Die grafische Oberfläche

Nachdem jetzt die Codierung funktioniert, können wir uns nun der Benutzeroberfläche widmen. Diese werde ich in Java-Swing gestalten. Swing bietet mehr Gestaltungsmöglichkeiten als das AWT (Advanced Windowing Toolkit).

Die Gestaltungsmöglichkeiten mit Swing werden im *Handbuch der Java-Programmierung* von Guido Krüger und in ähnlichen Büchern beschrieben.

In diesem Kapitel werde ich die in Abbildung 1.8 dargestellte Benutzeroberfläche beschreiben. Um die Oberfläche übersichtlich zu gestalten, habe ich die einzelnen Bereiche mit verschiedenen Farben dargestellt.



**Abbildung 1.8:** Benutzeroberfläche von *CaesarGUI*

In Swing werden die Fensterkomponenten in einer Art Container, so genannte Panels, gepackt. In der Abbildung 1.8 bezeichne ich den Teil des gesamten Fensters, in dem gezeichnet werden kann (also ohne der Menüleiste) als Hauptpanel *mainP*. Dieses Hauptpanel ist wiederum in das linke blaue Einstellungspanel *einstellungP* und in das rechte rote Darstellungspanel *darstellungP* unterteilt.

Das Einstellungspanel ist wiederum in zwei Panels unterteilt. Das obere Abfragepanel *abfrageP* enthält den Label *textL* (Codierverfahren nach Caesar). Das untere Buttonpanel *buttonP* enthält den Button (Codieren). Dieser Button bekommt einen *ActionListener*, der das Anklicken überwacht und dann die entsprechenden Aktionen auslöst. Diese Aktionen werden später in Kapitel 1.3 in der Methode *uebersetzBActionPerformed()* implementiert, so dass ein Klartext in den Geheimtext verschlüsselt werden kann.

Das Darstellungspanel enthält zwei Textfelder. Das obere Textfeld *eingabeTF* ist für die Aufnahme des Klartext bestimmt, das untere Textfeld *ausgabeTF* enthält das verschlüsselte Wort.

### Übung 1.17:

Hole vom Schulserver die Datei *CaesarGUI* und speichere diese als Projekt *Caesar02*. Studiere den Quelltext der Klasse *Ansicht*. Suche die Komponenten, die im obigen Text beschrieben worden sind.

In vielen grafischen Oberflächen wird die Anordnung der Komponenten durch Angaben absoluter Koordinaten vorgenommen. Da jedoch Java-Programme auf vielen unterschiedlichen Plattformen laufen sollen, ist eine solche Vorgehensweise nicht sinnvoll. Zur Anordnung der Komponenten verwendet man in Java verschiedene so genannte *Layoutmanager*. Diese sorgen dafür, dass die Benutzeroberfläche auf allen Plattformen das gleiche Aussehen hat (weitest gehend).

Ich verwende in diesem Beispiel den *GridLayout-Manager* und den *BorderLayout-Manager*. Im *GridLayout* werden die Komponenten innerhalb eines rechteckigen Gitters angeordnet. Die Parameter beim Aufruf des Konstruktors bestimmen vertikale und die horizontale Anzahl der Elemente. Im *BorderLayout* werden die Komponenten in fünf Bereiche aufgeteilt, und zwar in die vier Ränder (Norden Süden, Westen, Osten) und das Zentrum. Weitere Einzelheiten werden im *Handbuch der Java-Programmierung* von Guido Krüger und in ähnlichen Büchern beschrieben.

### Übung 1.18:

Speichere nun das Projekt als *Caesar02b*.

Ändere in der Methode *setTitle()* das Argument „Caesar“ um in „Codierung“. Ändere in der Methode *setBounds()* die Werte (50, 50, 400, 250) in (300, 200, 400, 250) bzw. in (50, 50, 600, 400). Formuliere schriftlich, welche Veränderungen dadurch stattfinden.

### Übung 1.19:

Das Abfragepanel soll nun den zusätzlichen Text „Verschiebung um 3“ enthalten. Ändere deshalb im Konstruktor `gridL = new GridLayout()` die Argumente von (1, 1, 10, 10) auf (2, 1, 10, 10). Bezeichne das neue Label als `verschL` und füge es analog dem Label `textL` dem Abfragepanel hinzu. Vergiss nicht den zusätzlichen Eintrag im Variablenkatalog.

Verändere die letzten beiden Werte 10 in der Parameterliste in den Wert 30. Formuliere die Bedeutung dieser beiden Parameter.

Übung 1.20:

Ändere den Text des Buttons in „Uebersetzen“.

Übung 1.21:

Im Darstellungspanel soll oberhalb des Textfeldes `eingabeTF` das Label `Klartext:` und oberhalb des Textfeldes `ausgabeTF` das Label `Geheimtext:` geschrieben stehen. Das Textfeld `ausgabeTF` wird mit folgender Anweisung `ausgabeTF.setEditable(false)` nicht editierbar. Führe die entsprechenden Änderungen im Quelltext durch.

Übung 1.22:

Gestalte den CENTER-Bereich des Abfragepanels selbstständig.

Nach Abschluss dieser Übungen sollte das Fenster des Projekts `Caesar02` der Abbildung 1.9 gleichen:



Abbildung 1.9: Die grafische Oberfläche von `Caesar02`

### 1.3 Verbindung zwischen Caesar und Ansicht

Unser Projekt, das die Codierung nach Caesar darstellt, ist also in zwei größere Teile zerlegt worden: Der eine Teil (die in Kapitel 2 erstellte Benutzeroberfläche) ermöglicht dem Benutzer, den Klartext und später auch weitere Daten einzugeben, der andere Teil (das in Kapitel 1 erstellte Codierungsverfahren) implementiert die arithmetische Logik für die Berechnungen zur Erstellung des Geheimtextes.

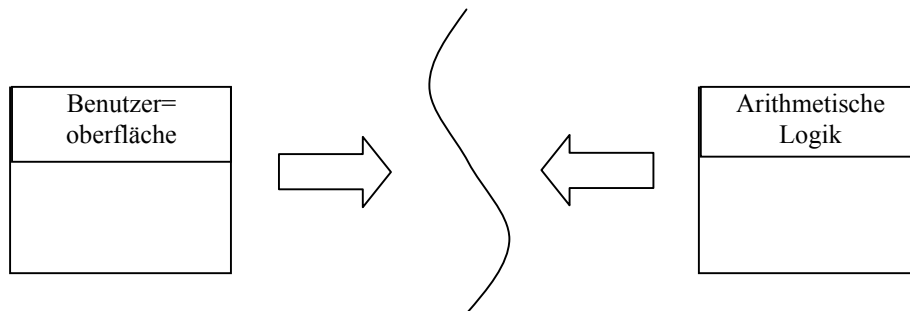


Abbildung 1.10: Festlegung der Schnittstelle

Die Abbildung 1.10 zeigt, dass für beide Teile klar definiert sein muss, wie sie den jeweils anderen Teil benutzen können – die Schnittstelle zwischen ihnen muss definiert werden. In größeren Projekten muss es also klare Richtlinien geben, welches Entwicklungsteam für welche Aufgaben zuständig ist und wie die verschiedenen Teile schließlich in der Gesamtanwendung zusammenspielen sollen. Es wird also notwendig sein, die Schnittstellen festzulegen, bevor an der Implementierung der Teile gearbeitet werden kann.

In dieser Einführung unseres Projekts habe ich aus didaktischen Gründen den umgekehrten Weg beschritten.

Wir werden nun die Benutzeroberfläche mit dem Codierungsverfahren verbinden. Dazu muss die Schnittstelle zwischen diesen beiden Klassen definiert werden.

Betrachten wir noch einmal die grafische Benutzeroberfläche. Wichtig sind hier die beiden Textfelder und der Button. In das Textfeld „Klartext“ wird das zu verschlüsselnde Wort eingegeben. Nach dem Drücken des Buttons „Uebersetze“ wird das codierte Wort im Textfeld „Geheimtext“ wieder ausgegeben. Dieser Vorgang wird in der Methode `uebersetzBActionPerformed()` der Klasse `Ansicht` durchgeführt.

Die beiden Methoden `holeKlartext()` und `gibGeheimtext()` sind also die beiden einzigen Methoden der Klasse `Caesar`, die von der Klasse `Ansicht` verwendet werden und somit auch von der Klasse `Ansicht` aus sichtbar sein sollen. Diese

beiden Methoden stellen die Schnittstelle mit der grafischen Benutzeroberfläche (GUI: Graphic User Interface) dar. Deshalb werden diese mit dem Schlüsselwort *public* versehen.

Die Methode *codiereWort()* erhält nun das Schlüsselwort *private* und ist deshalb nicht von der Klasse *Ansicht* sichtbar. Damit der Klartext trotzdem verschlüsselt wird, wird diese Methode innerhalb der Methode *holeKlartext()* aufgerufen.

### Übung 1.23:

Speichere in **BlueJ** das Programm *Caesar01e* unter dem Namen *Caesar03a* ab. Führe nun die oben aufgeführten Änderungen durch:

```
public void holeKlartext() { ... }  
public void gibGeheimtext() { ... }  
private void codiereWort() { ... }
```

Untersuche, welche dieser drei Methoden nur noch aufgerufen werden können.

Wird nun die Klasse *Caesar* von der Klasse *Ansicht* benutzt, so können nur die beiden Methoden *holeKlartext()* und *gibGeheimtext()* verwendet werden. Alle weiteren Aufgaben müssen die Klassen autonom bewerkstelligen, keine Klasse „pfuscht“ in der anderen herum.

Nun werden wir die Methode *uebersetzBActionPerformed()* implementieren. In dieser Methode werden genau diejenigen Schritte programmiert, die wir im Programm *Caesaer03a* manuell ausführen mussten.

### Übung 1.24:

Formuliere schriftlich alle Schritte, die im Programm *Caesar03a* durchgeführt werden, bis schließlich das Geheimwort ausgegeben wird.

Zuerst wird also ein Objekt *meinCode* der Klasse *Caesar* erzeugt. Nun holt die Methode *getText()* aus dem Textfeld *eingabeTF* den Klartext. Dieses Wort wird als Parameter der Methode *holeKlartext()* des Objekts *meinCode* mitgegeben. Die Methode *gibGeheimtext()* des Objekts *meinCode* gibt das verschlüsselte Wort wieder zurück. Dieses wird mit *setText()* in das Textfeld *ausgabeTF* geschrieben.

```
private void uebersetzBActionPerformed(ActionEvent evt)  
{  
    String wort;  
  
    Caesar meinCode = new Caesar();  
    wort = eingabeTF.getText();  
    meinCode.holeKlartext(wort);  
}
```

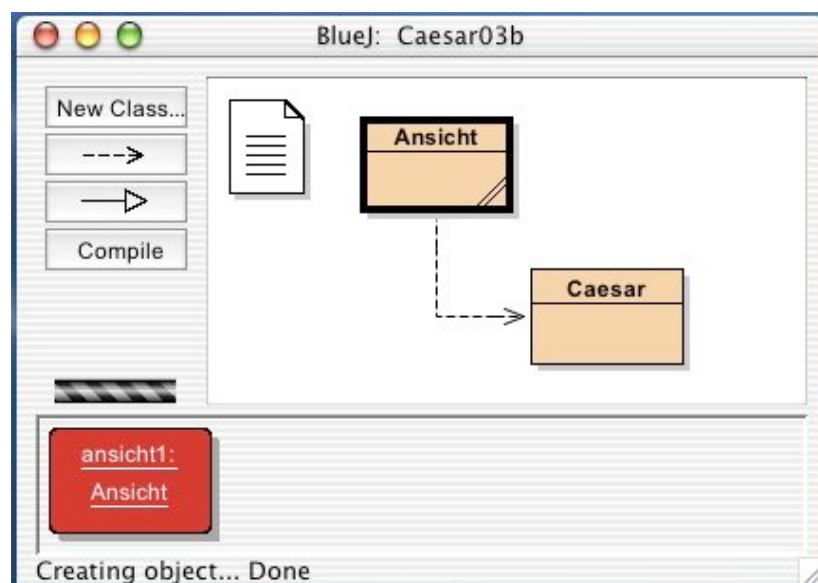
```
wort = meinCode.gibGeheimtext();
ausgabeTF.setText(wort);
}
```

**Abbildung 1.11:** Die Methode *uebersetzBActionPerformed()* in der Klasse *Ansicht*

Man sieht, beide Klassen erledigen ihre Aufgaben selbstständig. Die Klasse *Ansicht* benötigt ausschließlich die beiden Methoden *holeKlartext()* und *gibGeheimtext()* der Klasse *Caesar*. Alle anderen Methoden werden mit Hilfe des Schlüsselworts *private* unsichtbar gemacht und können somit nur innerhalb der jeweiligen Klassen angewendet werden.

### Übung 1.25:

Erstelle nun das Programm *Caesar03b*, in dem du die oben dargestellten Änderungen vornimmst. Überlege dir, ob weitere Methoden mit Hilfe des Schlüsselworts *private* für andere Klassen unsichtbar gemacht werden können.



**Abbildung 1.12:** Das Klassendiagramm von *Caesar03b*

## 1.4 Eine stand-alone Applikation

Bis jetzt läuft unser Programm nur in der Entwicklungsumgebung **BlueJ**. Wir wollen es jedoch als selbstständiges Programm an dritte Personen weitergeben können. Hierzu benötigen wir eine so genannte *main()*-Methode. Wird diese Methode beim Aufruf des Programms gefunden, so läuft der Rest sozusagen von allein ab.

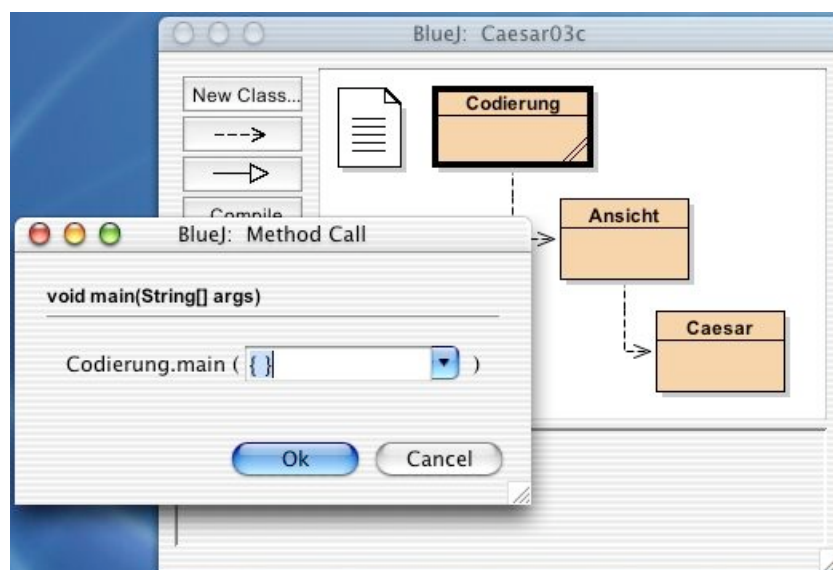
Ich lege diese Methode *main()* aus didaktischen Gründen in eine eigene Klasse *Codierung*. Im Prinzip kann nun diese Klasse als Rezept für alle Programme verwendet werden, um daraus eigenständige Applikationen zu machen.

```
public class Codierung
{
    Ansicht ansicht;

    public Codierung()
    {
        ansicht = new Ansicht();
    }

    public static void main(String[] args)
    {
        Codierung q = new Codierung();
    }
}
```

**Abbildung 1.13:** Quellcode der Klasse *Codierung*



**Abbildung 1.14:** Das Klassendiagramm von *Caesar03c* mit dem Aufruf der Methode *main()*

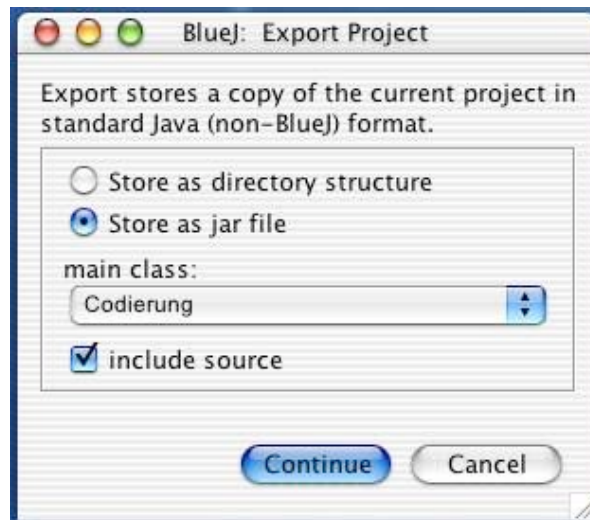
Übung 1.26:

Erstelle nun das Programm *Caesar03c*, in dem du die oben dargestellte Klasse *Codierung* hinzufügst.

Übung 1.27:

Rufe nun den Menüpunkt „Exportieren...“ auf. Wähle „Create Jar File...“ und suche die Klasse, in der die *main()*-Methode liegt. Auf Wunsch kann der

Quelltext mitgespeichert werden. Nach dem Klicken auf „Weiter“ gebe dem Programm nun noch den Namen *Caesar* und stelle die Applikation fertig. Im gewählten Ordner findest du nun die Datei *Caesar.jar*. Diese Datei stellt die stand-alone-Applikation dar.



**Abbildung 1.15:** Erzeugung von *Caesar.jar*

Übung 1.28: (Diese Übung ist sowohl umfang- als auch lehrreich!)

Erstelle ein Programm *Caesar03d* auf der Grundlage von *Caesar03c*, in dem der Benutzer erstens die Verschiebung der Buchstaben im Alphabet selbstständig einstellen kann und zweitens sowohl codieren als auch decodieren kann. Überlege dir zuerst eine geeignete Schnittstelle zwischen der grafischen Benutzeroberfläche (GUI) und Codierung.



**Abbildung 1.16:** Ein Beispiel für die GUI von *Caesar03d*