	Java Anwendung Objektorientierung		AnPr
	Name	Klasse	Datum

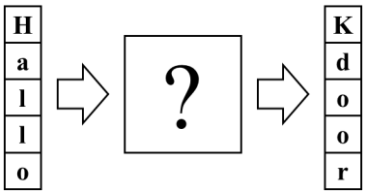
1 Ausgangslage

Um die Objektorientierung zu üben, werden wir uns ein Programm erstellen, welches die Elemente der Objektorientierung nutzt. Es soll ein Programm zur Chiffrierung und Dechiffrierung von Texten erstellt werden. Wir werden zwei verschiedene Chiffrieralgorithmen realisieren, das Cäsar Chiffre und die monoalphabetische Substitution.

~~Blöde~~ Frage:
Sind das nicht sehr unsichere Verschlüsselungsverfahren?
Antwort: Das kann man wohl sagen! Eine Cäsar Verschlüsselung zu knacken kostet selbst einem Programmieranfänger nur wenige Minuten. Unser Ziel ist es hier aber nicht unseren Mailverkehr vor der CIA zu schützen, sondern die grundlegenden Gedanken zur Objektorientierung und in zarten Ansätzen auch der Verschlüsselung zu verstehen.

Bevor wir nun loslegen, müssen wir uns aber erst einmal mit dem Algorithmus beschäftigen.

2 Monoalphabetische Verschlüsselung



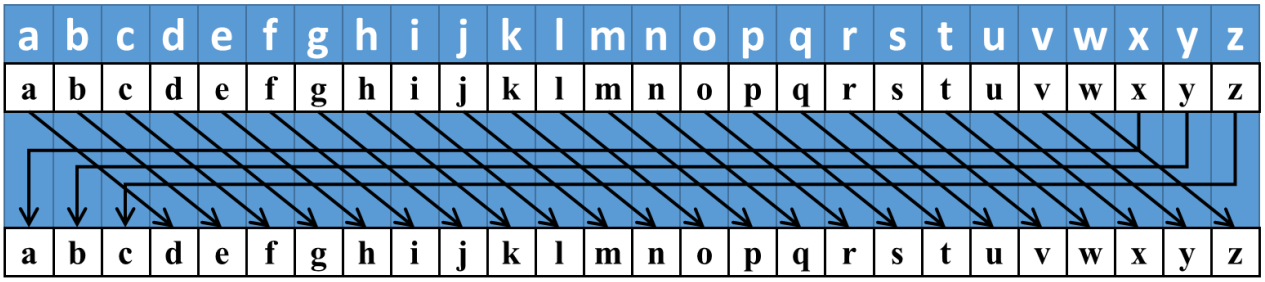
Verschlüsselung bedeutet, dass wir die Zeichen (oder bei weiterführenden Verfahren die dahinterliegenden Zeichencodes) eines Textes durch ein anderes Zeichen ersetzten, so dass die ursprüngliche Bedeutung des Textes nicht mehr ohne weiteres lesbar ist.

Monoalphabetisch bedeutet nun, dass für jedes Zeichen im Quelltext immer exakt ein Ersetzungszeichen für den Zieltext vorgesehen ist. In dem oben gezeigten Beispiel kann man das am Buchstaben „l“ erkennen, der immer mit einem „o“ ersetzt wird.

Zusatzinfo:
Dies lässt auch erahnen, wie wir einen solchen Code „knacken“. Wir machen einfach eine Häufigkeitsanalyse der Buchstaben. Wir finden beispielsweise in deutschen Texten am häufigsten das „e“. Somit haben wir mit einem einfachen mathematischen Verfahren bereits einen Buchstaben entschlüsselt. Aus diesem Grunde ist eine derartige Verschlüsselung eigentlich „keinen Pfifferling“ wert…

2.1 Cäsar Verschlüsselung

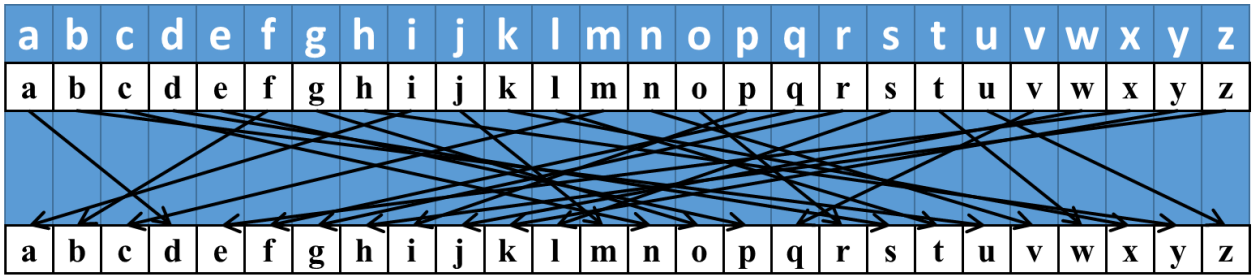
Um nun eine solche Verschlüsselung zu ermöglichen, gibt es zwei einfache Möglichkeiten. Der erste Ansatz geht auf den römischen Feldherrn „Gaius Julius Caesar“ zurück und ist aufgrund seines sehr simplen Aufbaus in Sekunden zu knacken. Hierbei werden die Buchstaben in einer Reihe angeordnet. Diese steht für die Quellbuchstaben. Nun werden die Buchstaben um einen festen Wert nach links oder rechts verschoben und man erhält somit die Ersetzungsbuchstaben. In unserem Beispiel um 3 Werte nach rechts:



Damit wir für die ersten drei Buchstaben auch „Partner“ brauchen, werden die letzten drei Buchstaben wieder nach vorne geschoben. Somit haben wir eine einfache Zuordnung geschaffen, welche jedem Quellbuchstaben einen Ersetzungsbuchstaben zuordnet. Das „a“ wird durch „d“ ersetzt, das „b“ mit „e“, „c“ mit „f“, „d“ wird mit „g“ ersetzt usw.

2.2 Monoalphabetische Substitution

Eine etwas „kompliziertere“ Version dieses Verfahrens ist die Monoalphabetische Substitution. Hier wird einfach jedem Buchstaben der Quelle ein fester, aber beliebiger anderer Buchstabe zugeordnet:



Auch hier kann man mit Hilfe einer Häufigkeitsverteilung den Code entschlüsseln, jedoch ist es hier viel aufwändiger, da es einige Buchstaben gibt, welche ähnlich häufig vorkommen wie andere.

2.3 Umsetzungshilfe

Um einen derartigen Algorithmus umsetzen zu können, müssen wir uns nochmal auf die Grundlagen der einfachen Datentypen besinnen. Wir wissen, dass Texte als Strings vorliegen und diese wiederum als eine Aneinanderreihung von Character Werten zu interpretieren sind. Ein Character jedoch ist nichts anderes als eine Zahl. Wir sehen uns folgenden Code mal an:

Zuerst lassen wir uns das Zeichen anzeigen:

Danach wandeln wir es in einen int - Wert um, damit println() uns die ASCII Zahl ausgibt.

```
String sValue = "hallo";
for (int i = 0; i < sValue.length(); i++) {
    System.out.print(sValue.charAt(i) + " ");
    System.out.println((int)sValue.charAt(i));
}
```

Wie wir sehen, kann jeder Character Wert auch als Zahl dargestellt und auch interpretiert werden. Mit Hilfe des Typecasts ist dies möglich. Wichtig ist aber auch, dass wir nicht nur von char -> int den Typecast vornehmen können, sondern auch umgekehrt. Insofern können wir nun mit Buchstaben „rechnen“:

Durch diese Rechnung verschieben wir die Buchstaben ‚a' bis ‚z' auf die Werte ‚0' bis ‚25'.

```
String sValue = "hallo";
for (int i = 0; i < sValue.length(); i++) {
    System.out.print(sValue.charAt(i) + " ");
    System.out.println((int)sValue.charAt(i) - 'a');
}
```

Das Programm gibt mir somit die Zahlen 7, 0, 11, 11 und 14 aus.

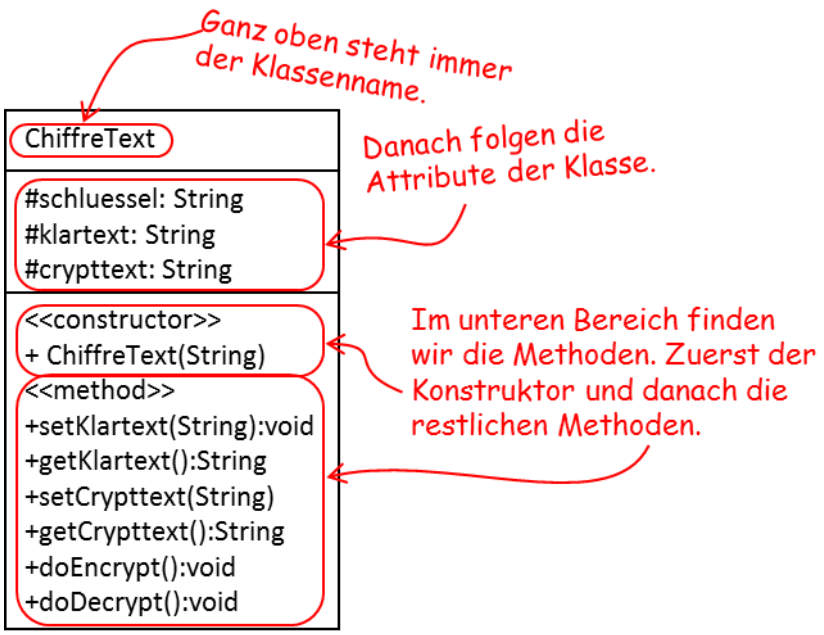
Basierend auf diesen Überlegungen können wir uns eine Substitutionstabelle erstellen, indem wir ein char[] Array schaffen, welches an der Position 0 den Substitutionswert für ‚a‘, an Position ‚1‘ den Substitutionswert für ‚b‘ usw. beinhaltet.

3 Definition unserer Zielapplikation

Wir werden nun ein Programm designen, welches Verschlüsselungen nach unterschiedlichen Algorithmen ermöglicht. Die ersten beiden Algorithmen werden der Cäsar Chiffre und die monoalphabetische Substitution sein. Das Programm soll nach dem Start einen Filenamen erwarten und eine Auswahl, welcher Algorithmus nun durchgeführt werden soll. Danach wird das File gelesen und als neues File chiffriert wieder auf die Festplatte geschrieben.

4 UML Notation von Klassen

Wir legen zuerst unsere Objektstruktur fest. Da wir flexibel sein wollen, werden wir eine Klassenhierarchie vorsehen. Um dies möglichst übersichtlich darzustellen, verwenden wir die UML Klassennotation. Im nebenstehenden Beispiel ist die UML Notation einer Klasse dargestellt.



Es werden im Minimalfall nur die „Signaturen[1]“ und die wichtigsten Variablen dargelegt, mitunter findet man auch die Rückgabewerte nach dem Doppelpunkt angegeben.

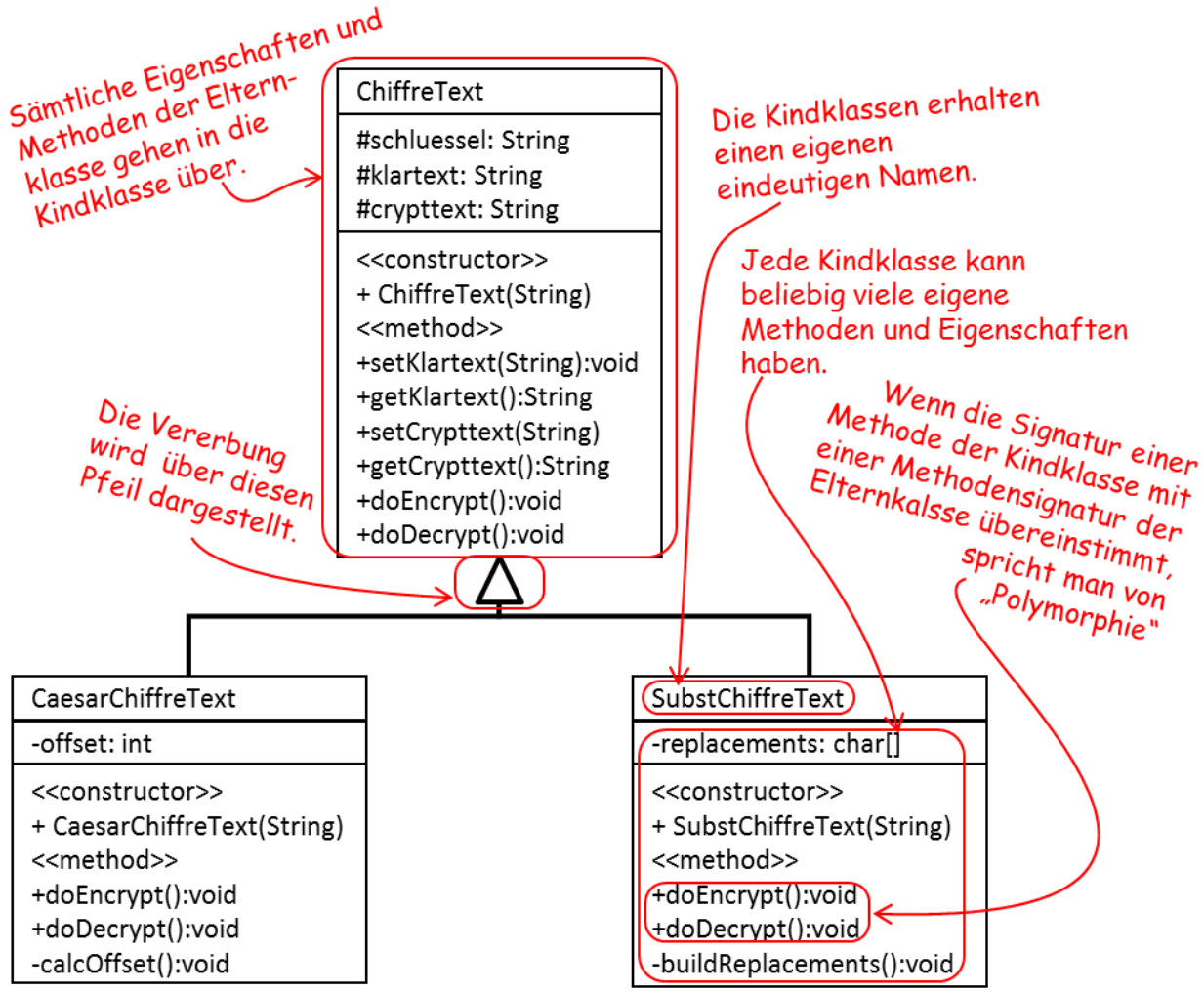
Hier findest Du die wichtigsten Begriffe kurz erklärt:

Klassenname	Jede Klasse erhält seinen eigenen Namen. Die Namenskonvention empfiehlt, dass der Name mit einem Großbuchstaben beginnt, keine Sonderzeichen enthält und möglichst sprechend ist. Weiterhin liegt in Java jede Klasse in einem eigenen File (wenn wir die „inner classes“ hier einmal ausnehmen). Mitunter wird der Packagename noch vorne angegeben und mit einem doppelten Doppelpunkt getrennt: `mypackage::ChiffreText`
Attribut	Die Attribute sind Variablen, welche als „Gedächtnis“ des Objektes fungieren. Man nennt diese Variablen „Instanzvariablen“ oder auch „Eigenschaften“ des Objekts. Die Variablen werden mit Zugriffsmodifikatoren (siehe Kapitel weiter Unten) versehen.
Konstruktor	Ein Konstruktor ist eine Methode, welche als Rückgabewert ein Objekt der entsprechenden Klasse zurückgibt und wird immer mit den gleichen Namen angegeben, wie die Klasse selbst. *Anmerkung: In Java gibt es keine Destruktoren – in anderen Programmiersprachen wie C++ existieren sie noch. Die Funktionsweise wird weiter unten beschrieben.*
Methoden	In der Objektorientierung werden Funktionen und Prozeduren als „Methoden“ bezeichnet. Da Methoden Zugriff auf die Instanzvariablen und somit auf das Gedächtnis des Objektes haben, besitzen sie auch ein anderes Verhalten als Prozeduren und Funktionen, die bei jedem Aufruf immer gleich agieren. Methoden können von außen zugreifbar sein, insofern wird zu den Methoden auch immer ein Zugriffsmodifikator angegeben.

Klassen können auch voneinander „erben“. Das heißt, dass eine Klasse definiert werden und sämtliche Eigenschaften auf eine Kindklasse vererbt werden können. Wenn wir in unserem Beispiel zwei Kindklassen von der

[1] Signatur einer Methode ist der Methodenname und die Anzahl, Reihenfolge und Datentyp der Parameter.

Elternklasse ChiffreText erzeugen wollen – „CaesarChiffreText“ und „SubstChiffreText“ – wird dies auch in dem UML Diagramm dargestellt.



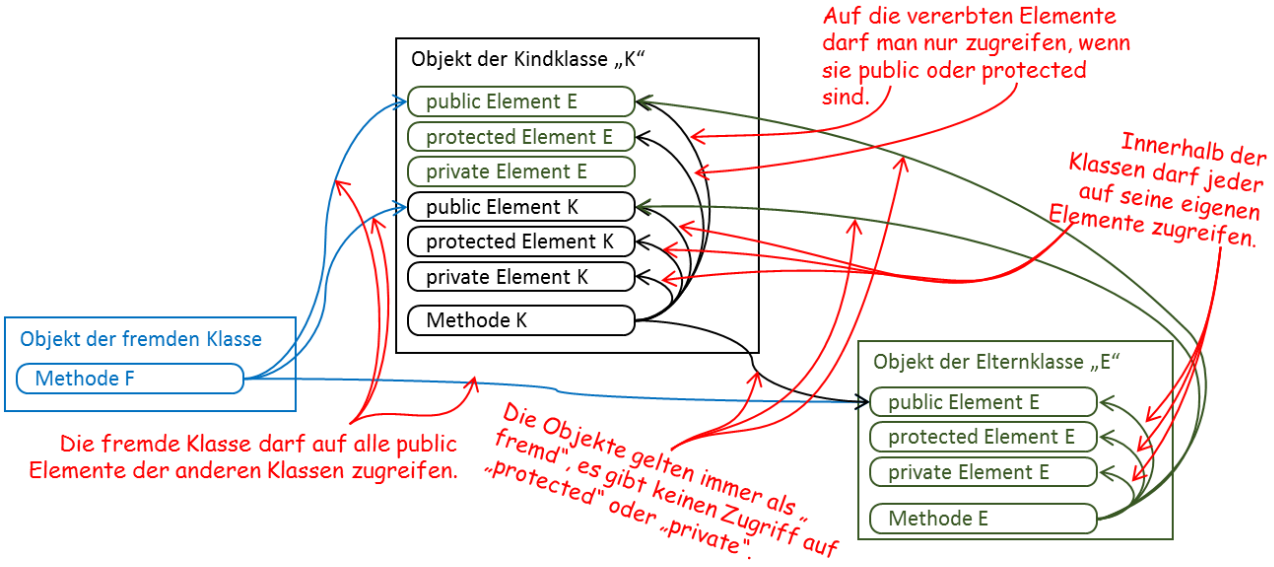
Das Diagramm zeigt also mittels des Pfeiles die Vererbungsrichtung (bzw. korrekterweise „Erweiterungs-richtung“) an und zusätzlich noch die zu implementierenden Methoden und Eigenschaften.

5 Zugriffsmodifikatoren

Bis jetzt haben wir in unseren Programmen immer nur mit lokalen Variablen gearbeitet. Der Gültigkeitsbereich – also die Zugriffsmöglichkeit – hat sich auf die einschließenden Klammern beschränkt. Durch die Einführung von Klassen und Objekten können wir nun die Zugriffsmöglichkeit auf einer ganz anderen Ebene und hier auch detaillierter steuern. Gehen wir von folgender Situation aus – wir haben eine Elternklasse und eine davon erbende Kindklasse, sie sind also verwandt. Weiterhin gibt es eine komplett „fremde“ Klasse, welche die Ob-jekte von den anderen beiden Klassen lediglich nutzt. Dadurch gibt sich folgende Matrix der Zugriffsrechte:

Hierauf wird zugegriffen:		***Diese Objekte dürfen zugreifen:***		
Klasse:	**Zugriffsmodifikator:**	**Elternklasse:**	**Kindklasse:**	**fremde Klasse:**
Elternklasse:	public	ja	ja	ja
	protected	ja	ja	nein
	private	ja	nein	nein
Kindklasse:	public	-	ja	ja
	protected	-	ja	nein
	private	-	ja	nein

Hierzu muss man allerdings noch eine kleine Ergänzung machen – der Zugriff der Kindklasse auf die Elemente der Elternklasse erfolgt innerhalb des eigenen Objektes – das heißt, dass wenn ein Attribut von der Elternklasse geerbt wurde, dann kann es innerhalb der Kindklasse genutzt werden, so als wäre es ein Attribut, welches in der Kindklasse deklariert wurde. Das heißt nicht, dass ein Objekt der Kindklasse auf ein anderes Objekt der Elternklasse zugreifen kann (sie können als „fremd“ angesehen werden):



Im UML Klassendiagramm werden die vier „Sichtbarkeiten“ mit folgenden Symbolen vor den Attributen und Methoden dargestellt:

public	+
protected	#
private	-
package	~

Zusatzinfo:
In Java gibt es noch die Unterscheidung „package“ (Details besprechen wir später). Innerhalb eines Packages darf auch auf alle protected Elemente zugegriffen werden. Ohne Modifikator ist der Zugriff auf alle Elemente der des Packages ebenfalls möglich, was mit dem ~ Zeichen gekennzeichnet wird.

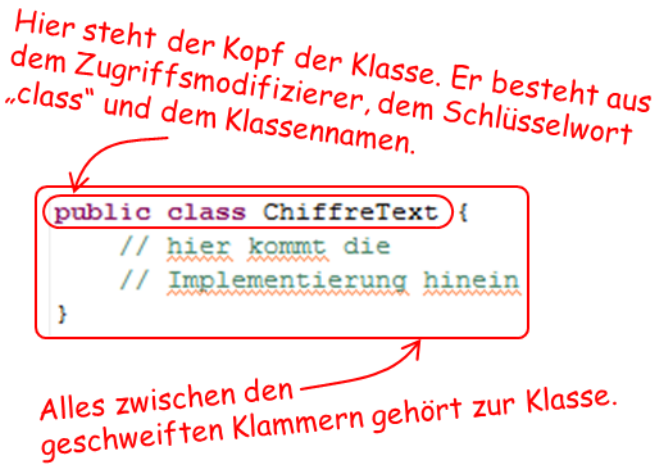
6 Syntax in Java

Nun wollen wir die Implementierung angehen. Hierzu müssen wir uns zuerst den Syntax ansehen. Wir gehen die wichtigsten Punkte durch. In anderen Programmiersprachen funktioniert das mitunter anders, wobei die meisten Begrifflichkeiten wiederzufinden sind.

Zusatzinfo:
Es gibt Programmiersprachen, welche unter „objektorientiert“ laufen, jedoch bei genauerem Hinsehen eine prozedurale Sprache darstellen, bei denen die Objektorientierung nicht vollumfänglich realisiert wurde. Ein Vertreter davon ist Javascript, welche als „prototypenbasierte Programmiersprache“ bezeichnet wird.

6.1 Die Klasse

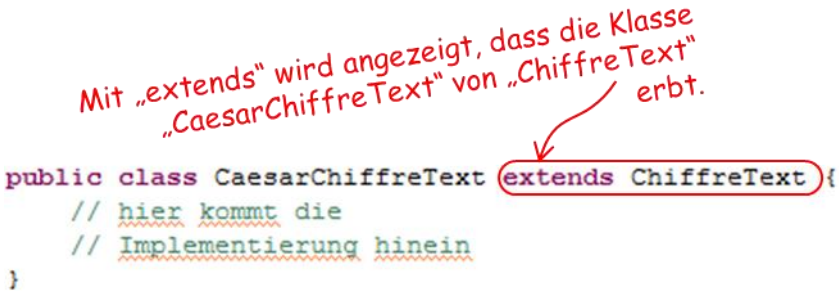
Die Klasse umfasst sämtliche, der Klasse zugehörigen Elemente. Dies sind Methoden und Eigenschaften, in manchen Fällen auch innere Klassen („inner class“), wobei wir das an dieser Stelle ausklammern. In Java sieht eine Klassendefinition wie rechts dargestellt aus.



Zuerst haben wir den Zugriffsmodifikator, der sinnvollerweise immer public ist. Danach kommt das zwingend erforderliche Schlüsselwort „class“, gefolgt vom Klassennamen, der gleich dem Filenamen sein muss.

Der Klassenrumpf wird (wie bei anderen Elementen in Java auch) durch geschweifte Klammern markiert. Alles innerhalb dieser Klammern gehört also zur Klasse.

Das hier dargestellte Beispiel ist die Minimalversion. Es gibt noch weitere Schlüsselwörter, welche im Kopf der Klasse angewendet werden können. Einige davon werden wir später klären, wobei das für die Vererbung jetzt schon angesprochen wird. Wenn eine Kindklasse von einer Elternklasse erben soll, dann kommt der Begriff „extends“ ins Spiel. Das Wort „extends“ – also „erweitert“ – soll transparent machen, dass die Kindklasse alle Eigenschaften und Methoden der Elternklasse erbt und zusätzlich noch weitere Methoden und Eigenschaften implementieren kann – also erweitert. Man sagt auch, die Kindklasse wurde von der Elternklasse abgeleitet.

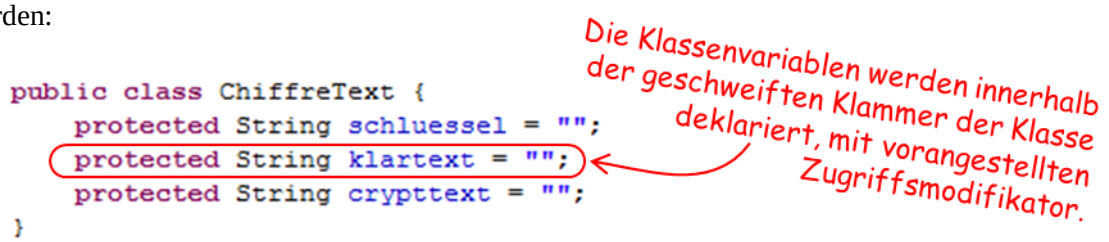


6.2 Instanzvariablen, Getter und Setter

Bevor wir nun loslegen, müssen wir die Eigenschaften der Klasse festlegen. Wir gehen bei der Klasse ChiffreText von drei Eigenschaften – also Instanzvariablen aus: schluessel, klartext und crypttext.

~~Blöde~~ Frage:
Ich habe mal was von Klassenvariablen gehört. Ist das das gleiche wie Instanzvariablen?
Antwort: Nein, das sind zwei unterschiedliche Dinge. Eine Klassenvariable gehört zur Klasse, und muss nicht instanziiert werden. Eine Instanzvariable muss instanziiert werden, kann also nur innerhalb eines Objektes genutzt werden. Wem das nun zu „merkwürdig“ klingt, sollte weiter unten das Kapitel zu „static“ lesen!

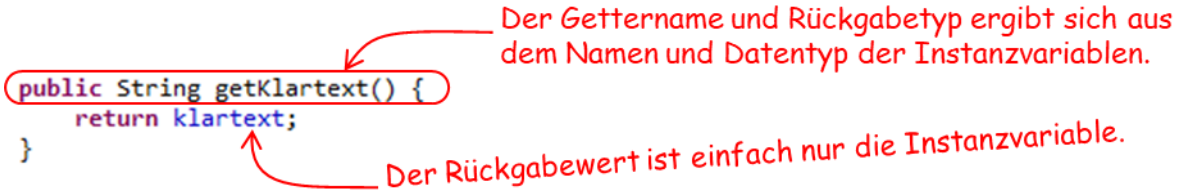
Alle drei Instanzvariablen sind vom Typ String und müssen laut unserem UML Diagramm als „protected“ ausgelegt werden:



So – nun haben wir uns aber ein potentielles Problem geschaffen. Wenn wir die Elemente alle als protected (oder auch private) anlegen, dann können wir von außen nicht drauf zugreifen. Was bringt uns also eine Variable, deren Inhalt außer uns keiner kennt?
Die Antwort auf diese Frage hat etwas mit „Programmierstil“ zu tun. Wir wollen im Regelfall nicht, dass Außenstehende unkontrollierten Zugriff auf unsere inneren Variablen haben. Ein Grund dafür ist bspw., dass wir beim Ändern von Werten irgendwelche Aktionen innerhalb der Klasse oder des gesamten Programms

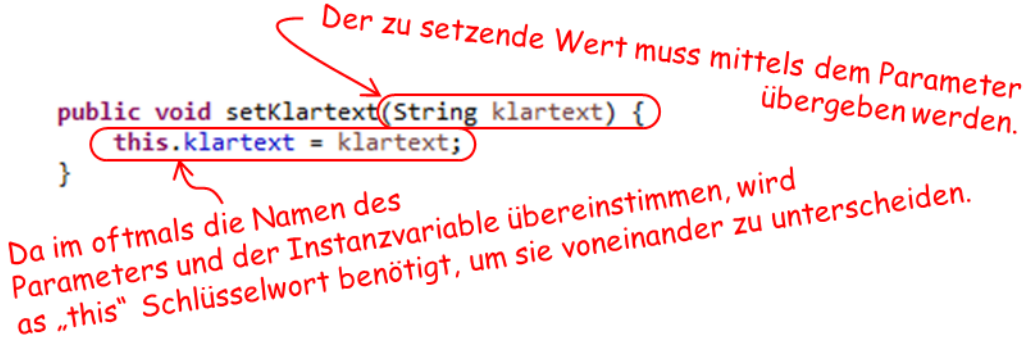
durchführen müssen. Eventuell möchten wir bei einem Grafikprogramm das Neuzeichnen triggern, sobald sich ein Wert geändert hat. Das Triggern ist aber ein Methodenaufruf und kann somit nur innerhalb von Code – also wiederum nur innerhalb einer Methode durchgeführt werden. Um dies nun sicherzustellen, verstecken wir unsere Variablen von der Außenwelt und ermöglichen den Zugriff über kleine „Türsteher“, die Getter- und Settermethoden, wo wir auch das Neuzeichnen auslösen könnten. Diese Methoden machen wir public und sind wirklich einzig und allein dafür da, den außenstehenden Nutzern unserer Klasse einen von uns kontrollierten Zugriff auf unsere Variablen zu ermöglichen.

Ein Getter sollte immer genau wie die Variable heißen, nur mit dem Wort „get“ davor:



An dem Code sehen wir, dass da eigentlich nicht viel dahintersteckt. Trotzdem sehen wir noch eine wichtige Eigenschaft von Instanzvariablen – ich kann sie innerhalb der Methoden meiner Klasse ganz normal nutzen, als wären sie innerhalb der Methode deklariert worden. In Eclipse (mit Standardkonfiguration) wird durch die blaue Farbe indiziert, dass es sich um eine Instanzvariable handelt.

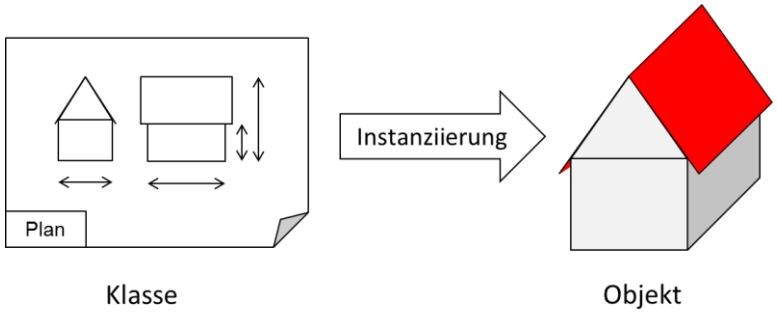
Sehen wir uns nun den Setter an. Dieser heißt ebenfalls wie die Instanzvariable, allerdings mit dem Wort „set“ davor:



Hier noch ein kleiner Hinweis auf das Schlüsselwort „this“. Wenn es, wie im Falle unseres Setters, die Instanzvariable den gleichen Namen trägt wie die (lokale) Parametervariable, kann man mit „this“ und dem Punktoperator auf die Instanzvariable zugreifen. Wir kennen den Punktoperator ja schon von den Strings – mit dem greift man auf die Eigenschaften und Methoden der Objekte zu (bspw. „myString.toUpperCase()“). Das Schlüsselwort „this“ ist eine Referenz auf sich selbst, insofern greifen wir damit innerhalb eines Objektes also auf die Eigenschaften und Methoden vom eigenen Objekt zu. Somit können wir zwischen der lokalen Variablen und der Instanzvariablen mit gleichem Namen unterscheiden.

6.3 Konstruktor

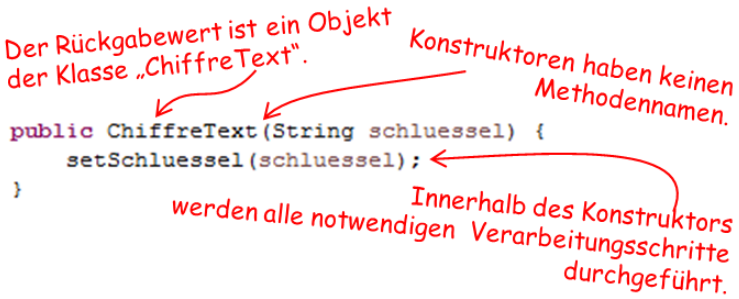
Am Anfang der Lebenszeit eines Objekts steht immer der Konstruktor. Dies ist eine ganz besondere Methode, welche aus der Klasse das Objekt – bzw. korrekterweise die „Objektinstanz“ erzeugt. Doch erst mal langsam – was ist ein Objekt und was ist eine Klasse? Die Klasse ist das, was wir als Programmierer erzeugen – also das fertig kompilierte Programm. Das Objekt ist etwas, was erst zur Laufzeit im Speicher erzeugt wird. Man kann es also am besten so verstehen, dass die Klasse ein Bauplan ist und das Objekt das davon erzeugte Haus.



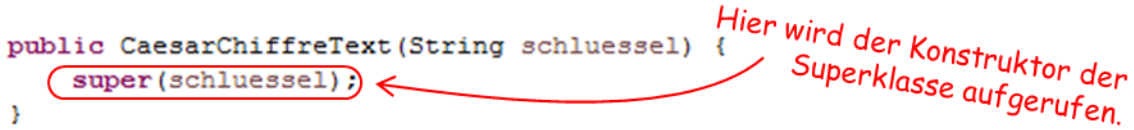
Damit sollte auch klar sein, dass wir mit einem Plan sehr viele gleichartige Objekte instanziieren können – so wie wir in einer Wohnsiedlung oftmals auch gleichartige Häuser sehen, welche wohl alle nach demselben Plan gebaut wurden.
Ein Konstruktor hat folgende Eigenschaften:

- Der Konstruktor benötigt keinen eigenen Namen, da er ja 1:1 wie die Klasse heißt. Man kann das auch so interpretieren, dass der Konstruktor immer ein Objekt der Klasse zurückgibt – der Name also gleich dem Rückgabetypen entspricht.
- Der Konstruktor muss public sein, damit wir von überall aus ein Objekt der Klasse erzeugen können
- Der Konstruktor hat keinen „return“ Befehl, da die Rückgabe des Objektes fest vorgegeben ist; schließlich ist das ja auch der Sinn des Konstruktors
- Es kann mehrere Konstruktoren mit unterschiedlichen Parametern geben (er kann also „überladen“ sein)
- Wenn kein Konstruktor vorgegeben wird, existiert immer der Standardkonstruktor – das ist ein Konstruktor ohne Parameter. Dieser kann aber ggf. auch überschrieben werden.

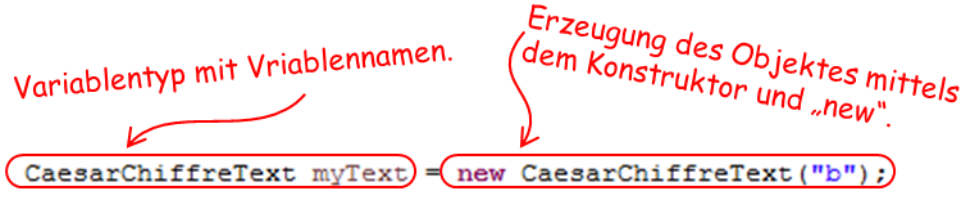
Sehen wir uns mal einen Konstruktor mit einem Parameter an. Der Rückgabedatentyp entspricht dem der Klasse und die Parameter werden entsprechend den Notwendigkeiten der Initialisierung festgelegt.



Wenn man nun, wie im Falle der der Klasse „CeasarChiffreText“, eine abgeleitete Klasse hat, dann muss man auch den Konstruktor der Elternklasse aufrufen – man würde hier von der „Superklasse“ sprechen. Genau wie es eine Referenz auf sich selbst gibt (das war ja das Schlüsselwort „this“) gibt es auch eine Referenz auf die Elemente der Superklasse. Sinnigerweise ist dies das Schlüsselwort „super“. Der Konstruktorzugriff auf die Superklasse ist demnach auch „super()“:



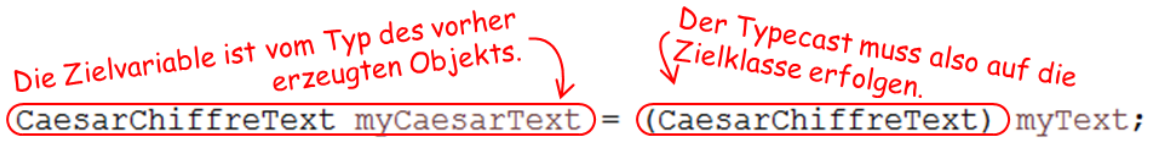
Der Aufruf eines Konstruktors zur Erzeugung eines neuen Objektes erfolgt immer mit dem Schlüsselwort „new“. Dadurch wird die JVM dazu veranlasst aus der Klasse (also dem „Plan“) ein Objekt (also ein „Haus“) zu erzeugen.



6.4 Cast von Objekten

Die Vererbungsstruktur ermöglicht es uns nun, ein Objekt einer Kindklasse in einer Variablen vom Typ der Elternklasse abzulegen:

Somit können wir immer Objekte von allen Kindklassen in die Variablen der Elternklassen ablegen. Sollten Variablen mehrfach vererbt worden sein (es also „Großeltern“, „Urgroßeltern“ usw. gibt), können die Objekte in Variablen des gesamten Vererbungsbaums nach oben abgelegt werden. Wichtig ist hierbei zu verstehen, dass wir an dieser Stelle aber nur auf die Methoden und Eigenschaften zugreifen können, die auch in der Klasse des Variablentyps der entsprechenden Ebene implementiert wurden. Wenn wir alle Methoden des eigentlichen Objektes zugreifen wollen, müssten wir einen Typecast (wie bei den einfachen Datentypen) auf die ursprünglich instanziierte Klasse realisieren:

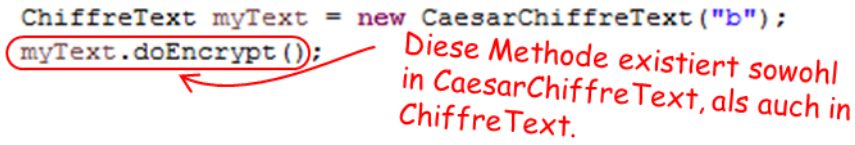


Zusatzinfo:
In Java gibt es eine Elternklasse, welche am Anfang jeder Vererbungskette steht – sozusagen „Adam“ bzw. „Eva“ ☺. Das ist die Klasse „Object“. Jedes Objekt in Java, egal ob String oder CaesarChiffreText, kann in eine Variable des Typs „Object“ gelegt werden.

6.5 Polymorphie

Wir haben in unserer Klasse „ChiffreText“ eine Methode eingebaut, welche auf der Ebene ChiffreText noch gar nicht implementiert werden kann – die Methoden „doEncrypt()“ und „doDecrypt()“. Diese kann sinnvollerweise erst dann umgesetzt werden, wenn wir wissen, welcher Algorithmus umgesetzt werden soll. Im Falle von „CaesarChiffreText“ und „SubstChiffreText“ sind das zwei verschiedene Implementierungen. Da die Klassen „CaesarChiffreText“ und „SubstChiffreText“ aber diese Methoden beide implementieren, spricht man hier von „Überschreiben der Methoden“. Dies bedeutet, dass die Methode zweimal existiert – einmal in der Elternklasse und einmal in der jeweiligen Kindklasse.

Das praktische daran ist nun, dass wenn wir ein Objekt der Klasse „CaesarChiffreText“ erzeugen und in eine Variable der Klasse „ChiffreText“ legen, dann hat diese – genau wie die „CaesarChiffreText“ Klasse – diese Methode:



Die große Frage ist nun, welche der beiden Implementierungen wird nun aufgerufen – die von „ChiffreText„ oder von „CaesarChiffreText“? Die Antwort ist: von „CaesarChiffreText“, weil dieses Objekt ja auch instanziiert wurde. Damit wird immer die richtige Methode aufgerufen – wenn wir also in der Variablen myText ein Objekt des Typs „CaesarChiffreText“ hineingelegt haben, dann wird die Implementierung des Caesar Chiffres aufgerufen, wenn ein Objekt des Typs „SubstChiffreText“ hineinlegen, eben die Implementierung der Klasse „SubstChiffreText“. Es ist also hierfür entscheidend, welche Klasse das instanziierte Objekt hat und nicht, welche Klasse die Variable hat.

Zusatzinfo:
Mit Hilfe des „super“ Schlüsselworts kann man innerhalb der doEncrypt() Methode der Kindklasse auf die Methode der Elternklasse zugreifen. Wenn wir bspw. in der Klasse „SubstChiffreText“ super.doEncrypt()“ aufrufen, wird aus der Elternklasse die Methode „doEncrypt()“ aufgerufen.

Da die Implementierung dieser Methode in ChiffreText eigentlich keinen Sinn macht (schließlich steht auf diesem Abstraktionslevel noch kein Algorithmus fest), können wir auf der Ebene von ChiffreText auch das Schlüsselwort „abstract“ davor schreiben. Dadurch entfällt die Implementierung der Methode in der Elternklasse (also „ChiffreText“).

Das Schlüsselwort „abstract" bewirkt, dass die Methode nicht implementiert werden muss.

```
public abstract void doEncrypt();
```

Die Signatur der Methode ist vorhanden, jedoch kein Methodenrumpf.

Wenn dies dergestalt designt wurde, ist es ab jetzt nicht mehr möglich die Klasse zu instanziieren. Es ist also nicht mehr möglich folgenden Code zu schreiben: new ChiffreText(„xyz").

Der Grund liegt darin, dass für die Methode „doEncrypt" keine Implementierung vorgegeben ist und somit eine Erzeugung des Objektes unmöglich ist. Die Forderung von Java ist es nun, dass die ganze Klasse als „abstract" markiert werden muss, was bei der Klassendefinition erfolgt:

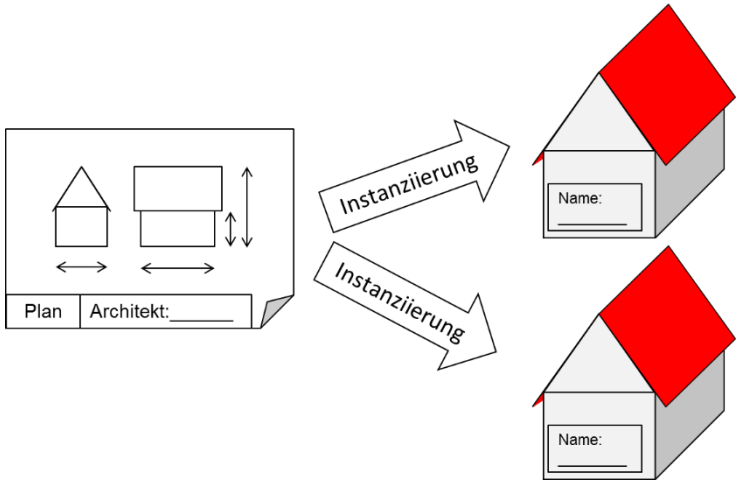
Wenn mindestens eine Methode „abstract" ist, dann muss die ganze Klasse „abstract" sein.

```
public abstract class ChiffreText {
```

Für unser Beispiel ist dies absolut sinnvoll und sollte im Zuge einer sauberen Programmierweise auch so umgesetzt werden. Dadurch wird für den Leser unseres Codes von vorneherein klar: diese Klasse kann nicht instanziiert werden, sondern dient lediglich als Prototyp für abgeleitete Klassen. (Ein ähnliches Verhalten kann man mit „Interfaces" realisieren, was ein Thema für später sein soll).

6.6 Das Schlüsselwort „static"

Ein weiteres wichtiges Schlüsselwort in der objektorientierten Programmierung ist „static". Wir kennen den Begriff ja bereits von „`public static void main(String[] args)`…". Static bedeutet, dass wir die Variable nutzen können, obwohl kein Objekt instanziiert wurde. Um das transparent zu machen, sehen wir uns nochmal unseren Vergleich mit dem Haus und dem Plan an.



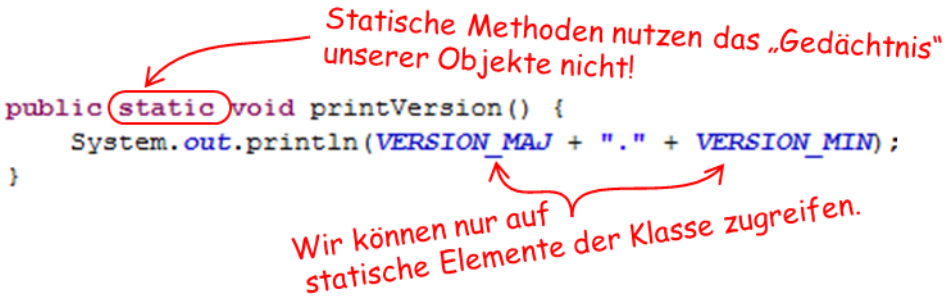
Gehen wir davon aus, dass wir basierend auf dem Plan zwei Häuser bauen (also aus einer Klasse zwei Objekte instanziizeren). Jedes Haus hat eine Variable, in der wir den Besitzer des Hauses eintragen können (also das „Türschild" des Hauses). Da wir zwei Häuser haben, können wir auch zwei verschiedene Namen eintragen. Somit sprechen wir hier von „Instanzvariablen", da wir pro Instanz eine Variable vorfinden.

Im Gegensatz dazu finden wir auf dem Plan ebenfalls einen Platz, in dem wir den Namen (diesmal des Architekten) eintragen können. Da der Plan aber nur einmal existiert, darf es diese Variable auch nur einmal geben. Da der Bauplan beim Programmieren die Klasse ist, sprechen wir hier von einer Klassenvariablen. Wir brauchen also kein Haus zu bauen, um in dem Plan den Architekten abzulegen. Eine solche Variable würden wir in Java als „static" implementieren. Da wir aber selten die Notwendigkeit haben statische Variablen zu ändern, werden meist nur Konstanten als statisch implementiert:

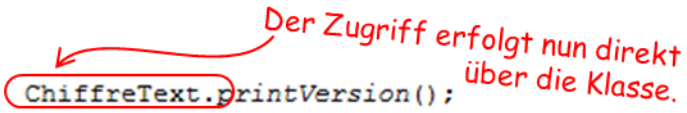
Mit „static" definieren wir eine Klassenvariable, die nicht instanziiert werden muss.

```
public static final int VERSION_MAJ = 1;
```

Was wir häufiger finden sind statische Methoden. Das sind Methoden, welche nicht auf die Instanzvariablen zugreifen dürfen – wir erinnern uns, statisch bedeutet, es existiert nur auf dem Plan! Hier ein Beispiel einer statischen Methode:



Wie wir sehen, können wir auf die statischen Variablen (bzw. in unserem Fall „Konstanten“) zugreifen. Wenn wir die Methode irgendwo in unserem Code aufrufen wollen, dann dürfen wir nun nicht mehr das Objekt voranstellen (also myTxt.printVersion()), sondern müssen den Klassennamen voranstellen:



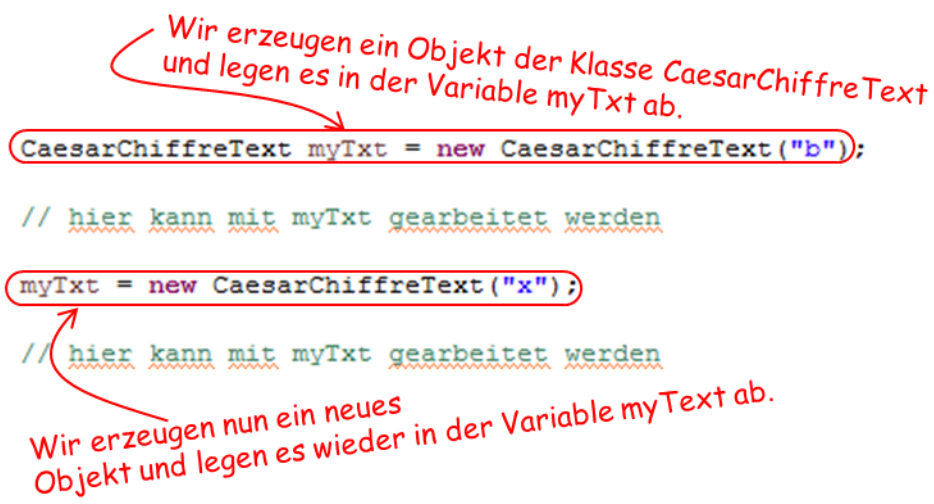
Wir kennen solche Methoden bereits, bspw. aus der Integer Klasse (`Integer.parseInt("123")`). Weiterhin haben wir alle schon mit einer ganz wichtigen statischen Methode gearbeitet, die „main“ Methode. Dies war unser Einstiegspunkt in die Java Welt. Da wir beim Aufruf unseres Programms aber noch keine Objekte erzeugen konnten, muss die main Methode statisch sein – am Anfang haben wir ja nur unsere Klasse!

6.7 Destruktor

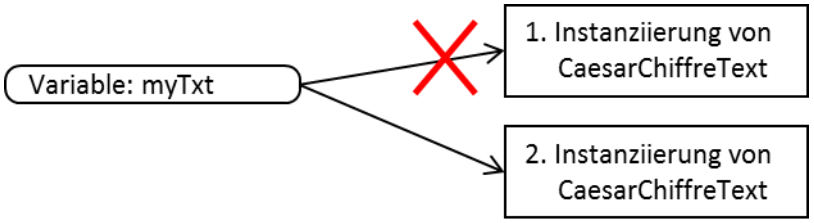
Wie weiter oben schon erwähnt, gibt es in Java keinen Destruktor. C++ sieht ihn noch vor – aber was ist eigentlich ein Destruktor?
Grundsätzlich wissen wir, dass sobald wir Variablen und Objekte erzeugen, im Speicher hierfür Platz geschaffen werden muss. Dies erledigt der Konstruktor. Der Destruktor wiederum ist dafür da, diesen Platz wieder freizugeben. In C++ müssen wir also explizit den Destruktor aufrufen, damit der Platz im RAM wieder freigegeben wird. Vergessen wir dies, so würden wir einen „Memory Leakage“ erzeugen, der Speicher würde also mit der Zeit volllaufen, ohne dass sinnvoll genutzte Daten darin liegen würden.

~~Blöde~~ Frage:
Heißt das, Java gibt einmal genutzten Speicher nicht mehr frei?
Antwort: Nein, das wäre das Todesurteil für eine Programmiersprache! Java nutzt den Destruktor nicht, weil man den Programmieren schlichtweg nicht über den Weg traut und davon ausgeht, dass sie immer mal wieder vergessen den Destruktor aufzurufen. Insofern hat man versucht diesen Prozess zu automatisieren. Der „Garbage Collector“ war geboren.

In Java nutzen wir den sogenannten Garbage Collector, der sich darum kümmert, die nicht mehr benötigten Resourcen wieder freizugeben. Dazu prüft er in regelmäßigen Abständen, ob zu einem Objekt noch irgendwelche Referenzen existieren. Sehen wir uns das folgende Beispiel mal an, um diesen Zusammenhang zu verstehen. Es wird ein Objekt erzeugt und in einer Variablen gespeichert. Wir wissen, dass rein technisch gesehen in dieser Variablen nun die Id (eine Art Adresse) des Objektes gespeichert ist. Das Objekt liegt also im Speicher und wir können es mit dem Inhalt der Variablen (der Adresse) finden und damit arbeiten. Was wir allerdings nicht wissen ist die tatsächliche Adresse. Das brauchen wir aber auch nicht, weil wir ja ohnehin nur mit der Variablen arbeiten.



Erzeugen wir nun ein neues Objekt, so wird dies auch im Speicher residieren, allerdings unter einer anderen Adresse. Wenn wir also dieses Objekt in der Variablen ablegen, überschreiben wir den aktuellen Inhalt und somit den Wert der Adresse des ersten Objektes. Da wir die eigentliche Adresse aber nicht kennen, verlieren wir somit jegliche Referenz auf dieses Objekt und können es de facto nie wiederfinden.



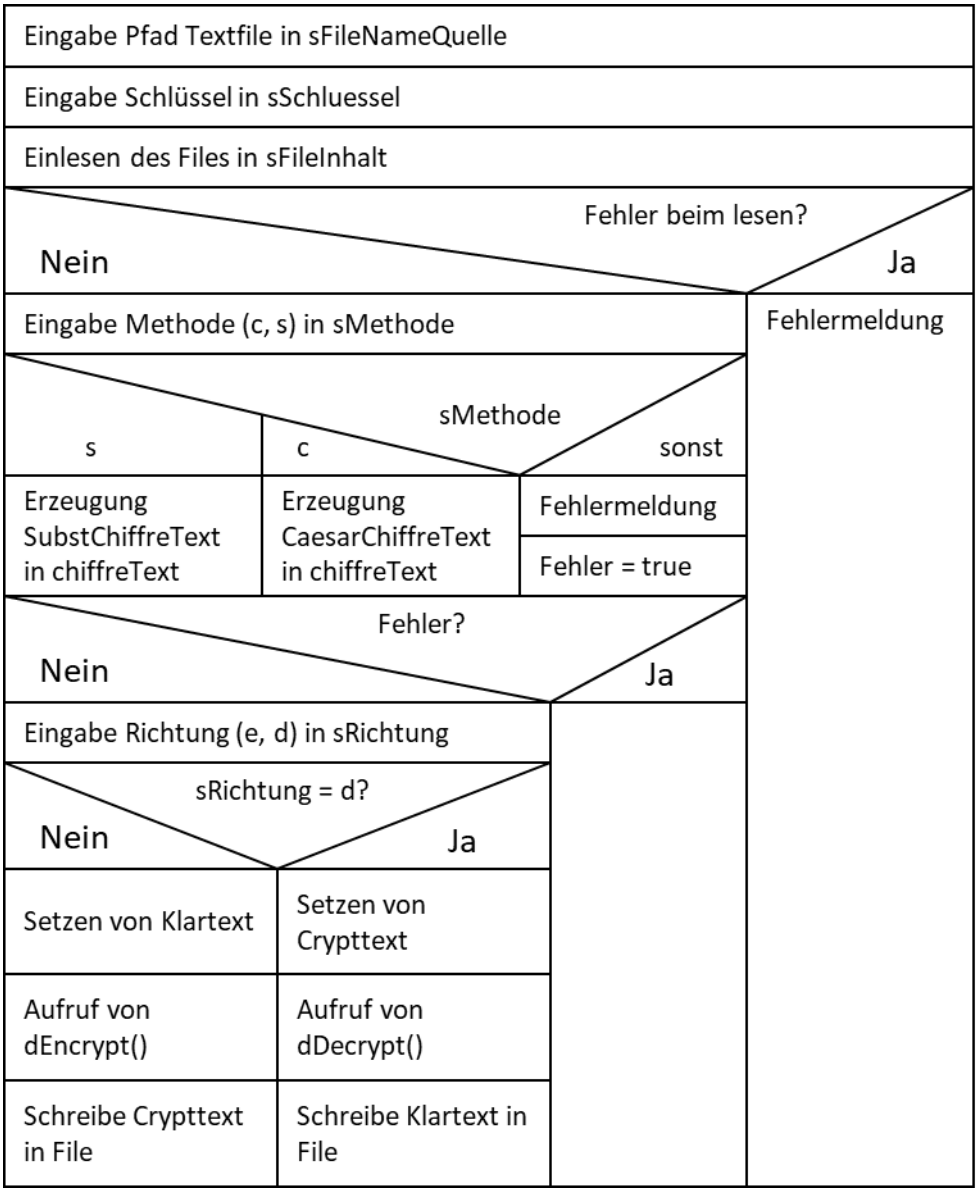
Der Garbage Collector wiederum hat den Überblick über alle Objekte im Speicher und prüft, ob es Objekte gibt, auf die kein anderer Programmteil eine Referenz besitzt. Das kann man sich so vorstellen, dass der Garbage Collector prüfen kann, ob es Telefonnummern gibt, welche in keinem Adressbuch der Welt auftauchen, also auch keiner diese Telefonnummer anrufen kann. Wenn er solche Objekte findet, dann werden sie automatisch freigegeben – also „entsorgt“ und wir müssen uns somit darum nicht mehr kümmern. Wenn wir dies wieder mit unserem Telefonbeispiel betrachten ist das aber auch kein Problem – eine Telefonnummer die keiner anruft wird auch nicht benötigt! Das ist natürlich vorteilhaft für den Programmierer. Der Nachteil jedoch ist, dass dieser Prozess das Performanceverhalten der JVM negativ beeinflusst. Gerade bei sehr performancekritischen Programmen wie 3D Spielen versucht man die Menge an Objekten, welche nur über einen begrenzten Zeitraum verwendet werden, zu minimieren. Für die allermeisten unserer Programme gilt aber, dass wir uns darüber keine Gedanken machen müssen.

7 Die Implementierung der Logik

Nun wollen wir uns der eigentlichen Implementierung widmen. Unser Ziel ist es, ein funktionsfähiges Programm zu erstellen, welches die Chiffrierung und Dechiffrierung umsetzt und auch testet. Insofern teilen wir unser Programm in zwei Teile. Zum einen haben wir die drei oben beschriebenen Klassen, welche die Chiffrierlogik umsetzen und zum anderen das (Haupt-)Programm, welches die Klassen nutzt. Auch wenn es jetzt paradox klingt – das Hauptprogramm ist für uns nur eine Nebensache. Wir wollen primär die Logik der Chiffrierung implementieren und das sogenannte Hauptprogramm ist lediglich dafür da, die Logik in einer Testumgebung laufen zu lassen! Wenn wir die Logik erfolgreich getestet haben, können wir die Chiffrierklassen in jedes beliebige Java Programm integrieren.

7.1 Main Methode im Hauptprogramm

Das Hauptprogramm soll in einer Klasse namens „TestChiffre“ laufen. Hier werden alle wichtigen Schritte für die Nutzung unserer Chiffrierungsklassen durchgeführt



Nach der Erzeugung des Objektes SubstChiffeText bzw. CaesarChiffreText und Ablegen in der Variablen „chiffreText“ wird vom Handling her nicht mehr zwischen den beiden implementierten Algorithmen unterschieden. Intern weiß das Programm nun selbst, welcher Algorithmus aufgerufen wurde und welcher entsprechend auszuführen ist.

7.2 Allgemeines Verhalten

Um den Code nicht zu umfangreich werden zu lassen, werden Großbuchstaben wie Kleinbuchstaben behandelt. Bei der Ausgabe wird der komplette Text in Kleinbuchstaben ausgegeben. Es werden nur die Buchstaben ‚a‘ bis ‚z‘ berücksichtigt. Alle anderen Zeichen werden 1:1 übernommen.
Die Ausgabefiles werden am gleichen Ort gespeichert wie die Quellfiles, lediglich mit einem Zusatz „clr“ für Klartext und „crp“ für Encrypted.

Zusatzinfo:
Den Filenamen ohne Pfad bekommt man, wenn man ein Objekt der Klasse „File“ erzeugt und die Methode „getName()“ aufruft. Der Gesamtpfad kann über die Methode „getAbsolutePath()“ ermittelt werden.

7.3 Parameter von Caesar

Wie wir bereits weiter oben festgelegt haben, wird der Caesar Algorithmus lediglich mit einem Verschiebungsoffset belegt, so dass er die Verschlüsselung vornehmen kann. Der Parameter ist aber „lediglich“ ein String. Wir vereinbaren, dass der übergebene String immer ein einzelnes Zeichen ist und zwar der Buchstabe, mit dem das „a“ ersetzt wird. Dadurch können wir den Offset berechnen.
Wenn wir bspw. den Parameter „c“ angeben, dann wird aus dem ‚a‘ ein ‚c‘ => der Offset beträgt also 2. Diese Berechnung erfolgt in der Methode „calcOffset()“ und kann direkt im Konstruktor aufgerufen werden.

7.4 Parameter von Substitution

Bei der Substitution muss für jedes Zeichen zwischen ‚a‘ und ‚z‘ ein Tauschpartner angegeben werden. Insofern vereinbaren wir, dass der Schlüssel für die Substitution ein String von immer exakt 26 unterschiedlichen Buchstaben zwischen ‚a‘ und ‚z‘ sein muss. Aus dieser Information wird in der Methode „buildReplacements()“ das Array für die Ersetzungen erzeugt.

8 Lizenz