	Base 64		AnPr
	Name	Klasse	Datum

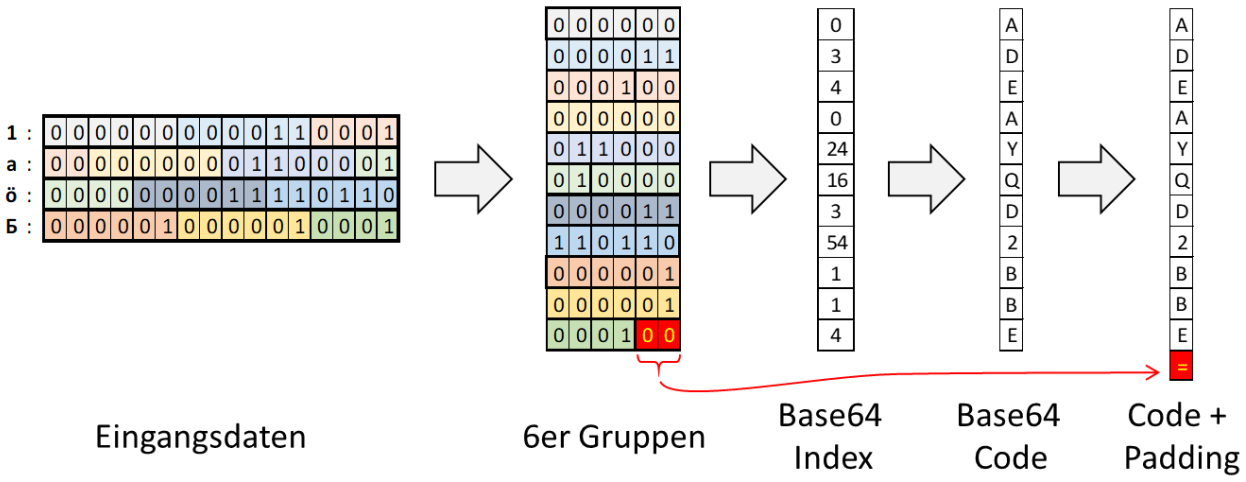
1 Situation

Bei der Übertragung von Informationen gibt es das Problem, dass Steuerzeichen beim Versenden von Daten in der „Payload“ – also den zu sendenden Informationen stecken können und somit die Übertragung zu unvorhergesehenen Fehlern führt. Um dies zu verhindern, hat man ein Verfahren definiert, wie beliebige Daten – also auch Binärdaten – in ein Textformat von 64 Basiszeichen überführt werden können: „Base64“. In der nebenstehenden Tabelle sind die definierten Zeichen für die Codierung.

Zeichen	Position
A-Z	0-25
a-z	26-51
0-9	52-61
+	62
/	63

Der Prozess sieht folgende Schritte vor:

- Bits der Eingangsdaten werden links beginnend in 6er Gruppen zerteilt.
- Ist die letzte Gruppe < 6 Bits, so wird rechts mit 0 aufgefüllt („Padding“).
- Interpretation der Bits aus den 6er Gruppen als Integer.
- Integer Werte werden als Index der Codierungstabelle verwendet („Base64 Index“).
- Zeichen aus Tabelle werden als ASCII Zeichen aneinandergehängt („Base64 Code“).
- Für je zwei Padding-Bits wird ein „=“ als Paddingindikator hinzugefügt.



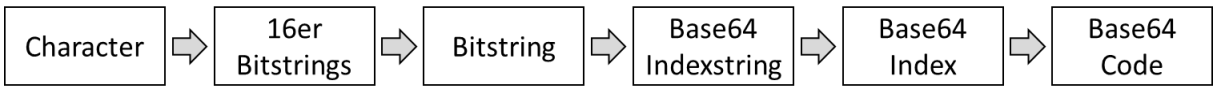
2 Mögliche Eingangsformate

Im Prinzip können wir erstmal reine Binärdaten, so wie wir sie bspw. aus einem File bekommen, codieren. Dabei ist es dann unerheblich, ob es sich um Textdaten im ASCII, UTF8 oder einer 8Bit Codierung handelt oder um „normale“ Binärdaten wie bspw. Bilder. Die andere Möglichkeit wäre, Strings direkt in unserem Programm zu codieren. Java nutzt für seine Character die ersten beiden Bytes von UTF16, so wie es oben dargestellt ist – insofern haben wir pro Zeichen 16 Bit.

Für unsere Umsetzung gehen wir entsprechend erstmal von Strings aus.

3 Einfache Lösung für Codierung

Ein relativ einfaches Vorgehen ist, die Base64-Codierung im String-Format vorzunehmen.



Die Binärdaten aus den Eingangsdaten können mit `Integer.toBinaryString()` erzeugt und danach komplett in einem langen String aneinandergehängt werden. Nun können die einzelnen 6er Gruppen als Base64 Indizes mit Hilfe von `substring()`extrahiert und über `parseInt()` wieder zu einer Zahl („Base64 Index“) umgewandelt und von dort zum Base64 Code konvertiert werden.

Ermitteln Sie die Bits des Strings `"5XäΦ"` und tragen Sie diese rechtsbündig unten ein. Alle Felder müssen danach mit 0 oder 1 gefüllt sein:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
'5'																
'X'																
'ä'																
'Φ'																

Wenn wir den Algorithmus nun umsetzen wollen, benötigen wir eine Funktion, welche die 0-Werte links auffüllt. Erstellen Sie eine Methode, welche einen String und die gewünschte Länge als Parameter erwartet und als Rückgabewerte den übergebenen String bis zur Länge links mit 0-Werten auffüllt:

```
System.out.println(fillWithZeroLeft("1100001", 16));
      0000000001100001
```

Anmerkung: für den ersten Prototypen ist es vorerst in Ordnung, die Strings mit `+=` zu verketten. Wir werden uns später um eine Alternative kümmern.

Nun beginnen wir mit der eigentlichen Codierung. Erstellen Sie zuerst einen Algorithmus, welcher für einen beliebigen Eingangsstring alle auf 16 aufgefüllten Bits in einen neuen String zusammenfügt. Da wir mittels Modulo feststellen können, ob die Länge des Strings durch 6 teilbar ist, können wir diesen String gleich um die notwendige Anzahl von nullen rechts so ergänzen, dass wir den Gesamtstring später ohne Rest in 6er Gruppen aufteilen können. Nun können wir diese 6er Gruppen erstmal nur zur Kontrolle ausgeben.

Info: Die Bitreihenfolge unserer Zahlen ist „Big Endian" was bedeutet, dass die rechte Position die Einerstelle ist, links daneben die Zweier etc. Wenn wir aber die Stringpositionen mittels Substring ermitteln, ist es genau anders herum – `substring(0,6)` liefert uns die linken sechs Stellen – also die höherwertigen Bits.

```
textToBase64("1aöБ");
      000000
      000011
      000100
      000000
      011000
      010000
      000011
      110110
      000001
      000001
      000100
```

Als nächstes benötigen wir eine Funktion, welche uns die Base64 Zuordnungstabelle in einem `char[64]` Array liefert. Wir könnten sie zwar hart codieren – nachdem die Tabelle jedoch systematisch aufgebaut wird, erledigen wir dies durch ein kleines Programm.

```
char[] encoding = buildEncoding();
```

Nun können wir, anstatt die Zahlen auf der Konsole auszugeben, den Binärstring als Zahl parsen und mittels der encoding Tabelle die Codierung durchführen und den Base64 String ausgeben:

```
textToBase64("1aöБ");
      ADEAYQD2BBE=
```

Hinweis: Mittels `Integer.parseInt(myInt, 2)` können wir einen String mit Einsen und Nullen als Binärwert parsen.

4 Einfache Lösung für Decodieren

Nun gehen wir anders herum vor und decodieren den String „ADEAYQD2BBE=“ wieder in den Originalstring:

- Umkehr der Tabelle, so dass mit dem Character der Index ermittelt werden kann (bspw. mit `Hashtable`).
- Ermittlung der Anzahl Füllbits.
- Ermittlung aller Base64 Indexwerte als Bits aus den Characterwerten und Aneinanderreihung in einen String.
- Löschen der Füllbits.
- Zerlegung des Strings in 16er Gruppen, Umwandlung in Charwerte und Verkettung zum String.

5 Test

Testen Sie Ihre Applikation mit Hilfe von Zufallsstrings. Erstellen Sie hierfür eine Funktion, welche einen zufälligen String der Länge `size` aus dem gesamten möglichen Wertebereich von Character erstellt:

```
private static String buildRandomString(int size)
```

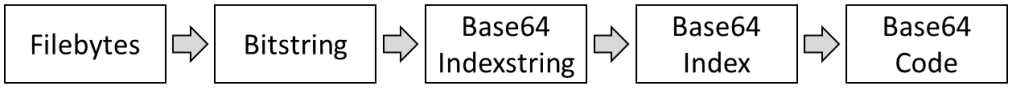
Hinweis: Zufallszahlen können Sie mit Hilfe von `Random` erstellen. Erzeugen Sie eine `int` Zufallszahl zwischen 0 und 65536 und casten sie auf `char`.

Erstellen Sie nun einen Algorithmus, der 1000-mal ein Zufallsstring zwischen 10 und 500 Zeichen erstellt, diesen Codiert, das Ergebnis decodiert und dann vergleicht, ob das Decodier-Ergebnis dem ursprünglichen String entspricht.

6 Filezugriff

Nun ändern Sie das Programm derart ab, dass ein Filestream gelesen, decodiert und als Textfile geschrieben wird. Hier gibt es aber eine wichtige Änderung. Wir lesen das File auf Binärebene aus – also Byteweise (nicht in zwei Byte-Stücken, wie im vorausgegangenen Beispiel). Das bedeutet, dass wir auf die Umwandlung von Zeichen in Binärzahlen verzichten können, da wir ja bereits Binärdaten als Bytes erhalten. Insofern ist die Vorgehensweise jetzt:

- Lesen der Bytes aus dem Filesystem.
- Umwandlung des Bytes in einen String von 1en und 0en (inklusive linkes Auffüllen mit 0 bis zur Länge 8).
- Padding, Zerlegung in 6er Binärwerte und Codierung mittels Tabelle (wie im vorherigen Algorithmus auch).



Achtung: wenn Sie bspw. das Bitmuster 10101011 einer Bytevariablen ausgeben, erhalten Sie den Dezimalwert -85. Die Wrapperklasse `Byte` besitzt aber nicht die Methode `toBinaryString()`, das finden wir nur bei `Integer`. Damit Sie aber die Integer-Methode nutzen können, müssen Sie den Bytewert -85 in eine Integerzahl umwandeln, was aber Bitmuster 11111111111111111111111110101011 zur Folge hat, was wiederum logischerweise auch -85 in `Integer` ist. Wenn Sie aber nur an den relevanten Bits 10101011 interessiert sind, müssen Sie diese von rechts „herausschneiden“, oder mittels `Byte.toUnsignedInt(myByte)` casten.

Beim Decodieren fällt entsprechend die Umwandlung von 16er Bitsequenzen in Character ebenfalls weg, da wir hier ja in 8er Bitschritten vorgehen, die wiederum direkt ins Filesystem geschrieben werden können. Damit muss das decodierte File wieder die gleiche Struktur haben, wie das originale.

Nehmen Sie ein Zip-File, codieren und decodieren Sie es wieder. Prüfen Sie anschließend, ob es sich wieder öffnen lässt. Zur Kontrolle schreiben Sie auch den Base64 Code in ein extra Textfile.

Hinweis: Für den Umgang mit Binärfiles müssen Sie den `InputStreamReader` bzw. `InputStreamWriter` verwenden.

7 Performanceoptimierung, Teil 1

Wenn Sie nun ein größeres File codieren und decodieren merken Sie, dass der Algorithmus extrem langsam ist. Die Codierung und Decodierung des Zipfiles für die Übungsdatenbank (oder eines beliebigen anderen Zipfiles mit ca. 50kB Größe) hat auf meinem Rechner bspw. 2,002 Sekunden gedauert. Dies hat mehrere Gründe:

1. Die Verkettung auf Stringebene ist sehr langsam, da mit jeder neuen Verkettung ein neues String Objekt erzeugt und das alte gelöscht werden muss (es sein denn, Sie haben gleich auf den `StringBuilder` zurückgegriffen). Dies liegt an der „immutable“ Eigenschaft von Strings (zu Deutsch „unveränderlich“)
2. Trotz einer relativ hohen Performance der `Hashtable` dauert das Decodieren aufgrund der vielen Zugriffe zur `Hashtable` relativ lange.

Beginnen wir mit der Optimierung auf Stringebene. Hier bietet sich der `StringBuilder` an. Dies ist ein dynamisch veränderbares Objekt, welches einen inkrementellen Aufbau ermöglicht. Ersetzen Sie alle Stellen, bei denen der String mittels += aufgebaut wird durch einen `StringBuilder` und wandeln Sie das `StringBuilder` Objekt anschließend mit `toString()` in einen `String` um.

Geben Sie die Verarbeitungszeit Ihrer Algorithmen für mit und ohne `StringBuilder` an:

Version:	**Zeit in Millisekunden:**
String	
Stringbuilder	

Wie wir sehen, ist die Performance schon bedeuten besser geworden. Codieren/decodieren wir nun das File BigDB.zip (oder ein beliebiges anderes Zipfile mit ca. 17MB Größe). Hier erhalte ich auf meinem Gerät eine Verarbeitungszeit von 1,564 Sekunden.

Versuchen wir nun, die `Hashtable` zu ersetzen. Wenn wir uns die erlaubten Codierungs-Character ansehen, ist der höchste ASCII Zahlenwert die 122 für das kleine 'z'. Insofern können wir auch ein String-Array mit 123 Elementen erzeugen, und somit direkt mit dem Character-Wert auf den Index gehen:

```
decoding[c]
```

Version:	**Zeit in Millisekunden:**
Hashtable	
Stringarray	

8 Performanceoptimierung, Teil 2

Die Nutzung eines Stringbuilders hat die Performance also am meisten beeinflusst. Trotzdem ist es noch nicht der Idealzustand. Die Daten, welche wir verarbeiten sind ja bereits als Binärcodes im Rechner vorhanden. Es ist demnach nicht zielführend, die Binärdaten in einen String zu verwandeln, dort binär zu interpretieren, um aus ihnen anschließend wieder Binärdaten zu erzeugen. Besser wäre es, gleich im Binärformat zu bleiben.

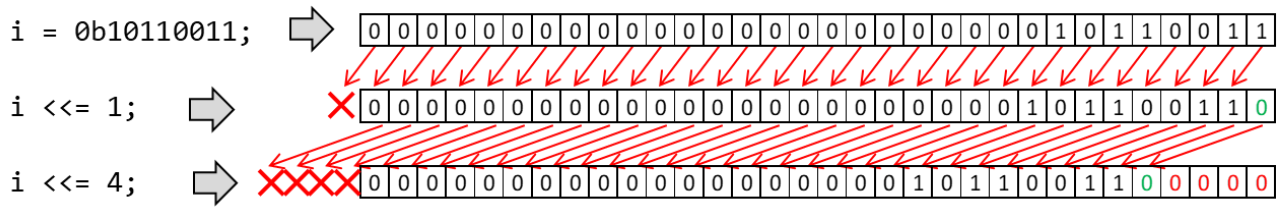
Hierzu benötigen wir aber einige „Werkzeuge“. Im Wesentlichen ist es „Bitshift“ und das Herausschneiden und Setzen beliebiger Bereiche aus einer Binärzahl.

8.1 Bitshift

Beginnen wir mit Bitshift, was eine Verschiebung der Binärdaten um n Stellen bedeutet. Der Operator in Java (und den meisten anderen Programmiersprachen) ist >> für Bitshift nach rechts und << für Bitschift nach links. Folgender Code:

```
i <<= 1
```

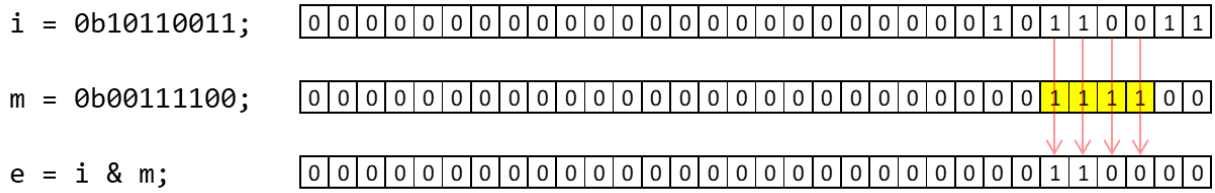
Schiebt die Bits der Zahl in der Variablen i um eine Stelle nach links, `i <== 4` entsprechend um 4 Stellen.



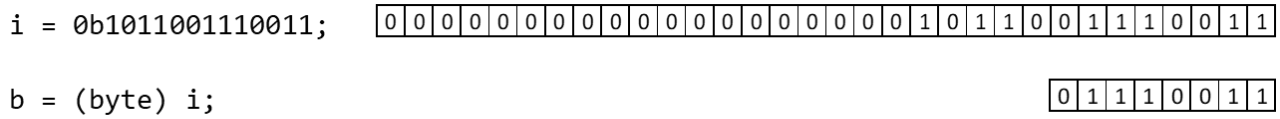
Wie wir sehen, werden die stellen rechts mit 0 aufgefüllt und die linken Stellen „verschwinden“. Beim Bitshift nach rechts verschwinden die rechten Stellen und die linken werden aufgefüllt. Hier gilt in Java aber, dass negative Zahlen mit 1 und positive mit 0 aufgefüllt werden.

8.2 Bits „herausschneiden“

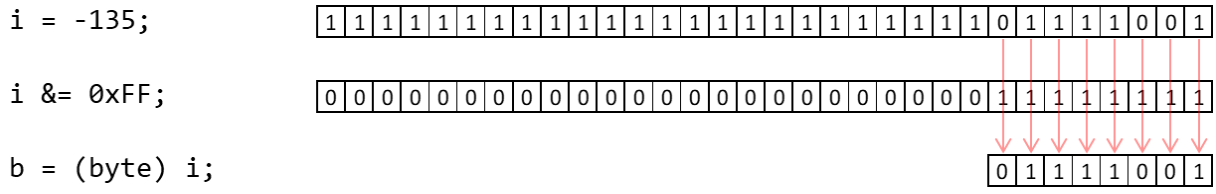
Hier gibt es zwei Ansätze – erstens einen Teil an beliebiger Position extrahieren. Hierfür nutzen wir das bitweise UND (Operator „&“) mit einer Maske.



Der zweite Ansatz ist der Typecast auf einen „kleineren“ Datentyp:

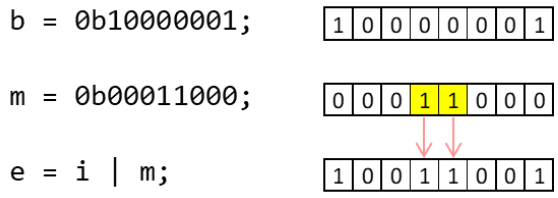


Das funktioniert jedoch nur bei positiven Zahlen. Bei negativen Zahlen muss der Weg über `Integer.toUnsignedLong(myByte)` gecastet werden, oder – wenn wir wirklich zwingend einen Bytewert benötigen über eine Maske mit anschließendem Typecast:

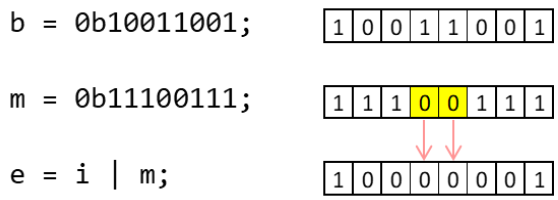


8.3 Bits „setzen“

Hiermit können wir an einer beliebigen Stelle einer Bitsequenz einzelne Bits auf 1 oder 0 setzen. Beginnen wir mit dem Setzen auf 1. Dies wird durch ein bitweises ODER (Operator „|“) durchgeführt:

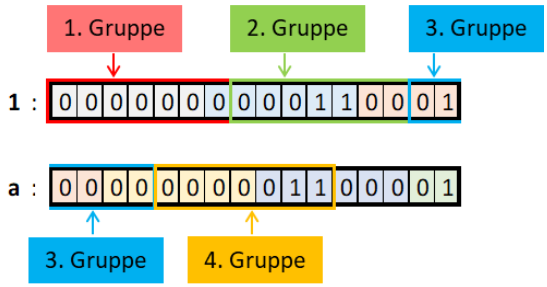


Das Setzen auf 0 kann wieder mit dem bitweisen UND arbeiten:



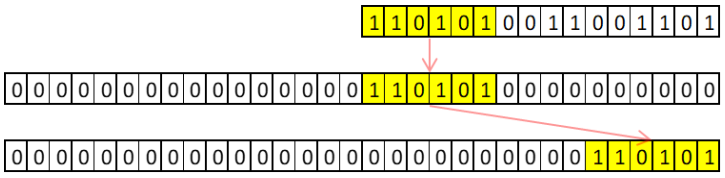
8.4 Umsetzung Codieren

Bei der Stringlösung konnten wir in einer Variablen eine beliebig große Anzahl an Nullen und Einsen speichern. Wenn wir aber auf Binärebene arbeiten können wir keinen Datentypen definieren, der beliebig viele Bits aufnehmen kann. Wir müssen die 6er Gruppen also direkt aus den Rohdaten ermitteln:

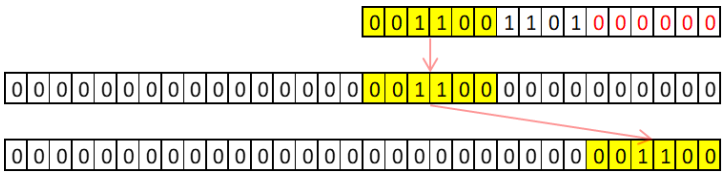


Die Bildung der 6er Gruppen muss also über die Grenzen der einzelnen Eingangselemente hinweggehen, wie wir es bei der 3. Gruppe sehen – der Algorithmus wird somit komplizierter.

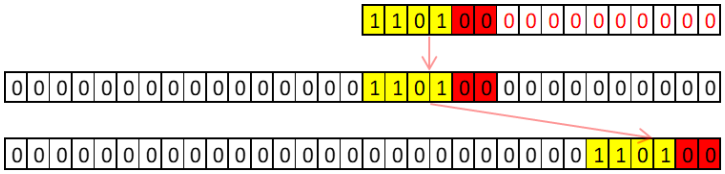
Da wir von links nach rechts vorgehen müssen, schneiden wir zuerst die linken 6 Bits heraus und schieben sie an die Bitpositionen 0-6:



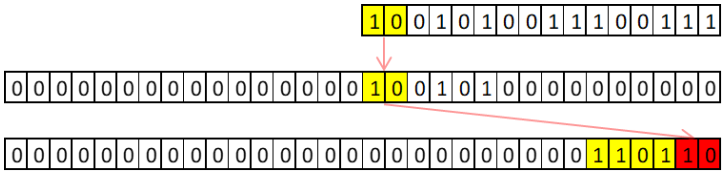
Danach schieben wir um 6 Positionen nach links und wiederholen das Ganze:



Bei der dritten Wiederholung haben wir jedoch zwei Bits zu wenig:



Also müssen wir diese vom nächsten Characterwert übernehmen:



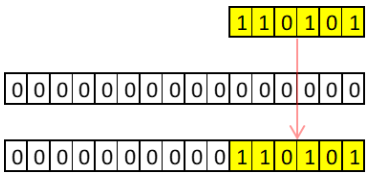
Nun darf im Character der Bitshift natürlich nur um 2 nach links erfolgen. Somit haben wir auch beim letzten Character gleich die rechten Bits als Paddingbits korrekt eingefügt. Hier müssen wir uns nur merken, wie viele Paddingbits übriggeblieben sind.

Setzen Sie nun den Algorithmus um und prüfen Sie, ob beim String "1aöB" der korrekte Code "ADEAYQD2BBE=" ermittelt wird.

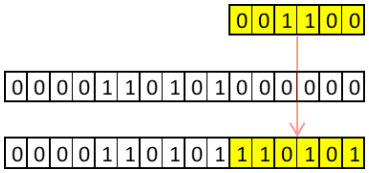
Anmerkung: Unterschätzen Sie den Algorithmus nicht. Sie müssen mit einigen Hilfsvariablen arbeiten, welche vor allem den Übertrag sauber verarbeitbar machen!

8.5 Umsetzung Decodieren

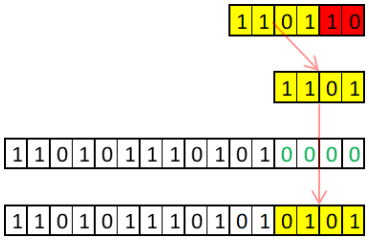
Auch beim Decodieren müssen wir uns um den Übertrag kümmern – diesmal eben anders herum. Gehen wir es an einem Beispiel wieder durch. Wir verarbeiten das erste Byte unserer Base64 Codes, indem wir das Bitmuster der ersten 6 Bits in unsere erstmal leere Zielvariable übernehmen:



Danach schieben wir in der Zielvariablen die Bits um sechs stellen nach links und verarbeiten das nächste Byte:



Nun müssen wir aufpassen! Wir dürfen die Bits der Zielvariablen jetzt nur um 4 Positionen nach links schieben, da sonst die linken zwei Bits verschwinden. Weiterhin müssen wir nun im aktuell verarbeiteten Byte nur die linken vier Bits übernehmen:



Als nächstes müssen wir die restlichen zwei Bits (oben rot markiert) in die nächste Zielvariable schieben. Wenn wir davon ausgehen, dass wir bei den Zielvariablen immer eine gerade Anzahl an Bits erwarten können, dann gibt es hier nur zwei mögliche Konstellationen – wir müssen zwei Bits übernehmen, oder vier. Insofern können wir für die restlichen Bits zwei Masken vorbereiten: `0b11` und `0b1111`.

Codieren/decodieren wir nun das File BigDB.zip (oder ein beliebiges anderes Zipfile mit ca. 17MB Größe) nochmal. Die Verarbeitungszeit nach der Hashtable-Optimierung war mit den Strings 1,176 Sekunden. Tragen Sie nun die Zeit mit der Binärverarbeitung ein:

Version:	**Zeit in Millisekunden:**
Binärdaten	

9 Standard Java Bibliotheken

Diese Übung hatte zum Ziel, Sie beim Thema „Algorithmen“ weiterzubringen, weniger Ihnen eine Funktion für die Base64 Konvertierung an die Hand zu geben. Für Standardprobleme lohnt es sich immer zu prüfen, ob es bereits fertige Bibliotheken für deren Lösung gibt. Da die Base64 Codierung eben solch ein Standard ist verwundert es nicht, dass entsprechende fertige Lösungen existieren. Die einfachste Möglichkeit ist es, die `Base64` Implementierung aus der `java.util` Bibliothek zu verwenden. Für die Codierung und Decodierung kann folgender Code verwendet werden:

```
String sCode = Base64.getEncoder().encodeToString(myByteArray);
byte[] result = Base64.getDecoder().decode(myBase64String);
```

Verwenden Sie die Bilbiothek und machen Sie wieder eine Performancemessung:

Version:	**Zeit in Millisekunden:**
Java.util.Base64	

Die Java-Umsetzung ist also nochmal schneller als unser Algorithmus.