Programmieren II (Java)

3. Praktikum: Arrays und Strings

Sommersemester 2024
Christopher Auer

Abgabetermine

Lernziele

- Arrays: Erstellung, Zugriff und Literale
- `Strings`: arbeiten mit `Strings` und `StringBuilder`

Hinweise

- Sie dürfen die Aufgaben *alleine* oder zu *zweit* bearbeiten und abgeben
- Sie müssen *4 der 5* Praktika bestehen
- *Kommentieren* Sie Ihren Code
  - Jede *Methode* (wenn nicht vorgegeben)
  - *Wichtige* Anweisungen/Code-Blöcke
  - *Nicht kommentierter* Code führt zu *Nichtbestehen*
- Bestehen Sie eine Abgabe *nicht* haben Sie einen *zweiten Versuch*, in dem Sie Ihre Abgabe *verbessern müssen*.
- *Wichtig*: Sie sind einer *Praktikumsgruppe* zugewiesen, *nur* in dieser werden Ihre Abgaben *akzeptiert*!

Aufgabe 1: Erweiterte Caesar-Chiffre ★☆ bis ★★

In dieser Aufgabe implementieren wir eine *erweiterte* „Caesar-Chiffre“ um geheime Nachrichten zu verschlüsseln und entschlüsseln.

Hinweis: Das Internet ist voll von Implementierungen der Caesar-Chiffre in allen möglichen Sprachen. Versuchen Sie trotzdem die Lösung so weit es irgendwie geht *selbstständig zu erarbeiten*. Die Praktikumsleiter werden Fragen zu Ihrer Lösung stellen, die Sie nur beantworten können, wenn Sie Ihre eigene Lösung wirklich verstanden haben. Und haben Sie das Thema dieser Aufgaben verstanden, haben Sie die Chance auf *geheime Hinweise zum Bestehen der Klausur*!

Bevor es losgeht

Importieren Sie das Projekt in `SupportMaterial/caesar` in VSCode. Das Projekt enthält bereits ein Grundgerüst in der Datei `Caesar.java` mit einer `main`-Methode zum Testen. Machen Sie sich mit dem Programm vertraut: Zur Einfachheit ver-/entschlüsseln wir im Folgenden *nur Großbuchstaben*. Bei der (erweiterten) Caesar-Chiffre handelt es sich um eine einfache (und *unsichere*) Substitutions-Chiffre, die jeden Buchstaben durch einen anderen ersetzt, z.B., `'A'` durch `'M'`, `'B'` durch `'I'`, usw. Zum Ver-/Entschlüsseln benötigen wir einen *Schlüssel*, den wir als `String` darstellen und der die Abbildungen definiert. Ein Beispiel hierfür ist `Caesar.EXAMPLE_KEY`:

```
EXAMPLE_KEY  =  HAMNOGPQFEISTLJVWDXKCYZRBU
ABC          =  ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Der `String ABC` beinhaltet das Alphabet. Damit wird `A` auf `B` abgebildet, `B` auf `Y`, `C` auf `U`, usw. D.h. zuerst wird wird im Schlüssel die Stelle gesucht, an der das zu verschlüsselnde Zeichen (z.B. `W`) steht. Die entsprechende Stelle im Alphabet (`ABC`) entspricht dem verschlüsselten Zeichen (`Q`). Ein Schlüssel ist eine *Permutation* des *Alphabets.*

Caesar-Chiffre ★☆

Implementieren Sie die Methode `applyCaesar(text, key, clear)`, die den `String text` mit Hilfe des Schlüssels `key` verschlüsselt. Gehen Sie dabei Zeichen für Zeichen durch den `text` und wenden Sie den Schlüssel `key` wie folgt an:

- Für jedes Zeichen **`char`** `c` ermitteln Sie *an welchem Index* `c` im `String key` steht Sollte das Zeichen nicht in `key` vorkommen, dann wird das Zeichen *nicht verschlüsselt* sondern einfach an das verschlüsselte Ergebnis *unverschlüsselt* angehängt.
- Die Substitution von `c` steht an vorher ermitteln Index im `String clear` (`clear` entspricht dem Alphabet und wird beim Verschlüsseln der `String ABC` sein)
- Hängen Sie das substituierten Zeichen an den verschlüsselten String an.

Verwenden Sie für die Erzeugung des verschlüsselten Strings einen `StringBuilder`!

Für das angegebene Beispiel in der `main`-Methode ist die verschlüsselte Nachricht:

```
GXEFXBCCKJXJD KD OBPB CBUAM DEUA CJAX LGBLL BNL KD GVMAED ERJX U
```

Hinweise:

- Wenn Sie `applyCaesar` mit vertauschten Parametern `key` und `clear` aufrufen, so wird die Nachricht *entschlüsselt*!
- Betrachten Sie sich die Dokumentation von `String` und `StringBuilder` für geeignete Methoden.

Häufigkeitsanalyse

Die Caesar-Chiffre kann man mit Häufigkeitsanalyse *knacken*. Dazu muss man zunächst die *relative Häufigkeit* (in Prozent) eines jeden Buchstabens im *verschlüsselten Text* berechnen. Für den Beispieltext in der `main`-Methode ergibt sich:

```
A: 7.55%
B: 11.32%
C: 7.55%
...
Z: 0.00%
```

Implementieren Sie die Methode `printStatistics`, die diese Ausgabe für den übergebenen verschlüsselten `String` wie gezeigt ausgibt!

Optional: Geheime Hinweise für die Klausur entschlüsseln

Vielleicht ist Ihnen schon aufgefallen, dass eine weitere *geheime Botschaft* in `Caesar.java` versteckt ist. Der String `EXAM_HINTS_ENCRYPTED` beinhaltet *geheime Hinweise zum **Bestehen der Klausur*** dieser Veranstaltung! Mit Hilfe von Häufigkeitsanalyse können Sie den Schlüssel ermitteln. Betrachten Sie sich dazu die Buchstabenhäufigkeiten im Deutschen und vergleichen Sie sich mit den Häufigkeiten der Buchstaben in `EXAM_HINTS_ENCRYPTED`. Der häufigste Buchstabe in der deutschen Sprache *entspricht* dem häufigsten Buchstaben in `EXAM_HINTS_ENCRYPTED`. Das gleiche gilt für den zweihäufigsten, usw.

Aufgabe 2: Bingo! bis

In dieser Aufgabe implementieren wir eine „Bingo!-Karte“. In Bingo! bekommt jeder Spieler eine Bingo!-Karte mit Zahlen von 1 bis 75 in einem 5×5-Schema angeordnet:

```
 B  I  N  G  O
 9 25 31 54 72
 7 22 37 47 65
14 26    58 62
 6 27 41 48 68
15 17 36 46 64
```

Eine weitere Person zieht zufällig Zahlen und gibt diese für jeden bekannt, z.B. 25., die dann auf der Karte weggestrichen wird (falls vorhanden):

```
 B  I  N  G  O
 9    31 54 72
 7 22 37 47 65
14 26    58 62
 6 27 41 48 68
15 17 36 46 64
```

Dies passiert solange, bis jemand

- eine komplette *Zeile*
- oder komplette *Spalte*
- oder eine der beiden *Diagonalen*

gestrichen hat, z.B., nach den Zahlen 15, 72, 47, 27 sieht die Karte wie folgt aus:

```
 B  I  N  G  O
 9     31 54
 7 22 37     65
14 26     58 62
 6     41 48 68
   17 36 46 64
```

Der Spieler ruft „Bingo!“ und gewinnt die Runde.

Wir programmieren in diesem Praktikum eine Konsolenanwendung, in der der Spieler die Zahlen eingibt, die automatisch auf der Bingo!-Karte gestrichen werden. Zudem teilt uns das Programm mit, wenn wir „Bingo!“ rufen dürfen!

Bevor es losgeht

Importieren Sie das Projekt in `SupportMaterial/bingo` in VSCode. Das Projekt enthält eine JUnit-Testklasse in der alle Testmethoden *auskommentiert* sind. Testen Sie im Folgenden jeder der Methoden, die Sie implementieren, indem Sie die entsprechende Testmethode *einkommentieren*. *Achtung*: Im Praktikum werden nur Abgaben akzeptiert, bei der *alle Tests unverändert durchlaufen*!

Bingo!-Beispiel ★

Wir modellieren eine Bingo!-Karte als *zweidimensionales* **`int`**-Array der Größe 5×5. Die Einträge entsprechen den Zahlen, wobei 0 heißt, dass die Zahl bereits gestrichen wurde. Der erste Index des Arrays gibt die Zeile an, der zweite Index die Spalte. Erstellen Sie eine Klasse `Bingo` und definieren Sie darin ein *statisches, unveränderliches* Attribut `BINGO_EXAMPLE` das Sie mit Hilfe eines *Array-Literals* mit folgenden Werten initialisieren:

```
 B  I  N  G  O
 4 27 32 55 73
15 25 41 58 75
 8 26     59 70
 7 22 33 54 63
13 17 43 48 67
```

Testen Sie Ihre Deklaration mit Hilfe der Testmethode `testBingoExample`!

Ausgabe ★–★★

Implementieren Sie eine *öffentliche, statische* Methode `printBingoCard`, die eine Bingo!-Karte als Parameter aufnimmt und auf der Konsole wie oben gezeigt ausgibt! Implementieren Sie außerdem eine `main`-Methode, in der Sie das Bingo!-Beispiel von oben ausgeben. Wir werden die `main`-Methode im Folgenden erweitern. Für diese Methoden gibt es keine JUnit-Tests.

Auf Duplikate prüfen ★–★★

Eine Bingo!-Karte darf *keine Duplikate* von Zahlen enthalten. Implementieren Sie eine Methode `containsDuplicates`, die für eine übergene Bingo!-Karte **`true`** zurückgibt, wenn diese *Duplikate* enthält, sonst **`false`**! Testen Sie Ihre Methode mit `testContainsDuplicates`.

Gültige Bingo!-Karten ★★

Der zweidimensionale Array, mit der wir eine Bingo!-Karte modellieren ist gültig, wenn...

- es nicht **`null`** ist
- keine Zeile **`null`** ist
- die Bingo!-Karte ein *zweidimensionaler quadratischer* Array der Dimension 5×5 ist.

Zudem gibt es folgende Eigenschaften, die eine gültige Bingo!-Karte erfüllen muss:

- Der Eintrag in der Mitte ist immer *gestrichen*.
- Für die *Spalten* gilt:
  - 1. Spalte enthält nur Zahlen von 1 bis 15 (inklusive)
  - 2. Spalte von 16 bis 30
  - 3. Spalte von 31 bis 45
  - 4. Spalte von 46 bis 60
  - 5. Spalte von 61 bis 75
- Die Karte enthält *keine Duplikate* (dies können Sie mit `containsDuplicates` prüfen).

Implementieren Sie eine Methode `checkBingoCard`, die die obigen Eigenschaften für eine übergebene Bingo!-Karte prüft und bei *nicht Erfüllung* eine `IllegalArgumentException` mit *aussagekräftiger* Fehlermeldung erzeugt! Testen Sie Ihre Methode mit `testCheckBingoCard`.

Zahlen streichen ★★

Implementieren Sie eine Methode `fillBingoCard(bingoCard, number)`, die zunächst mit `checkBingoCard` prüft ob die Bingo!-Karte gültig ist und `number` zwischen 1 und 75 liegt! Danach wird `number`, wenn vorhanden, von `bingoCard` gestrichen. Testen Sie Ihre Methode mit `testFillBingoCard`.

Bingo! prüfen ★★

Implementieren Sie eine Methode `bingo(bingoCard)`, die **`true`** zurückgibt, wenn ein Bingo! erreicht wurde, d.h., eine Spalte, eine Zeile oder eine der Diagonalen komplett gestrichen wurden! Testen Sie Ihre Methode mit `testBingo`.

`main`-Methode vervollständigen ★★

Vervollständigen Sie Ihre `main`-Methode wie folgt: Deklarieren Sie eine *lokale Variable* `bingoCard` und initialisieren Sie sie mit `BINGO_EXAMPLE`. Zeigen Sie die aktuelle `bingoCard` an und lassen Sie den Nutzer eine Zahl zwischen 1 und 75 eingeben (die *gezogene Zahl*), die Sie dann streichen und den Nutzer informieren, ob die Zahl vorhanden war. Wiederholen Sie dies solange, bis ein Bingo! erreicht wird und machen Sie dann eine entsprechende Ausgabe. Das Programm könnte wie folgt ablaufen:

```
  B  I  N  G  O
  4 27 32 55 73
 15 25 41 58 75
  8 26    59 70
  7 22 33 54 63
 13 17 43 48 67
Gezogene Zahl: 8
Treffer!
  B  I  N  G  O
  4 27 32 55 73
 15 25 41 58 75
    26    59 70
  7 22 33 54 63
 13 17 43 48 67
Gezogene Zahl: 1
Kein Treffer!
  B  I  N  G  O
  4 27 32 55 73
 15 25 41 58 75
    26    59 70
  7 22 33 54 63
 13 17 43 48 67
 ...
  B  I  N  G  O
  4    32 55 73
 15    41 58 75
    26
  7 22 33 54 63
 13 17 43 48 67
Gezogene Zahl: 26
Treffer!
BINGO! BINGO! BINGO! BINGO! BINGO! BINGO!
  B  I  N  G  O
  4    32 55 73
 15    41 58 75
  7 22 33 54 63
 13 17 43 48 67
```

Bonus-Aufgabe: Bingo!-Karten erzeugen

Implementieren Sie eine Methode `generateBingoCard`, die eine Bingo!-Karte *zufällig* nach den oben genannten Bedingungen erzeugt! *Hinweis*: Verwenden Sie zum erzeugen von **`int`**-Zufallszahlen in einem Bereich die Methode `Random.nextInt`.