TU Berlin, Fakultät IV
Brock/Pfisterer/Koenig
MAR 5-1

Aufgabenblatt 2

Stapel und Warteschlangen

Wichtige Ankündigungen

- Das Vorlesungsmaterial, die Übungsblätter und die Vorlagen für die Hausaufgaben finden Sie unter `https://git.tu-berlin.de/algodat-sose25/material.git`.
- Alle Hausaufgaben sind in Einzelarbeit zu erledigen. Kopieren Sie niemals Code und geben Sie Code in keiner Form weiter.
- Wenn Ihre Abgabe nicht im richtigen Ordner liegt, nicht kompiliert, unerlaubte packages oder imports enthält oder zu spät abgegeben wird, gibt es **0 Punkte** auf diese Abgabe.
- Es gibt drei Aufgabentypen:
  - **Tutorium** → Besprechung im Tutorium
  - **Hausaufgabe** → Eigenarbeit mit Hilfe in Rechnerübungen
  - **Klausurvorbereitung** → Optionale Klausurübung

Abgabe (bis 26.05.2025 23:59 Uhr)

Die folgenden Dateien werden für die Bepunktung berücksichtigt:

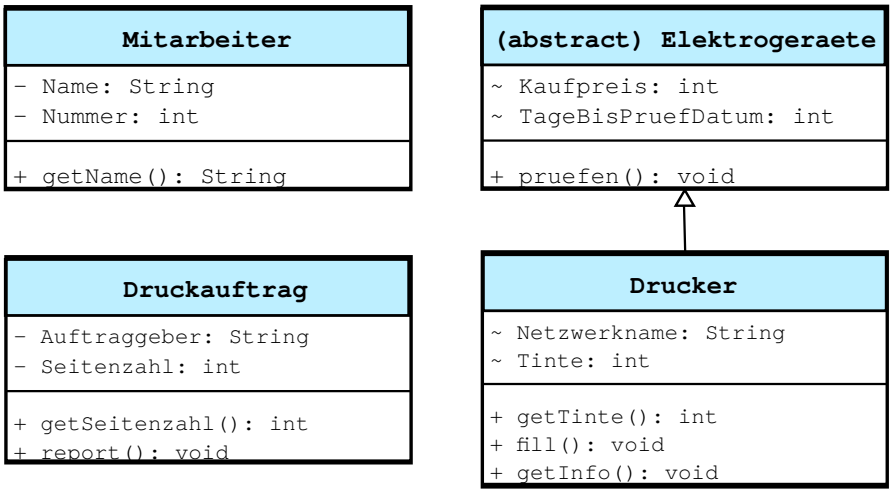
Geforderte Dateien:	
`Blatt02/src/Dec2Bin.java`	Aufgaben 3.1 und 3.2
`Blatt02/src/Bettelmann.java`	Aufgabe 4.2

Als Abgabe wird jeweils nur die letzte Version im main branch in git gewertet.

Aufgabe 1: Das Drucker-Dilemma – Warteschlangen (Tutorium)

1.1 Wiederholung Vererbung

Implementieren Sie die folgendenden Klassen:
(`abstract`) `Elektrogeraete`, `Drucker`, `Mitarbeiter` und `Druckauftrag`, wobei Sie die Attribute und Methoden der Klassen diesen UML Diagrammen entnehmen können:



wobei die Klasse `Drucker` von der Klasse `Elektrogeraete` erbt.
Die Methoden sollen folgendes implementieren:

- `report()`: gibt Auftraggeber und Seitenzahl auf der Kommandozeile aus.
- Der default für `TageBisPruefDatum = 365`
- `pruefen()`: setzt `TageBisPruefDatum` auf 365
- Der default für `Drucker.Tinte = 100`
- `fill()`: setzt `Tinte` auf 100
- `getInfo()`: gibt Druckernamen, Tinte, und TageBisPruefDatum auf der Kommandozeile aus

1.2 Queues (und Stacks) in Java

Definieren Sie Stapel und Warteschlagen. Erstellen Sie eine `main`-Methode in der Klasse `Druckauftrag` und legen Sie zunächst einen Stapel und dann eine Warteschlange von Druckaufträgen in Java an. Beachten Sie, dass Warteschlangen in Java als Interface implementiert sind und Stapel als Klasse und in beiden Generics benutzt werden.

1.3 Drucken

Nutzen Sie die vorgegebenen Klassen `Firma`, `Drucker` und Ihre in Aufgabe 1.1 erstellten Klassen `Elektrogeraete`, `Mitarbeiter` und `Druckauftrag`. Implementieren Sie in der Klasse `Drucker` die Methoden `fill`, `getTinte` und `drucken`.
`drucken()` soll alle Aufträge aus der Queue Druckaufträge bearbeiten. Beachten Sie, dass nur gedruckt werden kann, wenn genügend Tinte vorhanden ist und `TageBisPruefDatum > 0` ist. Ansonsten muss die Tinte aufgefüllt oder das Elektrogerät geprüft werden. Eine Seite braucht beim Drucken eine Einheit Tinte. Der Einfachheit halber können nur Druckaufträge angelegt werden, die zwischen 0 und 100 Seiten groß sind. Diese Funktionalität ist schon über

`Firma.Druckauftragssteuerung()` implementiert. Das bedeutet, dass sie immer nur zu Beginn des Druckens prüfen müssen, ob genug Tinte vorhanden ist und diese bei Bedarf auffüllen müssen.
Um eine Wartezeit beim Drucken zu simulieren, können sie die Methode warten benutzen. Testen können sie Ihre Implementierung, in dem sie die `main` in der Klasse Firma ausführen.

Aufgabe 2: Laufzeit kleiner Programmabschnitte (Tutorium)

Bestimmen Sie die Wachstumsordnung der Laufzeit von den in Listing 1 angegebenen Funktionen `f1` bis `f6`.

Listing 1: Java Snippets zur Laufzeitbestimmung

```java
public static int f1(int N) {
  int x = 0;
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      x++;
  return x;
}
```

(a) Funktion `f1`

```java
public static int f2(int N) {
  int x = 0;
  for (int i = 0; i < N/2; i++)
    for (int j = 0; j < i; j++)
      x++;
  return x;
}
```

(b) Funktion `f2`

```java
public static int f3(int N) {
    if (N == 0)
        return 1;
    else
        return 2 * f3(N -1) + 1;
}
```

(c) Funktion `f3`

```java
public static int f4(int N) {
  int x = 0;
  for (int i = 1; i < N; i *= 2)
    x += f3(i);    // f3!
  return x;
}
```

(e) Funktion `f4`

```java
public static int f5(int N) {
    if (N <= 1)
        return N+1;
    else
        return f5(N-2) / f5(N-1);
}
```

(d) Funktion `f5`

```java
public static int f6(int N) {
  if (N == 0)
    return 1;
  return 2 * f6(N/2);
}
```

(f) Funktion `f6`

Es gibt ein Videotutorium zu dieser Aufgabe. Die darin vorgestellte Lösung zu `f4` ist nicht scharf, für ein Beispiel, wie man für diese Funktion auch das richtige Θ bestimmt, können Sie die schwereren Beispiele im Skript durchgehen. Für die Klausur reichen die im Video vorgestellten Vorgehensweisen.

Aufgabe 3: Dezimaldarstellung in Binärdarstellung umwandeln (Hausaufgabe)

Sei N eine natürliche Zahl, betrachtet in Dezimaldarstellung. Die Binärdarstellung dieser Zahl lässt sich leicht von der hintersten zur vordersten Ziffer iterativ bestimmen:

- Die hinterste Ziffer in der Binärdarstellung ist `N % 2` (der Rest der Division von N durch 2).
- Dann wird N durch 2 geteilt (ganzzahlig und ohne Rest, Integerdivision) und das Verfahren am Ergebnis wiederholt bis $N = 0$ ist.

Um die Ziffern in der richtigen Reihenfolge auszugeben, können Sie auf einem Stapel abgelegt werden. Implementieren Sie in diesem Sinne die folgenden beiden Methoden.

3.1 Methode `convert()` implementieren (15 Punkte)

Implementieren Sie in der Klasse `Dec2Bin` die Methode

```
public void convert(int N)
```

Sie soll die übergebene Zahl `N` in der Objektvariablen `N` speichern und in Binärdarstellung umwandeln. Die Binärdarstellung soll in der Objektvariable `binStack` gespeichert werden, so dass die höchstwertige Ziffer mit dem ersten `pop()` von dem Stapel geholt wird. Bedenken sie dabei, in welchem Zustand sich der Stapel befinden muss, bevor die Binärdarstellung korrekt gespeichert werden kann!

3.2 Methode `toString()` implementieren (15 Punkte)

Implementieren Sie in der Klasse `Dec2Bin` die Methode

```
public String toString()
```

Sie soll die gespeicherte Binärdarstellung als String in der üblichen Weise zurückgeben, also `110010` für die Eingabezahl 50. Durch einen Aufruf von `toString()` soll der Stapel **nicht** gelöscht werden.

Hinweise:

- Es kann vorausgesetzt werden, dass die Eingabe ≥ 0 ist.
- Die Methode `convert()` kann problemlos mit weniger als 10 Zeilen und ohne Rekursion implementiert werden.
- Die Methode `toString()` muss die Objektvariable `binStack` in einen String umwandeln und darf nicht die Objektvariable `N` benutzen. In den Tests wird `binStack` geändert (ohne einen Aufruf von `convert()`) und dann `toString()` getestet.

Aufgabe 4: Implementierung des Kartenspiels „Bettelmann“ (Hausaufgabe)

Hinweis:

- Am besten wird das Kartenspiel im Videotutorium erklärt!

Das Kartenspiel „Bettelmann“ ist ein Glücksspiel für zwei Spielerinnen, das schon seit dem 19. Jahrhundert bekannt ist. Es gibt unterschiedliche Varianten und die Regeln variieren in den unterschiedlichen Beschreibungen. Wir benutzen die folgende: Gespielt wird mit einem Skatblatt. Es zählt nur der Wert der Karte (7 < 8 < 9 < 10 < Bube < Dame < König < As), die Farbe (Kreuz, Pik, Herz, Karo) ist nicht relevant. Der Kartenstapel wird gemischt und die Karten abwechselnd an die Spielerinnen verteilt. Jede Spielerin hat nun ihren eigenen Stapel verdeckt vor sich liegen.

Ablauf einer Runde
In jeder Runde nehmen beide Spielerinnen die oberste Karte jeweils von ihrem eigenen verdeckten Stapel und legen sie offen vor sich ab. Falls der *Wert* der Karten gleich ist, nehmen beide Spielerinnen die nächste Karte vom eigenen verdeckten Stapel, drehen sie um und legen sie auf ihren offenen Stapel. Dies wird solange wiederholt, bis die beiden aufgedeckten Karten unterschiedliche Werte haben (dies ist meist schon bei dem ersten Kartenpaar der Fall). Dann nimmt die Spielerin, die die Karte mit dem höheren Wert hatte den offenen Stapel der Mitspielerin, legt ihn auf ihren eigenen offenen Stapel,

dreht den Pack (die beiden kombinierten offenen Stapel) um und schiebt ihn unter ihren verdeckten Stapel, siehe Abbildung 1. Dann beginnt die nächste Runde.
Sobald eine Spielerin während des Nachziehens oder am Ende einer Runde keine verdeckten Karten mehr hat, ist die andere Spielerin die Siegerin des Spiels. Falls beide Spielerinnen während des Nachziehens keine verdeckten Karten mehr haben (unwahrscheinlicher Fall) endet das Spiel unentschieden.

Abbildung 1: Kartenspiel Bettelmann

Links ist der gemischte Kartenstapel. Die Karten werden von oben nach unten einzeln an die beiden Spielerinnen verteilt (Spielanfang). In der Spielrunde decken die Spielerinnen zunächst ihre oberste Karte auf. Da beide Karten den gleichen Wert haben (As), decken sie auch ihre nächste Karte auf und legen sie auf ihren offenen Stapel. Spielerin 1 hat die Karte mit dem höheren Wert (D > 8) und gewinnt den Stich (Zwischenzustand). Sie legt die offenen Karten von Spielerin 2 auf ihre offenen Karten, dreht alle um und schiebt die Karten unter ihren verdeckten Stapel (nach 1. Runde).

In dieser Programmieraufgabe geht es darum, den Spielablauf Runde für Runde zu simulieren. Betrachten Sie zunächst die Klasse `Card`, die eine Spielkarte des Skatspiels darstellen kann. Die Klasse implementiert die Schnittstelle `Comparable`. Für zwei Objekte `card1` und `card2` vom Typ `Card` gibt

```
card1.compareTo(card2)
```

eine negative Zahl zurück, wenn die Karte `card1` einen niedrigeren Wert als die Karte `card2` besitzt, eine positive Zahl, wenn sie einen höheren Wert besitzt und 0, falls die Karten denselben Wert besitzen, sich also nur in der Farbe (Kreuz, Pik, Herz, Karo) unterscheiden. Eine Karte wird über den Aufruf des Konstruktors

```
public Card(int id)
```

mit einem Eingabeargument `id` zwischen 0 und 31 erzeugt. Die konkrete Zuordnung von `id` zu Karte ist für die Aufgabe nicht relevant, kann aber bei Interesse durch Ausführen der `main()` Methode von `Card` ausgegeben werden.

Der Spielablauf wird in der Klasse `Bettelmann` simuliert. Für die verdeckten Kartenstapel `closedPile1` und `closedPile2` wird die Struktur `Deque` (*double-ended queue*) verwendet (im Gegensatz zum Begriff Karten*stapel*). Diese Datenstruktur kann sowohl als Stapel (z. B. mit den Methoden `addLast()` und `pollLast()`), als auch als Warteschlange (z. B. mit den Methoden `addLast()` und `pollFirst()`) benutzt werden. Der Grund für die Verwendung von `Deque` ist, dass hier beide Funktionalitäten benötigt werden. Bei dem Verteilen der Karten werden die Karten auf den verdeckten Stapeln abgelegt. Die zuletzt verteilten Karten werden als erstes gezogen (*last-in first-out*). Die in einer Runde gewonnenen Karten werden unter den Stapel geschoben werden und neue Karten von oben abgenommen werden, also *first-in first-out*.

Die Beschreibung der einzelnen Methoden können Sie der folgenden API und den Javadoc Kommentaren in der Klasse entnehmen. Beachten Sie, dass der Begriff „Kartenstapel“ hier keine Bezug zu der Datenstruktur mit dem Namen „Stapel“ (`Stack`) hat.

API des Spiels Bettelmann

public class Bettelmann

	`Bettelmann()`	erzeugt zwei leere verdeckte Kartenstapel.
	`Bettelmann(Deque<Card> pile1, Deque<Card> pile2)`	übernimmt die übergebenen Karten als verdeckte Stapel.
`void`	`playRound()`	erzeugt zwei leere offene Kartenstapel und führt eine Spielrunde aus.
`int`	`getWinner()`	gibt Gewinnerin zurück: `-1` spiel noch nicht entschieden, `1` Spielerin 1 hat gewonnen, `2` Spielerin 2 hat gewonnen, `0` unentschieden.
`void`	`distributeCards(Stack<Card> deck)`	verteilt die Karten des Stapels `deck` abwechselnd auf die verdeckten Stapel der Spielerinnen. Die zuletzt verteilten Karten werden in der ersten Runde gezogen.
`void`	`distributeCards()`	mischt alle Karten eines Skatblattes und verteilt sie an die beiden Spielerinnen.
`String`	`toString()`	gibt die verdeckten Kartenstapel als `String` zurück.

4.1 Methode `distributeCards(Stack<Card> deck)` inspizieren

Die Methode verteilt die übergebenen Karten `deck` auf die verdeckten Stapel der Spielerinnen. Die Methode soll zunächst die verdeckten Stapel leeren. Die Karten werden dann einzeln vom Stapel genommen und abwechselnd auf die verdeckten Stapel abgelegt, so dass die zuletzt verteilten Karten zuerst gezogen werden. Die Methode soll sie für beliebige Eingaben funktionieren, also auch, wenn `deck` nur einen Teil eines ganzen Kartenspiels enthält.

4.2 Methode `playRound()` implementieren (70 Punkte)

In dieser Methode sollen Sie **eine** Runde gemäß der oben erklärten Regeln realisieren. Überlegen Sie, welche Datenstruktur für die offenen Kartenstapel, die sich während einer Runde bilden (und am Ende eingesammelt werden) am besten passt; Stapel oder Warteschlange. Sie können auch `Deque` mit den entsprechenden Methoden nutzen, aber eine der Strukturen `Stack` oder `Queue` reicht auch (nur welche?).
Achten Sie in geeigneten Momenten einer Runde (wie z.B. vor dem Ziehen) darauf, ob bereits

(oder noch) Karten auf den jeweiligen unterschiedlichen Stapeln liegen. Auch wenn ein Unentschieden, wie oben erklärt, als eher unwahrscheinlich gilt, sollen Sie in Ihrer Implementierung den Fall natürlich betrachten.

Die Methode muss die Objektvariable `winner` setzen, wenn das Spiel bei einer Runde beendet wird. Dies ist der Fall, wenn während des Nachziehens in einer Runde oder am Ende einer Runde mindestens einer der verdeckten Stapel leer ist.

Hinweise:

- Es gibt Anfangsstellungen, die zu unendlichen Spielsequenzen führen. Bei den Tests werden solche Fälle nicht getestet. Sie brauchen also nicht berücksichtigt zu werden.

Aufgabe 5: Stapel und Warteschlangen umkehren (Klausurvorbereitung)

Stapel und Warteschlagen funktionieren in Bezug auf die Abrufreihenfolge gegensätzlich. Daher kann man die Reihenfolge der Elemente in einem Stapel umkehren, indem man sie aus dem Stapel holt, in eine (leere) Warteschlange einfügt, sie dann aus der Warteschlange holt und wieder auf den (leeren) Stapel legt. Für Warteschlangen geht die Umkehrung entsprechend über einen Stapel.

Implementieren Sie in der Klasse `QueuesAndStacks` die statischen Methoden

```
public static <E> void reverse(Stack<E> stack)
public static <E> void reverse(Queue<E> queue)
```

Sie sollen die Reihenfolge der Elemente eines Stapels bzw. einer Warteschlange umkehren, in dem sie das oben skizzierte Verfahren benutzen. Die Methoden benutzen den generischen Typ `E` für die Elemente der jeweiligen Datenstruktur. Testen Sie Ihre Methoden mit der vorbereiteten `main()` Methode. Sie benutzt die ebenfalls schon implementierten Methoden `stackToString()` und `queueToString()`. Schauen Sie auch, zu welcher Ausgabe die Benutzung der Methoden `Stack.toString()` und `Queue.toString()` führen. Sie werden implizit in den beiden auskommentierten Zeilen aufgerufen.

Was Sie nach diesem Blatt wissen sollten:

- was Objekte und Klassen sind und wie Vererbung funktioniert.
- wie man Konstruktoren schreibt, Objekte erzeugt und Methoden auf diesen Objekten aufruft.
- wie Sie die Lauzeit von Methoden bestimmen.
- was Collections in Java sind.
- wie Stacks und Queues funktionieren, was die Unterschiede sind und wie man sie nutzt.
- was die Schnittstellen Comparable (und Comparator, Iterator und Iterable) sind und wie man sie implementiert.
- wie Sie eine toString()-Methode schreiben und was das ist.

- wie Sie while-Schleifen verwenden.
- wie Sie eine main-Methode zu Testzwecken anpassen und ausführen.
- wie Sie Ihre geschriebenen Methoden selbst ausprobieren und überprüfen.