Im MS Teams Raum "P1_s23s_ITS1", Channel "Allgemein" und dann
Dateien/Dokumente → General → P1_s23s_ITS1 → CODE → LAB → Aufgabenzettel2
finden Sie die Code-Templates zu den einzelnen Aufgaben als ZIP-Dateien.

Aufgabe V2.1 Vorbereitungsaufgabe: char[] versus String Demonstrator

Diese Vorbereitungsaufgabe macht erst Sinn, sobald Arrays besprochen wurden und ist ab dem Moment Pflicht.

An der abgesprochenen "Ablagestelle" finden Sie die Zip-Datei `v2x1.zip`. Entpacken Sie die Zip-Datei, binden Sie den gestellten Code in Eclipse ein und beachten Sie dabei, dass Sie diese Aufgabe in einem eigenen Eclipse-Projekt v2x1 lösen.

Sie werden die in dieser Demo gezeigten Dinge für diesen Aufgabenzettel noch nicht brauchen, aber bis zum nächsten Aufgabenzettel schreitet die Vorlesung voran. Schauen Sie sich zu einem geeigneten Zeitpunkt diese Demo an. Diese Demo soll den Unterschied zwischen einem **`char[]`** und einem **`String`** verdeutlichen. Für Sie interessant ist die Klasse/Datei **`Demonstrator`**.

Aufgabe A2.1 Simplified Black Jack Agent

An der abgesprochenen "Ablagestelle" finden Sie die Zip-Datei `a2x1.zip`. Entpacken Sie die Zip-Datei, binden Sie den gestellten Code in Eclipse ein und beachten Sie dabei, dass Sie diese Aufgabe in einem eigenen Eclipse-Projekt a2x1 lösen.
Nach dem Entpacken finden Sie 2 Klassen vor:

- SimplifiedBlackJackAgent ist ein Code-Template in dem Sie Ihre Lösung einbauen sollen
- TestFrameAndStarter ist eine Möglichkeit Ihre Lösung anzustarten und zu testen.

Weiterhin gibt es das cards-Package, das die benötigten Klassen Card und Deck enthält.

Ergänzen Sie die Methode **`playBlackJack()`** um Code und implementieren Sie einen einfachen Black-Jack-"Agenten". Ziel ist möglichst dicht an 21 Punkte zu kommen. Wenn Sie 21 Punkte überschreiten, haben Sie verloren.
Konkret: Ziehen Sie so lange Karten vom Stapel bis Sie min. 17 Punkte zusammen haben. Geben Sie nach jedem Ziehen einer Karte in einer Zeile die Karte und den aktuellen Punktestand aus. Die Karten haben für diese Aufgabe den folgenden Wert:

- Asse zählen elf Punkte.
- Zweier bis Zehner zählen entsprechend ihres Kartenranges bzw. ihrer "Augen".
- Bildkarten (Buben, Damen, Könige) zählen zehn Punkte.

Sofern 21 Punkte überschritten wurden geben Sie noch LOST aus.

Zusatzteilaufgabe:
Erweitern Sie die Aufgabenstellung nach Belieben in Richtung "echtes Black Jack" (https://de.wikipedia.org/wiki/Black_Jack). Z.B.: Asse zählen nach Belieben ein oder elf Punkte. (So ist es bei Black Jack üblich – Black Jack hat noch viel mehr Sonderregeln). Da nun erst zum Schluss bestimmt wird, was die Asse Wert sind, könnte der aktuelle Punktestand unklar bzw. noch offen sein. Entscheiden Sie, was nun für die Ausgabe nach dem jeweiligen Ziehen einer Karte sinnvoll ist.

Aufgabe A2.2 Primfaktorzerlegung

Vorweg, ja dies war auch schon Aufgabe Z1x8 ☺.
Weiterhin gilt, dass in dieser Aufgabe **keine** Arrays, Strings oder Collections verwendet werden dürfen. Da deren Verwendung keinen Sinn macht. Im Zweifelsfall vorher(!) Rückfrage halten.

An der abgesprochenen "Ablagestelle" finden Sie die Zip-Datei `a2x2.zip`. Entpacken Sie die Zip-Datei, binden Sie den gestellten Code in Eclipse ein und beachten Sie dabei, dass Sie diese Aufgabe in einem eigenen Eclipse-Projekt a2x2 lösen.

Nach dem Entpacken finden Sie 3 Klassen vor:

- PrimeFactorPrinter ist ein Code-Template in dem Sie Ihre Lösung einbauen sollen
- TestFrameAndStarter ist eine Möglichkeit Ihre Lösung anzustarten und zu testen.
- Für ProposalForAlternativeTestFrameAndStarter siehe einleitenden Text.

Berechnen Sie die Primfaktorzerlegung für eine positive Zahl, die in einer Variablen **`zahl`** vom Typ long abgelegt ist. Die Ausgabe soll wie im Beispiel dargestellt aussehen. Also:
<Wert> "=" {<Primfaktor> "*"} <Primfaktor>

Beispiele:

```
                  12 =  2*2*3
                  13 =  13
                  15 =  3*5
                  28 =  2*2*7
                 768 =  2*2*2*2*2*2*2*2*3
          1080664428 =  2*2*3*90055369
         51539607816 =  2*2*2*3*2147483659
         65498029444 =  2*2*37*12689*34877
      37513856612736 =  2*2*2*2*2*2*2*3*3*3*3*3*7*7*7*17*17*23*23*23
     950052134362500 =  2*2*3*3*3*5*5*5*5*5*13*13*23*23*23*37*37
9223372036854775807 =  7*7*73*127*337*92737*649657
```

Achtung: Die Ausgabe schließt mit einem Primfaktor ab und nicht dem Multiplikationszeichen "*"
Bedenke: Was passiert z.B. bei Startwerten wie 2, 1, 0, -1 oder -13 ?

Zusatz-Teil-Aufgabe:

Zerlegen Sie die Zahl 9223372036854775643 in ihre Primfaktoren.
In welches Problem laufen Sie (möglicherweise) dabei? (Dauert es vielleicht etwas lange?)
Für diese (Zusatz-Teil-)Aufgabe dürfen Sie auf *integerSquareRoot()* zurückgreifen.
Da die Berechnung einer Wurzel **aufwendig** ist, sollten Sie dies nur zu gut überlegten "Momenten" tun.
Konkret: Für eine Primfaktorzerlegung sollte nur einmal ("am Anfang") eine Wurzel gezogen werden ☺
Oder alternativ das Quadrat einmalig für den "aktuellen Wurzelkandidaten-Wert" berechnet werden.

Unterstützende Fragen:
Ein Zahl zerfalle in 2 Faktoren f1 und f2 bzw.: **`zahl = f1 * f2`**
Sofern f1 größer der Wurzel ist, was gilt dann für f2? Bzw.: **`f1 > wurzel`** ⇒ …**`f2`**… **?**
Wie lassen sich die zugehörigen "Erkenntnisse" für diese (Zusatz-Teil-)Aufgabe nutzen?
Ist es kritisch, dass **`wurzel`** nur der ganzzahlige Anteil der Wurzel ist?

Syntax-Beispiel:

```
long wurzel = integerSquareRoot(zuZerlegendeZahl);
```

Weitere Rechen-/Test-Beispiele:

```
9223372036854775549 =  9223372036854775549
9223372036854775643 =  9223372036854775643
9223372036854775673 =  175934777*52424950849
9223372036854775771 =  19*485440633518672409
9223372036854775777 =  584911*15768846947407
9223372036854775782 =  2*3*3*3*3*17*23*319279*456065899
9223372036854775783 =  9223372036854775783
9223372036854775797 =  3*3074457345618258599
```

Aufgabe A2.3 Die letzten Drei

Vorweg, ja dies war auch schon Aufgabe Z1x3 ☺.
Weiterhin gilt, dass in dieser Aufgabe **keine** Arrays, Strings oder Collections verwendet werden dürfen.
An der abgesprochenen "Ablagestelle" finden Sie die Zip-Datei `a2x3.zip`. Entpacken Sie die Zip-Datei, binden Sie den gestellten Code in Eclipse ein und beachten Sie dabei, dass Sie diese Aufgabe in einem eigenen Eclipse-Projekt a2x3 lösen.
Nach dem Entpacken finden Sie 2 Klassen vor:

- LastThree ist ein Code-Template in dem Sie Ihre Lösung einbauen sollen
- TestFrameAndStarter ist eine Möglichkeit Ihre Lösung anzustarten und zu testen.

In der Klasse `LastThree` finden Sie eine Methode `processNewValue()` vor. Beim Aufruf dieser Methode wird eine Zahl "entgegengenommen" (konkret vom Typ `int`) und im formalen Parameter `value` abgelegt. Jeder Wert aus dem Wertebereich `int` ist hierbei zulässig.
Ihre Aufgabe ist es, sich die letzten 3 Werte zu merken. Beim Aufruf der Methode `printLastThree()` sollen die letzten 3 Werte ausgegeben werden.

Freiwillige Zusatzaufgabe Z2.1 Die letzten vier Karten

Weiterhin gilt, dass in dieser Aufgabe **keine** Arrays, Strings oder Collections verwendet werden dürfen.
An der abgesprochenen "Ablagestelle" finden Sie die Zip-Datei `z2x1.zip`. Entpacken Sie die Zip-Datei, binden Sie den gestellten Code in Eclipse ein und beachten Sie dabei, dass Sie diese Aufgabe in einem eigenen Eclipse-Projekt z2x1 lösen.
Nach dem Entpacken finden Sie im Package theLastFourCards 2 Klassen vor:

- LastFourCards ist ein Code-Template in dem Sie Ihre Lösung einbauen sollen.
- TestFrameAndStarter ist eine Möglichkeit Ihre Lösung anzustarten und zu testen.

In der Klasse `LastFourCards` finden Sie eine Methode `processNewCard()` vor. Beim Aufruf dieser Methode wird eine Karte "entgegengenommen" (konkret vom Typ `Card` und im formalen Parameter) `card` abgelegt. Ihre Aufgabe ist es, sich die letzten 4 Karten zu merken. Beim Aufruf der Methode `printLastFourCards()` sollen die letzten 4 Karten ausgegeben werden.

Freiwillige Zusatzaufgabe Z2.2 Sieb des Erastosthenes

Vorweg: Diese Zusatzaufgabe macht erst Sinn, sobald Arrays besprochen wurden.
An der abgesprochenen "Ablagestelle" finden Sie die Zip-Datei `z2x2.zip`. Entpacken Sie die Zip-Datei, binden Sie den gestellten Code in Eclipse ein und beachten Sie dabei, dass Sie diese Aufgabe in einem eigenen Eclipse-Projekt z2x2 lösen.
Nach dem Entpacken finden Sie im Package sieveOfErastosthenes 2 Klassen vor:

- PrimeFinder ist ein Code-Template in dem Sie Ihre Lösung einbauen sollen.
- TestFrameAndStarter ist eine Möglichkeit Ihre Lösung anzustarten und zu testen.

Die Aufgabe:
Auch wenn der Algorithmus schon vor dem griechischen Mathematiker Erastosthenes entdeckt wurde, findet dieser sich im Namen "Sieb des Erastosthenes" wieder. Der Algorithmus wird z.B. unter http://www.primzahlen.de/primzahltests/sieb_des_eratosthenes.htm erklärt und ist ein einfaches Verfahren zum Finden von Primzahlen.
Implementieren Sie den Algorithmus und nutzen ihn um alle Zahlen bis zur einer gegebenen Zahl zu bestimmen und in aufsteigender Reihenfolge auszugeben. Konkret ist (der Parameter der Methode printPrimes) sieveEnd die letzte Zahl, die überprüft werden soll, ob sie eine Primzahl ist.

Achtung! Dieser Algorithmus ist **nicht** für das Lösen von A2.3 Primfaktorzerlegung geeignet.

Freiwillige Zusatzaufgabe Z2.3 Lage zweier Rechtecke zueinander bestimmen

Vorweg: Diese Zusatzaufgabe macht erst Sinn, sobald Arrays besprochen wurden.
An der abgesprochenen "Ablagestelle" finden Sie die Zip-Datei z2x3.zip. Entpacken Sie die Zip-Datei und lösen Sie die Aufgabe Z2.3.
Nach dem Entpacken finden Sie 3 Klassen vor:

- ContactAnalyzer ist ein Code-Template in dem Sie Ihre Lösung einbauen sollen.
- ProposalForYourTestFrameAndStarter ist eine Möglichkeit Ihre Lösung anzustarten und zu testen.
- UnitTestFrameAndStarter ist ein einfacher Unit-Test.

Die Aufgabe:
In einem *kartesischen zweidimensionalen* Koordinatensystem ist ein *achsenparalleles* Rechteck durch die Koordinaten von zwei gegenüberliegenden Eckpunkten P(p_x,p_y) und Q(q_x,q_y) bestimmt. *Bezüglich der Lage von P und Q zueinander darf es keine Einschränkungen in Ihrem Programm geben! Jedoch dürfen Sie selbstverständlich basierend auf P und Q andere Punkte oder andere Dinge berechnen, die Sie "zum Arbeiten" verwenden.* Zur Vereinfachung sind alle Kantenlängen >=1 und alle Koordinaten ganzzahlig.
Bestimmen sie für 2 solche achsenparallele Rechtecke, die sich an beliebigen Positionen in einem *kartesischen* Koordinatensystem befinden, die Lage zueinander.
Dabei wird unterschieden zwischen:

aligned — Der Durchschnitt der beiden Rechtecke ist eine Linie.
Alle gemeinsamen Punkte liegen auf einer Linie der Länge > 0.

contained — Der Durchschnitt der beiden Rechtecke ist genau eines der beiden Rechtecke. Ein Rechteck ist also echt im anderen enthalten. D.h. auch, dass die beiden Rechtecke verschieden sind.
Jeder Punkt eines der beiden Rechtecke ist auch ein Punkt des jeweils anderen Rechtecks - aber nicht umgekehrt.

disjoint — Die beiden Rechtecke berühren sich nicht.
Sie haben keine gemeinsamen Punkte.

intersecting — Der Durchschnitt der beiden Rechtecke ist ein neues drittes Rechteck, das von den ersten beiden verschieden ist und eine Fläche > 0 enthält.
Die beiden Rechtecke haben gemeinsame Punkte, aber jedes der beiden Rechtecke hat auch Punkte, die das andere nicht hat.

same — Die Rechtecke sind gleich, sie haben die gleiche Lage und die gleiche Größe.
Der Durchschnitt der beiden Rechtecke enthält alle Punkte der beiden Rechtecke.

touching — Der Durchschnitt der beiden Rechtecke ist ein Punkt.
Die beiden Rechtecke haben genau einen gemeinsamen Punkt.

Beispiel für 2 Rechtecke:

```
int[][] r1 = {
                { 1, 2 },   // Eckpunkt P(px,py) vom 1.Rechteck
                { 7, 5 }    // Eckpunkt Q(qx,qy) vom 1.Rechteck
              };

int[][] r2 = {
                { 4, 9 },   // Eckpunkt P(px,py) vom 2.Rechteck
                { 6, 3 }    // Eckpunkt Q(qx,qy) vom 2.Rechteck
              };
```

Schreiben Sie nun eine Methode **`computeRelation`**, die zwei Rechtecke (jeweils vom Typ **`int[][]`**) als Parameter entgegen nimmt und als Ergebnis einen **`String`** abliefert, der einen Wert entsprecht der obigen Anforderung enthält.

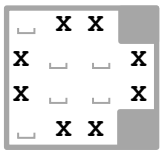
Freiwillige Zusatzaufgabe Z2.4 Muster addieren

Vorweg: Diese Zusatzaufgabe macht erst Sinn, sobald Arrays besprochen wurden.
An der abgesprochenen "Ablagestelle" finden Sie die Zip-Datei z2x4.zip. Entpacken Sie die Zip-Datei und lösen Sie die Aufgabe Z2.4.
Nach dem Entpacken finden Sie 3 Klassen vor:

- ArrayProcessor ist ein Code-Template in dem Sie Ihre Lösung einbauen sollen.
- ProposalForYourTestFrameAndStarter soll Ihnen helfen Ihre Lösung anzustarten und mit eigenen Tests reproduzierbar zu testen.
- UnitTestFrameAndStarter ist ein einfacher Unit-Test, der als Akzeptanz-Test konzipiert ist und Ihnen eine gewisse Sicherheit vermitteln soll, dass Sie die Aufgabe auch wirklich gelöst haben.
- Weiterhin finden Sie auch den Code (die UnitTestFrames) für die Zusatzaufgaben vor

Vorweg: In den folgenden Array-Darstellungen ist die erste Dimension als Y-Achse und die zweite Dimension als X-Achse dargestellt. Das Array "fängt links oben an". Dort sind die "Koordinaten (0,0)".

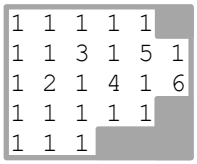
Durchlaufen Sie ein als Parameter gegebenes Array `long[][]` mit dem nachfolgenden Muster



(das Muster muss zum jeweiligen Zeitpunkt in das als Parameter übergebene Array passen)
und addieren Sie jeweils alle Grundelemente im als Parameter übergebenen Array, die jeweils dem obigen Muster (die mit X markierten Felder) genügen. Die aus der Addition resultierende Summe ist das Ergebnis.
Wie Sie das Muster in Ihrem Code umsetzen/implementieren - ob z.B. als Array etwa `boolean[][]` (oder anders) - ist Ihnen freigestellt.
Achtung! Weder das Muster noch das als Parameter übergebene Array muss echt zweidimensional sein. Wir erinnern uns, in Java gibt es keine echt zweidimensionale Arrays. Es gibt nur eindimensionale Arrays über eindimensionale Arrays.
Es wird zugesichert, dass das Ergebnis im Wertebereich von long liegt – dies müssen sie also nicht überprüfen.
Sollte es gemäß Aufgabenstellung keine Werte/Grundelemente zum Aufaddieren geben, so ist das Ergebnis 0.

Beispiel:

Das als Parameter übergebene Array soll wie folgt aussehen:



Das Array hat in der ersten Dimension 5 Einträge und in der zweiten Dimension ist die Anzahl der Einträge unterschiedlich (zunächst 5, dann 6, 6, 5 und zuletzt 3).
Das Muster hat in der ersten Dimension 4 und in der zweiten Dimension jeweils 3, 4, 4, 3 Einträge.

Für obiges Beispiel-Array und das zuvor gegebene Muster lautet das Ergebnis:

52

Da das Muster in diesem Beispiel an vier Stellen in das Array passt: